GIÁO TRÌNH
KIỂM THỬ PHẦN MỀM

Phạm Ngọc Hùng, Trương Anh Hoàng và
Đặng Văn Hưng

Tháng 5 năm 2014

Mục lục

Mục lục **ii**

Danh sách hình vẽ **ix**

Danh sách bảng **xiii**

Thuật ngữ **xv**

Lời nói đầu **xix**

1 Tổng quan về kiểm thử **1**

1.1 Các khái niệm cơ bản về kiểm thử 1

1.2 Ca kiểm thử 7

1.3 Mô tả bài toán kiểm thử qua biểu đồ Venn 9

1.4 Việc xác định các ca kiểm thử 11

1.4.1 Kiểm thử chức năng 12

1.4.2 Kiểm thử cấu trúc 14

1.4.3 Tranh luận về kiểm thử chức năng so với kiểm thử cấu trúc 15

1.5 Phân loại các lỗi và sai 17

1.6 Các mức kiểm thử 18

1.7 Tổng kết 23

1.8 Bài tập 23

2 Một số ví dụ **25**
2.1 Bài toán tam giác 25
2.1.1 Phát biểu bài toán 26
2.1.2 Nhận xét 26
2.1.3 Cài đặt truyền thống 27
2.1.4 Cài đặt có cấu trúc 30
2.2 Hàm NextDate (ngày kế tiếp) 32
2.2.1 Phát biểu bài toán 32
2.2.2 Nhận xét 32
2.2.3 Cài đặt 33
2.3 Hệ thống rút tiền tự động đơn giản 35
2.3.1 Phát biểu bài toán 35
2.3.2 Nhận xét 38
2.4 Bộ điều khiển gạt nước ô tô 39
2.5 Bài tập 39

3 Cơ sở toán rời rạc cho việc kiểm thử **41**
3.1 Lý thuyết tập hợp 42
3.1.1 Phần tử của tập hợp 42
3.1.2 Định nghĩa tập hợp 43
3.1.3 Tập hợp rỗng 44
3.1.4 Biểu đồ Venn 44
3.1.5 Các phép toán về tập hợp 46
3.1.6 Quan hệ giữa các tập hợp 48
3.1.7 Phân hoạch tập hợp 48
3.1.8 Các đồng nhất thức về tập hợp 50
3.2 Hàm 51
3.2.1 Miền xác định và miền giá trị 52
3.2.2 Các loại hàm 52
3.2.3 Hàm hợp 54

3.3 Quan hệ 55
3.3.1 Quan hệ giữa các tập hợp 55
3.3.2 Quan hệ trên một tập hợp 57
3.4 Lôgic mệnh đề 59
3.4.1 Các phép toán lôgic 59
3.4.2 Biểu thức lôgic 60
3.4.3 Tương đương lôgic 61
3.5 Lý thuyết xác suất 62
3.6 Lý thuyết đồ thị 64
3.6.1 Đồ thị 64
3.6.2 Đồ thị có hướng 71
3.6.3 Các loại đồ thị dùng cho kiểm thử 79
3.7 Bài tập 87

4 Khảo sát đặc tả và mã nguồn 89
4.1 Khảo sát đặc tả 90
4.1.1 Tiến hành duyệt đặc tả mức cao 90
4.1.2 Các kỹ thuật kiểm thử đặc tả ở mức thấp . 93
4.2 Khảo sát mã nguồn 96
4.2.1 Khảo sát thiết kế và mã nguồn hay là việc kiểm thử hộp trắng tĩnh 96
4.2.2 Phản biện hình thức 97
4.2.3 Phản biện chéo 99
4.2.4 Thông qua 100
4.2.5 Thanh tra 100
4.2.6 Các chuẩn và hướng dẫn trong lập trình . . 101
4.2.7 Danh sách các hạng mục chung cho việc khảo sát mã nguồn 104
4.3 Tổng kết 107
4.4 Bài tập 107

5 Kiểm thử chức năng 109

5.1 Tổng quan 109
5.1.1 Sự phức tạp của kiểm thử chức năng 112
5.1.2 Phương pháp hệ thống 115
5.1.3 Lựa chọn phương pháp phù hợp 120
5.2 Kiểm thử giá trị biên 122
5.2.1 Giá trị biên 122
5.2.2 Một số dạng kiểm thử giá trị biên 126
5.2.3 Ví dụ minh họa 129
5.2.4 Kinh nghiệm áp dụng 130
5.3 Kiểm thử lớp tương đương 131
5.3.1 Lớp tương đương 131
5.3.2 Phân loại kiểm thử lớp tương đương 133
5.3.3 Ví dụ minh họa 136
5.3.4 Kinh nghiệm áp dụng 139
5.4 Kiểm thử bằng bảng quyết định 141
5.4.1 Bảng quyết định 142
5.4.2 Ví dụ minh họa 144
5.4.3 Kinh nghiệm áp dụng 147
5.5 Kiểm thử tổ hợp 148
5.5.1 Kiểm thử đôi một 149
5.5.2 Ma trận trực giao 149
5.5.3 Kinh nghiệm áp dụng 150
5.6 Tổng kết 151
5.7 Bài tập 152

6 Kiểm thử dòng điều khiển 155
6.1 Kiểm thử hộp trắng 155
6.2 Đồ thị dòng điều khiển 156
6.3 Các độ đo kiểm thử 158
6.4 Kiểm thử dựa trên độ đo 161
6.4.1 Kiểm thử cho độ đo C_1 162
6.4.2 Kiểm thử cho độ đo C_2 164

6.4.3 Kiểm thử cho độ đo C_3 165
6.4.4 Kiểm thử vòng lặp 167
6.5 Tổng kết 171
6.6 Bài tập 173

7 Kiểm thử dòng dữ liệu 181
7.1 Kiểm thử dựa trên gán và sử dụng các biến 182
7.1.1 Ý tưởng 182
7.1.2 Các vấn đề phổ biến về dòng dữ liệu 183
7.1.3 Tổng quan về kiểm thử dòng dữ liệu động . 187
7.1.4 Đồ thị dòng dữ liệu 189
7.1.5 Các khái niệm về dòng dữ liệu 193
7.1.6 Các độ đo cho kiểm thử dòng dữ liệu 197
7.1.7 Sinh các ca kiểm thử 202
7.2 Kiểm thử dựa trên lát cắt 205
7.2.1 Ý tưởng về kiểm thử dựa trên lát cắt 205
7.2.2 Ví dụ áp dụng 209
7.2.3 Một số lưu ý với kiểm thử dựa trên lát cắt . 217
7.3 Tổng kết 219
7.4 Bài tập 220

8 Kiểm thử dựa trên mô hình 225
8.1 Khái niệm về kiểm thử dựa trên mô hình 225
8.2 Các phương pháp đặc tả mô hình 227
8.2.1 Máy hữu hạn trạng thái 227
8.2.2 Ôtômat đơn định hữu hạn trạng thái 229
8.2.3 Biểu đồ trạng thái 229
8.2.4 Máy trạng thái UML 229
8.2.5 Các phương pháp đặc tả khác 231
8.3 Sinh các ca kiểm thử từ mô hình 232
8.4 Sinh đầu ra mong muốn cho các ca kiểm thử 233
8.5 Thực hiện các ca kiểm thử 234

8.6 Ví dụ minh họa 235
8.6.1 Đặc tả hệ thống 235
8.6.2 Sinh các ca kiểm thử 237
8.6.3 Thực hiện các ca kiểm thử 238
8.7 Thảo luận về kiểm thử dựa trên mô hình 240
8.8 Một số công cụ kiểm thử dựa trên mô hình 242
8.8.1 AGEDIS 242
8.8.2 Spec Explorer 243
8.8.3 Conformiq Qtronic 244
8.8.4 JCrasher 244
8.8.5 Selenium 245
8.8.6 SoapUI 246
8.8.7 W3af 246
8.9 Tổng kết 247
8.10 Bài tập 247

9 Kiểm thử tự động và công cụ hỗ trợ 251
9.1 Tổng quan về kiểm thử tự động 251
9.2 Kiến trúc của bộ kiểm thử tự động 253
9.3 Ưu nhược điểm của kiểm thử tự động 257
9.4 Một số công cụ kiểm thử tự động 260
9.4.1 JUnit 260
9.4.2 NUnit 261
9.4.3 QuickTest Professional 262
9.4.4 Apache JMeter 262
9.4.5 Load Runner 263
9.4.6 Cucumber 263
9.4.7 CFT4CUnit 264
9.4.8 JDFT 265
9.5 Tổng kết 267
9.6 Bài tập 268

10 KT tích hợp, hệ thống & chấp nhận 269
10.1 Tổng quan 269
10.2 Kiểm thử tích hợp 271
10.2.1 Các loại giao diện và lỗi giao diện 273
10.2.2 Tích hợp dựa trên cấu trúc mô-đun 277
10.2.3 Tích hợp từ trên xuống 279
10.2.4 Tích hợp từ dưới lên 280
10.2.5 Tích hợp bánh kẹp 282
10.2.6 Tích hợp dựa trên đồ thị gọi hàm 282
10.2.7 Tích hợp đôi một 282
10.2.8 Tích hợp láng giềng 282
10.3 Kiểm thử hệ thống 283
10.3.1 Kiểm thử chức năng hệ thống 284
10.3.2 Kiểm thử chất lượng hệ thống 286
10.4 Kiểm thử chấp nhận 297
10.5 Kiểm thử hồi quy 298
10.5.1 Giới thiệu 298
10.5.2 Kỹ thuật kiểm thử hồi quy 300
10.6 Tổng kết 302
10.7 Bài tập 303

Tài liệu tham khảo 305

Sơ lược về các tác giả 315

Danh sách hình vẽ

1.1 Một vòng đời của việc kiểm thử. 6
1.2 Thông tin về một ca kiểm thử tiêu biểu. 8
1.3 Các hành vi được cài đặt và được đặc tả. 9
1.4 Các hành vi được cài đặt, được đặc tả và được kiểm thử. 10
1.5 Một hộp đen kỹ thuật. 12
1.6 So sánh các phương pháp sinh các ca kiểm thử chức năng. 13
1.7 So sánh các phương pháp sinh ca kiểm thử cấu trúc. . 14
1.8 Nguồn các ca kiểm thử. 16
1.9 Phân loại sai bằng độ nghiêm trọng. 18
1.10 Các mức kiểm thử trong mô hình thác nước. 22

2.1 Sơ đồ khối cho cài đặt chương trình tam giác truyền thống. 28
2.2 Sơ đồ dòng dữ liệu cho cài đặt của chương trình tam giác. 31
2.3 Trạm rút tiền ATM. 36
2.4 Các màn hình của máy ATM đơn giản. 38

3.1 Biểu đồ Venn của tập các tháng có 30 ngày. 45
3.2 Các biểu đồ Venn cho các phép toán cơ sở. 47
3.3 Biểu đồ Venn của một phân hoạch. 49
3.4 Ví dụ về dòng nhân quả và không nhân quả. 54

3.5 Một ví dụ về đồ thị. 66
3.6 Một đồ thị có hướng. 72
3.7 Đồ thị có hướng với chu trình. 77
3.8 Đồ thị cô đọng của đồ thị trong hình 3.7. 79
3.9 Đồ thị của các cấu trúc của lập trình có cấu trúc. . . . 80
3.10 FSM cho một phần của máy rút tiền tự động đơn giản. 83
3.11 Một mạng Petri. 84
3.12 Mạng Petri được đánh dấu. 85
3.13 Trước và sau khi cháy một chuyển. 86
3.14 Các đồ thị cho bài tập 10. 88

4.1 Ví dụ về chuẩn lập trình trong một số ngôn ngữ lập trình.103

5.1 Các bước chính của phương pháp kiểm thử chức năng. 116
5.2 Miền xác định của hàm hai biến. 123
5.3 Các ca kiểm thử giá trị biên cho một hàm hai biến. . . 124
5.4 Các ca kiểm thử mạnh cho hàm hai biến. 127
5.5 Các ca kiểm thử biên tổ hợp của hàm hai biến. 128

6.1 Các thành phần cơ bản của đồ thị chương trình. 157
6.2 Các cấu trúc điều khiển phổ biến của chương trình. . . 157
6.3 Mã nguồn của hàm `foo` và đồ thị dòng điều khiển của nó.158
6.4 Quy trình kiểm thử đơn vị chương trình dựa trên độ đo. 162
6.5 Mã nguồn của hàm `foo` và đồ thị dòng điều khiển của nó.163
6.6 Hàm `foo` và đồ thị dòng điều khiển ứng với độ đo C_3. . 166
6.7 Hàm `average` và đồ thị dòng điều khiển ứng với độ đo C_3.168

7.1 Tuần tự các câu lệnh có vấn đề thuộc loại 1. 184
7.2 Tuần tự các câu lệnh có vấn đề thuộc loại 2. 185
7.3 Sơ đồ chuyển trạng thái của một biến. 187
7.4 Đồ thị dòng dữ liệu của hàm `ReturnAverage`. 193
7.5 Mối quan hệ giữa các độ đo cho kiểm thử dòng dữ liệu. 202
7.6 Mối quan hệ bao gồm chặt giữa các độ đo thực thi được. 204

7.7 Một ví dụ về lát cắt chương trình. 206
7.8 Hàm `ReturnAverage` sau khi phân mảnh và đồ thị của nó. 210
7.9 Mạng tinh thể của hàm `ReturnAverage` trong hình 7.8. 216
7.10 Một ví dụ về đồ thị dòng dữ liệu. 221
7.11 Một ví dụ về đồ thị dòng dữ liệu và việc sử dụng các biến. 222

8.1 Quy trình kiểm thử dựa trên mô hình [KJ02]. 226
8.2 Một ví dụ về máy hữu hạn trạng thái. 228
8.3 Một ví dụ về biểu đồ trạng thái [BBH05]. 230
8.4 Một ví dụ về máy trạng thái UML. 231
8.5 Một ví dụ về đường đi trong máy hữu hạn trạng thái. . 233
8.6 Sinh các đường đi từ máy hữu hạn trạng thái. 233
8.7 DFA cho trang đăng nhập của Hệ thống đăng ký môn học. 235
8.8 Biểu diễn DFA cho trang đăng nhập bằng Excel. . . . 237
8.9 Kiến trúc của Spec Explorer. 243
8.10 FSM cho một phần của máy ATM đơn giản. 249
8.11 Máy hữu hạn trạng thái của máy điện thoại. 250

9.1 Kiến trúc chung của một bộ kiểm thử tự động. 254
9.2 Các công cụ kiểm thử tự động trong phát triển phần mềm. 257
9.3 Giao diện của công cụ CFT4CUnit. 265
9.4 Giao diện cho phép chọn tệp mã nguồn .java cần kiểm thử. 266
9.5 Giao diện hiển thị mã nguồn và đồ thị dòng điều khiển. 266
9.6 Báo cáo kiểm thử được sinh bởi công cụ JDFT. 267

10.1 Cấu trúc phân cấp mô-đun. 278
10.2 Tích hợp từ dưới lên mô-đun E, F và G. 281
10.3 Tích hợp từ dưới lên mô-đun B, C, D với E, F và G. . 281

Danh sách bảng

1.1 Các sai lầm về đầu vào/đầu ra 19
1.2 Các sai lầm về lôgic 19
1.3 Các sai lầm về tính toán 20
1.4 Các sai lầm về giao diện 20
1.5 Các sai lầm về dữ liệu 21

2.1 Tốc độ của cần gạt ứng với vị trí của chốt và núm vặn 39

5.1 Các ca kiểm thử các giá trị biên cho bài toán Tam giác 129
5.2 Một số ca kiểm thử biên tổ hợp cho hàm NextDate . . 130
5.3 Các ca kiểm thử lớp tương đương cho bài toán Tam giác 133
5.4 Các ca kiểm thử lớp tương đương mạnh cho hàm Tam giác 134
5.5 Các ca kiểm thử lớp tương đương cho hàm Tam giác . 136
5.6 Các ca kiểm thử biên tổ hợp hàm NextDate 138
5.7 Các ca kiểm thử biên tổ hợp hàm NextDate 139
5.8 Một phần ca kiểm thử lớp tương đương mạnh cho NextDate 140
5.9 Ví dụ về một bảng quyết định 142
5.10 Bảng quyết định để khắc phục sự cố máy in 143
5.11 Bảng quyết định cho hàm Triangle 144

5.12 Ca kiểm thử bằng bảng quyết định cho hàm Triangle . 145
5.13 Ca kiểm thử bằng bảng quyết định cho hàm NextDate 146
5.14 Các ca kiểm thử đôi một cho hàm g 149
5.15 Mảng trực giao $L_4(2^3)$ 150

6.1 Các ca kiểm thử cho độ đo C_1 của hàm `foo` 159
6.2 Các trường hợp cần kiểm thử của độ đo C_2 với hàm `foo` 160
6.3 Các ca kiểm thử cho độ đo C_2 của hàm `foo` 160
6.4 Các trường hợp cần kiểm thử của độ đo C_3 với hàm `foo` 161
6.5 Các ca kiểm thử cho độ đo C_3 của hàm `foo` 161
6.6 Các ca kiểm thử cho độ đo C_1 của hàm `foo` 164
6.7 Các ca kiểm thử cho độ đo C_2 của hàm `foo` 165
6.8 Các ca kiểm thử cho độ đo C_3 của hàm `foo` 167
6.9 Các ca kiểm thử cho độ đo C_3 của hàm `average` 169
6.10 Các ca kiểm thử cho vòng lặp `while` của hàm `average` . 171

7.1 $def()$ và $c\text{-}use()$ của các đỉnh trong Hình 7.4 195
7.2 Các điều kiện và $p\text{-}use()$ của các cạnh trong Hình 7.4 . 195
7.3 $I\text{-}def$, $A\text{-}def$ và `Const` của các đỉnh trong Hình 7.8 . . 211
7.4 $C\text{-}def$, $L\text{-}def$, $I\text{-}def$, $O\text{-}def$ và $P\text{-}def$ của các đỉnh . . 212

8.1 Bảng chuyển của máy hữu hạn trạng thái trong hình 8.2 228

10.1 Kiểm thử hệ thống, kiểm thử chấp nhận và hồi quy . . 271

Thuật ngữ

Từ viết tắt	Từ đầy đủ	Ý nghĩa
	Acceptance Testing	Kiểm thử chấp nhận
	Alpha Testing	Kiểm thử Alpha
ATM	Automated Teller Machine	Máy rút tiền tự động
ATDD	Acceptance Test Driven Development	Phát triển định hướng kiểm thử chấp nhận
	Automated Testing	Kiểm thử tự động
	Auto-test execution	Thực thi tự động các ca kiểm thử
	Axiomatic set theory	Lý thuyết tập hợp tiên đề
	Beta Testing	Kiểm thử Beta
BDD	Behavior Driven Development	Phát triển định hướng hành vi
	Baseline Test	Kiểm thử cơ sở
	Benchmark Test	Kiểm thử chuẩn
	Black-box Testing	Kiểm thử hộp đen
	Boundary value testing	Kiểm thử giá trị biên
CFG	Control Flow Graph	Đồ thị dòng điều khiển
CFT	Control Flow Testing	Kiểm thử dòng điều khiển

c-use	computation use	Một biến được sử dụng trong một câu lệnh tính toán
DFA	Deterministic Finite state Automaton	Ôtômat đơn định hữu hạn trạng thái
$def(i)$	Definition(i)	Một biến được định nghĩa (gán giá trị) tại câu lệnh i
DAG	Directed Acyclic Graph	Đồ thị có hướng không có chu trình
DFG	Data Flow Graph	Đồ thị dòng dữ liệu
DFT	Data Flow Testing	Kiểm thử dòng dữ liệu
	Decision table testing	Kiểm thử bằng bảng quyết định
	Dynamic Data Flow Testing	Kiểm thử dòng dữ liệu động
	Integration Testing	Kiểm thử tích hợp
	Endurance Test	Kiểm thử độ bền
EO	Expected Output	Đầu ra mong muốn
XP	Extreme Programming	Phương pháp lập trình XP (cực độ)
	Error	Lỗi
	Equivalence partitioning testing	Kiểm thử phân lớp tương đương
	Failure	Thất bại
	Fault	Sai
FSM	Finite State Machine	Máy hữu hạn trạng thái
	Functional Specification	Đặc tả chức năng
	Functional Testing	Kiểm thử chức năng
GUI	Graphical User Interface	Giao diện đồ họa
	Incident	Sự cố
	Load Test	Kiểm thử tải
	Naive set theory	Lý thuyết tập hợp ngây thơ
MBT	Model-based Testing	Kiểm thử dựa trên mô hình

MC	Model Checking	Kiểm chứng mô hình
OMT	Object Modeling Technique	Kỹ thuật mô hình hóa đối tượng
OOT	Object-oriented Testing	Kiểm thử hướng đối tượng
	Pairwise testing	Kiểm thử đôi một
	Pre-condition	Tiền điều kiện
	Program Slicing	Phân mảnh chương trình
p-use	Predicate use	Một biến được sử dụng trong một biểu thức điều kiện
	Post-condition	Hậu điều kiện
	Regression Testing	Kiểm thử hồi quy
RO	Real Output	Đầu ra thực tế
	Slice-based Testing	Kiểm thử dựa trên lát cắt
	Static Data Flow Testing	Kiểm thử dòng dữ liệu tĩnh
	Stress Test	Kiểm thử quá tải
	Spike Test	Kiểm thử đột biến
	System Testing	Kiểm thử hệ thống
SUT	System Under Test	Hệ thống cần kiểm thử
SQA	Software Quality Assurance	Đảm bảo chất lượng phần mềm
	Test Design	Thiết kế kiểm thử
	Theorem Proving	Chứng minh định lý
UML	Unified Modeling Language	Ngôn ngữ mô hình hóa thống nhất
	Use case	Ca sử dụng
	User stories	Các kịch bản người dùng
	Unit Testing	Kiểm thử đơn vị
V&V	Verification and Validation	Kiểm chứng và thẩm định
	White-box Testing	Kiểm thử hộp trắng

Lời nói đầu

Chúng ta đã và đang chứng kiến sự tăng trưởng đáng kinh ngạc của ngành công nghiệp phần mềm trong vài thập kỷ qua. Nếu như trước đây phần mềm máy tính chỉ được sử dụng để tính toán khoa học kỹ thuật và xử lý dữ liệu thì ngày nay nó đã được ứng dụng vào mọi mặt của của đời sống hàng ngày của con người, từ các ứng dụng nhỏ để điều khiển các thiết bị dùng trong gia đình như các thiết bị nghe nhìn, điện thoại, máy giặt, lò vi sóng, nồi cơm điện, đến các ứng dụng lớn hơn như trợ giúp điều khiển các phương tiện và hệ thống giao thông, trả tiền cho các hoá đơn, quản lý và thanh toán về tài chính, v.v. Vì thế con người ngày càng phụ thuộc chặt chẽ vào các sản phẩm phần mềm và do vậy đòi hỏi về chất lượng của các sản phẩm phần mềm ngày càng cao, tức là các phần mềm phải được sản xuất với giá thành hạ, dễ dùng, an toàn và tin cậy được. Kiểm thử có phương pháp là một hoạt động không thể thiếu trong quy trình sản xuất phần mềm để đảm bảo các yếu tố chất lượng nêu trên của các sản phẩm phần mềm.

Theo thống kê thì việc kiểm thử tiêu tốn khoảng 50% thời gian và hơn 50% giá thành của các dự án phát triển phần mềm. Tăng năng suất kiểm thử là một nhu cầu thiết yếu để tăng chất lượng phần mềm. Vì thế nghiên cứu để phát triển các kỹ thuật, công cụ kiểm thử hữu hiệu và đào tạo đội ngũ kiểm thử có kỹ năng và kinh

nghiệm là các đóng góp thiết thực nhất để tăng cường chất lượng của các sản phẩm phần mềm. Từ yêu cầu thực tế này, ngày nay rất nhiều trường đại học trong nước và quốc tế đã đưa môn *"Kiểm thử và đảm bảo chất lượng Phần mềm"* thành một môn học chuyên ngành của ngành/chuyên ngành công nghệ phần mềm ở cả bậc đại học và cao học. Chúng tôi thấy rằng các học viên cao học và sinh viên cần được đào tạo bài bản về cơ sở của kiểm thử phần mềm, bao gồm cả các kiến thức hàn lâm cơ bản lẫn các kỹ thuật thực hành trong ngành công nghiệp phần mềm để có thể đáp ứng công việc của cả nghiên cứu viên lẫn kiểm thử viên. Chúng tôi viết cuốn giáo trình này không ngoài mục đích nhằm đáp ứng yêu cầu thiết yếu đó. Cuốn giáo trình này sẽ cung cấp cho sinh viên, học viên cao học và giảng viên những chất liệu cơ bản bao phủ những nét chính về những phát triển lý thuyết cơ sở của việc kiểm thử phần mềm và các thực hành kiểm thử chung trong ngành công nghiệp phần mềm. Vì các khái niệm về chất lượng phần mềm là quá rộng, chúng tôi chỉ định giới thiệu những nét chung nhất và cái nhìn tổng thể về kiểm thử và đảm bảo chất lượng phần mềm mà thôi. Thực ra thì phần mềm có rất nhiều loại khác nhau, với nhiều miền ứng dụng khác nhau. Ở mỗi loại và mỗi miền ứng dụng riêng biệt lại có các đặc thù riêng và cần được bổ trợ bởi các kỹ thuật kiểm thử riêng cho chúng. Chúng tôi không có tham vọng đi vào các chi tiết như vậy mà chỉ giới thiệu lý thuyết và thực hành kiểm thử chung và cơ bản nhất nhằm trang bị cho sinh viên những kỹ năng cơ bản để có thể hiểu và tự phát triển các kỹ thuật kiểm thử thích hợp cho các hệ thống phức tạp và chuyên dụng hơn trong thực tiễn sau này.

Giáo trình này được viết dựa vào kinh nghiệm giảng dạy môn kiểm thử và đảm bảo chất lượng phần mềm của chúng tôi trong nhiều năm qua tại Bộ môn Công nghệ Phần mềm, Khoa Công nghệ Thông tin, Trường Đại học Công nghệ, Đại học Quốc gia Hà Nội và hàng chục năm kinh nghiệm của chúng tôi trong thực tế nghiên

cứu và phát triển phần mềm. Để viết giáo trình này, chúng tôi đã tham khảo nhiều cuốn sách được dùng phổ biến trên thế giới về kiểm thử và đảm bảo chất lượng phần mềm. Chúng tôi cũng sử dụng thêm các tài liệu nghiên cứu gần đây để cập nhật các phương pháp và kết quả nghiên cứu hiện nay về lĩnh vực này như được nêu trong phần tài liệu tham khảo ở cuối giáo trình này.

Các chủ đề chính được trình bày trong giáo trình này bao gồm:

- Cơ sở toán học cho kiểm thử phần mềm
- Các khái niệm cơ bản về kiểm thử phần mềm
- Các phương pháp phân tích và khảo sát đặc tả và mã nguồn
- Các phương pháp kiểm thử chức năng hay kiểm thử hộp đen
- Các phương pháp kiểm thử hộp trắng hay kiểm thử cấu trúc
- Các phương pháp và quy trình kiểm thử đơn vị, kiểm thử tích hợp, kiểm thử hệ thống, kiểm thử chấp nhận và kiểm thử hồi quy
- Các phương pháp kiểm thử dựa trên mô hình, kiểm thử tự động và các công cụ hỗ trợ

Để hoàn thành cuốn giáo trình này, chúng tôi đã nhận được sự giúp đỡ tận tình, ý kiến đóng góp qúy báu và sự và động viên chân thành từ các đồng nghiệp, các nghiên cứu sinh, học viên cao học và sinh viên Khoa Công nghệ Thông tin của Trường Đại học Công nghệ, Đại học Quốc gia Hà Nội. Nhiều đồng nghiệp và sinh viên đã dành thời gian đọc cẩn thận, "kiểm thử" đến từng chi tiết nhằm giúp chúng tôi nâng cao chất lượng của cuốn giáo trình này, đặc biệt là PGS. TS. Nguyễn Việt Hà, PGS. TS. Trương Ninh Thuận, TS. Võ Đình Hiếu, TS. Trần Thị Minh Châu (Trường Đại học Công nghệ), PGS. TS. Nguyễn Đình Hóa (Viện Công nghệ Thông tin, ĐHQGHN) và PGS. TS. Đặng Văn Đức (Viện Công nghệ Thông tin, Viện hàm lâm Khoa học Việt Nam). Chúng tôi xin chân thành

cảm ơn các đồng nghiệp, các bạn nghiên cứu sinh, học viên cao học và sinh viên vì những đóng góp to lớn đó.

Mặc dù chúng tôi đã rất nỗ lực nhưng vì thời gian và trình độ còn hạn chế, cuốn tài liệu này không tránh khỏi các thiếu sót. Chúng tôi rất mong cuốn giáo trình sẽ được bạn đọc đón nhận, thông cảm và góp ý. Chúng tôi xin trân trọng cám ơn.

Hà Nội, tháng 5 năm 2014
Các tác giả.

Chương 1

Tổng quan về kiểm thử

Kiểm thử nhằm đánh giá chất lượng hoặc tính chấp nhận được của sản phẩm. Kiểm thử cũng nhằm phát hiện lỗi hoặc bất cứ vấn đề gì về sản phẩm. Chúng ta cần kiểm thử vì biết rằng con người luôn có thể mắc sai lầm. Điều này đặc biệt đúng trong lĩnh vực phát triển phần mềm và các hệ thống điều khiển bởi phần mềm. Chương này nhằm phác họa một bức tranh tổng thể về kiểm thử và đảm bảo chất lượng phần mềm. Các chương còn lại sẽ nằm trong khuôn khổ của bức tranh này và ở mức chi tiết hơn.

1.1 Các khái niệm cơ bản về kiểm thử

Kỹ nghệ kiểm thử đã phát triển và tiến hoá hàng mấy chục năm nên các thuật ngữ trong các tài liệu khác nhau thường không thống nhất và thiếu tương thích. Các thuật ngữ được trình bày trong cuốn sách này dựa vào các thuật ngữ chuẩn được phát triển bởi IEEE (Viện Kỹ nghệ điện và điện tử) với việc chọn lọc cẩn thận các thuật ngữ tiếng Việt tương ứng.

Lỗi (Error): Con người luôn có thể phạm lỗi. Khi lập trình viên phạm lỗi trong lập trình, ta gọi các lỗi đó là bug (con bọ). Lỗi có thể phát tán. Chẳng hạn, một lỗi về xác định yêu cầu có thể dẫn đến sai lầm về thiết kế và càng sai khi lập trình theo thiết kế này. Lỗi là nguyên nhân dẫn đến sai.

Sai (Fault): Sai là kết quả của lỗi, hay nói khác đi, lỗi sẽ dẫn đến sai. Cũng có thể nói sai là một biểu diễn của lỗi dưới dạng một biểu thức, chẳng hạn chương trình, văn bản, sơ đồ dòng dữ liệu, biểu đồ lớp, v.v. Sai có thể khó phát hiện. Khi nhà thiết kế mắc lỗi bỏ sót trong quá trình thiết kế, sai về kết quả gây ra do lỗi đó là cái gì đó bị thiếu. Sai về nhiệm vụ xuất hiện khi vào sai thông tin, còn sai về bỏ quên xuất hiện khi không vào đủ thông tin. Loại sai về bỏ quên khó phát hiện và khó sửa hơn loại sai về nhiệm vụ.

Thất bại (Failure): Thất bại xuất hiện khi một lỗi được thực thi. Có hai điều cần lưu ý trong định nghĩa này là: một là thất bại chỉ xuất hiện ở dạng có thể được thực thi mà thông thường là mã nguồn, hai là các thất bại chỉ liên kết với các lỗi về nhiệm vụ. Câu hỏi là các thất bại tương ứng với các lỗi về bỏ thì quên được xử lý thế nào và những lỗi không bao giờ được thực thi, hoặc không được thực thi trong khoảng thời gian dài cần được xử lý ra sao. Virus Michaelangelo là một ví dụ về lỗi loại này. Nó chỉ được tiến hành vào ngày sinh của Michaelangelo, tức ngày 6/3 mà thôi. Câu trả lời nằm ở chỗ: việc khảo sát, tức là phân tích và duyệt mã, thiết kế hoặc đặc tả có thể ngăn chặn nhiều thất bại bằng cách phát hiện ra các lỗi thuộc cả hai loại.

Sự cố (Incident): Một khi thất bại xuất hiện, nó có thể hiển hoặc không hiển, tức là rõ ràng hoặc không rõ ràng đối với người dùng hoặc người kiểm thử. Sự cố nhằm giúp để nhận biết về sự xuất hiện của thất bại. Sự cố là triệu chứng liên kết với một thất bại và giúp cho người dùng hoặc người kiểm thử nhận biết về sự xuất hiện của thất bại này.

Yêu cầu của khách hàng và đặc tả của phần mềm: Phần mềm được viết để thực hiện các nhu cầu của khách hàng. Các nhu cầu của khách hàng được thu thập, phân tích đánh giá và là cơ sở để quyết định chính xác các đặc trưng cần thiết mà sản phẩm phần mềm cần phải có. Dựa trên yêu cầu của khách hàng và các yêu cầu bắt buộc khác, đặc tả được xây dựng để mô tả chính xác các yêu cầu mà sản phẩm phần mềm cần đáp ứng, và có giao diện thế nào. Tài liệu đặc tả là cơ sở để đội ngũ phát triển phần mềm xây dựng sản phẩm phần mềm. Khi nói đến thất bại trên đây là nói đến việc sản phẩm phần mềm không hoạt động đúng như đặc tả. Lỗi một khi được thực thi có thể dẫn đến thất bại. Do đó, lỗi về bỏ quên được coi là tương ứng với các lỗi khi xây dựng đặc tả.

Kiểm chứng và thẩm định: Kiểm chứng (verification) và thẩm định (validation) hay được dùng lẫn lộn, nhưng thực ra chúng có ý nghĩa khác nhau. Kiểm chứng là quá trình để đảm bảo rằng một sản phẩm phần mềm thỏa mãn đặc tả của nó. Còn thẩm định là quá trình để đảm bảo rằng sản phẩm đáp ứng được yêu cầu của người dùng (khách hàng). Trong thực tế, chúng ta cần thực hiện kiểm chứng trước khi thực hiện việc thẩm định sản phẩm. Vì vậy, chúng ta có thuật ngữ V&V (Verification & Validation). Lý do của việc này là chúng ta cần đảm bảo sản phẩm đúng với đặc tả trước. Nếu thực hiện việc thẩm định trước, một khi phát hiện ra lỗi, chúng ta không thể xác định được lỗi này do đặc tả sai hay do lập trình sai so với đặc tả.

Chất lượng và độ tin cậy của phần mềm: Theo từ điển, chất lượng của một sản phẩm được thể hiện bằng các đặc trưng phù hợp với đặc tả của nó. Theo cách hiểu này, chất lượng của một sản phẩm phần mềm là sự đáp ứng các yêu cầu về chức năng (tức là các hàm cần được tính bởi chương trình), sự hoàn thiện và việc tuân thủ nghiêm ngặt các chuẩn đã được đặc tả, cùng các đặc trưng mong chờ từ mọi sản phẩm phần mềm chuyên nghiệp. Chất lượng

phần mềm đặc trưng cho "độ tốt, độ tuyệt hảo" của phần mềm, và gồm có các yếu tố về chất lượng như: tính đúng đắn (hành vi đúng như đặc tả), tính hiệu quả (tiết kiệm thời gian và tiền bạc), độ tin cậy, tính khả kiểm thử (kiểm thử được và dễ dàng), dễ học, dễ sử dụng, dễ bảo trì, v.v. Như vậy, độ tin cậy chỉ là một yếu tố để đánh giá chất lượng của một sản phẩm phần mềm. Trong thực tế, người kiểm thử nói riêng và người phát triển nói chung thường hay nhầm lẫn độ tin cậy với chất lượng. Khi kiểm thử đạt tới mức phần mềm chạy ổn định, có thể tin và dựa vào nó được (theo thuật ngữ công nghệ phần mềm là có thể phụ thuộc vào nó), người kiểm thử thường cho rằng phần mềm đã đạt chất lượng cao. Các yếu tố về mặt chất lượng mà liên quan trực tiếp đến việc phát triển phần mềm được gọi là các tiêu chuẩn chất luợng như tính có cấu trúc, tính đơn thể, tính khả kiểm thử, v.v.

Độ tin cậy của phần mềm là xác suất để phần mềm chạy không có thất bại trong một khoảng thời gian nhất định. Nó được xem là một yếu tố quan trọng của chất lượng phần mềm. Ngoài ra, thời gian trung bình cho việc khắc phục một sự cố cũng là một thông số quan trọng trong việc đánh giá độ tin cậy của sản phẩm phần mềm.

Kiểm thử: Rõ ràng việc kiểm thử liên quan đến các khái niệm trên: lỗi, sai, thất bại và sự cố. Có hai đích của một phép thử: tìm thất bại hoặc chứng tỏ việc tiến hành của phần mềm là đúng đắn.

Vai trò của kiểm thử phần mềm: Kiểm thử phần mềm đóng vai trò quan trọng trong việc đánh giá chất lượng và là hoạt động chủ chốt trong việc đảm bảo chất lượng cao của sản phẩm phần mềm trong quá trình phát triển. Thông qua chu trình "*kiểm thử - tìm lỗi - sửa lỗi*" ta hy vọng chất lượng của sản phẩm phần mềm sẽ được cải tiến. Mặt khác, thông qua việc tiến hành kiểm thử mức hệ thống trước khi cho lưu hành sản phẩm, ta biết được sản phẩm của ta tốt ở mức nào. Vì thế, nhiều tác giả đã mô tả việc kiểm thử

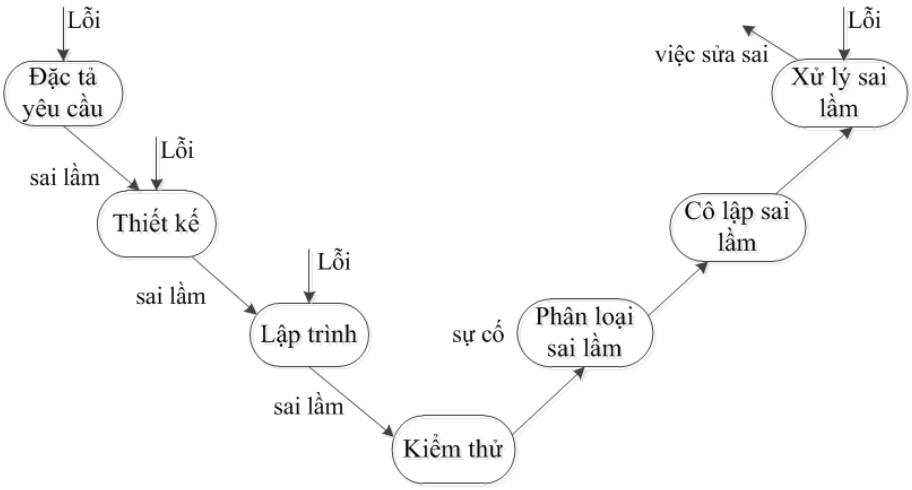
phần mềm là một quy trình kiểm chứng để đánh giá và tăng cường chất lượng của sản phẩm phần mềm. Quy trình này gồm hai công việc chính là phân tích động và phân tích tĩnh.

- **Phân tích tĩnh:** Việc phân tích tĩnh được làm dựa trên việc khảo sát các tài liệu được xây dựng trong quá trình phát triển sản phẩm như tài liệu đặc tả nhu cầu người dùng, mô hình phần mềm, hồ sơ thiết kế và mã nguồn phần mềm. Các phương pháp phân tích tĩnh truyền thống bao gồm việc khảo sát đặc tả và mã nguồn cùng các tài liệu thiết kế. Các kỹ thuật khảo sát này sẽ được giới thiệu trong chương 4.

  Chúng ta cũng có thể dùng các kỹ thuật phân tích hình thức như kiểm chứng mô hình (model checking) và chứng minh định lý (theorem proving) để chứng minh tính đúng đắn của thiết kế và mã nguồn. Các kỹ thuật này tương đối phức tạp và nằm ngoài khuôn khổ của cuốn giáo trình này. Công việc này không liên quan trực tiếp đến việc thực thi chương trình mà chỉ duyệt và lý giải về tất cả các hành vi có thể của chương trình khi được thực thi. Tối ưu hóa các chương trình dịch là các ví dụ về phân tích tĩnh.

- **Phân tích động:** Phân tích động liên quan đến việc thực thi chương trình để phát hiện những thất bại có thể có của chương trình, hoặc quan sát các tính chất nào đó về hành vi và hiệu năng (performance). Vì gần như không thể thực thi chương trình trên tất cả các dữ liệu vào có thể, ta chỉ có thể chọn một tập con các dữ liệu vào để thực thi, gọi là các "ca kiểm thử".

  Chọn như thế nào để được các bộ dữ liệu vào hiệu quả (tức là các bộ dữ liệu có xác suất phát hiện thất bại (nếu có) cao hơn là công việc cần suy nghĩ và là nội dung cơ bản của giáo trình này.



Hình 1.1: Một vòng đời của việc kiểm thử.

Bằng việc phân tích tĩnh và động, người kiểm thử muốn phát hiện nhiều lỗi nhất có thể được để chúng có thể được sửa ở giai đoạn sớm nhất trong quá trình phát triển. Phân tích tĩnh và động là hai kỹ thuật bổ sung cho nhau và cần được làm lặp đi lặp lại nhiều trong quá trình kiểm thử.

Ca kiểm thử: Mỗi ca kiểm thử có một tên (định danh) và được liên kết với một hành vi của chương trình. Ca kiểm thử gồm một tập các dữ liệu đầu vào và một xâu các giá trị đầu ra mong đợi đối với phần mềm.

Hình 1.1 mô tả vòng đời của việc kiểm thử. Lưu ý rằng trong giai đoạn phát triển phần mềm, có ba cơ hội mắc lỗi và tạo ra những sai lan truyền sang phần còn lại của quá trình phát triển là lúc đặc tả, lúc thiết kế và lúc lập trình. Một nhà kiểm thử lỗi lạc đã tóm tắt vòng đời này như sau: Ba giai đoạn đầu là "đưa lỗi vào", giai đoạn kiểm thử là để tìm lỗi, và ba giai đoạn cuối là "khử lỗi đi" [Pos90]. Bước sửa sai lại là cơ hội mới cho việc đưa vào lỗi (và các sai mới). Việc sửa sai có thể làm cho phần mềm từ đúng trở thành sai. Trong trường hợp này, việc sửa sai đó là không đầy đủ.

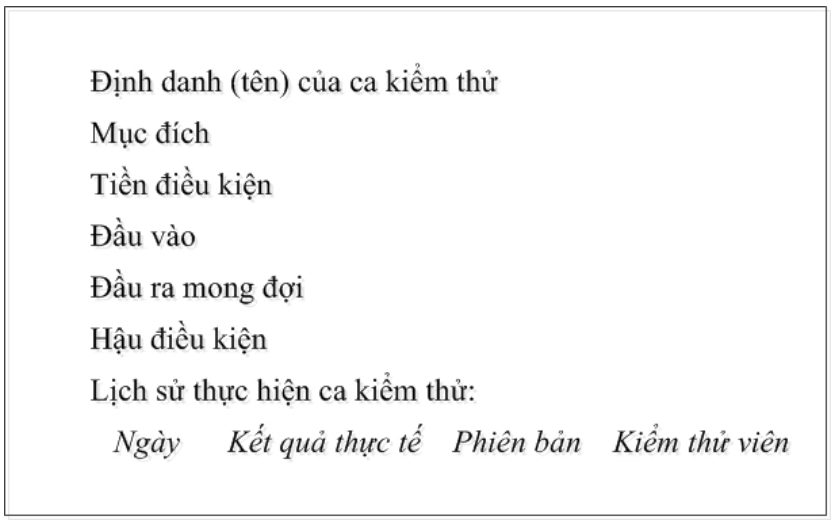
Phương pháp kiểm thử hồi quy được xem là giải pháp hữu hiệu để giải quyết vấn đề này. Phương pháp này sẽ được giới thiệu chi tiết trong chương 10.

Các khái niệm mô tả các thuật ngữ trên đây cho thấy các ca kiểm thử chiếm vị trí trung tâm trong việc kiểm thử dựa trên phân tích động. Quá trình kiểm thử dựa trên phân tích động được chia thành các bước sau: lập kế hoạch kiểm thử, phát triển ca kiểm thử, chạy các ca kiểm thử, và đánh giá kết quả kiểm thử. Tiêu điểm của cuốn giáo trình này là việc xác định tập hữu ích các ca kiểm thử, tức là các ca kiểm thử giúp ta cải tiến tốt hơn chất lượng của sản phẩm phần mềm.

1.2 Ca kiểm thử

Cốt lõi của kiểm thử phần mềm dựa trên phân tích động là việc xác định tập các ca kiểm thử sao cho chúng có khả năng phát hiện nhiều nhất các lỗi (có thể có) của hệ thống cần kiểm thử. Vậy cái gì cần đưa vào các ca kiểm thử? Rõ ràng thông tin đầu tiên là đầu vào. Đầu vào có hai kiểu: tiền điều kiện (pre-condition) - tức là điều kiện cần thỏa mãn trước khi tiến hành ca kiểm thử - và dữ liệu đầu vào thực sự được xác định bởi phương pháp kiểm thử. Thông tin tiếp theo cần đưa vào là đầu ra mong đợi. Cũng có hai loại đầu ra: hậu điều kiện (post-condition) và dữ liệu đầu ra thực sự. Phần đầu ra của ca kiểm thử thường hay bị bỏ quên vì nó là phần khó xác định. Giả sử ta cần kiểm thử phần mềm tìm đường đi tối ưu cho máy bay khi cho trước các ràng buộc về hành lang bay và dữ liệu về thời tiết trong ngày của chuyến bay. Đường đi tối ưu thực sự là gì? Có nhiều câu trả lời cho câu hỏi này. Câu trả lời lý thuyết là giả thiết về sự tồn tại của một cây đũa thần (oracle) biết được tất cả các câu trả lời. Câu trả lời thực tế, được gọi là kiểm thử tham chiếu, là hệ thống được kiểm thử dưới sự giám sát của các chuyên gia về lĩnh vực ứng dụng của phần mềm, người có thể phán xét

xem liệu các dữ liệu đầu ra đối với việc tiến hành trên các dữ liệu đầu vào của ca kiểm thử có chấp nhận được hay không. Hoạt động kiểm thử dẫn đến việc thiết lập các tiền điều kiện cần thiết, việc cung cấp các ca kiểm thử, quan sát dữ liệu đầu ra và so sánh chúng với các đầu ra mong đợi để phát hiện các lỗi/khiếm khuyết của sản phẩm phần mềm.



Hình 1.2: Thông tin về một ca kiểm thử tiêu biểu.

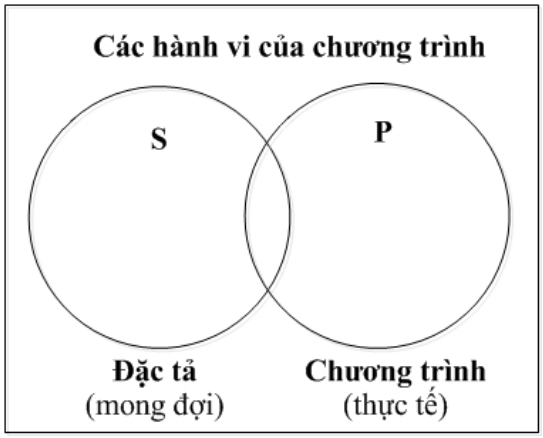
Hình 1.2 mô tả các thông tin cơ bản trong một ca kiểm thử được phát triển đầy đủ, chủ yếu là để trợ giúp việc quản lí. Các ca kiểm thử cần phải định danh bằng tên/chỉ số và lý do tồn tại (chẳng hạn đặc tả nhu cầu tương ứng là một lý do). Chúng ta cũng nên bổ sung thêm lịch sử tiến hành của một ca kiểm thử bao gồm cả việc chúng được chạy bởi ai và chạy khi nào, kết quả của mỗi lần chạy ra sao, thành công hay thất bại và được chạy trên phiên bản nào của phần mềm.

Với các ca kiểm thử cho các hoạt động kiểm thử giao diện người dùng, ngoài thông tin về đầu vào, chúng ta cần bổ sung thêm các thông tin về trình tự nhập các đầu vào cho giao diện. Tóm lại, ta cần nhận thức rằng ca kiểm thử ít nhất cũng quan trọng như mã

nguồn. Các ca kiểm thử cần được phát triển, phân tích đánh giá, sử dụng, quản lý và lưu trữ một cách khoa học.

1.3 Mô tả bài toán kiểm thử qua biểu đồ Venn

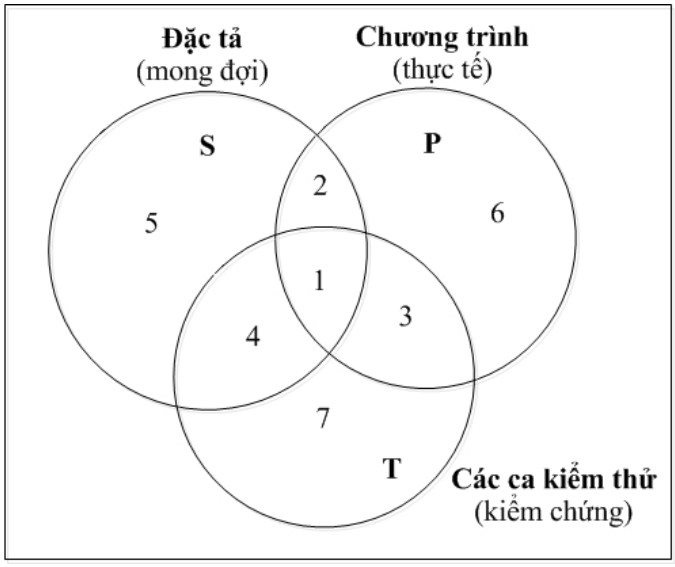
Kiểm thử chủ yếu liên quan tới hành vi của chương trình nơi mà hành vi phản ánh quan điểm về cấu trúc phổ biến đối với các nhà phát triển hệ thống hoặc phần mềm. Sự khác biệt là quan điểm cấu trúc tập trung vào "là cái gì", còn quan điểm hành vi lại tập trung vào "làm gì". Một trong những nguyên nhân gây khó cho người kiểm thử là các tài liệu cơ sở thường được viết bởi và viết cho người phát triển. Kết quả là các tài liệu này thường thiên về thông tin cấu trúc và coi nhẹ thông tin về hành vi của chương trình cần kiểm thử. Trong mục này, chúng ta sẽ phát triển một biểu đồ Venn đơn giản nhằm làm sáng tỏ một số vấn đề về kiểm thử. Chi tiết về biểu đồ Venn sẽ được trình bày trong chương 3.



Hình 1.3: Các hành vi được cài đặt và được đặc tả.

Xét một vũ trụ của hành vi chương trình cần kiểm thử, lưu ý rằng chúng ta đang quan tâm đến bản chất của việc kiểm thử. Cho trước một chương trình cùng đặc tả của nó. Gọi S là tập các hành vi được đặc tả và P là tập các hành vi của chương trình. Hình 1.3

mô tả mối quan hệ giữa vũ trụ các hành vi được lập trình và hành vi được đặc tả. Trong tất cả các hành vi có thể của chương trình, những hành vi được đặc tả nằm trong vòng tròn với nhãn S, còn những hành vi được lập trình là ở trong vòng tròn với nhãn P. Từ biểu đồ này, ta thấy rõ các bài toán mà người kiểm thử cần đối mặt là gì. Nếu có hành vi được đặc tả nhưng không được lập trình thì theo thuật ngữ trước đây, đấy là những sai lầm về bỏ quên. Tương tự, nếu có những hành vi được lập trình nhưng không được đặc tả, thì điều đó tương ứng với những sai lầm về nhiệm vụ, và chúng tương ứng với những lỗi xuất hiện sau khi đặc tả đã hoàn thành. Tương giao giữa S và P là phần đúng đắn, gồm có các hành vi vừa được đặc tả vừa được cài đặt. Chú ý rằng tính đúng đắn chỉ có nghĩa đối với đặc tả và cài đặt và là khái niệm mang tính tương đối.



Hình 1.4: Các hành vi được cài đặt, được đặc tả và được kiểm thử.

Vòng tròn mới (vòng tròn T) trong hình 1.4 là cho các ca kiểm thử. Lưu ý rằng tập các hành vi của chương trình nằm trọn trong vũ trụ chuyên đề của ta. Ở đây một ca kiểm thử cũng được coi là

xác định một hành vi. Xét mối quan hệ giữa S, P và T. Có thể có các hành vi được đặc tả mà không được kiểm thử (các miền 2 và 5), các hành vi được đặc tả và được kiểm thử (các miền 1 và 4), và các ca kiểm thử tương ứng với các hành vi không được đặc tả (các miền 3 và 7).

Tương tự, có thể có các hành vi được lập trình mà không được kiểm thử (các miền 2 và 6), các hành vi được lập trình và được kiểm thử (các miền 1 và 3), và các ca kiểm thử tương ứng với các hành vi không được lập trình (các miền 4 và 7). Việc xem xét tất cả các miền này là hết sức quan trọng. Nếu có các hành vi được đặc tả mà không có các ca kiểm thử tương ứng, việc kiểm thử là chưa đầy đủ. Nếu có các ca kiểm thử tương ứng với các hành vi không được đặc tả, có thể có hai khả năng: hoặc đặc tả còn thiếu hoặc ca kiểm thử không đảm bảo. Theo kinh nghiệm, một người kiểm thử giỏi sẽ thường phát triển các ca kiểm thử thuộc loại đầu, và đấy chính là lý do người kiểm thử cần tham gia vào giai đoạn khảo duyệt đặc tả và thiết kế (xem chương 4).

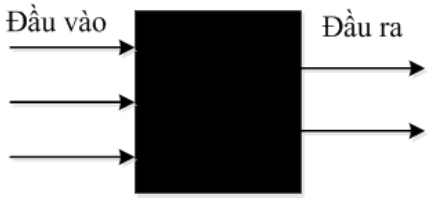
Ta có thể thấy việc kiểm thử như là công việc của một nghệ nhân: người kiểm thử có thể làm gì để làm cho miền tương giao của các tập (miền 1) là lớn nhất có thể? Làm thế nào để xác định các ca kiểm thử trong tập T? Câu trả lời là các ca kiểm thử cần được xác định bởi một phương pháp kiểm thử. Chính khuôn khổ này cho phép ta so sánh tính hiệu quả của các phương pháp kiểm thử khác nhau như sẽ được giới thiệu trong các chương 5, 6 và 7.

1.4 Việc xác định các ca kiểm thử

Có hai cách tiếp cận cơ bản để xác định các ca kiểm thử là kiểm thử chức năng hay kiểm thử hộp đen (black-box testing) và kiểm thử cấu trúc hay kiểm thử hộp trắng (white-box testing). Mỗi cách tiếp cận có các phương pháp xác định các ca kiểm thử khác nhau và được gọi chung là các phương pháp kiểm thử.

1.4.1 Kiểm thử chức năng

Kiểm thử chức năng (kiểm thử hộp đen) dựa trên quan niệm rằng bất kỳ chương trình nào cũng được coi là một hàm ánh xạ các giá trị từ miền dữ liệu đầu vào tới miền dữ liệu đầu ra của nó. Khái niệm này được dùng chung trong kỹ thuật khi các hệ thống đều được coi là các hộp đen. Chính điều này dẫn đến thuật ngữ kiểm thử hộp đen, trong đó nội dung của hộp đen (việc cài đặt) không được biết hoặc không cần quan tâm, và chức năng của hộp đen được hiểu theo các dữ liệu đầu vào và dữ liệu đầu ra của nó. Trong thực tế, chúng ta thường thao tác hiệu quả với những kiến thức về hộp đen. Chính điều này là trung tâm của khái niệm định hướng đối tượng. Chẳng hạn, hầu hết mọi người lái xe thành thạo với kiến thức hộp đen.

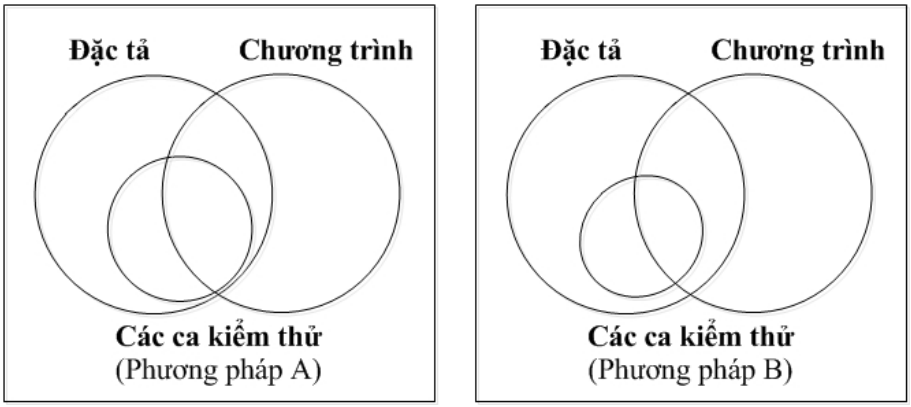


Hình 1.5: Một hộp đen kỹ thuật.

Với cách tiếp cận của kiểm thử chức năng, để xác định các ca kiểm thử, thông tin duy nhất được dùng là đặc tả của phần mềm cần kiểm thử. Có hai lợi điểm chính của các ca kiểm thử được sinh ra bởi cách tiếp cận kiểm thử chức năng: chúng độc lập với việc phần mềm được cài đặt thế nào, và vì thế khi cài đặt thay đổi thì các ca kiểm thử vẫn dùng được, đồng thời các ca kiểm thử được phát triển song song và độc lập với việc cài đặt hệ thống. Do đó, cách tiếp cận này rút gọn được thời gian phát triển của dự án. Điểm yếu của cách tiếp cận này là các ca kiểm thử thường có thể có tính dư thừa đáng kể trong các ca kiểm thử và vấn đề hố phân cách.

Hình 1.6 mô tả các ca kiểm thử được xác định bởi các phương pháp kiểm thử chức năng khác nhau. Phương pháp A xác định một tập các ca kiểm thử lớn hơn so với phương pháp B. Lưu ý rằng đối với cả hai phương pháp này, tập các ca kiểm thử đều chứa trọn trong tập các hành vi được đặc tả.

Do các phương pháp kiểm thử chức năng đều dựa trên các hành vi đặc tả, các phương pháp này khó có thể xác định được các hành vi không được đặc tả. Trong chương 5 ta sẽ so sánh các ca kiểm thử sinh bởi các phương pháp kiểm thử chức năng khác nhau cho các ví dụ được mô tả trong chương 2.



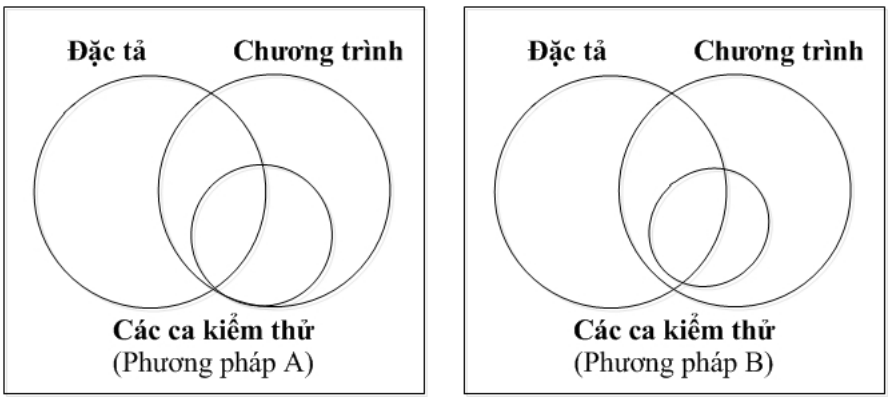
Hình 1.6: So sánh các phương pháp sinh các ca kiểm thử chức năng.

Trong chương 5, chúng ta sẽ khảo sát các cách tiếp cận chủ yếu cho các phương pháp kiểm thử chức năng bao gồm phân tích giá trị biên, kiểm thử tính bền vững, phân tích trường hợp xấu nhất, kiểm thử giá trị đặc biệt, kiểm thử phân lớp tương đương của miền dữ liệu đầu vào, lớp tương đương của miền dữ liệu đầu ra, kiểm thử dựa trên bảng quyết định. Điều xuyên suốt trong các kỹ thuật này là tất cả đều dựa trên thông tin xác định về các thành phần đang được kiểm thử. Cơ sở toán học trình bày trong chương 3 chủ yếu được áp dụng cho cách tiếp cận kiểm thử chức năng.

1.4.2 Kiểm thử cấu trúc

Kiểm thử cấu trúc (kiểm thử hộp trắng) là cách tiếp cận khác để xác định các ca kiểm thử. Biểu tượng hộp trong suốt (hộp trắng) là thích hợp cho cách tiếp cận này vì sự khác nhau cốt lõi của cách tiếp cận này so với kiểm thử hộp đen là việc cài đặt của hộp đen (mã nguồn) được cung cấp và được dùng làm cơ sở để xác định các ca kiểm thử. Việc hiểu biết được bên trong của hộp đen cho phép người kiểm thử dựa trên việc cài đặt để xác định các ca kiểm thử.

Kiểm thử cấu trúc đã trở thành chủ đề của một lý thuyết tương đối mạnh. Để hiểu rõ kiểm thử cấu trúc, các khái niệm về lý thuyết đồ thị tuyến tính được trình bày trong chương 3 là cần thiết. Với những khái niệm này, người kiểm thử có thể mô tả chính xác các yêu cầu kiểm thử và hệ thống cần kiểm thử. Do có cơ sở lý thuyết mạnh, kiểm thử cấu trúc cho phép định nghĩa chính xác và sử dụng các độ đo về độ bao phủ. Các độ đo về độ phủ cho phép khẳng định tường minh phần mềm đã được kiểm thử tới mức nào và do đó giúp cho việc quản lý quá trình kiểm thử tốt hơn.



Hình 1.7: So sánh các phương pháp sinh ca kiểm thử cấu trúc.

Hình 1.7 phản ánh các ca kiểm thử được xác định bởi hai phương pháp kiểm thử cấu trúc khác nhau. Giống như trước đây, phương

pháp A xác định tập các ca kiểm thử lớn hơn so với phương pháp B. Có chắc là tập các ca kiểm thử lớn hơn là tốt hơn không? Đây là một câu hỏi thú vị và kiểm thử cấu trúc cung cấp các giải pháp để tìm câu trả lời cho vấn đề này.

Lưu ý rằng cả hai phương pháp A và B đều cho các tập các ca kiểm thử nằm trọn trong tập các hành vi được lập trình. Do các ca kiểm thử của các phương pháp này được sinh ra dựa trên chương trình nên rất khó để xác định các lỗi liên quan đến các hành vi đã được đặc tả nhưng không được lập trình. Tuy nhiên, dễ thấy rằng tập các ca kiểm thử cấu trúc là tương đối nhỏ so với tập tất cả các hành vi được lập trình.

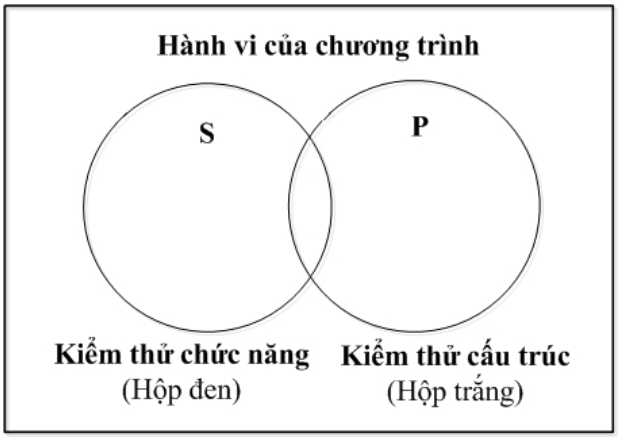
Chúng ta sẽ tìm hiểu các so sánh đánh giá về các ca kiểm thử được sinh bởi các phương pháp kiểm thử cấu trúc khác nhau ở mục 1.4.3. Một số phương pháp kiểm thử cấu trúc (kiểm thử dòng điều khiển, kiểm thử dòng dữ liệu và kiểm thử dựa trên lát cắt) sẽ được giới thiệu chi tiết trong các chương 6 và 7.

1.4.3 Tranh luận về kiểm thử chức năng so với kiểm thử cấu trúc

Cho trước hai cách tiếp cận khác nhau để xác định các ca kiểm thử, câu hỏi tự nhiên được đặt ra là phương pháp nào tốt hơn? Cho đến nay chúng ta vẫn chưa có câu trả lời thỏa đáng cho câu hỏi này.

Nói về kiểm thử cấu trúc, Robert Poston viết: công cụ này lãng phí thời gian của người kiểm thử vì từ những năm bảy mươi (của thế kỷ trước) nó chẳng trợ giúp tốt việc thực hành kiểm thử phần mềm và đừng có đưa nó vào bộ công cụ của người kiểm thử [Pos91].

Nhằm bảo vệ cho việc kiểm thử cấu trúc, Edward Miller [Mil91] viết: Độ bao phủ nhánh (một độ đo độ bao phủ của kiểm thử), nếu đạt được 85% hoặc cao hơn, có thể xác định số lỗi gấp đôi so với số lỗi phát hiện bởi kiểm thử trực quan (kiểm thử chức năng).



Hình 1.8: Nguồn các ca kiểm thử.

Biểu đồ Venn được mô tả trong hình 1.8 có thể giúp ta trả lời câu hỏi mà cuộc tranh luận này đã đề cập. Chúng ta cần khẳng định lại rằng mục đích của cả hai cách tiếp cận trên là để xác định các ca kiểm thử. Kiểm thử chức năng chỉ dùng đặc tả để xác định ca kiểm thử, trong khi kiểm thử cấu trúc dùng mã nguồn của chương trình (cài đặt) để làm cơ sở xác định các ca kiểm thử. Những bàn luận trước đây cho thấy chẳng có cách tiếp cận nào là đủ tốt.

Xét các hành vi chương trình: nếu tất cả các hành vi được đặc tả vẫn chưa được cài đặt, kiểm thử cấu trúc sẽ không thể nhận biết được điều đó. Ngược lại, nếu các hành vi được cài đặt chưa được đặc tả, điều đó chẳng khi nào có thể được phơi bày nhờ kiểm thử chức năng. Một con vi rút là một ví dụ tốt về các hành vi không được đặc tả. Câu trả lời sơ bộ cho câu hỏi trên là cả hai cách tiếp cận đều là rất cần thiết; còn câu trả lời cẩn thận hơn là cách kết hợp khôn khéo giữa hai cách tiếp cận này sẽ cung cấp niềm tin cho kiểm thử chức năng và độ đo của kiểm thử cấu trúc. Ta đã khẳng định ở trên rằng kiểm thử chức năng có khiếm khuyết về tính dư thừa và hố phân cách. Nếu kiểm thử chức năng được tiến hành kết hợp với các số đo về độ phủ của kiểm thử cấu trúc thì khiếm khuyết trên có thể được phát hiện và giải quyết.

Quan điểm biểu đồ Venn cho việc kiểm thử đặt ra câu hỏi về quan hệ giữa tập các ca kiểm thử (T) với các tập S và P của các hành vi cài đặt và đặc tả như thế nào? Rõ ràng, các ca kiểm thử trong T được xác định bởi phương pháp xác định ca kiểm thử được dùng. Một câu hỏi rất hay cần đặt ra là thế thì phương pháp này thích hợp và hiệu quả ra sao. Ta có thể đóng lại vòng luẩn quẩn này bằng những lời bàn trước đây. Với đường đi từ lỗi đến sai, thất bại và sự cố, nếu biết loại lỗi nào ta hay phạm, và loại sai nào hay có trong phần mềm được kiểm thử, ta có thể dùng thông tin này để lựa chọn phương pháp thích hợp để xác định các ca kiểm thử. Chính điểm này làm cho việc kiểm thử thành một nghệ thuật.

1.5 Phân loại các lỗi và sai

Các định nghĩa về lỗi và sai được trình bày trong mục 1.1 xoay quanh sự phân biệt giữa quy trình và sản phẩm. Trong khi quy trình cho chúng ta biết cần làm điều gì đó như thế nào thì sản phẩm là kết quả cuối cùng của quy trình. Kiểm thử phần mềm và đảm bảo chất lượng phần mềm (Software Quality Assurance - SQA) gặp nhau ở điểm là SQA cố gắng cải tiến chất lượng sản phẩm bằng việc cải tiến quy trình. Theo nghĩa này thì kiểm thử là các hoạt động định hướng sản phẩm. SQA quan tâm nhiều hơn đến việc giảm thiểu lỗi trong quá trình phát triển, còn kiểm thử quan tâm chủ yếu đến phát hiện sai trong sản phẩm. Cả hai nguyên lý này đều sử dụng định nghĩa về các loại sai. Các sai được phân loại theo nhiều cách khác nhau: giai đoạn phát triển khi cái sai tương ứng xuất hiện, các hậu quả của các thất bại tương ứng, độ khó cho việc giải quyết, độ rủi ro của việc không giải quyết được, v.v. Một cách phân loại được ưa thích là dựa trên việc xuất hiện bất thường: chỉ một lần, thỉnh thoảng, xuất hiện lại hoặc lặp đi lặp lại nhiều lần. Hình 1.9 minh họa một cách phân loại sai [Bor84] dựa trên độ nghiêm trọng của hậu quả mà các lỗi gây ra.

1	Nhẹ	Lỗi chính tả
2	Vừa	Hiểu lầm hoặc thừa thông tin
3	Khó chịu	Tên bị thiếu, cụt chữ hoặc hóa đơn có giá trị 0.0 đồng
4	Bực mình	Vài giao dịch không được xử lý
5	Nghiêm trọng	Mất giao dịch
6	Rất nghiêm trọng	Xử lý giao dịch sai
7	Cực kỳ nghiêm trọng	Lỗi rất nghiêm trọng xảy ra thường xuyên
8	Quá quắt	Hủy hoại cơ sở dữ liệu
9	Thảm họa	Hệ thống bị tắt
10	Dịch họa	Thảm họa lây lan

Hình 1.9: Phân loại sai bằng độ nghiêm trọng.

Để xử lý các loại sai, chúng ta có thể tham khảo [IEE93] về việc phân loại các chuẩn cho các bất thường của phần mềm. Chuẩn IEEE định nghĩa quy trình giải quyết bất thường một cách chi tiết gồm bốn giai đoạn: nhận biết, khảo sát, hành động và bố trí lại. Một số bất thường phổ biến được mô tả từ Bảng 1.1 đến Bảng 1.5. Hầu hết các bất thường này đều đã được đề cập trong chuẩn IEEE. Ngoài ra, chúng tôi cũng bổ sung thêm một số bất thường không được đề cập.

1.6 Các mức kiểm thử

Một trong các khái niệm then chốt của việc kiểm thử là các mức của việc kiểm thử. Các mức của việc kiểm thử phản ánh mức độ trừu tượng được mô tả trong mô hình thác nước của vòng đời phát triển phần mềm. Dù có một số nhược điểm, mô hình này vẫn rất hữu ích cho việc kiểm thử, là phương tiện để xác định các mức kiểm thử khác nhau và làm sáng tỏ mục đích của mỗi mức. Một dạng của mô hình thác nước được trình bày trong hình 1.6. Dạng này

Bảng 1.1: Các sai lầm về đầu vào/đầu ra

Loại	Các trường hợp
dữ liệu đầu vào	dữ liệu đầu vào đúng nhưng không được chấp nhận
	dữ liệu đầu vào sai nhưng được chấp nhận
	mô tả sai hoặc thiếu
	tham số sai hoặc thiếu
dữ liệu đầu ra	khuôn dạng sai
	kết quả sai
	kết quả tính toán đúng nhưng không đúng thời điểm (quá sớm hoặc quá muộn)
	kết quả không đầy đủ hoặc thiếu
	kết quả giả tạo
	văn phạm/chính tả
	các trường hợp khác

Bảng 1.2: Các sai lầm về lôgic

thiếu trường hợp
lặp thừa trường hợp
điều kiện cực đoan bị bỏ qua
thể hiện sai
thiếu điều kiện
điều kiện ngoại lai
kiểm thử sai biến
việc lặp của chu trình không đúng
phép toán sai (chẳng hạn dùng $<$ cho $\leq$)

nhấn mạnh sự tương ứng của việc kiểm thử với các mức phân tích thiết kế. Lưu ý rằng theo các thuật ngữ của việc kiểm thử chức năng, ba mức của định nghĩa sản phẩm phần mềm (đặc tả, thiết kế sơ bộ và thiết kế chi tiết) tương ứng trực tiếp với bốn mức của việc kiểm thử là kiểm thử đơn vị (unit testing), kiểm thử tích hợp (integration testing), kiểm thử hệ thống (system testing) và kiểm

Bảng 1.3: Các sai lầm về tính toán

thuật toán sai
thiếu tính toán
toán hạng sai
sai về dấu ngoặc
chưa đủ độ chính xác (làm tròn hoặc cắt đuôi)
hàm đi kèm sai

Bảng 1.4: Các sai lầm về giao diện

xử lý gián đoạn sai (trong các hệ thống nhúng)
thời gian vào ra (time out)
gọi sai thủ tục
gọi đến một thủ tục không tồn tại
tham số không khớp (mismatch) (chẳng hạn về kiểu và số)
kiểu không tương thích
bao hàm thừa

thử chấp nhận (acceptance testing). Các mức của kiểm thử cũng làm nảy sinh vấn đề về thứ tự kiểm thử: dưới lên, trên xuống hoặc các khả năng khác. Các mức kiểm thử có thể được mô tả sơ bộ như sau:

- Kiểm thử đơn vị: Kiểm thử đơn vị là việc kiểm thử các đơn vị chương trình một cách độc lập. Thế nào là một đơn vị chương trình? Câu trả lời phụ thuộc vào ngữ cảnh công việc. Một đơn vị chương trình là một đoạn mã nguồn như hàm hoặc phương thức của một lớp, có thể được gọi từ ngoài, và cũng có thể gọi đến các đơn vị chương trình khác. Đơn vị cũng còn được coi là một đơn thể để kết hợp sao cho nó có thể thực thi một cách độc lập. Đơn vị chương trình cần được kiểm thử riêng biệt để phát hiện lỗi trong nội tại và khắc phục chúng trước khi được tích hợp với các đơn vị khác. Kiểm thử đơn vị thường

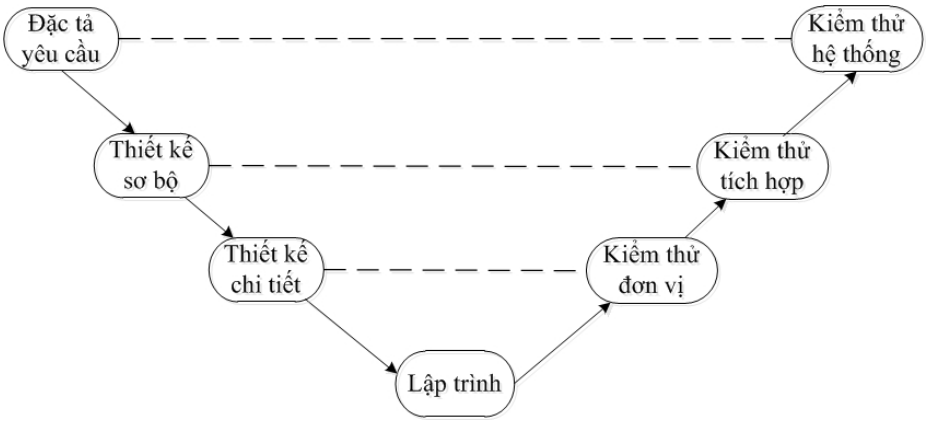
Bảng 1.5: Các sai lầm về dữ liệu

khởi tạo sai
lưu trữ và truy cập sai
giá trị chỉ số và cờ báo sai
gói và mở sai
sử dụng sai biến
tham chiếu sai dữ liệu
đơn vị hoặc thang chia sai
chiều của dữ liệu sai
chỉ số sai
sai về kiểu
sai về phạm vi
dữ liệu cảm biến vượt ra ngoài miền cho phép
lỗi thừa, thiếu một so với biên
dữ liệu không tương thích

được thực hiện bởi chính tác giả của chương trình (lập trình viên), và có thể tiến hành theo hai giai đoạn: kiểm thử đơn vị tĩnh, sử dụng các kỹ thuật ở chương 4, và kiểm thử đơn vị động sử các kỹ thuật ở các chương 5, 6 và 7.

- Kiểm thử tích hợp: Mức kế tiếp với kiểm thử đơn vị là kiểm thử tích hợp. Sau khi các đơn vị chương trình để cấu thành hệ thống đã được kiểm thử, chúng cần được kết nối với nhau để tạo thành hệ thống đầy đủ và có thể làm việc. Công việc này không hề đơn giản và có thể có những lỗi về giao diện giữa các đơn vị, và cần phải kiểm thử để phát hiện những lỗi này.

  Công đoạn này gồm hai giai đoạn: giai đoạn kiểm thử tích hợp và giai đoạn kiểm thử hệ thống. Kiểm thử tích hợp nhằm đảm bảo hệ thống làm việc ổn định trong môi trường thí nghiệm để sẵn sàng cho việc đưa vào môi trường thực sự bằng cách



Hình 1.10: Các mức kiểm thử trong mô hình thác nước.

đặt các đơn vị với nhau theo phương pháp tăng dần. Kỹ thuật kiểm thử tích hợp sẽ được mô tả chi tiết trong chương 10.

- Kiểm thử hệ thống: Kiểm thử mức này được áp dụng khi đã có một hệ thống đầy đủ sau khi tất cả các thành phần đã được tích hợp. Mục đích của kiểm thử hệ thống là để đảm bảo rằng việc cài đặt tuân thủ đầy đủ các yêu cầu được đặc tả của người dùng. Công việc này tốn nhiều công sức, vì có nhiều khía cạnh về yêu cầu người dùng cần được kiểm thử. Các phương pháp kiểm thử chức năng được trình bày trong chương 5 là thích hợp nhất cho việc kiểm thử này. Các kỹ thuật kiểm thử hệ thống được trình bày trong chương 10.

- Kiểm thử chấp nhận: Khi kiểm thử hệ thống đã được hoàn tất và sản phẩm cần kiểm thử thỏa mãn các yêu cầu của đặc tả phần mềm mức hệ thống, sản phẩm đó vẫn chưa sẵn sàng để đưa vào sử dụng. Lý do là phần mềm cần kiểm thử cần trải qua giai đoạn kiểm thử chấp nhận để kiểm tra xem sản phẩm có đáp ứng các yêu cầu của khách hàng. Kiểm thử chấp nhận được thực thi bởi chính các khách hàng nhằm đảm bảo

rằng sản phẩm phần mềm làm việc đúng như họ mong đợi. Có hai loại kiểm thử chấp nhận: kiểm thử chấp nhận người dùng, được tiến hành bởi người dùng, và kiểm thử chấp nhận doanh nghiệp, được tiến hành bởi nhà sản xuất ra sản phẩm phần mềm. Chương 10 sẽ mô tả chi tiết các kỹ thuật hỗ trợ kiểm thử chấp nhận.

1.7 Tổng kết

Chúng ta đã trình bày trong chương này một bức tranh tổng thể về việc kiểm thử phần mềm bao gồm các thuật ngữ và khái niệm cơ bản trong lĩnh vực kiểm thử, mục đích và vai trò của việc kiểm thử phần mềm. Chúng ta cũng mô tả sơ lược về bài toán kiểm thử và quy trình để giải bài toán này gồm việc phân tích tĩnh và phân tích động. Dóng vai tró trung tâm của phân tích động là việc xây dựng các ca kiểm thử. Hai cách tiếp cận cơ bản để xác định các ca kiểm thử là kiểm thử chức năng (kiểm thử hộp đen) và kiểm thử cấu trúc (kiểm thử hộp trắng). Cả hai cách tiếp cận này bổ sung cho nhau để xây dựng tập các ca kiểm thử hiệu quả. Chúng ta cũng đã giới thiệu trong chương này bốn mức kiểm thử, mỗi mức tương ứng với một giai đoạn của việc phát triển phần mềm. Các mức đó là: kiểm thử đơn vị, kiểm thử tích hợp, kiểm thử hệ thống và kiểm thử chấp nhận. Chi tiết về các phương pháp kiểm thử sẽ được trình bày trong các chương sau.

1.8 Bài tập

1. Hãy vẽ biểu đồ Venn phản ánh khẳng định: ta đã không làm cái mà lẽ ra ta cần phải làm, và làm cái mà lẽ ra ta không được làm.

2. Mô tả mỗi miền trong bảy miền trong hình 1.4

3. Một trong các câu chuyện cũ về lĩnh vực phần mềm mô tả một nhân viên cáu kỉnh viết một chương trình quản lý lương. Chương trình có chức năng kiểm tra số chứng minh thư của cán bộ và nhân viên trước khi đưa ra bản tính lương. Nếu có lúc người nhân viên này bị đuổi việc, chương trình sẽ tạo ra một mã độc gây hại cho cơ quan. Hãy bàn về tình trạng này theo các thuật ngữ trên đây về lỗi, sai, dạng thất bại và quyết định dạng kiểm thử nào là thích hợp.

4. Hãy so sánh hai cách tiếp cận kiểm thử chắc năng và kiểm thử cấu trúc.

Chương 2

Một số ví dụ

Chương này trình bày một số ví dụ mà sẽ được dùng trong các chương tiếp theo nhằm minh họa cho các phương pháp kiểm thử. Các ví dụ này gồm: bài toán tam giác và hàm NextDate tương đối phức tạp về mặt lôgic. Các ví dụ này liên quan đến một số vấn đề mà người kiểm thử sẽ gặp trong quá trình kiểm thử. Khi bàn về kiểm thử tích hợp và kiểm thử hệ thống trong chương 10, ta sẽ dùng ví dụ về một phiên bản đơn giản của máy rút tiền tự động (ATM).

Trong chương này các ví dụ mức đơn vị, cài đặt bằng ngôn ngữ C, sẽ được trình bày cho mục đích kiểm thử cấu trúc. Các mô tả mức hệ thống của máy ATM dưới dạng một tập các sơ đồ dòng dữ liệu và máy hữu hạn trạng thái sẽ được trình bày trong các chương tiếp theo.

2.1 Bài toán tam giác

Kể từ ngày được công bố lần đầu dưới dạng một ví dụ của kiểm thử cách đây 30 năm [Gru73], bài toán tam giác đã được nhắc tới

trong nhiều bài báo và sách về kiểm thử, chẳng hạn trong các tài liệu [Gru73, BL75, Mye75, S.82, AJ83, AJ84, Mal87, Bil88].

2.1.1 Phát biểu bài toán

Bài toán tam giác nhận ba số nguyên làm các dữ liệu đầu vào; các dữ liệu này là số đo các cạnh của một tam giác. Đầu ra của chương trình là loại của tam giác xác định bởi ba cạnh ứng với các số đo này: tam giác đều, tam giác cân, tam giác thường, hoặc không là tam giác. Ta sẽ dùng các từ tiếng Anh làm dữ liệu đầu ra tương ứng cho các loại này như lấy từ ví dụ nguyên thủy: Equilateral, Isosceles, Scalene, hoặc NotATriangle. Bài toán này đôi khi được mở rộng với đầu ra thứ năm là tam giác vuông (right triangle). Trong các bài tập, ta sẽ dùng bài toán mở rộng như vậy.

2.1.2 Nhận xét

Một trong các lý do làm bài toán này được sử dụng rất phổ biến có thể là vì nó tiêu biểu cho việc định nghĩa không đầy đủ làm phương hại đến việc trao đổi thông tin giữa khách hàng, người phát triển và người kiểm thử. Đặc tả này giả thiết rằng người phát triển biết các chi tiết về tam giác, đặc biệt tính chất sau của tam giác: tổng của hai cạnh bất kỳ cần thực sự lớn hơn cạnh còn lại. Nếu a, b và c ký hiệu cho ba cạnh của tam giác thì tính chất trên được biểu diễn chính xác bằng ba bất đẳng thức toán học $a < b + c$, $b < a + c$ và $c < a + b$. Nếu bất kỳ một trong ba bất đẳng thức này không được thỏa mãn thì a, b và c không tạo thành ba cạnh của một tam giác. Nếu cả ba cạnh đều bằng nhau, chúng tạo thành tam giác đều, nếu chỉ có một cặp cạnh bằng nhau, chúng tạo thành tam giác cân và nếu không có cặp cạnh nào bằng nhau thì chúng là độ dài ba cạnh của một tam giác thường. Một người kiểm thử giỏi có thể làm rõ ý nghĩa bài toán này hơn nữa bằng việc đặt giới hạn cho các độ dài của các cạnh. Ví dụ, câu trả lời nào cho trường hợp khi đưa vào

chương trình ba số $-5, -4$ và -3? Ta sẽ đòi hỏi là các cạnh phải ít nhất là bằng 1, và khi đó ta cũng có thể khai báo giới hạn của cận trên, chẳng hạn 20000.

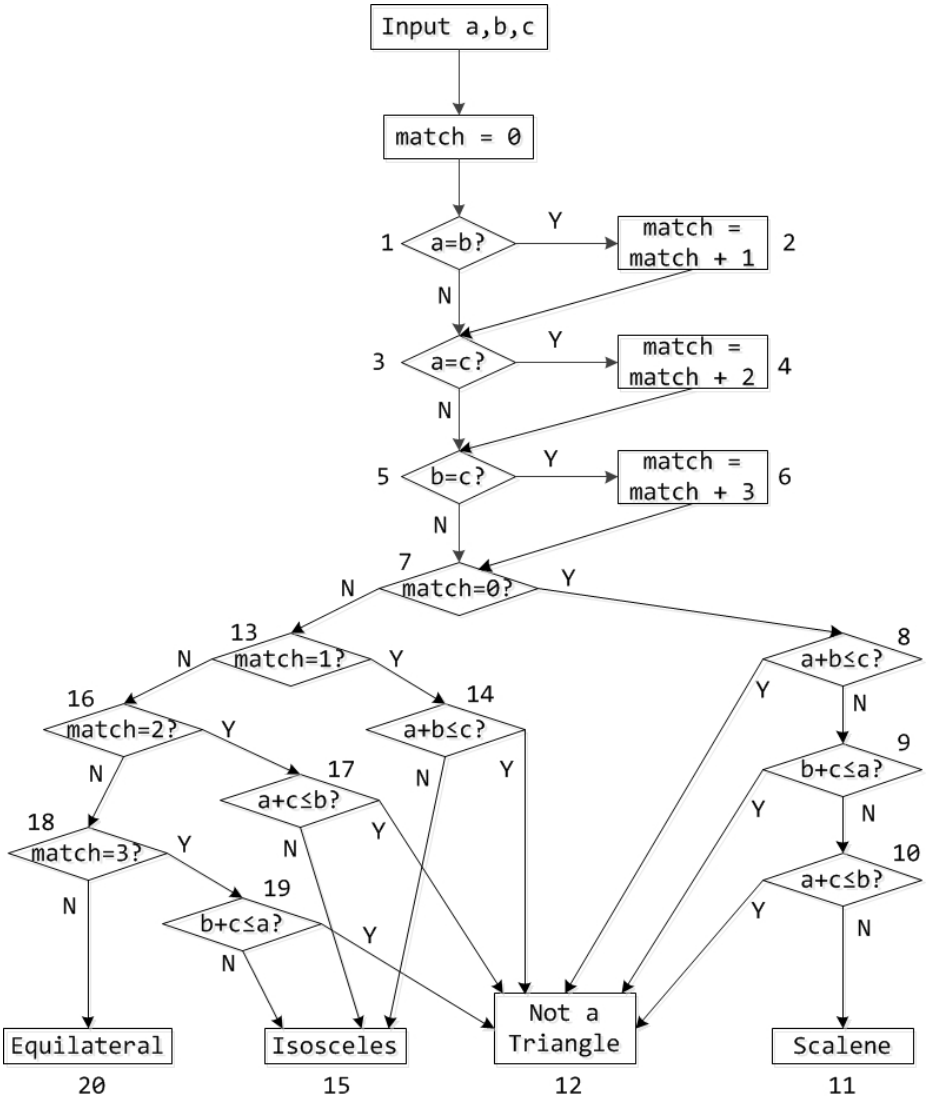
2.1.3 Cài đặt truyền thống

Cài đặt truyền thống của ví dụ cổ điển này có kiểu tựa FORTRAN [S.82]. Tuy nhiên, chúng tôi chuyển cài đặt của ví dụ này sang ngôn ngữ C để thống nhất với các ví dụ khác trong giáo trình này. Sơ đồ khối của ví dụ này được biểu thị trong hình 2.1. Các số của khối trong sơ đồ này tương ứng với các chú giải trong chương trình sau đây. Một cài đặt có cấu trúc hơn sẽ được cho trong mục 2.1.4.

Biến `match` được dùng để ghi nhận sự bằng nhau giữa các cặp cạnh. Nếu hai cạnh bằng nhau, chẳng hạn $a = c$, thì chỉ cần so sánh $a + c$ với b (do $b > 0$, $a + b > c$ sẽ phải thỏa mãn vì $a = c$). Nhờ quan sát này, chúng ta có thể rút gọn số các so sánh cần làm. Cái giá phải trả cho tính hiệu quả này chỉ là sự rõ ràng và dễ kiểm thử!.

Trong các chương tiếp theo, ta sẽ thấy lợi thế của phiên bản này khi bàn đến các đường đi thực thi được của chương trình. Đó là lý do tốt nhất để giữ lại bản này.

```
int main(){
  int  a, b, c, match;
  printf("Enter 3 sides (a, b, c) of a triangle \n");
  printf("a = ");
  scanf("%d",&a);
  printf("b = ");
  scanf("%d",&b);
  printf("c = ");
  scanf("%d",&c);
  printf ("Side A is %d\n", a);
  printf ("Side B is %d\n", b);
```



Hình 2.1: Sơ đồ khối cho cài đặt chương trình tam giác truyền thống.

```
printf ("Side C is %d\n", c);
match = 0;
if(a == b)                                          {1}
    match = match + 1;                              {2}
if(a == c)                                          {3}
```

```
    match = match + 2;                                  {4}
  if(b == c)                                            {5}
    match = match + 3;                                  {6}
  if(match == 0)                                        {7}
    if((a+b) <= c)                                      {8}
       printf("Not a Triangle");                      {12.1}
    else if((b+c) <= a)                                 {9}
            printf("Not a Triangle");                {12.2}
         else  if((a+c) <= b)                          {10}
                  printf("Not a Triangle");          {12.3}
               else printf("Triangle is Scalene");{11}
  else
    if(match == 1)                                      {13}
         if((a+c) <= b)                                 {14}
            printf("Not a Triangle");                {12.4}
         else printf("Triangle is Isosceles");      {15.1}
    else
         if(match == 2)                                 {16}
            if((a+c) <= b)                              {17}
                printf("Not a Triangle");            {12.5}
            else printf("Triangle's Isoscel.");{15.2}
         else if(match == 3)                            {18}
                if((b+c) <= a)                          {19}
                  printf("Not a Triangle");          {12.6}
                else
                  printf("Triangle's Isoscel.");{15.3}
            else printf("Triangle's Equilat.");      {20}
   return 0;
}//the end.
```

Lưu ý là có sáu cách để đi đến nút "Not A Triangle" (12.1 – 12.6) và có ba cách để đi đến nút "Isosceles" (15.1 – 15.3).

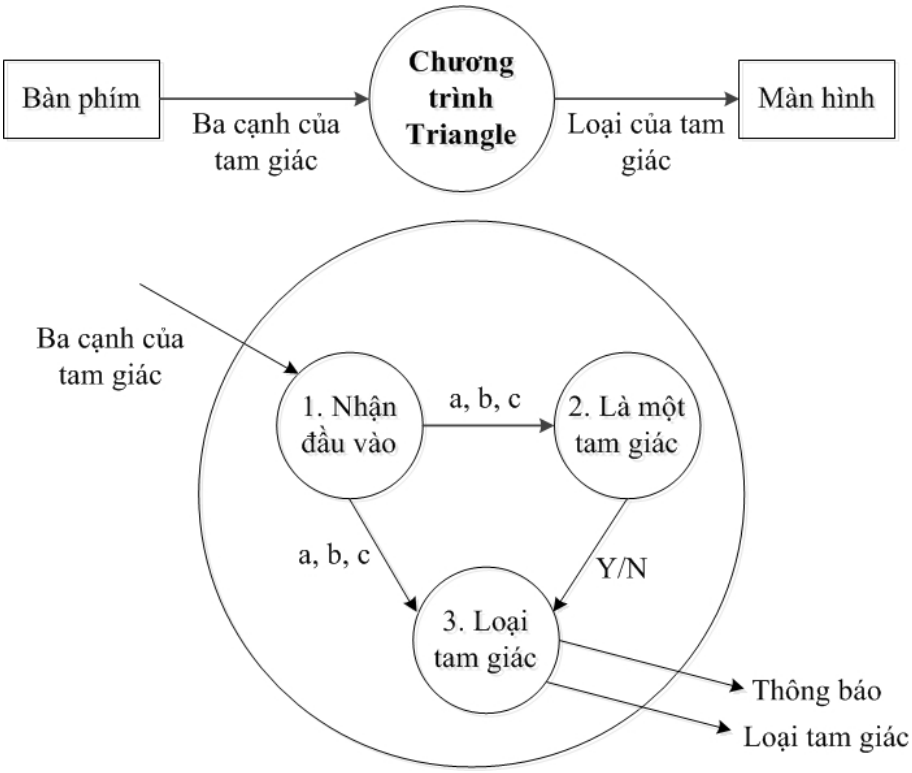
2.1.4 Cài đặt có cấu trúc

Hình 2.2 là một mô tả sơ đồ dòng dữ liệu cho cài đặt có cấu trúc của chương trình tam giác. Ta có thể cài đặt bằng một chương trình chính và bốn thủ tục. Vì ta sẽ dùng ví dụ này cho việc kiểm thử đơn vị, bốn thủ tục đã được kết hợp thành một chương trình C. Các dòng chú giải liên kết các đoạn mã với việc phân rã cho trong hình 2.2.

```
int main(){
     int a, b, c, IsATriangle;

     //Function 1: Get Input
     printf("Enter 3 sides (integers) of a triangle");
     printf("a = ");
     scanf("%d",&a);
     printf("b = ");
     scanf("%d",&b);
     printf("c = ");
     scanf("%d",&c);
     printf ("Side A is %d\n", a);
     printf ("Side B is %d\n", b);
     printf ("Side C is %d\n", c);

     //Function 2: Is A Triangle?
     if((a < b + c) && (b < a + c) && (c < a + b))
          IsATriangle = 1;
     else IsATriangle = 0;

     //Function 3: Determine Triangle Type
     if(IsATriangle)
          if((a == b) && (b == c))
               printf("Triangle is Equilateral");
```

```
        else if((a != b) && (a != c) && (b != c))
                printf("Triangle is Scalene");
            else printf("Triangle is Isosceles");
    else printf("Not a Triangle");

    return 0;
}//the end
```

Lưu ý: Function 4 và Output Controller đã được kết hợp thành các lệnh trong Function 3.



Hình 2.2: Sơ đồ dòng dữ liệu cho cài đặt của chương trình tam giác.

2.2 Hàm NextDate (ngày kế tiếp)

Độ phức tạp của chương trình tam giác nằm ở các mối quan hệ giữa dữ liệu đầu vào và dữ liệu đầu ra. Hàm NextDate nhằm minh họa một loại độ phức tạp khác: mối quan hệ giữa các biến đầu vào.

2.2.1 Phát biểu bài toán

NextDate là một hàm có ba biến biểu diễn ngày, tháng và năm là `day`, `month` và `year`. Hàm này trả về ngày kế tiếp của ngày đầu vào. Các biến `day, month, year` có các giá trị số thỏa mãn các ràng buộc: $1 \leq \texttt{day} \leq 31, 1 \leq \texttt{month} \leq 12, 1812 \leq \texttt{year} \leq 2012$.

2.2.2 Nhận xét

Có hai nguyên nhân tạo nên độ phức tạp của hàm NextDate: độ phức tạp nêu trong các ràng buộc trên đây của miền dữ liệu đầu vào, và quy tắc phân biệt giữa năm nhuận và năm không nhuận. Do trung bình một năm có $365,2422$ ngày, năm nhuận được dùng để giải quyết ngày "vượt trội". Nếu ta chấp thuận cứ bốn năm lại có một năm nhuận thì sẽ có một sai số nhỏ. Lịch Gregorian (đề xuất năm 1582 bởi Giáo hoàng Gregory) giải quyết vấn đề này bằng cách điều chỉnh các năm nhuận theo năm thế kỷ (những năm chia hết cho 100). Do đó, một năm là nhuận nếu nó chia hết cho 4 nhưng không là năm thế kỷ. Các năm thế kỷ là nhuận khi và chỉ khi nó là bội của 400 [Ing61, fS91]. Do đó các năm 1992, 1996 và 2000 là năm nhuận, nhưng năm 1900 lại không phải là năm nhuận.

Hàm NextDate cũng minh họa một khía cạnh của kiểm thử phần mềm. Ta thường thấy các ví dụ về luật Zipf - nói rằng 80% các hoạt động xảy ra tại chỉ 20% của không gian. Ta cũng thấy ở đây phần lớn mã nguồn (80% các hoạt động) được dành cho các năm nhuận (20% của không gian).

2.2.3 Cài đặt

```
int main(){
typedef struct{
     int month;
        int day;
        int year;
}dataType;

dataType today, tomorrow;

printf("Enter day:\t");
scanf("%d", &today.day);
printf("Enter month:\t");
scanf("%d", &today.month);
printf("Enter year:\t");
scanf("%d", &today.year);

tomorrow = today;

if(today.month==1 || today.month==3 || today.month==5
|| today.month==7 || today.month==8 || today.month==10)
     if(today.day < 31)
             tomorrow.day = today.day + 1;
     else{
         tomorrow.day = 1;
         tomorrow.month = today.month + 1;
     }

    if(today.month==4 || today.month==6
       || today.month==9 || today.month == 11)
        if(today.day < 30)
```

```
    tomorrow.day = today.day + 1;
else{
     tomorrow.day = 1;
     tomorrow.month = today.month + 1;
 }

if(today.month == 12)
  if(today.day < 31)
    tomorrow.day = today.day + 1;
else{
     tomorrow.day = 1;
     tomorrow.month = 1;
     if(today.year == 2012)
       printf("2012 is over");
     else  tomorrow.year = today.year + 1;
 }

if(today.month == 2)
  if(today.day < 28)
    tomorrow.day = today.day + 1;
else{
 if(today.day == 28)
 if(((today.year%4 == 0)&&(today.year%100 != 0))||
 (today.year%400 != 0))
       tomorrow.day = 29;//leap year
 else{
       tomorrow.day = 1;
       tomorrow.month = 3;
 }
 else
 if(today.day == 29){
       tomorrow.day = 1;
```

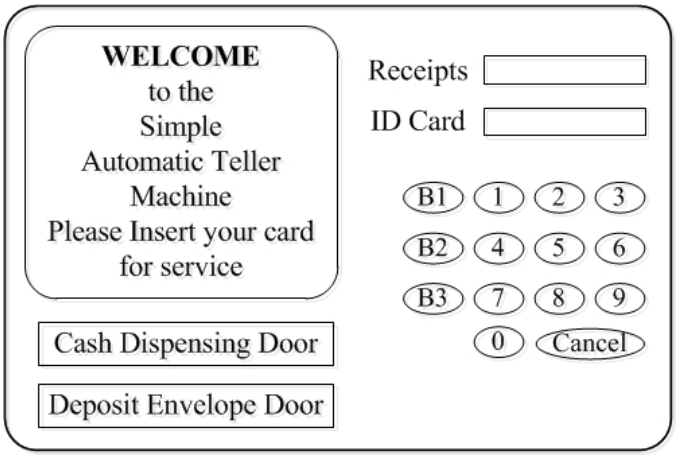
```
            tomorrow.month = 3;
      }
      else printf("Cannot have Feb.", today.day);
       }

    printf("Tomorrow's date: %3d %3d %5d", tomorrow.day,
            tomorrow.month, tomorrow.year);

return 0;
}//the end
```

2.3 Hệ thống rút tiền tự động đơn giản

Để bàn về các vấn đề về kiểm thử tích hợp và kiểm thử hệ thống dễ dàng hơn, ta cần một ví dụ với phạm vi lớn hơn. Hệ thống rút tiền tự động (ATM) được trình bày ở đây là một phiên bản chi tiết hơn của hệ thống trong [ADP93]. Hệ thống này gồm một biến thể khá hay về chức năng và giao diện. Dù nó đặc trưng cho các hệ thời gian thực nhưng vẫn thích hợp với mục đích của ta vì các nhà thực hành trong lĩnh vực thương mại điện tử đều cho rằng ngay cả các hệ COBOL truyền thống cũng gặp nhiều vấn đề liên quan đến các hệ thời gian thực.

2.3.1 Phát biểu bài toán

Hệ thống rút tiền tự động giao tiếp với các khách hàng của ngân hàng thông qua 15 màn hình như được thấy trong hình 2.4. Khách hàng có thể chọn một trong ba loại giao dịch: gửi tiền vào tài khoản, rút tiền và kiểm tra số dư trong tài khoản. Những loại giao dịch này có thể được thực hiện đối với hai loại tài khoản là tiết kiệm hoặc vãng lai.



Hình 2.3: Trạm rút tiền ATM.

Khi một khách hàng của ngân hàng đến một trạm rút tiền, màn hình 1 sẽ xuất hiện. Mỗi khách hàng truy cập máy ATM với thẻ nhựa có chứa mã hóa của số tài khoản cá nhân (PAN) mà chính là khóa để truy cập tệp tài khoản nội bộ của khách hàng, trong đó có tên và thông tin về tài khoản của khách hàng. Nếu PAN trùng khớp với thông tin trong tệp khách hàng, hệ thống sẽ hiển thị màn hình 2 cho khách hàng giao dịch. Ngược lại, nếu PAN của khách hàng không tồn tại, màn hình 4 được hiển thị và thẻ của khách hàng sẽ bị giữ.

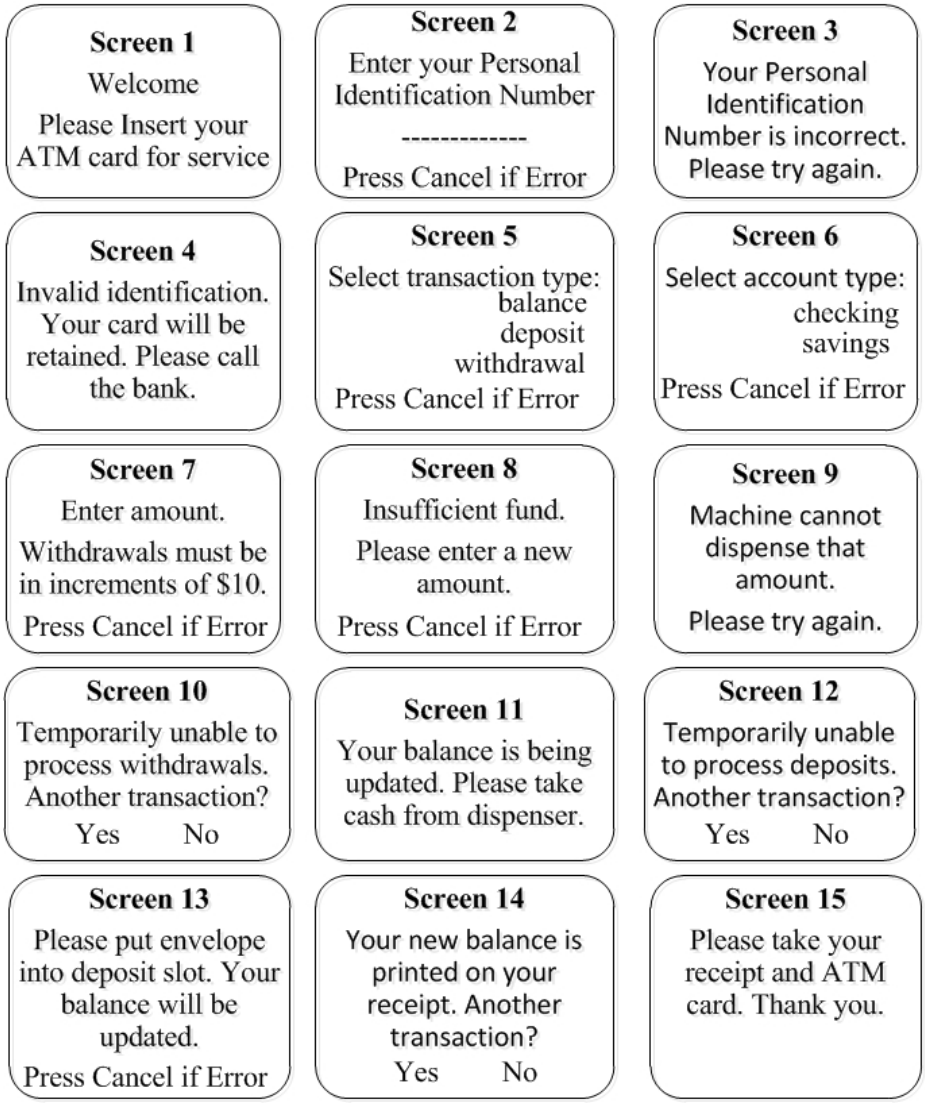
Trên màn hình 5, hệ thống thêm hai mẩu thông tin vào tệp tài khoản của khách hàng: ngày hiện tại và tăng số các khoản mục về ATM. Khách hàng chọn giao dịch mong muốn từ các tùy chọn ở màn hình 5. Sau đó hệ thống lập tức chuyển qua màn hình 6 và ở đây khách hàng có thể chọn tài khoản để áp dụng giao dịch đã chọn.

Nếu yêu cầu kiểm tra số dư, hệ thống kiểm tra tệp ATM địa phương đối với bất kỳ các giao dịch chưa gửi đi và kết hợp với các giao dịch này với số dư ban đầu đối với ngày đó ở tệp tài khoản khách hàng. Sau đó màn hình 14 xuất hiện. Nếu yêu cầu gửi tiền

vào tài khoản, trạng thái của khe phong bì gửi tiền được xác định từ một trường trong tệp điều khiển của trạm. Nếu không có vấn đề gì, hệ thống hiển thị màn hình 7 để nhận lượng tiền giao dịch. Nếu có vấn đề với khe phong bì gửi tiền, hệ thống hiển thị màn hình 12. Một khi lượng tiền gửi đã được nạp vào, hệ thống hiển thị màn hình 13, chấp nhận phong bì tiền gửi và xử lý việc gửi tiền. Lượng tiền gửi được nạp là lượng tiền chưa gửi đi trong tệp ATM địa phương, và số đếm số lần gửi trong tháng được tăng lên. Cả hai số này được xử lý bởi hệ ATM chủ (tập trung) một lần mỗi ngày. Hệ thống sau đó hiển thị màn hình 14.

Nếu yêu cầu rút tiền từ tài khoản, hệ thống kiểm tra trạng thái của khe nhả tiền (kẹt hay không) trong tệp điều khiển trạm. Nếu bị kẹt thì màn hình 9 xuất hiện. Trái lại màn hình 7 được hiển thị để khách hàng có thể khai báo lượng tiền định rút. Khi dữ liệu này đã vào xong, hệ thống kiểm tra tệp trạng thái của trạm để biết lượng tiền trong máy có đủ trả không. Nếu không đủ, hiển thị màn hình 9, trái lại việc rút tiền được xử lý. Hệ thống kiểm tra số dư tài khoản của khách hàng (như được mô tả trong giao dịch kiểm tra số dư tài khoản) và nếu không đủ để rút thì hiển thị màn hình 8. Nếu số dư trong tài khoản là đủ thì hiển thị màn hình 11 và nhả tiền. Lượng tiền được rút khi đó được ghi vào tệp ATM địa phương chưa gửi và số lần rút trong tháng được tăng lên. Số dư trong tài khoản được in ra trong sao kê. Sau khi khách hàng lấy tiền, hệ thống hiển thị màn hình 14.

Khi nút No được chọn trong các màn hình 10, 12, hoặc 14, hệ thống xuất hiện màn hình 15 và nhả thẻ ATM trả lại cho khách hàng. Khi thẻ đã được khách hàng nhận và lấy đi, hệ thống trở lại màn hình 1. Khi nút Yes được chọn trong các màn hình 10, 12, hoặc 14, hệ thống hiển thị màn hình 5 để khách hàng chọn các giao dịch khác.



Hình 2.4: Các màn hình của máy ATM đơn giản.

2.3.2 Nhận xét

Có một lượng thông tin đáng kể đã bị bỏ quên trong hệ thống ATM vừa mô tả. Ví dụ, ta có thể suy ra rằng trạm chỉ gồm các tờ mười (10) ngàn đồng (xem màn hình 7). Định nghĩa bằng văn bản có lẽ

chính xác hơn so với tình huống ta gặp trong thực tế. Ví dụ này đã được đơn giản hóa một cách có tính toán.

Có nhiều câu hỏi cần giải quyết bởi xâu các giả thiết. Chẳng hạn, có chăng một giới hạn cho vay? Cơ chế nào để ngăn chặn việc khách hàng lấy số tiền vượt quá số dư thực sự của tài khoản bằng cách đến các máy ATM khác nhau trong cùng ngày? Có nhiều câu hỏi về khởi tạo: có bao nhiêu tiền trong máy? Bao nhiêu khách hàng mới được thêm vào hệ thống? Những vấn đề này và nhiều chi tiết đời thực khác đã bị bỏ qua để đảm bảo tính đơn giản của bài toán này.

2.4 Bộ điều khiển gạt nước ô tô

Gạt nước ô tô trên xe Saturn (bản năm 1992) được điều khiển bởi một chốt và một núm vặn. Chốt có bốn vị trí: OFF, INT (thỉnh thoảng), LOW và HIGH. Núm vặn có ba vị trí, được đánh số 1, 2 và 3. Các vị trí của núm vặn tương ứng với ba tốc độ chậm khác nhau. Vị trí của núm vặn chỉ có tác dụng khi chốt ở vị trí INT. Bảng 2.1 chỉ ra các tốc độ của cần gạt tương ứng với vị trí của chốt và núm vặn, trong đó giá trị n/a nghĩa là trường hợp này không áp dụng.

Bảng 2.1: Tốc độ của cần gạt ứng với vị trí của chốt và núm vặn

Chốt	OFF	INT	INT	INT	LOW	HIGH
Núm vặn	n/a	1	2	3	n/a	n/a
Cần gạt	0	4	6	12	30	60

2.5 Bài tập

1. Xem lại sơ đồ khối của bài toán tam giác trong hình 2.1. Liệu biến chương trình `match` có thể có: giá trị 4?, giá trị 5? Liệu có thể tiến hành dãy các khối 1, 2, 5, 6?

2. Nhắc lại lời bàn trong chương 1 về mối quan hệ giữa đặc tả và cài đặt của chương trình. Nếu xem xét cẩn thận việc cài đặt của hàm NextDate, bạn sẽ thấy một vấn đề như sau. Hãy nhìn vào lệnh `CASE` đối với các tháng có 30 ngày (4, 6, 9, 11). Không có một hành động đặc biệt cho trường hợp `day` = 31. Hãy bàn xem cài đặt này có đúng đắn hay không. Lặp lại lời bàn này cho việc xử lý của trường hợp `day` $\geq$ 29 trong lệnh `CASE` đối với tháng hai (February).

3. Trong chương 1, ta đã nói rằng một bộ phận của ca kiểm thử là dữ liệu đầu ra mong đợi. Đối với hàm NextDate bạn dùng cái gì làm dữ liệu đầu ra mong đợi cho các ca kiểm thử của ngày 31 tháng 6 năm 1812 (June 31, 1812)? Tại sao?

4. Một mở rộng của bài toán tam giác là kiểm tra đối với tam giác vuông. Ba cạnh tạo thành một tam giác vuông nếu hệ thức Pythagor thỏa mãn:

$$c^2 = a^2 + b^2.$$

   Thay đổi này đưa đến một đòi hỏi hợp lý là các cạnh cần được cho theo thứ tự giảm dần về độ dài, tức là $a \leq b \leq c$. Hãy mở rộng chương trình `Triangle2` để đưa thêm đặc trưng tam giác vuông. Mở rộng này sẽ được dùng trong các chương tiếp theo về kiểm thử hộp đen và kiểm thử hộp trắng.

5. Chương trình tam giác sẽ làm gì với các cạnh $-3, -3$ và 5?

6. Hàm YesterDate ngược lại với hàm NextDate. Cho trước ngày, tháng và năm, hãy tìm ngày trước đó. Hãy xây dựng một chương trình cho YesterDate bằng bất kể ngôn ngữ lập trình nào.

7. Theo lịch Gregorian được đề xuất năm 1582, hãy tính năm đầu tiên trong đó hệ thống năm nhuận cho dư một ngày đầy đủ.

Chương 3

Cơ sở toán học rời rạc cho việc kiểm thử

Việc kiểm thử cần đến các mô tả và phân tích của toán học. Chương này cung cấp những kiến thức cơ sở về toán cần thiết đối với người kiểm thử. Các chủ đề toán học được chọn để trình bày ở đây là các công cụ mà mỗi người kiểm thử cần nắm vững để sử dụng. Với những công cụ này, người kiểm thử sẽ có được độ chính xác và tính hiệu quả từ toán học nhằm cải tiến công việc của mình. Các lời bàn ít tính hình thức trong các chương này có thể không làm vừa ý các nhà toán học nhưng thích hợp với môi trường kiểm thử. Nếu bạn cảm thấy bạn đã vững về toán rồi thì có thể bỏ qua chương này.

Nói chung, lý thuyết tập hợp, hàm, quan hệ, lôgic và lý thuyết xác suất được áp dụng nhiều trong các phương pháp kiểm thử chức năng (kiểm thử hộp đen), còn lý thuyết đồ thị được dùng nhiều trong các phương pháp kiểm thử cấu trúc (kiểm thử hộp trắng). Từ "rời rạc" có thể làm nảy sinh câu hỏi: thế thì cái gì là toán không rời rạc? Đó là toán liên tục, mà thường là các phép tính vi và tích phân, các hàm liên tục mà các nhà kiểm thử ít khi dùng

đến. Toán rời rạc liên quan đến các cấu trúc rời rạc như lý thuyết tập hợp, hàm và quan hệ, lôgic mệnh đề, phép tính xác suất rời rạc v.v. Chúng ta sẽ đề cập đến các chủ đề này trong các mục sau đây.

3.1 Lý thuyết tập hợp

Rất tiếc là không có một định nghĩa tường minh, chính xác và dễ hiểu về tập hợp dù tập hợp đóng vai trò trung tâm trong toán học. Các nhà toán học phân biệt lý thuyết tập hợp ngây thơ (naive set theory) với lý thuyết tập hợp tiên đề (axiomatic set theory). Trong khuôn khổ của giáo trình này, chúng ta sẽ dùng lý thuyết tập hợp ngây thơ. Trong lý thuyết này, một tập hợp được thừa nhận như là một thứ nguyên thủy giống như các khái niệm điểm và đường trong hình học. Một vài từ thông dụng khác cho tập hợp cũng được dùng phổ biến như: đám đông, bó, nhóm, v.v. Khái niệm của ta về tập hợp là một nhóm các đối tượng, một vài hoặc tất cả các đối tượng được xét. Chẳng hạn tập hợp các tháng có đúng 30 ngày (chúng ta đã dùng tập hợp này trong hàm NextDate ở chương 2). Trong lý thuyết tập hợp, ta viết:

$$M_1 = \{\text{tháng tư}, \text{tháng sáu}, \text{tháng chín}, \text{tháng mười một}\}$$

và đọc là "M_1 là tập hợp mà các phần tử của nó là các tháng tư, sáu, chín và mười một."

3.1.1 Phần tử của tập hợp

Các thứ ở trong một tập hợp được gọi là các phần tử hay thành viên của tập hợp đó. Mối quan hệ thành viên này được ký hiệu bởi $\in$. Do đó, ta có thể viết tháng mười một $\in M_1$. Khi đối tượng nào đó không ở trong tập hợp, ta dùng ký hiệu $\notin$. Vì thế ta viết tháng mười hai $\notin M_1$.

3.1.2 Định nghĩa tập hợp

Có ba cách để định nghĩa một tập hợp: liệt kê tất cả các phần tử của nó, hoặc bằng một quy tắc quyết định xác định các phần tử của nó, hoặc bằng việc xây dựng từ các tập hợp khác đã biết. Cách thứ nhất dùng cho các tập có một số ít các phần tử và các tập có các phần tử tuân theo một quy luật nào đó có thể liệt kê được. Ví dụ ta có thể định nghĩa tập các năm trong chương trình NextDate như sau:

$$Y = \{1812, 1813, 1814, \ldots, 2011, 2012\}$$

Lưu ý là thứ tự của các phần tử là không quan trọng trong biểu diễn bằng phương pháp liệt kê. Cách biểu diễn bằng quy tắc quyết định là phức tạp hơn và có cả lợi lẫn bất lợi. Ta cũng có thể định nghĩa tập các năm trong chương trình NextDate theo cách này:

$$Y = \{\texttt{năm} \ : \ 1812 \leq \texttt{năm} \leq 2012\}$$

và đọc là "Y là tập các năm sao cho chúng nằm giữa 1812 và 2012, bao gồm cả hai cận". Quy tắc quyết định dùng để định nghĩa tập hợp cần phải không nhập nhằng. Cho bất kỳ một năm, ta có thể quyết định nhờ quy tắc này xem nó có ở trong tập Y hay không.

Lợi thế của việc định nghĩa tập hợp bằng quy tắc quyết định là nó bắt buộc tính rõ ràng và không nhập nhằng. Người kiểm thử có kinh nghiệm hay gặp phải các yêu cầu "không khả kiểm". Thường thì lý do các yêu cầu như vậy không thể được kiểm thử là do quy tắc quyết định bị nhập nhằng. Trong ví dụ chương trình tam giác chẳng hạn, nếu ta định nghĩa một tập hợp N:

$$N = \{t \ : \ t \text{ là một tam giác gần đều}\}$$

Ta có thể nói tam giác với ba cạnh $(500, 500, 501)$ là ở trong N, nhưng xử lý thế nào với các tam giác với ba cạnh là $(50, 50, 51)$ hoặc $(5, 5, 6)$?

Lợi thế nữa của việc định nghĩa tập hợp bằng các quy tắc quyết định là ta có thể viết ra các tập hợp mà các phần tử của chúng không thể hoặc khó có thể liệt kê được. Chẳng hạn ta có thể phải quan tâm đến các tập hợp dạng:

$$S = \{sales \ : \ \text{sales có tỷ lệ hoa hồng } 15\%\}$$

Chẳng dễ gì để có thể viết ra các phần tử của tập này, nhưng cho trước một giá trị cho sales ta luôn quyết định được nó có thuộc tập này hay không.

Điều bất lợi của cách biểu diễn này là đôi khi khá phức tạp, chẳng hạn quy tắc được biểu diễn bằng các biểu thức lôgic với các lượng từ tồn tại (ký hiệu là $\exists$) và với mọi (ký hiệu là $\forall$). Nếu ai cũng hiểu cách ký hiệu này thì chúng rất lợi do tính chính xác, nhưng lại có rất nhiều khách hàng chẳng hiểu gì về các lượng từ này cả. Một vấn đề phiền phức khác trong cách biểu diễn này là khi các quy tắc là đệ quy, chúng sẽ thiếu tính trực quan và khó hiểu.

3.1.3 Tập hợp rỗng

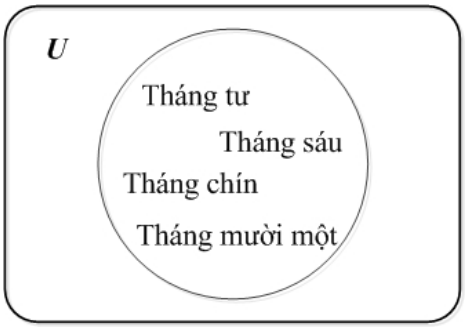
Tập hợp rỗng là tập không có một phần tử nào và được ký hiệu là $\emptyset$. Tập hợp rỗng đóng vai trò đặc biệt trong lý thuyết tập hợp với các tính chất

- Tập rỗng là duy nhất, chỉ có một tập rỗng mà thôi.
- Các tập $\emptyset$, $\{\emptyset\}$, $\{\{\emptyset\}\}$, ... đều là khác nhau.

Một tập hợp được định nghĩa bởi một quy tắc không thể thỏa mãn được chính là tập rỗng. Ví dụ, $\{y \ : \ 2012 \leq y \leq 1983\} = \emptyset$.

3.1.4 Biểu đồ Venn

Các tập hợp có thể được minh họa bởi các biểu đồ Venn như đã giới thiệu ở chương 1 khi bàn về tập các hành vi đặc tả và tập các



Hình 3.1: Biểu đồ Venn của tập các tháng có 30 ngày.

hành vi được lập trình. Trong biểu đồ Venn, một tập hợp được biểu diễn bởi một hình tròn, và các điểm bên trong hình tròn tương ứng với các phần tử của tập hợp. Ta có thể biểu diễn tập M_1 của các tháng gồm 30 ngày như ở hình 3.1.

Các biểu đồ Venn truyền tải nhiều thông tin về mối quan hệ khác nhau của tập hợp một cách trực quan. Tuy nhiên, cũng có vài vấn đề nho nhỏ với các biểu đồ Venn như việc phân biệt giữa tập vô hạn và tập hữu hạn ra sao. Cả hai đều được vẽ bằng các biểu đồ Venn. Trong trường hợp hữu hạn, ta không thể giả thiết mỗi điểm trong đều biểu diễn một phần tử của tập hợp. Dù không lo về điều đó, nhưng cũng nên biết về các hạn chế này. Đôi khi ta thấy hữu ích khi gán nhãn cho các phần tử riêng biệt nào đó. Một điểm cần lưu ý khác nữa là việc xử lý tập hợp rỗng. Làm thế nào để chỉ rằng một tập hợp hoặc một phần của nó là tập rỗng? Ta có thể gạch chéo các miền rỗng và gắn chú thích vào.

Thông thường, các tập hợp được xét trong một bài toán là các tập con của một tập lớn hơn nào đó được gọi là vũ trụ chuyên đề. Trong chương 1 ta chọn tập tất cả các hành vi chương trình làm vũ trụ chuyên đề. Vũ trụ chuyên đề luôn được dự đoán từ các tập được cho. Trong hình 3.1, ta có thể lấy vũ trụ chuyên đề là tập tất cả các tháng trong năm. Người kiểm thử cần nhận ra rằng việc giả thiết các vũ trụ chuyên đề dễ dẫn đến nhầm lẫn và tạo nên sự hiểu lầm giữa khách hàng và người phát triển phần mềm.

3.1.5 Các phép toán về tập hợp

Phần lớn khả năng biểu diễn của tập hợp đến từ các phép toán tập hợp cơ bản: hợp, tương giao và phép lấy phần bù. Có một số phép toán khác nữa như phần bù tương đối, hiệu đối xứng và tích Đề-các. Các phép toán đó sẽ được định nghĩa dưới đây. Trong các định nghĩa này, ta bắt đầu với hai tập A và B, chứa trong tập vũ trụ chuyên đề U. Các phép toán của lôgic mệnh đề được dùng trong các định nghĩa này là $\vee$, $\wedge$, $\oplus$ và $\neg$.

Định nghĩa 3.1. Cho trước hai tập hợp A và B,

- hợp của chúng là tập $A \cup B = \{x \ : \ x \in A \vee x \in B\}$,
- tương giao của chúng là tập $A \cap B = \{x \ : \ x \in A \wedge x \in B\}$,
- phần bù của A là tập $A' = \{x \ : \ x \notin A\}$,
- phần bù tương đối của B đối với tập A là tập $A - B = \{x \ : \ x \in A \wedge x \notin B\}$,
- hiệu đối xứng của A và B là tập $A \oplus B = \{x : x \in A \oplus x \in B\}$.

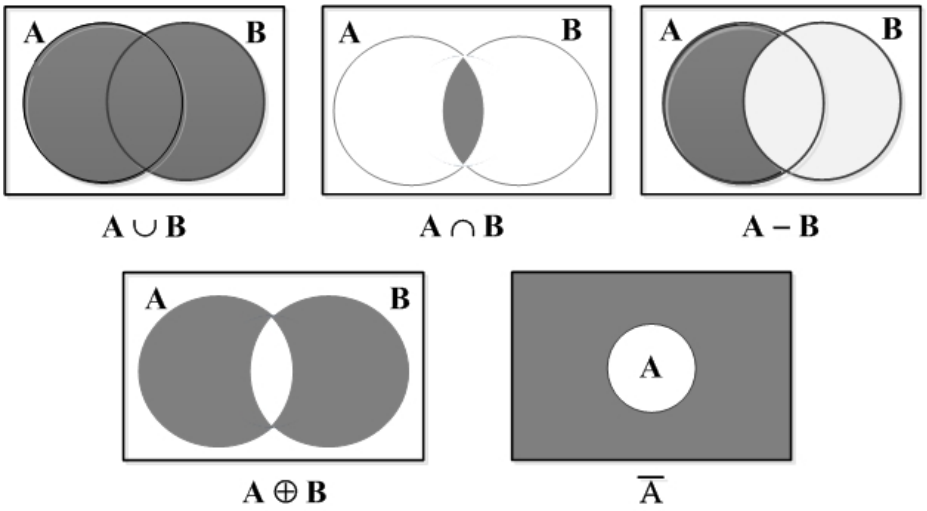
Các biểu đồ Venn cho các tập này được cho ở hình 3.2. Tính trực quan của các biểu đồ Venn rất hữu ích trong việc mô tả mối quan hệ giữa các ca kiểm thử và giữa các thánh phần cần kiểm thử.

Từ hình 3.2, chúng ta dễ dàng kết luận tính chất sau:

$$A \oplus B = (A \cup B) - (A \cap B).$$

Điều này đúng và chúng ta có thể được chứng minh bằng lôgic mệnh đề.

Biểu đồ Venn được dùng rộng rãi trong công nghệ phần mềm: cùng với các đồ thị có hướng, chúng là cơ sở của khái niệm sơ đồ trạng thái [Dav88] và là một trong các kỹ thuật đặc tả nghiêm túc được trợ giúp bởi máy tính. Sơ đồ trạng thái cũng là ký pháp



Hình 3.2: Các biểu đồ Venn cho các phép toán cơ sở.

cho việc biểu diễn điều khiển trong các ngôn ngữ mô hình hóa OMT (Object-Modeling Technique) và UML (Unified Modeling Language).

Tích Đề-các của hai tập hợp phức tạp hơn một chút và dựa trên khái niệm cặp được sắp mà gồm hai phần tử trong đó thứ tự giữa chúng là quan trọng. Tập gồm hai phần tử a và b, tức là $\{a, b\}$ là một cặp không được sắp. Cặp được sắp của hai phần tử a và b được ký hiệu là $\langle a, b\rangle$. Vai trò của thứ tự kéo theo, nếu $a \neq b$ thì

$$\begin{aligned} \{a, b\} &= \{b, a\} \\ \langle a, b\rangle &\neq \langle b, a\rangle \end{aligned}$$

Định nghĩa 3.2. Tích Đề-các của hai tập A và B là tập $A \times B = \{\langle x, y\rangle \ : \ x \in A \wedge y \in B\}$

Biểu đồ Venn không minh họa cho tích Đề-các. Cho $A = \{1, 2, 3\}$ and $B = \{w, x, y, z\}$, thì

$$\begin{aligned} A \times B = \{&\langle 1, w\rangle, \langle 1, x\rangle, \langle 1, y\rangle, \langle 1, z\rangle, \langle 2, w\rangle, \langle 2, x\rangle, \\ &\langle 2, y, \rangle, \langle 2, z\rangle, \langle 3, w\rangle, \langle 3, x\rangle, \langle 3, y\rangle, \langle 3, z\rangle\} \end{aligned}$$

Tích Đề-các liên quan trực tiếp với số học. Lực lượng của A là số các phần tử của nó và được ký kiệu bởi $|A|$ (hoặc $Card(A)$). Ta có $|A \times B| = |A| \times |B|$. Khi nói về kiểm thử chức năng trong chương 5, ta sẽ dùng tích Đề-các để biểu diễn miền các ca kiểm thử của chương trình có nhiều biến đầu vào hoặc nhiều biến đầu ra. Quy tắc nhân nói trên về lực lượng của tích Đề-các cho thấy loại chương trình như vậy sinh số ca kiểm thử rất lớn.

3.1.6 Quan hệ giữa các tập hợp

Các tập hợp có thể có một số quan hệ đặc biệt với nhau. Chúng ta sẽ tìm hiểu ba loại quan hệ cơ bản giữa hai tập hợp gồm: quan hệ tập con, quan hệ tập con thực sự và quan hệ bằng nhau.

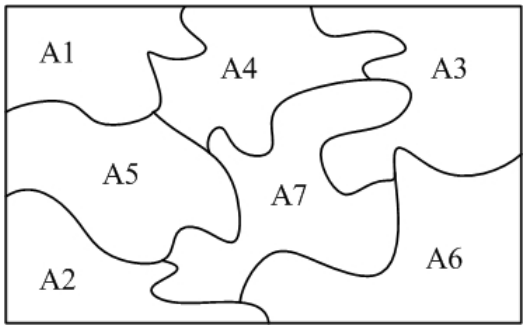
Định nghĩa 3.3.

- Tập hợp A là tập hợp con của tập hợp B, ký hiệu là $A \subseteq B$, nếu và chỉ nếu $a \in A \rightarrow a \in B$.
- A và B bằng nhau, ký hiệu là $A = B$, nếu và chỉ nếu $A \subseteq B$ và $B \subseteq A$.
- Tập hợp A là tập hợp con thực sự của tập hợp B, ký hiệu là $A \subsetneq B$, nếu và chỉ nếu $A \subseteq B$ và $B - A \neq \emptyset$.

Nói bằng lời thì A là tập hợp con của B nếu mỗi phần tử của A cũng là một phần tử của B, A và B là bằng nhau nếu mỗi tập này là tập con của tập kia, và A là tập hợp con thực sự của B nếu A là tập hợp con của B và có phần tử của B không ở trong A.

3.1.7 Phân hoạch tập hợp

Phân hoạch tập hợp là một khái niệm cực kỳ quan trọng trong kiểm thử phần mềm như sẽ thấy sau này trong việc chọn các ca kiểm thử một cách hiệu quả. Trong đời sống hàng ngày, chúng ta thường



Hình 3.3: Biểu đồ Venn của một phân hoạch.

thấy những thứ tương ứng với phân hoạch: tường phân hoạch trong cơ quan để phân chia không gian cơ quan thành các văn phòng cá nhân. Một phân hoạch hành chính là việc chia một nước thành các tỉnh. Nghĩa của từ phân hoạch trong ngôn ngữ tự nhiên là chia toàn bộ thành các mẩu sao cho các mẩu này là rời nhau và mọi thứ đều phải ở trong một mẩu nào đó, không chừa thứ nào. Điều này được định nghĩa hình thức như sau.

Định nghĩa 3.4. Cho một tập hợp B và một tập các tập con không rỗng của B là $\{A_1, A_2, \ldots, A_n\}$. Tập hợp $\{A_1, A_2, \ldots, A_n\}$ được gọi là một phân hoạch của B nếu các điều kiện sau được thỏa mãn:

1. $A_1 \cup A_2 \cup \ldots \cup A_n = B$, và
2. Với bất kỳ $i, j \in \{1, 2, \ldots, n\}, i \neq j$ ta đều có $A_i \cap A_j = \emptyset$.

Ta gọi mỗi tập con A_i là một phần tử của phân hoạch. Một bức họa về mê cung là một ví dụ tốt về phân hoạch. Biểu đồ Venn của phân hoạch cũng thường được vẽ như hình 3.3. Trong hình này, chúng ta dễ dàng nhận thấy các điều kiện về phân hoach như đã nêu trên đều thỏa mãn.

Đối với người kiểm thử, phân hoạch rất hữu ích vì hai tính chất nêu trong định nghĩa của phân hoạch sinh ra một đảm bảo quan

trọng: tính đầy đủ (chứa mọi điều) và tính không dư thừa (không bị lặp lại). Khi nghiên cứu về các phương pháp kiểm thử chức năng, ta thấy có hai điểm yếu của các phương pháp kiểm thử này là nguy cơ cho cả bỏ sót lẫn dư thừa: có cái gì đó chưa được kiểm thử trong khi lại có cái được kiểm thử lặp lại vài lần. Do đó, tìm được một phân hoạch thích hợp là một nhiệm vụ trung tâm và quan trọng nhất trong kiểm thử chức năng.

Trong chương trình tam giác, vũ trụ chuyên đề là tập tất cả các bộ ba số nguyên (tức là $INT \times INT \times INT$, trong đó INT là tập các số nguyên). Ta có thể phân hoạch vũ trụ này theo ba cách sau: *(i)* thành các bộ là số đo ba cạnh của tam giác và các bộ không là số đo ba cạnh của tam giác, *(ii)* thành các bộ là số đo ba cạnh của tam giác đều, cân, thường và không tam giác, và *(iii)* thành các bộ là số đo ba cạnh của tam giác đều, cân, thường, vuông và không tam giác.

Cách cuối cùng có vấn đề là các bộ tạo thành tam giác thường và các bộ tạo thành tam giác vuông là không rời nhau nên cách này không tạo thành phân hoạch.

3.1.8 Các đồng nhất thức về tập hợp

Một lớp quan trọng các đồng nhất thức tập hợp được tạo thành nhờ các tính chất của các phép toán về tập hợp được định nghĩa trước đây. Các đồng nhất thức này được liệt kê trong bảng sau:

Tên	**Đồng nhất thức**
Luật đồng nhất	$A \cup \emptyset = A$ $A \cap U = A$
Luật trội	$A \cup U = U$ $A \cap \emptyset = \emptyset$
Luật lũy đẳng	$A \cup A = A$ $A \cap A = A$

Luật phần bù	$(A')' = A$
Luật giao hoán	$A \cup B = B \cup A$ $A \cap B = B \cap A$
Luật kết hợp	$A \cup (B \cup C) = (A \cup B) \cup C$ $A \cap (B \cap C) = (A \cap B) \cap C$
Luật phân phối	$A \cup (B \cap C) = (A \cup B) \cap (A \cup C)$ $A \cap (B \cup C) = (A \cap B) \cup (A \cap C)$
Luật DeMorgan	$(A \cup B)' = A' \cap B'$ $(A \cap B)' = A' \cup B'$

3.2 Hàm

Khái niệm hàm đóng vai trò trung tâm trong phát triển và kiểm thử phần mềm. Nói một cách không hình thức, hàm liên kết các phần tử của các tập hợp. Trong chương trình NextDate, hàm được tính bởi chương trình liên kết một ngày được cho với ngày kế tiếp. Trong bài toán tam giác, hàm của ba số nguyên đầu vào là loại của tam giác tạo bởi các cạnh có độ dài ứng với dữ liệu đầu vào. Các hàm trong máy ATM phức tạp hơn nhiều và chính điều đó tạo thêm độ phức tạp cho việc kiểm thử.

Bất kỳ một chương trình có thể được coi là một hàm liên kết dữ liệu đầu ra với dữ liệu đầu vào. Trong phát biểu toán học của hàm, các dữ liệu đầu vào tạo thành miền xác định, và các dữ liệu đầu ra tạo thành miền giá trị.

Định nghĩa 3.5. Cho hai tập hợp A và B, một hàm f là một tập con của $A \times B$ sao cho bất kỳ $a_1, a_2 \in A$ và $b_1, b_2 \in B$, nếu $\langle a_1, b_1 \rangle \in f$ và $\langle a_2, b_2 \rangle \in f$ và $b_1 \neq b_2$ thì $a_1 \neq a_2$. Mối quan hệ này có nghĩa là mỗi phần tử của A được liên kết với không quá một phần tử của B. Vì lẽ đó, ta viết $f(a) = b$ thay cho $\langle a, b \rangle \in f$. Khi $f(a) = b$, ta nói f xác định trên a, hay $f(a)$ là xác định. Hàm f là toàn bộ nếu nó xác định trên mọi phần tử của A. Ngược lại, f

được gọi là hàm bộ phận. Sau này, nếu không nói gì thêm, chúng ta ngầm định rằng hàm f là hàm toàn bộ.

3.2.1 Miền xác định và miền giá trị

Trong định nghĩa trên, tập hợp A được gọi là miền xác định, và tập hợp B gọi là miền giá trị của hàm f. Vì dữ liệu đầu vào và dữ liệu đầu ra đã bao hàm một thứ tự tự nhiên giữa chúng, hàm f được tính bởi chương trình chính là tập các cặp được sắp trong đó phần tử thứ nhất của cặp là từ miền xác định, và phần tử thứ hai là từ miền giá trị. Để viết một hàm f với miền xác định là tập A và miền giá trị là tập B, ta dùng một trong hai cách:

$$f : A \to B$$
$$f \subseteq A \times B$$

Lưu ý rằng không có hạn chế gì đối với các tập A và B, chúng có thể là các tích Đề-các, và có thể có (và rất hay gặp) trường hợp $A = B$.

3.2.2 Các loại hàm

Một từ khác dành cho hàm là ánh xạ. Cho $f : A \to B$. Miền ảnh của ánh xạ f được định nghĩa là tập hợp:

$$f(A) = \{b \in B \ : \ b = f(a) \text{ với } a \in A \text{ nào đó}\}$$

Ta cũng gọi $f(A)$ là ảnh của tập A qua ánh xạ f.

Định nghĩa 3.6. Cho $f : A \to B$.

- f là hàm từ A **lên** B nếu và chỉ nếu $f(A) = B$.
- f là hàm từ A **vào** B nếu và chỉ nếu $f(A) \subsetneq B$ (ký hiệu $\subsetneq$ dùng để chỉ $f(A)$ là tập con thực sự của B).

- f là hàm **một-một** từ A sang B nếu và chỉ nếu với bất kỳ $a_1, a_2 \in A$ sao cho $f(a_1), f(a_2)$ xác định và $a_1 \neq a_2$, ta đều có $f(a_1) \neq f(a_2)$.

Nói bằng lời, nếu f là hàm từ A lên B thì mỗi phần tử của B đều liên kết với ít nhất một phần tử của A bởi f. Nếu f là hàm từ A vào B thì có ít nhất một phần tử của B không liên kết với một phần tử nào của A qua f. Hàm một-một đảm bảo tính duy nhất: các phần tử khác nhau không bao giờ được ánh xạ vào cùng một phần tử. Nếu một hàm không là một-một, nó được gọi là nhiều-một.

Quay lại ví dụ trước đây về kiểm thử, đối với chương trình NextDate, nếu ta lấy A, B và C là các tập hợp các ngày định nghĩa như sau:

$$\begin{aligned} A &= \{date \ : \ date \text{ là ngày giữa } 1/1/1812 \text{ và } 31/12/2012\} \\ B &= \{date \ : \ date \text{ là ngày giữa } 2/1/1812 \text{ và } 1/1/2013\} \\ C &= A \cup B \end{aligned}$$

Thế thì hàm NextDate: $A \rightarrow B$ là một-một và lên, và hàm NextDate: $A \rightarrow C$ là một-một và vào. Hàm của chương trình tam giác là hàm nhiều-một vì nó ánh xạ các bộ số $3, 4, 5$ và $12, 16, 20$ vào cùng một loại tam giác. Khi hàm f là một-một và lên, mỗi phần tử của miền giá trị được liên kết với đúng một phần tử của miền xác định và ngược lại như hàm NextDate: $A \rightarrow B$ trên đây. Khi đó ta có thể tìm hàm ngược từ miền giá trị của nó vào miền xác định của nó. Hàm ngược của NextDate: $A \rightarrow B$ chính là hàm YesterDate: $B \rightarrow A$.

Tất cả các khái niệm trên đây là quan trọng với người kiểm thử. Phân biệt giữa "vào" và "lên" được dùng trong kiểm thử hàm dựa trên miền giá trị và miền xác định, và các hàm một-một thường yêu cầu nhiều ca kiểm thử hơn các hàm nhiều-một khi chúng có cùng miền xác định.

3.2.3 Hàm hợp

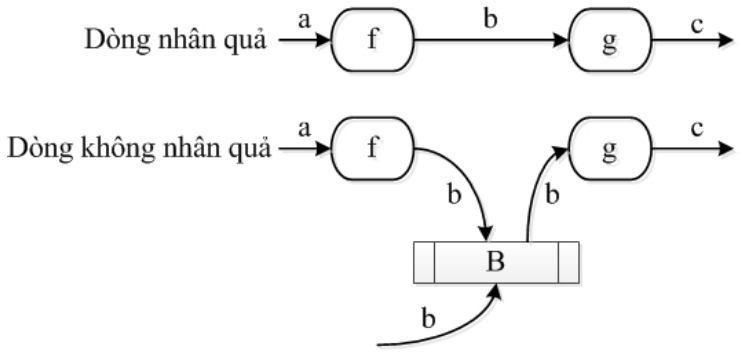
Giả sử ta có ba hàm sao cho miền giá trị của một hàm là miền xác định của hàm kế tiếp:

$$f : A \to B$$
$$g : B \to C$$
$$h : C \to D$$

Ta có thể kết hợp chúng theo cách sau đây. Với bất kỳ $a \in A$, gọi $b = f(a)$, $c = g(b)$, $d = h(c)$, $b \in B, c \in C, d \in D$. Kết hợp của f, g, h là hàm

$$\begin{aligned} h \circ g \circ f(a) &= h(g(f(a))) \\ &= h(g(b)) \\ &= h(c) \\ &= d \end{aligned}$$

Ta gọi $h \circ g \circ f$ là hàm hợp của f, g, h. Hàm hợp được dùng phổ biến trong phát triển phần mềm. Khái niệm này cũng như các thủ tục và các chương trình con. Chuỗi các việc kết hợp các hàm có thể gây khó khăn cho người kiểm thử, đặc biệt đối với các hàm vào, như được vẽ trong các sơ đồ dòng dữ liệu trong hình 3.4.



Hình 3.4: Ví dụ về dòng nhân quả và không nhân quả.

Trong dòng nhân quả, $g \circ f(a)$ (được định nghĩa là $g(b)$ và sinh ra c) là quá trình giống như lắp ráp các đường. Còn trong dòng

không nhân quả, khả năng có quá một nguồn tạo ra các giá trị b cho kho dữ liệu B có thể gây ra hai vấn đề cho người kiểm thử: việc nhiều nguồn tạo ra các giá trị b có thể làm mất tương thích giữa miền xác định và miền giá trị, và ngay cả khi không bị mất tương thích thì cũng có thể nảy sinh vấn đề về các bản khác nhau của b và hàm g có thể được áp dụng vào bản cũ của b.

Có một trường hợp đặc biệt của hàm hợp có ích với người kiểm thử. Nhắc lại rằng hàm một-một có hàm ngược duy nhất. Nếu $f : A \to B$ là một-một, gọi $f^{-1} : B \to A$ là hàm ngược của nó. Khi đó đối với $a \in A$ và $b \in B$, ta có $f^{-1} \circ f(a) = a$ và $f \circ f^{-1}(b) = b$. Khi đó khi cho f, f^{-1} đóng vai trò giúp người kiểm thử kiểm tra chéo. Chẳng hạn như các hàm NextDate: $A \to B$ và YesterDate: $B \to A$. Việc kiểm tra chéo được tiến hành trên các ca kiểm thử hàm đồng nhất.

3.3 Quan hệ

Hàm là trường hợp đặc biệt của quan hệ. Cả hai đều là tập con của tích Đề-các. Trong định nghĩa hàm, mỗi phần tử của miền xác định không được liên kết với quá một phần tử của miền giá trị, trong khi quan hệ là không bị ràng buộc bởi yêu cầu này. Vì thế, không phải tất cả các quan hệ đều là hàm. Xét mối quan hệ điều trị giữa bệnh nhân và thầy thuốc. Một bệnh nhân có thể được điều trị bởi nhiều thầy thuốc và một thầy thuốc có thể điều trị cho nhiều bệnh nhân. Rõ ràng, mối quan hệ này là phép tương ứng nhiều-nhiều.

3.3.1 Quan hệ giữa các tập hợp

Trước hết, ta cho một định nghĩa hình thức về quan hệ.

Định nghĩa 3.7. Cho hai tập A và B. Một quan hệ R từ A sang B là một tập con của tích Đề-các $A \times B$, tức là $R \subseteq A \times B$.

Thông thường ta viết aRb thay cho $\langle a, b\rangle \in R$. Ta quan tâm đến quan hệ trong tài liệu này vì khái niệm quan hệ là cốt lõi cho việc mô hình hoá dữ liệu và phân tích hướng đối tượng.

Quan hệ có các loại sau đây:

Định nghĩa 3.8. Cho A, B là hai tập hợp, $R \subseteq A \times B$ là một quan hệ. Loại của quan hệ R là một trong các trường hợp sau:

- **một-một** nếu và chỉ nếu R là hàm bộ phận một-một từ A sang B,
- **nhiều-một** nếu và chỉ nếu R là hàm bộ phận nhiều-một từ A sang B,
- **một-nhiều** nếu và chỉ nếu có ít nhất $a \in A$ sao cho a có quan hệ với hai phần tử khác nhau của B: có $b_1, b_2 \in B$, $b_1 \neq b_2$ và $\langle a, b_1\rangle \in R$, $\langle a, b_2\rangle \in R$,
- **nhiều-nhiều** nếu và chỉ nếu ít nhất một phần tử của A có quan hệ với hai phần tử khác nhau của B, và có ít nhất hai phần tử khác nhau của A có quan hệ với cùng một phần tử của B, tức là có $a, a_1, a_2 \in A$ và $b, b_1, b_2 \in B$ sao cho $a_1 \neq a_2$, $b_1 \neq b_2$ và $\langle a, b_1\rangle \in R$, $\langle a, b_2\rangle \in R$, $\langle a_1, b\rangle \in R$, $\langle a_2, b\rangle \in R$.

Tương tự như đối với các hàm, ta cũng có khái niệm bộ phận và toàn bộ cho quan hệ và gọi tính chất đó là **tham gia** của quan hệ.

Định nghĩa 3.9. Cho A, B là hai tập hợp, $R \subseteq A \times B$ là một quan hệ. Tham gia của quan hệ R là:

- **toàn bộ** nếu và chỉ nếu mỗi phần tử của A là ở trong cặp được sắp nào đó trong R,
- **bộ phận** nếu và chỉ nếu R không có tham gia toàn bộ,

- **lên** nếu và chỉ nếu mỗi phần tử của B đều ở trong cặp được sắp nào đó trong R,
- **vào** nếu và chỉ nếu R không có tham gia lên.

Việc phân biệt giữa khái niệm toàn bộ và bộ phận là tương tự với việc phân biệt giữa bắt buộc và tùy chọn.

Trên đây các quan hệ được định nghĩa giữa hai tập hợp và chúng dễ dàng được mở rộng thành quan hệ giữa nhiều tập hợp. Chẳng hạn, nếu A, B, C là ba tập hợp, thì bất kỳ tập con $R \subseteq A \times B \times C$ là một quan hệ. Định nghĩa loại và tham gia của quan hệ trong trường hợp này phức tạp và phong phú hơn và có thể được mở rộng từ các định nghĩa trên dùng các kết hợp khác nhau của tích Đề-các.

Các định nghĩa của quan hệ được dùng làm các tính chất của phần mềm cần kiểm thử, do đó người kiểm thử cần nắm vững chúng. Phân biệt của bắt buộc và tùy chọn là bản chất cho việc xử lý các ngoại lệ (exception handling) nên cũng không kém quan trọng đối với người kiểm thử.

3.3.2 Quan hệ trên một tập hợp

Có hai quan hệ quan trọng được định nghĩa trên một tập hợp. Đó là quan hệ thứ tự và quan hệ tương đương. Các quan hệ này được định nghĩa nhờ một số tính chất đặc biệt của quan hệ.

Cho A là một tập hợp, $R \subseteq A \times A$ là một quan hệ trên A. Có bốn tính chất quan trọng đối với các quan hệ trên A.

Định nghĩa 3.10. Quan hệ R là

- **phản xạ** nếu và chỉ nếu với bất kỳ $a \in A$, $\langle a, a\rangle \in R$,
- **đối xứng** nếu và chỉ nếu $\langle a, b\rangle \in R \rightarrow \langle b, a\rangle \in R$,
- **phản đối xứng** nếu và chỉ nếu $\langle a, b\rangle \in R \wedge \langle b, a\rangle \in R \rightarrow a = b$,

- **bắc cầu** nếu và chỉ nếu $\langle a, b\rangle \in R \wedge \langle b, c\rangle \in R \rightarrow \langle a, c\rangle \in R$.

Quan hệ gia đình là các ví dụ thú vị về các tính chất trên. Hãy nghĩ về các quan hệ sau đây và quyết định xem chúng có những tính chất nào: "anh em của", "anh chị em của", "tổ tiên của".

Định nghĩa 3.11. Quan hệ $R \subseteq A \times A$ là một quan hệ thứ tự nếu R là phản xạ, phản đối xứng và bắc cầu.

Quan hệ thứ tự có một ý niệm về hướng; các quan hệ thứ tự phổ biến như "già hơn", $\geq$, kéo theo về lôgic $\rightarrow$, "tổ tiên của". Trong các phần mềm, các quan hệ thứ tự xuất hiện khá phổ biến như: các kỹ thuật truy cập dữ liệu, mã băm, cấu trúc cây, mảng, ...

Tập mũ của tập đã cho là tập tất cả các tập con của nó. Tập mũ của A được ký hiệu là $P(A)$ hoặc 2^A. Quan hệ tập con $\subseteq$ là một quan hệ thứ tự trên $P(A)$ và điều này được kiểm chứng dễ dàng.

Định nghĩa 3.12. Quan hệ $R \subseteq A \times A$ là một quan hệ tương đương nếu R là phản xạ, đối xứng và bắc cầu.

Trong toán học ta gặp vô số các quan hệ tương đương: quan hệ bằng nhau hoặc tương đẳng đều là quan hệ tương đương. Có mối liên quan mật thiết giữa quan hệ tương đương trên A và phân hoạch của A. Cho trước một phân hoạch $A_1, A_2, \ldots, A_n$ của A. Ta định nghĩa một quan hệ trên A như sau: với $a, b \in A$, a và b có quan hệ nếu và chỉ nếu a, b nằm trong cùng một phần tử của phân hoạch. Quan hệ này là phản xạ vì bất kỳ một phần tử của A đều nằm trong một phần tử nào đó của phân hoạch, quan hệ này là đối xứng vì nếu a, b ở trong cùng một phần tử của phân hoạch thì b, a cũng cùng nằm trong phần tử đó, quan hệ này cũng bắc cầu vì a, b ở trong cùng một tập A_j và b, c nằm trong cùng một tập A_i thì $A_i = A_j$, và do đó a, c cùng nằm trong A_j. Quan hệ này được gọi là quan hệ tương đương sinh bởi phân hoạch $A_1, A_2, \ldots, A_n$ của A. Điều ngược lại cũng hoàn toàn tương tự. Nếu R là quan hệ tương

đương trên A, R sinh ra một phân hoạch của A như sau: với mỗi $a \in A$, gọi $[a] = \{b \ : \ b \in A \wedge aRb\}$. $[a]$ là một tập con của A và được gọi là lớp tương đương chứa a. Tập tất cả các lớp tương đương của các phần tử của A tạo thành một phân hoạch của A. Vì thế phân hoạch và quan hệ tương đương được đồng nhất với nhau, và đóng vai trò quan trọng cho người kiểm thử. Như đã giới thiệu, phân hoạch có tính đầy đủ và không dư thừa, cho phép người kiểm thử biết mình đã kiểm thử được đến đâu đối với phần mềm đang được kiểm thử. Tính hiệu quả là chọn một phần tử của một lớp tương đương để xác định ca kiểm thử và các phần tử còn lại của lớp có hành vi hoàn toàn tương tự.

3.4 Lôgic mệnh đề

Ta đã sử dụng các ký pháp của lôgic mệnh đề ở các mục trên. Quan hệ giữa lý thuyết tập hợp và lôgic mệnh đề cũng như quan hệ giữa gà con và trứng. Khó quyết định xem cái nào cần đưa vào trước. Cũng như tập hợp là khái niệm nguyên thủy không được định nghĩa, mệnh đề cũng là các phần tử nguyên thủy của lôgic mệnh đề. Một mệnh đề là một câu khẳng định mà giá trị của nó hoặc là đúng hoặc là sai. Ta gọi chúng là các giá trị chân lý của mệnh đề. Các mệnh đề cần phải không nhập nhằng: cho một mệnh đề, luôn có thể nói nó là đúng hay sai. Ví dụ, câu "toán học là khó" là nhập nhằng nên ta không xem nó như là một mệnh đề. Các mệnh đề được ký hiệu là p, q hoặc r. Lôgic mệnh đề có các phép toán, biểu thức và các đồng nhất thức như lý thuyết tập hợp.

3.4.1 Các phép toán lôgic

Các phép toán lôgic được định nghĩa theo hiệu quả của nó trên các giá trị chân lý của các mệnh đề mà phép toán được áp dụng. Điều này tương đối dễ dàng vì mỗi mệnh đề chỉ có thể có một trong hai

giá trị là T (đúng) và F (sai). Nếu các phép toán số học được định nghĩa theo cách này thì bảng giá trị sẽ lớn hơn nhiều. Các phép toán cơ sở của lôgic mệnh đề gồm $\neg$ (phép phủ định), $\wedge$ (phép hội, và), $\vee$ (phép tuyển, hoặc). Hai phép toán khác nữa cũng được dùng phổ biến là $\oplus$ (phép tuyển loại trừ) và $\rightarrow$ (phép kéo theo). Các phép toán này được định nghĩa bằng bảng giá trị chân lý dưới đây.

p	q	$\neg p$	$p \wedge q$	$p \vee q$	$p \oplus q$	$p \rightarrow q$	$p \leftrightarrow q$
T	T	F	T	T	F	T	T
T	F	F	F	T	T	F	F
F	T	T	F	T	T	T	F
F	F	T	F	F	F	T	T

Hội và tuyển là quá quen thuộc trong đời sống hàng ngày: hội là đúng khi cả hai thành phần là đúng, còn tuyển là đúng khi có ít nhất một thành phần là đúng. Phủ định làm việc như nghĩa mong đợi của nó. Tuyển loại trừ là đúng nếu có đúng một trong hai thành phần là đúng. Phép kéo theo, hay còn gọi là if-then, là khó nhất vì nó khó được chuyển đổi ngắn gọn thành ngôn ngữ tự nhiên. Nó có quan hệ mật thiết với việc suy diễn: nếu "tiền đề" thì "kết luận". $p \rightarrow q$ là đúng nếu hoặc q là đúng hoặc p là sai.

3.4.2 Biểu thức lôgic

Chúng ta dùng các phép toán lôgic để xây dựng các biểu thức lôgic như cách xây dựng các biểu thức số học. Ta cũng gọi các biểu thức lôgic là các công thức lôgic. Chúng ta cũng có thể đặc tả thứ tự mà các phép toán lôgic trong một biểu thức lôgic cần thực hiện với các quy ước thông thường về các dấu ngoặc và bằng cách sử dụng thứ tự ưu tiên của các phép toán là: phép phủ định, phép hội, và sau cùng là phép tuyển.

Cho một biểu thức lôgic, ta luôn có thể xây dựng bảng chân lý cho nó dùng thứ tự nói trên. Ví dụ, biểu thức $\neg(p \to q) \vee r \wedge p$ có bảng chân lý như sau:

p	q	r	$p \to q$	$\neg(p \to q)$	$r \wedge p$	$\neg(p \to q) \vee r \wedge p$
T	T	T	T	F	T	T
T	T	F	T	F	F	F
T	F	T	F	T	T	T
T	F	F	F	T	F	T
F	T	T	T	F	F	F
F	T	F	T	F	F	F
F	F	T	T	F	F	F
F	F	F	T	F	F	F

3.4.3 Tương đương lôgic

Khái niệm tương đương trong lôgic cũng tương tự như các khái niệm bằng nhau trong số học và đồng nhất của tập hợp. Có thể thấy các biểu thức $\neg(p \to q)$ và $p \oplus q$ có cùng bảng chân lý. Điều này nghĩa là bất kỳ giá trị nào được cho đối với p và q, giá trị chân lý của hai biểu thức này là như nhau. Các biểu thức như vậy được gọi là tương đương.

Định nghĩa 3.13. Hai biểu thức lôgic P và Q được gọi là tương đương, ký hiệu là $P \Leftrightarrow Q$, nếu và chỉ nếu chúng có cùng giá trị chân lý đối với bất kể giá trị chân lý nào của các mệnh đề cơ sở thành phần chứa trong các biểu thức này. Hay nói khác đi, chúng "có cùng" bảng chân lý.

Ta có thể thấy $p \leftrightarrow q$ là tương đương với $p \to q \wedge q \to p$.

Định nghĩa 3.14. Một biểu thức lôgic mà luôn có giá trị chân lý là đúng với bất kể giá trị chân lý nào của các mệnh đề cơ sở thành phần sẽ được gọi là một hằng đúng. Mệnh đề mà luôn luôn nhận giá trị sai được gọi là một mâu thuẫn (hằng sai).

Chúng ta có thể ký hiệu hằng đúng là T và mâu thuẫn/hằng sai là F. Bây giờ, chúng ta cũng có các luật sau đây giống như các luật đối với các tập hợp.

Luật	**Biểu thức**
Luật đồng nhất	$p \wedge T \Leftrightarrow T$ $p \vee F \Leftrightarrow p$
Luật trội	$p \vee T \Leftrightarrow T$ $p \wedge F \Leftrightarrow F$
Luật lũy đẳng	$p \vee p \Leftrightarrow p$ $p \wedge p \Leftrightarrow p$
Luật phần bù	$\neg(\neg p)) \Leftrightarrow p$
Luật giao hoán	$p \wedge q \Leftrightarrow q \wedge p$ $p \vee q \Leftrightarrow q \vee p$
Luật kết hợp	$(p \wedge q) \wedge r \Leftrightarrow p \wedge (q \wedge r)$ $(p \vee q) \vee r \Leftrightarrow p \vee (q \vee r)$
Luật phân phối	$p \wedge (q \vee r) \Leftrightarrow (p \wedge q) \vee (p \wedge r)$ $p \vee (q \wedge r) \Leftrightarrow (p \vee q) \wedge (p \vee r)$
Luật DeMorgan	$\neg(p \wedge q) \Leftrightarrow \neg p \vee \neg q$ $\neg(p \vee q) \Leftrightarrow \neg p \wedge \neg q$

3.5 Lý thuyết xác suất

Ta sẽ có hai cơ hội để dùng đến lý thuyết xác suất trong giáo trình này: một là khi liên quan đến xác suất của một đường đi ứng với các lệnh của chương trình được tiến hành, hai là khi liên quan đến một khái niệm công nghiệp là các hồ sơ thao tác. Do đó ta chỉ nêu những điều đủ để hiểu các khái niệm đó.

Các khái niệm nguyên thủy trong phép tính xác suất là xác suất của biến cố. Gọi S là không gian mẫu của số hữu hạn các kết quả có thể và đồng khả năng xuất hiện của một phép thử. Một tập con

hữu hạn của S sẽ được gọi là một biến cố. Khi đó, chúng ta có một định nghĩa đơn giản của xác suất là: xác suất của biến cố $E \subseteq S$, ký hiệu là $p(E)$, được tính là $p(E) = |E|/|S|$.

Là người kiểm thử phần mềm, ta sẽ đưa ra định nghĩa mà liên quan hơn cả đến công việc của mình. Ta dùng các khái niệm về tập hợp và mệnh đề ở phần trên. Ta gọi biến cố cơ sở là cái gì mà có thể xảy ra, và gọi tập tất cả các biến cố cơ sở là một vũ trụ chuyên đề. Ta sẽ sử dụng các mệnh đề cho các biến cố, các mệnh đề là về các phần tử của vũ trụ. Cho một vũ trụ chuyên đề U và một mệnh đề p về các phần tử của U. Tập chân lý của một mệnh đề được định nghĩa như sau:

Định nghĩa 3.15. Tập chân lý của một mệnh đề p, ký hiệu là $T(p)$, là tập tất cả các phần tử của U làm cho p đúng.

Vì mệnh đề là hoặc đúng hoặc sai, p chia U thành hai tập $T(p)$ và $(T(p))'$. Lưu ý rằng $(T(p))' = T(\neg p)$, và do đó $T(p) \cup T(\neg p) = U$. Các tập chân lý làm sáng tỏ mối tương ứng giữa mệnh đề, tập hợp và xác suất.

Định nghĩa 3.16. Xác suất để mệnh đề p là đúng, ký hiệu là $Pr(p)$, là $|T(p)|/|U|$.

Vì tập chân lý của một hằng đúng là U và của một mâu thuẫn là $\emptyset$, ta có xác suất của U (hằng đúng) là 1 và xác suất của $\emptyset$ (mâu thuẫn) là 0.

Xét ví dụ hàm NextDate, xét biến `month` và mệnh đề sau:

$$p(m) : m \text{ là tháng có 30 ngày}$$

Vũ trụ chuyên đề trong bài toán này là $U = \{1, 2, ..., 12\}$, và khi đó thì $T(p) = \{4, 6, 9, 11\}$. Vậy xác suất để một tháng cho trước có 30 ngày là:

$$Pr(p) = |T(p)|/|U| = 4/12 = 1/3$$

Vai trò của vũ trụ chuyên đề là quan trọng trong việc tính xác suất, và việc xác định vũ trụ chuyên đề (hay không gian xác suất) cho đúng với bài toán là một nghệ thuật và cần phải được thực hành nhiều. Giả sử ta muốn biết xác suất để một tháng là tháng hai. Câu trả lời ngay lập tức là 1/12. Bây giờ, giả sử ta muốn biết xác suất để một tháng có 29 ngày, khi đó câu trả lời không dễ như vậy. Ta cần một vũ trụ chuyên đề gồm cả năm nhuận lẫn không nhuận. Bằng cách sử dụng số học đồng dư, ta có thể sử dụng một chu kỳ bốn năm liên tiếp, chẳng hạn 1991, 1992, 1993 và 1994. Trong chu kỳ này, chỉ có một tháng là có 29 ngày và chu kỳ có tổng cộng 48 tháng. Khi đó xác suất cần tính là 1/48.

Sau đây là một số công thức quan trọng về tính xác suất:

1. $Pr(\neg p) = 1 - Pr(p)$
2. $Pr(p \vee q) = Pr(p) + Pr(q) - Pr(p \wedge q)$

3.6 Lý thuyết đồ thị

Phần này sẽ giới thiệu những kiến thức cơ sở về lý thuyết đồ thị cần thiết cho người kiểm thử trong việc phân tích chương trình và hiểu các kỹ thuật kiểm thử liên quan đến các sơ đồ biểu diễn thiết kế và đặc tả phần mềm.

Trong việc trình bày về đồ thị, ta cố gắng liên hệ các khái niệm với các thể hiện của chúng trong các ứng dụng kiểm thử sau này.

Có hai loại đồ thị cơ bản: đồ thị có hướng và đồ thị vô hướng hướng. Chúng ta sẽ bắt đầu với các đồ thị vô hướng.

3.6.1 Đồ thị

Một đồ thị là một cấu trúc toán học gồm một tập các đỉnh và một tập các cạnh giữa các đỉnh. Một mạng máy tính là một ví dụ về đồ thị. Đồ thị được định nghĩa hình thức như sau.

Định nghĩa 3.17. Đồ thị $G = (V, E)$ gồm một tập hữu hạn khác rỗng $V = \{v_1, v_2, \ldots, v_m\}$ các đỉnh, và một tập $E = \{e_1, e_2, \ldots, e_n\}$ các cạnh, trong đó mỗi cạnh $e_k = \{v_i, v_j\}$ với $v_i, v_j \in V$.

Lưu ý rằng $\{v_i, v_j\}$ là cặp không được sắp. Đôi khi ta cũng viết $e_k = (v_i, v_j)$ và cặp (v_i, v_j) hiểu là không được sắp tùy thuộc vào ngữ cảnh. Cạnh của đồ thị còn được gọi là cung và đỉnh cũng thường được gọi là nút. Trong việc biểu diễn đồ thị bằng hình vẽ, ta dùng các vòng tròn để biểu diễn đỉnh và dùng các đường nối hai đỉnh để biểu diễn các cạnh. Hình 3.5 là một ví dụ về biểu diễn bằng hình vẽ của một đồ thị. Trong hình vẽ này, các tập đỉnh và cạnh là

$$\begin{aligned} V &= \{v_1, v_2, v_3, v_4, v_5, v_6, v_7\} \\ E &= \{e_1, e_2, e_3, e_4, e_5\}, \end{aligned}$$

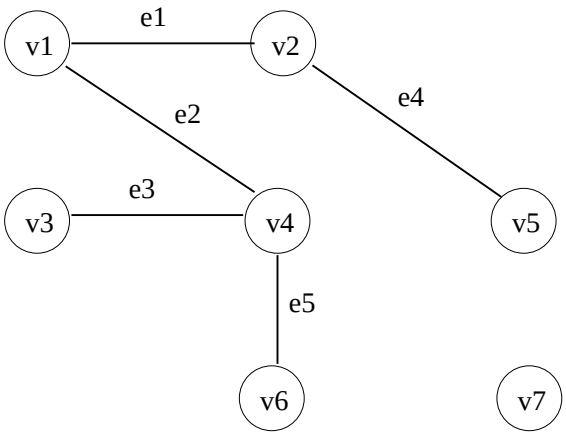
trong đó, $e_1 = (v_1, v_2)$, $e_2 = (v_1, v_4)$, $e_3 = (v_3, v_4)$, $e_4 = (v_2, v_5)$, và $e_5 = (v_4, v_6)$.

Để xác định một đồ thị, trước hết ta cần xác định tập các đỉnh của nó và sau đó là tập các cạnh giữa các cặp đỉnh. Ta có thể coi các lệnh của chương trình là các đỉnh và dùng các cạnh để biểu diễn các mối liên hệ giữa các lệnh như dòng điều khiển hoặc quan hệ định nghĩa/sử dụng (def/use) đối với biến nào đó của chương trình.

3.6.1.1 Bậc của đỉnh

Định nghĩa 3.18. Bậc của một đỉnh v của một đồ thị G là số các cạnh của G nối với v và được ký hiệu là $deg(v)$.

Ta cũng có thể coi bậc của một đỉnh đặc trưng cho tính "phổ biến" của đỉnh đó. Các nhà khoa học xã hội đã dùng đồ thị để mô tả sự tương tác trong xã hội, trong đó mỗi người là một đỉnh và các cạnh biểu diễn mối quan hệ "là bạn" hoặc "liên lạc với". Nếu ta vẽ một đồ thị với các đỉnh là các đối tượng và các cạnh là các thông



Hình 3.5: Một ví dụ về đồ thị.

điệp giữa các đối tượng, thì bậc của đỉnh cho biết quy mô của kiểm thử tích hợp thích hợp đối với đối tượng của đỉnh đó.

Bậc của các đỉnh trong đồ thị của hình 3.5 là:

$$\begin{aligned} deg(v_1) &= 2 \\ deg(v_2) &= 2 \\ deg(v_3) &= 1 \\ deg(v_4) &= 3 \\ deg(v_5) &= 1 \\ deg(v_6) &= 1 \\ deg(v_7) &= 0. \end{aligned}$$

3.6.1.2 Ma trận tới

Đồ thị không nhất thiết được biểu diễn bởi hình vẽ. Chúng có thể được biểu diễn bằng ma trận tới, mà dạng biểu diễn này rất quan trọng đối với người kiểm thử. Khi một đồ thị được gán cho một thể hiện nào đó, ma trận tới luôn cung cấp những thông tin có ích về thể hiện này.

Định nghĩa 3.19. Ma trận tới của đồ thị m đỉnh và n cạnh, $G = (V, E) = (\{v_1, \ldots, v_m\}, \{e_1, \ldots, e_n\})$, là một ma trận $m \times n$

trong đó phần tử ở hàng i và cột j là 1 nếu đỉnh v_i là một đầu mút của cạnh e_j, là 0 nếu ngược lại.

Ta có ma trận tới của đồ thị được mô tả trong hình 3.5 như sau:

	e_1	e_2	e_3	e_4	e_5
v_1	1	1	0	0	0
v_2	1	0	0	1	0
v_3	0	0	1	0	0
v_4	0	1	1	0	1
v_5	0	0	0	1	0
v_6	0	0	0	0	1
v_7	0	0	0	0	0

Ta có các quan sát sau đây đối với ma trận tới: thứ nhất là tổng các cột luôn bằng 2 vì mỗi cạnh chỉ có hai đầu mút, và được dùng làm điều kiện để kiểm tra tính tương thích, thứ hai là tổng của một hàng luôn bằng bậc của đỉnh tương ứng với hàng đó. Khi bậc của một đỉnh bằng 0, ta gọi đỉnh đó là đỉnh cô lập. Đỉnh cô lập có thể tương ứng với mã không đạt được hoặc đối tượng thừa.

3.6.1.3 Ma trận liền kề

Ma trận liền kề của đồ thị là một dạng biểu diễn khác bổ sung cho ma trận tới. Vì ma trận liền kề liên quan đến tính liên thông, khái niệm này là cơ sở cho nhiều khái niệm sau này về đồ thị.

Định nghĩa 3.20. Ma trận liền kề của đồ thị m đỉnh và n cạnh, $G = (V, E) = (\{v_1, \ldots, v_m\}, \{e_1, \ldots, e_n\})$, là một ma trận $m \times m$, trong đó phần tử ở hàng i và cột j là 1 nếu và chỉ nếu đỉnh v_i được nối với đỉnh v_j bởi một cạnh, là 0 nếu ngược lại.

Ma trận liền kề của một đồ thị luôn luôn là ma trận đối xứng, nghĩa là mỗi phần tử hàng i cột j luôn có giá trị bằng phần tử hàng

j cột i. Ngoài ra, cũng như đối với ma trận tới, tổng của một hàng luôn là bậc của đỉnh tương ứng đối với hàng đó. Ma trận liền kề của đồ thị trong hình 3.5 là:

	v_1	v_2	v_3	v_4	v_5	v_6	v_7
v_1	0	1	0	1	0	0	0
v_2	1	0	0	0	1	0	0
v_3	0	0	0	1	0	0	0
v_4	1	0	1	0	0	1	0
v_5	0	1	0	0	0	0	0
v_6	0	0	0	1	0	0	0
v_7	0	0	0	0	0	0	0

3.6.1.4 Đường đi trong đồ thị

Trong cách tiếp cận cấu trúc của việc kiểm thử, các loại của đường đi đóng vai trò trung tâm. Để giúp hiểu chính xác về cách tiếp cận đó, trong mục này ta đưa vào định nghĩa hình thức về đường đi.

Định nghĩa 3.21. Một đường đi giữa đỉnh v và đỉnh v' trong đồ thị G là một dãy liên tiếp các cạnh nối v và v', tức là dãy các cạnh $e_1 = (v, v_1)$, $e_2 = (v_1, v_2)$, $e_3 = (v_2, v_3), \ldots, e_{k+1} = (v_k, v')$.

Trong hình 3.5, e_1, e_4 là đường đi nối v_1 và v_5, e_5, e_2, e_1, e_4 là đường đi nối v_6 và v_5, và e_3, e_2, e_1 là đường đi giữa v_3 và v_2.

Các đường đi cũng có thể được sinh trực tiếp từ ma trận liền kề của đồ thị. Nếu A là ma trận liền kề của đồ thị G thì trong ma trận A^k, phần tử hàng i cột j bằng 1 nếu và chỉ nếu có đường đi gồm k cạnh giữa đỉnh v_i và đỉnh v_j. Ví dụ, trong tích của ma trận liền kề của đồ thị trong hình 3.5 với chính nó, phần tử ở vị trí $(1, 2)$ nhân với phần tử ở vị trí $(2, 5)$ tạo ra phần tử ở vị trí $(1, 5)$ của ma trận tích, và tương ứng với đường đi có hai cạnh giữa v_1 và v_5.

Đồ thị trong hình 3.5 không tổng quát lắm. Nó không có chu trình, tức là không chứa đường đi nối một đỉnh với chính nó. Nếu ta thêm vào cạnh (v_3, v_6) thì đồ thị này sẽ có chu trình.

3.6.1.5 Tính liên thông

Các đường đi cho phép ta nói về tính liên thông và cung cấp phương tiện đơn giản nhưng rất quan trọng đối với người kiểm thử.

Định nghĩa 3.22. Các đỉnh v_i và v_j là liên thông với nhau nếu chúng nằm trên cùng một đường đi.

Quan hệ "liên thông" là một quan hệ tương đương.

1. Quan hệ "liên thông" là phản xạ, vì mỗi đỉnh đều nằm trên một đường đi có độ dài 0 nối nó với chính nó.

2. Quan hệ "liên thông" là đối xứng, vì nếu các đỉnh v_i và v_j nằm trên cùng một đường, thì v_j và v_i cũng nằm trên cùng đường đó.

3. Quan hệ "liên thông" là bắc cầu vì nếu v_i và v_j nằm trên đường $e_1, e_2, \ldots, e_h$ và v_j và v_k nằm trên đường $e_r, \ldots, e_s$, thì v_i và v_k cùng nằm trên đường $e_1, e_2, \ldots, e_h, e_r, \ldots, e_s$.

Vì mỗi quan hệ tương đương sinh ra một phân hoạch, nên quan hệ liên thông sinh ra một phân hoạch của tập các đỉnh của đồ thị. Nhờ đó ta định nghĩa các thành phần của đồ thị như sau:

Định nghĩa 3.23. Một thành phần (liên thông) của đồ thị là một tập hợp cực đại các đỉnh liên thông với nhau.

Vậy là một thành phần là một lớp tương đương của quan hệ liên thông. Trong hình 3.5, ta có hai thành phần là $\{v_1, v_2, v_3, v_4, v_5, v_6\}$ và $\{v_7\}$.

3.6.1.6 Rút gọn đồ thị

Bây giờ ta đã có thể hình thức hóa một cơ chế đơn giản hóa quan trọng cho người kiểm thử.

Định nghĩa 3.24. Cho trước một đồ thị $G = (V, E)$, đồ thị rút gọn của G được tạo thành bằng cách thay mỗi thành phần bằng một đỉnh rút gọn.

Quá trình xây dựng đồ thị rút gọn của một đồ thị đã cho có thể được tiến hành bằng thuật toán. Ta dùng ma trận liền kề để xác định các đỉnh liên thông và sau đó dùng quan hệ tương đương để xác định các thành phần. Điều quan trọng của quá trình này là đồ thị rút gọn của đồ thị được cho là duy nhất (theo nghĩa tự nhiên).

Không có cạnh nối các đỉnh của đồ thị rút gọn vì các lý do sau:

1. Cạnh nối hai đỉnh chứ không phải hai tập đỉnh.
2. Hai tập đỉnh thành phần là độc lập theo nghĩa là không có cạnh trong đồ thị ban đầu nối một đỉnh của thành phần này với một đỉnh của thành phần kia (vì nếu có thì chúng phải bị kết hợp trong một thành phần!).

Các thành phần là độc lập nên chúng có thể được kiểm thử một cách riêng biệt.

3.6.1.7 Chỉ số chu trình

Một thuộc tính khác nữa của đồ thị mà có ảnh hưởng đến việc kiểm thử là độ phức tạp chu trình.

Định nghĩa 3.25. Chỉ số chu trình của đồ thị G được định nghĩa là $V(G) = e - n + p$, trong đó:

- e là số cạnh của G,
- n là số đỉnh của G, và

- p là số các thành phần của G.

$V(G)$ là số các miền khác nhau trong đồ thị. Một công thức của việc kiểm thử cấu trúc giả thiết khái niệm của các đường đi cơ sở trong chương trình, mà từ đó tất cả các đường đi khác có thể được dẫn ra, và chỉ ra rằng chỉ số chu trình của đồ thị chương trình chính là số các đường đi cơ sở này.

Chỉ số chu trình trong đồ thị ví dụ của ta là $V(G) = 5-7+2 = 0$. Khi ta dùng độ phức tạp chu trình trong kiểm thử, ta thường có trong tay các đồ thị liên thông mạnh. Chính điều đó tạo ra độ phức tạp chu trình lớn.

3.6.2 Đồ thị có hướng

Các đồ thị có hướng là một trường hợp đặc biệt của đồ thị được định nghĩa trước đây. Đối với các đồ thị loại này, ta đòi hỏi các cạnh có một hướng nào đó. Thay cho các cặp không được sắp xếp, ta sẽ dùng các cặp được sắp xếp $\langle v_i, v_j \rangle$ để biểu diễn cạnh có hướng từ đỉnh v_i đến đỉnh v_j.

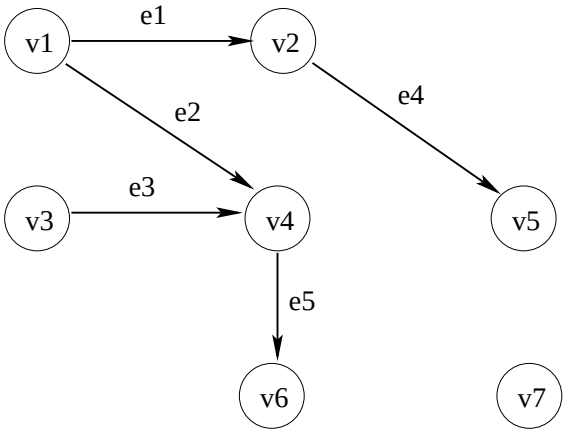
Định nghĩa 3.26. Một đồ thị có hướng (hay digraph) $D = (V, E)$ gồm một tập hữu hạn các đỉnh $V = \{v_1, v_2, \ldots, v_m\}$ và một tập $E = \{e_1, e_2, \ldots, e_n\}$ các cạnh, trong đó mỗi cạnh $e_k = \langle v_i, v_j \rangle$ là một cặp được sắp các đỉnh $v_i, v_j \in V$.

Cạnh có hướng $e_k = \langle v_i, v_j \rangle$ có đỉnh khởi hành (hay đỉnh đầu) là v_i và có đỉnh kết thúc (hay đỉnh cuối) là v_j. Rất nhiều khái niệm trong phần mềm được mô tả bởi cạnh có hướng như: hành vi tuần tự, ngôn ngữ lập trình lệnh, các sự kiện được sắp theo thứ tự thời gian, cặp định nghĩa/tham chiếu, thông điệp, lời gọi hàm hay thủ tục. Vậy tại sao ta vẫn cần đến đồ thị nguyên thủy vô hướng? Sự khác nhau giữa hai loại đồ thị này cũng tương tự như sự khác nhau giữa các ngôn ngữ lập trình lệnh và các ngôn ngữ lập trình mô tả. Trong các ngôn ngữ lập trình lệnh (như PORTRAN, COBOL,

PASCAL, C), thứ tự tuần tự của mã nguồn xác định thứ tự thời gian tiến hành của mã đích. Tình trạng là khác hẳn đối với các ngôn ngữ lập trình mô tả (như PROLOG). Trong loại mô tả, các nhà phát triển phần mềm thường dùng mô hình thực thể/quan hệ (E/R). Trong một mô hình E/R, ta dùng các đỉnh để biểu diễn thực thể, và dùng các cạnh để biểu diễn quan hệ. Khi đó ta được một đồ thị vô hướng.

Khi kiểm thử một chương trình viết trong ngôn ngữ mô tả, ta chỉ cần dùng đến các đồ thị vô hướng. Tuy nhiên, ta lại phải dùng đến các đồ thị có hướng khi kiểm thử các chương trình viết trong ngôn ngữ lập trình thủ tục (lệnh).

Các định nghĩa sau đây là song song với các định nghĩa đối với đồ thị nguyên thủy. Ta sẽ dùng ví dụ trong hình 3.6 để minh họa cho các định nghĩa này.



Hình 3.6: Một đồ thị có hướng.

Tập các đỉnh và tập các cạnh cũng giống như trong hình 3.5, chỉ khác là các cạnh được có hướng như sau: $e_1 = \langle v_1, v_2 \rangle, e_2 = \langle v_1, v_4 \rangle, e_3 = \langle v_3, v_4 \rangle, e_4 = \langle v_2, v_5 \rangle, e_5 = \langle v_4, v_6 \rangle$.

3.6.2.1 Bậc vào và bậc ra

Vì mỗi đỉnh trong đồ thị có hướng có thể là đỉnh đầu của một số cạnh, và cũng lại có thể là đỉnh cuối của một số cạnh, khái niệm bậc trong đồ thị nguyên thủy được làm mịn như sau:

Định nghĩa 3.27. (**Bậc vào và bậc ra trong đồ thị có hướng**.)

- Bậc vào của một đỉnh v trong đồ thị có hướng G là số các cạnh khác nhau trong G có đỉnh kết thúc là v. Bậc vào của v được ký hiệu là $indeg(v)$.
- Bậc ra của một đỉnh v trong đồ thị có hướng G là số các cạnh khác nhau trong G có đỉnh đầu là v. Bậc ra của v được ký hiệu là $outdeg(v)$.

Đối với đồ thị trong hình 3.6, bậc vào và bậc ra của các đỉnh như sau:

$$\begin{array}{llll} indeg(v_1) & = 0 & outdeg(v_1) & = 2 \\ indeg(v_2) & = 1 & outdeg(v_2) & = 1 \\ indeg(v_3) & = 0 & outdeg(v_3) & = 1 \\ indeg(v_4) & = 2 & outdeg(v_4) & = 1 \\ indeg(v_5) & = 1 & outdeg(v_5) & = 0 \\ indeg(v_6) & = 1 & outdeg(v_6) & = 0 \\ indeg(v_7) & = 0 & outdeg(v_7) & = 0 \end{array}$$

Từ định nghĩa, nếu coi một đồ thị có hướng là một đồ thị vô hướng sau khi bỏ qua hướng của các cạnh thì mỗi đỉnh có bậc như được định nghĩa cho đồ thị nguyên thủy. Rõ ràng: $deg(v) = indeg(v) + outdeg(v)$.

3.6.2.2 Loại của đỉnh

Trong đồ thị có hướng, ta có hai loại đặc biệt của các đỉnh là đỉnh vào (nguồn) và đỉnh ra (đích) được định nghĩa như sau:

Định nghĩa 3.28. Đỉnh nguồn là đỉnh có bậc vào bằng 0. Đỉnh đích là đỉnh có bậc ra bằng 0. Đỉnh không phải là đích và không phải là nguồn được gọi là đỉnh trung chuyển.

Các đỉnh nguồn và đích tạo thành ngoại biên của đồ thị. Trong phân tích có cấu trúc, đồ thị có hướng biểu diễn sơ đồ ngữ cảnh có các đỉnh nguồn và đỉnh đích tương ứng với các thực thể bên ngoài.

Trong ví dụ trên, các đỉnh v_1, v_3 và v_7 là các đỉnh nguồn, còn các đỉnh v_5, v_6 và v_7 là các đỉnh đích. Các đỉnh trung chuyển là v_2 và v_4. Các đỉnh vừa là nguồn vừa là đích là các đỉnh cô lập.

3.6.2.3 Ma trận liền kề của đồ thị có hướng

Việc đưa vào hướng thay đổi định nghĩa của ma trận liền kề và ma trận tới. Vì ma trận tới ít được dùng, ở đây ta chỉ đưa vào định nghĩa của ma trận liền kề.

Định nghĩa 3.29. Ma trận liền kề của một đồ thị có hướng $D = (V, E)$ với m đỉnh là ma trận $m \times m$ $A = (a_{ij})$, trong đó $a_{ij} = 1$ nếu và chỉ nếu có cạnh (có hướng) từ đỉnh i đến đỉnh j. Trái lại $a_{ij} = 0$.

Ta có thể dùng đồ thị có hướng để biểu diễn mối quan hệ họ hàng trong gia đình. Cũng như đối với trường hợp đồ thị nguyên thủy, các ô trong lũy thừa của ma trận liền kề chỉ sự tồn tại của đường đi có hướng giữa các đỉnh.

Khác với đồ thị nguyên thủy, ma trận liền kề của một đồ thị có hướng không nhất thiết là đối xứng. Trong ma trận liền kề, tổng của một hàng là bậc ra của đỉnh tương ứng với hàng đó, còn tổng của một cột là bậc vào của đỉnh tương ứng với cột đó. Ma trận liền kề của đồ thị trong hình 3.6 là:

	v_1	v_2	v_3	v_4	v_5	v_6	v_7
v_1	0	1	0	1	0	0	0
v_2	0	0	0	0	1	0	0
v_3	0	0	0	1	0	0	0
v_4	0	0	1	0	0	1	0
v_5	0	0	0	0	0	0	0
v_6	0	0	0	0	0	0	0
v_7	0	0	0	0	0	0	0

3.6.2.4 Đường đi và tựa đường đi

Hướng cho phép làm chính xác hơn ý nghĩa của đường đi nối các đỉnh. Nhờ hướng ta có thể biểu diễn đường đi một chiều, trong khi đồ thị vô hướng chỉ có thể biểu diễn đường đi hai chiều.

Định nghĩa 3.30. Đường đi có hướng là một dãy các cạnh sao cho bất kỳ hai cạnh liên tiếp e_i và e_j trong dãy đều thỏa mãn tính chất là đỉnh cuối của cạnh trước là đỉnh đầu của cạnh sau. Đỉnh đầu của cạnh đầu tiên trong dãy cũng được gọi là đỉnh đầu của đường đi, và đỉnh kết thúc của cạnh cuối cùng trong dãy cũng được gọi là đỉnh cuối của đường đi. Chu trình là một đường đi sao cho đỉnh đầu và đỉnh cuối của đường đi là như nhau. Tựa đường đi có hướng là một dãy các cạnh sao cho bất kỳ hai cạnh liên tiếp trong dãy đều có một đỉnh chung, trong đó có ít nhất một cặp cạnh liên tiếp có chung đỉnh đầu hoặc chung đỉnh cuối.

Các đường đi có hướng đôi khi được gọi là các xích hoặc các dây chuyền. Trong ví dụ trên đây về đồ thị có hướng:

- có đường đi từ v_1 đến v_6,
- có tựa đường đi giữa v_1 và v_3,
- có tựa đường đi giữa v_2 và v_4,
- có tựa đường đi giữa v_5 và v_6.

3.6.2.5 Ma trận đạt được

Tính đạt được một đỉnh cho trước từ một đỉnh khác cho trước bằng một đường đi có hướng đóng vai trò quan trọng trong nhiều ứng dụng của đồ thị. Tính đạt được này được biểu diễn thông qua "ma trận đạt được".

Định nghĩa 3.31. Ma trận đạt được của đồ thị có hướng $G = (V, E)$ với m đỉnh là ma trận $m \times m$ $R = (r_{ij})$, trong đó $r_{ij} = 1$ nếu có đường đi từ đỉnh v_i đến đỉnh v_j của G, trái lại $r_{ij} = 0$.

Ma trận đạt được của đồ thị có hướng $G = (V, E)$ có thể được tính từ ma trận liền kề A của G như sau:

$$R \;=\; I + A + A^2 + \ldots + R^k,$$

trong đó k là độ dài của đường đi dài nhất không chứa trong nó một chu trình nào, và I ký hiệu cho ma trận đơn vị. Ma trận đạt được của đồ thị có hướng trong hình 3.6 là:

	v_1	v_2	v_3	v_4	v_5	v_6	v_7
v_1	0	1	0	1	1	1	0
v_2	0	0	0	0	1	0	0
v_3	0	0	0	1	0	1	0
v_4	0	0	1	0	0	1	0
v_5	0	0	0	0	0	0	0
v_6	0	0	0	0	0	0	0
v_7	0	0	0	0	0	0	0

Ma trận đạt được trên cho chúng ta biết các đỉnh v_2, v_4, v_5, v_6 có thể đạt được từ đỉnh v_1, đỉnh v_5 có thể đạt được từ v_2, các đỉnh v_4, v_6 có thể đạt được từ v_3, các đỉnh v_3, v_6 có thể đạt được từ v_4, đỉnh v_7 không thể đạt được từ đỉnh nào, v.v.

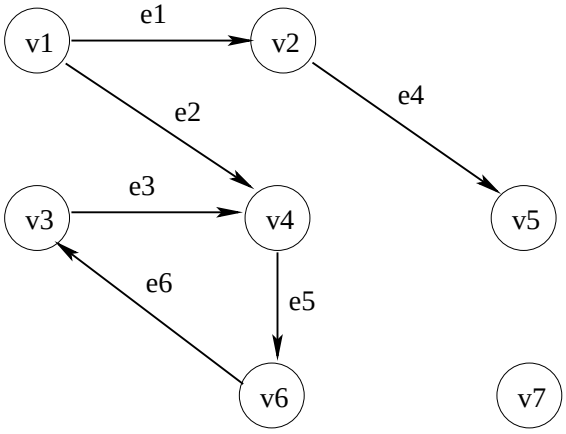
3.6.2.6 Tính N-liên thông

Tính liên thông trong đồ thị nguyên thủy được mở rộng cho đồ thị định hướng như sau.

Định nghĩa 3.32. Hai đỉnh v_i và v_j trong một đồ thị có hướng G là:

- 0-liên thông nếu không có đường đi từ v_i đến v_j,
- 1-liên thông nếu không có đường đi từ v_i đến v_j nhưng lại có một tựa đường đi từ v_i đến v_j,
- 2-liên thông nếu có đường đi từ v_i đến v_j,
- 3-liên thông nếu có đường đi từ v_i đến v_j và có đường đi từ v_j đến v_i.

Không có bậc liên thông cao hơn 3.



Hình 3.7: Đồ thị có hướng với chu trình.

Trong hình 3.7, ta có:

- v_1 và v_7 là 0-liên thông,
- v_2 và v_6 là 1-liên thông,

- v_1 và v_6 là 2-liên thông, và
- v_3 và v_6 là 3-liên thông.

3.6.2.7 Thành phần liên thông mạnh

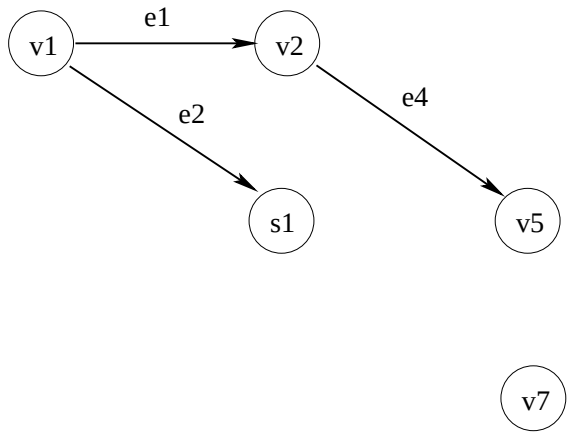
Từ định nghĩa của tính n-liên thông trên đây, ta có quan hệ 1-liên thông và quan hệ 3-liên thông là quan hệ tương đương. Tính 1-liên thông cũng được gọi là liên thông yếu và sinh ra thành phần liên thông yếu. Tính 3-liên thông có đặc thù thú vị cho các đồ thị định hướng. Các lớp tương đương của quan hệ 3-liên thông sẽ được gọi là các thành phần liên thông mạnh.

Định nghĩa 3.33. Thành phần liên thông mạnh của một đồ thị có hướng là tập hợp cực đại các đỉnh 3-liên thông của đồ thị đó.

Trong ví dụ ở hình 3.7, $\{v_3, v_4, v_6\}$ và $\{v_7\}$ là các thành phần liên thông mạnh, $\{v_2\}$ và $\{v_1\}$ cũng là các thành phần liên thông mạnh.

Các thành phần liên thông mạnh cho phép ta đơn giản hóa đồ thị bằng cách "co" mỗi trong chúng thành một đỉnh. Đồ thị nhận được không có chu trình nữa và được gọi là đồ thị cô đọng của đồ thị ban đầu. Các đồ thị có hướng không có chu trình thường được gọi là một DAG (Directed Acyclic Graph). Các đồ thị cô đọng đều là các DAG. Hình 3.8 là đồ thị cô đọng của đồ thị trong hình 3.7.

Trong kiểm thử cấu trúc, người ta đã chỉ ra rằng các đồ thị của chương trình có thể có số rất lớn các đường đi và vì thế việc kiểm thử vét cạn là không thể. Nguyên nhân của việc sở hữu số rất lớn các đường đi là các chu trình trong đồ thị. Với việc đơn giản hóa bằng đồ thị cô đọng, ta có thể tránh được các chu trình và đơn giản hóa việc kiểm thử.



Hình 3.8: Đồ thị cô đọng của đồ thị trong hình 3.7.

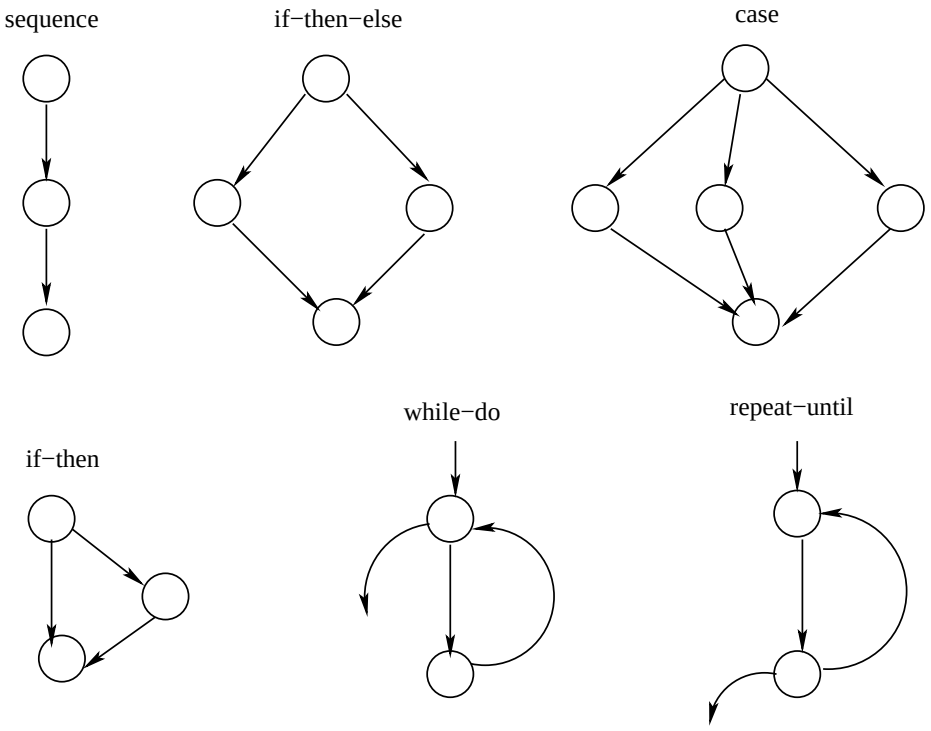
3.6.3 Các loại đồ thị dùng cho kiểm thử

Chúng ta kết thúc chương này bằng việc làm quen với một số loại đồ thị đặc biệt được sử dụng rộng rãi trong kiểm thử. Đó là các đồ thị chương trình (sẽ được sử dụng trong các chương 6 và 7), các đồ thị máy hữu hạn trạng thái và các mạng Petri. Các loại đồ thị này là các công cụ tốt nhất để mô tả hành vi mức hệ thống của các sản phẩm phần mềm cần kiểm thử mặc dù chúng cũng được dùng để mô tả các mức kiểm thử thấp hơn (kiểm thử tích hợp và kiểm thử đơn vị).

Định nghĩa 3.34. Cho một chương trình viết trong ngôn ngữ thủ tục (tức là ngôn ngữ lệnh). Đồ thị chương trình của chương trình này là một đồ thị định hướng, trong đó các đỉnh là các lệnh của chương trình, và các cạnh biểu diễn dòng điều khiển (có cạnh đi từ đỉnh i đến đỉnh j nếu lệnh tương ứng của đỉnh j có thể được tiến hành ngay sau lệnh tương ứng với đỉnh i).

Định nghĩa trên đây cũng được cải tiến để chi tiết hóa các lệnh theo cách là mỗi đỉnh của đồ thị chương trình có thể biểu diễn toàn bộ một câu lệnh hoặc các phần của một câu lệnh hoặc một tập các

câu lệnh tuần tự (lệnh ghép), và các cạnh thì vẫn biểu diễn dòng điều khiển: có cạnh đi từ đỉnh i đến đỉnh j nếu lệnh hoặc phần lệnh tương ứng của đỉnh j có thể được tiến hành ngay sau lệnh hoặc phần lệnh tương ứng với đỉnh i.



Hình 3.9: Đồ thị của các cấu trúc của lập trình có cấu trúc.

Nhờ đồ thị chương trình mà ta có thể mô tả chính xác các khía cạnh kiểm thử của chương trình. Các cấu trúc của lập trình có cấu trúc (tuần tự, rẽ nhánh và lặp) cũng có các biểu diễn đồ thị rất sáng sủa như được cho trong hình 3.9.

Khi các cấu trúc này được dùng trong một chương trình có cấu trúc, các đồ thị tương ứng của cấu trúc hoặc được lồng nhau hoặc được ghép với nhau. Do các tiêu chuẩn "một lối vào và một lối ra" của các cấu trúc, đồ thị chương trình của chương trình có cấu trúc có duy nhất một đỉnh nguồn và duy nhất một đỉnh đích. Chương

trình không có cấu trúc, tức là chương trình có lệnh "GO TO" sẽ tạo ra các đồ thị chương trình rất phức tạp và khó phân tích. Khi một chương trình được tiến hành, dãy tất cả các lệnh của một tiến hành tạo thành một đường đi trong đồ thị chương trình. Chu trình và các đỉnh rẽ nhánh đóng góp vào việc tăng trưởng của số đường đi trong đồ thị chương trình và do đó tăng nhu cầu của việc kiểm thử chương trình/phần mềm.

Một vấn đề mà đồ thị chương trình gặp phải là biểu diễn các lệnh không tiến hành như lệnh mô tả dữ liệu và các chú giải thế nào. Cách đơn giản nhất là bỏ qua các lệnh này. Một vấn đề nữa là sự phân biệt các đường đi có thể về tô pô trong đồ thị và đường đi không khả thi. Vấn đề này sẽ được bàn trong phần sau.

3.6.3.1 Máy hữu hạn trạng thái

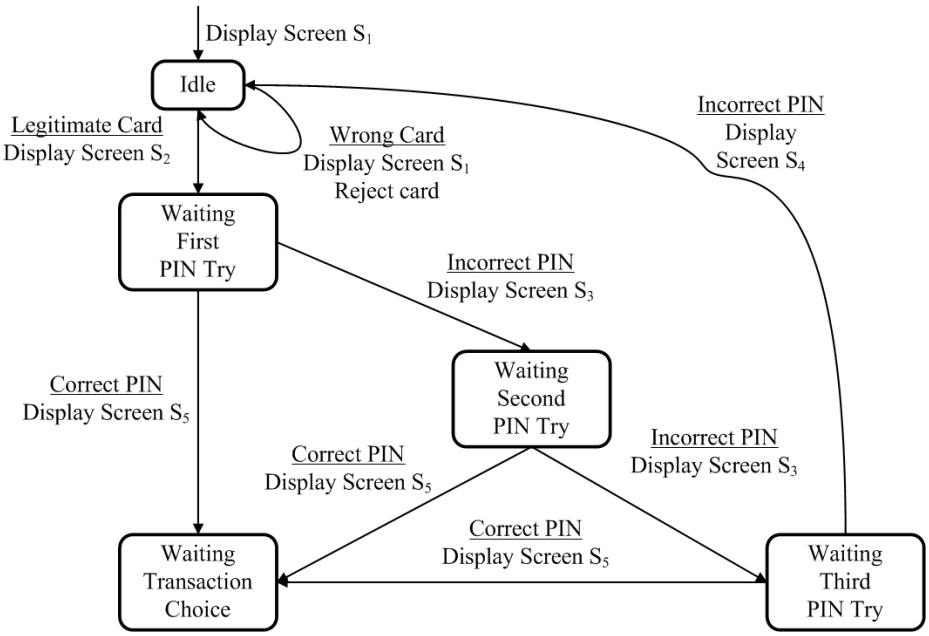
Máy hữu hạn trạng thái là một khái niệm chuẩn để đặc tả yêu cầu. Các mở rộng thời gian thực của việc phân tích có cấu trúc cũng sử dụng một dạng của máy hữu hạn trạng thái. Hầu như tất cả các dạng của phân tích định hướng đối tượng đều cần đến máy hữu hạn trạng thái. Một máy trạng thái hữu hạn là một đồ thị có hướng trong đó trạng thái là các đỉnh, các chuyển trạng thái là các cạnh. Trạng thái đầu là đỉnh nguồn. Hầu hết máy trạng thái có thêm thông tin bổ sung cho các cạnh tương ứng với nguyên nhân và kết quả hành động của chuyển tương ứng. Sau đây là một định nghĩa hình thức của máy hữu hạn trạng thái.

Định nghĩa 3.35. Máy hữu hạn trạng thái (viết tắt là FSM - Finite State Machine) là một đồ thị có hướng (S, T, Ev, Act), trong đó S là tập hữu hạn các trạng thái và là đỉnh của đồ thị, T là tập các chuyển trạng thái và là các cạnh của đồ thị, Ev và Act là các tập các sự kiện và hành động liên kết với các cạnh trong T.

Hình 3.10 là một máy hữu hạn trạng thái cho chức năng kiểm tra mật khẩu của máy rút tiền tự động đơn giản. Máy gồm năm trạng

thái như trong hình (Idle, Awaiting, ...) và tám phép chuyển trạng thái. Nhãn của các chuyển được làm theo quy ước sau: "numerator" là sự kiện gây ra việc chuyển trạng thái, còn "denumerator" là hành động liên kết với việc chuyển trạng thái (cạnh). Các sự kiện là bắt buộc phải có, còn các hành động là tùy chọn. Các máy hữu hạn trạng thái là các cách đơn giản để biểu diễn tình trạng trong đó có nhiều sự kiện có thể xảy và sự xuất hiện của chúng có các hệ quả khác nhau. Trong ví dụ trên đây, khách hàng có ba cơ hội để đưa vào mật khẩu đúng. Nếu mật khẩu đưa vào đúng ngay ở lần đầu tiên, hành động kết quả là hiển thị màn hình 5 (để mời khách hàng chọn loại giao dịch). Nếu vào mật khẩu không đúng, máy chuyển đến một trạng thái khác, một trong các trạng thái đó là chờ cho lần vào mật khẩu lần hai. Lưu ý rằng cùng các sự kiện và hành động xuất hiện trên các chuyển từ trạng thái chờ thử mật khẩu lần hai. Đây chính là cách để máy hữu hạn trạng thái nhớ lịch sử quá khứ.

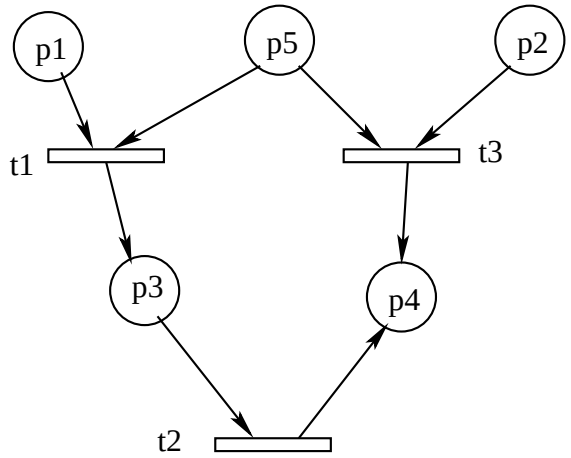
Các máy trạng thái có thể tiến hành được nhưng cần có một số quy ước. Trước hết là khái niệm trạng thái kích hoạt. Tại mỗi thời điểm, một hệ thống luôn ở một trạng thái nào đó. Khi hệ thống được mô hình bởi một máy hữu hạn trạng thái, trạng thái kích hoạt là trạng thái mà hiện tại hệ thống đang ở. Quy ước khác nữa là các máy hữu hạn trạng thái có thể có trạng thái đầu tiên, mà chính là trạng thái mà khi bắt đầu hệ thống đi vào. Ví dụ trạng thái "nhàn rỗi" (Idle) là trạng thái đầu của ví dụ trên, được chỉ vào bởi mũi tên không có đỉnh đầu. Vì các chuyển trạng thái xảy ra tức thời, tại mỗi thời điểm chỉ có một trạng thái được kích hoạt. Để tiến hành một máy trạng thái, ta xuất phát từ trạng thái đầu tiên, và cung cấp dãy các sự kiện gây ra việc chuyển trạng thái của các chuyển. Khi một sự kiện xuất hiện, cái chuyển tương ứng sẽ thay đổi trạng thái được kích hoạt, sau đó, một sự kiện mới xuất hiện. Theo cách đó, dãy các sự kiện sẽ chọn một đường đi qua một dãy các trạng thái và các chuyển của máy.



Hình 3.10: FSM cho một phần của máy rút tiền tự động đơn giản.

3.6.3.2 Mạng Petri

Mạng Petri là chủ đề của luận án tiến sĩ của nhà vật lý học Carl Adam Petri năm 1960. Ngày nay, mạng Petri đã được chấp nhận là một mô hình cho các thể thức và các ứng dụng khác có liên quan đến tính tương tranh và việc xử lý phân tán. Các mạng Petri là trường hợp đặc biệt của đồ thị có hướng có tên là đồ thị lưỡng phân có hướng. Một đồ thị lưỡng phân có hướng là một đồ thị có hướng trong đó tập đỉnh V được phân hoạch thành hai tập hợp V_1 và V_2 sao cho mỗi cạnh có đỉnh đầu ở tập này thì phải có đỉnh cuối ở tập kia. Trong mạng Petri, một trong hai tập đỉnh, chẳng hạn tập đỉnh V_1 được coi là tập các "vị trí", và tập đỉnh còn lại V_2 được coi là tập các chuyển. Thông thường ta đặt tên tập các vị trí là P và tập các chuyển là T. Các vị trí là đầu vào và đầu ra của các chuyển.



Hình 3.11: Một mạng Petri.

Quan hệ đầu vào và đầu ra là các hàm và được ký hiệu là In và Out, như trong định nghĩa dưới đây.

Định nghĩa 3.36. Một mạng Petri là một đồ thị lưỡng phân có hướng (P, T, In, Out), trong đó P và T là hai tập hữu hạn rời nhau của các đỉnh, In và Out là tập các cạnh, $In \subseteq P \times T$ và $Out \subseteq T \times P$.

Hình 3.11 minh họa một mạng Petri, trong đó:

$$\begin{aligned} P &= \{p1, p2, p3, p4, p5\} \\ T &= \{t1, t2, t3\} \\ In &= \{\langle p1, t1\rangle, \langle p5, t1\rangle, \langle p5, t3\rangle, \langle p2, t3\rangle, \langle p3, t2\rangle\} \\ Out &= \{\langle t1, p3\rangle, \langle t2, p4\rangle, \langle t3, t4\rangle\} \end{aligned}$$

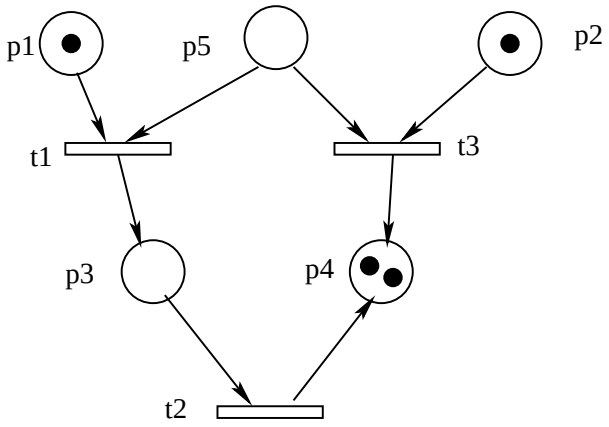
Mạng Petri có thể được tiến hành theo cách thú vị hơn là máy trạng thái hữu hạn. Một mạng Petri, để được tiến hành, trước hết nó cần được "đánh dấu".

Định nghĩa 3.37 (Mạng Petri được đánh dấu)**.** Mạng Petri được đánh dấu là bộ năm (P, T, In, Out, M), trong đó (P, T, In, Out) là một mạng Petri và M là một tập các ánh xạ từ P vào tập các số tự nhiên $\mathbb{N}$.

Tập M như được định nghĩa trên cũng được gọi là tập đánh dấu của mạng Petri (P, T, In, Out). Khi các phần tử của P được đánh số từ 1 đến n ($|P| = n$), các phần tử của M sẽ là các bộ gồm n phần tử. Với mạng Petri trong hình 3.11, các phần tử của M có dạng $\langle n1, n2, n3, n4, n5\rangle$, trong đó các n là các số nguyên liên kết với các vị trí tương ứng. Số liên kết với một vị trí p được biểu diễn bởi số các dấu đen trong vị trí đó. Trong các ứng dụng khác nhau, các dấu sẽ có các thể hiện khác nhau.

Bộ đánh dấu của mạng Petri trong hình 3.12 là $\langle 1, 1, 0, 2, 0\rangle$. Khái niệm đánh dấu giúp ta có thể đưa vào hai khái niệm cơ bản sau:

Định nghĩa 3.38. Cho một mạng Petri và một đánh dấu của nó. Một chuyển t trong mạng Petri được gọi là được kích hoạt nếu có ít nhất một dấu đen trong mỗi vị trí vào của nó, tức là nếu $\langle p, t\rangle \in In$ thì trong p phải có ít nhất một dấu đen.



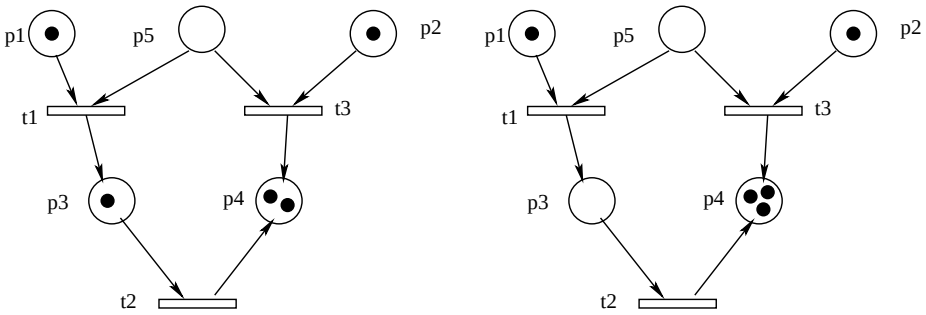
Hình 3.12: Mạng Petri được đánh dấu.

Mạng trong hình 3.12 không có chuyển nào được kích hoạt. Nếu ta thêm một dấu đen vào vị trí $p3$ thì chuyển $t2$ sẽ được kích hoạt.

Định nghĩa 3.39. Khi một chuyển trong mạng Petri được kích hoạt, nó có thể cháy. Khi một chuyển t được kích hoạt cháy, một

dấu sẽ được khử đi từ mỗi trong các vị trí vào của t, và một dấu sẽ được đặt thêm vào mỗi trong các vị trí ra của t, tức là các vị trí p sao cho $\langle t, p\rangle \in Out$.

Hình 3.13 minh họa việc cháy của $t2$ trong mạng bên trái, và kết quả của việc cháy là mạng bên phải.



Hình 3.13: Trước và sau khi cháy một chuyển.

Tập các đánh dấu của mạng trong hình 3.13 gồm hai bộ, bộ đầu trong đó $t2$ được kích hoạt, còn bộ sau nhận được sau khi $t2$ cháy:

$$M \quad = \quad \{\langle 1, 1, 1, 2, 0\rangle, \langle 1, 1, 0, 3, 0\rangle\}.$$

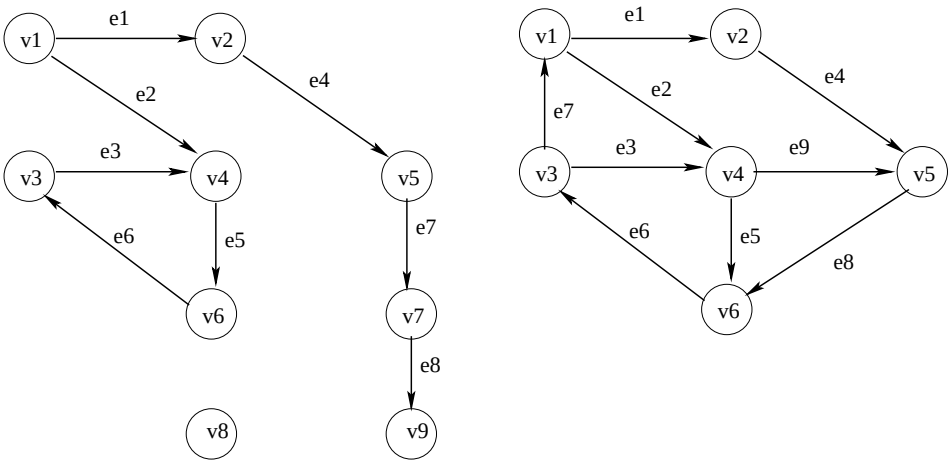
Các dấu có thể được tạo ra và hủy đi khi cháy các chuyển được kích hoạt. Trong một số điều kiện đặc biệt, tổng số các dấu không bao giờ thay đổi. Các mạng như thế được gọi là các mạng bảo thủ. Việc đánh dấu cho phép ta tiến hành các mạng Petri như là các máy hữu hạn trạng thái.

Về bản chất thì máy hữu hạn trạng thái là các trường hợp đặc biệt của mạng Petri. Giả sử ta có một đánh dấu khác cho mạng trong hình 3.11 trong đó các vị trí $p1$, $p2$ và $p5$ là được đánh dấu. Khi đó thì cả $t1$ và $t3$ đều được kích hoạt. Nếu ta chọn $t1$ để cháy thì dấu đen trong $p5$ sẽ bị khử và $t3$ sẽ không được kích hoạt nữa. Tương tự, nếu ta chọn $t3$, chúng ta sẽ hóa giải tính bị kích hoạt của $t1$. Các chuyển $t1$ và $t3$ khi đó sẽ được gọi là ở trong tình trạng

xung đột tương ứng với vị trí $p5$. Mạng Petri xung đột biểu diễn một dạng khá hay về sự tương tác của các chuyển mà ta sẽ bàn đến sau này.

3.7 Bài tập

1. Biểu diễn $A \oplus B$ bằng lời.
2. Biểu diễn $(A \cup B) - (A \cap B)$ bằng lời
3. Lý giải tại sao $A \oplus B = (A \cup B) - (A \cap B)$
4. Đẳng thức $A \oplus B = (A - B) \cup (B - A)$ có đúng không, tại sao?
5. Quan hệ tham gia cùng một môn học nào đó có là quan hệ tương đương trong lớp không? Tại sao. Khi nào quan hệ đó là quan hệ tương đương?
6. Hãy đề xuất một định nghĩa về độ dài đường đi trong đồ thị.
7. Những chu trình nào sẽ được tạo ra nếu một cạnh được thêm vào giữa các đỉnh v_5 và v_6 trong đồ thị ở hình 3.11.
8. Hãy chứng minh rằng quan hệ 3-liên thông là quan hệ tương đương trên tập các đỉnh của đồ thị có hướng.
9. Tính độ phức tạp (chỉ số) chu trình của mỗi cấu trúc trong lập trình có cấu trúc trong hình 3.9.
10. Các đồ thị có hướng như hình 3.14 thu được bằng việc thêm đỉnh và cạnh vào đồ thị có hướng ở hình 3.7. Hãy tính độ phức tạp chu trình của mỗi trong các đồ thị mới này và giải thích các thay đổi đó ảnh hưởng đến độ phức tạp như thế nào.



Hình 3.14: Các đồ thị cho bài tập 10.

11. Giả sử ta xây dựng một đồ thị trong đó các đỉnh là người, và cạnh tương ứng với một dạng nào đó của mối tương tác xã hội, chẳng hạn "nói với" hoặc "bạn bè với". Tìm các khái niệm trong lý thuyết đồ thị tương ứng với các khái niệm xã hội như tính phổ biến, hội.

Chương 4

Khảo sát đặc tả và mã nguồn

Chương này giới thiệu các kỹ thuật khảo sát đặc tả và mã nguồn. Mã nguồn được phát triển dựa trên đặc tả và vì thế việc khảo sát đặc tả cần được tiến hành trước khi phát triển mã nguồn để tránh các rủi ro về các lỗi có thể có trong đặc tả về sản phẩm phần mềm. Vì đặc tả không thể thực thi được trên máy nên chúng ta chỉ có thể phát hiện các lỗi bằng các kỹ thuật khảo sát đặc tả. Tuy nhiên, mã nguồn thì có thể thực thi được và có thể được kiểm thử thông qua việc chạy trên máy. Vì vậy, liệu chúng ta có cần phải khảo sát mã nguồn trước không? Câu trả lời là rất cần vì khảo sát giúp ta phát hiện các lỗi sớm, các lỗi về lôgic, các lỗi về cấu trúc và giúp đề xuất các ca kiểm thử hiệu quả hơn. Khảo sát mã nguồn và không tiến hành phần mềm để phát hiện lỗi trong quá trình phát triển phần mềm được gọi là kiểm thử hộp trắng tĩnh (static white-box testing). Đây là một kỹ thuật kiểm thử bổ sung vào các kỹ thuật kiểm thử khác để đảm bảo chất lượng phần mềm. Còn khảo sát đặc tả được liệt vào phạm trù của kiểm thử hộp đen tĩnh (static black-box testing) vì việc này thường được tiến hành khi chưa có mã nguồn của sản phẩm phần mềm.

4.1 Khảo sát đặc tả

Đặc tả phần mềm là một tài liệu quan trọng, mô tả các chức năng mà phần mềm cần có cũng như các ràng buộc mà phần mềm cần thỏa mãn. Tài liệu này được tạo ra từ nhiều nguồn khác nhau như thông qua các nghiên cứu về nhu cầu sử dụng của người dùng, về các biểu mẫu, về thị trường, v.v. Việc tài liệu này được tạo ra thế nào và viết ra dưới dạng nào không phải là mối quan tâm của người kiểm thử, miễn là nó đã được đúc kết thành một tài liệu mô tả sản phẩm để phát triển và người kiểm thử sẽ tiến hành các khảo sát trên tài liệu này để tìm các lỗi đặc tả có thể có. Cũng có trường hợp đặc tả không được viết ra. Trong trường hợp này, nó ở trong đầu của người thiết kế và lập trình, và người kiểm thử phải khảo sát các tài liệu này bằng việc phỏng vấn họ. Mục này giới thiệu hai kỹ thuật khảo sát đặc tả là duyệt đặc tả mức cao và duyệt đặc tả mức thấp nhằm nâng cao chất lượng của các đặc tả phần mềm. Đây là những công việc đầu tiên và không thể thiếu trước khi tiến hành các hoạt động đảm bảo chất lượng về sau.

4.1.1 Tiến hành duyệt đặc tả mức cao

Định nghĩa một sản phẩm phần mềm là một việc khó. Đặc tả thường liên quan đến nhiều thứ chưa biết, việc xây dựng đặc tả lấy vô số đầu vào thay đổi, kết hợp chúng với nhau để tạo nên một tài liệu mô tả sản phẩm mới. Quá trình này là khoa học không chính xác và rất dễ mắc sai lầm.

Bước thứ nhất để kiểm thử đặc tả không phải đi vào chi tiết ngay để tìm lỗi mà là xem xét nó từ mức cao. Hãy khảo sát đặc tả để tìm các lỗi cơ bản lớn, những lỗi về bỏ sót trước đã. Việc xem xét đặc tả lúc này là theo quan điểm nghiên cứu nhiều hơn là kiểm thử. Chỉ khi đã hiểu những "tại sao" và "làm thế nào" ở đằng sau đặc tả, bạn mới có thể phản biện tốt các chi tiết trong nó. Sau đây là các kỹ thuật để tiến hành duyệt đặc tả mức cao.

4.1.1.1 Hãy là khách hàng của sản phẩm

Khi nhận một tài liệu đặc tả để kiểm thử, điều dễ nhất người kiểm thử cần làm là hãy giả định mình là khách hàng của sản phẩm. Vì thế, trước hết cần tìm hiểu ai sẽ là khách hàng của sản phẩm. Hãy nói chuyện với những người chào hàng và ma-ket-ting cho sản phẩm để tìm hiểu về người dùng cuối cùng của sản phẩm. Nếu sản phẩm là trung gian trong một dự án phần mềm khác, hãy tìm hiểu ai sẽ dùng nó và nói chuyện với họ.

Điều quan trọng là cần phải hiểu khách hàng chờ đợi gì ở sản phẩm này. Thỏa mãn yêu cầu người dùng là yếu tố chất lượng quan trọng nhất. Để hiểu yêu cầu người dùng không nhất thiết phải là một chuyên gia trong lĩnh vực ứng dụng. Tuy nhiên, hiểu biết chút ít về nó sẽ giúp cho việc kiểm thử tốt hơn.

Không được giả thiết bất cứ cái gì khi tìm hiểu về đặc tả. Khi ta khảo sát một phần của đặc tả và không hiểu thì không được cho là nó đúng. Ta sẽ dùng đặc tả để thiết kế các ca kiểm thử sau này. Vì thế, nếu không hiểu được đặc tả thì sẽ không thiết kế tốt các ca kiểm thử. Xem xét đặc tả theo quan điểm người dùng giúp ta phát hiện những lỗi bỏ sót hoặc sai với yêu cầu của họ. Một điều cần lưu ý là tính an ninh và bảo mật thường được giả thiết bởi người dùng. Khi giả định là người dùng, người kiểm thử không được bỏ qua yêu cầu này.

4.1.1.2 Hãy nghiên cứu các chuẩn và hướng dẫn hiện hành

Trên thế giới đã có nhiều nghiên cứu về cách con người sử dụng máy tính, và hiện nay đã có những chuẩn cả về phần cứng lẫn phần mềm về giao diện với người sử dụng, tương tác người máy, v.v. Vì thế, người phát triển phần mềm tốt nhất hãy tuân thủ nghiêm ngặt các chuẩn này. Các chuẩn về tương tác người máy được cải tiến để đáp ứng tốt nhất các yêu cầu của người dùng. Sau đây là một số chuẩn có thể nghiên cứu để áp dụng trong đặc tả phần mềm:

- **Hợp thức các thuật ngữ và quy ước.** Nếu phần mềm được làm riêng cho một công ty nào đó hãy sử dụng các thuật ngữ và quy ước của họ.
- **Yêu cầu công nghiệp.** Trong mỗi lĩnh vực ứng dụng như y tế, dược phẩm, tài chính, v.v. đều có các chuẩn riêng và nghiêm ngặt của họ mà phần mềm phải tuân thủ.
- **Chuẩn quy định bởi chính phủ.** Chính phủ có những quy định, đặc biệt trong các lĩnh vực quốc phòng, an ninh và quản lý mà phần mềm phải tuân thủ.
- **Giao diện đồ họa với người sử dụng (GUI).** Các phần mềm chạy trong Windows hoặc Macintosh phải tuân thủ các quy định về giao diện đồ họa của các hệ điều hành này.
- **Chuẩn bảo mật.** Phần mềm có thể phải thỏa mãn một số quy định về bảo mật mà cần phải được chứng nhận và cấp phép.

Người kiểm thử cần nắm được các chuẩn này để kiểm thử xem phần mềm có thỏa mãn hay không, có gì bị bỏ qua hay không. Các chuẩn này được coi là một phần của đặc tả khi thẩm định phần mềm.

4.1.1.3 Hãy xem xét và kiểm thử các phần mềm tương tự

Một trong các phương pháp để hiểu phần mềm của ta sẽ như thế nào là nghiên cứu các sản phẩm tương tự. Đó có thể là sản phẩm cạnh tranh hoặc sản phẩm nào đó giống như sản phẩm ta đang phát triển. Rất có thể điều đó đã được làm bởi người quản lý dự án hoặc chính người viết đặc tả cho sản phẩm ta đang phát triển. Các phần mềm không thể giống hệt nhau (là lý do để ta xây dựng sản phẩm tương tự), nhưng việc nghiên cứu các sản phẩm tương tự này giúp ta xây dựng cách tiếp cận kiểm thử sản phẩm của mình.

Những điểm cần tìm kiếm khi xem xét các sản phẩm tương tự bao gồm:

- **Kích cỡ.** Các đặc trưng sẽ nhiều hơn hay ít hơn? Chương trình sẽ ít hay nhiều lệnh hơn, việc kiểm thử có bị ảnh hưởng bởi kích cỡ không?

- **Độ phức tạp.** Tương tự như kích cỡ, độ phức tạp sẽ cao hơn hay thấp hơn, và điều đó ảnh hưởng thế nào đến kiểm thử?

- **Tính khả kiểm thử.** Liệu ta có đủ tài nguyên, thời gian và trình độ để kiểm thử phần mềm như vậy?

- **Chất lượng và độ tin cậy.** Liệu phần mềm này đại diện cho chất lượng của phần mềm đang xây dựng, độ tin cậy sẽ cao hơn hay thấp hơn?

- **Tính bảo mật.** Phần mềm cạnh tranh này có tính bảo mật so với sản phẩm ta đang phát triển thế nào?

Ta sẽ thu được nhiều kinh nghiệm để khảo sát đặc tả của sản phẩm của mình bằng việc xem xét các vấn đề trên.

4.1.2 Các kỹ thuật kiểm thử đặc tả ở mức thấp

Sau khi đã hoàn thành việc khảo sát đặc tả bậc cao, ta hiểu rõ hơn về sản phẩm của mình và những yếu tố bên ngoài ảnh hưởng đến thiết kế của sản phẩm phần mềm. Khi được trang bị những thông tin này, chúng ta sẽ tiếp tục khảo sát đặc tả ở mức thấp và chi tiết hơn. Mục này sẽ giải thích các chi tiết cần tiến hành nhằm đạt được mục tiêu này.

4.1.2.1 Danh sách các hạng mục cần thẩm định về các thuộc tính của đặc tả

Một đặc tả sản phẩm phần mềm được xây dựng tốt cần thỏa mãn tám thuộc tính sau đây:

- **Đầy đủ.** Liệu đặc tả còn thiếu cái gì không? Đã đủ chi tiết chưa? Liệu nó đã bao gồm mọi điều cần thiết để không phụ thuộc vào tài liệu khác?
- **Trúng đích.** Liệu nó đã cung cấp lời giải đúng đắn cho bài toán, liệu nó đã xác định đầy đủ các mục tiêu và không có lỗi.
- **Chính xác, không nhập nhằng và rõ ràng.** Mô tả có chính xác không, có rõ ràng và dễ hiểu không, liệu còn có gì là nhập nhằng với ý nghĩa không xác định?
- **Tương thích.** Các đặc trưng và chức năng được mô tả có bị xung đột với nhau không?
- **Hợp lệ.** Các khẳng định có thực sự cần thiết để mô tả đặc trưng sản phẩm không? Có gì thừa không? Có thể truy ngược về yêu cầu của người dùng không?
- **Khả thi.** Liệu đặc tả có thể được cài đặt trong khuôn khổ nhân lực, công cụ, tài nguyên, thời gian và kinh phí cho phép hay không?
- **Phi mã lệnh.** Trong đặc tả không được dùng các câu lệnh hoặc thuật ngữ cho người lập trình. Ngôn ngữ dùng trong đặc tả phải là phổ biến với người dùng.
- **Khả kiểm thử.** Liệu các đặc trưng có thể kiểm thử được? Liệu đã cung cấp đủ thông tin để có thể kiểm thử và xây dựng các ca kiểm thử?

Khi xây dựng đặc tả phần mềm, hãy duyệt đặc tả (văn bản, hình vẽ, v.v.) và đánh giá xem nó có thỏa mãn các thuộc tính nêu trên chưa.

4.1.2.2 Danh sách các hạng mục cần thẩm định về các thuật ngữ của đặc tả

Bên cạnh danh sách các thuộc tính trên đây là danh sách các từ hay có vấn đề cần tìm trong đặc tả khi phản biện. Sự xuất hiện của các từ này có thể tiết lộ rằng các đặc trưng được mô tả chưa được suy nghĩ thấu đáo và không gặp đủ các thuộc tính trên đây. Hãy tìm các từ này trong đặc tả và xem xét cẩn thận xem có lỗi ở đó không.

- **Luôn luôn, mỗi một, tất cả, không có, không bao giờ.** Những từ như vậy mô tả sự tuyệt đối và chắc chắn. Hãy xét xem tình trạng có đúng như vậy không, có trường hợp nào vi phạm các khẳng định đó hay không.

- **Tất nhiên, do đó, rõ ràng là, hiển nhiên là.** Các từ này được dùng để thuyết phục người khác chấp nhận cái gì đó. Đừng có rơi vào các bẫy đó.

- **Nào đó, đôi khi, thông thường, thường gặp, hầu hết.** Các từ này là nhập nhằng, không có nghĩa rõ ràng và không thể kiểm thử, chẳng hạn bạn phải kiểm thử tính "đôi khi" thế nào?

- **Vân vân, chẳng hạn, như vậy.** Các từ này thường mô tả thứ mà không thể kiểm thử được.

- **Tốt, nhanh, rẻ, hiệu quả, ổn định.** Những từ không định lượng như vậy sẽ mô tả các hạng mục không thể kiểm thử được nếu không được làm chi tiết hóa sau này.

- **Xử lý, từ chối, bỏ qua, bị khử.** Những thuật ngữ này thường dấu trong nó những chức năng cần phải đặc tả đầy đủ.

- **Nếu ... thì (thiếu trái lại).** Trường hợp "trái lại" có thể bị bỏ quên. Hãy tránh các khẳng định như vậy.

Trên đây là kỹ thuật kiểm thử hộp đen tĩnh, tức là phản biện đặc tả của sản phẩm, tìm xem đặc tả có lỗi không. Khi đặc tả đã được hoàn thành và đã được phản biện, mã nguồn của chương trình được phát triển dựa trên đặc tả này. Bây giờ, chúng ta sẽ sử dụng kỹ thuật tương tự để phản biện/khảo sát mã nguồn.

4.2 Khảo sát mã nguồn

Mục này giới thiệu các kỹ thuật cơ sở để tiến hành kiểm chứng thiết kế và mã nguồn của phần mềm. Nội dung chính của mục này bao gồm việc phân tích lợi ích của công việc, các kiểu khảo sát mã nguồn, một số hướng dẫn về chuẩn lập trình, và cách chung để khảo sát mã nguồn nhằm tìm lỗi.

4.2.1 Khảo sát thiết kế và mã nguồn hay là việc kiểm thử hộp trắng tĩnh

Kiểm thử tĩnh tức là việc kiểm thử chỉ gồm việc khảo sát mà không cần tiến hành chương trình, còn kiểm thử hộp trắng là việc kiểm thử có trong tay mã nguồn của chương trình. Vì thế kiểm thử hộp trắng tĩnh chính là việc khảo sát thiết kế và mã nguồn của chương trình. Công việc này bao gồm một quy trình để khảo sát một cách cẩn thận và có phương pháp đối với thiết kế, kiến trúc và mã nguồn của phần mềm để tìm lỗi mà không cần tiến hành phần mềm. Công việc này còn có tên khác nữa là *phân tích cấu trúc*.

Lý do hiển nhiên để tiến hành việc kiểm thử hộp trắng tĩnh là nhằm phát hiện lỗi sớm, và có thể phát hiện những lỗi mà khó có thể được phát hiện và định vị bằng các kỹ thuật kiểm thử hộp đen động. Vì thế việc tập trung nhân lực vào việc khảo sát thiết kế của phần mềm ở giai đoạn sớm của quá trình phát triển sẽ cho hiệu quả cao về chi phí. Một lợi ích thiết thực nữa của việc tiến hành kiểm thử hộp trắng tĩnh là nó cung cấp cho cho người kiểm thử hộp đen động những thông tin quan trọng để xác định những đặc trưng dễ bị mắc lỗi để tìm các ca kiểm thử thích hợp và hiệu quả.

Tuy nhiên, việc kiểm thử hộp trắng tĩnh không phải lúc nào cũng được tiến hành. Nhiều người không nhận thức đầy đủ tầm quan trọng của nó và còn hiểu lầm rằng công việc này tiêu tốn thời gian, tốn kém và không năng suất. Tính hiệu quả của việc lập trình theo cặp (lập trình viên) của phương pháp phát triển phần mềm linh hoạt (Agile) đã chứng tỏ hiệu quả của kiểm thử hộp trắng tĩnh. Ngày nay ngày càng nhiều công ty phát triển phần mềm áp dụng kỹ thuật này, và vì thế, cơ hội cho những người làm kiểm thử hộp trắng tĩnh ngày càng nhiều. Sau đây là các loại kiểm thử hộp trắng tĩnh phổ biến.

4.2.2 Phản biện hình thức

Phản biện hình thức là quy trình để tiến hành kiểm thử hộp trắng tĩnh. Quy trình này có thể được áp dụng ở nhiều mức độ khác nhau, từ một cuộc họp đơn giản giữa hai lập trình viên cho đến cuộc thanh tra chi tiết và nghiêm túc đối với thiết kế và mã nguồn. Có bốn hạng mục cơ bản cho một lần phản biện hình thức như sau:

- **Xác định vấn đề.** Mục đích của phản biện là tìm xem phần mềm có vấn đề gì không, không chỉ là những thứ bị sai, mà cả những thứ còn thiếu. Tất cả những phê phán này là nhằm vào thiết kế hoặc mã nguồn đang được khảo sát chứ không

nhằm vào người tạo ra chúng. Người tham gia phản biện hình thức không được phép nhằm vào cá nhân nào.

- **Tuân thủ các quy tắc.** Cần xác định một tập các quy tắc để tuân thủ. Những quy tắc này có thể là: lượng mã nguồn để phản biện (thường là vài trăm dòng lệnh), thời gian dành cho việc phản biện (vài giờ), những gì cần được nhận xét, vân vân. Điều này là quan trọng vì người tham gia sẽ biết vai trò của họ là gì, và họ kỳ vọng cái gì, nhằm làm cho việc phản biện trơn tru hơn.

- **Chuẩn bị.** Mỗi người tham gia đều phải chuẩn bị và đóng góp vào việc phản biện. Tùy thuộc vào kiểu phản biện mà mỗi người tham gia có thể có các vai trò khác nhau. Họ cần biết nhiệm vụ và trách nhiệm của mình và sẵn sàng hoàn thành tốt chúng trong việc phản biện. Hầu hết các vấn đề được phát hiện là ở giai đoạn chuẩn bị chứ không phải ở giai đoạn phản biện thực sự.

- **Viết báo cáo.** Nhóm phản biện cần đưa ra một báo cáo bằng văn bản tóm tắt kết quả của việc phản biện và thông báo cho những người tham gia phần còn lại của dự án phát triển sản phẩm này. Những người này cần biết kết quả của buổi họp: bao nhiêu vấn đề được phát hiện, chúng được phát hiện ở chỗ nào, v.v.

Việc phản biện hình thức chỉ có hiệu quả khi tuân thủ một quy trình được xác định trước. Việc này, khi được tiến hành thực sự nghiêm túc sẽ là cách rất tốt để phát hiện lỗi.

Ngoài việc tìm ra các vấn đề đối với thiết kế và mã nguồn, việc phản biện hình thức còn có vài kết quả gián tiếp nữa là:

- **Truyền tải thông tin.** Những thông tin không gồm trong báo cáo hình thức cũng được truyền tải. Ví dụ, người kiểm

thử hộp đen động có thể biết được vấn đề đang nằm ở chỗ nào. Người lập trình ít kinh nghiệm có thể học hỏi thêm được từ những người có kinh nghiệm. Người quản lí có thể biết rõ hơn về tiến độ của dự án so với kế hoạch.

- **Chất lượng.** Khi mà mã của một người lập trình sẽ bị kiểm duyệt một cách chi tiết từng dòng một, người lập trình sẽ cẩn thận hơn trong việc viết chương trình.

- **Tính đồng đội.** Khi việc phản biện được tiến hành một cách thực sự, nó sẽ là chỗ để người lập trình và người kiểm thử gặp gỡ và phát triển lòng kính trọng lẫn nhau về kỹ năng công việc và hiểu biết về công việc của nhau cũng như tầm quan trọng của họ.

- **Lời giải.** Lời giải cho các vấn đề được phát hiện có thể được tìm thấy thông qua việc thảo luận ngoài cuộc họp.

4.2.3 Phản biện chéo

Phản biện chéo là cách phản biện dễ nhất và ít hình thức nhất. Điều này cũng giống như loại thảo luận "tôi cho bạn biết phần của tôi và bạn cho tôi biết phần của bạn". Phản biện chéo thường được tổ chức giữa lập trình viên thiết kế kiến trúc hoặc viết mã nguồn và người lập trình khác hoặc người kiểm thử trong vai trò người phản biện. Nhóm nhỏ này sẽ khảo sát mã nguồn cùng nhau để tìm lỗi hoặc các vấn đề tồn tại. Để đảm bảo việc khảo sát là hiệu quả và không trở thành cuộc tán gẫu, mỗi người cần đảm bảo bốn hạng mục nêu trong phản biện hình thức. Vì phản biện chéo là ít hình thức hơn, nên bốn hạng mục này thường được giảm nhẹ. Tuy nhiên, chúng vẫn giúp việc thảo luận về mã nguồn và tìm lỗi hiệu quả hơn.

4.2.4 Thông qua

Thông qua là bước kế tiếp hình thức sau phản biện chéo. Trong buổi thông qua, người lập trình sẽ trình bày hình thức về mã nguồn của mình trước một nhóm gồm năm người gồm những người lập trình khác và người kiểm thử để họ thông qua. Những người này đã nhận được một bản sao của mã nguồn này để họ xem xét trước và chuẩn bị nhận xét, câu hỏi hoặc thắc mắc cho buổi thông qua.

Người trình bày đọc mã nguồn theo từng dòng hoặc từng hàm, giải thích chúng làm gì và tại sao lại được xây dựng như vậy. Người nghe sẽ hỏi khi có điều gì thắc mắc. So với phản biện chéo, số người thông qua sẽ nhiều hơn, nên những người này phải chuẩn bị trước cho việc phản biện và tuân thủ các quy tắc. Điều quan trọng là sau phản biện, người trình bày phải viết báo cáo về các vấn đề được phát hiện và kế hoạch khắc phục.

4.2.5 Thanh tra

Thanh tra là hoạt động có tính hình thức nhất của việc khảo sát mã nguồn. Việc này có tổ chức cao và đòi hỏi người tham gia phải được đào tạo bài bản. Thanh tra khác với phản biện chéo và thông qua ở chỗ người trình bày mã nguồn, gọi là người trình bày hoặc độc giả, không phải là người viết ra mã nguồn. Điều này đòi hỏi người trình bày phải đọc và hiểu được tư liệu cần trình bày và sẽ cho ý kiến khách quan và thể hiện khác trong cuộc họp thanh tra. Những người tham gia khác được gọi là các thanh tra viên. Nhiệm vụ của mỗi thanh tra viên là phản biện mã nguồn từ các khía cạnh và góc độ khác nhau, chẳng hạn của người dùng đầu cuối, người kiểm thử, hoặc người bảo trì sản phẩm. Điều này giúp việc đưa các quan điểm khác nhau về sản phẩm ra phản biện, và thường là xác định được các lỗi khác nhau của sản phẩm. Cần có một thanh tra viên duyệt mã nguồn từ cuối đến đầu (theo kiểu hồi quy) để đảm bảo mã nguồn được duyệt đầy đủ. Chúng ta cũng cần một vài

thanh tra viên làm điều hành viên và thư ký để đảm bảo các quy tắc được tuân thủ và buổi họp thanh tra được tiến hành hiệu quả.

Sau buổi họp thanh tra, các thanh tra viên có thể cần gặp nhau lần nữa để thảo luận về các lỗi được phát hiện và làm việc với điều hành viên để chuẩn bị báo cáo và xác định điều cần làm để khắc phục lỗi. Người lập trình sau đó sẽ thay đổi mã nguồn để đưa ra bản đã được sửa lỗi và điều hành viên sẽ kiểm tra xem việc sửa đã được làm thực sự hay chưa.

Phụ thuộc và phạm vi, kích thước và tầm quan trọng của phần mềm mà các buổi thanh tra tiếp theo có nên tổ chức hay không để phát hiện các lỗi còn lại.

Thanh tra đã chứng tỏ là một kỹ thuật phát hiện lỗi hiệu quả trong các sản phẩm phần mềm, đặc biệt trong các tài liệu thiết kế và mã nguồn. Kỹ thuật này đang trở thành phổ dụng vì các công ty và đội ngũ phát triển sản phẩm đã phát hiện ra những lợi ích to lớn mà nó mang lại.

4.2.6 Các chuẩn và hướng dẫn trong lập trình

Trong việc phản biện hình thức, các thanh tra viên tìm kiếm các vấn đề tồn tại và các thiếu sót trong chương trình. Chúng là các lỗi kinh điển chẳng hạn cái gì đó bị viết sai. Chúng được phát hiện bằng cách phân tích mã nguồn. Công việc này được tiến hành hiệu quả hơn bởi những người lập trình và kiểm thử giàu kinh nghiệm.

Còn một vấn đề khác nữa trong lập trình là chương trình chạy đúng nhưng không được viết theo các chuẩn và hướng dẫn quy định trước. Các chuẩn khi đã được thiết lập thì cần phải được tuân thủ. Còn các hướng dẫn là những kinh nghiệm thực hành được đề nghị là cách làm tiện lợi hơn cho công việc. Chuẩn là nghiêm ngặt, còn hướng dẫn có thể ít nghiêm ngặt hơn. Có thể có các mẩu phần mềm chạy ổn định nhưng vẫn không đúng do không thỏa mãn chuẩn. Chúng ta nên tuân theo chuẩn và hướng dẫn vì ba lý do sau:

- **Độ tin cậy.** Thực tế đã chứng tỏ rằng chương trình viết theo chuẩn và hướng dẫn có độ tin cậy và bảo mật cao hơn.
- **Tính dễ hiểu và dễ bảo trì.** Các chương trình viết theo chuẩn và hướng dẫn dễ đọc, dễ hiểu và dễ bảo trì hơn.
- **Tính dễ chuyển đổi.** Các chương trình viết theo chuẩn và hướng dẫn dễ được chuyển đổi để dịch bởi các chương trình dịch khác nhau cho các thiết bị tính toán với các hệ điều hành khác nhau.

Các dự án khác nhau có thể đòi hỏi các chuẩn khác nhau (nội bộ, quốc gia hoặc quốc tế). Điều quan trọng là đội ngũ dự án phải có chuẩn, có hướng dẫn cho lập trình và phải được kiểm chứng xem chúng có được tuân thủ không trong khi phản biện. Sau đây là vài ví dụ về chuẩn và hướng dẫn trong lập trình.

Hình 4.1 giới thiệu một ví dụ về chuẩn lập trình liên quan đến việc sử dụng các lệnh **go to**, **if-then-else** và **while** trong ngôn ngữ C. Sử dụng các lệnh này không cẩn thận dễ gây lỗi và hầu hết các chuẩn lập trình đều đặt ra các quy tắc cho việc sử dụng các lệnh này.

Chuẩn thường có bốn phần chính như sau:

- Tiêu đề mô tả chủ đề mà chuẩn đó nói đến.
- **Chuẩn (hoặc hướng dẫn)** mô tả chính xác cái gì cho phép và cái gì không cho phép.
- **Thuyết minh** nêu lý do tại sao phải có chuẩn đó.
- **Ví dụ** chỉ ra vài mẫu về việc sử dụng chuẩn. Phần này không luôn là cần thiết.

Câu hỏi đặt ra là các chuẩn sẽ được lấy từ đâu. Chẳng hạn khi xây dựng các chuẩn cho một dự án phần mềm, có một số nguồn

CHỦ ĐỀ: 3.05 Hạn chế về điều khiển đối với các cấu trúc điều khiển

CHUẨN
Không được dùng lệnh **go to** (và do đó cả nhãn lệnh).

Dùng chu trình **while** thay cho **do-while** trừ khi lôgic của bài toán đòi hỏi tường minh là phải tiến hành thân chu trình ít nhất một lần không phụ thuộc vào điều kiện chu trình.

Nếu lệnh **if-then-else** có thể thay thế **continue** thì phải dùng **if-then-else**.

THUYẾT MINH
Lệnh **go to** bị cấm vì dễ gây lỗi và khó đọc, khó theo dõi dòng điều khiển của chương trình. Thuật toán cần được trình bày theo cách có cấu trúc.

Lệnh **do-while** không nên dùng vì chu trình cần được viết dưới dạng thống nhất là điều kiện chu trình phải được kiểm tra trước khi tiến hành thân chu trình.
...

Hình 4.1: Ví dụ về chuẩn lập trình trong một số ngôn ngữ lập trình.

để ta tham khảo. Các chuẩn quốc tế và các chuẩn của Mỹ và của quốc gia sẵn có với hầu hết các ngôn ngữ lập trình và công nghệ thông tin. Một số nguồn đó là:

- Viện tiêu chuẩn quốc gia Hoa Kỳ (ANSI), www.ansi.org

- Tập đoàn công nghệ quốc tế (IEC), www.iec.org
- Tổ chức quốc tế về chuẩn (ISO), www.iso.ch
- Ủy ban quốc gia về chuẩn trong công nghệ thông tin của Hoa Kỳ (NCITS), www.ncits.org

Các tổ chức và hiệp hội chuyên môn cũng cung cấp các tài liệu hướng dẫn và thực hành về lập trình như:

- Hiệp hội máy tính Mỹ (ACM), www.acm.org
- Viện kỹ nghệ điện và điện tử (IEEE), www.ieee.org

4.2.7 Danh sách các hạng mục chung cho việc khảo sát mã nguồn

Mục này nêu những vấn đề cần tập trung tìm tòi khi khảo sát mã nguồn ngoài việc đối chiếu với các chuẩn và hướng dẫn.
Lỗi tham chiếu dữ liệu: Đó là các lỗi gây ra do việc dùng các biến, hằng, xâu hoặc bản ghi mà chưa được mô tả hoặc khởi tạo.

- Liệu có trong chương trình việc tham chiếu đến biến chưa khởi tạo giá trị?
- Giá trị của chỉ số mảng hoặc xâu có là nguyên và có nằm trong cận và số chiều cho phép?
- Các phép tính trên chỉ số mảng có vượt ra ngoài cận cho phép?
- Lựa chọn giữa biến và hằng có hợp lý?
- Có vi phạm về kiểu trong khi gán giá trị cho biến?
- Giá trị tham chiếu của con trỏ có nằm trong vùng nhớ được phân phối?

- Tương ứng giữa tham số hình thức và thực sự trong các lời gọi hàm và chương trình con.

Lỗi mô tả dữ liệu: Các lỗi này gây bởi việc mô tả không đầy đủ hoặc do việc dùng bất cẩn các biến và hằng số.

- Liệu các biến có được gán độ dài, kiểu, lớp đúng đắn? Chẳng hạn dùng kiểu xâu thay cho kiểu mảng.
- Liệu các biến có được khởi tạo và khởi tạo hợp kiểu khi mô tả?
- Các biến với tên tương tự nên tránh vì dễ dùng nhầm.
- Có biến không được tham chiếu hoặc tham chiếu chỉ một lần?
- Phạm vi của mô tả biến có bị vi phạm?

Lỗi tính toán: Đó là các lỗi về toán, gây ra kết quả tính toán không như mong đợi.

- Có hay không việc tính toán số học với các kiểu khác nhau?
- Tính toán với các biến cùng kiểu nhưng khác độ dài.
- Các quy tắc biến đổi kiểu của chương trình dịch có được hiểu và dùng đúng?
- Giá trị của biểu thức vế phải của phép gán vượt ra ngoài miền giá trị của biến vế trái.
- Lỗi tràn ô nhớ trong khi ước lượng giá trị của biểu thức, lỗi chia cho không.
- Lỗi về sai số làm tròn.
- Có nhầm lẫn về thứ tự ước lượng các phép toán trong biểu thức.

Lỗi so sánh: Các lỗi về so sánh và quyết định chính là các vấn đề hay gặp về điều kiện biên. Chúng là:

- Quan hệ trong so sánh có đúng đắn, chẳng hạn chúng ta hay nhầm lẫn giữa việc sử dụng $\leq$ và $<$.
- Ảnh hưởng của sai số làm tròn trong so sánh.
- Các biểu thức Boolean có được viết đúng không? Các toán hạng có được dùng đúng kiểu và nghĩa (kiểu boolean và kiểu nguyên)?

Lỗi điều khiển: Đó là các lỗi gây ra, trực tiếp hay gián tiếp, do dùng không đúng các cấu trúc điều khiển như lệnh chu trình và rẽ nhánh.

- Các từ khóa mở và đóng nhóm các lệnh có đối sánh được với nhau?
- Đơn thể, chu trình có kết thúc như mong muốn?
- Có chu trình đan xen?
- Có nhánh chương trình không bao giờ được tiến hành? Nếu có thì liệu có là hợp lệ?
- Liệu các nhánh của lệnh **switch**–**case** có tương thích với điều kiện của nó.
- Liệu có chu trình với việc lặp thân chu trình không mong đợi do chỉ số vượt cận.

Lỗi truyền tham số trong chương trình con: Đó là các lỗi về sự không tương thích giữa tham số thực sự với tham số hình thức, lỗi về truyền theo tham chiếu hay theo giá trị, v.v.

Lỗi về đầu vào và đầu ra: Đó là các lỗi về đọc tệp đầu vào, khuôn dạng không tương thích khi vào dữ liệu từ bàn phím, lỗi đọc từ thiết bị không sẵn sàng cho trao đổi dữ liệu, v.v.

Các lỗi khác: Đó là các lỗi chưa được liệt kê trên đây, chẳng hạn lỗi về ngôn ngữ, mã ký tự (UNICODE or ASCII), lỗi hiển thị, lỗi về chuyển đổi giữa các hệ điều hành, lỗi về tương thích với phần cứng khác nhau, v.v.

4.3 Tổng kết

Trong chương này, chúng ta đã đề cập đến tầm quan trọng của việc phân tích tĩnh đối với đặc tả và mã nguồn và các kỹ thuật cơ bản để tiến hành việc này. Các kỹ thuật này bao gồm việc duyệt, khảo sát, phản biện, thanh tra. Việc khảo sát để phát hiện lỗi ngày càng chứng tỏ là một kỹ thuật kiểm thử hiệu quả nhưng nó đòi hỏi người kiểm thử phải được chuẩn bị và đào tạo, và làm việc với năng suất cao, và phải tuân thủ các quy tắc khi áp dụng kỹ thuật này. Các kỹ thuật phân tích tĩnh được xem là bước tiền xử lý và không thể thiếu trước khi tiến hành quá trình kiểm thử (phân tích động).

4.4 Bài tập

1. Liệu người kiểm thử có thể tiến hành kiểm thử hộp trắng trên đặc tả?

2. Hãy trích dẫn một vài chuẩn và hướng dẫn của Windows và Macintosh.

3. Khẳng định sau đây trong đặc tả sai ở điểm nào, tại sao: khi người dùng chọn tùy chọn rút gọn bộ nhớ, chương trình sẽ nén danh sách các địa chỉ e-mail nhỏ nhất có thể dùng cách tiếp cận ma trận thưa Huffman.

4. Cái gì làm cho người kiểm thử băn khoăn trong đặc tả sau đây: phần mềm sẽ cho phép tới 100 triệu kết nối đồng thời, dù rằng thông thường không có quá một triệu người dùng.

5. Hãy liệt kê các lợi ích của việc kiểm thử hộp trắng tĩnh.

6. Kiểm thử hộp trắng tĩnh có thể phát hiện cả các vấn đề tồn tại của mã lẫn những thứ còn thiếu. Nói thế đúng hay sai?

7. Phản biện hình thức gồm những hạng mục nào?

8. Ngoài việc là hình thức hơn, sự khác nhau lớn nhất giữa thanh tra và các loại phản biện khác là gì?

9. Nếu một lập trình viên được thông báo là có thể dùng tên biến với 8 ký tự với ký tự đầu viết hoa (chữ in). Điều đó là chuẩn hay hướng dẫn?

10. Bạn có định áp dụng danh sách các hạng mục chung cho việc khảo sát mã nguồn trong chương này để phản biện mã trong dự án phần mềm tương lai?

11. Lỗi về bảo mật gây ra bởi việc tràn bộ đệm thuộc loại nào trong danh sách nói trên?

Chương 5

Kiểm thử chức năng

Cách tiếp cận kiểm thử chức năng (cũng có tên gọi khác là kiểm thử hộp đen) đang được các công ty phần mềm sử dụng như là giải pháp chủ yếu nhằm đảm bảo chất lượng cho các sản phẩm phần mềm. Chương này giới thiệu chi tiết về cách tiếp cận này thông qua các phương pháp kiểm thử chức năng phổ biến. Có nhiều phương pháp kiểm thử chức năng đã được đề xuất và sử dụng trong thực tế. Chúng tôi sẽ giới thiệu chi tiết ba phương pháp phổ biến nhất gồm phương pháp phân tích giá trị biên, phân lớp tương đương và kiểm thử bằng bảng quyết định. Mỗi phương pháp là không đủ để đảm bảo chất lượng sản phẩm. Vì vậy, chúng tôi cũng sẽ giới thiệu các chiến lược kết hợp các phương pháp này nhằm tăng khả năng phát hiện lỗi của quá trình kiểm thử.

5.1 Tổng quan

Kiểm thử chức năng (functional testing) là các hoạt động kiểm tra chương trình dựa trên tài liệu mô tả chức năng, yêu cầu phần mềm, hay còn gọi là *đặc tả chức năng* (functional specification).

Đặc tả chức năng là tài liệu mô tả yêu cầu, hành vi mong muốn của chương trình. Tài liệu này là cơ sở để chúng ta tiến hành xây dựng các ca kiểm thử, thuật ngữ chuyên môn gọi là *thiết kế kiểm thử* (test design) và người thực hiện việc này là người thiết kế kiểm thử. Thiết kế kiểm thử là hoạt động chính trong kiểm thử chức năng.

Cần nhấn mạnh rằng kiểm thử chức năng là việc thiết kế các ca kiểm thử nhằm phát hiện các lỗi hoặc khiếm khuyết về chức năng của chương trình cần kiểm thử. Công việc này chỉ dựa trên đặc tả của chương trình mà không dựa trên việc phân tích mã nguồn của chương trình. Kiểm thử chức năng còn được gọi với tên khác chính xác hơn là kiểm thử dựa trên đặc tả, hay kiểm thử hộp đen, hay kiểm thử hàm. Còn việc thiết kế kiểm thử có dựa vào mã nguồn của chương trình được gọi là kiểm thử cấu trúc hay kiểm thử hộp trắng.

Kiểm thử chức năng là một cách tiếp cận cơ bản giúp phát hiện nhiều lỗi mà kiểm thử hộp trắng không phát hiện được. Ví dụ dễ thấy nhất là các lỗi do cài đặt (implementation) không đầy đủ so với đặc tả. Nếu phần mềm thiếu hẳn một chức năng thì chúng ta dễ nhận thấy, nhưng nếu phần mềm thiếu một trường hợp của một chức năng thì việc này không dễ phát hiện nếu không kiểm tra kỹ chương trình so với đặc tả.

Tài liệu đặc tả dùng để thiết kế các ca kiểm thử có thể đơn giản là mô tả về chức năng của chương trình viết bằng ngôn ngữ của người sử dụng, nhưng thường là cả các tài liệu phân tích, thiết kế ở các mức độ chi tiết hơn với các hình vẽ, bảng biểu, sơ đồ, v.v. Các tài liệu này đặc tả hành vi của chương trình vì chúng mô tả sát hơn các tính chất của chương trình mà chúng ta sẽ dựa vào đó để thiết kế kiểm thử. Chúng ta gọi chung các tài liệu này là đặc tả chức năng.

Dễ nhận thấy do chỉ dựa trên đặc tả chức năng, kiểm thử chức năng có ưu điểm là có thể thực hiện được sớm, trước khi cài đặt

chương trình. Thời điểm tốt để chúng ta bắt đầu thiết kế kiểm thử là khi tài liệu đặc tả đã ổn định. Tuy nhiên, cũng có nhiều quy trình phần mềm hiện đại đang đẩy việc này lên sớm hơn. Cụ thể là ngay trong giai đoạn xác định yêu cầu, chúng ta đã lồng ghép việc xác định các ca kiểm thử. Các ca kiểm thử này thường được mô tả dưới dạng các ví dụ hoặc các kịch bản người sử dụng (user stories). Chúng vừa có tác dụng giải thích và làm rõ tài liệu đặc tả, vừa có tác dụng sử dụng như các kịch bản kiểm thử, các ca kiểm thử [Bec02, Kar12].

Thông thường, kết quả của việc thiết kế kiểm thử chức năng sẽ đưa ra *đặc tả ca kiểm thử*. Tương tự như khái niệm đặc tả chương trình là mô tả chương trình mong muốn, đặc tả kiểm thử mô tả các ca kiểm thử cần được tạo ra. Từ một đặc tả ca kiểm thử chúng ta có thể tạo ra nhiều ca kiểm thử cụ thể để tiến hành kiểm tra với chương trình. Ví dụ với một hàm giải phương trình bậc hai với đầu vào là ba hệ số là a, b, c, đặc tả ca kiểm thử cho nghiệm kép là với mọi a, b, c thỏa mãn $b^2 = 4ac$. Từ đặc tả này ta có thể tạo ra các ca kiểm thử cụ thể, ví dụ $a = 1, b = 2$, $c = 1$, và kết quả mong đợi là nghiệm kép -1.

Bản chất của các đặc tả ca kiểm thử là mỗi đặc tả ca kiểm thử sẽ xác định một lớp hành vi của chương trình. Do đó việc thiết kế kiểm thử chức năng là việc phân tách đặc tả các hành vi có thể của chương trình thành một số lớp mà mỗi lớp này sẽ có hành vi nhất quán hay tương tự. Với ví dụ hàm giải phương trình bậc hai như mô tả trên, chúng ta có thể chia đặc tả thành hành vi phương trình cho nghiệm kép, hành vi phương trình cho kết quả vô nghiệm, v.v.

Việc phân tích đặc tả sẽ giúp chúng ta xác định và phân tách được các lớp này. Bên cạnh đó, việc phân tích cũng thường giúp chúng ta xác định những điểm không rõ ràng, không đầy đủ của đặc tả phần mềm. Đây là quá trình phân tích tổng hợp từ nhiều nguồn thông tin, từ đặc tả, từ kiến thức, kinh nghiệm của người thiết kế kiểm thử, v.v. Do đó, quá trình này thường mang tính thủ

công và cũng rất dễ mắc lỗi. Trong thực tế, nhiều chuyên gia về thiết kế kiểm thử chức năng nhiều khi vẫn bỏ sót một số ca kiểm thử quan trọng.

Để giảm thiểu các sai sót trên, chúng ta cần có phương pháp hệ thống giúp chia quá trình thiết kế này thành các bước cơ bản để thực hiện. Từ đó một số bước sẽ được tự động hóa để giảm công sức, giảm sai sót, và tăng được chất lượng công việc. Thực tế việc tự động hoàn toàn chưa thực hiện được và một số bước cơ bản vẫn phải thực hiện thủ công. Do đó, kỹ năng và kinh nghiệm của người thiết kế kiểm thử đóng vai trò quyết định đến chất lượng của bộ kiểm thử.

Để hiểu rõ hơn các công việc trong thiết kế kiểm thử chức năng, chúng ta giả sử đang thiết kế kiểm thử cho chương trình P và giả sử $Y = P(X)$, trong đó X là vec-tơ của n biến đầu vào và Y là vec-tơ của m biến đầu ra. Dựa trên hiểu biết của chúng ta về đặc tả của chương trình P, với một vec-tơ đầu vào $X = (a_1, .., a_n)$ với a_i là các giá trị cụ thể thì chúng ta tính được một vec-tơ kết quả cụ thể $Y = (b_1, .., b_m)$. Kết quả Y này được gọi là *kết quả mong đợi*. Kết quả thực của chương trình P với đầu vào X khi chạy chương trình là $P(X)$. Chúng ta cần kiểm tra kết quả mong đợi đúng bằng kết quả thực của chương trình, tức là Y và $P(x)$ là một hay không. Nếu đúng thì chương trình đã chạy đúng với X. Nếu chương trình chạy đúng với mọi giá trị X như mô tả trong đặc tả thì chúng ta kết luận chương trình P đã được cài đặt đúng so với đặc tả.

5.1.1 Sự phức tạp của kiểm thử chức năng

Để hiểu sự phức tạp của kiểm thử chức năng, chúng ta xem một chương trình đơn giản $Y = roots(X)$ để tính nghiệm của phương trình bậc hai và được thiết kế như Đoạn mã 5.1. Theo hiểu biết của chúng ta, đầu vào X là ba tham số `a, b, c`, và kết quả trả về được gán cho hai biến `root_one, root_two`, tức là Y là cặp giá trị

này. Với cách trả về kết quả như đoạn mã này thì hàm trong mã nguồn không nhất thiết trả về tham số như hàm toán học. Hàm `roots` trong đoạn mã này trả về `void`, tức là không trả về gì mà kết quả được lấy từ hai biến toàn cục. Tương tự như vậy, đầu vào `a, b, c` cũng có thể truyền qua các biến toàn cục, nên chúng ta cần cẩn thận khi xác định X và Y.

Đoạn mã 5.1: Chương trình giải phương trình bậc hai

```
double root_one, root_two; // two solutions
//Solving the equation: ax^2 + bx + c = 0

void roots(double a, double b, double c){
  ...
}
```

Với những hiểu biết về hàm giải phương trình bậc hai như trên, chúng ta sẽ thiết kế kiểm thử như thế nào? Kết quả cuối cùng của quá trình này là chúng ta phải đưa ra các bộ giá trị cho `a, b, c` và tính toán bằng tay kết quả mong đợi cho hai nghiệm của phương trình này. Sau đó, chúng ta phải chạy chương trình với các bộ giá trị của `a, b, c` và so sánh kết quả của chương trình với kết quả mong đợi. Ở đoạn mã trên, chúng ta đã cho rằng các biến này có kiểu `double`. Giả sử các biến này có kiểu số nguyên 32-bit thì chúng ta có đến $2^{32^3} \simeq 10^{28}$ giá trị hợp lệ cho `a, b, c`. Nếu giả sử chương trình chạy với tốc độ 1.000.000.000.000 (1 tỷ) ca kiểm thử mỗi giây thì chúng ta vẫn cần đến hơn 2.5 tỷ năm mới chạy xong (chính xác là 2,510,588,971 năm, 32 ngày, và 20 giờ).

Như vậy, kiểm thử theo kiểu vét cạn cho phép chúng ta khẳng định một cách chắc chắn rằng hàm đã được cài đúng hay chưa. Tuy nhiên, chúng ta không thể tính toán thủ công các kết quả mong đợi với số lượng nhiều như vậy. Khối lượng tính toán lớn này cũng không khả thi khi thực hiện với máy tính.

Một cách khác là chúng ta kiểm thử hàm `roots` này bằng cách lấy ngẫu nhiên các bộ giá trị của X. Phương pháp *kiểm thử ngẫu*

nhiên này sẽ khó có thể tạo ra ca kiểm thử thỏa mãn $b^2 - 4ac = 0$. Phân phối của các khả năng có nghiệm, có nghiệm kép và không có nghiệm không như nhau nên kiểm thử ngẫu nhiên sẽ khó phát hiện những lỗi ở các trường hợp đặc biệt. Do đó, chúng ta cần phương pháp tốt hơn để có thể áp dụng cho các trường hợp tổng quát.

Trước hết, chúng ta hãy xem các công việc chính của thiết kế kiểm thử cho chương trình $Y = P(X)$:

- Xác định các biến đầu vào, đầu ra và miền giá trị của từng biến. Tức là chúng ta phải xác định X và Y gồm có những biến nào, và miền giá trị của từng biến. Giả sử X gồm các biến $x_1, .., x_n$ và Y gồm các biến $y_1, .., y_m$, chúng ta phải xác định từng $Dom(x_i)$ và $Dom(y_j)$ với $i = 1..n, j = 1..m$, trong đó $Dom(x)$ là tập các giá trị x có thể nhận.

- Chọn một số bộ giá trị của X làm ca kiểm thử, và tính (tay) kết quả mong đợi tương ứng cho từng bộ giá trị đầu vào. Với một bộ giá trị đầu vào cụ thể, ví dụ $X = (a_1, .., a_n)$, chúng ta phải tính được $Y = (b_1, .., b_m)$ để có thể so sánh với kết quả của chương trình là $P(a_1, .., a_n)$ từ đó biết chương trình đã đúng như mong đợi chưa. Việc tính thủ công này là rất khó khăn với những chương trình phức tạp.

  Trong thực tế, việc chọn bộ giá trị đầu vào thường dựa trên hiểu biết của chúng ta về miền dữ liệu và chúng ta thường lấy các giá trị đặc biệt và giá trị bình thường cho mỗi miền rồi kết hợp chúng với nhau. Sinh ca kiểm thử dựa trên miền đầu vào của các biến có hai đặc trưng chính. Một là số lượng ca kiểm thử có thể có quá nhiều khi chúng ta phải tạo các tổ hợp của các giá trị đặc biệt hoặc đại diện của các biến. Hai là việc tính kết quả mong đợi thường là đơn giản vì chúng ta đã hiểu chức năng của chương trình hoặc có thể tính "xuôi" kết quả theo đặc tả.

- Ngược lại, chúng ta có thể lấy một số bộ giá trị đầu ra mong đợi Y trước, rồi tính ngược một bộ giá trị đầu vào X để xây dựng các ca kiểm thử. Khi lấy bộ giá trị đầu ra mong đợi chúng ta sẽ thường chọn giá trị đặc biệt của miền đầu ra, hoặc chọn các bộ đầu ra dựa trên hiểu biết của chúng ta về chương trình. Như bài toán giải phương trình bậc hai chúng ta có thể lấy Y có nghiệm kép, có hai nghiệm tách biệt, v.v.

  Việc sinh ca kiểm thử dựa trên miền đầu ra có đặc trưng ngược lại so với trên. Thông thường các hàm có số đầu ra ít hơn và chúng ta cũng không thường phải lấy tổ hợp của các giá trị đặc biệt của đầu ra. Việc tính ngược lại giá trị đầu vào sẽ khó khăn hơn vì chúng ta phải lần ngược lại các bước trong đặc tả.

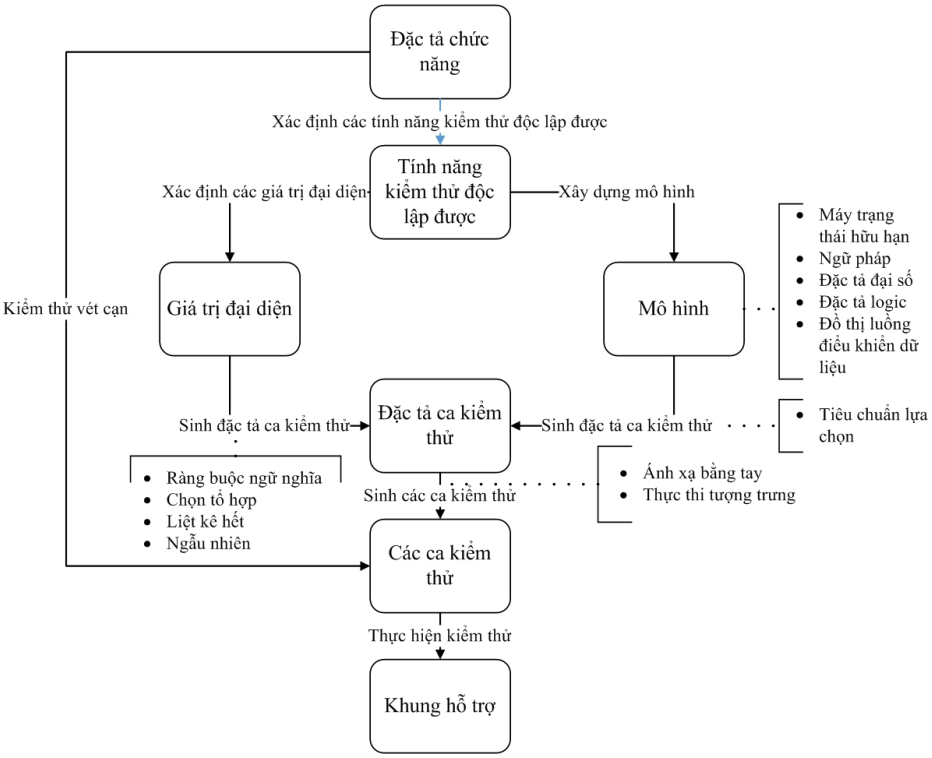
Do không thể kiểm thử vét cạn được nên chúng ta cần xác định một số ca kiểm thử đại diện cho các lớp hành vi của chương trình. Bài toán thiết kế kiểm thử trở thành bài toán xác định các lớp đại diện của chương trình. Các lớp này có thể là các lớp của đầu vào, các lớp của đầu ra, hay các lớp hành vi của chương trình dựa trên đặc tả của nó.

5.1.2 Phương pháp hệ thống

Việc xác định các ca kiểm thử từ đặc tả chức năng là quá trình phân tích để xác định và chia không gian đầu vào/đầu ra/hành vi của chương trình thành các miền con. Phương pháp hệ thống sẽ chia quá trình này thành các bước cơ bản, từ đó làm từng bước được dễ dàng hơn hoặc tự động hóa một số bước nếu có thể. Phương pháp hệ thống giúp chúng ta xác định bộ ca kiểm thử đã đủ chưa đồng thời tránh tạo ra nhiều ca kiểm thử trùng lặp, không cần thiết.

Hình 5.1 mô tả các bước chính của một phương pháp hệ thống. Một số bước trong quy trình này có thể phức tạp, một số bước khá đơn giản và một số bước có thể bỏ qua tùy theo miền ứng dụng và

đặc tả chương trình cũng như kinh nghiệm của người thiết kế. Dựa trên quy trình tổng quát này chúng ta có thể đưa ra các quy trình cụ thể bằng việc cụ thể hóa các bước trong quy trình. Chúng ta sẽ xem xét chi tiết hơn các bước trong phương pháp này.



Hình 5.1: Các bước chính của phương pháp kiểm thử chức năng.

Xác định các chức năng kiểm thử độc lập được: Đặc tả phần mềm trên thực tế là một công việc phức tạp và chứ đựng nhiều sai sót ngay cả với những hệ thống nhỏ. Chúng thường được phân rã thành các đơn vị nhỏ hơn như các hệ thống con hay cụ thể hơn là các ca sử dụng (use cases) hoặc các kịch bản sử dụng (user stories). Chúng ta gọi chúng là các chức năng. Ví dụ trang Web tìm kiếm của Google có thể có tính năng tìm kiếm, hiển thị kết quả, hiện thị quảng cáo, hay đăng ký người sử dụng mới, v.v. Tuy nhiên, ranh

giới của một chức năng nhiều khi không rõ ràng. Ví dụ, chức năng tìm kiếm của máy tìm kiếm Google có chức năng tính toán một biểu thức số học nếu văn bản tìm kiếm là một biểu thức số học.

Khi thiết kế kiểm thử chúng ta tách các chức năng càng chi tiết, cụ thể càng tốt vì nó vừa giúp đơn giản việc thiết kế kiểm thử và vừa cho phép chúng ta xác định lỗi trong chương trình dễ dàng hơn. Mỗi ca kiểm thử chỉ nên kiểm tra một kịch bản cụ thể, chi tiết của một chức năng của hệ thống.

Tiếp theo, với mỗi chức năng, chúng ta sẽ xác định tất cả các đầu vào có ảnh hưởng đến việc thực hiện chức năng đó và các đầu ra có thể bị tác động bởi chức năng đó. Các đầu vào và đầu ra thường được mô tả tường minh trong tài liệu đặc tả. Nhưng cũng có nhiều trường hợp các đầu vào và đầu ra này nằm ẩn trong tài liệu đặc tả mà người thiết kế phải phát hiện ra. Ví dụ như các đặc tả phi hình thức mô tả chức năng đăng ký tài khoản mới thường chỉ nêu các trường dữ liệu cần khai báo nhưng không nói gì đến chi tiết kiểu dữ liệu sẽ lưu thông tin hồ sơ người sử dụng. Hay một ví dụ khác khá rắc rối với tên tiếng Việt khi chúng ta chỉ có hai trường họ và tên để nhập vào. Một là thông thường chúng ta không nói ra cụ thể chiều dài tối đa, tối thiểu, hay quy định bảng mã ký tự của các trường này. Hai là chúng ta không nói rõ phần tên đệm sẽ nằm ở phần họ hay phần tên, nên khi người sử dụng nhập dữ liệu sẽ xảy ra tình trạng mỗi người nhập một kiểu. Tên là Nguyễn Văn Hùng thì có người nhập phần họ là Nguyễn, phần tên là Văn Hùng; có người lại nhập họ là Nguyễn Văn, tên là Hùng; lại có người chỉ nhập họ là Nguyễn, tên là Hùng, vì nghĩ phần Văn là tên đệm không hỏi thì không nhập.

Xác định các lớp giá trị đại diện: Các giá trị đại diện có thể được xác định trực tiếp từ đặc tả phi hình thức, như ngôn ngữ tự nhiên. Chúng cũng có thể được xác định gián tiếp dựa trên một mô hình được xây dựng để phục vụ cho việc này. Trong cả hai trường

hợp này, mục tiêu đều là xác định các giá trị của mỗi đầu vào một cách độc lập, bằng việc lấy các giá trị đã liệt kê tường minh trong đặc tả, hoặc thông qua một mô hình phù hợp. Chúng ta chưa xét đến tổ hợp của các giá trị này. Lý do là chúng ta đang tách bài toán xác định các lớp giá trị của mỗi đầu vào và bài toán tổ hợp chúng để tạo thành các ca kiểm thử. Tức là chúng ta đang tách một bước phức tạp thành các bước đơn giản hơn.

Với các đặc tả phi hình thức, người thiết kế kiểm thử phải dựa vào việc liệt kê tường minh các lớp giá trị đại diện. Trong trường hợp này, người thiết kế kiểm thử cần cẩn thận xét tất cả các trường hợp có thể và khai thác tối đa những thông tin có sẵn trong đặc tả. Tương tự, với kết quả mong đợi đầu ra, chúng ta cũng có thể xác định các lớp giá trị của miền đầu ra, cũng như các giá trị biên và các giá trị đặc biệt dễ có lỗi.

Ví dụ, khi xem xét các thao tác trên một danh sách không rỗng, chúng ta cần xét cả trường hợp danh sách rỗng (giá trị lỗi) và danh sách có một phần tử (giá trị biên chiều dài danh sách) là các trường hợp đặc biệt. Ở bước này, chúng ta chỉ quan tâm đến tính chất của giá trị chứ chưa cần cụ thể giá trị là gì. Tức là chúng ta quan tâm đến đặc tả ca kiểm thứ chứ chưa phải ca kiểm thử cụ thể. Với ví dụ danh sách một phần tử, chúng ta chưa cần quan tâm phần tử của danh sách này cụ thể là gì, chỉ biết danh sách chỉ có một phần tử.

Để liệt kê các giá trị không tường minh, chúng ta cần xây dựng mô hình hoặc một phần của mô hình của đặc tả. Ví dụ chúng ta có thể xây dựng văn phạm để mô tả ngôn ngữ và sinh các từ của ngôn ngữ này để liệt kê các giá trị cần xác định. Mô hình này có thể có sẵn trong đặc tả nhưng thông thường người thiết kế kiểm thử phải xây dựng chúng.

Trong hai cách trên, việc liệt kê trực tiếp có vẻ đơn giản hơn và ít tốn kém hơn việc xây dựng mô hình để sinh ra các giá trị. Tuy nhiên, về lâu về dài, các mô hình sẽ có giá trị hơn, khi khối lượng

giá trị lớn chúng ta có thể tự động hóa việc liệt kê, hay chúng ta cũng có thể điều chỉnh số lượng ca kiểm thử nhiều hay ít để cân bằng với các ràng buộc kiểm thử khác, như chi phí, thời gian. Các mô hình cũng có thể được sử dụng lại, sửa đổi dễ dàng hơn trong quá trình phát triển hệ thống. Quyết định cuối cùng nên dùng cách nào vẫn tùy thuộc vào lĩnh vực ứng dụng, kỹ năng của người thiết kế kiểm thử, và các công cụ thích hợp đang có.

Sinh các đặc tả ca kiểm thử: Đặc tả kiểm thử được tạo ra bằng việc tổ hợp các giá trị của các đầu vào của chương trình đang được kiểm thử. Bước trên chúng ta đã xác định các giá trị đại diện thì ở bước này chúng ta sinh các tổ hợp của chúng bằng tích Đề-các của các giá trị đại diện đã chọn. Nếu ở bước trước chúng ta tạo ra mô hình hình thức thì ở bước này các đặc tả ca kiểm thử sẽ là các hành vi cụ thể hoặc tổ hợp của các tham số của mô hình và một đặc tả ca kiểm thử có thể được thỏa mãn bởi nhiều đầu vào cụ thể. Trong cả hai cách này, việc tạo ra tất cả các tổ hợp thường tạo ra số lượng quá lớn các ca kiểm thử.

Ví dụ một hàm có năm đầu vào, mỗi đầu vào có sáu giá trị đại diện thì tích Đề-các sẽ tạo ra $6^5 = 7776$ đặc tả ca kiểm thử. Số lượng này là quá lớn với một chức năng, ngay cả khi được tự động hóa và chạy bằng máy tính hoàn toàn. Thông thường, rất nhiều tổ hợp là không hợp lệ hoặc không khả thi. Chúng ta sẽ học các phương pháp để giảm số tổ hợp này nhưng vẫn đảm bảo chất lượng của bộ kiểm thử.

Để giảm số ca kiểm thử, chúng ta cần loại bỏ các tổ hợp giá trị không hợp lệ bằng việc đưa thêm các ràng buộc về cách tổ hợp. Ví dụ hàm *NextDate* chúng ta cần đưa ràng buộc để tránh tạo ra các giá trị không hợp lệ như ngày 31/11/2013. Hoặc chúng ta cũng không xét hết tất cả các tích Đề-các mà chỉ cần mỗi cặp giá trị của hai biến bất kỳ đều xuất hiện là đủ. Quá trình này có thể cần điều chỉnh và thử một số lần để có kết quả là bộ ca kiểm thử phù

hợp. Với cách làm này, chúng ta vừa hạn chế được các ca kiểm thử không cần thiết, vừa tiết kiệm được chi phí của việc thực hiện các ca kiểm thử.

Sinh ca kiểm thử và thực hiện kiểm thử: Quá trình kiểm thử hoàn tất khi chúng ta cụ thể hóa các đặc tả kiểm thử thành các ca kiểm thử cụ thể, rồi tính kết quả mong đợi của từng ca kiểm thử, và tiến hành kiểm tra với phần mềm. Việc thực hiện kiểm thử cần được tự động hóa tối đa khi có thể, vì một hệ thống phần mềm cần được kiểm thử rất nhiều lần. Ngay cả khi sản phẩm hoàn tất, trong tương lai chúng ta vẫn sẽ cần thực hiện kiểm thử nhiều lần nữa, khi phần mềm tiến hóa, thay đổi theo những yêu cầu mới, hay các lỗi được phát hiện và phần mềm cần phải sửa.

5.1.3 Lựa chọn phương pháp phù hợp

Trong các mục tiếp theo của chương này, chúng ta sẽ tìm hiểu một số kỹ thuật kiểm thử chức năng thích hợp cho các loại đặc tả chương trình khác nhau. Với một đặc tả có thể có một số kỹ thuật phù hợp để xây dựng các ca kiểm thử và một số kỹ thuật sẽ rất khó áp dụng hoặc kết quả sẽ không thỏa mãn. Một số kỹ thuật có thể cho kết quả như nhau với một số đặc tả. Một số kỹ thuật lại bổ sung cho nhau nên cần kết hợp chúng. Ví dụ như khi mỗi kỹ thuật lại áp dụng tốt cho những khía cạnh khác nhau của cùng một đặc tả, hoặc ở các giai đoạn khác nhau của việc sinh ca kiểm thử.

Việc lựa chọn kỹ thuật để tạo ra các ca kiểm thử phụ thuộc vào các yếu tố: tính tự nhiên của đặc tả, dạng của đặc tả, kinh nghiệm của người thiết kế kiểm thử, cấu trúc của tổ chức, công cụ sẵn có, ngân sách và các ràng buộc về chất lượng, chi phí thiết kế và cài đặt các hệ thống hỗ trợ.

Tính tự nhiên và dạng của đặc tả: Các kỹ thuật kiểm thử khai thác các đặc điểm khác nhau của đặc tả. Ví dụ, khi biến đầu vào có một số ràng buộc thì có thể kỹ thuật kiểm thử lớp tương đương

(sẽ được giới thiệu chi tiết trong mục 5.3) sẽ phù hợp. Nếu có sự chuyển dịch trong các trạng thái của hệ thống thì phương pháp kiểm thử dựa trên máy trạng thái có thể nên áp dụng. Hay khi các đầu vào thay đổi với kích thước không xác định được thì phương pháp dùng ngữ pháp có thể phù hợp.

Kinh nghiệm của người thiết kế kiểm thử: Kinh nghiệm của người kiểm thử nhiều khi sẽ ảnh hưởng đến kỹ thuật kiểm thử nào được sử dụng. Ví dụ một người thiết kế kiểm thử giỏi về bảng quyết định sẽ thích kỹ thuật đó khi một kỹ thuật khác cũng có thể áp dụng được.

Công cụ: Một số kỹ thuật đòi hỏi sử dụng các công cụ mà đôi khi phải mua nên chi phí cũng là một yếu tố ảnh hưởng đến việc lựa chọn kỹ thuật kiểm thử. Ví dụ đặc tả sử dụng UML và có công cụ thương mại sinh ca kiểm thử dựa trên một số biểu đồ UML thì công cụ đóng vai trò quan trọng ảnh hưởng đến phương pháp thực hiện kiểm thử, nếu ngân sách cho kiểm thử cho phép. Một số đơn vị làm phần mềm chuyên nghiệp đôi khi phải xây dựng các công cụ này để phục vụ cho đơn vị, hoặc đôi khi cho từng dự án cụ thể.

Ngân sách và ràng buộc chất lượng: Ngân sách và ràng buộc chất lượng có thể dẫn đến sự lựa chọn khác nhau về phương pháp kiểm thử. Ví dụ nếu yêu cầu là nhanh, kiểm thử tự động, nhưng không cần độ tin cậy cao thì kiểm thử ngẫu nhiên là phù hợp. Ngược lại, yêu cầu độ tin cậy phải rất cao thì chúng ta phải sử dụng các phương pháp tinh vi để kiểm thử một cách kỹ lưỡng nhất có thể. Khi chọn một phương pháp, điều quan trọng là phải đánh giá tất cả các yếu tố và chi phí liên quan.

Chi phí nền tảng kiểm thử: Mỗi đặc tả ca kiểm thử sẽ phải được cụ thể hóa thành một ca kiểm thử và được chạy đi chạy lại nhiều lần trong suốt vòng đời của sản phẩm phần mềm. Với mỗi lần chạy, một ca kiểm thử bao gồm việc phải đưa các đầu vào cho chương trình, thực hiện chương trình, và so sánh kết quả đầu ra

thực tế với đầu ra mong đợi trong ca kiểm thử này. Nếu chúng ta tự động hóa được tất cả các công việc này thì phương pháp tổ hợp sinh ra tất cả các tổ hợp có thể áp dụng. Tuy nhiên nếu một số bước phải thực hiện bằng tay, hoặc chúng ta phải viết mã cho từng đặc tả kiểm thử này thì có lẽ chúng ta phải chọn phương pháp khác để khả thi về mặt thời gian và công sức.

5.2 Kiểm thử giá trị biên

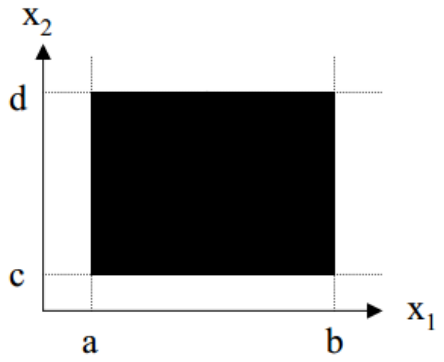
Kiểm thử giá trị biên (boundary value testing) là một trong những kỹ thuật được áp dụng phổ biến nhất trong cách tiếp cận kiểm thử chức năng (kiểm thử hộp đen). Trong thực tế, các lỗi hay xảy ra ở các giá trị biên hoặc cận biên của các biến đầu vào của chương trình cần kiểm thử. Kỹ thuật kiểm thử giá trị biên được đề xuất nhằm phát hiện những lỗi này. Chúng ta sẽ coi một chương trình là một hàm toán học với đầu vào của chương trình tương ứng với các tham số của hàm và đầu ra của chương trình là giá trị trả về của hàm. Vì hàm toán học là ánh xạ từ miền xác định của hàm đến miền giá trị của hàm, chúng ta sẽ tập trung vào các giá trị biên và cận biên của hai miền đầu vào và đầu ra này của hàm để xây dựng các ca kiểm thử. Khi chúng ta dùng biên đầu ra tức là chúng ta cho các kết quả mong đợi nằm ở trên biên và cận biên đầu ra.

5.2.1 Giá trị biên

Giả sử $y = f(x_1, x_2)$ với $x_1, x_2, y \in \mathbb{N}$ là một hàm toán học của một chương trình. Khi đó thông thường x_1 và x_2 có miền xác định thể hiện bằng các biên. Ví dụ:

$$a \leq x_1 \leq b \text{ và } c \leq x_2 \leq d$$

trong đó a, b, c, d là các hằng số nào đó. Phần tô đậm trong hình 5.2 thể hiện miền xác định của hai biến này.

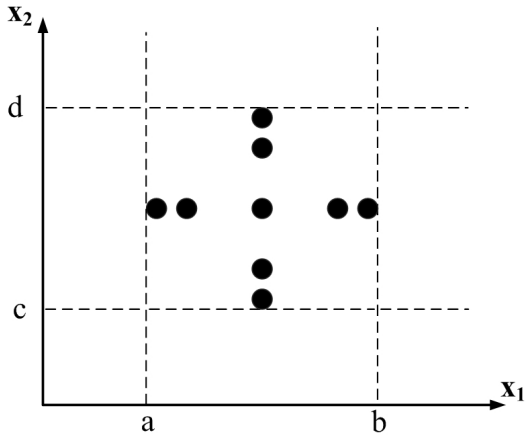


Hình 5.2: Miền xác định của hàm hai biến.

Ý tưởng kiểm thử giá trị biên xuất phát từ quan sát nhiều lỗi xảy ra với các giá trị biên này, khi chúng ta không kiểm tra khoảng giá trị hợp lệ của dữ liệu được nhập vào, hay do lỗi lập trình hoặc đặc tả làm các biểu thức điều kiện không chính xác. Ví dụ, đúng ra phải là dấu <= nhưng người lập trình hoặc đặc tả lại chỉ viết < hoặc ngược lại. Kiểm thử với các giá trị biên giúp chúng ta phát hiện các lỗi này. Trong nhiều trường hợp các biên này là ẩn, không được viết rõ ra trong yêu cầu nên lập trình viên dễ mắc lỗi không kiểm tra các giá trị đầu vào, hoặc không kiểm tra kết quả có hợp lệ không trước khi trả về.

Để tăng khả năng phát hiện lỗi, kiểm thử giá trị biên thường lấy năm ca kiểm thử cho mỗi biến là các giá trị: cực đại, cực tiểu, các giá trị cạnh chúng trong miền xác định (gọi là cận biên hoặc cạnh biên), và một giá trị ở giữa miền xác định đại diện cho giá trị thông thường. Chúng ta đánh chỉ số cho chúng lần lượt là max, min, $min+$, $max-$ và nom. Hình 5.3 minh họa các giá trị này.

Thông thường lỗi chương trình xảy ra ngay khi có một sai sót trong phần mềm, chứ không cần kết hợp của nhiều sai sót mới gây ra lỗi. Khi đó các ca kiểm thử theo phương pháp kiểm thử giá trị biên được xây dựng bằng cách lấy một bộ giá trị nom của các biến, rồi lần lượt thay mỗi giá trị đó của từng biến bằng giá trị biên và



Hình 5.3: Các ca kiểm thử giá trị biên cho một hàm hai biến.

cận biên để tạo ra ca kiểm thử mới. Ví dụ với hàm f trên ta có bộ kiểm thử sau, thể hiện trên Hình 5.3.

$$\{(x_{1nom}, x_{2nom}),$$
$$(x_{1min}, x_{2nom}), (x_{1min+}, x_{2nom}),$$
$$(x_{1max-}, x_{2nom}), (x_{1max}, x_{2nom}),$$
$$(x_{1nom}, x_{2min}), (x_{1nom}, x_{2min+}),$$
$$(x_{1nom}, x_{2max-}), (x_{1nom}, x_{2max})\}$$

Tổng quát hóa kiểm thử giá trị biên cho hàm n biến số và mỗi biến có các giá trị biên và cận biên khác nhau ta có thể dễ thấy sẽ có $1 + 4n$ ca kiểm thử vì xuất phát từ một ca kiểm thử gồm các giá trị trung bình của các biến, ta thay nó bằng bốn giá trị biên và cận biên: min, $min+$, max, và $max-$. Với n biến sẽ tạo thêm $4n$ bộ kiểm thử. Do đó tổng số ca kiểm thử là $1 + 4n$. Tuy nhiên tùy theo miền xác định của biến mà số lượng này thực tế có thể ít hơn. Ví dụ biến nguyên thuộc khoảng $[1, 2]$ thì không có cận biên và không có giá trị ở giữa. Với cách tạo bộ kiểm thử giá trị biên này, mỗi giá trị biên và cận biên xuất hiện một lần với các giá trị trung bình của các biến còn lại, chứ không phải tổ hợp các bộ giá

trị của các biên. Tổ hợp của các biên được gọi là kiểm thử giá trị biên mạnh sẽ được thảo luận trong phần tiếp theo.

Trong thực tế, miền xác định của các biến không chỉ là các miền số, nên chúng ta cần có các vận dụng thích hợp để xác định các giá trị biên tinh tế hơn. Ví dụ trong chương trình NextDate biến tháng `month` có thể là số nguyên trong khoảng từ 1 đến 12. Nhưng không phải ngôn ngữ lập trình nào cũng cho phép khai báo miền xác định của biến trong khoảng này. Chúng ta cũng có thể dùng kiểu liệt kê (enumeration) để khai báo các tháng bằng tên `Januray, ..., December` và chúng ta vẫn có giá trị biên và cận biên. Tuy nhiên như trong ngôn ngữ lập trình Java chúng ta phải dùng biến kiểu nguyên `int` và không khai báo miền giá trị cho chúng được, thì chúng ta sẽ phải dùng các cận của kiểu làm biên. Trong Java, biên của số nguyên là `Interger.MIN_VALUE` và `Interger.MAX_VALUE`. Trong ngôn ngữ lập trình C, chúng ta cũng có các khoảng giá trị này.

Với biến kiểu `Boolean`, các giá trị của miền chỉ có TRUE và FALSE, chúng ta không có cận biên và giá trị ở giữa. Với kiểu xâu ký tự, mảng, hay danh sách, tập hợp (collection), chúng ta có thể dựa vào chiều dài/kích thước để làm các giá trị biên. Với các kiểu cấu trúc dữ liệu khác chúng ta có thể xây dựng các giá trị biên và cận biên dựa theo các thành phần của cấu trúc.

Ưu nhược điểm: Khi các biến của hàm là độc lập, không có quan hệ ràng buộc lẫn nhau, thì kiểm thử giá trị biên tỏ ra hiệu quả. Nhưng khi chúng có quan hệ phụ thuộc nào đó thì phương pháp này dễ tạo ra các ca kiểm thử không hợp lý, vì các giá trị cực đại, cực tiểu có thể không kết hợp với nhau để tạo thành ca kiểm thử hợp lệ.

Ví dụ, khi x_1 và x_2 của hàm f trên có các ràng buộc là $x_1 \in [1, 100]$, $x_2 \in [1, 100]$ and $x_1 + x_2 \leq 120$ thì một biến đạt cực đại sẽ không thể kết hợp với một biến khác ở giá trị trung bình. Nếu

chúng ta xuất phát từ ca kiểm thử của hai giá trị trung bình $(50, 50)$ thì khi thay giá trị đầu bằng giá trị cực đại sẽ tạo ra ca kiểm thử $(100, 50)$. Ca kiểm thử này không thỏa mãn quan hệ $x_1 + x_2 \leq 120$.

Ví dụ khác với hàm NextDate thì biến ngày, tháng và năm có ràng buộc. Một số tháng có 31 ngày, một số lại có 30 ngày. Năm nhuận lại có ràng buộc với số ngày trong tháng hai. Nếu không xét sự phụ thuộc này kiểm thử giá trị biên không tạo ra được ca kiểm thử ngày 28 tháng 2 năm 2012, vì 28 không là biên và cận biên của khoảng ngày từ 1 đến 31.

Kiểm thử giá trị biên cũng không quan tâm đến tính chất, đặc trưng của hàm, đồng thời cũng không xét đến ngữ nghĩa của biến hay quan hệ giữa các biến. Nó chỉ máy móc lấy các giá trị biên, cận biên và trung bình để tổ hợp tạo ra các ca kiểm thử. Nên khi áp dụng chúng ta cần xác định sơ bộ các tính chất trên của hàm có không.

Ưu điểm của phương pháp này là đơn giản và có thể tự động hóa việc sinh các ca kiểm thử khi chúng ta đã xác định được các biên của các biến. Với nhiều kiểu dữ liệu cơ bản có sẵn chúng ta cũng có thể xây dựng sẵn các biên này để sử dụng.

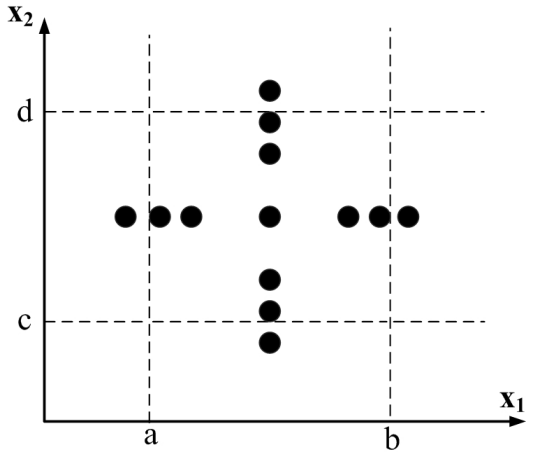
5.2.2 Một số dạng kiểm thử giá trị biên

5.2.2.1 Kiểm thử giá trị biên mạnh

Kiểm thử giá trị biên mạnh là mở rộng của kiểm thử giá trị biên bằng việc bổ sung các giá trị cận biên bên ngoài miền xác định. Ngoài 5 giá trị biên đã nêu ở phần trước chúng ta lấy thêm các giá trị cận biên ở ngoài miền xác định là $max+$ và $min-$ như trong Hình 5.4. Các ca kiểm thử này sẽ giúp ta kiểm tra chương trình với dữ liệu không hợp lệ, nằm ngoài khoảng mong đợi. Ví dụ khi nhập ngày 32/1/2013 chương trình cần có thông báo lỗi thích hợp.

Các ca kiểm thử mạnh này giúp chúng ta kiểm tra xem chương trình có xử lý các ngoại lệ hay kiểm tra biến đầu vào hay không.

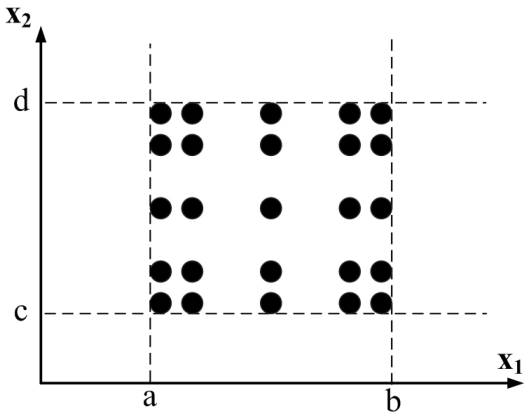
Với các ngôn ngữ lập trình không kiểm tra kiểu khi biên dịch hay dữ liệu được nhập vào bên ngoài, do người sử dụng đưa vào hoặc lấy từ hệ thống khác, việc này là rất cần thiết. Rất nhiều lập trình viên mới vào nghề thường xuyên không kiểm tra dữ liệu đầu vào nên chương trình chỉ chạy với dữ liệu nhập đúng như suy nghĩ của người lập trình. Hậu quả của việc này không chỉ đơn thuần là chương trình bị sai, mà thường gây vi phạm bộ nhớ, dẫn đến tắt chương trình (crash), và là một trong những lỗ hổng an ninh dễ bị khai thác.



Hình 5.4: Các ca kiểm thử mạnh cho hàm hai biến.

5.2.2.2 Kiểm thử giá trị biên tổ hợp

Ở trên, chúng ta chỉ tạo các ca kiểm thử với giá trị biên và cận biên cho từng biến. Nếu chúng ta mở rộng với hai hoặc với tất cả các biến đều được đẩy đến giá trị biên và cận biên thì chúng ta sẽ tạo ra được các ca kiểm thử giá trị biên tổ hợp. Tức là từ bộ giá trị 5 phần tử min, $min+$, nom, $max-$ và max của mỗi biến ta lấy tích Đề-các (Cartesian) của chúng để tạo ra các ca kiểm thử. Hình 5.5 minh họa các ca kiểm thử giá trị biên tổ hợp của hàm hai biến.



Hình 5.5: Các ca kiểm thử biên tổ hợp của hàm hai biến.

Có thể thấy cách tổ hợp các biên và cận biên này sẽ kiểm tra kỹ hơn kiểm thử giá trị biên thông thường. Tuy nhiên số ca kiểm thử theo cách này tăng lên đáng kể, lên đến 5^n ca kiểm thử so với $4n + 1$ ca theo kiểm thử biên thông thường.

Tương tự như kiểm thử giá trị biên mạnh, ta có thể mở rộng kiểm thử biên tổ hợp với bộ 7 giá trị của kiểm thử giá trị biên mạnh. Chúng ta sẽ kiểm tra được kỹ hơn nhưng cũng mất nhiều công sức hơn, đến 7^n ca kiểm thử.

5.2.2.3 Kiểm thử các giá trị đặc biệt

Kiểm thử các giá trị đặc biệt cũng là một phương pháp phổ biến. Đây cũng là phương pháp trực quan nhất và không theo một khuôn dạng cụ thể nào. Dựa trên hiểu biết về bài toán và miền ứng dụng kết hợp với kinh nghiệm cá nhân, người kiểm thử đưa ra các giá trị kiểm thử. Do đó không có hướng dẫn cụ thể nào cho phương pháp này. Mức độ hiệu quả của phương pháp này phụ thuộc nhiều vào khả năng của người kiểm thử. Trên thực tế các đơn vị phát triển phần mềm vẫn áp dụng phương pháp này, vì nhiều khi nó giúp phát hiện lỗi nhanh, không tốn nhiều công sức.

Ví dụ, dựa trên hiểu biết về số ngày của các tháng trong một năm, chúng ta sẽ kiểm thử các ngày 28 tháng 2 và 29 tháng 2 ở cả năm nhuận và năm thường. Số lượng ca kiểm thử sẽ ít hơn nhiều so với kiểm thử giá trị biên và cũng hiệu quả trong việc kiểm tra lỗi. Đây cũng là tính chất "thủ công" của nghề kiểm thử phần mềm - "khéo tay" và có kinh nghiệm sẽ làm tốt hơn.

5.2.3 Ví dụ minh họa

5.2.3.1 Kiểm thử giá trị biên cho bài toán Tam giác

Bảng 5.1: Các ca kiểm thử các giá trị biên cho bài toán Tam giác

TT	a	b	c	Kết quả mong đợi
1	100	100	1	Tam giác cân
2	100	100	2	Tam giác cân
3	100	100	100	Tam giác đều
4	100	100	199	Tam giác cân
5	100	100	200	Không phải tam giác
6	100	1	100	Tam giác cân
7	100	2	100	Tam giác cân
8	100	100	100	Tam giác đều
9	100	199	100	Tam giác cân
10	100	200	100	Không phải tam giác
11	1	100	100	Tam giác cân
12	2	100	100	Tam giác cân
13	100	100	100	Tam giác đều
14	199	100	100	Tam giác cân
15	200	100	100	Không phải tam giác

Chúng ta xem việc áp dụng kiểm thử giá trị biên với bài toán Tam giác (Triangle). Bài toán không nêu điều kiện cho cạnh của tam giác ngoài điều kiện là số nguyên. Nhưng chúng ta biết cận dưới của cạnh tam giác là 1. Và chúng ta giả sử cận trên là 200.

Bảng 5.1 chứa các ca kiểm thử các giá trị biên cho bài toán Tam giác như đã mô tả.

5.2.3.2 Kiểm thử giá trị biên cho NextDate

Như chúng ta thấy trong Bảng 5.2, phương pháp kết hợp các giá trị biên có thể tạo ra ngày không hợp lệ - tức là ngày không có thực trên thực tế, không thuộc miền xác định của hàm NextDate. Hơn nữa bảng này đúng ra có đến 125 ca kiểm thử, nhưng ở đây chúng ta chỉ liệt kê một số ca kiểm thử trong số đó.

Bảng 5.2: Một số ca kiểm thử biên tổ hợp cho hàm NextDate

TT	Tháng	Ngày	Năm	Kết quả mong đợi
1	1	1	1812	2/1/1812
2	1	1	1813	2/1/1813
3	1	1	1912	2/1/1912
4	1	1	2011	2/1/2011
5	1	1	2012	2/1/2012
6	1	2	1812	3/1/1812
7	1	2	1813	3/1/1813
8	1	2	1912	3/1/1912
9	1	2	2011	3/1/2011
10	1	2	2012	3/1/2012
11	1	15	1812	16/1/1812
12	1	15	1813	16/1/1813
13	1	15	1912	16/1/1912
14	1	15	2011	16/1/2011
15	1	15	2012	16/1/2012
16	1	30	1812	31/1/1812

5.2.4 Kinh nghiệm áp dụng

Kiểm thử giá trị biên là phương pháp "thô" nhất trong các phương pháp kiểm thử chức năng. Phương pháp này nên áp dụng cho các hàm có các biến đầu vào độc lập, không phụ thuộc vào nhau. Khi

chúng ta có giả định một khiếm khuyết trong chương trình sẽ gây ra lỗi ngay thì chúng ta sử dụng kiểm thử biên thông thường. Trái lại, khi lỗi xuất hiện với nhiều hơn một khiếm khuyết trong chương trình thì chúng ta cần sử dụng kiểm thử giá trị biên mạnh. Các phương pháp này đều đơn giản, dễ dàng tự động hóa việc sinh và chạy các ca kiểm thử.

Trong phần này, chúng ta chủ yếu tập trung vào biên của biến đầu vào. Chúng ta cũng có thể mở rộng ý tưởng về phân tích giá trị biên cho biến đầu ra, tức là kết quả của hàm, sao cho các kết quả này có thể nằm ở biên hoặc cận biên. Ngoài ra, trong chương trình máy tính còn có các biến cục bộ, hay các biến toàn cục khác. Chúng ta có thể vận dụng mở rộng ý tưởng trên cho các biến này để có thể phát hiện thêm các lỗi tinh vi khác trong chương trình.

5.3 Kiểm thử lớp tương đương

5.3.1 Lớp tương đương

Ý tưởng của kiểm thử lớp tương đương là ta chia (phân hoạch) miền dữ liệu đầu vào thành các miền con sao cho các giá trị trong mỗi miền con có tác động tương tự nhau với chương trình cần kiểm thử. Việc phân hoạch này còn phải thỏa mãn tính chất là không có miền con nào là tập rỗng, hợp tất cả các miền con là miền dữ liệu đầu vào ban đầu và giao giữa hai miền con (khác nhau) bất kỳ là rỗng. Khi đã phân hoạch miền dữ liệu đầu vào thành các miền con như vậy, chúng ta chỉ cần lấy ngẫu nhiên một giá trị bất kỳ trong mỗi miền con để xây dựng bộ kiểm thử. Phương pháp kiểm thử lớp tương đương này cho ta cảm giác đã kiểm thử đầy đủ hệ thống đồng thời cũng không có các ca kiểm thử trùng lặp. Chúng ta cũng có thể áp dụng lớp tương đương cho miền đầu ra.

Kiểm thử lớp tương đương là phương pháp chia miền dữ liệu kiểm thử thành các miền con sao cho dữ liệu trong mỗi miền con

có cùng tính chất đối với chương trình, có nghĩa là các ca kiểm thử của một miền con sẽ cùng gây lỗi cho chương trình, hay cùng cho kết quả đúng, hay cùng cho kết quả sai tương tự nhau. Sau khi chia miền dữ liệu của chương trình thành các miền con tương đương, chúng ta chỉ cần chọn một phần tử đại diện của mỗi miền con này làm bộ dữ liệu kiểm thử. Các miền con này chính là các lớp tương đương.

Ví dụ, trong bài toán Tam giác như đã giới thiệu trong chương 2, chúng ta lấy một lớp tương đương là tất cả các tam giác đều. Khi đó ta chọn một ca kiểm thử là (5, 5, 5) để thực hiện thì sau đó ta có thể kết luận tương tự về kết quả kiểm thử cho các ca kiểm thử khác cùng lớp như (6, 6, 6) hay (100, 100, 100). Bản chất ở đây là chúng ta cho rằng chương trình sẽ thực hiện cùng các lệnh giống nhau với các ca kiểm thử cùng một lớp tương đương, nên hành vi của chương trình là tương tự nhau – cùng có lỗi hay ra kết quả sai hay cùng ra các kết quả đúng. Bây giờ, vấn đề chính là làm sao xác định được các lớp tương đương của miền dữ liệu đầu vào. Việc này có thể thực hiện bằng cách phân tích đặc tả của chương trình đặc biệt là các thông tin về miền dữ liệu của các biến đầu vào và mối quan hệ giữa các đầu vào và đầu ra của chương trình này.

Kiểm thử lớp tương đương được chia thành một số loại. Chúng ta sẽ sử dụng chương trình minh họa sau để giải thích các loại kiểm thử lớp tương đương dễ dàng hơn.

Giả sử chương trình của chúng ta là một hàm của ba biến a, b, c và không gian đầu vào là ba tập A, B, C với các miền tương đương cho mỗi không gian đầu vào như sau:

$$A = A_1 \cup A_2 \cup A_3$$

$$B = B_1 \cup B_2 \cup B_3 \cup B_4$$

$$C = C_1 \cup C_2$$

Chúng ta ký hiệu phần tử của các miền tương đương trên bằng chữ thường với cùng chỉ số trong các phần sau. Ví dụ $a_1 \in A_1$, $b_3 \in B_3$ và $c_2 \in C_2$.

5.3.2 Phân loại kiểm thử lớp tương đương

5.3.2.1 Kiểm thử lớp tương đương yếu

Kiểm thử lớp tương đương yếu chỉ yêu cầu mỗi lớp (phân hoạch) tương đương có ít nhất một phần tử xuất hiện trong một ca kiểm thử nào đó. Với chương trình tam giác như minh họa trên thì phải có một giá trị thuộc A_1 trong một ca kiểm thử nào đó. Tương tự với các tập con khác. Ta có bộ kiểm thử thỏa mãn kiểm thử lớp tương đương yếu như trong Bảng 5.3.

Bảng 5.3: Các ca kiểm thử lớp tương đương cho bài toán Tam giác

TT	a	b	c
1	a_1	b_1	c_1
2	a_2	b_2	c_2
3	a_3	b_3	c_1
4	a_1	b_4	c_2

Bộ kiểm thử trên sử dụng một giá trị từ mỗi lớp tương đương. Để kiểm tra chúng ta chỉ cần để ý đến các cột, ví dụ cột a có xuất hiện cả a_1, a_2, a_3 của ba miền con của A, như vậy là đủ. Chúng ta có thể lập quy tắc để tự động xây dựng bộ kiểm thử lớp tương đương yếu này dễ dàng và cũng có thể dễ nhận thấy là số ca kiểm thử tối thiểu chính là số lớp tương đương lớn nhất của các miền đầu vào, trong ví dụ này là B.

5.3.2.2 Kiểm thử lớp tương đương mạnh

Kiểm thử lớp tương đương mạnh sẽ kết hợp các tổ hợp có thể của các lớp tương đương. Với ví dụ của hàm ba biến a, b, c và các phân

lớp tương đương cho mỗi không gian đầu vào A, B, C tương ứng như đã mô tả trên, chúng ta sẽ có $3 * 4 * 2 = 24$ tổ hợp tương ứng với 24 ca kiểm thử. Chi tiết của các ca kiểm thử này được mô tả như bảng 5.4.

Bảng 5.4: Các ca kiểm thử lớp tương đương mạnh cho hàm Tam giác

TT	a	b	c
1	a_1	b_1	c_1
2	a_1	b_1	c_2
3	a_1	b_2	c_1
4	a_1	b_2	c_2
5	a_1	b_3	c_1
6	a_1	b_3	c_2
7	a_1	b_4	c_1
8	a_1	b_4	c_2
9	a_2	b_1	c_1
10	a_2	b_1	c_2
11	a_2	b_2	c_1
12	a_2	b_2	c_2
13	a_2	b_3	c_1
14	a_2	b_3	c_2
15	a_2	b_4	c_1
16	a_2	b_4	c_2
17	a_3	b_1	c_1
18	a_3	b_1	c_2
19	a_3	b_2	c_1
20	a_3	b_2	c_2
21	a_3	b_3	c_1
22	a_3	b_3	c_2
23	a_3	b_4	c_1
24	a_3	b_4	c_2

Việc xác định các ca kiểm thử ở bảng trên tương tự như việc xây dựng bảng giá trị chân lý cho mỗi công thức lôgic mệnh đề. Vì vậy, việc xác định các ca kiểm thử theo cách này cũng dễ dàng tự

động hóa. Tích Đề-các này cho ta một bộ kiểm thử đầy đủ, có tất cả các khả năng kết hợp của các miền tương đương của các miền đầu vào của hàm cần kiểm thử.

Ở hai phần trên chúng ta chỉ chú trọng các miền tương đương của biến đầu vào. Chúng ta có thể áp dụng các phương pháp này cho miền đầu ra. Việc này thường khó hơn vì chúng ta phải tính ngược bộ giá trị kiểm thử ở miền đầu vào để chúng tạo ra giá trị ở miền đầu ra mong muốn. Với chương trình có nhiều biến đầu ra, chúng ta cũng có hai dạng mạnh và yếu tương tự với đầu vào.

5.3.2.3 Kiểm thử lớp tương đương đơn giản

Kiểm thử lớp tương đương đơn giản chỉ phân chia một lớp gồm các giá trị hợp lệ và các miền giá trị không hợp lệ. Ví dụ nếu A là khoảng 1 đến 200 các số nguyên thì miền hợp lệ là các giá trị từ 1 đến 200, miền không hợp lệ gồm hai miền. Miền thứ nhất là các giá trị từ 0 trở xuống. Miền còn lại là các giá trị từ 201 trở lên. Để đơn giản ta không xét miền các giá trị không phải là số nguyên. Sau khi đã có các lớp tương đương theo cách đơn giản này chiến lược kiểm thử có thể áp dụng tương tự kiểm thử tương đương mạnh và yếu.

Có hai vấn đề với kiểm thử lớp tương đương đơn giản. Thứ nhất là thông thường khi mô tả bài toán thì tài liệu không nói rõ các trường hợp khi đầu vào không hợp lệ thì kết quả là gì. Nếu có nói thì cũng thường chỉ đề cập đến một số trường hợp đã biết mà không xét đến mọi khả năng có thể. Ví dụ chúng ta viết chương trình giải phương trình bậc hai với ba hệ số nguyên a, b, c được nhập vào từ bàn phím. Rất nhiều lập trình viên sẽ chuyển ngay xâu ký tự người sử dụng nhập vào thành các số và gán cho các hệ số này. Nhưng nếu người sử dụng nhập một xâu chữ thì chương trình không kiểm tra để có thông báo phù hợp cho người sử dụng. Khi thiết kế kiểm thử chúng ta phải bổ sung một loạt các trường hợp mà hệ thống phải đưa ra kết quả bổ sung hoặc các thông báo lỗi phù hợp cho các

trường hợp này. Thứ hai là một số ngôn ngữ lập trình có hệ thống kiểm tra kiểu mạnh như Java, C# thì không thể đưa giá trị sai kiểu vào được nên chúng ta không cần các ca kiểm thử cho giá trị sai kiểu, nhưng vẫn cần kiểm tra dữ liệu do người sử dụng nhập vào, và chỉ kiểm tra dữ liệu không hợp lệ cùng kiểu. Tuy nhiên trong các ngôn ngữ lập trình script như Javascript, PHP, Python, không có kiểm tra kiểu mạnh nên các lỗi loại này khá phổ biến do đó ta cần xét kỹ việc chia miền đầu vào thành các lớp tương đương.

5.3.3 Ví dụ minh họa

5.3.3.1 Kiểm thử lớp tương đương cho chương trình Tam giác

Chúng ta áp dụng kiểm thử lớp tương đương cho bài toán tam giác, nhưng chúng ta áp dụng lớp tương đương cho đầu ra thay vì đầu vào. Ở đầu ra chúng ta thấy chương trình có thể in ra bốn khả năng: Không phải tam giác, Tam giác thường, Tam giác cân và Tam giác đều. Dựa trên các lớp đầu ra này, chúng ta có thể xác định bộ kiểm thử như bảng 5.5.

Bảng 5.5: Các ca kiểm thử lớp tương đương cho hàm Tam giác

TT	a	b	c	Kết quả mong đợi
1	5	5	5	Tam giác đều
2	2	2	3	Tam giác cân
3	3	4	5	Tam giác không cân
4	4	1	2	Không là tam giác

Nếu chúng ta kiểm thử lớp đương đương với miền đầu vào, chúng ta có một bộ kiểm thử phong phú hơn. Với bộ ba số tự nhiên a, b, c, có thể hai trong chúng hoặc cả ba bằng nhau hoặc tất cả khác nhau. Chúng ta có năm (5) lớp tương đương đầu vào như sau:

$$D1 = \{\langle a, b, c\rangle \mid \ a = b = c\}$$

$$D2 = \{\langle a, b, c\rangle \mid \ a = b, a \neq c\}$$

$$D3 = \{\langle a, b, c\rangle \mid \ a = c, a \neq b\}$$
$$D4 = \{\langle a, b, c\rangle \mid \ b = c, a \neq b\}$$
$$D5 = \{\langle a, b, c\rangle \mid \ a \neq b, a \neq c, b \neq c\}$$

Nếu chúng ta dựa trên hiểu biết về tam giác ta là tổng hai cạnh phải lớn hơn cạnh còn lại thì ta có thể phân tiếp thành các lớp tương đương mịn hơn. Ví dụ $<1, 4, 1>$ thuộc $D3$ nhưng không là tam giác.

$$D6 = \{\langle a, b, c\rangle \ \mid a > b + c\}$$
$$D7 = \{\langle a, b, c\rangle \ \mid b > a + c\}$$
$$D8 = \{\langle a, b, c\rangle \ \mid c > a + b\}$$

Từ đây, chúng ta có thể áp dụng tiếp kiểm thử tương đương mạnh để tìm ra các ca kiểm thử. Ở ví dụ này, chúng ta không áp dụng kiểm thử lớp tương đương đơn giản vì chia thành miền giá trị các số dương và không dương sẽ không kiểm thử được các tính chất của tam giác.

5.3.3.2 Kiểm thử lớp tương đương cho hàm NextDate

Hàm NextDate minh họa rất rõ tính thủ công của việc xác định quan hệ tương đương. Qua đây ta cũng thấy rõ khác biệt giữa các loại kiểm thử lớp tương đương: đơn giản, yếu và mạnh. NextDate là một hàm gồm ba (3) biến, ngày, tháng và năm, và chúng có các khoảng xác định như sau:

$$1 <= day <= 31$$

$$1 <= month <= 12$$

$$1812 <= year <= 2012$$

Theo loại kiểm thử lớp tương đương đơn giản, các lớp tương đương hợp lệ là:

$$D1 = \{day \ \mid 1 <= day <= 31\}$$

$$M1 = \{month \mid 1 <= month <= 12\}$$

$$Y1 = \{year \mid 1812 <= year <= 2012\}$$

Các lớp tương đương không hợp lệ là:

$$D2 = \{day \mid day < 1\}$$

$$D3 = \{day \mid day > 31\}$$

$$M2 = \{month \mid month < 1\}$$

$$M3 = \{month \mid month > 12\}$$

$$Y2 = \{year \mid year < 1812\}$$

$$Y3 = \{year \mid year > 2012\}$$

Dựa trên các lớp tương đương này ta có thể có bộ kiểm thử như Bảng 5.6 (các giá trị hợp lệ được lấy giá trị giữa khoảng).

Bảng 5.6: Các ca kiểm thử biên tổ hợp hàm NextDate

TT	Tháng	Ngày	Năm	Kết quả mong đợi
1	6	15	1912	16/6/1912
2	-1	15	1912	ngày không hợp lệ
3	13	15	1912	ngày không hợp lệ
4	6	-1	1912	ngày không hợp lệ
5	6	32	1912	ngày không hợp lệ
6	6	15	1811	ngày không hợp lệ
7	6	15	2013	ngày không hợp lệ

5.3.3.3 Kiểm thử tương đương yếu cho NextDate

Các ca kiểm thử lớp tương đương yếu của hàm NextDate được liệt kê trong Bảng 5.7. Chúng ta có thể thấy phương pháp này thực sự yếu vì nó chỉ đưa ra được một vài ca kiểm thử không đủ đảm bảo độ tin cậy cho chương trình.

Bảng 5.7: Các ca kiểm thử biên tổ hợp hàm NextDate

TT	Tháng	Ngày	Năm	Kết quả mong đợi
1	6	14	1900	15/6/1900
2	7	29	1912	30/7/1912
3	2	30	1913	Không hợp lệ
4	6	31	1900	Không hợp lệ

5.3.3.4 Kiểm thử tương đương mạnh cho NextDate

Sử dụng các lớp tương đương như trên, chúng ta có các ca kiểm thử lớp tương đương mạnh ở bảng dưới. Chúng ta vẫn chưa có một bộ kiểm thử tốt nhất vì thiếu ca kiểm thử ngày 28 tháng Hai, nhưng 36 ca kiểm thử lớp tương đương mạnh này đã tốt hơn rất nhiều so với phương pháp trước.

5.3.4 Kinh nghiệm áp dụng

Từ các ví dụ trên, chúng ta có một số quan sát và định hướng sử dụng kiểm thử lớp tương đương như sau.

- Kiểm thử lớp tương đương đơn giản kém hiệu quả hơn so với kiểm thử lớp tương đương yếu, kiểm thử lớp tương đương yếu kém hiệu quả hơn so với kiểm thử lớp tương đương mạnh.
- Chúng ta có thể sử dụng kiểm thử tương đương đơn giản khi ngôn ngữ lập trình không có cơ chế kiểm tra kiểu mạnh.
- Nếu cần kiểm tra ngoại lệ chúng ta nên mở rộng kiểm thử lớp tương đương với các lớp giá trị ngoài miền xác định.
- Kiểm thử lớp tương đương phù hợp với dữ liệu đầu vào có miền giá trị là khoảng và hữu hạn.
- Kiểm thử lớp tương đương kết hợp với kiểm thử giá trị biên sẽ tốt hơn.

Bảng 5.8: Một phần ca kiểm thử lớp tương đương mạnh cho NextDate

TT	Tháng	Ngày	Năm	Kết quả mong đợi
1	6	14	1900	15/6/1900
2	6	14	1912	15/6/1912
3	6	14	1913	15/6/1913
4	6	29	1900	30/6/1900
5	6	29	1912	30/6/1912
6	6	29	1913	30/6/1913
7	6	30	1900	1/7/1900
8	6	30	1912	1/7/1912
9	6	30	1913	1/7/1913
10	6	31	1900	Không hợp lệ
11	6	31	1912	Không hợp lệ
12	6	31	1913	Không hợp lệ
13	7	14	1900	15/7/1900
14	7	14	1912	15/7/1912
15	7	14	1913	15/7/1913
16	7	29	1900	30/7/1900
17	7	29	1912	30/7/1912
18	7	29	1913	30/7/1913
19	7	30	1900	31/7/1900
20	7	30	1912	31/7/1912

- Nên dùng kiểm thử lớp tương đương với các chương trình phức tạp.

- Nên dùng kiểm thử lớp tương đương mạnh khi các biến là độc lập. Khi các biến phụ thuộc nhau thì dễ tạo ra các ca kiểm thử vô lý. Kiểm thử bằng bảng quyết định sẽ được sử dụng cho trường hợp này.

- Cần một số lần thử để tìm xác định các lớp tương đương đúng, như trong ví dụ NextDate. Trong trường hợp khác, ở đó là quan hệ tương đương rõ ràng và tự nhiên. Khi các lớp

tương đương không hiển nhiên và dễ thấy thì ta có thể áp dụng chiến thuật thử và đoán để xác định dần.

- Trong kiểm thử lớp tương đương mạnh, nếu chúng ta phân hoạch các miền dữ liệu của các biến đầu vào thành các lớp tương đương trong đó có một số lớp tương đương là miền con chứa các giá trị không hợp lệ của mỗi biến, chúng ta không cần các ca kiểm thử có nhiều hơn một (1) giá trị không hợp lệ. Lý do của việc loại bỏ các ca kiểm thử này là vì chúng đã được kiểm thử bởi các ca kiểm thử có đúng một giá trị không hợp lệ. Với kinh nghiệm này, chúng ta có thể giảm đáng kể công sức cho việc sinh các ca kiểm thử cho kiểm thử lớp tương đương mạnh cũng như chi phí để thực thi chúng trên chương trình cần kiểm thử. Việc loại bỏ các ca kiểm thử này không hề ảnh hưởng đến khả năng phát hiện lỗi của tập các ca kiểm thử còn lại.

5.4 Kiểm thử bằng bảng quyết định

Kỹ thuật kiểm thử lớp tương đương và kiểm thử giá trị biên thích hợp cho các hàm có các biến đầu vào không có quan hệ ràng buộc với nhau. Kỹ thuật kiểm thử dựa trên bảng quyết định sẽ phù hợp cho các hàm có các hành vi khác nhau dựa trên tính chất của bộ giá trị của đầu vào. Nói cách khác, kỹ thuật này phù hợp với các hàm/chương trình có các biến đầu vào phụ thuộc lẫn nhau.

Kiểm thử dựa trên bảng quyết định là phương pháp chính xác nhất trong các kỹ thuật kiểm thử chức năng. Bảng quyết định là phương pháp hiệu quả để mô tả các sự kiện, hành vi sẽ xảy ra khi một số điều kiện thỏa mãn.

5.4.1 Bảng quyết định

Cấu trúc của một bảng quyết định chia thành bốn phần chính như trong Bảng 5.9, bao gồm:

- Các biểu thức điều kiện C_1, C_2, C_3;
- Giá trị điều kiện T, F, –;
- Các hành động A_1, A_2, A_3, A_4; và
- Giá trị hành động, có (xảy ra) hay không. Chúng ta ký hiệu X để chỉ hành động là có xảy ra ứng với các điều kiện tương ứng của cột.

Khi lập bảng quyết định, chúng ta thường tìm các điều kiện có thể xảy ra để xét các tổ hợp của chúng mà từ đó chúng ta sẽ xác định được các ca kiểm thử tương ứng cho các điều kiện được thỏa mãn. Các hành động xảy ra chính là kết quả mong đợi của ca kiểm thử đó.

Bảng quyết định với các giá trị điều kiện chỉ là T, F, và – được gọi là *bảng quyết định lôgic*. Chúng ta có thể mở rộng các giá trị này bằng các tập giá trị khác, ví dụ 1, 2, 3, 4, khi đó chúng ta có *bảng quyết định tổng quát*.

Bảng 5.9: Ví dụ về một bảng quyết định

		Quy tắc					
		1	2	3	4	5	6
Điều kiện	C_1	T	T	T	F	F	F
	C_2	T	T	F	T	F	F
	C_3	T	F	-	-	T	F
Hành động	A_1	X	X		X		
	A_2	X				X	
	A_3		X		X	X	
	A_4			X			X

Bảng 5.10 là một ví dụ đơn giản về một bảng quyết định để khắc phục sự cố máy in. Khi máy in có sự cố, chúng ta sẽ xem xét tình trạng dựa trên các điều kiện trong bảng là đúng (T) hay sai (F), từ đó xác định được cột duy nhất có các điều kiện thỏa mãn, và thực hiện các hành động khắc phục sự cố tương ứng.

Chú ý là ở đây thứ tự các điều kiện và thứ tự thực hiện hành động không quan trọng, nên chúng ta có thể đổi vị trí các hàng. Với các hành động cũng vậy, tuy nhiên tùy trường hợp chúng ta có thể làm mịn hơn bằng việc đánh số thứ tự hành động xảy ra thay cho dấu X để chỉ ra hành động nào cần làm trước. Với bảng quyết định tổng quát, các giá trị của điều kiện không chỉ nhận giá trị đúng (T) hoặc sai (F), khi đó ta cần tăng số cột để bao hết các tổ hợp có thể của các điều kiện.

Bảng 5.10: Bảng quyết định để khắc phục sự cố máy in

		1	2	3	4	5	6	7	8
Điều kiện	Máy in không in	T	T	T	T	F	F	F	F
	Đèn đỏ nhấp nháy	T	T	F	F	T	T	F	F
	Không nhận ra máy in	T	F	T	F	T	F	T	F
Hành động	Kiểm tra dây nguồn			X					
	Kiểm tra cáp máy in	X		X					
	Kiểm tra phần mềm in	X		X		X		X	
	Kiểm tra mực in	X	X			X	X		
	Kiểm tra kẹt giấy		X		X				

Kỹ thuật thực hiện: Để xác định các ca kiểm thử dựa trên bảng quyết định, chúng ta dịch các điều kiện thành các đầu vào và các hành động thành các đầu ra. Đôi khi các điều kiện sẽ xác định các lớp tương đương của đầu vào và các hành động tương ứng với các mô-đun xử lý chức năng đang kiểm thử. Do đó mỗi cột tương ứng với một ca kiểm thử. Vì tất cả các cột bao phủ toàn bộ các tổ hợp đầu vào nên chúng ta có một bộ kiểm thử đầy đủ.

Trên thực tế không phải tổ hợp đầu vào nào cũng là hợp lệ, do đó khi sử dụng bảng quyết định người ta thường bổ sung thêm

một giá trị đặc biệt "–" để đánh dấu các điều kiện không thể cùng xảy ra này. Các giá trị – (không quan tâm) có thể hiểu là luôn sai, không hợp lệ. Nếu các điều kiện chỉ là T và F ta có 2^n cột qui tắc. Mỗi giá trị "–" sẽ đại diện cho hai cột. Để dễ kiểm tra không sót cột nào ta có thể thêm hàng đếm "Số luật" như trong Bảng 5.11 và khi tổng hàng này bằng 2^n ta biết số cột qui tắc đã đủ.

Bảng 5.11: Bảng quyết định cho hàm Triangle

Điều kiện	1	2	3	4	5	6	7	8	9	10	11
c1: a<b+c?	F	T	T	T	T	T	T	T	T	T	T
c2: b<a+c?	-	F	T	T	T	T	T	T	T	T	T
c3: c<a+b?	-	-	F	T	T	T	T	T	T	T	T
c4: a = b?	-	-	-	T	T	T	T	F	F	F	F
c5: a = c?	-	-	-	T	T	F	F	T	T	F	F
c6: b = c?	-	-	-	T	F	T	F	T	F	T	F
Số luật	32	16	8	1	1	1	1	1	1	1	1
a1: Không là tam giác	X	X	X								
a2: Tam giác lệch											X
a3: Tam giác cân							X		X	X	
a4: Tam giác đều				X							
a5: Không hợp lệ					X	X		X			

5.4.2 Ví dụ minh họa

Kiểm thử bằng bảng quyết định cho hàm Triangle: Sử dụng bảng quyết định được mô tả ở Bảng 5.11, ta có 11 ca kiểm thử nhằm kiểm tra tính đúng đắn của hàm Triangle. Cụ thể, có ba (3) trường hợp không hợp lệ, ba (3) trường hợp không phải là tam giác, một (1) trường hợp tam giác đều, một (1) trường hợp tam giác thường và ba (3) trường hợp tam giác cân. Bảng 5.12 là kết quả chi tiết về các ca kiểm thử này. Nếu ta thêm điều kiện kiểm tra bất đẳng thức tam giác ta sẽ có thêm ba (3) ca kiểm thử nữa (trường hợp một cạnh có độ dài bằng tổng hai cạnh còn lại).

Kiểm thử bằng bảng quyết định cho hàm NextDate: Có nhiều cách xác định các điều kiện. Ví dụ chúng ta sẽ đặc tả ngày

Bảng 5.12: Ca kiểm thử bằng bảng quyết định cho hàm Triangle

TT	a	b	c	Kết quả mong đợi
1	4	1	2	Không phải tam giác
2	1	4	2	Không phải tam giác
3	1	2	4	Không phải tam giác
4	5	5	5	Tam giác đều
5	?	?	?	Không khả thi
6	?	?	?	Không khả thi
7	2	2	3	Tam giác cân
8	?	?	?	Không khả thi
9	2	3	2	Tam giác cân
10	3	2	2	Tam giác cân
11	3	4	5	Tam giác thường

và tháng trong năm và quy đổi về dạng của một năm nhuận hay một năm thông thường giống như trong lần thử đầu tiên, do đó năm 1900 sẽ không có gì đặc biệt. Các miền tương đương bây giờ như sau:

- M1 = { tháng | tháng có 30 ngày }
- M2 = { tháng | tháng có 31 ngày, trừ tháng 12 }
- M3 = { tháng | tháng 12 }
- M4 = { tháng | tháng 2 }

Ngày

- D1 = {ngày | $1 \leq$ ngày ≤ 27 }
- D2 = {ngày | ngày = 28 }
- D3 = {ngày | ngày = 29 }
- D4 = {ngày | ngày = 30 }

- D5 = {ngày | ngày = 31 }

Năm

- Y1 = {năm | năm nhuận }
- Y2 = {năm | năm thông thường }

Bảng 5.13: Ca kiểm thử bằng bảng quyết định cho hàm NextDate

TT	Tháng	Ngày	Năm	Kết quả mong đợi
1	4	15	1993	16/4/1993
2	4	28	1993	29/4/1993
3	4	29	1993	30/4/1993
4	4	30	1993	1/5/1993
5	4	31	1993	Không hợp lệ
6	1	15	1993	16/1/1993
7	1	28	1993	29/1/1993
8	1	29	1993	30/1/1993
9	1	30	1993	31/1/1993
10	1	31	1993	1/2/1993
11	12	15	1993	16/12/1993
12	12	28	1993	29/12/1993
13	12	29	1993	30/12/1993
14	12	30	1993	31/12/1993
15	12	31	1993	1/1/1994
16	2	15	1993	16/2/1993
17	2	28	1992	29/2/1992
18	2	28	1993	1/3/1993
19	2	29	1992	1/3/1992
20	2	29	1993	Không khả thi
21	2	30	1993	Không khả thi
22	2	31	1993	Không khả thi

Trong khi tích Đề-các sẽ tạo ra 40 bộ giá trị nếu áp dụng kiểm thử lớp tương đương mạnh, bảng quyết định được lập như Bảng 5.13 chỉ cần 22 bộ giá trị ứng với 22 ca kiểm thử. Có 22 quy tắc, so với

36 trong thử lần hai. Chúng ta có một bảng quyết định với 22 quy tắc. Năm quy tắc đầu tiên cho tháng có 30 ngày. Hai bộ tiếp theo (6-10 và 11-15) cho tháng có 31 ngày, với các tháng khác Tháng Mười Hai và với Tháng Mười Hai.

5.4.3 Kinh nghiệm áp dụng

So với các kỹ thuật kiểm thử khác, kiểm thử bằng bảng quyết định tốt hơn với một số bài toán (như NextDate) nhưng cũng kém hơn với một số bài toán (như Commission). Những bài toán phù hợp với bảng quyết định khi chương trình có nhiều lệnh rẽ nhánh (như Triangle) và các biến đầu vào có quan hệ với nhau (như NextDate). Khi sử dụng phương pháp này, chúng ta có thể tham khảo một số gợi ý sau:

- Nên dùng kỹ thuật bảng quyết định cho các ứng dụng có một trong các tính chất sau:
  - Chương trình có nhiều lệnh rẽ nhánh – nhiều khối If-Then-Else
  - Các biến đầu vào có quan hệ với nhau
  - Có các tính toán giữa các biến đầu vào
  - Có quan hệ nhân quả giữa đầu vào và đầu ra
  - Có độ phức tạp chu trình (Cyclomatic) [McC76a] cao
- Bảng quyết định không dễ áp dụng cho các bài toán lớn (với n điều kiện có 2^n quy tắc). Nên dùng dạng mở rộng và sử dụng đại số để đơn giản hóa bảng.
- Có thể cần một số lần thử và rút kinh nghiệm để dần dần lập được bảng tối ưu.

5.5 Kiểm thử tổ hợp

Thông thường, lỗi xảy ra trong các sản phẩm phần mềm do một điều kiện nào đó làm phát sinh và không liên quan gì đến các điều kiện khác. Ví dụ, khi nhập ngày sinh của một học sinh lớp bốn là 19/8/2013 vào danh sách lớp của năm học 2013-2014 thì thông báo lỗi cần xuất hiện vì học sinh này chưa được một tuổi nên không thể là học sinh lớp bốn. Tuy nhiên, cũng có những lỗi phần mềm chỉ xuất hiện do kết hợp của một số điều kiện. Để kiểm thử cẩn thận một sản phẩm phần mềm, chúng ta phải kiểm thử tất cả các tổ hợp của các điều kiện có thể. Tuy nhiên, thông thường số tổ hợp của tất cả các điều kiện này sẽ bùng nổ rất nhanh và sẽ rất tốn công để kiểm thử tất cả chúng. Ví dụ, giả sử hàm $y = f(x_1, x_2)$ với x_1 và x_2 chỉ nhận giá trị nguyên từ 1 đến 5. Chúng ta có $5 * 5 = 25$ tổ hợp phải kiểm thử là $(1, 1), (1, 2), .., (5, 5)$. Trong thực tế, miền giá trị của các biến đầu vào thường chưa rất nhiều giá trị. Vì vậy, số tổ hợp cần kiểm thử theo phương pháp này là rất lớn.

Tổng quát hóa với hàm $Y = P(X)$ trong đó $X = (x_1, .., x_n)$ và để đơn giản ta giả sử với mỗi biến x_i, $1 \leq i \leq n$ chỉ có k_i giá trị đáng chú ý. Nếu tính tất cả các tổ hợp của các giá trị đáng chú ý này chúng ta có $k_1 \times .. \times k_n$ ca kiểm thử.

Bằng cách không lấy hết tất cả các tổ hợp có thể mà chúng ta chỉ yêu cầu mọi tổ hợp của n biến bất kỳ đều xuất hiện trong một ca kiểm thử nào đó thì chúng ta có thể giảm đáng kể số ca kiểm thử so với tất cả các tổ hợp có thể. Kết quả nghiên cứu thực tế cho thấy tổ hợp đôi một đã có khả năng phát hiện khoảng 80% lỗi. Kiểm thử tổ hợp bốn đến sáu biến có thể phát hiện được 100% lỗi [KWG04]. Chúng ta sẽ xem xét một dạng phổ biến nhất với $n = 2$ gọi là kiểm thử đôi một.

5.5.1 Kiểm thử đôi một

Kiểm thử đôi một (pairwise testing) là một trường hợp đặc biệt của kiểm thử tất cả các tổ hợp. Kiểm thử đôi một chỉ yêu cầu mỗi cặp giá trị của (x_i, x_j), $1 \leq i \neq j \leq n$ xuất hiện trong một ca kiểm thử nào đó. Một ca kiểm thử thường có nhiều cặp giá trị này với các i, j khác nhau nên dễ thấy số lượng ca kiểm thử sẽ giảm đáng kể so với tổ hợp tất cả các ca kiểm thử.

Chúng ta xem một ví dụ. Giả sử chúng ta có hàm $y = g(x_1, x_2, x_3)$ và $x_1 \in \{True, False\}$, $x_2 \in \{0, 5\}$, và $x_3 \in \{a, b\}$. Tổ hợp tất cả các bộ giá trị có thể sẽ tạo ra 3^2 ca kiểm thử. Với kiểm thử đôi một, mỗi cặp giá trị của hai biến bất kỳ cần xuất hiện trong ít nhất một ca kiểm thử. Bảng 5.14 là bộ kiểm thử đôi một, tuy chỉ có bốn ca kiểm thử, nhưng chúng chứa tất cả các cặp giá trị có thể của các biến.

Bảng 5.14: Các ca kiểm thử đôi một cho hàm g

TT	x_1	x_2	x_3
TC1	$True$	0	a
TC2	$True$	5	b
TC3	$False$	0	b
TC4	$False$	5	a

5.5.2 Ma trận trực giao

Maldl [Rob85] là người đầu tiên đề xuất sử dụng ma trận trực giao để thiết kế các ca kiểm thử cặp đôi. Nhiều chiến lược kiểm thử đôi một cũng đã được trình bày trong các nghiên cứu [GOA05], cụ thể và đáng chú ý là thuật toán dựa trên mảng trực giao [Shr89] và thuật toán IPO [TL02].

Chúng ta hay xem xét mảng hai chiều trong Bảng 5.15. Bảng này có hai tính chất thú vị. Tất cả các cặp (1,1), (1,2), (2,1), and (2,2) đều xuất hiện trong hai cột bất kỳ. Tuy nhiên không phải tất

Bảng 5.15: Mảng trực giao $L_4(2^3)$

TT	1	2	3
1	1	1	1
2	1	2	2
3	2	1	2
4	2	2	1

cả các tổ hợp của 1 và 2 đều có trong bảng. Ví dụ (2, 2, 2) là tổ hợp hợp lệ nhưng nó không nằm trong bảng. Chỉ bốn trong tám tổ hợp nằm trong bảng. Trong bảng chỉ có 4 tổ hợp trên tổng số 8 tổ hợp có thể. Đây là một ví dụ về ma trận trực giao $L_4(2^3)$. Chỉ số 4 là số hàng của mảng, 3 là số cột của mảng, và 2 là số giá trị lớn nhất một phần tử của mảng có thể nhận.

Xem lại ví dụ của hàm g trên chúng ta thấy bảng này là một thể hiện của bảng trực giao tổng quát $L_4(2^3)$ trong Bảng 5.15. Các giá trị 1, 2 trong Bảng 5.15 đựa thay bằng các giá trị tương ứng của biến của hàm g. Ví dụ 1 tương ứng với True, 2 tương ứng với False. Chúng ta có thể kiểm tra là mỗi cặp giá trị bất kỳ của hai biến bất kỳ đều xuất hiện trong ít nhất một ca kiểm thử ở trong Bảng 5.14.

5.5.3 Kinh nghiệm áp dụng

Đến đây chúng ta đã hiểu về khái niệm ma trận trực giao và ứng dụng của nó trong việc sinh các ca kiểm thử cặp đôi. Sau đây là một qui trình thực hiện việc ứng dụng ma trận trực giao để sinh các ca kiểm thử theo phương pháp kiểm thử cặp đôi.

- **Bước 1.** Xác định tất cả các biến đầu vào, vì chúng sẽ tương ứng với các cột của ma trận trực giao.
- **Bước 2.** Xác định số lượng giá trị của mỗi biến có thể nhận để xác định được số lượng lớn nhất vì các giá trị của ma trận sẽ nằm trong khoảng này.

- **Bước 3.** Tìm ma trận trực giao thích hợp với số lượng hàng nhỏ nhất với hai số cột và giá trị của các phần tử ma trận đã xác định ở Bước 1 và Bước 2.
- **Bước 4.** Ánh xạ các biến với các cột của ma trận và giá trị của biến với các phần tử của ma trận.
- **Bước 5.** Kiểm tra còn phần tử nào trong ma trận chưa được ánh xạ không. Chọn các giá trị hợp lệ tùy ý cho những phần tử này.
- **Bước 6.** Chuyển các hàng của ma trận thành các ca kiểm thử.

Ở Bước 3 chúng ta có một bài toán đặt ra là làm thế nào để tìm được ma trận trực giao thích hợp. Có rất nhiều thuật toán và công cụ để sinh các ma trận trực giao theo các tham số chúng ta đã có ở Bước 1 và Bước 2 tại địa chỉ [1]. Ngoài ra cũng có nhiều ma trận trực giao được tính sẵn, chúng ta chỉ cần chọn và áp dụng.

5.6 Tổng kết

Tóm lại, kiểm thử chức năng cũng giống với nhiều hoạt động kỹ thuật khác đòi hỏi phải phân tích kỹ lưỡng các ưu nhược điểm để chọn phương pháp cân bằng phù hợp. Cách tiếp cận này cũng có các khó khăn và các vấn đề không lường hết được. Điều này đòi hỏi người thiết kế kiểm thử cần có kiến thức và kinh nghiệm mới có có thể tận dụng các ưu điểm và đương đầu với các vấn đề phát sinh. Kiểm thử chức năng không phải là bài toán chọn một phương pháp kiểm thử tối ưu mà bao gồm một loạt các hoạt động để tìm ra sự tổ hợp của một số mô hình và kỹ thuật để tạo ra một tập các ca kiểm thử thỏa mãn chi phí và ràng buộc chất lượng đặt ra. Sự cân bằng này cũng có thể mở rộng ra ngoài thiết kế kiểm thử sang

[1] http://www.pairwise.org/tools.asp

thiết kế phần mềm để thuận lợi cho việc kiểm thử. Thiết kế phần mềm thích hợp sẽ không chỉ giúp quá trình phát triển được tốt hơn mà còn giúp việc kiểm thử cũng thuận lợi để giúp tiết kiệm chi phí tổng thể của dự án phần mềm.

Trong cách tiếp cận kiểm thử chức năng, các phương pháp được sử dụng phổ biến hiện nay bao gồm: kiểm thử dựa trên phân tích giá trị biên, kiểm thử phân lớp tương đương và kiểm thử bằng bảng quyết định. Mỗi phương pháp đều có những ưu và nhược điểm riêng và hướng đến phát hiện các lỗi theo các triết lý riêng nên chúng không thể thay thế cho nhau. Một quy trình kiểm thử hiệu quả cần khai thác tối đa các ưu điểm của các kỹ thuật này bằng cách tích hợp chúng theo một tỷ lệ hợp lý nhằm phát huy tối đa khả năng phát hiện lỗi của các phương pháp kiểm thử này.

5.7 Bài tập

1. Áp dụng kiểm thử giá trị biên cho các chương trình ví dụ ở Chương 2.

2. Viết chương trình tính nghiệm của phương trình bậc hai $ax^2 + bx + c$. Hãy áp dụng các kỹ thuật kiểm thử của chương này để xây dựng bộ dữ liệu kiểm thử cho chương trình.

3. Cho hàm $z = f(x, y)$ là một hàm đã cài đặt. Hãy giả sử các kiểu dữ liệu cơ bản cho x, y trong một ngôn ngữ lập trình nào đó và viết chương trình để sinh các ca kiểm thử đơn vị bằng phương pháp kiểm thử giá trị biên.

4. Hãy áp dụng các kỹ thuật kiểm thử lớp tương đương cho chương trình giải phương trình bậc hai.

5. Bổ sung tam giác vuông vào bài toán Triangle, hãy xây dựng bảng quyết định. Chú ý có trường hợp tam giác vuông cân có cạnh là số nguyên.

6. Một hàm kiểm tra dữ liệu hợp lệ của hồ sơ học sinh gồm các trường Họ Tên, Ngày Sinh, Email và Giới tính (Nam, Nữ). Hãy cài đặt hàm này và áp dụng kiểm thử lớp tương đương để viết các kiểm thử đơn vị bằng Java/JUnit (hoặc ngôn ngữ/công cụ tương đương khác).

7. Hãy thử thêm hai cách phân chia lớp tương đương cho bài toán NextDate.

8. Xây dựng bảng quyết định cho PreviousDate, hàm ngược của bài toàn NextDate.

9. Xây dựng bảng quyết định cho bài toán giải phương trình bậc hai.

Chương 6

Kiểm thử dòng điều khiển

Trong chương này, chúng ta sẽ tìm hiểu chi tiết về phương pháp kiểm thử dòng điều khiển (control flow testing) nhằm phát hiện các lỗi tiềm ẩn bên trong chương trình/đơn vị chương trình cần kiểm thử. Các lỗi này thường khó phát hiện bởi các kỹ thuật kiểm thử chức năng hay kiểm thử hộp đen được trình bày trong chương 5. Để áp dụng phương pháp này, chúng ta cần phân tích mã nguồn và xây dựng các ca kiểm thử ứng với các dòng điều khiển của chương trình/đơn vị chương trình. Các độ đo hay tiêu chí kiểm thử cho phương pháp này cũng sẽ được giới thiệu.

6.1 Kiểm thử hộp trắng

Kiểm thử hộp trắng sử dụng các chiến lược cụ thể và sử dụng mã nguồn của chương trình/đơn vị phần mềm cần kiểm thử nhằm kiểm tra xem chương trình/đơn vị phần mềm có thực hiện đúng so với thiết kế và đặc tả hay không. Trong khi các phương pháp kiểm thử hộp đen hay kiểm thử chức năng chỉ cho phép phát hiện các lỗi/khiếm khuyết có thể quan sát được, kiểm thử hộp trắng cho

phép phát hiện các lỗi/khiếm khuyết tiềm ẩn bên trong chương trình/đơn vị phần mềm. Các lỗi này thường khó phát hiện bởi các phương pháp kiểm thử hộp đen. Khác với các phương pháp kiểm thử hộp đen nơi mà các ca kiểm thử được sinh ra từ đặc tả của hệ thống, các ca kiểm thử trong các phương pháp kiểm thử hộp trắng được sinh ra từ mã nguồn. Kiểm thử hộp đen và kiểm thử hộp trắng không thể thay thế cho nhau mà chúng cần được sử dụng kết hợp với nhau trong một quy trình kiểm thử thống nhất nhằm đảm bảo chất lượng phần mềm. Tuy nhiên, để áp dụng các phương pháp kiểm thử hộp trắng, người kiểm thử không chỉ cần hiểu rõ giải thuật mà còn cần có các kỹ năng và kiến thức tốt về ngôn ngữ lập trình được dùng để phát triển phần mềm, nhằm hiểu rõ mã nguồn của chương trình/đơn vị phần mềm cần kiểm thử. Do vậy, việc áp dụng các phương pháp kiểm thử hộp trắng thường tốn thời gian và công sức nhất là khi chương trình/đơn vị phần mềm có kích thước lớn. Vì lý do này, các phương pháp kiểm thử hộp trắng chủ yếu được sử dụng cho kiểm thử đơn vị [D.95].

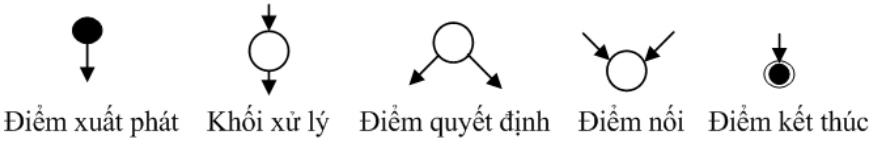
Hai phương pháp được sử dụng trong kiểm thử hộp trắng là kiểm thử dòng điều khiển (control flow testing) và kiểm thử dòng dữ liệu (data flow testing). Phương pháp kiểm thử dòng điều khiển tập trung kiểm thử tính đúng đắn của các giải thuật sử dụng trong các chương trình/đơn vị phần mềm. Phương pháp kiểm thử dòng dữ liệu tập trung kiểm thử tính đúng đắn của việc sử dụng các biến dữ liệu sử dụng trong chương trình/đơn vị phần mềm. Trong chương này, chúng ta sẽ tìm hiểu chi tiết về phương pháp kiểm thử dòng điều khiển. Phương pháp kiểm thử dòng dữ liệu sẽ được giới thiệu trong chương 7.

6.2 Đồ thị dòng điều khiển

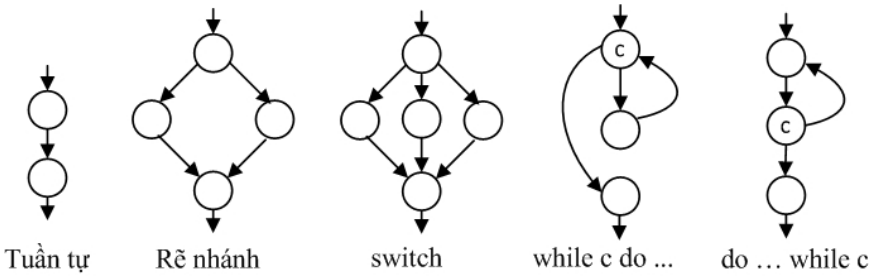
Phương pháp kiểm thử dòng điều khiển dựa trên khái niệm đồ thị dòng điều khiển (control flow graph). Đồ thị này được xây dựng từ

mã nguồn của chương trình/đơn vị chương trình. Đồ thị dòng điều khiển là một đồ thị có hướng gồm các đỉnh tương ứng với các câu lệnh/nhóm câu lệnh và các cạnh là các dòng điều khiển giữa các câu lệnh/nhóm câu lệnh. Nếu i và j là các đỉnh của đồ thị dòng điều khiển thì tồn tại một cạnh từ i đến j nếu lệnh tương ứng với j có thể được thực hiện ngay sau lệnh tương ứng với i.

Xây dựng một đồ thị dòng điều khiển từ một chương trình/đơn vị chương trình khá đơn giản. Hình 6.1 mô tả các thành phần cơ bản của đồ thị dòng điều khiển bao gồm điểm bắt đầu của đơn vị chương trình, khối xử lý chứa các câu lệnh khai báo hoặc tính toán, điểm quyết định ứng với các câu lệnh điều kiện trong các khối lệnh rẽ nhánh hoặc lặp, điểm nối ứng với các câu lệnh ngay sau các lệnh rẽ nhánh, và điểm kết thúc ứng với điểm kết thúc của đơn vị chương trình. Các cấu trúc điều khiển phổ biến của chương trình được mô tả trong Hình 6.2. Chúng ta sẽ sử dụng các thành phần cơ bản và các cấu trúc phổ biến này để dễ dàng xây dựng đồ thị dòng điều khiển cho mọi đơn vị chương trình viết bằng mọi ngôn ngữ lập trình.

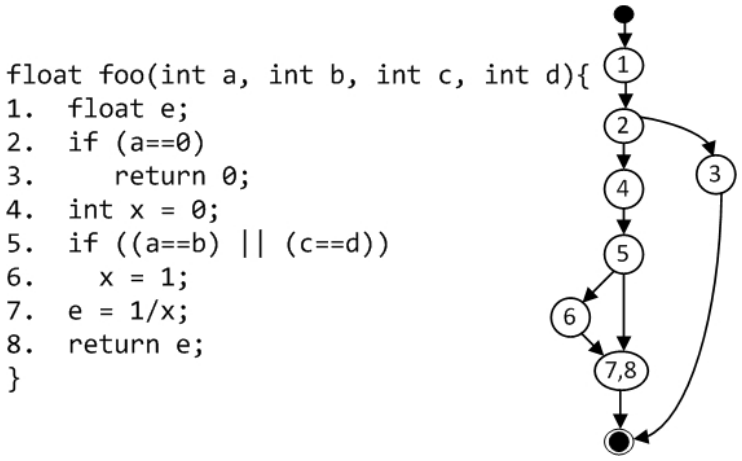


Hình 6.1: Các thành phần cơ bản của đồ thị chương trình.



Hình 6.2: Các cấu trúc điều khiển phổ biến của chương trình.

Chúng ta thử xem cách dựng đồ thị dòng điều khiển cho đơn vị chương trình có mã nguồn bằng ngôn ngữ C như Hình 6.3. Chúng ta đánh số các dòng lệnh của đơn vị chương trình và lấy số này làm đỉnh của đồ thị. Điểm xuất phát của đơn vị chương trình ứng với câu lệnh khai báo hàm `foo`. Đỉnh 1 ứng với câu lệnh khai báo biến e. Các đỉnh 2 và 3 ứng với câu lệnh `if`. Đỉnh 4 ứng với câu lệnh khai báo biến x trong khi các đỉnh 5 và 6 ứng với câu lệnh `if`. Đỉnh 7,8 đại diện cho hai câu lệnh 7 và 8. Trong trường hợp này, chúng ta không tách riêng thành hai đỉnh vì đây là hai câu lệnh tuần tự nên chúng ta ghép chúng thành một đỉnh nhằm tối thiểu số đỉnh của đồ thị dòng điều khiển. Với cách làm này, chúng ta xây dựng được đồ thì dòng điều khiển với số đỉnh nhỏ nhất. Chúng ta sẽ sử dụng đồ thị này để phân tích và sinh các ca kiểm thử nên đồ thị càng ít đỉnh thì độ phức tạp của thuật toán phân tích càng nhỏ.



Hình 6.3: Mã nguồn của hàm `foo` và đồ thị dòng điều khiển của nó.

6.3 Các độ đo kiểm thử

Kiểm thử chức năng (kiểm thử hộp đen) có hạn chế là chúng ta không biết có thừa hay thiếu các ca kiểm thử hay không so với chương trình cài đặt và thiếu thừa ở mức độ nào. Độ đo kiểm thử

là một công cụ giúp ta đo mức độ bao phủ chương trình của một tập ca kiểm thử cho trước. Mức độ bao phủ của một bộ kiểm thử (tập các ca kiểm thử) được đo bằng tỷ lệ các thành phần thực sự được kiểm thử so với tổng thể sau khi đã thực hiện các ca kiểm thử. Thành phần liên quan có thể là câu lệnh, điểm quyết định, điều kiện con, đường thi hành hay là sự kết hợp của chúng. Độ bao phủ càng lớn thì độ tin cậy của bộ kiểm thử càng cao. Độ đo này giúp chúng ta kiểm soát và quản lý quá trình kiểm thử tốt hơn. Mục tiêu của chúng ta là kiểm thử với số ca kiểm thử tối thiểu nhưng đạt được độ bao phủ tối đa. Có rất nhiều độ đo kiểm thử đang được sử dụng hiện nay, dưới đây là ba độ đo kiểm thử đang được sử dụng phổ biến nhất trong thực tế [Lee03].

Độ đo kiểm thử cấp 1 (C_1): mỗi câu lệnh được thực hiện ít nhất một lần sau khi chạy các ca kiểm thử (test cases). Ví dụ, với hàm `foo` có mã nguồn như trong Hình 6.3, ta chỉ cần hai ca kiểm thử như Bảng 6.1 là đạt 100% độ phủ cho độ đo C_1 với EO (expected output) là giá trị đầu ra mong đợi và RO (real output) là giá trị đầu ra thực tế (giá trị này sẽ được điền khi thực hiện ca kiểm thử).

Bảng 6.1: Các ca kiểm thử cho độ đo C_1 của hàm `foo`

ID	Inputs	EO	RO	Note
tc1	0, 1, 2, 3	0		
tc2	1, 1, 2, 3	1		

Độ đo kiểm thử cấp 2 (C_2): các điểm quyết định trong đồ thị dòng điều khiển của đơn vị kiểm thử đều được thực hiện ít nhất một lần cả hai nhánh đúng và sai. Ví dụ, Bảng 6.2 mô tả các trường hợp cần kiểm thử để đạt được 100% độ phủ của độ đo C_2 ứng với hàm `foo` được mô tả trong Hình 6.3.

Như vậy, với hai ca kiểm thử trong độ đo kiểm thử cấp 1 (tc1 và tc2), ta chỉ kiểm thử được $3/4 = 75\%$ ứng với độ đo kiểm thử

Bảng 6.2: Các trường hợp cần kiểm thử của độ đo C_2 với hàm `foo`

Điểm quyết định	Điều kiện tương ứng	Đúng	Sai
2	a==0	tc1	tc2
5	(a == b) || (c == d)	tc2	?

cấp 2. Chúng ta cần một ca kiểm thử nữa ứng với trường hợp sai của điều kiện `(a == b) || (c == d)` nhằm đạt được 100% độ phủ của độ đo C_2. Bảng 6.3 mô tả các ca kiểm thử cho mục đích này.

Bảng 6.3: Các ca kiểm thử cho độ đo C_2 của hàm `foo`

ID	Inputs	EO	RO	Note
tc1	0, 1, 2, 3	0		
tc2	1, 1, 2, 3	1		
tc3	1, 2, 1, 2	Lỗi chia cho 0		

Độ đo kiểm thử cấp 3 (C_3): Với các điều kiện phức tạp (chứa nhiều điều kiện con cơ bản), việc chỉ quan tâm đến giá trị đúng sai là không đủ để kiểm tra tính đúng đắn của chương trình ứng với điều kiện phức tạp này. Ví dụ, nếu một điều kiện phức tạp gồm hai điều kiện con cơ bản, chúng ta có bốn trường hợp cần kiểm thử chứ không phải hai trường hợp đúng sai như độ đo C_2. Với các đơn vị chương trình có yêu cầu cao về tính đúng đắn, việc tuân thủ độ đo C_3 là hết sức cần thiết. Điều kiện để đảm bảo độ đo này là các điều kiện con thuộc các điều kiện phức tạp tương ứng với các điểm quyết định trong đồ thị dòng điều khiển của đơn vị cần kiểm thử đều được thực hiện ít nhất một lần cả hai nhánh đúng và sai. Ví dụ, Bảng 6.4 mô tả các trường hợp cần kiểm thử để đạt được 100% độ phủ của độ đo C_3 ứng với hàm `foo` được mô tả trong Hình 6.3.

Như vậy, với ba ca kiểm thử trong độ đo kiểm thử cấp 2 (tc1, tc2 và tc3), ta chỉ kiểm thử được 7/8 = 87,5% ứng với độ đo kiểm thử cấp 3. Chúng ta cần một ca kiểm thử nữa ứng với trường hợp

Bảng 6.4: Các trường hợp cần kiểm thử của độ đo C_3 với hàm `foo`

Điểm quyết định	Điều kiện tương ứng	Đúng	Sai
2	a==0	tc1	tc2
5	(a == b)	tc2	tc3
5	(c == d)	?	tc2

sai của điều kiện con cơ bản `(c == d)` nhằm đạt được 100% độ phủ của độ đo C_3. Bảng 6.5 mô tả các ca kiểm thử cho mục đích này.

Bảng 6.5: Các ca kiểm thử cho độ đo C_3 của hàm `foo`

ID	Inputs	EO	RO	Note
tc1	0, 1, 2, 3	0		
tc2	1, 1, 2, 3	1		
tc3	1, 2, 1, 2	Lỗi chia cho 0		
tc4	1, 2, 1, 1	1		

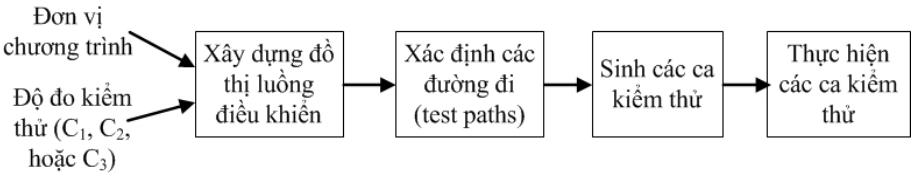
6.4 Kiểm thử dựa trên độ đo

Kiểm thử dựa trên độ đo là phương pháp phân tích mã nguồn và thực hiện chương trình/đơn vị chương trình sao cho thỏa mãn một độ đo kiểm thử cho trước. Hình 6.4 mô tả quy trình kiểm thử dựa trên độ đo cho các đơn vị chương trình. Với mỗi đơn vị chương trình, đồ thị dòng điều khiển ứng với các độ đo C_1 và C_2 là giống nhau trong khi chúng khác với đồ thị dòng điều khiển ứng với độ đo C_3. Sở dĩ có sự khác biệt này là bởi vì đồ thì dòng điều khiển ứng với độ đo C_3 sẽ bao gồm nhiều đỉnh điều khiển ứng với từng điều kiện con cơ bản. Với mỗi đơn vị chương trình và mỗi độ đo kiểm thử, chúng ta tiến hành xây dựng đồ thị dòng điều khiển tương ứng. Các đường đi của chương trình (xuất phát từ điểm bắt đầu, đi qua các đỉnh của đồ thị và kết thúc ở điểm cuối) được xác định sao cho khi chúng được thực hiện thì độ đo kiểm thử tương ứng được

thỏa mãn. Dựa trên ý tưởng của T. J. McCabe [McC76b, WM96], số đường đi chương trình ứng với đồ thị dòng điều khiển của nó được tính bằng một trong các phương pháp sau:

- Số cạnh – số đỉnh + 2
- Số đỉnh quyết định + 1

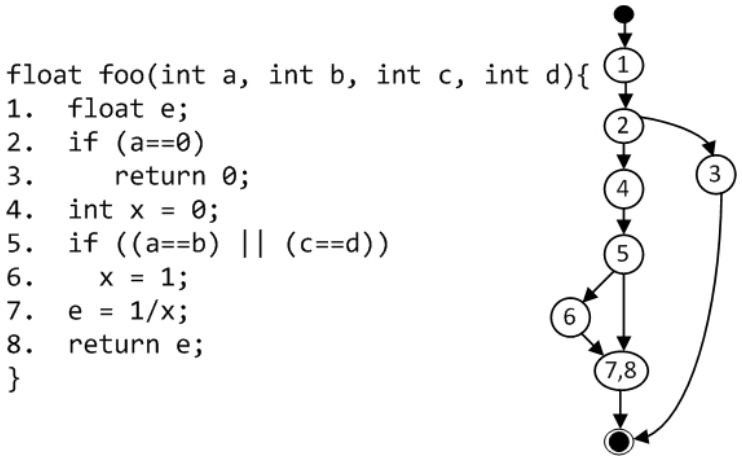
Sau khi có được các đường đi của đơn vị chương trình cần kiểm thử, với mỗi đường đi, chúng ta sẽ sinh một ca kiểm thử tương ứng. Có rất nhiều kỹ thuật để sinh các ca kiểm thử từ các đường đi này. Để sinh bộ dữ liệu đầu vào cho ca kiểm thử ứng với mỗi đường đi, một kỹ thuật phổ biến là chọn ngẫu nhiên một bộ dữ liệu đầu vào sao cho nó thỏa mãn tất cả các điều kiện trên đường đi này. Nếu chúng ta tìm được một bộ dữ liệu đầu vào như vậy, đường đi tương ứng được gọi là đường đi thực thi được. Ngược lại, đường đi này ứng với các câu lệnh không bao giờ xảy ra (thừa?) hoặc nó là một phần của một đường đi khác liên quan đến vòng lặp trong chương trình. Khi có được bộ giá trị đầu vào, chúng ta sẽ phân tích và xác định giá trị đầu ra mong muốn cho ca kiểm thử này. Cuối cùng, các ca kiểm thử được thực hiện trên đơn vị chương trình nhằm phát hiện các lỗi.



Hình 6.4: Quy trình kiểm thử đơn vị chương trình dựa trên độ đo.

6.4.1 Kiểm thử cho độ đo C_1

Xét lại hàm `foo` có mã nguồn như Hình 6.5. Chúng ta xây dựng đồ thị dòng điều khiển ứng với độ phủ C_1 cho hàm này như Hình 6.5. Để đạt được 100% độ phủ của độ đo C_1, ta chỉ cần hai đường đi như



Hình 6.5: Mã nguồn của hàm `foo` và đồ thị dòng điều khiển của nó.

sau để đảm bảo được tất cả các câu lệnh của hàm `foo` được kiểm thử ít nhất một lần. Để kiểm tra việc đảm bảo độ đo C_1, chúng ta cần kiểm tra tất cả các lệnh/khối lệnh (1-8) đều được xuất hiện ít nhất một lần trong các đường đi này. Rõ ràng, hai đường đi này thỏa mãn điều kiện trên nên chúng ta đạt được 100% độ phủ C_1.

1. 1; 2(F); 4; 5(T); 6; 7,8

2. 1; 2(T); 3

Với đường đi 1; 2(F); 4; 5(T); 6; 7,8, ta sẽ sinh một ca kiểm thử để nó được thực thi khi thực hiện ca kiểm thử này. Ý tưởng của việc sinh ca kiểm thử này là tìm một bộ giá trị đầu vào cho a, b, c và d sao cho điều kiện ứng với điểm quyết định 2 `(a == 0)` là sai và điều kiện ứng với điểm quyết định 5 `((a == b) || (c == d))` là đúng. Giá trị đầu ra mong đợi (EO) của ca kiểm thử này là 1. Tương tự, ta sẽ sinh ca kiểm thử ứng với đường đi 1; 2(T); 3 với đầu ra mong đợi là 0. Chúng ta sẽ tìm một bộ đầu vào sao cho điều kiện `(a == 0)` là đúng. Bảng 6.6 là một ví dụ về hai ca kiểm thử được sinh ra bằng ý tưởng trên.

Bảng 6.6: Các ca kiểm thử cho độ đo C_1 của hàm `foo`

ID	Test Path	Inputs	EO	RO	Note
tc1	1; 2(F); 4; 5(T); 6; 7,8	2, 2, 3, 5	1		
tc2	1; 2(T); 3	0, 3, 2, 7	0		

6.4.2 Kiểm thử cho độ đo C_2

Như chúng ta đã biết, với mỗi đơn vị chương trình, đồ thị dòng điều khiển ứng với các độ đo C_1 và C_2 là giống nhau. Vì vậy, đồ thị dòng điều khiển ứng với độ đo C_1 của hàm `foo` được mô tả ở Hình 6.5 cũng là đồ thị dòng điều khiển của hàm này ứng với độ đo C_2. Tuy nhiên, để được 100% độ phủ của độ đo C_2 chúng ta cần tối thiểu ba đường đi. Tại sao chúng ta biết được điều này? Như đã trình bày ở mục 6.4, chúng ta có hai cách để tính được con số này. Ví dụ, đồ thị dòng điều khiển của hàm `foo` có hai điểm quyết định là 2 và 5 nên chúng ta cần $2 + 1 = 3$ đường đi để đạt được 100% độ phủ của độ đo C_2. Các đường đi cần thiết được liệt kê như sau. Rõ ràng với ba đường đi này, cả hai nhánh đúng và sai của hai điểm quyết định 2 và 5 đều được kiểm tra.

1. 1; 2(F); 4; 5(T); 6; 7,8
2. 1; 2(T); 3
3. 1; 2(F); 4; 5(F); 7,8

Để sinh các ca kiểm thử ứng với các đường đi trên, chúng ta chỉ cần quan tâm đến đường đi (3) vì việc sinh các ca kiểm thử cho các đường đi (1) và (2) đã được trình bày ở mục kiểm thử cho độ đo C_1 (mục 6.4.1). Với đường đi (3), ta chỉ cần chọn một bộ đầu vào sao cho điều kiện ứng với điểm quyết định 2 `(a == 0)` là sai và điều kiện ứng với điểm quyết định 5 `((a == b) || (c == d))` cũng là sai. Giá trị đầu ra mong đợi của đường đi này là lỗi chia cho 0.

Bảng 6.7: Các ca kiểm thử cho độ đo C_2 của hàm `foo`

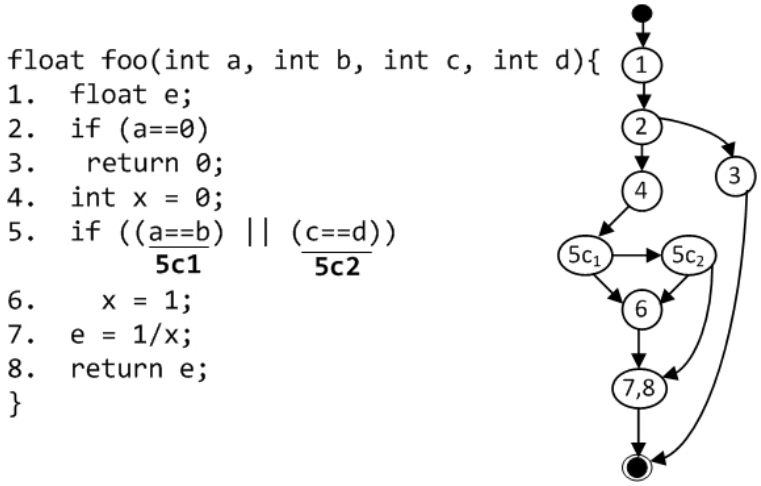
ID	Test Path	Inputs	EO	RO	Note
tc1	1; 2(F); 4; 5(T); 6; 7,8	2,2,3, 5	1		
tc2	1; 2(T); 3	0,3,2,7	0		
tc3	1; 2(F); 4; 5(F); 7,8	2,3,4,5	lỗi chia cho 0		

Bảng 6.7 là một ví dụ về ba ca kiểm thử được sinh ra bằng ý tưởng trên ứng với các đường đi (1), (2), và (3).

Độ đo C_1 đảm bảo các câu lệnh được "viếng thăm" ít nhất một lần khi thực hiện tất cả các ca kiểm thử được sinh ra ứng với độ đo này. Đây là độ đo khá tốt và việc đảm bảo độ đo này trong thực tế cũng khá tốn kém. Tuy nhiên, qua ví dụ trên, chúng ta thấy rằng nếu chỉ sử dụng độ đo C_1 với hai ca kiểm thử như trong bảng 6.6, lỗi chia cho không sẽ không được phát hiện. Chỉ khi kiểm tra cả hai nhánh đúng sai của tất cả các điểm quyết định (các lệnh điều khiển) thì lỗi này mới được phát hiện như ca kiểm thử tc3 trong bảng 6.7.

6.4.3 Kiểm thử cho độ đo C_3

Như đã trình bày ở mục 6.3, ứng với mỗi đơn vị chương trình, đồ thị dòng điều khiển ứng với độ đo C_3 khác với đồ thị dòng điều khiển ứng với các độ đo C_1 và C_2. Ví dụ, đồ thị dòng điều khiển của hàm `foo` ứng với độ đo C_3 được xây dựng như Hình 6.6. Với câu lệnh điều kiện 5, vì đây là điều kiện phức tạp nên ta phải tách thành hai điều kiện con cơ bản là `(a == b)` và `(c == d)` ứng với hai điểm quyết định $5c1$ và $5c2$ trong đồ thị dòng điều khiển. Từ câu lệnh 4, nếu điều kiện con `(a == b)` đúng, ta không cần kiểm tra điều kiện con còn lại (vì điều kiện phức tạp là hoặc của hai điều kiện con cơ bản) và thực hiện câu lệnh 6. Nếu điều kiện con `(a == b)` là sai, ta cần tiến hành kiểm tra điều kiện con cơ bản còn lại `(c == d)`. Nếu điều kiện này đúng, ta tiến hành câu lệnh 6. Ngược lại, chúng



Hình 6.6: Hàm `foo` và đồ thị dòng điều khiển ứng với độ đo C_3.

ta thực hiện các câu lệnh 7 và 8. Trong đồ thị này, chúng ta gộp hai lệnh 7 và 8 trong một đỉnh (đỉnh (7,8)) vì đây là hai câu lệnh tuần tự. Mục đích của việc này là nhằm tối thiểu số đỉnh của đồ thị dòng điều khiển. Một đồ thị có số đỉnh càng nhỏ thì chúng ta càng dễ dàng trong việc sinh các đường đi của chương trình và tránh các sai sót trong quá trình này.

Đồ thị dòng điều khiển của hàm `foo` ứng với độ đo C_3 như Hình 6.6 có ba điểm quyết định là 2, 5c1 và 5c2 nên chúng ta cần 3 + 1 = 4 đường đi để được 100% độ phủ của độ đo C_3. Các đường đi cần thiết được liệt kê như sau:

1. 1; 2(F); 4; 5c1(T); 6; 7,8
2. 1; 2(F); 4; 5c1(F); 5c2(T); 6; 7,8
3. 1; 2(F); 4; 5c1(F); 5c2(F); 7,8
4. 1; 2(T); 3

Tương tự như các phương pháp kiểm thử độ đo C_1 và C_2, chúng ta dễ dàng sinh các ca kiểm thử tương ứng cho các đường đi chương

Bảng 6.8: Các ca kiểm thử cho độ đo C_3 của hàm `foo`

ID	Test Path	Inputs	EO	RO	Note
tc1	1; 2(F); 4; 5c1(T); 6; 7,8	0, 2, 3, 5	0		
tc2	1; 2(F); 4; 5c1(F); 5c2(T); 6; 7,8	2, 2, 2, 7	1		
tc3	1; 2(F); 4; 5c1(F); 5c2(F); 7,8	2,3,4,5	lỗi chia cho 0		
tc4	1; 2(T); 3	2,3,4,4	1		

trình như đã mô tả trên. Bảng 6.8 là một ví dụ về các ca kiểm thử cho hàm `foo` ứng với độ đo C_3. Vì đồ thị dòng điều khiển của hàm `foo` ứng với độ đo C_3 có ba (3) đỉnh điều khiển nên chúng ta cần tối thiểu bốn (4) đường đi từ đỉnh đầu đến đỉnh cuối của đồ thị này (bảng 6.8). Ứng với mỗi đường đi, chúng ta cũng sẽ tìm một bộ đầu vào cho các tham số a, b, c và d sao cho các điều kiện ứng với mỗi đường đi thỏa mãn.

6.4.4 Kiểm thử vòng lặp

Cho dù chúng ta tiến hành kiểm thử các đơn vị chương trình với độ đo C_3 (độ đo với yêu cầu cao nhất), phương pháp kiểm thử dòng điều khiển không thể kiểm thử các vòng lặp xuất hiện trong các đơn vị chương trình. Lý do là các đường đi sinh ra từ đồ thị dòng điều khiển không chứa các vòng lặp. Trong thực tế, lỗi hay xảy ra ở các vòng lặp. Vì lý do này, chúng ta cần sinh thêm các ca kiểm thử cho các vòng lặp nhằm giảm tỷ lệ lỗi của các đơn vị chương trình.

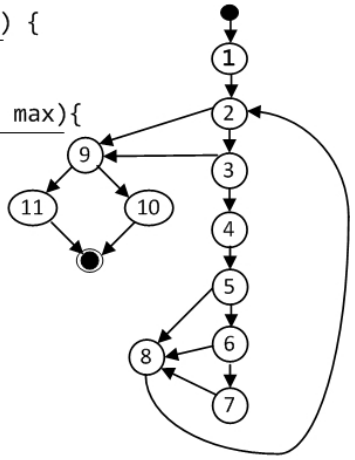
Với mỗi đơn vị chương trình có vòng lặp, chúng ta cần quan tâm đến ba trường hợp sau:

- **Lệnh lặp đơn giản:** đơn vị chương trình chỉ chứa đúng một vòng lặp (thân của vòng lặp không chứa các vòng lặp khác).

- **Lệnh lặp liền kề:** đơn vị chương trình chỉ chứa các lệnh lặp kế tiếp nhau.
- **Lệnh lặp lồng nhau:** đơn vị chương trình chỉ chứa các vòng lặp chứa các lệnh lặp khác.

Để kiểm thử các đơn vị chương trình chỉ có lệnh lặp đơn giản, ta xét hàm average với mã nguồn đồ thị dòng điều khiển tương ứng với độ đo C_3 như Hình 6.7.

```
double average(double value[], double min, double
     max, int& tcnt, int& vcnt) {
     double sum = 0;
     int i = 1;          |1
     tcnt = vcnt = 0;    |
     while (value[i] <> -999 && tcnt <100) {
                 2              3
          tcnt++; |4
          if (min<=value[i] && value[i]<= max){
                    5              6
                    sum += value[i];|7
                    vcnt ++;
          }
          i++;|8
     }//end while
     if (vcnt > 0)
           9
          return sum/vcnt; |10
     return -999; |11
}//end
```



Hình 6.7: Hàm average và đồ thị dòng điều khiển ứng với độ đo C_3.

Đồ thị dòng điều khiển của hàm `average` ứng với độ đo C_3 như Hình 6.7 có năm điểm quyết định là 2, 3, 5, 6 và 9 nên chúng ta cần $5 + 1 = 6$ đường đi để được 100% độ phủ của độ đo C_3. Các đường đi cần thiết được liệt kê như sau:

1. 1; 2(F); 9(T); 10
2. 1; 2(F); 9(F); 11

3. 1; 2(T); 3(F); 9(T); 10

4. 1; 2(T); 3(T); 4; 5(F); 8; 2(F); 9(F); 11

5. 1; 2(T); 3(T); 4; 5(T); 6(F); 8; 2(F); 9(F); 11

6. 1; 2(T); 3(T); 4; 5(T); 6(T); 7; 8; 2(F); 9(T); 10

Với mỗi đường đi, chúng ta sẽ sinh một ca kiểm thử tương ứng. Bảng 6.9 mô tả các ca kiểm thử cho hàm `average` ứng với độ đo C_3. Với mỗi ca kiểm thử, bộ dữ liệu đầu vào gồm ba thành phần: `double value[]`, `double min`, và `double max`. Với đường đi 1; 2(F); 9(T); 10, ta không thể tìm được bộ dữ liệu đầu vào để nó được thực thi. Thực vậy, điều kiện để thực thi đường đi này là `value[0] = -999` và điều kiện 9 đúng (tức là `vcnt > 0`). Điều này không bao giờ xảy ra vì nếu `value[0]` = -999 thì `vcnt = 0`. Tương tự, chúng ta cũng không thể sinh bộ kiểm thử với đường đi 1; 2(T); 3(F); 9(T); 10. Trong các trường hợp này, chúng ta không cần sinh các ca kiểm thử cho những đường đi này. Chúng sẽ được kiểm thử bởi các đường đi khác (Ví dụ: đường đi 1; 2(T); 3(T); 4; 5(T); 6(T); 7; 8; 2(F); 9(T); 10 chứa các đỉnh của các đường dẫn trên).

Bảng 6.9: Các ca kiểm thử cho độ đo C_3 của hàm `average`

ID	Test Path	Inputs	EO	RO	Note
tc1	1; 2(F); 9(T); 10				tc6
tc2	1; 2(F); 9(F); 11	[-999,...], 1, 2	-999		
tc3	1; 2(T); 3(F); 9(T); 10				tc6
tc4	2(T); 3(T); 4; 5(F); 8; 2(F); 9(F); 11	[0,-999], 1, 2	-999		
tc5	1; 2(T); 3(T); 4; 5(T); 6(F); 8; 2(F); 9(F); 11	[3,-999], 1, 2	-999		
tc6	1; 2(T); 3(T); 4; 5(T); 6(T); 7; 8; 2(F); 9(T); 10	[1,-999], 1, 2	1		

Với các đường đi trên, vòng lặp while của hàm `average` chỉ được thực hiện tối đa một lần lặp nên chúng ta rất khó để phát hiện các lỗi tiềm ẩn (có thể có) bên trong vòng lặp này. Các lỗi này có thể xảy ra khi vòng lặp này được thực hiện nhiều lần lặp. Ví dụ trên đã chỉ ra những hạn chế của phương pháp kiểm thử dòng điều khiển khi áp dụng cho các chương trình/đơn vị chương trình có chứa vòng lặp. Để giải quyết vấn đề này, chúng ta cần sinh thêm bảy ca kiểm thử ứng với bảy trường hợp sau:

1. Vòng lặp thực hiện 0 lần
2. Vòng lặp thực hiện 1 lần
3. Vòng lặp thực hiện 2 lần
4. Vòng lặp thực hiện k lần, $2 < k < n - 1$, với n là số lần lặp tối đa của vòng lặp
5. Vòng lặp thực hiện n - 1 lần
6. Vòng lặp thực hiện n lần
7. Vòng lặp thực hiện n + 1 lần

Chú ý rằng trong một số trường hợp chúng ta có thể không xác định được số lần lặp tối đa của các vòng lặp. Trong trường hợp này, chúng ta chỉ cần sinh bốn ca kiểm thử đầu tiên. Tương tự, trong một số các trường hợp khác, chúng ta không thể sinh ca kiểm thử để vòng lặp thực hiện n + 1 lần (trường hợp thứ 7). Khi đó, chúng ta chỉ cần sinh sáu ca kiểm thử còn lại (các trường hợp từ 1–6). Ví dụ, với vòng lặp `while` trong hàm `average` như Hình 6.7, vòng lặp này chỉ thực hiện lặp tối đa 100 lần nên chúng ta không thể sinh ca kiểm thử để nó thực hiện n + 1 = 101 lần. Kết quả là chúng ta chỉ cần sinh sáu ca kiểm thử đầu tiên như trong Bảng 6.10 nhằm kiểm thử vòng lặp này.

Bảng 6.10: Các ca kiểm thử cho vòng lặp `while` của hàm `average`

ID	Lần lặp	Inputs	EO	RO	Note
tcl0	0	[-999, ...], 1, 2	-999		
tcl1	1	[1,-999], 1, 2	1		
tcl2	2	[1,2,-999], 1, 2	1.5		
tclk	5	[1,2,3,4,5,-999], 1, 10	3		
tcl(n-1)	99	[1,2,...,99,-999], 1, 100	50		
tcln	100	[1,2,...,100], 1, 2	50.5		
tcl(n+1)					

Với các chương trình/đơn vị chương trình có các vòng lặp liền kề, chúng ta tiến hành kiểm thử tuần tự từ trên xuống. Mỗi vòng lặp được kiểm thử bằng bảy ca kiểm thử như vòng lặp đơn giản (như đã mô tả ở trên). Trong trường hợp các vòng lặp lồng nhau, chúng ta tiến hành kiểm thử tuần tự các vòng lặp theo thứ tự từ trong ra ngoài (mỗi vòng lặp cũng dùng bảy ca kiểm thử như đã mô tả ở trên).

6.5 Tổng kết

Kiểm thử dòng điều khiển là một trong những phương pháp kiểm thử quan trọng nhất của chiến lược kiểm thử hộp trắng cho các chương trình/đơn vị chương trình. Phương pháp này cho phép phát hiện ra các lỗi (có thể có) tiềm ẩn bên trong chương trình/đơn vị chương trình bằng cách kiểm thử các đường đi của nó tương ứng với các dòng điều khiển có thể có. Để áp dụng phương pháp này, chúng ta cần xác định độ đo kiểm thử (cần kiểm thử với độ đo nào?). Tiếp theo, đồ thị dòng điều khiển của chương trình/đơn vị chương trình ứng với độ đo kiểm thử sẽ được tạo ra. Dựa vào đồ thị này, chúng ta sẽ sinh ra các đường đi độc lập. Số đường đi độc lập này là các trường hợp tối thiểu nhất để đảm bảo 100% độ bao phủ ứng với độ đo yêu cầu. Với mỗi đường đi, chúng ta sẽ sinh ra

một ca kiểm thử sao cho khi nó được dùng để kiểm thử thì đường đi này được thực thi. Việc sinh các đầu vào cho các ca kiểm thử này là một bài toán thú vị. Chúng ta sẽ chọn một bộ đầu vào sao cho thỏa mãn các điểm quyết định có trong đường đi tương ứng. Giá trị đầu ra mong muốn ứng với mỗi bộ đầu vào của mỗi ca kiểm thử cũng sẽ được tính toán. Đây là bài toán khó và thường chỉ có các chuyên gia phân tích chương trình mới có thể trả lời chính xác giá trị này. Cuối cùng, các ca kiểm thử được chạy nhằm phát hiện ra các lỗi của chương trình/đơn vị chương trình cần kiểm thử. Khi một lỗi được phát hiện bởi một ca kiểm thử nào đó, nó sẽ được thông báo tới lập trình viên tương ứng. Lập trình viên sẽ tiến hành sửa lỗi (phát hiện vị trí của lỗi ở câu lệnh nào và sửa nó). Trong trường hợp này, chúng ta không chỉ thực hiện lại ca kiểm thử phát hiện ra lỗi này mà phải thực hiện lại tất cả các ca kiểm thử của đơn vị chương trình. Lý do chúng ta phải thực hiện công việc này là vì khi sửa lỗi này có thể gây ra một số lỗi khác.

Việc áp dụng phương pháp kiểm thử dòng điều khiển là khó và tốn kém hơn các phương pháp kiểm thử hộp đen (phân hoạch tương đương, phân tích giá trị biên, bảng quyết định, v.v.). Để áp dụng kỹ thuật này, chúng ta cần đội ngũ nhân lực về kiểm thử có kiến thức và kỹ năng tốt. Hơn nữa, chúng ta cần một sự đầu tư lớn về các nguồn lực khác (tài chính, thời gian, v.v.) mới có thể thực hiện tốt phương pháp này. Đây là một yêu cầu khó và không nhiều công ty phần mềm đáp ứng được. Tự động hóa phương pháp kiểm thử dòng điều khiển hứa hẹn sẽ là một giải pháp tốt nhằm giúp cho các công ty giải quyết những khó khăn này. Hiện nay, đã có nhiều công cụ hỗ trợ một phần hoặc hoàn toàn các bước trong phương pháp này. Một môi trường lập trình mới nơi mà các công cụ lập trình trực quan được tích hợp với công cụ kiểm thử tự động nhằm giải quyết những khó khăn nêu trên. Eclipse[1] là một ví dụ điển hình về xu hướng này. Trong các phiên bản hiện nay của môi

[1] www.eclipse.org

trường này đã được tích hợp công cụ kiểm thử tự động có tên là JUnit[1]. Công cụ kiểm thử này chưa hỗ trợ việc sinh các ca kiểm thử tự động nhưng nó cho phép chúng ta viết các kịch bản kiểm thử độc lập với mã nguồn và thực thi chúng một cách tự động trên một môi trường thống nhất.

Một giải pháp khác có thể giảm thiểu chi phí cho việc kiểm thử dòng dữ liệu nói chung và kiểm thử hộp trắng nói riêng là kết hợp phương pháp này với các phương pháp kiểm thử hộp đen (kiểm thử chức năng) trong một quy trình thống nhất. Sau khi tiến hành các phương pháp kiểm thử hộp đen, chúng ta tiến hành phân tích các ca kiểm thử đã được sử dụng để xác định các nhánh chương trình nào đã được kiểm thử, các nhánh nào chưa được kiểm thử (trong kiểm thử dòng điều khiển). Với cách làm này, chúng ta chỉ cần sinh các ca kiểm thử cho các nhánh chưa được kiểm thử bởi các ca kiểm thử hộp đen mà vẫn đảm bảo 100% độ bao phủ cho kiểm thử dòng điều khiển.

6.6 Bài tập

1. Tại sao chúng ta cần thực hiện kiểm thử hộp trắng?

2. Phân biệt kiểm thử hộp trắng và kiểm thử hộp đen.

3. Tại sao kiểm thử hộp trắng thường có chi phí và độ khó cao hơn kiểm thử hộp đen?

4. Thế nào là đồ thị dòng điều khiển của một chương trình/đơn vị chương trình?

5. Trình bày các độ đo kiểm thử cho kiểm thử dòng điều khiển.

6. Chứng minh rằng độ đo C_2 đảm bảo độ đo C_1.

[1] http://junit.org/

7. Chứng minh rằng độ đo C_3 đảm bảo độ đo C_2.

8. Trình bày các bước nhằm kiểm thử một đơn vị chương trình theo phương pháp kiểm thử dòng điều khiển với một độ đo kiểm thử cho trước.

9. Cho hàm được viết bằng ngôn ngữ C như Đoạn mã 6.1.

Đoạn mã 6.1: Mã nguồn của hàm `Grade`

```
char Grade(int score){

      int res;
      if(score < 0 || score > 10)
        return ‘’I;
      if(score>=9)
        res = ’A’;
      else
        if(score >=8)
              res =’B’;
        else
              if(score >=6.5)
                    res = ’C’;
              else
                    if(score >=5)
                       res = ’D’;
                    else
                       res = ’F’;
      return res;
}
```

- Hãy xây dựng đồ thị dòng điều khiển cho hàm `Grade` ứng với độ đo C_1 và C_2.
- Hãy sinh các đường đi và các ca kiểm thử với độ đo C_1.
- Hãy sinh các đường đi và các ca kiểm thử với độ đo C_2.

10. Cho hàm được viết bằng ngôn ngữ C như Đoạn mã 6.2.

Đoạn mã 6.2: Mã nguồn của hàm `BinSearch`

```
int Binsearch(int x, int v[], int n){
        int low = 0, high, mid;
        high = n - 1;
        while (low <= high) {
                mid = (low + high)/2;
                if (x < v[mid])
                        high = mid - 1;
                else
                        if (x > v[mid])
                                low = mid + 1;
                        else
                                return mid;
        }//end while
        return -1;
}//the end
```

- Hãy xây dựng đồ thị dòng điều khiển cho hàm `BinSearch` ứng với độ đo C_1 và C_2.
- Hãy sinh các đường đi và các ca kiểm thử với độ đo C_1.
- Hãy sinh các đường đi và các ca kiểm thử với độ đo C_2.
- Hãy sinh các ca kiểm thử để kiểm thử vòng lặp `while`.

11. Cho hàm được viết bằng ngôn ngữ C như Đoạn mã 6.3.

Đoạn mã 6.3: Mã nguồn của hàm `LaSoNguyenTo`

```
//tra lai 1 neu n la so nguyen to
//nguoc lai, tra lai 0
int LaSoNguyenTo(int n){
        int i=2;
        do{
                if((n % i) == 0)
```

```
                        return 0;
                i++;
        } while(i <= n/2);
        return 1;
}
```

- Xây dựng đồ thị dòng điều khiển cho hàm `LaSoNguyenTo` ứng với độ đo C_1 và C_2.
- Hãy sinh các đường đi và các ca kiểm thử với độ đo C_1.
- Hãy sinh các đường đi và các ca kiểm thử với độ đo C_2.
- Sinh các ca kiểm thử để kiểm thử vòng lặp `do ... while`.

12. Cho hàm được viết bằng ngôn ngữ C như Đoạn mã 6.4.

Đoạn mã 6.4: Mã nguồn của hàm `UCLN`

```
int UCLN(int m, int n){
        if (m < 0) m = -m;
        if (n < 0) n = -n;
        if (m == 0)   return n;
        if (n == 0)   return m;
        while (m != n) {
                if(m > n)
                        m = m - n;
                else
                        n = n - m;
        }//end while
        return m;
}
```

- Hãy xây dựng đồ thị dòng điều khiển cho hàm `UCLN` ứng với độ đo C_1 và C_2.
- Hãy sinh các đường đi và các ca kiểm thử với độ đo C_1.

- Hãy sinh các đường đi và các ca kiểm thử với độ đo C_2.
- Hãy sinh các ca kiểm thử để kiểm thử vòng lặp `while`.

13. Cho hàm được viết bằng ngôn ngữ C như Đoạn mã 6.5.

Đoạn mã 6.5: Mã nguồn của hàm `Sum`

```
int Sum(int a[], int n){
        int i, total = 0;
        for(i=0; i<n; i++)
                total = total + a[i];
        return total;
}
```

- Hãy xây dựng đồ thị dòng điều khiển cho hàm `Sum` ứng với độ đo C_1 và C_2.
- Hãy sinh các đường đi và các ca kiểm thử với độ đo C_1.
- Hãy sinh các đường đi và các ca kiểm thử với độ đo C_2.
- Hãy sinh các ca kiểm thử để kiểm thử vòng lặp `for`.

14. Cho hàm được viết bằng ngôn ngữ C như Đoạn mã 6.6.

Đoạn mã 6.6: Mã nguồn của hàm `Power`

```
double  Power(double x, int n)
{
        double  result = 1;
        for(int i=1; i< n; i++)
                result *= x;
        return result;
}
```

- Hãy xây dựng đồ thị dòng điều khiển cho hàm `Power` ứng với độ đo C_1 và C_2.
- Hãy sinh các đường đi và các ca kiểm thử với độ đo C_1.

- Hãy sinh các đường đi và các ca kiểm thử với độ đo C_2.
- Hãy sinh các ca kiểm thử để kiểm thử vòng lặp `for`.

15. Cho hàm được viết bằng ngôn ngữ C như Đoạn mã 6.7.

Đoạn mã 6.7: Mã nguồn của hàm `LaNamNhuan`

```
//tra ve 1 neu year la nham nhuan
//nguoc lai, tra ve 0
int LaNamNhuan(int year){
        if((year%400==0)
         ||(year%4==0 && year%100!=0))
                  return 1;
        return 0;
}
```

- Hãy xây dựng đồ thị dòng điều khiển cho hàm `LaNamNhuan` ứng với độ đo C_1 và C_2.
- Hãy sinh các đường đi và các ca kiểm thử với độ đo C_1.
- Hãy sinh các đường đi và các ca kiểm thử với độ đo C_2.
- Hãy xây dựng đồ thị dòng điều khiển cho hàm `LaNamNhuan` ứng với độ đo C_3.
- Hãy sinh các đường đi và các ca kiểm thử với độ đo C_3.

16. Cho hàm được viết bằng ngôn ngữ C như Đoạn mã 6.8.

Đoạn mã 6.8: Mã nguồn của hàm `SoKyTu`

```
//tra lai so ky tu trong xau
int SoKyTu(char a[]){
        int i, total = 0;
        int len = strlen(a);
        for (i = 0; i<len; i++){
                if(((a[i']>='A)&&(a[i']<='Z))
                 ||((a[i']>='a)&&(a[i']<='z)))
```

```
                        total ++;
        }//end for
        return total;
}
```

- Hãy xây dựng đồ thị dòng điều khiển cho hàm `SoKyTu` ứng với độ đo C_1 và C_2.
- Hãy sinh các đường đi và các ca kiểm thử với độ đo C_1.
- Hãy sinh các đường đi và các ca kiểm thử với độ đo C_2.
- Hãy xây dựng đồ thị dòng điều khiển cho hàm `SoKyTu` ứng với độ đo C_3
- Hãy sinh các đường đi và các ca kiểm thử với độ đo C_3.
- Hãy sinh các ca kiểm thử để kiểm thử vòng lặp `for`.

17. Cho hàm được viết bằng ngôn ngữ C như Đoạn mã 6.9.

Đoạn mã 6.9: Mã nguồn của hàm `NoiMang`

```
//Noi 2 mang da duoc sxep (a va b)
//vao mang c (duoc sxep)
void NoiMang(int a[], int n, int b[], int m,
             int c[], int &k){
        int i=0, j=0;
        k=0;
        while(i<n&&j<m)
        {
            if(a[i] <= b[j]){
                        c[k] = a[i];
                        i++;
                }else{
                        c[k] = b[j];
                        j++;
                }//end if
                k++;
```

```
        }//end while
        while(i<n){
                c[k]=a[i];
                i++;
                k++;
        }//end while
        while(j<m){
                c[k]=b[j];
                j++;
                k++;
        }//end while
}//the end
```

- Hãy sinh các ca kiểm thử để kiểm thử các vòng lặp kế tiếp nhau của hàm `NoiMang`.

18. Cho hàm được viết bằng C như Đoạn mã 6.10.

Đoạn mã 6.10: Mã nguồn của hàm `SelectionSort`

```
void SelectionSort(int a[], int size){
        int i,j;
        for(i=0; i<size-1; i++){
                int min = i;
                for(j=i+1; j<size; j++)
                        if(a[j]<a[min])
                                min = j;
                int tem = a[i];
                a[i] = a[min];
                a[min] = tem;
        }//end for
}//the end
```

- Hãy sinh các ca kiểm thử để kiểm thử các vòng lặp lồng nhau của hàm `SelectionSort`.

Chương 7

Kiểm thử dòng dữ liệu

Như đã trình bày trong chương 6, kiểm thử dòng điều khiển và kiểm thử dòng dữ liệu (data flow testing) được xem là hai phương pháp chủ yếu trong chiến lược kiểm thử hộp trắng nhằm phát hiện các lỗi tiềm tàng bên trong các chương trình/đơn vị chương trình. Phương pháp kiểm thử dòng điều khiển cho phép sinh ra các ca kiểm thử tương ứng với các đường đi (dòng điều khiển) của chương trình. Tuy nhiên, chỉ áp dụng phương pháp này là chưa đủ để phát hiện tất cả các lỗi tiềm ẩn bên trong chương trình. Trong thực tế, các lỗi thường hay xuất hiện liên quan đến việc gán và sử dụng các biến trong chương trình/đơn vị chương trình. Phương pháp kiểm thử dòng dữ liệu được đề xuất nhằm phát hiện những lỗi này. Bằng cách áp dụng cả hai phương pháp kiểm thử dòng điều khiển và kiểm thử dòng dữ liệu, chúng ta khá tự tin về chất lượng của sản phẩm phần mềm. Trong chương này, chúng ta sẽ tìm hiểu phương pháp kiểm thử dòng dữ liệu nhằm sinh ra các ca kiểm thử phục vụ mục đích trên. Ngoài ra, phương pháp kiểm thử dựa trên lát cắt - một phương pháp cải tiến của phương pháp kiểm thử dòng dữ liệu cũng sẽ được giới thiệu nhằm giảm độ phức tạp của giải pháp này.

7.1 Kiểm thử dựa trên gán và sử dụng các biến

7.1.1 Ý tưởng

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. Ví dụ, khi khai báo một biến `int tem = 0`, chúng ta hy vọng biến `tem` sẽ được sử dụng ở các câu lệnh tiếp theo trong đơn vị 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 (tương tự như dòng điều khiển được trình bày trong chương 6) 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 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 mục 7.1.2. 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.

7.1.2 Các vấn đề phổ biến về dòng dữ liệu

Trong quá trình lập trình, các lập trình viên có thể viết các câu lệnh "bất thường" hoặc không tuân theo chuẩn lập trình. Chúng ta gọi những bất thường liên quan đến việc khai báo, khởi tạo giá trị cho các biến và sử dụng chúng là các vấn đề về dòng dữ liệu của đơn vị chương trình. Ví dụ, một lập trình viên có thể sử dụng một biến mà không khởi tạo giá trị sau khi khai báo nó (`int x; if(x == 100){ ...}`).

Các vấn đề phổ biến về dòng dữ liệu có thể được phát hiện bằng phương pháp kiểm thử dòng dữ liệu tĩnh. Theo Fosdick và Osterweil [DJ76], các vấn đề này được chia thành ba loại như sau:

- **Gán giá trị rồi gán tiếp giá trị (Loại 1):** Ví dụ, Hình 7.1 chứa hai câu lệnh tuần tự `x = f1(y); x = f2(z);` với `f1` và `f2` là các hàm đã định nghĩa trước và `y, z` lần lượt là các tham số đầu vào của các hàm này. Chúng ta có thể xem xét và lý giải hai câu lệnh tuần tự này với các tình huống sau:

  - Khi câu lệnh thứ hai được thực hiện, giá trị của biến `x` được gán và câu lệnh đầu không có ý nghĩa.
  - Lập trình viên có thể có nhầm lẫn ở câu lệnh đầu. Câu lệnh này có thể là gán giá trị cho một biến khác như là `w = f1(y)`.
  - Có thể có nhầm lẫn ở câu lệnh thứ hai. Lập trình viên định gán giá trị cho một biến khác như là `w = f2(z)`.
  - Một hoặc một số câu lệnh giữa hai câu lệnh này bị thiếu. Ví dụ như câu lệnh `w = f3(x)`.

  Chỉ có lập trình viên và một số thành viên khác trong dự án mới có thể trả lời một cách chính xác vấn đề trên thuộc trường hợp nào trong bốn tình huống trên. Mặc dù vậy, những

vấn đề tương tự như ví dụ này là khá phổ biến và chúng ta cần phân tích mã nguồn để phát hiện ra chúng trước khi tiến hành các hoạt động kiểm thử động.

```
.
.
.
x = f1(y);
x = f2(z);
.
.
.
```

Hình 7.1: Tuần tự các câu lệnh có vấn đề thuộc loại 1.

- **Chưa gán giá trị nhưng được sử dụng (Loại 2):** Ví dụ, Hình 7.2 chứa ba câu lệnh tuần tự với y là một biến đã được khai báo và gán giá trị (`y = f(x1);`). Trong trường hợp này, biến `z` chưa được gán giá trị khởi tạo nhưng đã được sử dụng trong câu lệnh để tính giá trị của biến `x` (`x = y + z`). Chúng ta cũng có thể lý giải vấn đề này theo các tình huống sau:
  - Lập trình viên có thể bỏ quên lệnh gán giá trị cho biến `z` trước câu lệnh tính toán giá trị cho biến `x`. Ví dụ, `z = f2(x2)`, với `f2` là một hàm đã được định nghĩa và `x2` là một biến đã được khai báo và gán giá trị.
  - Có thể có sự nhầm lẫn giữa biến z với một biến đã được khai báo và gán giá trị. Ví dụ, `x = y + x2`.

- **Đã được khai báo và gán giá trị nhưng không được sử dụng (Loại 3):** Nếu một biến đã được khai báo và gán giá trị nhưng không hề được sử dụng (trong các câu lệnh tính toán hoặc trong các biểu thức điều kiện), chúng ta cần xem xét cẩn thận vấn đề này. Tương tự như các trường hợp trên, các tình huống sau có thể được sử dụng để lý giải cho vấn đề này:

```
.
.
.
y=f(x1);
int z;
x=y+z;
.
.
.
```

Hình 7.2: Tuần tự các câu lệnh có vấn đề thuộc loại 2.

- Có sự nhầm lẫn giữa biến này và một số biến khác được sử dụng trong chương trình. Trong thiết kế, biến này được sử dụng nhưng nó đã bị thay thế (do nhầm lẫn) bởi một biến khác.
- Biến này thực sự không được sử dụng trong chương trình. Lúc đầu lập trình viên định sử dụng nó như là một biến tạm thời hoặc biến trung gian nhưng sau đó lại không cần dùng. Lập trình viên này đã quên xóa các câu lệnh khai báo và gán giá trị cho biến này.

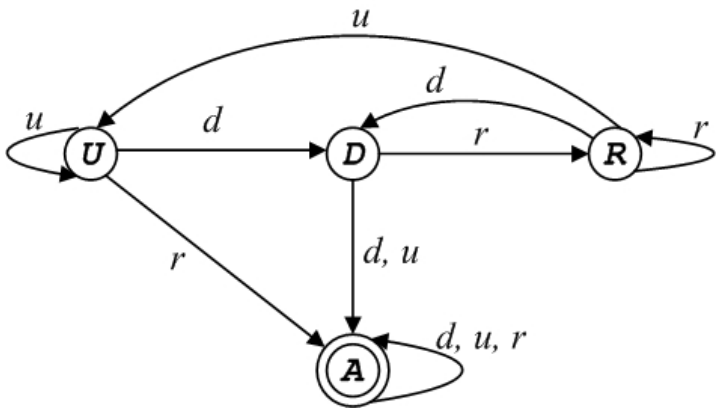
Huang [C.79] đã giới thiệu một phương pháp để xác định những bất thường trong việc sử dụng các biến dữ liệu bằng cách sử dụng sơ đồ chuyển trạng thái ứng với mỗi biến dữ liệu của chương trình. Các thành phần của sơ đồ chuyển trạng của một chương trình ứng với mỗi biến gồm:

- 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 7.3 mô tả sơ đồ chuyển trạng thái của một biến trong một chương trình/đơn vị chương trình. Ban đầu, biến này đã được khai báo và chưa được gán giá trị nên trạng thái của chương trình là U. Tại trạng thái này, nếu biến này được sử dụng (hành động r) thì chương trình có vấn đề và trạng thái của chương trình là A. Ngược lại, trạng thái U vẫn được giữ nguyên nếu các câu lệnh tiếp theo vẫn chưa chứa lệnh gán giá trị cho biến này (hành động u). Cho đến khi gặp câu lệnh gán giá trị cho biến này (hành động d), trạng thái của chương trình được chuyển thành D. Nếu biến này được sử dụng ở các câu lệnh tiếp theo (hành động r) thì trạng thái của chương trình chuyển thành R. Ngược lại, nếu các câu lệnh tiếp theo lại gán lại giá trị cho biến (hành động d) hoặc khai báo lại biến này và không gán giá trị cho nó (hành động u) thì xảy ra vấn đề và trạng thái của chương trình là A. Tại trạng thái này, mọi hành động (d, u và r) xảy ra đều không thay đổi trạng thái của chương trình. Tại trạng thái R, nếu biến này vẫn tiếp tục được sử dụng ở các lệnh tiếp theo (hành động r) thì trạng thái của chương trình vẫn không thay đổi. Ngược lại, nếu xuất hiện câu lệnh gán lại giá trị cho biến (hành động d) thì trạng thái của chương trình quay về D. Trong trường hợp xuất hiện câu lệnh khai báo lại biến này và không gán giá trị cho nó (hành động u) thì chương trình được chuyển từ trạng thái R sang trạng thái U.

Như vậy, 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



Hình 7.3: Sơ đồ chuyển trạng thái của một biến.

hiện các vấn đề này, chúng ta sẽ tiến hành xây dựng sơ đồ chuyển trạng thái ứng với mỗi biến như Hình 7.3. Nếu trạng thái A xuất hiện thì chương trình có vấn đề về dòng dữ liệu. Trong trường hợp này, chúng ta cần kiểm tra lại mã nguồn, tìm nguyên nhân của tình huống này và sửa lỗi. Tuy nhiên, cho dù trạng thái lỗi (trạng thái A) không xuất hiện trong quá trình phân tích chương trình, chúng ta vẫn không đảm bảo được rằng chương trình không còn lỗi. Các lỗi có thể xảy ra trong quá trình gán/gán lại giá trị cho các biến và trong quá trình sử dụng chúng (trong các câu lệnh tính toán, trong các biểu thức điều kiện, .v.v.). Để phát hiện những lỗi này, chúng ta cần phương pháp kiểm thử dòng dữ liệu động. Phương pháp này sẽ được trình bày chi tiết ở các mục tiếp theo.

7.1.3 Tổng quan về kiểm thử dòng dữ liệu động

Kiểm thử dòng dữ liệu tĩnh không đảm bảo phát hiện tất cả các lỗi liên quan đến việc khởi tạo, gán giá trị mới và sử dụng các biến (trong các câu lệnh tính toán và các biểu thức điều kiện như trong các lệnh rẽ nhánh và lặp). Nó được xem như một bước tiền xử lý mã nguồn trước khi áp dụng phương pháp kiểm thử dòng dữ liệu

động. Các lỗi còn lại (không được phát hiện bởi kiểm thử dòng dữ liệu tĩnh) sẽ được phát hiện bởi phương pháp này.

Các biến xuất hiện trong một chương trình theo một số trường hợp như: khởi tạo, gán giá trị mới, tính toán, và dùng làm điều khiển trong các biểu thức điều kiện của các lệnh rẽ nhánh và lặp. Có hai lý do chính khiến chúng ta phải tiến hành kiểm thử dòng dữ liệu của chương trình:

- Chúng ta cần chắc chắn rằng một biến phải được gán đúng giá trị, tức là chúng ta phải xác định được một đường đi của biến từ một điểm bắt đầu nơi nó được định nghĩa đến điểm mà biến đó được sử dụng. Mỗi khi chưa có các ca kiểm thử để kiểm tra đường đi này, chúng ta không thể tự tin khẳng định là biến này đã được gán giá trị đúng [SJ85].

- Ngay cả khi gán đúng giá trị cho biến thì các giá trị được sinh ra chưa chắc đã chính xác do tính toán hoặc các biểu thức điều kiện sai (biến được sử dụng sai).

Để áp dụng phương pháp kiểm thử dòng dữ liệu động, chúng ta phải xác định các đường dẫn chương trình có một điểm đầu vào và một điểm đầu ra sao cho nó bao phủ việc gán giá trị và sử dụng mỗi biến của chương trình/đơn vị chương trình cần kiểm thử. Cụ thể, chúng ta cần thực hiện các bước sau:

- Xây dựng đồ thị dòng dữ liệu của chương trình/đơn vị chương trình.

- Chọn một hoặc một số tiêu chí kiểm thử dòng dữ liệu.

- Xác định các đường dẫn chương trình phù hợp với tiêu chí kiểm thử đã chọn.

- Lấy ra các biểu thức điều kiện từ tập các đường đi, thực hiện giải các biểu thức điều kiện để có được các giá trị đầu vào

cho các ca kiểm thử tương ứng với các đường đi này và tính toán giá trị đầu ra mong đợi của mỗi ca kiểm thử.

- Thực hiện các ca kiểm thử để xác định các lỗi (có thể có) của chương trình.
- Sửa các lỗi (nếu có) và thực hiện lại tất cả các ca kiểm thử trong trường hợp bước trên phát hiện ra lỗi.

Như vậy, các bước trong quy trình kiểm thử dòng dữ liệu động cũng tương tự như các bước trong quy trình kiểm thử dòng điều khiển trong chương 6. Tuy nhiên, đồ thị dòng điều khiển hoàn toàn khác với đồ thị dòng dữ liệu. Hơn nữa, các tiêu chí kiểm thử (độ đo kiểm thử) ứng với hai kỹ thuật này cũng khác nhau. Kết quả là, các đường đi và phương pháp chọn chúng từ đồ thị cũng khác nhau. Các vấn đề này sẽ được trình bày chi tiết ở các mục tiếp theo.

7.1.4 Đồ thị dòng dữ liệu

Trong mục này, chúng ta sẽ tìm hiểu về đồ thị dòng dữ liệu và cách xây dựng nó từ đơn vị chương trình. Trong thực tế, chúng ta có thể sử dụng các tiện ích của các chương trình dịch để xây dựng đồ thị này một cách tự động. Đồ thị dòng dữ liệu của một chương trình/đơn vị chương trình sử dụng các khái niệm liên quan đến việc định nghĩa và sử dụng các biến.

Định nghĩa 7.1. (Định nghĩa của một biến.) Một câu lệnh thực hiện việc gán giá trị cho một biến được gọi là câu lệnh định nghĩa của biến đó (ký hiệu là *def*). Ví dụ, trong hàm `VarTypes` viết bằng ngôn ngữ C như Đoạn mã 7.1, câu lệnh `i = x` (dòng 3) là một ví dụ về việc định nghĩa của biến `i`.

Định nghĩa 7.2. (Chưa định nghĩa một biến.) Một câu lệnh khai báo một biến nhưng chưa thực hiện việc gán giá trị cho biến đó được

gọi là *undef* với biến đó. Ví dụ, trong hàm `VarTypes` viết bằng ngôn ngữ C như Đoạn mã 7.1, câu lệnh `iptr = malloc(sizeof(int));` (dòng 4) là một ví dụ về việc chưa định nghĩa biến `iptr`. Câu lệnh tiếp theo (`*iptr = i + x;` (dòng 5)) là một định nghĩa của biến này. Tuy nhiên, câu lệnh ở dòng 9 lại định nghĩa lại biến `iptr` và vì vậy câu lệnh này là một ví dụ khác về việc chưa định nghĩa biến này.

Đoạn mã 7.1: Ví dụ về định nghĩa và sử dụng các biến

```
int VarTypes(int x, int y){
1.      int i;
2.      int *iptr;
3.      i = x;
4.      iptr = malloc(sizeof(int));
5.      *iptr = i + x;
6.      if (*iptr > y)
7.              return (x);
8.      else {
9.              iptr = malloc(sizeof(int));
10.             *iptr = x + y;
11.             return(*iptr);
12.     }//end if
}//the end
```

Định nghĩa 7.3. (*use.*) Một câu lệnh sử dụng một biến (để tính toán hoặc để kiểm tra các điều kiện) được gọi là *use* của biến đó.

Định nghĩa 7.4. (*c-use.*) Một câu lệnh sử dụng một biến để tính toán giá trị của một biến khác được gọi là *c-use* với biến đó. Ví dụ, trong hàm `VarTypes` như Đoạn mã 7.1, câu lệnh `*iptr = i + x;` (dòng 5) là *c-use* ứng với biến `i` và biến `x`.

Định nghĩa 7.5. (*p-use.*) Một câu lệnh sử dụng một biến trong các biểu thức điều kiện (câu lệnh rẽ nhánh, lặp, v.v.) được gọi là *p-use* với biến đó.

Ví dụ, câu lệnh `if (*iptr > y) ...` trong hàm `VarTypes` (dòng lệnh 6) như Đoạn mã 7.1 là *p-use* ứng với biến `iptr` và biến `y`.

Định nghĩa 7.6 (Đồ thị dòng dữ liệu (Data Flow Graph - DFG))**.** Đồ thị dòng dữ liệu của một chương trình/đơn vị chương trình là một đồ thị có hướng $G = <N, E>$, với:

- N là tập các đỉnh tương ứng với các câu lệnh def hoặc *c-use* của các biến được sử dụng trong đơn vị chương trình. Đồ thị G có hai đỉnh đặc biệt là đỉnh bắt đầu (tương ứng với lệnh def của các biến tham số) và đỉnh kết thúc đơn vị chương trình, và
- E là tập các cạnh tương ứng với các câu lệnh *p-use* của các biến.

Đoạn mã 7.2: Mã nguồn của hàm `ReturnAverage` bằng ngôn ngữ C

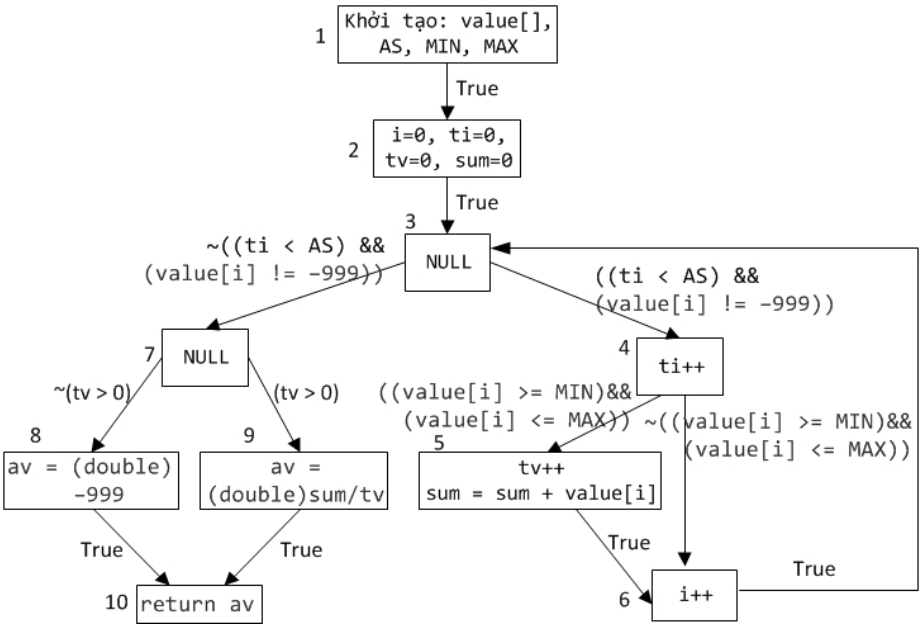
```
double ReturnAverage(int value[], int AS, int MIN,
                     int MAX){

      int i, ti, tv, sum;
      double av;
      i = 0; ti = 0; tv = 0; sum = 0;

      while (ti < AS && value[i] != -999) {
         ti++;
         if (value[i] >= MIN && value[i] <= MAX){
            tv++;
            sum = sum + value[i];
         }
         i++;
      }//end while

      if (tv > 0)
```

```
            av = (double)sum/tv;
        else
            av = (double) -999;

        return (av);
} //the end
```

Chúng ta sẽ tìm hiểu cách xây dựng đồ thị dòng dữ liệu của một đơn vị chương trình thông qua ví dụ về hàm `ReturnAverage` bằng ngôn ngữ C trong Đoạn mã 7.2. Đồ thị dòng dữ liệu của hàm này được trình bày ở Hình 7.4.

Trong đồ thị này, đỉnh bắt đầu (đỉnh 1) thể hiện việc định nghĩa (*def*) các biến tham số `value[], AS, MIN, MAX`. Đỉnh 2 tương ứng với việc định nghĩa các biến cục bộ `i, ti, tv, sum`. Tiếp đến, chúng ta sử dụng đỉnh `NULL` (đỉnh 3) với mục đích đây là điểm bắt đầu của vòng lặp `while`. Chú ý rằng đỉnh `NULL` không thực hiện bất cứ tính toán hay định nghĩa cho biến nào cả. Đỉnh này cũng sẽ là điểm kết thúc của vòng lặp này sau khi biểu thức điều kiện của vòng lặp là sai. Câu lệnh `ti++;` được biểu diễn bằng đỉnh 4. Cạnh (3, 4) biểu diễn biểu thức điều kiện của vòng lặp (`((ti < AS) && (value[i] != -999))`). Ý nghĩa của cạnh này là nếu biểu thức này đúng thì chuyển từ đỉnh 3 sang thực hiện đỉnh 4. Các câu lệnh `tv++;` và `sum = sum + value[i];` được biểu diễn bởi đỉnh 5. Để chuyển từ đỉnh 4 sang đỉnh 5 (cạnh (4, 5)), điều kiện tương ứng với cạnh này phải đúng (`((value[i] >= MIN) && (value[i] <= MAX))` là đúng). Tiếp theo, đỉnh 6 biểu diễn câu lệnh `i++;`. Trong trường hợp `((value[i]>=MIN)&&(value[i]<= MAX))` là sai, đồ thị sẽ chuyển từ đỉnh 4 sang đỉnh 6 (cạnh (4, 6)). Từ đỉnh 5, đồ thị sẽ chuyển đến đỉnh 6 (cạnh (5, 6)) vì không có biểu thức điều kiện của cạnh này. Vòng lặp `while` sẽ kết thúc khi điều kiện `((ti < AS) && (value[i] != -999))` là sai. Khi đó, đồ thị sẽ chuyển từ đỉnh 3 sang đỉnh 7 (cạnh (3, 7)). Đỉnh 7 là đỉnh `NULL` tương ứng với câu lệnh `if (tv > 0)`. Đỉnh 8 là biểu diễn của câu

lệnh `av = (double) -999;` trong khi câu lệnh `av = (double)sum/tv;` được biểu diễn bằng đỉnh 9. Từ đỉnh 7, nếu điều kiện `tv > 0` là đúng thì đồ thị chuyển sang đỉnh 9 (cạnh (7, 9)). Ngược lại, đồ thị chuyển từ đỉnh 7 sang đỉnh 8 (cạnh (7, 8)). Cuối cùng, đỉnh 10 được sử dụng để biểu diễn câu lệnh `return(av);`. Từ các đỉnh 8 và 9, đồ thị chuyển đến đỉnh 10 vì không có biểu thức điều kiện cho các cạnh (8, 10) và (9, 10).



Hình 7.4: Đồ thị dòng dữ liệu của hàm `ReturnAverage`.

7.1.5 Các khái niệm về dòng dữ liệu

Sau khi xây dựng đồ thị dòng dữ liệu của đơn vị chương trình, chúng ta cần xác định các đường đi của đơn vị chương trình của mỗi biến dữ liệu ứng với các độ đo kiểm thử (sẽ được trình bày trong mục 7.1.6). Trong mỗi đường dẫn này, biến dữ liệu được định nghĩa tại một đỉnh nào đó và được sử dụng tại các câu lệnh tiếp

theo ứng với các đỉnh hoặc các cạnh của đường đi này. Trong mục này, chúng ta sẽ tìm hiểu một số khái niệm cơ bản về dòng dữ liệu. Các khái niệm này sẽ được sử dụng để phục vụ mục đích trên.

Định nghĩa 7.7. (*Global c-use.*) Giả sử biến x được sử dụng để tính toán (*c-use*) tại đỉnh i của đồ thị dòng dữ liệu. Việc sử dụng biến x tại đỉnh i được gọi là *Global c-use* nếu x đã được định nghĩa ở các đỉnh trước đó.

Ví dụ 7.8. Trong đồ thị dòng dữ liệu của hàm `ReturnAverage` được mô tả ở Đoạn mã 7.2, việc sử dụng biến `tv` tại đỉnh 9 là *Global c-use* vì biến này đã được định nghĩa tại các đỉnh 2 và 5.

Định nghĩa 7.9. (*Def-clear path.*) Giả sử biến x được định nghĩa (*def*) tại đỉnh i và được sử dụng tại đỉnh j. Một đường đi từ i đến j ký hiệu là (i - n_1 - ... - n_m - j) với $m \geq 0$ được gọi là *Def-clear path* ứng với biến x nếu biến này không được định nghĩa tại các đỉnh từ n_1 đến n_m.

Ví dụ 7.10. Trong đồ thị dòng dữ liệu của hàm `ReturnAverage` được mô tả ở Hình 7.4, đường đi (2 - 3 - 4 - 5) là một *Def-clear path* ứng với biến `tv` vì biến này được định nghĩa tại đỉnh 2, được sử dụng tại đỉnh 5, và không được định nghĩa tại các đỉnh 3 và 4. Tương tự, đường đi (2 - 3 - 4 - 6 - 3 - 4 - 6 - 3 - 4 - 5) cũng là một *Def-clear path* ứng với biến `tv`.

Định nghĩa 7.11. (*Global def.*) Một đỉnh i được gọi là *Global def* của biến x nếu đỉnh này định nghĩa biến x (*def*) và có một *Def-clear path* của x từ đỉnh i tới đỉnh chứa một *Global c-use* hoặc cạnh chứa một *p-use* của biến này.

Bảng 7.1 liệt kê tất cả các *Global def* và *Global c−use* xuất hiện trong đồ thị dòng dữ liệu của hàm `ReturnAverage` như mô tả ở Hình 7.4. Trong bảng này, $def(i)$ được sử dụng để chỉ tập các biến có *Global def* tại đỉnh i. Tương tự, $c{-}use(i)$ chỉ tập các biến có

Bảng 7.1: $def()$ **và** c-$use()$ **của các đỉnh trong Hình 7.4**

Đỉnh i	$def(i)$	c-$use(i)$
1	{value, AS, MIN, MAX}	{}
2	{i, ti, tv, sum}	{}
3	{}	{}
4	{ti}	{ti}
5	{tv, sum}	{tv, i, sum, value}
6	{i}	{i}
7	{}	{}
8	{av}	{}
9	{av}	{sum, tv}
10	{}	{av}

Global $c-use$ tại đỉnh i. Tất cả các điều kiện ứng với các cạnh và các $p-use$ xuất hiện trong đồ thị dòng dữ liệu ở Hình 7.4 cũng được liệt kê trong Bảng 7.2, với $p-use(i, j)$ là tập các biến có $p-use$ tại cạnh (i, j).

Bảng 7.2: Các điều kiện và p-$use()$ **của các cạnh trong Hình 7.4**

(i, j)	**Điều kiện tại cạnh** (i, j)	p-$use(i, j)$
(1, 2)	True	{}
(2, 3)	True	{}
(3, 4)	(ti<AS)&&(value[i] != -999)	{i,ti,AS,value}
(4, 5)	(value[i]>=MIN)&&(value[i]<= MAX)	{i,MIN,MAX,value}
(4, 6)	∼((value[i]>=MIN)&&(value[i] <= MAX))	{i,MIN,MAX,value}
(5, 6)	True	{}
(6, 3)	True	{}
(3, 7)	∼((ti<AS) && (value[i] != -999))	{i, ti, AS, value}
(7, 8)	∼(tv > 0)	{tv}
(7, 9)	(tv > 0)	{tv}
(8, 10)	True	{}
(9, 10)	True	{}

Định nghĩa 7.12. (*Simple path.*) Một đường đi trong đồ thị dòng dữ liệu được gọi là một *Simple path* nếu các đỉnh chỉ xuất hiện đúng một lần trừ đỉnh đầu và đỉnh cuối.

Ví dụ 7.13. Trong đồ thị dòng dữ liệu của hàm `ReturnAverage` được mô tả ở Hình 7.4, các đường đi (2 - 3 - 4 - 5) và (3 - 4 - 6 - 3) là các *Simple paths*.

Định nghĩa 7.14. (*Loop-free path.*) Một đường đi trong đồ thị dòng dữ liệu được gọi là một *Loop-free path* nếu các đỉnh chỉ xuất hiện đúng một lần.

Định nghĩa 7.15. (*Complete-path.*) Một đường đi được gọi là một *Complete-path* nếu nó có điểm bắt đầu và điểm kết thúc chính là điểm bắt đầu và điểm kết thúc của đồ thị dòng dữ liệu.

Định nghĩa 7.16. (*Du-path.*) Một đường đi (n_1 - n_2 - ... - n_j - n_k) được gọi là một *Du-path* (definition-use path) ứng với biến x nếu đỉnh n_1 là *Global def* của biến x và:

- đỉnh n_k có một *Global c-use* với biến x và (n_1 - n_2 - ... - n_j - n_k) là một *Def-clear simple path* với biến x, hoặc
- cạnh (n_j, n_k) có *p-use* với biến x và (n_1 - n_2 - ... - n_j) là *Def-clear loop-free path* với biến này.

Ví dụ 7.17. Trong đồ thị dòng dữ liệu của hàm `ReturnAverage` (Hình 7.4), đường đi (2 - 3 - 4 - 5) là một *Du-path* ứng với biến `tv` vì đỉnh 2 là *Global def* của biến `tv`, đỉnh 5 có *Global c-use* với biến này và (2 - 3 - 4 - 5) là một *Def-clear simple path* với biến `tv`. Một ví dụ khác, đường đi (2 - 3 - 7 - 9) cũng là một *Du-path* ứng với biến `tv` vì đỉnh 2 là *Global def* của biến này, cạnh (7, 9) có *p-use* với biến `tv` và (2 - 3 - 7) là một *Def-clear loop-free path* với biến `tv`.

7.1.6 Các độ đo cho kiểm thử dòng dữ liệu

Như đã trình bày ở mục 7.1.3, các độ đo hay các tiêu chí kiểm thử dòng dữ liệu là đầu vào cùng với đồ thị dòng dữ liệu nhằm xác định các đường đi cho mục đích kiểm thử tương ứng. Trong mục này, chúng ta sẽ tìm hiểu các độ đo phổ biến đang được sử dụng trong kiểm thử dòng dữ liệu. Trước hết ta có ba độ đo đơn giản là *All-defs*, *All-c-uses*, và *All-p-uses* tương ứng với tất cả các đường đi, đường đi qua tất cả các đỉnh, và đường đi qua tất cả các cạnh. Các độ đo tiếp theo sẽ giả sử ta đã xác định được các đỉnh xác định và sử dụng các *Du-path* cho từng biến. Các định nghĩa sau cũng giả sử T là một tập hợp các (đoạn) đường trong đồ thị dòng dữ liệu $G = <N, E>$ và V là tập các biến được sử dụng trong đơn vị chương trình.
All-defs: Mỗi một biến $x \in V$ và mỗi đỉnh $i \in N$, giả sử x có một *Global def* tại i, chọn một *Complete-path* chứa một *Def-clear path* từ đỉnh i tới đỉnh j sao cho tại j có chứa một *Global c-use* của x, hoặc cạnh (j, k) chứa một *p-use* của biến x. Điều này có nghĩa là đối với mỗi một định nghĩa (def) của x tại một đỉnh ta cần ít nhất một đường đi xuất phát từ đỉnh đó tới một đỉnh khác sử dụng biến x sao cho đường đi này chứa một *Def-clear path* của biến đó và thuộc về một *Complete-path* nào đó.

Ví dụ 7.18. Biến `tv` có hai *Global def* tại các đỉnh 2 và 5 (Hình 7.4, Bảng 7.1 và Bảng 7.2). Trước hết, ta quan tâm đến *Global def* tại đỉnh 2. Chúng ta thấy rằng có một *Global c-use* của biến `tv` tại đỉnh 5 và tồn tại một *Def-clear path* (2 - 3 - 4 - 5) từ đỉnh 2 tới đỉnh 5. Ta cũng có được một *Complete-path* (1 – 2 – 3 – 4 – 5 – 6 – 3 – 7 – 9 – 10) chứa đường đi này. Do vậy ta kết luận (2 - 3 - 4 - 5) thỏa mãn độ đo *All-defs*. Tương tự, đường đi (2 - 3 - 7 - 8) của biến `tv` cũng thỏa mãn độ đo *All-defs* do có một *Global def* tại đỉnh 2, có một *p-use* tại cạnh (7, 8), có một *Def-clear path* là (2 - 3 - 7 - 8) từ đỉnh 2 tới cạnh (7, 8) và nó thuộc về một *Complete-path*

(1 - 2 - 3 - 7 - 8 - 10). Quan tâm đến *Global def* tại đỉnh 5 và có một *Global c-use* tại đỉnh 9, (5 - 6 - 3 - 7 - 9) là một *Def-clear path* và tồn tại một *Complete-path* là (1 - 2 - 3 - 4 - 5 - 6 - 3 - 7 - 9 - 10) chứa đường đi này. Vì vậy (5 - 6 - 3 - 7 - 9) cũng thỏa mãn độ đo *All-defs*. Để quá trình kiểm thử dòng dữ liệu thỏa mãn độ đo *All-defs*, chúng ta cần tiến hành một cách tương tự với các biến còn lại.

All-c-uses: Với mỗi một biến x và mỗi đỉnh i sao cho i là *Global def* với biến x, chọn các *Complete-path* bao gồm các *Def-clear path* từ đỉnh i tới tất cả các đỉnh j sao cho j là *Global c-use* của x. Điều này có nghĩa là cứ mỗi định nghĩa (def) của x ta tìm tất cả các đường đi xuất phát từ def của x tới tất cả các *c-use* của biến x sao cho các đường đi này có chứa một *Def-clear path* của x và thuộc về một *Complete-path* nào đó.

Ví dụ 7.19. Ta sẽ tìm tất cả các đường đi thỏa mãn độ đo *All-c-uses* ứng với biến `ti` (Hình 7.4, Bảng 7.1, và Bảng 7.2). Chúng ta tìm thấy hai *Global def* của biến này tại các đỉnh 2 và 4. Với đỉnh 2, có một *Global c-use* của biến `ti` tại đỉnh 4. Tuy nhiên, với *Global def* của biến này tại đỉnh 4, ta không tìm thấy *Global c-use* nào của biến `ti`. Từ đỉnh 2, ta có một *Def-clear path* tới đỉnh 4 là (2 - 3 - 4). Chúng ta có thể tìm thấy bốn *Complete-path* từ đồ thị dòng dữ liệu (Hình 7.4) chứa đường đi này như sau:

- (1 - 2 - 3 - 4 - 5 - 6 - 3 - 7 - 8 - 10),
- (1 - 2 - 3 - 4 - 5 - 6 - 3 - 7 - 9 - 10),
- (1 - 2 - 3 - 4 - 6 - 3 - 7 - 8 - 10), và
- (1 - 2 - 3 - 4 - 6 - 3 - 7 - 9 - 10).

Chúng ta có thể chọn một hoặc một số trong bốn đường đi trên để đảm bảo độ đo này ứng với biến `ti`. Để quá trình kiểm thử dòng

dữ liệu thỏa mãn độ đo *All-c-uses*, chúng ta cần tiến hành một cách tương tự với các biến còn lại `(i, tv,` và `sum)`.

All-p-uses: Với mỗi một biến x và mỗi đỉnh i sao cho i là *Global* def với biến x, chọn các *Complete-path* bao gồm các Def-*clear path* từ đỉnh i tới tất cả các cạnh (j, k) sao cho có một *p-use* của x tại cạnh này. Điều này có nghĩa là cứ mỗi định nghĩa (def) của x ta tìm tất cả các đường đi xuất phát từ def của x tới tất cả các *p-use* của biến đó sao cho các đường đi này có chứa một Def-*clear path* của x và thuộc về một *Complete-path* nào đó.

Ví dụ 7.20. Ta sẽ tìm tất cả các đường đi thỏa mãn độ đo *All-p-uses* ứng với biến `tv` (Hình 7.4, Bảng 7.1, và Bảng 7.2). Biến `tv` có hai *Global* def tại các đỉnh 2 và 5. Tại đỉnh 2, ta có hai *p-use* của biến này tại các cạnh (7, 8) và (7, 9). Dễ thấy (2 - 3 - 7 - 8) là một Def-*clear path* của `tv` từ đỉnh 2 đến cạnh (7, 8) và (2 - 3 - 7 - 9) là một Def-*clear path* của `tv` từ đỉnh 2 đến cạnh (7, 9). Tương tự, (5 - 6 - 3 - 7 - 8) là một Def-*clear path* của `tv` từ đỉnh 5 đến cạnh (7, 8) và (5 - 6 - 3 - 7 - 9) là một Def-*clear path* của `tv` từ đỉnh 5 đến cạnh (7, 9). Chúng ta có thể tìm thấy bốn *Complete-path* từ đồ thị dòng dữ liệu (Hình 7.4) chứa các đường đi này như sau:

- (1 - <u>2 - 3 - 7 - 8</u> - 10),
- (1 - <u>2 - 3 - 7 - 9</u> - 10),
- (1 - 2 - 3 - 4 - <u>5 - 6 - 3 - 7 - 8</u> - 10), và
- (1 - 2 - 3 - 4 - <u>5 - 6 - 3 - 7 - 9</u> - 10).

All-p-uses/Some-c-uses: Độ đo này tương tự như độ đo *All-p-uses* ngoại trừ trường hợp khi có một nghĩa (def) của biến x mà không có một *p-use* của biến này. Trong trường hợp này, ta sử dụng độ đo *Some-c-uses* được định nghĩa như sau.

Định nghĩa 7.21. (*Some-c-uses.*) Với mỗi một biến x và mỗi đỉnh i sao cho i là *Global def* với biến x, chọn các *Complete-path* bao gồm các *Def-clear path* từ đỉnh i tới một số đỉnh j sao cho j là *Global c-use* của x.

Ví dụ 7.22. Ta sẽ tìm tất cả các đường đi thỏa mãn độ đo *All-p-uses/Some-c-uses* ứng với biến `i` (Hình 7.4, Bảng 7.1, và Bảng 7.2). Biến `i` có hai *Global def* tại các đỉnh 2 và đỉnh 6. Dễ thấy rằng không có *p-use* của biến này (Hình 7.4). Vì vậy, ta quan tâm đến độ đo *Some-c-uses* của biến `i`. Tại đỉnh 2, có một *Global c-use* của `i` tại đỉnh 6 và có một *Def-clear path* (2 - 3 - 4 - 5 - 6). Vì vậy, để thỏa mãn độ đo *All-p-uses/Some-c-uses* với biến này, ta chọn *Complete-path* (1 - 2 - 3 - 4 - 5 - 6 - 3 - 7 - 9 - 10) chứa đường đi trên.

All-c-uses/Some-p-uses: Độ đo này tương tự như độ đo *All-c-uses* ngoại trừ trường hợp khi có một nghĩa (*def*) của biến x mà không có một *Global c-use* của biến này. Trong trường hợp này, ta sử dụng độ đo *Some-p-uses* được định nghĩa như sau.

Định nghĩa 7.23. (*Some-p-uses.*) Với mỗi một biến x và mỗi đỉnh i sao cho i là *Global def* với biến x, chọn các *Complete-path* bao gồm các *Def-clear path* từ đỉnh i tới một số cạnh (j, k) sao cho có một *p-use* của x tại cạnh này.

Ví dụ 7.24. Ta sẽ tìm tất cả các đường đi thỏa mãn độ đo *All-c-uses/Some-p-uses* ứng với biến `AS` (Hình 7.4, Bảng 7.1, và Bảng 7.2). Biến `AS` chỉ có một *Global def* tại đỉnh 1. Dễ thấy rằng không có *Global c-use* của biến này (Hình 7.4). Vì vậy, ta quan tâm đến độ đo *Some-p-uses* của biến `AS`. Từ đỉnh 1, có các *p-use* của `AS` tại các cạnh (3, 7) và (3, 4) và có các *Def-clear path* tương ứng với hai cạnh này là (1 - 2 - 3 - 7) và (1 - 2 - 3 - 4). Có rất nhiều *Complete-path* chứa hai đường đi này. Ví dụ (1 - 2 - 3 - 4 - 5 - 6 - 3 - 7 - 9 - 10).

All-uses: Độ đo này bao gồm các đường đi được sinh ra từ các độ đo *All-p-uses* và *All-c-uses* (như đã định nghĩa ở trên). Điều này có nghĩa là với mỗi việc sử dụng (*c-use* hoặc *p-use*) của một biến thì có một đường đi từ định nghĩa (def) của biến đó tới các sử dụng của nó.
All-du-paths: Với mỗi một biến x và mỗi đỉnh i sao cho i là *Global* def với biến x, chọn các *Complete-path* chứa tất cả các *Du-path* từ đỉnh i tới:

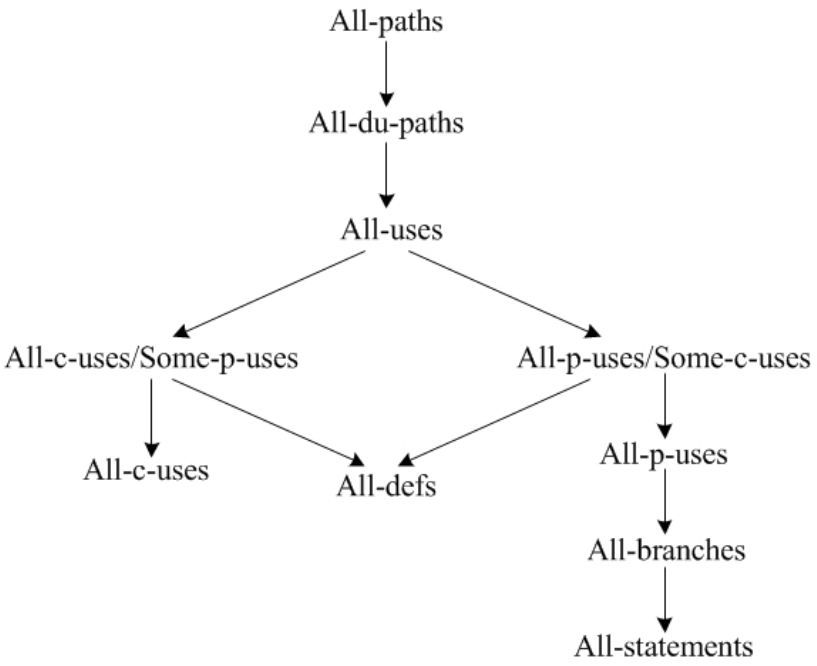
- tất cả các đỉnh j sao cho có một *Global c-use* của biến x tại j, và
- tất cả các cạnh (j, k) sao cho có một *p-use* của biến x tại (j, k).

So sánh các độ đo của kiểm thử dòng dữ liệu: Sau khi tìm hiểu các độ đo về kiểm thử dòng dữ liệu, chúng ta cần so sánh mối quan hệ giữa chúng để trả lời câu hỏi "chúng ta nên chọn độ đo nào cho kiểm thử dòng dữ liệu?". Với mỗi cặp độ đo ở trên, chúng có thể so sánh với nhau hoặc không. Rapps và Weyuker [SJ85] đã định nghĩa mối quan hệ "*bao gồm*" giữa hai độ đo như sau.

Định nghĩa 7.25. (Mối quan hệ bao gồm.) Cho hai độ đo c_1 và c_2, ta nói c_1 bao gồm c_2 nếu mọi đường đi đầy đủ (*Complete-paths*) sinh ra từ đồ thị dòng dữ liệu thỏa mãn c_1 thì cũng thỏa mãn c_2.

Định nghĩa 7.26. (Mối quan hệ bao gồm chặt.) Cho hai độ đo c_1 và c_2, ta nói c_1 bao gồm chặt c_2, ký hiệu là $c_1 \longrightarrow c_2$, nếu c_1 bao gồm c_2 và tồn tại một số đường đi đầy đủ sinh ra từ đồ thị dòng dữ liệu thỏa mãn c_2 nhưng không thỏa mãn c_1.

Dễ thấy mối quan hệ *bao gồm chặt* ($\longrightarrow$) có tính bắc cầu. Hơn nữa, với hai độ đo c_1 và c_2, có thể c_1 không bao gồm chặt c_2 và c_2 cũng không bao gồm chặt c_1. Trong trường hợp này ta nói hai độ đo này là không so sánh được. Frankl và Weyuker [GJ88] đã tổng



Hình 7.5: Mối quan hệ giữa các độ đo cho kiểm thử dòng dữ liệu.

kết các mối quan hệ giữa các độ đo cho kiểm thử dòng dữ liệu trong Hình 7.5. Với các độ đo như đã trình bày ở trên, *All-du-paths* là độ đo tốt nhất. Độ đo này bao gồm chặt độ đo *All-uses*. Tương tự, độ đo *All-uses* bao gồm chặt hai độ đo *All-p-uses/Some-c-uses* và *All-c-uses/Some-p-uses* trong khi chúng bao gồm chặt độ đo $All\text{-}defs$. Hơn nữa, độ đo *All-p-uses/Some-c-uses* bao gồm chặt độ đo *All-p-uses* trong khi độ đo *All-c-uses/Some-p-uses* bao gồm chặt độ đo *All-c-uses*. Tuy nhiên, chúng ta không thể tìm thấy mối quan hệ *bao gồm chặt* giữa hai độ đo *All-c-uses* và *All-p-uses*.

7.1.7 Sinh các ca kiểm thử

Để tiến hành phương pháp kiểm thử dòng dữ liệu, trước hết chúng ta phải sinh đồ thị dòng dữ liệu của đơn vị chương trình. Với độ đo kiểm thử C, chúng ta sẽ xác định tất cả các đường đi đầy

đủ (*Complete-paths*) thỏa mãn độ đo này. Tuy nhiên, không phải đường đi nào cũng có thể tìm được một bộ dữ liệu đầu vào để nó được thực thi khi chạy đơn vị chương trình. Nếu tồn tại một bộ dữ liệu đầu vào như vậy thì đường đi tương ứng được gọi là đường đi thực thi được. Như vậy, bài toán còn lại hiện nay là làm thế nào để sinh được bộ đầu vào cho từng đường đi đầy đủ trên. Bộ đầu vào này cùng với giá trị đầu ra mong đợi sẽ là ca kiểm thử cho đường đi này. Để trả lời câu hỏi này, ta xét ví dụ sau.

Ví dụ 7.27. Xét đường đi đầy đủ (1 - 2 - 3 - 4 - 5 - 6 - 3 - 7 - 9) từ đồ thị dòng dữ liệu trong Hình 7.4. Từ đường đi này, ta sẽ xác định các biểu thức thuộc các *p-uses* nằm trên các cạnh của nó. Cụ thể, ta có các biểu thức sau:

1. `((ti<AS) && (value[i]!=-999))` (thuộc cạnh (3, 4)),
2. `((value[i] >= MIN) && (value[i] <= MAX))` (điều kiện này thuộc cạnh (4, 5)),
3. ∼`((ti<AS) && (value[i] != -999))` (thuộc cạnh (3, 7)), và
4. `tv > 0` (thuộc cạnh (7, 9))

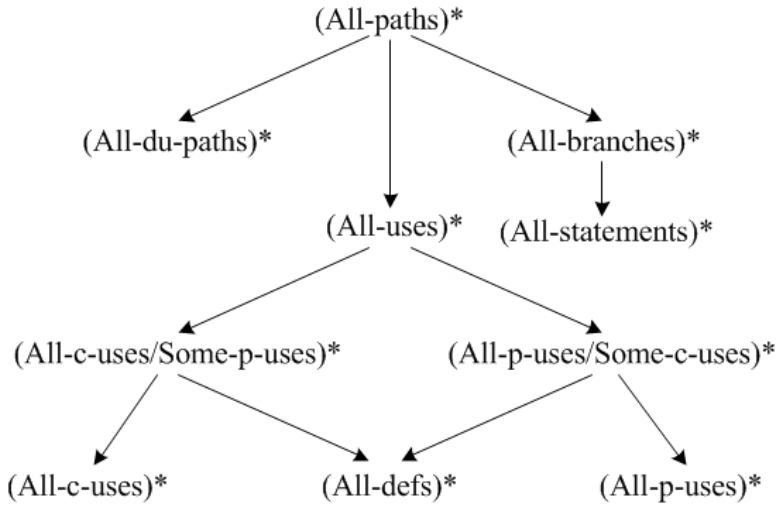
Chúng ta sẽ sinh ra các giá trị đầu vào và kiểm tra các giá trị đó sao cho thỏa mãn hết các điều kiện trên. Chú ý, các giá trị đầu vào ở đây là các tham số của hàm `ReturnAverage` (Đoạn mã 7.2), giá trị của biến `i` ở điều kiện (3) lớn hơn 1 so với giá trị nó tại các biểu thức (1) và (2), và giá trị của biến `ti` ở các điều kiện (2) và (3) lớn hơn 1 so với giá trị nó tại biểu thức (1). Ví dụ, bộ đầu vào ([1, -999], 2, 1, 2) thỏa mãn tất cả các điều kiện trên.

Để lựa chọn các bộ đầu vào ứng với các đường đi đầy đủ thỏa mãn một độ đo cho trước, chúng ta phải đảm bảo rằng các đường đi này là thực thi được. Giả sử P_C là tập các đường đi đầy đủ của đơn vị chương trình ứng với độ đo C. Độ đo này chỉ có ích khi tồn

tại ít nhất một đường đi thuộc P_C sao cho nó thực thi được. Trong trường hợp này, chúng ta gọi C là độ đo dòng dữ liệu thực thi được, ký hiệu là $(C)^*$. Ví dụ, $(All\text{-}c\text{-}uses)^*$ được sử dụng để chỉ tất cả các đường đi đầy đủ thỏa mãn độ đo *All-c-uses* sao cho chúng thực thi được. Cụ thể, $(All\text{-}c\text{-}uses)^*$ được định nghĩa như sau.

Định nghĩa 7.28. ($(All\text{-}c\text{-}uses)^*$.) Với mỗi một biến x và mỗi đỉnh i sao cho i là *Global def* với biến x, chọn các đường đi đầy đủ thực thi được bao gồm các *Def-clear path* từ đỉnh i tới tất cả các đỉnh j sao cho j là *Global c-use* của x.

Tương tự, chúng ta có thể định nghĩa các độ đo dòng dữ liệu thực thi được còn lại: $(All\text{-}paths)^*$, $(All\text{-}du\text{-}paths)^*$, $(All\text{-}uses)^*$, $(All\text{-}p\text{-}uses)^*$, $(All\text{-}p\text{-}uses/Some\text{-}c\text{-}uses)^*$, $(All\text{-}c\text{-}uses/Some\text{-}p\text{-}uses)^*$, $(All\text{-}defs)^*$, $(All\text{-}branches)^*$, và $(All\text{-}statements)^*$. Mối quan hệ bao gồm chặt giữa từng cặp này đã được Frankl và Weyuker [GJ88] đã tổng kết trong hình 7.6.



Hình 7.6: Mối quan hệ bao gồm chặt giữa các độ đo thực thi được.

7.2 Kiểm thử dựa trên lát cắt

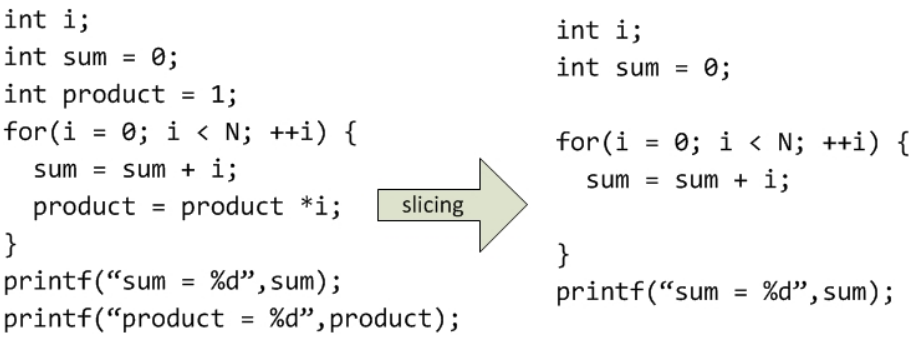
Kiểm thử dòng dữ liệu như đã trình bày ở trên là một phương pháp tốt nhằm phát hiện các lỗi tiềm tàng bên trong các đơn vị chương trình. Tuy nhiên, độ phức tạp của phương pháp này rất lớn. Với các đơn vị chương trình có kích thước lớn, phương pháp này khá tốn kém. Trong thực tế, để áp dụng phương pháp này, chúng ta không cần phân tích tất cả các câu lệnh thuộc đơn vị chương trình cần kiểm thử. Với mỗi biến, chỉ có một tập con các câu lệnh có liên quan (khai báo, gán giá trị và sử dụng) đến biến này. Dựa trên quan sát này, phương pháp kiểm thử chương trình dựa trên lát cắt được đề xuất nhằm giảm thiểu độ phức tạp trong việc sinh các ca kiểm thử của phương pháp kiểm thử dòng dữ liệu.

7.2.1 Ý tưởng về kiểm thử dựa trên lát cắt

Các lát cắt chương trình đã từng có những bước thăng trầm trong ngành công nghệ phần mềm kể từ đầu những năm 1980. Chúng được đề xuất bởi Weiser [Mar81, Mar84], được dùng như là một phương pháp tiếp cận bảo trì phần mềm, và gần đây nhất chúng được sử dụng như là một cách để kết dính các chức năng. Một phần sự linh hoạt này là do tính tự nhiên cũng như mục đích rõ ràng của lát cắt chương trình.

Thông thường, một lát cắt chương trình là một phần chương trình có ảnh hưởng tới giá trị của biến tại một vị trí trong chương trình. Hình 7.7 là một ví dụ về một lát cắt chương trình ứng với biến *sum* (phần bên phải). Lát cắt này có được bằng cách lựa chọn các câu lệnh có ảnh hưởng đến biến *sum* từ đoạn chương trình phía bên trái. Các câu lệnh `int product = 1;`, `product = product*i;`, và `printf("product␣=␣%d",product);` không có ảnh hưởng đến biến *sum* nên đã bị loại bỏ khỏi lát cắt này.

Chúng ta sẽ bắt đầu bằng việc định nghĩa thế nào là một lát cắt chương trình. Giả sử ta có một chương trình ký hiệu là P, đồ



Hình 7.7: Một ví dụ về lát cắt chương trình.

thị của chương trình là $G(P)$, và tập các biến của chương trình là V. Sau đây, chúng ta sẽ tìm hiểu chi tiết về kỹ thuật kiểm thử dựa trên lát cắt.

Định nghĩa 7.29. (Lát cắt.) Cho một chương trình P và V là tập các biến trong chương trình này. Một lát cắt trên V tại câu lệnh n, kí hiệu $S(V, n)$, là tập tất các lệnh trong P có góp phần làm thay đổi giá trị của tập biến trong V.

Tuy nhiên, định nghĩa trên còn khá chung chung nên rất khó để xác định $S(V, n)$. Định nghĩa sau giúp chúng ta giải quyết vấn đề này.

Định nghĩa 7.30. (Lát cắt chương trình.) Cho một chương trình P với đồ thị chương trình $G(P)$ (trong đó các câu lệnh và các đoạn câu lệnh được đánh số) và một tập các biến V trong P, lát cắt trên tập biến V tại đoạn câu lệnh n, ký hiệu là $S(V, n)$, là tập các số nút của tất cả các câu lệnh và đoạn câu lệnh trong P "trước thời điểm" n "ảnh hưởng" đến các giá trị của các biến trong V tại đoạn mã lệnh thứ n [Jor13].

Trong định nghĩa trên, thuật ngữ "các đoạn câu lệnh" có nghĩa là một câu lệnh có thể là một câu lệnh phức do vậy ta có thể tách các câu lệnh này thành từng câu lệnh riêng biệt. Ví dụ, câu lệnh phức

`int intMin=0, intMax=100;` sẽ được tách thành hai câu lệnh đơn `int intMin=0;` và `int intMax=100;`. Khái niệm "trước thời điểm" n "ảnh hưởng" không có nghĩa là thứ tự các câu lệnh mà là thời điểm trước khi câu lệnh đó được thực hiện. Ví dụ, trong hàm tính tổng các số chẵn nhỏ hơn n như Đoạn mã 7.3, câu lệnh `i++;` đứng sau nhưng lại ảnh hưởng trực tiếp đến câu lệnh `result += i;`.

Đoạn mã 7.3: Hàm tính tổng các số chẵn nhỏ hơn n

```
int TongCacSoChan(int n){
        int i = 0;
        int result = 0;
        while (i < n){
                if(i%2 == 0){
                        result += i;
                }
                i++;
        }
        return result;
}
```

Ý tưởng của các lát cắt là để tách một chương trình thành các thành phần, mỗi một thành phần có một số ý nghĩa nhất định. Các phần ảnh hưởng tới giá trị của các biến đã được giới thiệu trong mục 7.1.5 bằng việc sử dụng các định nghĩa và sử dụng của từng biến (Def, C-use, P-use), nhưng chúng ta cần phải tinh chỉnh lại một số hình thức sử dụng biến. Cụ thể là mối quan hệ sử dụng (Use) của biến gắn liền với năm hình thức sử dụng như sau.

- P-use: Biến được sử dụng trong các câu lệnh rẽ nhánh. Ví dụ, `if(x>0){...}`

- C-use: Biến được sử dụng trong các câu lệnh tính toán. Ví dụ, `x = x + y;`

- *O-use*: Biến được sử dụng cho các câu lệnh hiển thị hoặc trả về kết quả. Ví dụ, `return x;` hoặc `printf("%d",x);`
- *L-use*: Biến được sử dụng như một con trỏ trỏ đến các địa chỉ hoặc chỉ số của mảng. Ví dụ, `int x =100, *ptr; ptr = &x;`
- *I-use*: Biến được sử dụng như các biến đếm (trong các vòng lặp). Ví dụ, `i++;`

Chúng ta cũng có hai dạng xác định giá trị cho các biến như sau:

- *I-def*: xác định từ đầu vào (từ bàn phím, truyền tham số, v.v.)
- *A-def*: xác định từ phép gán

Giả sử lát cắt $S(V, n)$ là một lát cắt trên một biến, ở đây tập V chỉ chứa một biến v duy nhất. Nếu nút n chứa một định nghĩa của v thì ta thêm n vào lát cắt $S(V, n)$. Ngược lại, nếu nút n chứa một sử dụng của biến $v \in V$ thì n không được thêm vào lát cắt $S(V, n)$. Những nút chứa *P-use* và *C-use* của các biến khác (không phải biến v trong tập V) mà ảnh hưởng trực tiếp hoặc gián tiếp tới giá trị của biến v thì được thêm vào tập V. Đối với lát cắt $S(V, n)$, những định nghĩa và sử dụng của các biến sau được thêm vào lát cắt $S(V, n)$.

- Tất cả các *I-def* và *A-def* của biến v
- Tất cả các *C-use* và *P-use* của biến v sao cho nếu loại bỏ nó sẽ làm thay đổi giá trị của v
- Tất cả các *P-use* và *C-use* của các biến khác (không phải biến v) sao cho nếu loại bỏ nó thì sẽ làm thay đổi giá trị của biến v

- Loại bỏ khỏi lát cắt các *I-use*, *L-use* và *O-use* của biến v
- Loại bỏ toàn bộ các câu lệnh không được thực thi như các câu lệnh khai báo biến
- Kiểm tra các hằng số, nếu hằng số đó ảnh hưởng đến biến v thì ta thêm hằng số đó vào lát cắt

7.2.2 Ví dụ áp dụng

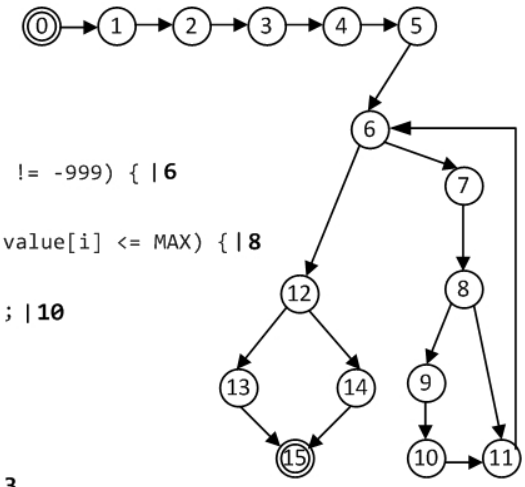
Quay trở lại với ví dụ về hàm `ReturnAverage` được trình bày ở Đoạn mã 7.2 trong mục 7.1.4, để áp dụng kỹ thuật kiểm thử dựa trên lát cắt, chúng ta phân mảnh hàm này như Đoạn mã 7.4. Tiếp đến, chúng ta xây dựng đồ thị của hàm sau khi phân mảnh như hình 7.8. Sau đó, chúng ta cũng sẽ định nghĩa lại các định nghĩa (Def) và sử dụng (Use) của các biến trong các bảng 7.3 và 7.4. Và cuối cùng, các lát cắt trên từng biến của hàm sẽ được tính toán.

Đoạn mã 7.4: Mã nguồn hàm `ReturnAverage` sau khi phân mảnh

```
double ReturnAverage(int value[], int AS, int MIN,
                     int MAX){
    int i = 0;
    int ti = 0;
    int tv = 0;
    int sum = 0;
    double av;
    while (ti < AS && value[i] != -999) {
       ti++;
       if (value[i] >= MIN && value[i] <= MAX) {
          tv++;
          sum = sum + value[i];
       }
       i++;
    }//end while
    if (tv > 0)
```

```
        av = (double)sum/tv;
    else
        av = (double) -999;
    return (av);
} //the end
```

```
double ReturnAverage(int value[],  int AS, int MIN, int MAX){|0
    int i = 0;  |1
    int ti = 0; |2
    int tv = 0; |3
    int sum = 0;|4
    double av;  |5
    while (ti < AS && value[i] != -999) { |6
       ti++; |7
       if (value[i] >= MIN && value[i] <= MAX) {|8
          tv++; |9
          sum = sum + value[i]; |10
       }
       i++; |11
    }//end while
    if (tv > 0) |12
       av = (double)sum/tv; |13
    else
       av = (double) -999; |14
    return (av); |15
} //the end
```



Hình 7.8: Hàm `ReturnAverage` sau khi phân mảnh và đồ thị của nó.

Chú ý, ta nhận thấy các biến i, ti và tv là các biến được sử dụng trong vòng lặp (trong hình 7.8). Tuy nhiên, chúng là các biến cục bộ nên không thể xếp các sử dụng của các biến này là *I-use*. Tiếp theo, chúng ta sẽ xét các lát cắt của hàm `ReturnAverage` trên từng biến trong tập $V = \{value, AS, MIN, MAX, i, ti, tv, sum, av\}$. Ta nhận thấy rằng, $value, AS, MIN$ và MAX là các biến chỉ có *I-def* và giá trị của chúng không bị ảnh hưởng khi thực hiện chương trình. Cụ thể, biến $value$ có một *I-def* tại đỉnh 0, có hai *P-use* tại các

Bảng 7.3: *I-def*, *A-def* **và** `Const` **của các đỉnh trong Hình 7.8**

Đỉnh i	*I-def*(i)	*A-def*(i)	**Const(i)**
0	{value, AS, MIN, MAX}	{}	{}
1	{}	{i}	{}
2	{}	{ti}	{}
3	{}	{tv}	{}
4	{}	{sum}	{}
5	{}	{}	{}
6	{}	{}	{}
7	{}	{ti}	{}
8	{}	{}	{}
9	{}	{tv}	{}
10	{}	{sum}	{}
11	{}	{i}	{}
12	{}	{}	{}
13	{}	{av}	{}
14	{}	{av}	{}
15	{}	{}	{}

đỉnh 6 và 8 và có một *C-use* tại đỉnh 10. Vì vậy, các lát cắt của các biến này được tính toán như sau:

- S_1 : $S(value, 0) = \{0\}$ (do chỉ có duy nhất một *I-def* của biến *value* tại đỉnh 0 nên 0 được thêm vào S_1)

- S_2 : $S(value, 6) = \{0\}$ (từ đỉnh 0 đến đỉnh 6 chỉ có một *I-def* của biến *value* tại đỉnh 0 ảnh hưởng đến giá trị của biến *value* tại đỉnh 6 nên 0 được thêm vào S_2)

- S_3 : $S(value, 8) = \{0\}$ (từ đỉnh 0 đến đỉnh 8 chỉ có một *I-def* của biến *value* tại đỉnh 0 nên 0 được thêm vào S_3. Với *P-use* của biến *value* tại đỉnh 6 không ảnh hương đến giá trị của biến này tại đỉnh 8 nên 6 không được thêm vào S_3)

Bảng 7.4: *C-def*, *L-def*, *I-def*, *O-def* **và** *P-def* **của các đỉnh**

Đỉnh	*C-def*(i)	*L-def*(i)	*I-def*(i)	*O-def*(i)	*P-def*(i)
0	{}	{}	{}	{}	{}
1	{}	{}	{}	{}	{}
2	{}	{}	{}	{}	{}
3	{}	{}	{}	{}	{}
4	{}	{}	{}	{}	{}
5	{}	{}	{}	{}	{}
6	{}	{}	{}	{}	{ti,AS,value}
7	{ti}	{}	{}	{}	{}
8	{}	{}	{}	{}	{value,MIN,MAX}
9	{tv}	{}	{}	{}	{}
10	{sum,value}	{}	{}	{}	{}
11	{i}	{}	{}	{}	{}
12	{}	{}	{}	{}	{tv}
13	{sum,tv}	{}	{}	{}	{}
14	{}	{}	{}	{}	{}
15	{}	{}	{}	{av}	{}

- S_4 : $S(value, 10) = \{0\}$ (từ đỉnh 0 đến đỉnh 10, biến *value* có một *I-def* tại đỉnh 0 và hai *P-use* tại các đỉnh 6 và 8. Do hai *P-use* này không làm ảnh hưởng đến giá trị của biến nên chúng không được thêm vào S_4.)

Tương tự, biến AS có một *I-def* tại đỉnh 0 và có một *P-use* tại đỉnh 6. Vì vậy, các lát cắt của các biến này như sau:

- S_5 : $S(AS, 0) = \{0\}$ (do chỉ có duy nhất một *I-def* của biến AS tại đỉnh 0 nên 0 được thêm vào S_5)
- S_6 : $S(AS, 6) = \{0\}$ (từ đỉnh 0 đến đỉnh 6 chỉ có duy nhất một *I-def* của biến AS tại đỉnh 0 nên ta thêm 0 vào S_6)

Có một *I-def* của biến MIN tại đỉnh 0 và có một *P-use* tại đỉnh 8. Vì vậy, các lát cắt của các biến này như sau:

- $S_7 : S(MIN, 0) = \{0\}$
- $S_8 : S(MIN, 8) = \{0\}$

Tương tự như với biến MIN, biến MAX cũng có một *I-def* tại đỉnh 0 và có một *P-use* tại đỉnh 8. Các lát cắt của biến này như sau:

- $S_9 : S(MAX, 0) = \{0\}$
- $S_{10} : S(MAX, 8) = \{0\}$

Chú ý, với các lát cắt S_2, S_3, S_4, S_6, S_8 và S_{10}, mỗi lát cắt tồn tại một *Def-clear-path*. Thường đối với các lát cắt tồn tại một *Def-clear-path* thì các lỗi tiềm ẩn ít xuất hiện.

Đối với biến i, có hai *A-def* của biến này tại các đỉnh 1 và 11, ba *L-use* tại các đỉnh 6, 8, 10 và một *C-use* tại đỉnh 11. Vì vậy, các lát cắt của các biến này như sau:

- $S_{11} : S(i, 1) = \{1\}$ (biến i có duy nhất một *A-def* tại đỉnh 1 nên 1 được thêm vào S_{11})
- $S_{12} : S(i, 6) = \{1, 6, 7, 11\}$ (biến i có hai *A-def* tại các đỉnh 1 và 11 nên chúng được thêm vào S_{12}. Xét các *P-use* và *C-use* của các biến khác ảnh hưởng đến giá trị của biến i tại đỉnh 6, theo bảng 7.4 ta có các *P-use* của các biến $ti, value, AS$ tại đỉnh này có ảnh hưởng đến giá trị của biến i nên 6 được thêm vào S_{12}. Ngoài ra, có một *C-use* của biến ti tại đỉnh 7 cũng ảnh hưởng đến giá trị của i tại đỉnh 6 nên 7 cũng được thêm vào lát cắt này.)
- $S_{13} : S(i, 8) = \{1, 6, 7, 11\}$ (tương tự như S_{12})
- $S_{14} : S(i, 10) = \{1, 6, 7, 11\}$ (tương tự như S_{12})
- $S_{15} : S(i, 11) = \{1, 6, 7, 11\}$ (tương tự như S_{12})

Đối với biến ti, có hai $A\text{-}def$ tại các đỉnh 2 và 7, một $P\text{-}use$ tại đỉnh 6 và một $C\text{-}use$ tại đỉnh 7. Các lát cắt của các biến này như sau:

- $S_{16} : S(ti, 2) = \{2\}$
- $S_{17} : S(ti, 6) = \{2, 6, 7, 11\}$ (có hai $A\text{-}def$ tại các đỉnh 2 và 7 ảnh hưởng đến giá trị của biến này tại đỉnh 6 nên chúng được thêm vào lát cắt này. Thêm nữa, các $P\text{-}use$ của các biến $value, AS$ tại đỉnh 6 và $C\text{-}use$ của biến i tại đỉnh 11 cũng ảnh hưởng đến giá trị của biến ti tại đỉnh 6 nên các đỉnh 6 và 11 cũng được thêm vào lát cắt.)
- $S_{18} : S(ti, 7) = \{2, 6, 7, 11\}$ (tương tự như S_{17})

Tương tự với biến tv, có hai $A\text{-}def$ tại các đỉnh 3 và 9, hai $C\text{-}use$ tại các đỉnh 9, 13 và có một $P\text{-}use$ tại đỉnh 12. Vì vậy, các lát cắt của các biến này như sau:

- $S_{19} : S(tv, 3) = \{3\}$
- $S_{20} : S(tv, 9) = \{3, 6, 7, 8, 9, 11\}$

  Có hai $A\text{-}def$ tại các đỉnh 3 và 9 nên chúng được thêm vào lát cắt. Xét các $P\text{-}use$ và $C\text{-}use$ của các biến khác có ảnh hưởng đến giá trị của biến tv tại đỉnh 9. Tại đỉnh 6, chúng ta có các $P\text{-}use$ của các biến $value, ti, AS$ cũng ảnh hưởng đến giá trị của biến tv tại đỉnh 9 nên đỉnh 6 được thêm vào lát cắt. Tương tự, tại đỉnh 7, ta cũng có một $C\text{-}use$ của tv ảnh hưởng đến giá trị của chính nó nên đỉnh 7 cũng được thêm vào S_{20}. Tiếp theo, tại đỉnh 8, các $P\text{-}use$ của các biến $value, MIN, MAX$ cũng ảnh hưởng đến giá trị biến này nên đỉnh 8 cũng được thêm vào lát cắt. Cuối cùng, tại đỉnh 11, chúng ta có một $C\text{-}use$ của biến i ảnh hưởng đến giá trị của biến tv tại đỉnh 9 nên đỉnh 11 được thêm vào S_{20}.

- $S_{21} : S(tv, 12) = \{3, 6, 7, 8, 9, 11\}$ (tương tự như S_{20})
- $S_{22} : S(tv, 13) = \{3, 6, 7, 8, 9, 11\}$ (tương tự như S_{20})

Biến *sum* có hai *A-def* tại các đỉnh 4 và 10 và có hai *C-use* tại các đỉnh 10 và 13. Do đó, các lát cắt của biến này như sau:

- $S_{23} : S(sum, 4) = \{4\}$
- $S_{24} : S(sum, 10) = \{4, 6, 8, 9, 10, 11\}$
- $S_{25} : S(sum, 13) = \{4, 6, 7, 8, 9, 10, 11\}$

Tương tự, biến *av* được khai báo tại đỉnh 5, có hai *A-def* tại các đỉnh 13, 14 và một *O-use* tại đỉnh 15. Các lát cắt của biến này như sau:

- $S_{26} : S(av, 5) = \{\}$
- $S_{27} : S(av, 13) = \{6, 8, 9, 10, 11, 12\}$
- $S_{28} : S(av, 14) = \{14\}$
- $S_{29} : S(av, 15) = \{6, 8, 9, 10, 11, 12, 13, 14\}$

Ta nhận thấy rằng lát cắt S_{29} của biến *av* bị ảnh hưởng bởi hai *A-def* của chính biến này tại các đỉnh 14 và 15 tương đương với các lát cắt S_{28} và S_{27}. Do vậy, ta có thể viết $S_{29} = S_{28} \cup S_{27}$. Ta cũng làm tương tự đối với các biến còn lại để có một cách viết khác về các lát cắt. Hơn nữa, ta có thể rút ra các nhận xét quan trọng sau liên quan đến các lát cắt.

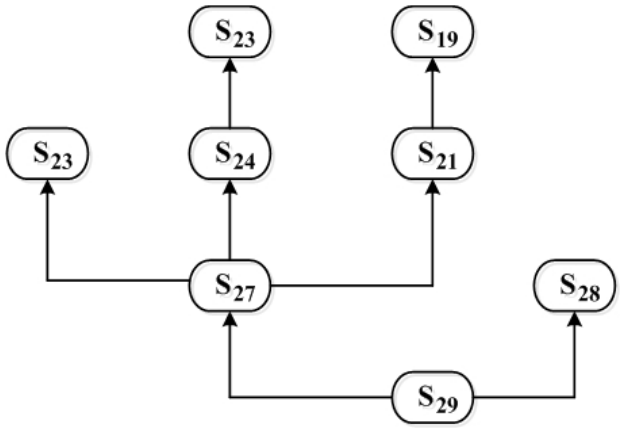
- Các lỗi tiềm ẩn có thể xuất hiện ở các lát cắt chứa nhiều hơn một *A-def*.
- Các lát cắt có thể là hợp của các lát cắt khác.

- Các lát cắt của cùng một biến tại các đỉnh khác nhau có thể trùng nhau.

Để tránh sự trùng lặp cũng như loại bỏ các lát cắt không cần thiết, chúng ta cần xây dựng mối quan hệ giữa các lát cắt cho toàn bộ chương trình. Các mối quan hệ này sẽ tạo nên một mạng được gọi là là mạng tinh thể của các lát cắt với định nghĩa như sau.

Định nghĩa 7.31. Mạng tinh thể của các lát cắt ứng với một chương trình/đơn vị chương trình là một đồ thị có hướng không có chu trình (directed acyclic graph), trong đó mỗi đỉnh chứa một lát cắt sao cho các đỉnh đó tương ứng với các vị trí trong mã nguồn. Mối quan hệ giữa các lát cắt được xác định bởi các *Def-clear-path* của các biến.

Ví dụ, ta nhận thấy rằng có hai *Def-clear-path* <13,15> và <14,15> của biến *av* tương ứng với các cạnh. Các cạnh này chỉ ra lát cắt S_{27} và S_{28} là tập con của lát cắt S_{29}. Hình 7.9 cho thấy một phần mạng tinh thể của các lát cắt trong hàm `ReturnAverage` được mô tả ở hình 7.8.



Hình 7.9: Mạng tinh thể của hàm `ReturnAverage` trong hình 7.8.

Sau khi có được mạng tinh thể, công việc cuối cùng của chúng ta là tạo các chương trình có thể biên dịch được tương ứng với các lát cắt cần thiết và kiểm thử chúng.

7.2.3 Một số lưu ý với kiểm thử dựa trên lát cắt

Để áp dụng một cách hiệu quả phương pháp kiểm thử dựa trên lát cắt, chúng ta cần chú ý đến các vấn đề sau:

- Không bao giờ tạo một lát cắt $S(V, n)$ đối với một biến $v \in V$ nơi mà biến này không xuất hiện trong câu lệnh/đoạn câu lệnh thứ n.
- Luôn tạo một lát cắt trên từng biến một. Tập V trong lát cắt $S(V, n)$ có thể chứa nhiều hơn một biến. Ví dụ, $S(V, 10)$ với $V = \{sume, value, i\}$. Do vậy, chúng ta phải xây dựng các lát cắt trên từng biến ứng với các lát cắt: $S(sum, 10), S(value, 10)$ và $S(i, 10)$.
- Tạo các lát cắt cho toàn bộ các đỉnh chứa $A\text{-}def$. Khi một biến được tính toán bằng các câu lệnh gán, một lát cắt trên biến đó sẽ bao gồm tất cả các *Du-paths* của các biến đã sử dụng trong việc tính toán. Lát cắt $S(10, sum)$ trong ví dụ ở mục 7.2.2 là một ví dụ rõ ràng nhất cho trường hợp này.
- Tạo các lát cắt cho các đỉnh chứa *P-use*. Khi một biến được sử dụng trong các câu lệnh điều khiển, lát cắt của nó cho biết làm thế nào một biến điều khiển nhận giá trị. Điều này thật sự hữu ích cho việc kiểm tra các điểm quyết định trong chương trình.
- Các lát cắt trên các đỉnh chứa các điều kiện không sử dụng đến các biến (ví dụ: `while(1){...}`) thì không cần quan tâm.
- Chúng ta có thể tạo ra một chương trình có thể biên dịch được từ các lát cắt vì chưa có một định nghĩa nào của lát cắt

yêu cầu tập các câu lệnh phải biên dịch được. Cụ thể, chúng ta sẽ tạo một chương trình con từ các câu lệnh của lát cắt đó. Ví dụ, chương trình con ứng với lát cắt $S_{12} = \{1, 6, 7, 11\}$ chứa các câu lệnh như Đoạn mã 7.5. Tuy nhiên, chương trình con này không thể biên dịch được do lỗi cú pháp (biến *ti* chưa được khai báo và chưa trả về kết quả kiểu *double*). Vì vậy, chúng ta sẽ sửa chương trình con này để có được một chương trình con có thể biên dịch được như Đoạn mã 7.6. Khi đó, chúng ta có thể kiểm thử chương trình con ứng với lát cắt này. Tương tự, chúng ta cũng kiểm thử các lát cắt khác, hoặc kết hợp chúng thành một khối chương trình hoàn chỉnh.

Đoạn mã 7.5: Chương trình con ứng với lát cắt S_{12}

```
double ReturnAverage(int value[], int AS,
                     int MIN, int MAX) {
    int i = 0;
    while (ti < AS && value[i] != -999) {
         ti++;
         i++;
    }//end while
} //the end
```

Đoạn mã 7.6: Chương trình con biên dịch được ứng với S_{12}

```
double ReturnAverage(int value[], int AS,
                     int MIN, int MAX){
    int i = 0;
    int ti = 0;
    while (ti < AS && value[i] != -999) {
         ti++;
         i++;
    }//end while
    double av;
    return av;
} //the end
```

7.3 Tổng kết

Phương pháp kiểm thử dòng dữ liệu cho phép chúng ta phát hiện các lỗi tiềm ẩn bên trong chương trình liên quan đến việc sử dụng các biến. Các lỗi này thường khó phát hiện bởi các phương pháp khác. Để áp dụng phương pháp này, chúng ta cần xây dựng đồ thị dòng dữ liệu cho chương trình cần kiểm thử. Ứng với mỗi độ đo kiểm thử, chúng ta sẽ xây dựng tập các đường đi đầy đủ (*Complete-paths*) để sinh các ca kiểm thử tương ứng. Chúng ta nên sử dụng kiểm thử dòng dữ liệu khi chương trình có nhiều tính toán. Khi chương trình có nhiều lệnh rẽ nhánh và biến của biểu thức điều kiện này cũng được tính toán (*p-use*) thì cũng nên sử dụng kiểm thử dòng dữ liệu. Có nhiều công cụ của các nhóm nghiên cứu hỗ trợ xác định các khái niệm kiểm thử dòng dữ liệu này và một số công cụ đã được thương mại hóa. Khi bạn gặp một số mô-đun khó kiểm thử, nên sử dụng một số gợi ý sau:

- Nên kết hợp phương pháp kiểm thử dòng dữ liệu với các phương pháp kiểm thử hộp đen (kiểm thử chức năng) trong một quy trình thống nhất. Sau khi tiến hành các phương pháp kiểm thử hộp đen, chúng ta tiến hành phân tích các ca kiểm thử đã được sử dụng để xác định các đường đi đầy đủ nào ứng với biến nào đã được kiểm thử, các đường đi đầy đủ chưa được kiểm thử ứng với mỗi biến. Với cách làm này, chúng ta chỉ cần sinh các ca kiểm thử cho các đường đi đầy đủ chưa được kiểm thử bởi các ca kiểm thử hộp đen mà vẫn đảm bảo 100% độ bao phủ cho kiểm thử dòng dữ liệu.

- Các lát cắt không tương ứng với các ca kiểm thử, vì nhiều lệnh không nằm trong lát cắt nhưng nằm trên đường đi của chương trình. Khi đó có thể cách kết hợp các lát cắt, tức là dựa trên lát cắt để cấu trúc lại mô-đun để thuận tiện cho việc kiểm thử.

- Phần bù giữa hai lát cắt giúp việc chuẩn đoán chương trình. Phần bù của một tập B so với tập A là tập các phần tử thuộc A mà không nằm trong B, ký hiệu là $A - B$.

Tuy nhiên, so với phương pháp kiểm thử dòng điều khiển, phương pháp kiểm thử dòng dữ liệu khó áp dụng hơn và có độ khó hơn. Hơn nữa, phương pháp này thường khá phức tạp khi áp dụng với các đơn vị chương trình có kích thước lớn. Phương pháp kiểm thử dựa trên lát cắt đang được xem là một giải pháp hứa hẹn nhằm giải quyết vấn đề này. Trong phương pháp kiểm thử dựa trên lát cắt, chúng ta không cần phân tích tất cả các câu lệnh thuộc đơn vị chương trình. Với mỗi biến, chúng ta chỉ quan tâm đến một tập con các câu lệnh có liên quan (khai báo, gán giá trị và sử dụng) đến biến này.

7.4 Bài tập

1. So sánh phương pháp kiểm thử dòng điều khiển và phương pháp kiểm thử dòng dữ liệu. Liệu chúng có thay thế lẫn nhau được không? Tại sao?

2. Thế nào là kiểm thử dòng dữ liệu tĩnh? Thế nào là kiểm thử dòng dữ liệu động? Trình bày mối quan hệ giữa chúng.

3. Mô tả ba loại vấn đề phổ biến về dòng dữ liệu. Với mỗi loại vấn đề, hãy lấy ví dụ minh họa.

4. Trình bày phương pháp xác định các vấn đề về dòng dữ liệu sử dụng sơ đồ chuyển trạng thái. Hãy áp dụng phương pháp này cho một chương trình/đơn vị chương trình cụ thể.

5. Tại sao kiểm thử dòng dữ liệu tĩnh vẫn không đảm bảo được rằng chương trình không còn lỗi liên quan đến dòng dữ liệu của chương trình?

6. Trình bày các bước trong quy trình kiểm thử dòng dữ liệu động.

7. Cho hàm `calFactorial` viết bằng ngôn ngữ C như Đoạn mã 7.7.

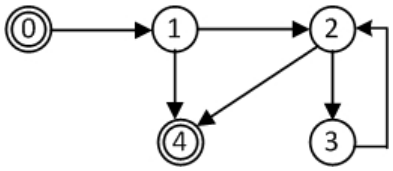
   - Hãy liệt kê các câu lệnh ứng với các khái niệm def, $c-use$, và $p-use$ ứng với các biến được sử dụng trong hàm này.
   - Hãy vẽ đồ thị dòng dữ liệu của hàm này.

Đoạn mã 7.7: Mã nguồn C của hàm `calFactorial`

```
int calFactorial (int n){
        int result = 1;
        int i=1;
        while (i <= n){
                result = result *i;
                i++;
        }//end while
        return result;
}//the end
```

8. Hãy chứng minh rằng chúng ta không thể tìm thấy mối quan hệ *bao gồm chặt* giữa hai độ đo *All-c-uses* và *All-p-uses* (hay nói cách khác, hai độ đo này không so sánh được).

9. Cho đồ thị dòng dữ liệu như hình 7.10.

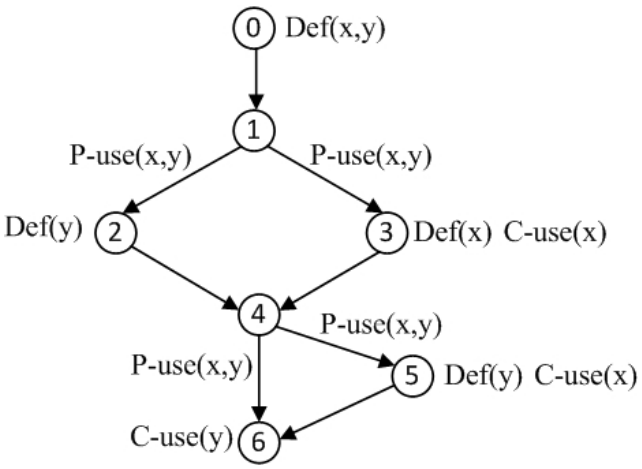


Hình 7.10: Một ví dụ về đồ thị dòng dữ liệu.

   - Hãy xác định tất cả các *Complete-path* từ đồ thị này.

- Các đường đi (0 - 1 - 2 - 3), (1 - 2 - 3 - 2 - 4), (3 - 2 - 4), (2 - 3 - 2) và (0 - 1 - 4) có phải là các simple-paths không? Giải thích.
- Các đường đi (2 - 3 - 2), (2 - 3) và (3 - 2 - 4) có phải là các *loop-free-paths* không? Giải thích.

10. Cho đồ thị dòng dữ liệu như hình 7.11.



Hình 7.11: Một ví dụ về đồ thị dòng dữ liệu và việc sử dụng các biến.

- Hãy xác định tất cả các *Def-clear-path* của các biến x và y.
- Hãy xác định tất cả các du-paths của các biến x và y.
- Hãy xác định tất cả các *All-p-uses/Some-c-uses* và *All-c-uses/Some-p-uses* (dựa vào các chuẩn của kiểm thử dòng dữ liệu).
- Biểu thức của các *p-use*(x, y) tại cạnh (1,3) và (4,5) lần lượt là $x + y = 4$ và $x^2 + y^2 > 17$. Đường đi (0 - 1 - 3 - 4 - 5 - 6) có thực thi được không? Giải thích.
- Tại sao tại đỉnh 3 biến x được định nghĩa và sử dụng nhưng không tồn tại mối quan hệ def-use?

11. Tại sao khi xây dựng xong các lát cắt người ta lại phải xây dựng mạng tinh thể cho các lát cắt này. Mối quan hệ các lát cắt được thể hiện thông qua mối quan hệ nào?

12. Cho các lệnh như Đoạn mã 7.8.

Đoạn mã 7.8: Đoạn lệnh bằng ngôn ngữ C++

```
1 #define MAX=50
2 int intValue=0;
3 int iCount=0;
4 for(int i=0;i<MAX;i++){
5         if(i%2==0){
6                 intValue = intValue+i;
          }
7         else {
8                 intValue = intValue*i;
9         }//end if-else
10 }//end for
```

- Biến i trong vòng lặp for có ảnh hưởng đến giá trị của biến $intValue$ không? Tại sao?
- Hãy xác định các câu lệnh ảnh hưởng đến giá trị của biến $intValue$ tại các câu lệnh 6 và 8.

13. Cho mã nguồn của hàm `returnSumArray` như Đoạn mã 7.9.

Đoạn mã 7.9: Mã nguồn của hàm `returnSumArray`

```
int returnSumArray(int intArr[], int size){
  int intCount, intSum;
  intCount = 10, intSum = 0;
  while(intCount < size){
        intSum = intSum + intArr[intCount];
        intCount++;
  }//end while
```

```
    return intSum;
}
```

- Hãy xây dựng hàm phân mảnh của hàm này.
- Xây dựng đồ thị dòng dữ liệu và xác định các định nghĩa (def) và sử dụng (use) của tất cả các biến trong chương trình trên.
- Áp dụng các quy tắc về xây dựng lát cắt để tạo ra tất cả các lát cắt của tất cả các biến trong chương trình trên.

Chương 8

Kiểm thử dựa trên mô hình

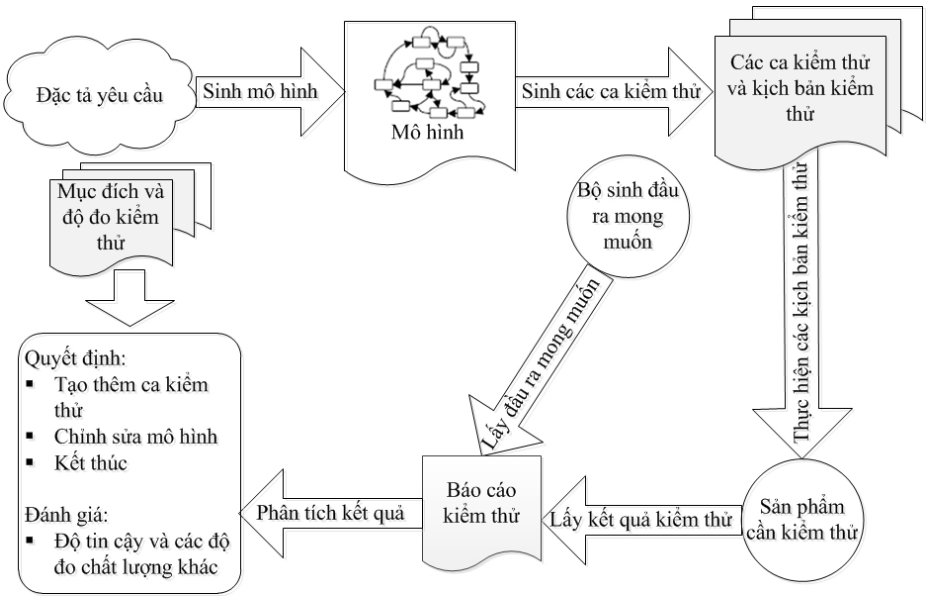
Chương này giới thiệu phương pháp và một số công cụ hỗ trợ kiểm thử dựa trên mô hình nhằm tăng tính hiệu quả và độ chính xác của các hoạt động kiểm thử. Kiểm thử dựa trên mô hình có thể sử dụng với nhiều mục đích khác nhau trong việc tự động hóa các hoạt động kiểm thử. Chúng tôi giới thiệu phương pháp này nhằm kiểm tra tính đúng đắn của việc lập trình so với thiết kế. Các phương pháp và công cụ hỗ trợ kiểm thử dựa trên mô hình có vai trò hết sức quan trọng trong việc nâng cao chất lượng của các sản phẩm và tăng tính cạnh tranh cho các công ty phần mềm.

8.1 Khái niệm về kiểm thử dựa trên mô hình

Có nhiều khái niệm khác nhau về kiểm thử dựa trên mô hình. Tựu trung lại, chúng ta có thể hiểu kiểm thử dựa trên mô hình là một phương pháp kiểm thử nơi mà các ca kiểm thử được sinh ra từ mô hình đặc tả hành vi của hệ thống đang được kiểm thử. Mô hình này được biểu diễn bằng máy hữu hạn trạng thái, ôtômat, đặc tả đại số, biểu đồ trạng thái bằng UML, v.v.

Quá trình kiểm thử dựa trên mô hình được bắt đầu bằng việc xác định yêu cầu của hệ thống từ đó xây dựng mô hình dựa vào các yêu cầu và chức năng của hệ thống. Việc xây dựng mô hình còn phải dựa trên các yếu tố dữ liệu đầu vào và đầu ra. Mô hình này được sử dụng để sinh đầu vào cho các ca kiểm thử. Tiếp đến, chúng ta sẽ sinh giá trị đầu ra mong muốn ứng với mỗi bộ đầu vào. Khi kết thúc bước này, chúng ta đã có các ca kiểm thử. Các kịch bản kiểm thử sẽ được thiết kế và thực thi nhằm phát hiện các lỗi/khiếm khuyết của sản phẩm bằng cách so sánh đầu ra thực tế với đầu ra mong đợi tương ứng của ca kiểm thử. Từ các kết quả kiểm thử, chúng ta sẽ quyết định hành động tiếp theo như sửa đổi mô hình hoặc dừng kiểm thử.

Hình 8.1 mô tả các bước của quy trình kiểm thử dựa trên mô hình, bao gồm:



Hình 8.1: Quy trình kiểm thử dựa trên mô hình [KJ02].

- Sinh mô hình dựa trên các yêu cầu và chức năng của hệ thống.

- Sinh các ca kiểm thử (bộ đầu vào và giá trị đầu ra mong đợi cho mỗi ca kiểm thử).
- Chạy các kịch bản kiểm thử để phát hiện các lỗi/khiếm khuyết của sản phẩm.
- So sánh kết quả đầu ra thực tế với kết quả đầu ra dự kiến.
- Quyết định hành động tiếp theo (sửa đổi mô hình, tạo thêm ca kiểm thử, dừng kiểm thử, đánh giá chất lượng của phần mềm).

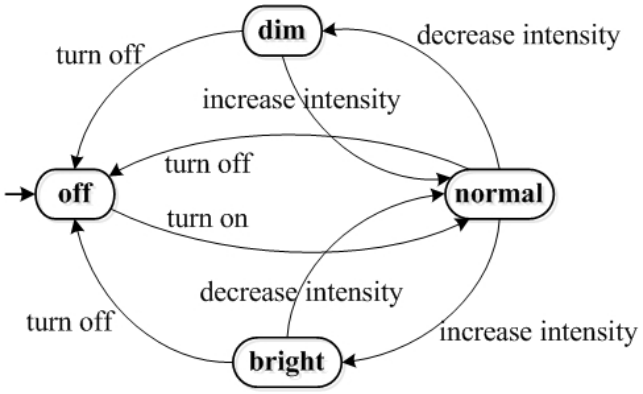
8.2 Các phương pháp đặc tả mô hình

Để áp dụng phương pháp kiểm thử dựa trên mô hình, chúng ta cần xây dựng mô hình đặc tả chính xác hành vi của hệ thống cần kiểm thử. Mô hình này được đặc tả bằng một trong các phương pháp hình thức như: máy hữu hạn trạng thái, biểu đồ trạng thái, máy trạng thái UML, chuỗi Markov, văn phạm, bảng quyết định, v.v. Trong mục này, chúng ta sẽ tìm hiểu một số phương pháp hình thức phổ biến được sử dụng để đặc tả mô hình của các hệ thống.

8.2.1 Máy hữu hạn trạng thái

Máy hữu hạn trạng thái (Finite State Machine - FSM) được biết đến như là phương pháp đặc tả phổ biến nhất cho thiết kế và kiểm thử phần mềm nói riêng và các hệ thống nói chung. FSM rất hiệu quả trong việc đặc tả hành vi dựa trên việc chuyển trạng thái của các hệ thống. Một cách hình thức, FSM được định nghĩa như sau.

Định nghĩa 8.1 (Máy hữu hạn trạng thái)**.** Máy hữu hạn trạng thái là một bộ bốn (S, Act, T, q_0), trong đó S là tập hữu hạn các trạng thái, T là tập các chuyển trạng thái, Act là các tập các sự kiện (còn có tên khác là bảng ký hiệu) và q_0 là trạng thái khởi tạo.



Hình 8.2: Một ví dụ về máy hữu hạn trạng thái.

Bảng 8.1: Bảng chuyển của máy hữu hạn trạng thái trong hình 8.2

	off	*dim*	*normal*	*bright*
off			turn on	
dim	turn off		incr. intensity	
normal	turn off	decr. intensity		incr. intensity
bright	turn off		decr. intensity	

Hình 8.2 mô tả một ví dụ về một máy hữu hạn trạng thái đặc tả hành vi của một hệ thống chuyển công tắc đèn [KJ02]. Trong hình này, *off* là trạng thái khởi đầu của hệ thống. Ở trạng thái này, đèn đang bị tắt. Với đầu vào là *turn on*, hệ thống sẽ chuyển đến trạng thái *normal* với đèn có độ sáng bình thường. Tại trạng thái này, chúng ta có thể tắt đèn (ứng với đầu vào *turn off* và hệ thống sẽ chuyển về trạng thái *off*), tăng độ sáng của đèn (ứng với đầu vào *increase intensity* và hệ thống sẽ chuyển về trạng thái *bright*) và giảm độ sáng của đèn (ứng với đầu vào *decrease intensity* và hệ thống sẽ chuyển về trạng thái *dim*). Tại các trạng thái *dim* và *bright*, chúng ta có thể tắt đèn, tăng và giảm độ sáng tương ứng. Bảng 8.1 là một dạng đặc tả khác của máy hữu hạn trạng thái trên dưới dạng bảng chuyển. Chúng ta sẽ dùng cấu trúc dữ liệu này làm đầu vào cho các công cụ kiểm thử tự động.

8.2.2 Ôtômat đơn định hữu hạn trạng thái

Tương tự như FSM, ôtômat đơn định hữu hạn trạng thái (Deterministic Finite state Automaton - DFA) cũng rất hiệu quả trong việc đặc tả hành vi dựa trên việc chuyển trạng thái của các hệ thống. Một cách hình thức, DFA được định nghĩa như sau.

Định nghĩa 8.2 (Ôtômat đơn định hữu hạn trạng thái)**.** Ôtômat đơn định hữu hạn trạng thái là một bộ năm (S, Act, T, q_0, F), trong đó: S, T, Act và q_0 được định nghĩa như trong định nghĩa của FSM, và $F \subseteq S$ là tập các trạng thái kết thúc.

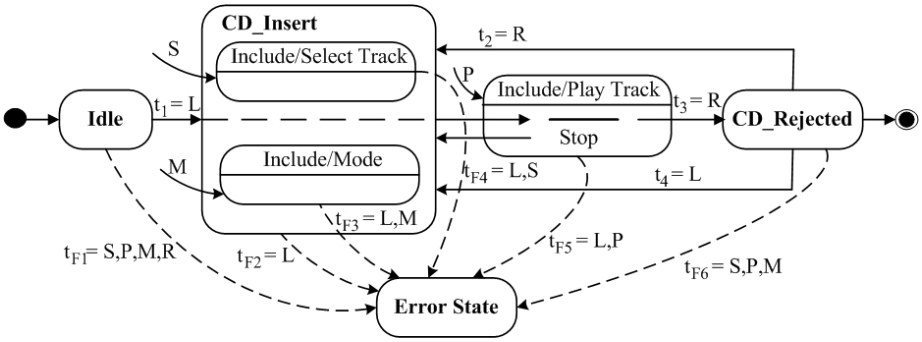
Một ví dụ minh họa về việc áp dụng DFA cho đặc tả hành vi của hệ thống trong kiểm thử dựa trên mô hình sẽ được giới thiệu trong mục 8.6.

8.2.3 Biểu đồ trạng thái

Hình 8.3 mô tả ví dụ về một biểu đồ trạng thái đặc tả hành vi của một máy nghe nhạc. Trong biểu đồ này, trạng thái *CD_ Insert* gồm hai trạng thái con (Include/Select Track và Include/Mode). Hai trạng thái này hoạt động đồng thời. Khi chèn một CD vào máy nghe nhạc, chúng ta có thể chọn bài hát và xem thông tin của nó hoặc chúng ta có thể chọn bài hát và nghe. Cả hai chế độ này được thực hiện tại cùng một thời điểm. Một cách khác, chúng ta có thể nói *CD_ Insert* là một "trạng thái ghép nối" với tính đồng thời. Đây chính là sự khác biệt chính của biểu đồ trạng thái so với máy hữu hạn trạng thái. Bởi cách biểu diễn này, biểu đồ trạng thái có thể đặc tả hệ thống với ít trạng thái hơn và vì vậy nó giảm độ phức tạp cho quá trình đặc tả và kiểm thử/kiểm chứng sau này.

8.2.4 Máy trạng thái UML

Các phương pháp đặc tả hình thức như máy hữu hạn trạng thái, biểu đồ trạng thái, v.v. giúp ta đặc tả các hệ thống một cách chính



Hình 8.3: Một ví dụ về biểu đồ trạng thái [BBH05].

xác với ý nghĩa duy nhất (vì chúng sử dụng các công cụ toán học). Tuy nhiên, các phương pháp này thường khó được áp dụng trong công nghiệp vì chúng đòi hỏi các chuyên gia về đặc tả hình thức.

Máy trạng thái UML được xem là giải pháp tốt để giải quyết vấn đề này. Nó có thể được sử dụng để đặc tả hành vi động (chuyển trạng thái) của các lớp đối tượng, các ca sử dụng (use cases), các hệ thống con và thậm chí là toàn bộ hệ thống. Tuy nhiên, máy trạng thái UML thường được sử dụng cho các lớp đối tượng. Theo [AJ00], biểu đồ cộng tác đặc tả bằng UML là một mô hình quan trọng trong việc kiểm thử hệ thống bởi mô hình này đặc tả chính xác hành vi (tương tác giữa các đối tượng) của hệ thống cần kiểm thử.

Trong UML, một trạng thái ứng với một điều kiện quan trọng của một đối tượng. Trạng thái này được quyết định bởi các giá trị hiện thời của đối tượng, các mối quan hệ với các đối tượng khác và các hành động (phương thức) mà đối tượng này thực hiện. Một phép chuyển trạng thái là mối quan hệ giữa hai trạng thái. Một phép chuyển trạng thái trong UML bao gồm một sự kiện được kích hoạt, điều kiện và hành động tương ứng. Các sự kiện được kích hoạt của các phép chuyển trạng thái có thể là một trong các sự kiện sau:

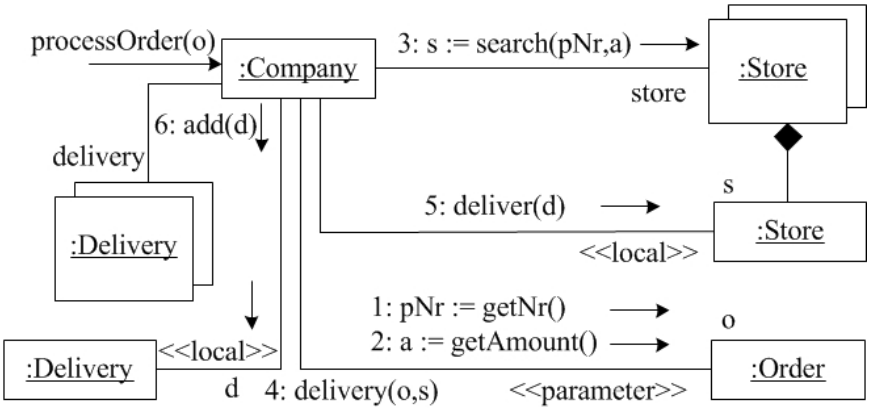
- Một lời gọi ứng với một phương thức

- Một tín hiệu nhận được từ các trạng thái khác trong máy trạng thái

- Một sự thay đổi giá trị của một thuộc tính nào đó của một đối tượng

- Hết thời gian (timeout)

Hình 8.4 là ví dụ về một máy trạng thái UML đặc tả hành vi của hệ thống quản lý bán hàng.



Hình 8.4: Một ví dụ về máy trạng thái UML.

8.2.5 Các phương pháp đặc tả khác

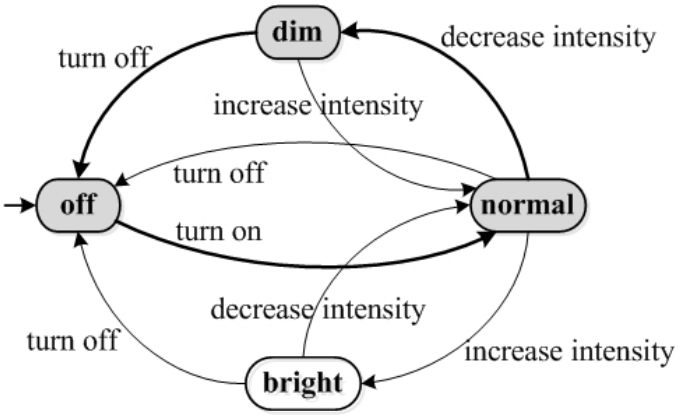
Ngoài những phương pháp đặc tả trên, có rất nhiều phương pháp đặc tả khác đã được đề xuất. Một số phương pháp đặc tả phổ biến như: mạng Petri (xem mục 3.6.3.2), chuỗi Markov, văn phạm, bảng quyết định/cây quyết định, ngôn ngữ ràng buộc đối tượng (OCL), các ngôn ngữ đặc tả đại số (Z, OBJ, v.v.), v.v. Phụ thuộc vào phương pháp và công cụ kiểm thử, chúng ta sẽ lựa chọn phương pháp đặc tả hệ thống tương ứng.

8.3 Sinh các ca kiểm thử từ mô hình

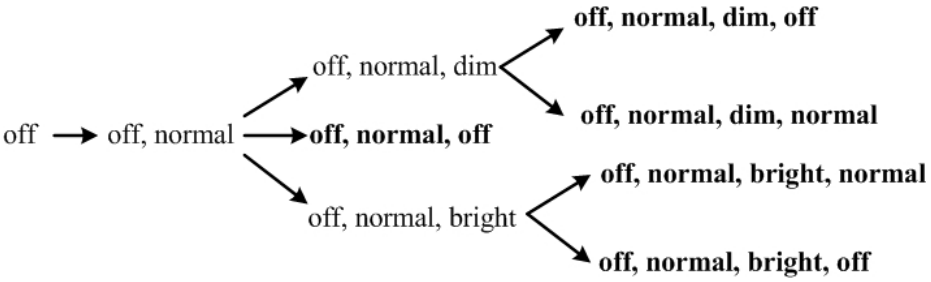
Khi chúng ta đã xây dựng được mô hình đặc tả chính xác hành vi của hệ thống, một trong những công việc khó khăn còn lại là làm thế nào để sinh được các ca kiểm thử từ mô hình này. Các ca kiểm thử cùng với giá trị đầu ra mong muốn sẽ được sử dụng để kiểm tra việc cài đặt hệ thống có đúng với đặc tả (bằng mô hình) hay không. Trong một số trường hợp, các ca kiểm thử được sinh ra bằng việc giải tất cả các hệ phương trình ứng với mỗi đường đi trong mô hình [Bor95]. Với mô hình được đặc tả bằng FSM, chúng ta có thể duyệt ngẫu nhiên trên FSM thông qua các trạng thái và các chuyển trạng thái giữa chúng [AW93, SvBGF+91]. Một đường đi từ trạng thái khởi tạo đến một trạng thái kết thúc tương ứng với một ca kiểm thử chúng ta muốn tạo ra. Trong hình 8.5, một đường đi *off*, *normal*, *dim*, *off* (phần bôi đậm) tương ứng chuỗi hành động *turn on*, *decrease intensity*, *turn off* sẽ là một ca kiểm thử từ FSM này. Để đảm bảo tất cả các đường đi có thể trong FSM phải được kiểm thử, chúng ta có thể áp dụng các thuật toán duyệt FSM theo chiều rộng hoặc theo chiều sâu (như duyệt các đồ thị) nhằm liệt kê tất cả các đường đi. Ví dụ, hình 8.6 mô tả quá trình sinh các đường đi từ máy hữu hạn trạng thái trong hình 8.5. Trong hình này, các đường đi bôi đậm chính là các đường đi chúng ta cần tạo ra nhằm kiểm thử hệ thống. Chúng ta bắt đầu từ trạng thái *off*. Trạng thái này chỉ có một phép chuyển trạng thái đến trạng thái *normal* nên chúng ta chỉ có một đường đi *off, normal*. Có ba phép chuyển trạng thái từ trạng thái*normal* nên chúng ta có ba đường đi được sinh ra là *off, normal, dim*, *off, normal, off* và *off, normal, bright*. Do trạng thái off đã được duyệt nên *off, normal, off* là một đường đi cần tìm. Chúng ta tiếp tục xử lý với hai đường đi còn lại một cách tương tự.

Với mỗi đường đi, chúng ta sẽ sinh các đầu vào cho ca kiểm thử tương ứng. Chúng ta có thể chọn ngẫu nhiên hoặc sử dụng các

thuật toán giải các hệ phương trình ứng với các điều kiện từ dãy các trạng thái của đường đi này. Bộ đầu vào có được đảm bảo điều kiện đường đi tương ứng sẽ được thực thi khi chạy chương trình. Tuy nhiên, trong một số trường hợp ví dụ như kiểm thử tương tác giao diện người dùng, chúng ta không cần sinh các giá trị đầu vào cho mỗi đường đi. Chi tiết về các trường hợp này có thể tham khảo trong ví dụ minh họa được trình bày trong mục 8.6.



Hình 8.5: Một ví dụ về đường đi trong máy hữu hạn trạng thái.



Hình 8.6: Sinh các đường đi từ máy hữu hạn trạng thái.

8.4 Sinh đầu ra mong muốn cho các ca kiểm thử

Để có được một ca kiểm thử hoàn chỉnh, chúng ta cần tính được giá trị đầu ra mong đợi để so sánh với giá trị thực khi chạy chương

trình ứng với bộ đầu vào tương ứng. Hiện tại, việc tính giá trị đầu ra mong muốn được thực hiện thủ công dựa vào tri thức của người kiểm thử đối với hệ thống. Việc tự động tính giá trị đầu ra mong muốn đang là một bài toán khó và chưa có giải pháp thỏa đáng. Hiện tại, có rất nhiều phương pháp hướng đến giải quyết bài toán này như phương pháp học máy [MMA04, MMA02], phương pháp thống kê [MG04], metamorphic [PZKH06]. Tuy nhiên, chi phí để áp dụng các phương pháp này là rất lớn và đòi hỏi tri thức của các chuyên gia về các lĩnh vực này. Một giải pháp khác nhằm hiện thực hóa bài toán này là dùng hai cài đặt tương đương tức là lập trình hai hệ thống với hai ngôn ngữ khác nhau hoặc với hai nhóm khác nhau. Với giải pháp này, một cài đặt dùng để tính kết quả thực và hệ thống còn lại dùng để tính giá trị mong đợi.

8.5 Thực hiện các ca kiểm thử

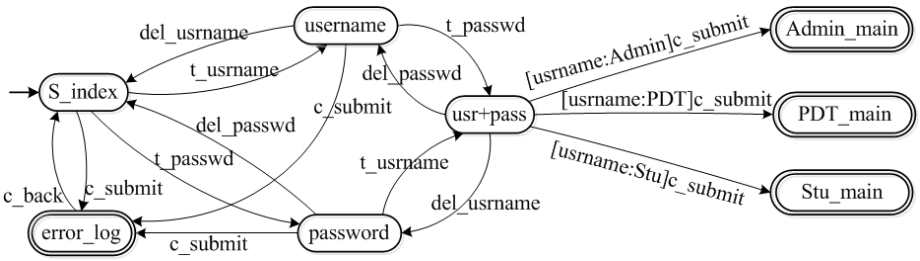
Sau khi đã có tập các ca kiểm thử, chúng ta sẽ sử dụng chúng nhằm phát hiện các lỗi lập trình. Để đạt được mục đích này, chúng ta sẽ tiến hành cài đặt hệ thống dựa trên mô hình đã được đặc tả. Mô hình này được xem như một thiết kế chuẩn của hệ thống. Khi đã cài đặt xong chương trình, chúng ta sẽ tiến hành thực thi các ca kiểm thử. Chúng ta có thể thực hiện công việc này một cách thủ công, viết các kịch bản để thực thi hoặc sử dụng các công cụ hỗ trợ sẵn có. Kết quả của việc kiểm thử sẽ được phân tích nhằm xác định các bước tiếp theo. Một trong những tình huống có thể xảy ra là thiết kế có thể sai và chúng ta cần quay lại bước xây dựng mô hình để chỉnh sửa lại nó. Trong trường hợp này, chúng ta phải thực hiện lại tất cả các bước của phương pháp kiểm thử dựa trên mô hình. Tình huống thường gặp là kết quả kiểm thử cho chúng ta biết các lỗi lập trình. Trong trường hợp này, chúng ta tiến hành sửa lại mã nguồn và thực hiện lại tất cả các ca kiểm thử.

8.6 Ví dụ minh họa

Trong mục này, chúng ta sẽ áp dụng phương pháp kiểm thử dựa trên mô hình nhằm kiểm thử tính đúng đắn của việc cài đặt trang Web đăng nhập cho hệ thống đăng ký môn học so với thiết kế của nó. Trong ví dụ này, chúng ta chỉ quan tâm đến thiết kế tương tác màn hình của hệ thống. Thiết kế này sẽ được đặc tả bằng DFA. Các kỹ thuật đặc tả hệ thống bằng DFA, sinh các ca kiểm thử từ DFA và thực hiện các ca kiểm thử để phát hiện các lỗi cài đặt sẽ được trình bày chi tiết trong ví dụ này.

8.6.1 Đặc tả hệ thống

Giả sử chúng ta cần một trang Web đăng nhập hệ thống đăng ký môn học cho ba loại người dùng: quản trị hệ thống, nhân viên phòng đào tạo và sinh viên. Để đăng nhập, mỗi người dùng phải nhập tên đăng nhập và mật khẩu rồi nhấn nút "Submit" để đăng nhập.



Hình 8.7: DFA cho trang đăng nhập của Hệ thống đăng ký môn học.

Hình 8.7 là đặc tả bằng DFA tương ứng với thiết kế tương tác màn hình của trang Web này. Ban đầu, trạng thái khởi tạo của trang Web là *S_index* ứng với trạng thái của hệ thống chuẩn bị đăng nhập. Ở trạng thái này, nếu người dùng nhấn nút "Submit" ứng với hành động *c_submit* thì hệ thống chuyển đến trạng thái *error_log* (lỗi đăng nhập). Đây cũng là một trạng thái kết thúc của

hệ thống. Tại trạng thái này, nếu người dùng sử dụng nút "Go back" của trình duyệt (ứng với hành động *c_back*), hệ thống sẽ quay lại trạng thái khởi tạo. Ngược lại, nếu người dùng gõ tên đăng nhập (ứng với hành động *t_usrname*) thì hệ thống chuyển từ trạng thái *S_index* sang trạng thái *username* (đã nhập tên đăng nhập). Tại trạng thái này, nếu người dùng xóa tên đăng nhập (ứng với hành động *del_usrname*), hệ thống sẽ quay về trạng thái *S_index*. Nếu người dùng nhập mật khẩu tại trạng thái *S_index* ứng với hành động *t_passwd* thì trạng thái mới của hệ thống sẽ là *password* (đã nhập mật khẩu). Nếu người dùng xóa mật khẩu tại trạng thái này (ứng với hành động *del_passwd*) thì hệ thống sẽ quay về trạng thái khởi tạo. Tại trạng thái *username*, nếu người dùng tiếp tục nhập mật khẩu thì hệ thống sẽ chuyển đến trạng thái mới có tên là *usr+pass* (đã nhập xong tên đăng nhập và mật khẩu). Tại trạng thái mới này, nếu mật khẩu bị xóa thì hệ thống sẽ trở lại trạng thái trước đó. Tương tự, nếu người dùng nhập tên đăng nhập tại trạng thái *password*, hệ thống sẽ chuyển đến trạng thái *usr+pass*. Tại trạng thái này, nếu tên đăng nhập bị xóa, hệ thống sẽ quay lại trạng thái *password*. Tại trạng thái *usr+pass*, cùng một hành động *c_submit* nhưng phụ thuộc vào điều kiện về loại người dùng mà hệ thống sẽ chuyển tới một trong ba trạng thái mới. Ba trạng thái này cũng là ba trạng thái kết thúc của hệ thống. Nếu người dùng là quản trị hệ thống (ứng với điều kiện [usrname:Admin]), trạng thái mới của hệ thống sẽ là *Admin_main*. Tương tự, nếu người dùng là nhân viên phòng đào tạo (ứng với điều kiện [usrname:PDT]), trạng thái mới của hệ thống sẽ là *PDT_main*. Cuối cùng, nếu người dùng là sinh viên (ứng với điều kiện [usrname:Admin]) thì hệ thống sẽ chuyển đến trạng thái *Stu_main*.

Chúng ta có thể biểu diễn đặc tả hệ thống (hình 8.7) dưới dạng bảng chuyển bằng tệp Excel như trong hình 8.8. Tệp Excel này sẽ được cung cấp làm đầu vào cho các công cụ hỗ trợ kiểm thử dựa trên mô hình.

C5 — c_submit:Admin->Admin_main/PDT->PDT_main/Stu->Stu_main

A	B	C	D	E	F	G	H	I	J	K
		Điều kiện	S_index	username	password	usr+pass	Admin_main	PDT_main	Stu_main	error_log
->	S_index			t_usrname	t_passwd					c_submit
	username		del_usrname			t_passwd				c_submit
	password		del_passwd			t_usrname				c_submit
	usr+pass	c_submit:		del_passwd	del_usrname		c_submit	c_submit	c_submit	
*	Admin_main									
*	PDT_main									
*	Stu_main									
*	error_log		c_back							

Hình 8.8: Biểu diễn DFA cho trang đăng nhập bằng Excel.

8.6.2 Sinh các ca kiểm thử

Mỗi ca kiểm thử là một đường đi từ trạng thái ban đầu đến một trạng thái kết thúc của hệ thống. Mỗi ca kiểm thử phải có ít nhất một phép chuyển trạng thái. Trong ví dụ này, chúng tôi biểu diễn các ca kiểm thử theo dạng như ví dụ sau.

Ví dụ, *S_index*t_usrname=username*c_submit=error_log* là một ca kiểm thử. Bắt đầu từ trạng thái khởi tạo *S_index*, ứng với hành động *t_usrname*, trạng thái mới của hệ thống là *username*. Tiếp đến, ứng với hành động *c_submit*, hệ thống sẽ chuyển đến trạng thái kết thúc là *error_log*.

Vấn đề còn lại của chúng ta là làm thế nào để sinh ra các ca kiểm thử có khả năng phát hiện tối đa các lỗi lập trình. Trước hết, chúng ta cần xác định tiêu chí của phương pháp sinh các ca kiểm thử từ đặc tả hệ thống. Có một số tiêu chí phổ biến như tất cả các trạng thái phải xuất hiện ít nhất một lần trong tất cả các ca kiểm thử, tất cả các phép chuyển trạng thái phải xuất hiện ít nhất một lần trong tất cả các ca kiểm thử, v.v. Trong ví dụ này, chúng ta sẽ sử dụng tiêu chí thứ hai.

Để sinh các ca kiểm thử đáp ứng tiêu chí này, chúng ta xuất phát từ trạng thái khởi tạo. Từ trạng thái này, ứng với các trạng thái tiếp theo của nó sao cho phép chuyển trạng thái này chưa được duyệt, chúng ta sẽ sinh ra các phần của các ca kiểm thử. Với mỗi phần này ứng với mỗi trạng thái mới, chúng ta tiến hành tương tự

như trên. Trong trường hợp tất cả các phép chuyển trạng thái từ trạng thái đang xét đã được viếng thăm, nếu trạng thái này không là trạng thái kết thúc thì ta sẽ chọn ngẫu nhiên một dãy các phép chuyển trạng thái tiếp theo tới một trạng thái kết thúc. Khi ta gặp trạng thái kết thúc thì thuật toán sẽ trả lại một ca kiểm thử tương ứng. Thuật toán sẽ kết thúc khi tất cả các phép chuyển trạng thái của hệ thống được viếng thăm. Với ý tưởng này, chúng ta sẽ sinh ra được bảy ca kiểm thử ứng với đặc tả DFA như trong hình 8.7.

- *S_index*c_submit=error_log*
- *S_index*t_usrname=username*c_submit=error_log*
- *S_index*t_usrname=username*t_passwd=usr+pass* c_submit=Admin_main*
- *S_index*t_usrname=username*t_passwd=usr+pass* c_submit=PDT_main*
- *S_index*t_usrname=username*t_passwd=usr+pass* c_submit=Stu_main*
- *S_index*t_passwd=password*t_usrname=usr+pass* del_passwd=username*c_submit=error_log*
- *S_index*t_usrname=username*t_passwd=usr+pass* del_usrname=password*c_submit=error_log*c_back= S_index*c_submit=error_log*

8.6.3 Thực hiện các ca kiểm thử

Sau khi chúng ta đã cài đặt trang Web đăng nhập hệ thống đăng ký môn học như đã mô tả, các ca kiểm thử trên sẽ được sử dụng nhằm kiểm tra xem việc cài đặt có tuân thủ đặc tả như hình 8.7 hay không. Để thực hiện mục tiêu này, chúng ta có thể sử dụng

các Web Driver. Trong ví dụ này, chúng ta sẽ sử dụng Selenium[1]. Khi cung cấp các ca kiểm thử cho Selenium cũng với tên miền (hoặc url) của trang Web cần kiểm thử, Selenium sẽ thực thi hệ thống và lấy các thông tin về các phần tử của trang Web tương ứng với mỗi trạng thái của hệ thống. Với mỗi ca kiểm thử, bắt đầu từ trạng thái khởi tạo, ứng với từng sự kiện của hệ thống như: click (*c_submit, c_back*), addtext (*t_usrname, t_passwd*), deltext (*del_usrname, del_passwd*), Selenium sẽ xác định trạng thái tiếp theo của hệ thống. Nếu trạng thái này không trùng với trạng thái tiếp theo như trong ca kiểm thử, công cụ này sẽ thông báo có lỗi. Nếu thực hiện hết các sự kiện của ca kiểm thử và không phát hiện ra lỗi, ca kiểm thử là thỏa mãn.

Ví dụ trên chỉ cho phép kiểm thử riêng biệt từng trang Web. Đây là một ví dụ đơn giản nhưng rất hiệu quả trong việc phát hiện các lỗi lập trình. Điều đặc biệt của phương pháp này là chúng ta không cần quan tâm đến ngôn ngữ lập trình mà chúng ta sử dụng để cài đặt trang Web vì Selenium chỉ tương tác với các trình duyệt khi thực hiện việc kiểm thử hệ thống.

Để kiểm thử một Website gồm nhiều trang Web tương tác với nhau, chúng ta cũng tiến hành tương tự như ví dụ trên. Khi chúng ta có được đặc tả của từng trang Web, đặc tả của hệ thống sẽ thu được bằng cách ghép nối tất cả các đặc tả ứng với các trang Web. Khi có được đặc tả của hệ thống, các bước còn lại sẽ được tiến hành như trên một trang Web. Chúng tôi đã phát triển các công cụ hỗ trợ phương pháp kiểm thử dựa trên mô hình cho các ứng dụng Web như ví dụ trên. Các công cụ này cùng tài liệu hướng dẫn sử dụng và các ví dụ áp dụng được cung cấp tại địa chỉ[2]. Chúng tôi cũng đã cung cấp mã nguồn của công cụ này nhằm cho phép các sinh viên có thể mở rộng công cụ này phục vụ các mục đích học tập và nghiên cứu.

[1] www.seleniumhq.org

[2] http://uet.vnu.edu.vn/~hungpn/ATWATool/

8.7 Thuận lợi và khó khăn của kiểm thử dựa trên mô hình

Trong quá trình phát triển phần mềm, các kiểm thử viên thường thực hiện công việc của mình bằng các phương pháp truyền thống (thủ công) nên thời gian và chi phí dành cho các hoạt động này thường rất cao. Kiểm thử dựa trên mô hình hứa hẹn sẽ là một giải pháp hiệu quả nhằm góp phần giải quyết vấn đề này. Cụ thể, kiểm thử dựa trên mô hình có các ưu điểm sau:

- **Giảm chi phí và thời gian:** Do quá trình kiểm thử hầu hết được thực hiện tự động nên tính hiệu quả của phương pháp này rất cao trong khi thời gian được giảm một cách tối thiểu.

- **Độ bao phủ tốt hơn:** Nếu mô hình của hệ thống được xây dựng tốt thì quá trình kiểm thử dựa trên mô hình sinh ra nhiều ca kiểm thử và phát hiện nhiều lỗi. Kiểm thử mô hình cũng cho phép giảm các lỗi chủ quan do người kiểm thử sinh ra trong quá trình kiểm thử sản phẩm.

- **Đầy đủ tài liệu:** Mô hình hệ thống, các đường đi, các ca kiểm thử, ... là các tài liệu quan trọng trong quá trình phát triển phần mềm nói chung và quá trình kiểm thử phần mềm nói riêng. Các tài liệu này cũng giúp cho các kiểm thử viên hiểu hơn về các ca kiểm thử và các kịch bản kiểm thử.

- **Khả năng sử dụng lại cao:** Mỗi khi phần mềm bị tiến hóa, chúng ta dễ dàng sinh thêm các ca kiểm thử và kiểm thử lại một cách nhanh chóng và hiệu quả.

- **Hiểu hơn về hệ thống:** Kiểm thử dựa trên mô hình giúp người phát triển hiểu hơn về hệ thống cần kiểm thử thông qua việc xây dựng và phân tích mô hình hệ thống.

- Sớm phát hiện lỗi và sự không rõ ràng trong đặc điểm kỹ thuật và thiết kế, vì vậy sẽ tăng thời gian giải quyết vấn đề trong kiểm thử.
- Tự động tạo và kiểm tra nhằm tránh các ca kiểm thử trùng nhau hoặc không hữu hiệu.
- Kiểm thử dựa trên mô hình có khả năng đánh giá chất lượng phần mềm.

Tuy nhiên, kiểm thử dựa trên mô hình không dễ được áp dụng trong thực tế vì một số khó khăn sau:

- **Khó xây dựng mô hình chính xác:** Kiểm thử dựa trên mô hình cần có mô hình đặc tả chính xác hành vi của hệ thống. Trong thực tế, việc xây dựng mô hình là rất khó, tốn kém và tiềm ẩn nhiều lỗi. Đây là một trong những hạn chế lớn nhất của phương pháp này.
- **Yêu cầu cao về kiểm thử viên:** Do phải xây dựng mô hình của hệ thống vì vậy người kiểm thử phần mềm phải yêu cầu là những người có khả năng phân tích và thiết kế hệ thống. Hơn nữa, người kiểm thử cần có kiến thức tốt về các phương pháp hình thức và đặc tả hình thức, có hiểu biết chi tiết và chính xác về hệ thống.
- Tạo giá trị đầu ra mong đợi cho các ca kiểm thử là một trong những vấn đề khó khăn nhất của kiểm thử dựa trên mô hình.
- **Khó khăn trong việc sử dụng các ca kiểm thử được tạo ra từ mô hình:** Lập trình viên tiến hành cài đặt hệ thống một cách độc lập nên khi đã cài đặt xong thường khó thực thi các ca kiểm thử được tạo ra từ mô hình vì rất nhiều lý do khác nhau. Thông thường, họ phải tiến hành nghiên cứu mô hình và đặc tả lại các ca kiểm thử mới sử dụng được

chúng. Hơn nữa, mô hình hệ thống thường trừu tượng và tổng quát hơn cài đặt của nó. Vấn đề này là một trong những lý do chính của hạn chế này.

8.8 Một số công cụ kiểm thử dựa trên mô hình

Hiện tại, có rất nhiều công cụ hỗ trợ phương pháp kiểm thử dựa trên mô hình nói riêng và kiểm thử tự động nói chung. Trong mục này, chúng ta sẽ tìm hiểu một số công cụ kiểm thử điển hình. Ngoài các công cụ hỗ trợ kiểm thử dựa trên mô hình, một số công cụ hỗ trợ kiểm thử tự động dựa trên mã nguồn cũng được giới thiệu nhằm cung cấp một cái nhìn tổng quan về các công cụ kiểm thử tự động.

8.8.1 AGEDIS

AGEDIS [AK04a, AK04b] là một công cụ kiểm thử dựa trên mô hình. Công cụ này là sản phẩm của dự án AGEDIS được tài trợ bởi Ủy ban châu Âu. AGEDIS cho phép đặc tả mô hình, tạo các ca kiểm thử, thực hiện các ca kiểm thử và các chức năng khác. Ba loại thông tin đầu vào của công cụ này nhằm đặc tả hệ thống cần kiểm thử gồm:

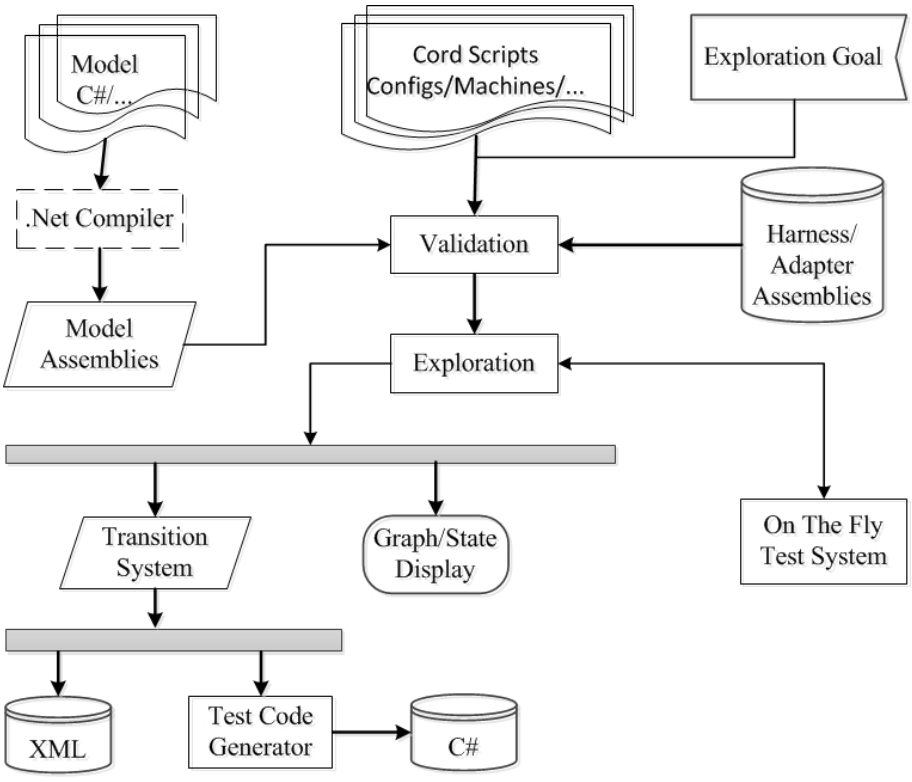
- Mô hình hành vi của hệ thống
- Các định hướng về việc thực hiện các ca kiểm thử nhằm mô tả quy trình kiểm thử cho hệ thống
- Các định hướng cho việc sinh các ca kiểm thử nhằm mô tả các chiến lược cho mục tiêu này

Hành vi của hệ thống cần kiểm thử và các định hướng về việc thực hiện các ca kiểm thử được đặc tả bằng các biểu đồ UML (biểu đồ lớp, biểu đồ trạng thái và biểu đồ đối tượng). Các định hướng cho việc sinh các ca kiểm thử được đặc tả bằng XML.

AGEDIS đã được sử dụng cho nhiều dự án trong thực tế. Nó được đánh giá là một công cụ tốt cho kiểm thử dựa trên mô hình.

8.8.2 Spec Explorer

Spec Explorer là một công cụ kiểm thử dựa trên mô hình được phát triển bởi Microsoft. Nó được tích hợp trong Visual Studio nhằm tạo mô hình từ chương trình, phân tích thăm dò mô hình bằng công cụ biểu đồ trực quan thông qua việc hiển thị các trạng thái của chương trình, kiểm tra tính đúng đắn của mô hình, sinh ra các ca kiểm thử từ mô hình và thực hiện chúng.



Hình 8.9: Kiến trúc của Spec Explorer.

Hình 8.9 mô tả các luồng làm việc của Spec Explorer. Trong kiến trúc này, mô hình của hệ thống được biên dịch thành mô hình

được đặc tả bằng Assembly. Các mã Assembly này và các cấu hình trong tệp Config.Cord được đưa vào bộ thăm dò. Công việc của bộ thăm dò là hiển thị biểu đồ trạng thái hoặc hệ thống chuyển tiếp, từ hệ thống chuyển tiếp có thể tạo ra các tệp XML chứa các thông tin về các kiểm thử hoặc có thể tự động sinh ra mã của các ca kiểm thử. Spec Explorer được đánh giá là một công cụ mạnh và đã được sử dụng trong rất nhiều dự án của Microsoft.

8.8.3 Conformiq Qtronic

Conformiq Qtronic[1] là một công cụ kiểm thử dựa trên mô hình cho phép kiểm thử tự động các hệ thống nhúng và các hệ thống giao dịch điện tử. Các chức năng chính của phương pháp kiểm thử dựa trên mô hình đều được hỗ trợ bởi công cụ này. Mô hình trong công cụ này được đặc tả bằng biểu đồ trạng thái UML và hỗ trợ các ngôn ngữ phổ biến như C/C++, C# và Java. Khi một biểu đồ trạng thái UML đặc tả hành vi của hệ thống cần kiểm thử được cung cấp, công cụ này tiến hành phân tích và sinh ra tập các ca kiểm thử. Điều đặc biệt ở công cụ này là nó có thể sinh các ca kiểm thử từ các hành vi không đơn định. Để thực hiện các ca kiểm thử, người kiểm thử cần định nghĩa các kịch bản cho phép chuyển đổi các ca kiểm thử thành định dạng có thể thực hiện trên hệ thống cần kiểm thử.

8.8.4 JCrasher

JCrasher là một công cụ kiểm thử tự động mạnh mẽ cho ngôn ngữ Java. Công cụ này kiểm tra dữ liệu của các chương trình Java và sinh ra các đoạn mã để tạo ra các đối tượng có kiểu khác nhau để kiểm thử các phương thức public với dữ liệu ngẫu nhiên. JCrasher cố gắng phát hiện lỗi bằng cách làm chương trình lắng nghe các ngoại lệ được sinh ra khi các chương trình Java được thực thi.

[1] http://www.conformiq.com/qtronic.php

Mặc dù các phương pháp kiểm thử ngẫu nhiên còn nhiều hạn chế như tính bao phủ của các ca kiểm thử không cao, v.v., lợi thế của phương pháp này là hoàn toàn tự động quá trình kiểm thử.

So với các công cụ khác, JCrasher có nhiều đặc trưng mới như sau:

- Phân tích các phương thức một cách tự động, xác định kích thước của các tham số của phương thức cần kiểm thử và lựa chọn tổ hợp các tham số để kiểm thử ngẫu nhiên.
- Có một cơ sở tri thức để quyết định một ngoại lệ có nên được xem là lỗi hay không.
- Cho phép khôi phục tất cả các trạng thái của quá trình kiểm thử trước đó một cách hiệu quả.
- Sinh ra các ca kiểm thử có thể thực hiện bởi công cụ JUnit.
- JCrasher có thể được tích hợp vào Eclipse như là một plug-in của môi trường lập trình này.

8.8.5 Selenium

Selenium[1] là một phần mềm mã nguồn mở, được phát triển bởi Jason Huggins, sau đó được tiếp tục bởi nhóm ThoughtWorks vào năm 2004. Phiên bản hoàn chỉnh mới nhất là 1.0.1 được phát hành vào 10/06/2009. Đây là một công cụ hỗ trợ kiểm thử tự động cho các ứng dụng Web. Selenium hỗ trợ kiểm thử trên hầu hết các trình duyệt phổ biến hiện nay như Firefox, Internet Explorer, Safari, v.v. cũng như các hệ điều hành chủ yếu như Windows, Linux, Mac, v.v. Selenium cũng hỗ trợ một số lớn các ngôn ngữ lập trình Web phổ biến hiện nay như C#, Java, Perl, PHP, Python, Ruby, v.v. Công cụ này có thể kết hợp thêm với một số công cụ khác như Bromien

[1] www.seleniumhq.org

và JUnit nhưng với người dùng thông thường chỉ cần chạy tự động mà không cần cài thêm các công cụ bổ trợ. Selenium đang được cộng đồng sử dụng đánh giá là một trong những công cụ tốt nhất cho kiểm thử tự động các ứng dụng Web.

8.8.6 SoapUI

SoapUI[1] là một phần mềm thương mại cung cấp một giải pháp kiểm thử chức năng hoàn chỉnh và đa nền tảng. SoapUI cũng cung cấp bản miễn phí với các tính năng giới hạn. Công cụ này được sử dụng chủ yếu cho kiểm thử chức năng các dịch vụ Web. SoapUI có giao diện đồ họa dễ sử dụng giúp người dùng dễ dàng tạo ra các kịch bản kiểm thử cũng như thực hiện việc kiểm thử chức năng, kiểm thử hồi quy một cách tự động và cực kỳ nhanh chóng. Ngoài các tính năng được cung cấp sẵn, chúng ta có thể sử dụng ngôn ngữ/thư viện Groovy/Javascript để lập trình thêm các tình huống kiểm thử đối với các chức năng khó. Đầu vào của công cụ này là địa chỉ của dịch vụ Web cần kiểm thử (dưới dạng URL hoặc IP) và đầu ra là báo cáo kiểm thử. Chúng ta cũng có thể tùy biến định dạng của báo cáo này theo nhu cầu của người kiểm thử.

8.8.7 W3af

W3af (Web Application Attack and Audit Framework[2]) là một công cụ hỗ trợ kiểm thử bảo mật cho các ứng dụng Web bằng cách cung cấp môi trường thích hợp để kiểm soát và tấn công các ứng dụng này. Đầu vào của công cụ này là địa chỉ của ứng dụng Web cần kiểm thử và cấu hình cho chiến lược kiểm thử ứng dụng này. Đầu ra của nó là một báo cáo về các lỗ hổng bảo mật tiềm ẩn cần khắc phục của ứng dụng.

[1] http://www.soapui.org
[2] http://w3af.org

8.9 Tổng kết

Kiểm thử dựa trên mô hình nói riêng và kiểm thử tự động nói chung đang được xem như là các giải pháp hiệu quả nhằm giảm chi phí và tăng hiệu quả cho các hoạt động kiểm thử và đảm bảo chất lượng của các sản phẩm phần mềm. Để áp dụng các phương pháp kiểm thử dựa trên mô hình, chúng ta cần xây dựng mô hình đặc tả chính xác hành vi của hệ thống cần kiểm thử. Tùy vào yêu cầu kiểm thử và đặc trưng của hệ thống, chúng ta sẽ lựa chọn phương pháp đặc tả phù hợp. Xây dựng mô hình là một trong những công việc khó khăn nhất trong kiểm thử dựa trên mô hình vì trong thực tế hoạt động này thường tiềm ẩn nhiều lỗi. Để đảm bảo tính chính xác của các mô hình này, chúng ta có thể áp dụng các phương pháp kiểm chứng mô hình (model checking) và chứng minh định lý (theorem proving). Một khi đã có được mô hình, chúng ta sẽ xây dựng các đường đi từ trạng thái khởi tạo đến các trạng thái kết thúc của hệ thống. Ứng với mỗi đường đi, chúng ta sẽ xây dựng một ca kiểm thử. Các ca kiểm thử này sẽ được sử dụng để phát hiện những lỗi lập trình. Trong thực tế, có rất nhiều công cụ hỗ trợ kiểm thử dựa trên mô hình và kiểm thử tự động. Các công cụ này đã góp phần không nhỏ trong việc nâng cao chất lượng cho các sản phẩm và tăng tính hiệu quả trong các hoạt động kiểm thử.

8.10 Bài tập

1. Trình bày mục đích của kiểm thử dựa trên mô hình?

2. Tại sao kiểm thử dựa trên mô hình có vai trò quan trọng trong việc nâng cao tính hiệu quả và giảm chi phí cho các hoạt động kiểm thử?

3. Hãy phân tích các ưu điểm của kiểm thử dựa trên mô hình.

4. Hãy phân tích những nhược điểm của kiểm thử dựa trên mô hình.

5. Dựa trên những kiến thức về kiểm thử dựa trên mô hình, hãy trình bày các bước nhằm tự động hóa phương pháp kiểm thử dòng điều khiển (đã được giới thiệu trong chương 6).

6. Dựa trên những kiến thức về kiểm thử dựa trên mô hình, hãy trình bày các bước nhằm tự động hóa phương pháp kiểm thử dòng dữ liệu (đã được giới thiệu trong chương 7).

7. Mô tả các bước nhằm áp dụng phương pháp kiểm thử dựa trên mô hình.

8. Cho công cụ kiểm thử dựa trên mô hình tại địa chỉ[1], hãy thực hiện các yêu cầu sau:

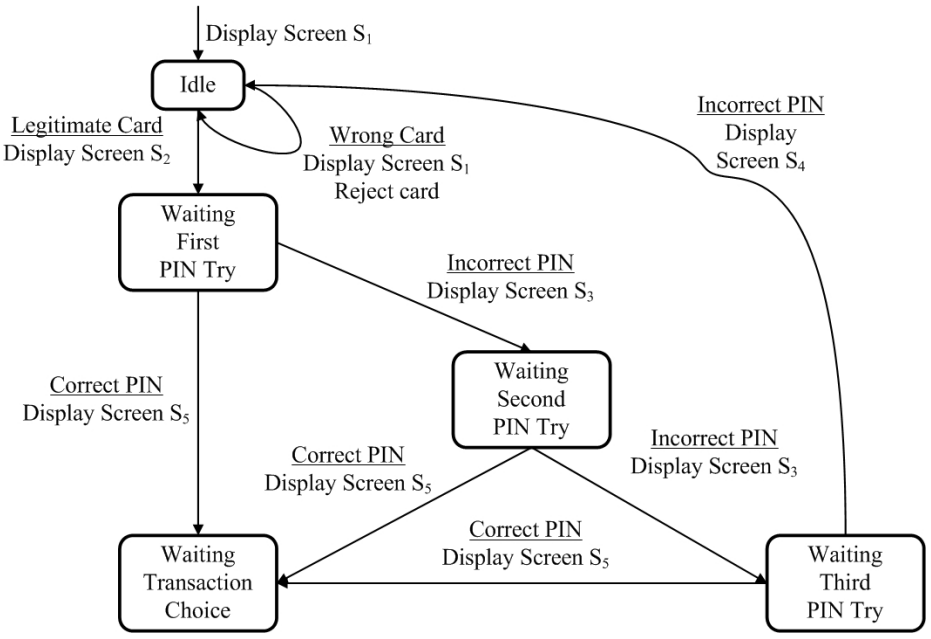
   - Tìm hiểu công cụ thông qua ví dụ đã được cung cấp
   - Tìm hiểu phương pháp đặc tả các máy hữu hạn trạng thái bằng các tệp MS. Excel
   - Sử dụng công cụ JFLAP để tạo các tệp MS. Excel. Vai trò của công cụ này là gì?
   - Tìm hiểu kỹ thuật ghép nối hai mô hình ứng với tương tác giao diện của hai trang Web
   - Tìm hiểu kỹ thuật thực hiện các ca kiểm thử trong công cụ này
   - Áp dụng công cụ này nhằm kiểm thử tương tác giao diện của một ứng dụng Web.

9. Hãy sử dụng một trong các công cụ đã giới thiệu để kiểm thử cho một ứng dụng cụ thể.

[1] http://uet.vnu.edu.vn/~hungpn/ATWATool/

10. Hãy lấy các ví dụ ứng với các phương pháp đặc tả mô hình (biểu đồ trạng thái, máy trạng thái UML).

11. Cho máy hữu hạn trạng thái mô tả trong hình 8.10.

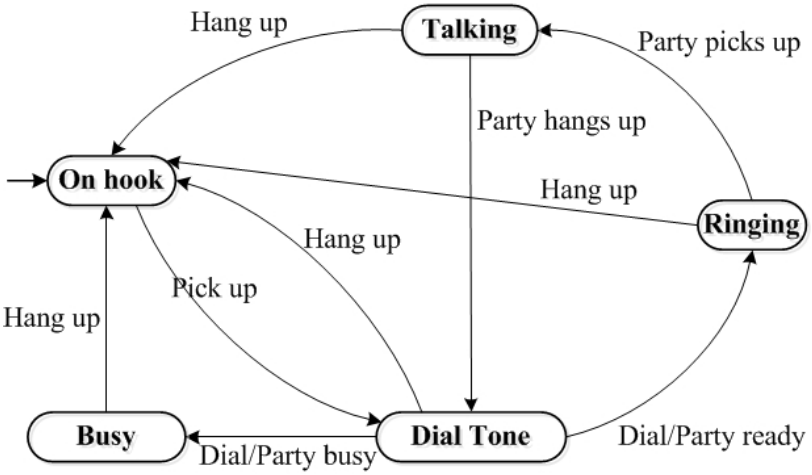


Hình 8.10: FSM cho một phần của máy ATM đơn giản.

- Hãy mô tả bằng lời hành vi của hệ thống được đặc tả trong máy hữu hạn trạng thái này.
- Xây dựng bảng chuyển cho FSM này.
- Hãy liệt kê tất cả các đường đi của hệ thống.
- Sinh các ca kiểm thử ứng với các đường đi.

12. Cho máy hữu hạn trạng thái mô tả trong hình 8.11.

- Hãy mô tả bằng lời hành vi của hệ thống được đặc tả trong máy hữu hạn trạng thái này.



Hình 8.11: Máy hữu hạn trạng thái của máy điện thoại.

- Xây dựng bảng chuyển cho FSM này.
- Hãy liệt kê tất cả các đường đi của hệ thống.
- Sinh các ca kiểm thử ứng với các đường đi.

Chương 9

Kiểm thử tự động và công cụ hỗ trợ

Kiểm thử đang được xem là giải pháp chủ yếu nhằm đảm bảo chất lượng cho các sản phẩm phần mềm. Tuy nhiên, các hoạt động kiểm thử hiện nay chủ yếu được thực hiện một cách thủ công và tiêu tốn khoảng 30-50% tài nguyên (thời gian, nhân lực và chi phí) của quá trình phát triển sản phẩm phần mềm. Hơn nữa, độ phức tạp của các phần mềm ngày càng tăng và trong môi trường cạnh tranh như hiện nay đòi hỏi các công ty phần mềm phải áp dụng các phương pháp và công cụ nhằm tự động hóa các hoạt động kiểm thử. Hơn nữa, các công cụ kiểm thử tự động là một giải pháp hữu hiệu cho kiểm thử hồi quy. Chương này giới thiệu tổng quan về kiểm thử tự động và các công cụ hỗ trợ nhằm giải quyết vấn đề này.

9.1 Tổng quan về kiểm thử tự động

Một thực tế đáng buồn hiện nay là chất lượng của hầu hết các sản phẩm phần mềm rất thấp. Hàng năm, chúng ta phải chịu thiệt hại

nhiều tỷ đô la do các lỗi phần mềm gây ra [oST02a, G.95]. Theo thống kê của NIST công bố năm 2002 [oST02b], chất lượng phần mềm thấp đã gây thiệt hại cho kinh tế Mỹ 60 tỷ đô la mỗi năm và tiêu tốn khoảng 22 tỷ đô la cho việc phát triển các công cụ nhằm phát hiện các lỗi và kiểm thử tự động. Có hai lý do chính dẫn đến tình trạng này. Thứ nhất, hầu hết các công cụ hiện nay đều tập trung vào việc thực thi tự động các ca kiểm thử (auto-test execution) trong khi vấn đề cốt lõi của kiểm thử là các phương pháp và kỹ thuật sinh các ca kiểm thử vẫn còn thiếu. Thứ hai, các công cụ hiện nay chưa hỗ trợ một cách hiệu quả cho kiểm thử hồi quy (regression testing). Một khi phần mềm bị tiến hóa/thay đổi, chúng ta cần kiểm thử lại sản phẩm. Làm thế nào để sử dụng lại các ca kiểm thử đã có và sinh ra các ca kiểm thử mới một cách hiệu quả đang là một vấn đề mở và chưa có giải pháp thỏa đáng. Hơn nữa, một trong những vấn đề khó nhất của kiểm thử tự động đó là việc sinh các giá trị đầu ra mong đợi tương ứng với các đầu vào của các ca kiểm thử. Đây là một bài toán khó và chưa có giải pháp hiệu quả nhằm giải quyết vấn đề này.

Giải pháp chủ yếu để giải quyết các vấn đề trên là đề xuất các phương pháp và công cụ hỗ trợ tối đa các hoạt động trong quy trình kiểm thử phần mềm. Trong quy trình kiểm thử, chúng ta cần một số công cụ ứng với các pha và các mục tiêu kiểm thử khác nhau. Ví dụ, để tự động hóa chiến lược kiểm thử hộp đen, phương pháp kiểm thử dựa trên mô hình (model-based testing) đang được biết đến như là một giải pháp tin cậy và hiệu quả [MRA04, KJ02, BFM04]. Với kiểm thử hộp trắng, ứng với mỗi phương pháp khác nhau chúng ta cũng có rất nhiều công cụ hỗ trợ. Ngoài ra, một số công cụ hỗ trợ kiểm thử các tính chất phi chức năng như độ an toàn, bảo mật, hiệu năng và khả năng chịu tải, v.v., cũng đã được phát triển và sử dụng rộng rãi.

Kiểm thử tự động là quá trình thực hiện một cách tự động các bước trong một kịch bản kiểm thử. Kiểm thử tự động bằng một

công cụ nhằm rút ngắn thời gian kiểm thử. Mục đích của kiểm thử tự động là giảm thiểu thời gian, công sức và kinh phí trong khi vẫn tăng độ tin cậy, tăng tính hiệu quả và giảm sự nhàm chán cho người kiểm thử trong quá trình kiểm thử sản phẩm phần mềm. Kiểm thử tự động sẽ được sử dụng khi dự án không đủ tài nguyên (thời gian, nhân lực và chi phí), phải thực hiện kiểm thử hồi quy khi sản phẩm được sửa đổi hoặc nâng cấp và cần kiểm thử lại các tính năng đã thực hiện tốt trước đó, kiểm tra khả năng vận hành của sản phẩm trong các môi trường đặc biệt (đo tốc độ xử lý trung bình ứng với mỗi yêu cầu, xác định khả năng chịu tải tối đa, xác định cấu hình tối thiểu để thực thi hệ thống, kiểm tra các cơ chế an ninh và an toàn, v.v.).

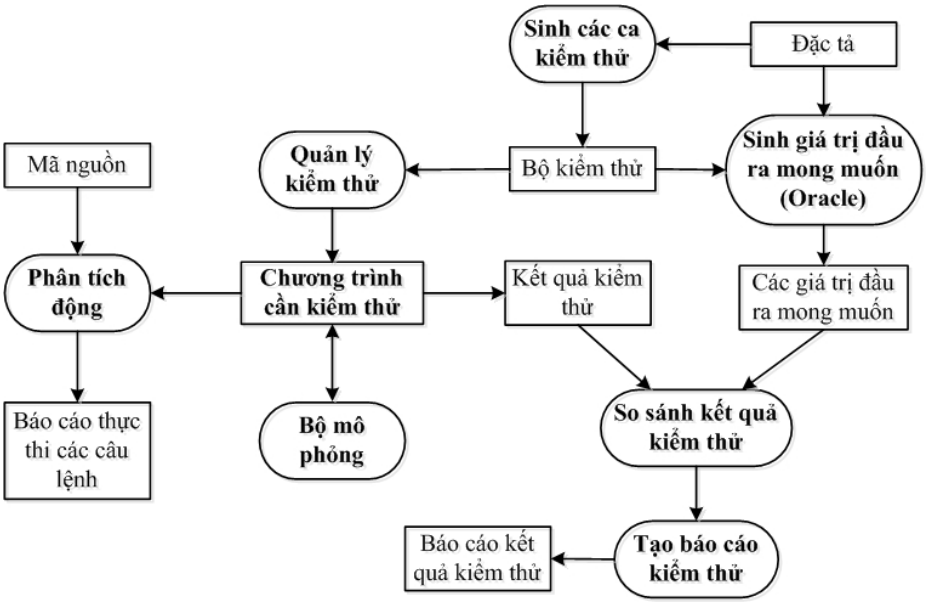
9.2 Kiến trúc của một bộ công cụ kiểm thử tự động

Trong thức tế, có rất nhiều bộ công cụ hỗ trợ kiểm thử tự động được phát triển nhằm góp phần giải quyết các vấn đề khó khăn của quy trình kiểm thử. Hình 9.1 mô tả kiến trúc chung nhất của một bộ kiểm thử tự động [Som10]. Trong kiến trúc này, các công cụ kiểm thử được tích hợp trong một quy trình thống nhất nhằm hỗ trợ đầy đủ các hoạt động kiểm thử trong quy trình kiểm thử các sản phẩm phần mềm.

Các công cụ cơ bản trong kiến trúc này bao gồm:

- **Quản lý kiểm thử:** công cụ này cho phép quản lý việc thực hiện/thực thi các ca kiểm thử. Nó giám sát việc thực hiện từng ca kiểm thử ứng với bộ giá trị đầu vào, giá trị đầu ra mong muốn và giá trị đầu ra thực tế. JUnit là một ví dụ điển hình về công cụ này.

- **Sinh các ca kiểm thử:** Đây là một trong những công cụ quan trọng nhất của các bộ kiểm thử tự động. Tùy thuộc vào



Hình 9.1: Kiến trúc chung của một bộ kiểm thử tự động.

các kỹ thuật kiểm thử được áp dụng, công cụ này sẽ sinh ra tập các ca kiểm thử (chưa gồm giá trị đầu ra mong muốn) cho chương trình/đơn vị chương trình cần kiểm thử. Các ca kiểm thử được sinh ra chỉ chứa giá trị đầu vào để thực hiện nó. Các giá trị này có thể được lựa chọn trong cơ sở dữ liệu hoặc được sinh một cách ngẫu nhiên.

- **Sinh giá trị đầu ra mong muốn:** Các ca kiểm thử được sinh ra bởi công cụ trên chỉ chứa các giá trị ứng với các biến đầu vào. Công cụ này cho phép sinh ra giá trị đầu ra mong muốn ứng với mỗi bộ dữ liệu đầu vào của mỗi ca kiểm thử. Giá trị đầu ra mong muốn này sẽ được sử dụng để so sánh với giá trị đầu ra thực tế khi thực hiện ca kiểm thử này nhằm phát hiện ra các lỗi/khiếm khuyết của sản phẩm.

- **So sánh kết quả kiểm thử:** Công cụ này so sánh giá trị đầu ra thực tế và giá trị đầu ra mong muốn của mỗi ca kiểm

thử khi nó được thực hiện trên chương trình/đơn vị chương trình cần kiểm thử. Đối với mục đích kiểm thử phi chức năng, chúng ta không thể sử dụng cách làm này. Các giải pháp cho bài toán này sẽ được trình bày chi tiết trong chương 10.

- **Tạo báo cáo kiểm thử:** Một trong những ưu điểm của các bộ công cụ kiểm thử tự động là nó có cơ chế sinh báo cáo kiểm thử một cách chính xác và nhất quán. Dựa vào kết quả của công cụ so sánh kết quả kiểm thử, công cụ này sẽ tự động sinh ra báo cáo kết quả kiểm thử theo định dạng mong muốn của đơn vị phát triển.

- **Phân tích động:** Công cụ này cung cấp một cơ chế nhằm kiểm tra việc thực hiện của các câu lệnh của chương trình cần kiểm thử nhằm phát hiện ra các lỗi và phát hiện các câu lệnh/đoạn lệnh không được thực hiện bởi một tập các ca kiểm thử cho trước. Công cụ này cũng rất hiệu quả trong việc đánh giá tính hiệu quả của một bộ kiểm thử cho trước.

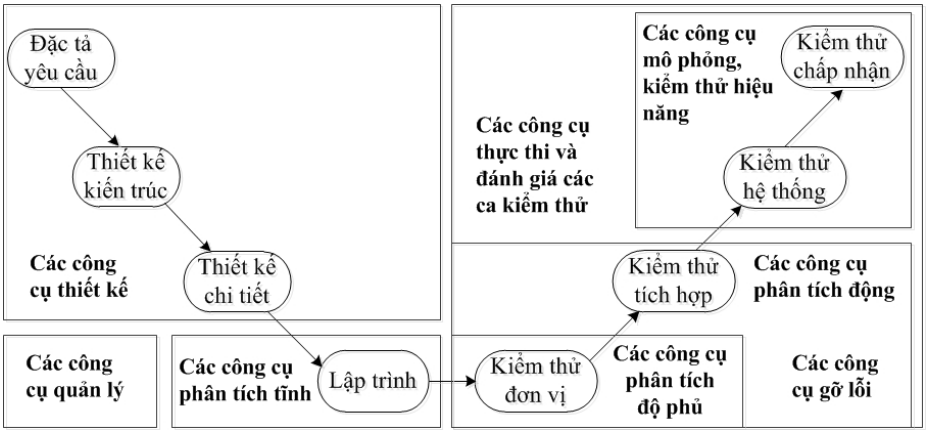
- **Bộ mô phỏng:** Có nhiều loại mình mô phỏng được cung cấp trong các bộ kiểm thử tự động. Mục đích của các công cụ này là mô phỏng quá trình thực hiện của chương trình cần kiểm thử. Ví dụ, các công cụ mô phỏng giao diện người dùng cho phép thực hiện tự động các tương tác giữa người dùng và sản phẩm. Selenium[1] là một ví dụ về một công cụ mô phỏng giao diện người dùng cho các ứng dụng Web.

Trong thực tế, các bộ công cụ kiểm thử tự động có thể có thêm một số công cụ khác như cho phép đặc tả các tính chất của hệ thống cần kiểm thử, vân vân. Một số bộ công cụ chỉ hỗ trợ một số công cụ trong các công cụ đã liệt kê ở trên.

[1] www.seleniumhq.org

Các công cụ hỗ trợ kiểm thử tự động rất đa dạng và phục vụ nhiều mục đích khác nhau. Hình 9.2 thể hiện các loại công cụ kiểm thử tự động ứng với từng pha trong quy trình phát triển phần mềm.

- **Các công cụ quản lý:** các công cụ này phục vụ pha lập trình và quản lý quá trình này như quản lý phiên bản, quản lý thay đổi, v.v.
- **Các công cụ phân tích tĩnh:** các công cụ này phục vụ pha lập trình và cho phép phân tích mã nguồn để tìm các lỗi hay gặp.
- **Các công cụ phân tích độ phủ:** các công cụ này hỗ trợ pha kiểm thử đơn vị và cho phép phân tích độ bao phủ của một bộ kiểm thử đối với mã nguồn.
- **Các công cụ gỡ lỗi:** các công cụ này cho phép định vị các lỗi được phát hiện bởi một ca kiểm thử.
- **Các công cụ phân tích động:** các công cụ này hỗ trợ cả pha kiểm thử đơn vị và kiểm thử tích hợp. Chúng cho phép sinh ra các ca kiểm thử từ mã nguồn và thực thi chúng nhằm phát hiện các lỗi lập trình.
- **Các công cụ mô phỏng, kiểm thử hiệu năng:** các công cụ này trợ giúp kiểm thử tự động hệ thống và kiểm thử chấp nhận. Chúng cho phép kiểm thử một số yêu cầu về hiệu năng của hệ thống như tính hiệu quả trong sử dụng tài nguyên, tính an toàn và bảo mật, khả năng chịu tải, v.v.
- **Các công cụ thực thi và đánh giá các ca kiểm thử:** các công cụ này cho phép thực hiện/thực thi các ca kiểm thử và giám sát việc thực hiện từng ca kiểm thử ứng với bộ giá trị đầu vào, giá trị đầu ra mong muốn và giá trị đầu ra thực tế nhằm tạo báo cáo kiểm thử.



Hình 9.2: Các công cụ kiểm thử tự động trong phát triển phần mềm.

9.3 Ưu nhược điểm của kiểm thử tự động

Một trong những lý do chính dẫn đến quyết định sử dụng kiểm thử tự động là chúng ta thường xuyên phải tiến hành kiểm thử hồi quy khi một hoặc một số chức năng của hệ thống bị thay đổi. Từ khi nhóm phát triển đưa ra một phiên bản đầu tiên của sản phẩm cho tới khi phiên bản này tới tay khách hàng chỉ trong một thời gian ngắn. Trong thời gian ngắn ngủi này, kiểm thử hồi quy sẽ được thực hiện rất nhiều lần. Điều này có nghĩa là một số lượng ca kiểm thử lớn phải được thực thi lặp đi lặp lại trong một khoảng thời gian ngắn. Đây là lúc lý tưởng để sử dụng kiểm thử tự động nhằm tăng độ chính xác và rút ngắn thời gian và công sức của quá trình kiểm thử sản phẩm. Hơn nữa, khi chúng ta thêm một chức năng mới của hệ thống, các công cụ kiểm thử tự động sẽ phát huy tác dụng trong việc đảm bảo tính đúng đắn (sinh và thực hiện các ca kiểm thử) của chức năng này trong một thời gian ngắn. Công việc này cũng thường xuyên được thực hiện trong quá trình kiểm thử hồi quy. Thêm nữa, trước khi đóng gói và chuyển giao sản phẩm, chúng ta cần kiểm thử các yêu cầu phi chức năng của hệ thống như

hiệu năng, an toàn và an ninh, khả năng chịu tải, v.v. Trong trường hợp này, các công cụ kiểm thử tự động gần như là sự lựa chọn duy nhất. Tóm lại, kiểm thử tự động có nhiều ưu điểm đặc biệt là trong quá trình kiểm thử hồi quy. Dưới đây là một số ưu điểm chính của kiểm thử tự động.

- **Độ tin cậy cao (Reliability):** Nhờ sự ổn định vượt trội của công cụ kiểm thử tự động so với các hoạt động thủ công của người kiểm thử, đặc biệt trong trường hợp có quá nhiều ca kiểm thử cần được thực thi, nên độ tin cậy của kiểm thử tự động thường cao hơn so với kiểm thử thủ công.

- **Khả năng lặp (Repeatability):** Khi chúng ta phải thực thi một ca kiểm thử với 50 bộ dữ liệu đầu vào khác nhau, nếu thực thi cách thủ công, ngồi trước màn hình, nhập dữ liệu và kiểm tra trong 50 lần có lẽ chúng ta sẽ gục ngã sớm trên bàn làm việc của mình. Tuy nhiên, với việc sử dụng các công cụ kiểm thử tự động, chúng ta chỉ chỉ cần nhập dữ liệu vào một tệp Excel hoặc một kịch bản (script) rồi cho công cụ thực hiện và ngồi nghỉ ngơi cho tới khi nhận được báo cáo kiểm thử từ công cụ này. Với độ ổn định cao, chúng ta hoàn toàn có thể tin tưởng vào kết quả thực thi của công cụ kiểm thử tự động. Hơn nữa, khi một ca kiểm thử phát hiện ra lỗi và chúng ta phải sửa nó, chúng ta cần thực hiện lại tất cả các ca kiểm thử từ đầu. Trong trường hợp này, các công cụ kiểm thử tự động sẽ hoàn toàn trợ giúp chúng ta công việc nặng nhọc và nhàm chán này.

- **Khả năng tái sử dụng (Reusability):** Với một bộ kiểm thử tự động, chúng ta có thể sử dụng cho nhiều phiên bản ứng dụng khác nhau, đây được gọi là tính tái sử dụng.

- **Nhanh (Fast):** Đây là điều không cần phải bàn cãi, nếu cần năm phút để thực thi một ca kiểm thử cách thủ công, có thể

bạn cần chưa đầy 30s để thực thi cách tự động.

- **Chi phí thấp (Cost Reduction):** Nếu áp dụng kiểm thử tự động đúng cách, chúng ta có thể tiết kiệm được rất nhiều chi phí, thời gian và nhân lực.

Bên cạnh những ưu điểm trên, các công cụ kiểm thử tự động cũng có một số nhược điểm chính sau.

- **Khó mở rộng, khó bảo trì (Poor scalability and maintainability):** Trong cùng một dự án, để mở rộng phạm vi cho kiểm thử tự động là khó hơn nhiều so với kiểm thử cách thủ công. Số lượng công việc phải làm để mở rộng phạm vi cho kiểm thử tự động là nhiều hơn và khó hơn kiểm thử thủ công. Tương tự, để cập nhật một ca kiểm thử một cách thủ công, chúng ta chỉ cần mở nó ra và cập nhật một cách đơn giản. Tuy nhiên, kiểm thử tự động lại không đơn giản như vậy, cập nhật hay chỉnh sửa một ca kiểm thử yêu cầu rất nhiều công việc như gỡ lỗi, thay đổi dữ liệu đầu vào và cập nhật mã nguồn mới.

- **Khả năng bao phủ thấp (Low coverage):** Khi áp dụng cho kiểm thử đơn vị, khả năng bao phủ của các công cụ kiểm thử tự động thường khá cao. Tuy nhiên, xét trên góc nhìn toàn dự án, chính vì việc khó ứng dụng, khó mở rộng, cũng như đòi hỏi quá nhiều kỹ năng lập trình nên độ bao phủ của kiểm thử tự động khá thấp.

- **Vấn đề công cụ và nhân lực (Technology vs. people issues):** Cho đến nay công cụ hỗ trợ kiểm thử tự động đã có những bước phát triển mạnh mẽ, chúng ta có các công cụ rất tốt như QTP, Selenium, Test Complete, LoadTest, Jmeter, Visual Studio, v.v. Tuy nhiên, chưa có một bộ công cụ đủ tốt để đáp ứng các yêu cầu kiểm thử và đảm bảo chất lượng hiện

nay. Hơn nữa, một số công ty cần một số công cụ kiểm thử đặc thù và vì vậy họ gặp khó khăn trong việc đầu tư phát triển các công cụ này. Ngoài ra, nguồn nhân lực đạt yêu cầu để áp dụng các công cụ kiểm thử tự động cũng còn nhiều hạn chế.

9.4 Một số công cụ kiểm thử tự động

9.4.1 JUnit

Công cụ kiểm thử cho các đơn vị chương trình viết bằng Java, JUnit[1], cung cấp một cơ sở hạ tầng chuẩn cho việc thiết lập các bộ kiểm thử. Một khi bộ kiểm thử được thiết lập, nó có thể tự động chạy mỗi khi mã thay đổi. JUnit khuyến khích các nhà phát triển viết các kịch bản kiểm thử, chèn các mã kiểm thử vào mã nguồn Java và thực hiện chúng để phát hiện các lỗi bên trong đơn vị chương trình. Khác với các công cụ khác, JUnit không hỗ trợ cơ chế sinh các ca kiểm thử. Trong JUnit có các ca kiểm thử là các lớp của Java, các lớp này bao gồm một hay nhiều phương thức cần kiểm thử và các ca kiểm thử này lại được nhóm với nhau để tạo thành bộ kiểm thử tương ứng.

Mỗi phép thử trong JUnit là một phương thức public. Phương thức này được đặt tên dưới dạng Test(testXYZ()) (không có tham số). Nếu chúng ta không tuân thủ theo quy tắc đặt tên này thì JUnit sẽ không xác định được phương thức kiểm thử một cách tự động. Các phương thức cơ bản trong JUnit bao gồm:

- Boolean assertEquals(): So sánh hai giá trị để kiểm tra liệu chúng có bằng nhau hay không. Phép thử thất bại nếu hai giá trị không bằng nhau.

[1]http://junit.org/

- Boolean assertFalse(): Đánh giá biểu thức lôgic. Phép thử thất bại nếu biểu thức đúng.
- Boolean assertNotNull(): So sánh tham chiếu của một đối tượng với Null. Phép thử thất bại nếu tham chiếu đối tượng Null.
- Boolean assertNotSame(): So sánh địa chỉ vùng nhớ của hai tham chiếu hai đối tượng bằng cách sử dụng toán tử "==". Phép thử thất bại trả về nếu cả hai đều tham chiếu đến cùng một đối tượng.
- Boolean assertNull(): So sánh tham chiếu của một đối tượng với giá trị Null. Phép thử thất bại nếu đối tượng không là Null.
- Boolean assertTrue(): Đánh giá một biểu thức logic. Phép thử thất bại nếu biểu thức này sai.
- Void fail(): Phương thức này làm cho test hiện tại thất bại, phương thức này thường được sử dụng khi xử lý các ngoại lệ.
- Setup() và Teardown(): Hai phương thức này là một phần của lớp junit.framework.TestCase. Khi sử dụng hai phương thức này sẽ giúp chúng ta tránh được việc trùng mã khi nhiều test cùng chia sẻ nhau ở phần khởi tạo và dọn dẹp các biến.

Hiện nay, JUnit đã được tích hợp trong Eclipse và hỗ trợ rất đắc lực cho quá trình kiểm thử.

9.4.2 NUnit

NUnit[1] là một bộ công cụ hỗ trợ kiểm thử tự động miễn phí được sử dụng khá rộng rãi trong kiểm thử đơn vị đối với ngôn ngữ .Net.

[1]http://www.nunit.org

Ban đầu, NUnit được chuyển từ JUnit. Phiên bản mới nhất của NUnit đang được sử dụng hiện nay là phiên bản 2.6. NUnit được viết hoàn toàn bằng C# và đã được hoàn toàn thiết kế lại để tận dụng lợi thế của nhiều người. Hiện nay, NUnit đã được tích hợp trong các phiên bản Visual Studio của Microsoft và hỗ trợ rất đắc lực cho quá trình kiểm thử.

9.4.3 QuickTest Professional

Quick Test Professional[1] là phần mềm kiểm soát việc kiểm thử tự động các chức năng của các sản phẩm phần mềm cần kiểm thử. Sản phẩm này bao gồm một tập các mô-đun có thể tương tác với nhau nhằm quản lý toàn bộ quy trình kiểm thử phần mềm. Quick Test Professional là một công cụ hỗ trợ cách tiếp cận kiểm thử chức năng (kiểm thử hộp đen) và cho phép tiến hành kiểm thử hồi quy một cách tự động.

9.4.4 Apache JMeter

Apache JMeter[2] được dùng để kiểm thử khả năng chịu tải và kiểm thử hiệu năng cho các ứng dụng Web và một số ứng dụng khác. Công cụ này hỗ trợ kiểm thử hiệu năng của các mã nguồn được viết bằng các ngôn ngữ khác nhau như PHP, Java, ASP.NET, v.v. Apache JMeter mô phỏng khả năng chịu tải của các máy chủ trên máy sử dụng để kiểm thử hệ thống. Công cụ này hỗ trợ giao diện đồ họa giúp phân tích tốt hiệu suất khi kiểm thử đồng thời nhiều ca kiểm thử. Ngoài ra, Apache JMeter còn hỗ trợ thêm nhiều tiện ích khác. Các tiện ích này được cung cấp tại địa chỉ [3].

[1] http://www.automation-consultants.com
[2] http://jmeter.apache.org/
[3] http://jmeter-plugins.org

9.4.5 Load Runner

Load Runner[1] giả lập một môi trường ảo gồm nhiều người dùng thực hiện các giao dịch cùng một lúc nhằm giám sát các thông số xử lý của phần mềm cần kiểm thử. Kết quả thống kê sẽ được lưu lại và cho phép kiểm thử viên thực hiện phân tích nhằm kiểm thử khả năng chịu tải và các yêu cầu phi chức năng khác của sản phẩm.

Trong quá trình kiểm thử, Load Runner tự động tạo ra các kịch bản kiểm thử để lưu lại các thao tác người dùng tương tác lên phần mềm. Mỗi kịch bản này còn được xem là hoạt động của một người dùng ảo mà Load Runner giả lập. Ngoài ra, công cụ này còn cho phép tổ chức, điều chỉnh, quản lý và giám sát hoạt động kiểm tra khả năng chịu tải.

9.4.6 Cucumber

Cucumber[2] là một công cụ kiểm thử tự động dựa trên việc thực thi các chức năng được mô tả dưới dạng các kịch bản kiểm thử. Các kịch bản này cho phép người phát triển đặc tả các hành vi của chức năng cần kiểm thử (Behavior Driven Development). Điều này có nghĩa rằng các kịch bản kiểm thử sẽ được viết trước và sau đó mã nguồn mới được cài đặt. Sau khi cài đặt xong, công cụ này cho phép kiểm tra liệu việc cài đặt có đúng với các hành vi đã được đặc tả trong các kịch bản hay chưa. Ngôn ngữ được Cucumber sử dụng để đặc tả các kịch bản có tên là "Gherkin". Gherkin là một ngôn ngữ thể hiện nghiệp vụ và có miền ngữ nghĩa xác định giúp cho người đọc có thể hiểu được kịch bản và hành động mà không cần biết chi tiết chúng được cài đặt như thế nào. Cucumber được viết bằng ngôn ngữ Ruby nhưng chúng ta có thể sử dụng công cụ này để kiểm thử các sản phẩm được viết bằng Ruby, Java, C# và Python.

[1] http://www.hp.com/LoadRunner/
[2] http://cukes.info

9.4.7 CFT4CUnit

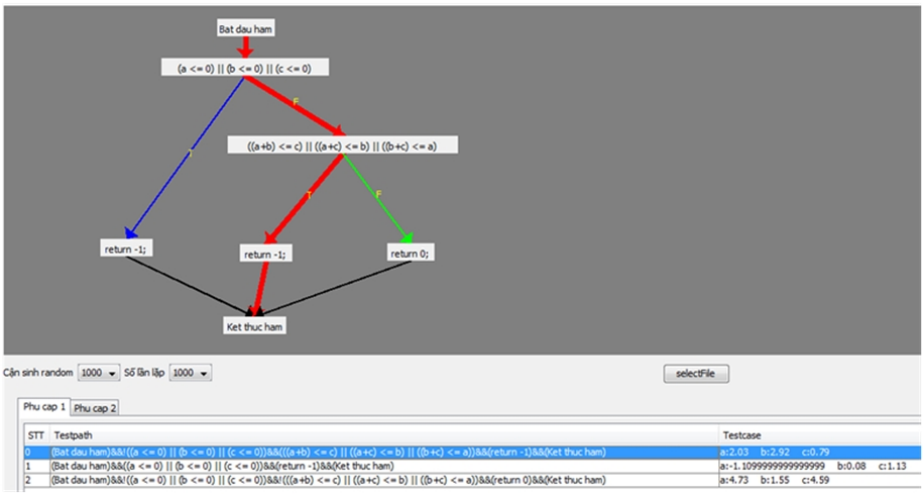
Nhằm cung cấp một công cụ kiểm thử tự động các đơn vị chương trình (các hàm) viết bằng ngôn ngữ C phục vụ các sinh viên trong việc nghiên cứu và học tập, chúng tôi đã phát triển một công cụ có tên CFT4CUnit (Control Flow Testing for C Unit). Công cụ này tự động hóa các bước trong quy trình kiểm thử dòng điều khiển như đã giới thiệu trong Chương 6. Đầu vào của công cụ này là các hàm/đơn vị chương trình viết bằng ngôn ngữ C và độ đo cần kiểm thử (như đã giới thiệu trong mục 6.3 của Chương 6). Công cụ sẽ xây dựng đồ thị dòng dữ liệu ứng với độ đo này, hiển thị đồ thị luồng điều khiển một cách trực quan và sinh ra các ca kiểm thử tương ứng. Các ca kiểm thử được sinh ra sẽ được xuất ra một tệp nhằm giúp cho kiểm thử viên thêm giá trị đầu ra mong muốn vào mỗi ca kiểm thử. Khi kiểm thử viên làm việc với mỗi ca kiểm thử, công cụ cho phép làm nổi bật dòng điều khiển của đơn vị chương trình ứng với ca kiểm thử này nhằm trợ giúp trong việc sinh giá trị đầu ra mong muốn một cách chính xác. Cuối cùng, công cụ cho phép thực hiện các ca kiểm thử và tạo ra báo cáo kiểm thử.

Đoạn mã 9.1: Hàm `IsTrangle` làm đầu vào cho công cụ CFT4CUnit

```
//tra lai 1 neu a, b, c la do dai 3 canh
//cua mot tam giac. Tra lai -1 neu nguoc lai
int IsTrangle(float a, float b, float c){
        if(a <= 0 || b <= 0 || c <= 0)
                return -1;
        if(a + b <= c || b + c <= a || a + c <= b)
                return -1;
        return 1;
}
```

Đoạn mã 9.1 là mã nguồn của hàm `IsTrangle` bằng ngôn ngữ C. Hàm này sẽ được cung cấp làm đầu vào cho công cụ CFT4CUnit. Đồ thị dòng điều khiển và các ca kiểm thử của Hàm `IsTrangle` sinh

bởi công cụ CFT4CUnit như giao diện trong hình 9.3. Công cụ này cùng tài liệu hướng dẫn sử dụng và các ví dụ áp dụng được cung cấp tại địa chỉ[1]. Chúng tôi cũng đã cung cấp mã nguồn của công cụ này nhằm cho phép các sinh viên có thể mở rộng công cụ này phục vụ các mục đích học tập và nghiên cứu.



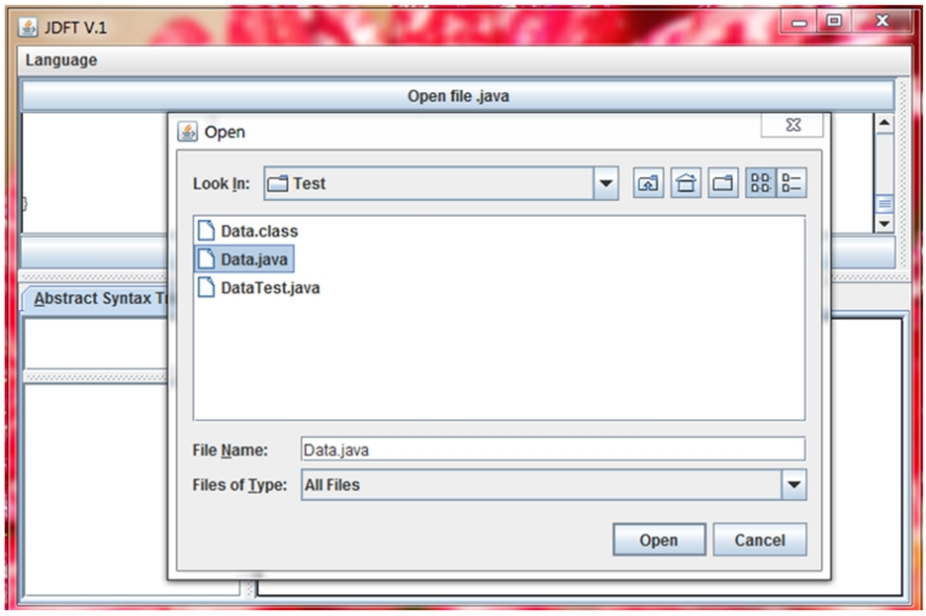
Hình 9.3: Giao diện của công cụ CFT4CUnit.

9.4.8 JDFT

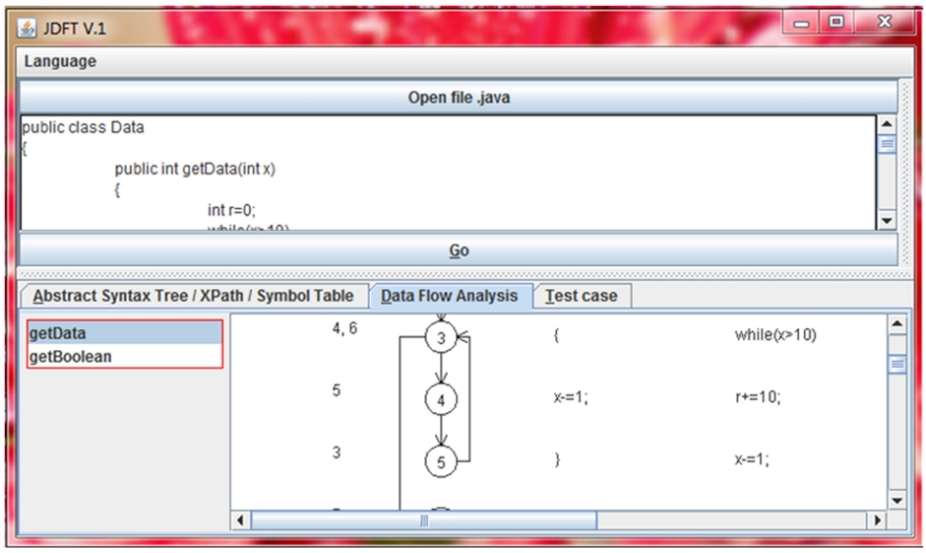
Với mục tiêu như công cụ CFT4CUnit, chúng tôi đã phát triển công cụ có tên JDFT (Data Flow Testing for Java Programs) nhằm tự động hóa phương pháp kiểm thử dòng dữ liệu như đã giới thiệu trong chương 7. Đầu vào của công cụ này là các tệp .java tương ứng với mã nguồn của một lớp đối tượng. Công cụ cho phép chọn tệp đầu vào như giao diện trong hình 9.4.

Sau khi chọn tệp đầu vào, JDFT sẽ tự động xây dựng đồ thị dòng dữ liệu và hiển thị nó cùng mã nguồn tương ứng như giao diện trong hình 9.5.

[1]http://uet.vnu.edu.vn/~hungpn/CFT4CUnit/



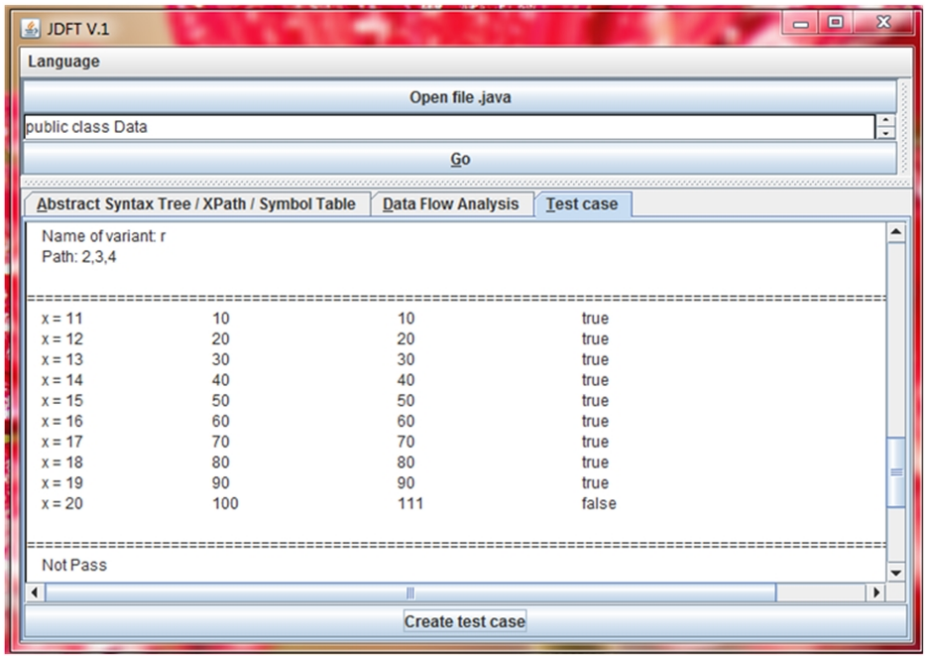
Hình 9.4: Giao diện cho phép chọn tệp mã nguồn .java cần kiểm thử.



Hình 9.5: Giao diện hiển thị mã nguồn và đồ thị dòng điều khiển.

Cuối cùng, công cụ sẽ sinh các ca kiểm thử cùng đầu ra mong muốn ứng với từng ca kiểm thử được sinh ra. Các ca kiểm thử này

sẽ được lưu lại dưới một tệp Excel nhằm sử dụng lại trong tương lai. JDFT sẽ phân tích và sinh báo cáo kiểm thử như hình 9.6.



Hình 9.6: Báo cáo kiểm thử được sinh bởi công cụ JDFT.

Công cụ này cùng tài liệu hướng dẫn sử dụng và các ví dụ áp dụng được cung cấp tại địa chỉ[1]. Chúng tôi cũng đã cung cấp mã nguồn của công cụ này nhằm cho phép các sinh viên có thể mở rộng công cụ này phục vụ các mục đích học tập và nghiên cứu.

9.5 Tổng kết

Kiểm thử tự động đang được quan tâm như là một giải pháp hiệu quả và duy nhất nhằm cải thiện tính chính xác và hiệu quả cũng như giảm kinh phí và rút ngắn thời gian trong quá trình kiểm thử các sản phẩm phần mềm. Đã có nhiều công cụ được phát triển hỗ trợ các mục đích trên. Tùy thuộc vào yêu cầu kiểm thử của từng

[1]http://uet.vnu.edu.vn/~hungpn/JDFT/

sản phẩm, các công ty sẽ lựa chọn các công cụ phù hợp. Tuy nhiên, rất khó để tìm được một bộ công cụ đáp ứng tất cả các yêu cầu kiểm thử. Các công cụ hỗ trợ kiểm thử tự động hiện nay thường chú trọng vào kiểm thử các yêu cầu phi chức năng hoặc thực thi các ca kiểm thử. Chưa có một công cụ đủ tốt để sinh tự động các ca kiểm thử trong khi đây chính là vấn đề quan trọng nhất quyết định đến chất lượng kiểm thử các sản phẩm phần mềm. Trong nhiều trường hợp, các công ty cần chủ động mở rộng và phát triển thêm các công cụ phục vụ các mục đích cụ thể. Vấn đề đào tạo nhân lực cho kiểm thử tự động cũng cần được quan tâm đặc biệt.

9.6 Bài tập

1. Trình bày khái niệm kiểm thử tự động.
2. Lợi ích của kiểm thử tự động? Khi nào chúng ta áp dụng kiểm thử tự động?
3. Những khó khăn khi áp dụng kiểm thử tự động?
4. Mô tả kiến trúc của bộ công cụ kiểm thử tự động. Lấy ví dụ minh họa ứng với mỗi công cụ trong kiến trúc này.
5. Sử dụng một trong các công cụ kiểm thử tự động (đã giới thiệu trong chương này hoặc các công cụ khác) để kiểm thử cho một hệ thống đơn giản.

Chương 10

Kiểm thử tích hợp, hệ thống, chấp nhận và hồi quy

Trong chương này, chúng ta sẽ xem xét các hoạt động kiểm thử ở trên mức kiểm thử đơn vị bao gồm: kiểm thử tích hợp, kiểm thử hệ thống, kiểm thử chấp nhận và kiểm thử hồi quy. Kiểm thử tích hợp có sự khác biệt với ba loại kiểm thử còn lại ở điểm lúc này hệ thống phần mềm chưa hoàn chỉnh mà còn dở dang, mới ghép một phần các mô-đun của hệ thống.

10.1 Tổng quan

Ba loại kiểm thử còn lại (kiểm thử hệ thống, kiểm thử chấp nhận và kiểm thử hồi quy) xem xét việc kiểm thử một hệ thống phần mềm hoàn chỉnh sau khi đã tích hợp tất cả các mô-đun và các thành phần của hệ thống. Trong trường hợp này, các lỗi về tích hợp đã được phát hiện và sửa hết. Ngoài việc xem xét việc kiểm tra các chức năng của hệ thống phần mềm đã đúng và đủ theo đặc tả yêu cầu chưa, chúng ta sẽ thảo luận về việc kiểm tra chất lượng hệ

thống. Sau đó chúng ta sẽ thảo luận khái niệm kiểm thử chấp nhận hoặc chấp thuận, là kiểm tra xem hệ thống đã đúng yêu cầu của người dùng chưa. Cuối cùng chúng ta sẽ bàn về các vấn đề kiểm thử khi hệ thống phần mềm thay đổi, tiến hóa theo thời gian, còn gọi là kiểm thử hồi quy.

Kiểm thử hệ thống thường do đội phát triển chịu trách nhiệm thực hiện và hệ thống được kiểm tra so với tài liệu đặc tả của hệ thống. Do chỉ đối chiếu với tài liệu đặc tả nên đây là công việc kiểm chứng (verification). Kiểm thử chấp nhận là hoạt động thẩm định (validation) và do người dùng, khách hàng kiểm tra xem hệ thống có đúng như họ mong muốn không, có đáp ứng yêu cầu thực tế công việc của họ không. Người dùng thường bỏ qua các tài liệu đặc tả kỹ thuật khi thẩm định phần mềm. Chúng ta cần kiểm thử chấp nhận vì nhiều lý do, ví dụ đặc tả có thể đã có sai sót trong quá trình xây dựng, hoặc người khách hàng và người phát triển cùng đọc một tài liệu nhưng hiểu nó theo các nghĩa khác nhau. Lý do chính là các tài liệu đặc tả này thường được viết bằng ngôn ngữ tự nhiên, có thể có nhập nhằng, không rõ nghĩa hoặc do kiến thức của người đọc khác nhau.

Khi phần mềm đã đưa vào sử dụng, đôi khi nó vẫn được sửa đổi, nâng cấp tiếp vì có thể có lỗi phát hiện trong quá trình sử dụng, hoặc người dùng yêu cầu thay đổi, thêm bớt một số chức năng của sản phẩm phần mềm. Lúc này chúng ta sử dụng kiểm thử hệ thống và kiểm thử chấp nhận theo cách thông thường thì sẽ rất tốn công sức và thời gian. Chúng ta đều mong muốn chỉ kiểm thử lại những chức năng bị thay đổi, bổ sụng. Nhưng việc sửa đổi một chức năng có thể gây ảnh hưởng tới các chức năng khác, nên chúng ta phải kiểm thử lại cả các chức năng bị ảnh hưởng khác. Hoạt động kiểm thử lại này được gọi là kiểm thử hồi quy.

Các kỹ thuật trong kiểm thử hồi quy giúp chúng ta xác định các ca kiểm thử cần thực hiện lại để chỉ kiểm tra hệ thống ở những phần có liên quan, thay vì phải thực hiện lại toàn bộ hai công việc

này. Bảng 10.1 so sánh các khái niệm kiểm thử hệ thống, kiểm thử chấp nhận và kiểm thử hồi quy.

Bảng 10.1: Kiểm thử hệ thống, kiểm thử chấp nhận và hồi quy

Kiểm thử hệ thống	Kiểm thử chấp nhận	Kiểm thử hồi quy
Kiểm tra so với đặc tả yêu cầu	Kiểm tra so với mong muốn của người dùng	Kiểm tra lại các ca kiểm thử đã thành công trước đó
Do đội phát triển thực hiện	Do đội phát triển và người dùng cùng thực hiện	Do đội phát triển thực hiện
Kiểm tra sản phẩm đạt yêu cầu kỹ thuật chưa	Khẳng định tính đúng đắn và đầy đủ của sản phẩm	Giúp những thay đổi không mong muốn không xảy ra

10.2 Kiểm thử tích hợp

Các chương trước chủ yếu tập trung vào các kỹ thuật kiểm thử các hàm đơn lẻ, còn gọi là kiểm thử đơn vị. Kiểm thử tích hợp tập trung vào việc kiểm thử khi ghép nối các đơn vị này, hay tổng quát hơn là các mô-đun đã được kiểm thử đơn vị. Kiểm thử ở mức này gọi là kiểm thử tích hợp. Kiểm thử tích hợp giúp kiểm tra sự tương thích giữa các mô-đun. Kiểm thử hệ thống và kiểm thử chấp thuận ở phần sau sẽ kiểm tra toàn bộ hệ thống so với đặc tả và yêu cầu của người sử dụng.

Một mô-đun phần mềm (hay còn gọi là một thành phần) là một phần tử tương đối độc lập trong một hệ thống. Khái niệm mô-đun mang tính tương đối. Mô-đun có thể đơn giản là một hàm, một thủ tục, một lớp, hay một tập các phần tử cơ bản này kết hợp với nhau để cung cấp một dịch vụ tích hợp mới. Các mô-đun thường có giao

diện rõ ràng để giao tiếp với các mô-đun khác. Một hệ thống là một tập các mô-đun kết nối với nhau theo một cách nhất định để thực hiện một mục đích đã đặt ra.

Trong các dự án lớn có hàng chục hoặc hàng trăm người lập trình, hệ thống thường được chia thành các mô-đun để nhiều nhóm cùng phát triển. Các mô-đun thường được kiểm thử độc lập và kiểm thử ở mức này gọi là kiểm thử đơn vị. Người lập trình thường chịu trách nhiệm thực hiện kiểm thử đơn vị. Các kỹ thuật kiểm thử hộp đen (kiểm thử chức năng) và kiểm thử hộp trắng (kiểm thử cấu trúc) chủ yếu để kiểm thử ở mức này. Sau khi các đơn vị được kiểm thử xong, chúng được ghép với nhau để tạo thành mô-đun lớn hơn, hay hệ thống con, hoặc thành hệ thống phần mềm tùy độ lớn và phức tạp của hệ thống. Việc ghép này không hề đơn giản vì lúc này các lỗi ở mức giao diện giữa các mô-đun có thể xảy ra. Việc kiểm tra lỗi trong quá trình ghép này là kiểm thử tích hợp và kiểm thử hệ thống. Kiểm thử tích hợp thường phải được thực hiện trước và làm trong nội bộ đội phát triển. Khi kiểm thử tích hợp đã ổn định, kiểm thử hệ thống mới được tiến hành để đảm bảo hệ thống hoạt động tốt ở môi trường thật.

Ba lý do chính chúng ta cần kiểm thử tích hợp là:

- Các mô-đun có thể do các nhóm khác nhau làm. Dù đã có thống nhất với nhau từ trước về giao diện của các mô-đun, việc hiểu sai, nhầm lẫn, và chủ quan nhiều khi vẫn xảy ra trên thực tế. Phần sau chúng ta sẽ xem những nguyên nhân có thể gây ra lỗi giao diện này.

- Các mô-đun thường được kiểm thử với các hàm giả trước khi tích hợp, hoặc là với hàm giả (stub), hoặc hàm giả gọi (driver). Các hàm giả (stub) chỉ trả về giá trị kết quả với một số tham số định trước, mô phỏng một vài trường hợp của hàm thật. Các hàm giả gọi (driver) gọi nhiều mô-đun khác theo

các đường đi khác nhau nên nếu hàm giả gọi này không được kiểm thử hết tất cả các đường đi thì khó có thể khẳng định việc thay hàm giả gọi bằng hàm thật chắc chắn không sinh ra lỗi.

- Một số mô-đun bản chất là phức tạp nên dễ có lỗi hơn. Chúng ta cần xác định mô-đun gây ra lỗi nhiều nhất.

Kiểm thử tích hợp kết thúc khi toàn bộ các mô-đun được tích hợp đầy đủ với nhau, các lỗi phát hiện được sửa chữa. Khi hệ thống có nhiều mô-đun thì cũng có nhiều cách ghép chúng lại. Các cách ghép khác nhau sẽ kéo theo các phương pháp kiểm thử tích hợp khác nhau và mỗi trong chúng đều có các ưu nhược điểm. Các phần sau chúng ta sẽ xem xét các yếu tố gây lỗi tích hợp, các phương pháp ghép nối các mô-đun chính, và ưu nhược điểm của việc kiểm thử tích hợp tương ứng.

10.2.1 Các loại giao diện và lỗi giao diện

Mô-đun hóa là một nguyên lý quan trọng trong thiết kế phần mềm và các mô-đun tương tác với nhau qua các giao diện để thực hiện các yêu cầu chức năng của hệ thống. Một giao diện giữa hai mô-đun cho phép một mô-đun truy cập dịch vụ cung cấp bởi mô-đun kia. Giao diện có cả cơ chế chuyển điều khiển (thực thi) và chuyển dữ liệu giữa các mô-đun. Ba loại giao diện chính chúng ta thường gặp là:

- **Giao diện gọi hàm/thủ tục (procedure call):** một hàm trong một mô-đun gọi một hàm trong một mô-đun khác. Phía gọi sẽ chuyển điều khiển cho mô-đun được gọi. Phía gọi cũng có thể chuyển dữ liệu cho hàm được gọi. Ngược lại hàm được gọi cũng có thể chuyển dữ liệu trả về cho hàm gọi khi nó trả điều khiển về cho hàm gọi.

Trong ví dụ dưới đây, khi thực thi lệnh đầu tiên trong hàm `main()`, điều khiển sẽ được được chuyển cho hàm `print_str()` và dữ liệu là xâu ký tự `"Hello␣World!"` cũng được chuyển cho hàm được gọi (hàm `print_str`). Giao diện ở đây chính là chữ ký/khai báo hàm `void print_str(char*)`.

Đoạn mã 10.1: Giao diện gọi hàm/thủ tục

```
#include <stdio.h>
void print_str(char* str){
  printf("%s", str);
}
int main(void){
  print_str("Hello␣World!");
  return 0;
}
```

- **Giao diện bộ nhớ chia sẻ (shared memory):** một khối bộ nhớ được chia sẻ giữa hai mô-đun. Khối bộ nhớ này có thể do một trong hai mô-đun cấp phát, hoặc cũng có thể do một mô-đun thứ ba cấp phát. Một mô-đun sẽ ghi dữ liệu lên khối bộ nhớ và mô-đun kia đọc dữ liệu từ khối bộ nhớ.

  Trong ví dụ dưới đây hàm `main` và hàm `print_str` sử dụng bộ nhớ chia sẻ là biến `str` để trao đổi dữ liệu giữa các hàm này. Hàm `main()` ghi dữ liệu và hàm `print_str()` đọc dữ liệu. Trong trường hợp này, bộ nhớ cho biến `str` không được cấp phát mà sử dụng hằng ký tự.

Đoạn mã 10.2: Giao diện bộ nhớ chia sẻ

```
#include <stdio.h>
char* str;
void print_str()
{
  printf("%s", str);
}
```

```
int main(void)
{
  str = "Hello␣World!";
  print_str();
  return 0;
}
```

- **Giao diện truyền thông điệp (message passing):** một mô-đun tạo một thông điệp và gửi thông điệp đó cho một mô-đun khác. Dạng này rất phổ biến trong các hệ thống có nhiều tiến trình, khách-chủ hay các hệ thống trên nền Web, dịch vụ Web.

  Trong Đoạn mã 10.3 chương trình tạo một đường ống (bằng hàm `pipe()`) để tiến trình cha liên lạc với tiến trình con (sinh ra bởi hàm `fork()`). Sau khi sinh ra tiến trình con, tiến trình cha truyền một xâu ký tự vào đường ống và tiến trình con sử dụng vòng lặp để đọc dữ liệu từ đường ống và in ra màn hình.

Lỗi giao diện là lỗi gắn với các cấu trúc tồn tại bên ngoài môi trường của mô-đun nhưng được mô-đun đó sử dụng [BP84]. Một số lỗi giao diện này được phân loại như sau [PE85]:

1. *Không đủ chức năng:* lỗi này do một mô-đun giả thiết sai về mô-đun kia. Mô-đun cung cấp dịch vụ không hoạt động như mô-đun sử dụng mong đợi - cố tình hoặc ngoài ý muốn của người lập trình mô-đun cung cấp dịch vụ.

2. *Thay đổi tính năng:* một mô-đun được sửa đổi nhưng các mô-đun sử dụng nó không được điều chỉnh theo nên chức năng của hệ thống bị ảnh hưởng.

Đoạn mã 10.3: Giao diện truyền thông điệp

```
//-----------------------------------------------
//Excerpt from "Linux Programmer's Guide-Chapter~6"
//(C)opyright 1994-1995, Scott Burkett
//-----------------------------------------------
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>

int main ( int argc, char* argv[] ) {
  int   fd[2], nbytes;
  pid_t childpid;
  char string[] = "Hello, world!\n",readbuffer[80];

  pipe( fd );
  //fd[0] is opened for reading (input side);
  //fd[1] is opened for writing (output side).

  if ((childpid = fork()) == -1) {
    perror( "fork" );
    exit( 1 );
  }

  if (childpid == 0) {
    //child closes input side of pipe, & writes
    close(fd[0]);

    //Send "string" through the out. side of pipe
    write(fd[1], string, (strlen(string)+1));
  }
  else {
    //parent proc. closes out. side of pipe,\& reads
    close(fd[1]);
    // Read in a string from the pipe
    nbytes=read(fd[0],readbuffer,sizeof(readbuffer));
    //Print the obtained string
    printf("parent:received string:\%s",readbuffer);
  }
  return 0;
}
```

3. *Sử dụng giao diện không đúng:* một mô-đun đã sử dụng không đúng giao diện của mô-đun được gọi. Với giao diện hàm việc sử dụng sai này có thể do truyền tham số không đúng thứ tự.

4. *Hiểu giao diện không đầy đủ:* một mô-đun khi thiết kế đã giả thiết một số điều kiện của tham số đầu vào, nhưng phía gọi lại không để ý đến giả thiết này nên đã truyền các tham số nằm ngoài giả thiết. Ví dụ hàm tìm kiếm nhị phân giả sử đầu vào là một mảng được sắp, nhưng phía gọi không sắp xếp mảng này trước khi gọi thì lỗi xảy ra thuộc kiểu này.

5. *Không xử lý lỗi trả về:* một mô-đun được gọi có thể trả về một mã lỗi nhưng mô-đun gọi lại không kiểm tra lỗi, coi nó là kết quả. Hoặc mô-đun được gọi bổ sung thêm mã lỗi trả về nhưng mô-đun gọi chưa kịp biết/sửa.

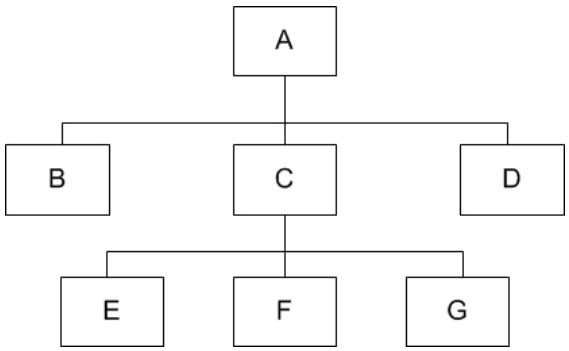
6. *Hiệu ứng phụ với tham số hoặc tài nguyên:* một mô-đun có thể sử dụng tài nguyên không mô tả trong giao diện. Ví dụ một mô-đun sử dụng file tạm tên là "temp", nhưng khi tích hợp một mô-đun khác cũng sử dụng file tạm với tên này sẽ gây lỗi xung đột. Hay ví dụ hàm `strdup` trong ngôn ngữ C cấp phát bộ nhớ mới và trả về con trỏ đến xâu mới. Nếu bên gọi không giải phóng bộ nhớ thì lỗi dò bộ nhớ sẽ xảy ra.

7. *Các vấn đề phi chức năng:* Các yêu cầu phi chức năng như tốc độ chỉ được nêu ra khi chúng có thể gây vấn đề. Các yêu cầu này ngay cả khi không nêu ra thì chúng ta vẫn ngầm định là chúng phải chạy không quá chậm. Khi tích hợp các vấn đề này mới thường phát sinh.

10.2.2 Tích hợp dựa trên cấu trúc mô-đun

Một chương trình trong ngôn ngữ C hay Java thường có hàm `main`, hàm này sẽ gọi các hàm khác trong thân của nó. Các hàm khác này

lỗi gọi tiếp các hàm khác nữa. Rộng hơn là một hệ thống có nhiều mô-đun thì theo cấu trúc phân cấp này chúng tạo thành một cấu trúc hình cây như trong Hình 10.1. Sơ đồ lớp trong nhiều ngôn ngữ hướng đối tượng sẽ có cấu trúc này.



Hình 10.1: Cấu trúc phân cấp mô-đun.

Khi đã có các đơn vị là đỉnh của cây chúng ta có thể lắp dần chúng với nhau và kiểm thử tích hợp trong quá trình lắp. Thứ tự lắp các mô-đun vào cây sẽ dẫn đến các chiến lược kiểm thử tương ứng. Có bốn cách ghép thông dụng là từ trên xuống (top down), dưới lên (bottom up), song song của cả trên xuống và dưới lên gọi là bánh kẹp (sandwitch), và một cách đơn giản khác là chỉ kiểm thử sau khi đã ghép hết tất cả các mô-đun (bigbang).

Nhược điểm của kiểm thử theo kiểu bigbang là khi có lỗi thì chúng ta rất khó để xác định vị trí của lỗi (khó định vị lỗi). Nếu chúng ta ghép dần dần và kiểm thử luôn thì khi có lỗi xuất hiện, chúng ta tập trung vào các mô-đun vừa ghép với nhau sẽ dễ xác định lỗi hơn. Chú ý ở đây chúng ta giả sử các mô-đun đã được kiểm thử đơn vị xong, nên kiểm thử tích hợp có thể coi là kiểm thử giao diện giữa các mô-đun.

10.2.3 Tích hợp từ trên xuống

Tích hợp từ trên xuống là tích hợp từ hàm chính (`main`) - gốc của cây. Các hàm được gọi trong hàm `main` trước khi tích hợp là các *hàm giả* (stub). Các hàm giả này là các hàm mô phỏng hàm được gọi và sẽ được bỏ đi khi tích hợp với hàm thật. Ví dụ hàm `main` gọi hàm `f(x)` và `g(y)` thì khi kiểm thử đơn vị hàm `main` chúng ta sử dụng hàm `f()` và `g()` là các hàm giả. Hàm giả sẽ nhận tham số và trả về kết quả như hàm thật nhưng nó không được cài đặt như hàm thật mà trả về luôn giá trị kết quả ứng với các một số tham số biết trước - tương ứng với ca kiểm thử sẽ được truyền cho `f(x)` và `g(y)`. Khi có các hàm giả này thì chúng ta có thể kiểm thử hàm `main` một cách độc lập, không phụ thuộc vào hàm thật. Lúc này chúng ta có thể áp dụng các kỹ thuật đã học về kiểm thử đơn vị để kiểm thử hàm `main`.

Khi chúng ta đã kiểm thử xong hàm `main`, chúng ta sẽ thay dần các hàm giả bằng các hàm thật. Bây giờ cũng có nhiều cách làm. Một là chúng ta thay toàn bộ các hàm giả bằng hàm thật rồi kiểm thử lại hàm `main`. Hai là chúng ta thay dần dần từng hàm một và kiểm thử. Với cách thứ hai này chúng ta phải chạy kiểm thử tích hợp hàm `main` nhiều lần hơn.

Trên thực tế người lập trình sẽ phải bỏ ra khá nhiều công sức để viết các hàm giả. Do đó phần việc viết các hàm giả này cũng cần coi như mã nguồn thông thường, và chúng ta cũng cần quản lý chúng tốt để có thể chuyển giữa mã giả và mã thật - do mã giả và mã thật không cùng tồn tại đồng thời được.

Với ví dụ ở Hình 10.1 tích hợp từ trên xuống có thể thực hiện với các bước như sau:

1. Tích hợp A với B. Lúc này chúng ta sẽ sử dụng hai mô-đun giả là C' và D'. Tương tác giữa hai A và B lúc này có thể vẫn bị hạn chế bởi các mô-đun C' và D' là mô-đun giả, chỉ chạy được với một số giá trị đã thiết kế trước. Một số ca kiểm thử

chỉ tác động đến mô-đun A và B thì có thể được thực hiện thêm. Nhớ rằng trước đó A được kiểm thử đơn vị độc lập thì phải sử dụng các mô-đun giả là B', C', và D'.

2. Tích hợp thêm D. Tức là với A, B đã tích hợp xong chúng ta bổ sung thêm mô-đun D. Hệ thống con sẽ gồm A, B và D. Chúng ta cần kiểm thử giữa A và D, sau đó chúng ta phải kiểm thử lại giữa A và B, vì bây giờ hệ thống con đã có thêm D, các tương tác giữa A và B đã kiểm thử trước đó có thể bị ảnh hưởng bởi D.

3. Tích hợp thêm C. Lúc này hệ thống con gồm A, B, C, D. Tương tự như bước trước.

4. Tích hợp thêm E. Tương tự bước trước.

5. Tích hợp thêm F. Tương tự bước trước.

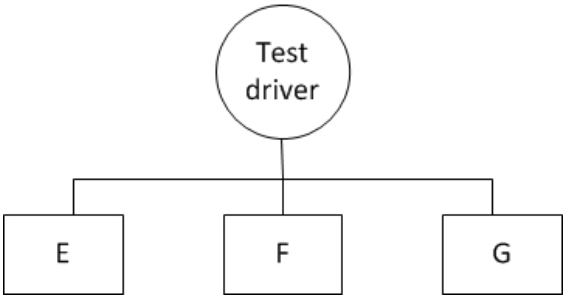
6. Tích hợp thêm G. Tương tự bước trước.

10.2.4 Tích hợp từ dưới lên

Tích hợp từ dưới lên ngược với tích hợp từ trên xuống. Khi kiểm thử chúng ta không cần các hàm giả, nhưng chúng ta cần các hàm giả gọi (driver). Hàm giả gọi này mô phỏng hàm ở mức trên mà trên sẽ gọi hàm chúng ta đang kiểm thử. Với ví dụ trên chúng ta phải mô phỏng hàm `main` để gọi các hàm `f` và `g` với các tham số phù hợp, và xử lý kết quả trả về như hàm `main` thật. So với các hàm giả, các hàm giả gọi này thường ít hơn về số lượng (vì cấu trúc cây luôn có số cành con, lá xòe ra), tuy nhiên việc giả lập các hàm giả gọi này lại thường phức tạp hơn.

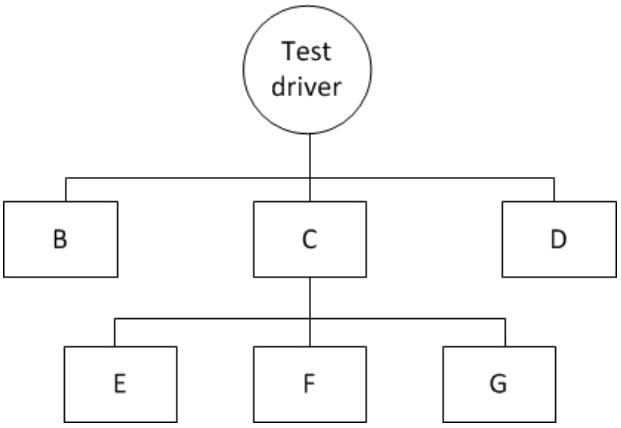
Với ví dụ ở Hình 10.1, chúng ta phải thiết kế hàm gọi giả để tích hợp ba mô-đun E, F, G như trong Hình 10.2. Chú ý là mặc dù E, F và G không có tương tác trực tiếp, giá trị trả về của một mô-đun có thể được truyền cho mô-đun khác, do đó chúng có thể

có tương tác gián tiếp. Hàm gọi giả phải được xây dựng đơn giản nhưng cần phản ánh đúng hành vi của mô-đun thật là C.



Hình 10.2: Tích hợp từ dưới lên mô-đun E, F và G.

Ở bước tiếp theo, chúng ta sẽ tích hợp các mô-đun B, C, D với E, F và G như mô tả trong Hình 10.3. Ở đây chúng ta thay C vào nhưng hệ thống không thực hiện được từ hàm C, nên chúng ta phải sử dụng hàm giả mô phỏng mô-đun A. Khi đó chúng ta cần tích hợp thêm các mô-đun B, D để hàm gọi giả có thể hoạt động và để kiểm tra được mô-đun C.



Hình 10.3: Tích hợp từ dưới lên mô-đun B, C, D với E, F và G.

10.2.5 Tích hợp bánh kẹp

Tích hợp bánh kẹp là tổ hợp của tích hợp từ trên xuống và từ dưới lên. Có thể hình dung cách tích hợp này là tích hợp bigbang với các cây con. Ưu điểm là chúng ta sẽ phải viết ít các hàm giả và các hàm giả gọi hơn, nhưng nhược điểm là chúng ta sẽ khó truy tìm lỗi.

10.2.6 Tích hợp dựa trên đồ thị gọi hàm

Một trong những nhược điểm của tích hợp dựa trên cấu trúc mô-đun hay còn gọi là đồ thị chương trình (program graph) là nó chỉ dựa trên cây phân rã cấu trúc. Đồ thị gọi hàm (call graph) là đồ thị có hướng và chúng ta có hai chiến lược tích hợp là tích hợp đôi một (pairwise) và tích hợp láng giềng (neighborhood).

10.2.7 Tích hợp đôi một

Tích hợp đôi một nhằm mục đích giảm công sức phải làm các hàm giả và hàm gọi giả. Chúng ta sử dụng luôn các hàm thật thay cho hàm giả. Mới nghe thì chúng ta có cảm giác kiểu tích hợp này giống kiểu bigbang, nhưng ở đây mỗi lần ghép chúng ta chỉ ghép một cặp đơn vị trên đồ thị gọi hàm. Nói cách khác mỗi lần tích hợp chúng ta chỉ ghép hai đỉnh của một cạnh của đồ thị. Số lần tích hợp như vậy cũng không thay đổi so với tích hợp từ trên xuống hay từ dưới lên, tuy nhiên chúng ta giảm được đáng kể các hàm giả và hàm gọi giả.

10.2.8 Tích hợp láng giềng

Láng giềng của một đỉnh N của một đồ thị là tập hợp các đỉnh có cạnh nối với N. Với đồ thị có hướng thì láng giếng là tập hợp các đỉnh có cạnh đi trực tiếp đến N và các đỉnh có cạnh đi trực tiếp

ra từ N. Dễ thấy, các đỉnh láng giếng chính là các hàm giả và hàm gọi giả chúng ta cần phải xây dựng.

Chúng ta có thể tính số lượng láng giềng của một đồ thị. Mỗi đỉnh bên trong đồ thị có một láng giềng, và thêm một láng giếng nếu có một đỉnh lá nối trực tiếp với đỉnh gốc. (Đỉnh bên trong của đồ thị là đỉnh có ít nhất một cạnh đi đến và một cạnh đi ra, hay nói cách khác số bậc vào và số bậc ra của chúng lớn hơn 0.)

Tích hợp láng giềng giúp giảm đáng kể số lần tích hợp đồng thời cũng giúp giảm số lượng và công sức viết các hàm giả và hàm gọi giả. Tích hợp láng giếng gần giống với tích hợp bánh kẹp. Khác biệt chính là chúng dựa trên các đồ thị khác nhau. Nhược điểm chung của chúng là khi có lỗi chúng ta khó khoanh vùng chính xác được, tương tự như tích hợp bigbang ở phạm vi nhỏ.

10.3 Kiểm thử hệ thống

Như đã nói ở phần trước, kiểm thử hệ thống có hai phần là kiểm thử chức năng và kiểm thử chất lượng. Kiểm thử chức năng kiểm tra xem các chức năng người dùng yêu cầu ghi trong đặc tả đã đủ và đúng chưa. Kiểm thử chất lượng kiểm tra các yêu cầu phi chức năng ghi trong đặc tả, nhưng thường đội phát triển cần kiểm tra những yêu cầu chất lượng ngầm định thông thường và hiển nhiên cần có. Ví dụ các phần mềm chạy trên nền web ngày nay đều ngầm định yêu cầu an ninh (security), sẵn sàng (avaiablity), và khả năng chịu tải cao.

Đặc trưng cơ bản của kiểm thử hệ thống là chúng ta dựa trên đặc tả hành vi quan sát được của phần mềm, không phụ thuộc vào chi tiết thiết kế và cài đặt. Kiểm thử hệ thống thành công có nghĩa sản phẩm đã hoàn thiện, không còn lỗi và có thể đưa vào sử dụng. Trên thực tế rất khó để đảm bảo hệ thống không còn lỗi mới đưa vào sử dụng, ví dụ vì ngày phát hành đã ấn định, hay các lỗi còn lại không nghiêm trọng. Do đó người ta đưa ra một số tiêu chuẩn

để sản phẩm có thể còn một số lỗi nhỏ, không nghiêm trọng thì vẫn đạt, tức là kiểm thử hệ thống là xong, có thể chuyển sang giai đoạn kiểm thử tiếp theo.

10.3.1 Kiểm thử chức năng hệ thống

Tương tự kiểm thử đơn vị, kiểm thử hệ thống cũng có bộ các ca kiểm thử cho từng chức năng. Mỗi ca kiểm thử chức năng thường mô tả từng bước thực hiện ở mức sử dụng phần mềm để hoàn thành một tác vụ nào đó. Mỗi bước thực hiện có thể có dữ liệu đầu vào, thao tác thực hiện, và kết quả mong đợi quan sát được tương ứng để người kiểm thử khi thực hiện từng bước này sẽ kiểm tra xem phần mềm hoạt động đúng không.

Trước đây việc thực hiện kiểm thử này phải làm bằng tay, tốn rất nhiều chi phí, cả máy móc, thời gian và sức người. Tuy nhiên gần đây việc thực hiện này đang được tự động hóa ngày càng nhiều nhờ có các công cụ mạnh đang được chú trọng nghiên cứu và phát triển tích cực. Một ví dụ có thể kể đến là Selenium[1] - một công cụ có thể thực hiện các ca kiểm thử tự động cho các ứng dụng trên nền Web. Người kiểm thử có thể thực hiện thao tác kiểm thử lần đầu và cho Seleinum ghi lại các thao tác của mình dưới dạng các mã dễ đọc của Selenium. Sau đó các ca kiểm thử này được chạy lại tự động. Người kiểm thử có thể chỉnh sửa mã Selenium sinh ra cho phù hợp hơn với ý đồ của ca kiểm thử. Khi phải kiểm tra tính tương thích của ứng dụng trên nhiều trình duyệt khác nhau, hoặc khi phải chạy lại các ca kiểm thử thì công cụ này giúp giảm đáng kể công sức phải thực hiện lại.

Việc xây dựng các ca kiểm thử chức năng được gọi là việc thiết kế kiểm thử. Các ca kiểm thử chức năng nên được thiết kế chỉ dựa trên đặc tả chức năng của phần mềm, độc lập với thiết kế và cài đặt của phần mềm. Cách làm này giúp phát hiện các lỗi trong thiết

[1] http://seleniumhq.org/

kế vì nếu dựa trên thiết kế có lỗi, các ca kiểm thử sẽ được xây dựng cho đúng với thiết kế chứ không đúng theo đặc tả chức năng. Tuy nhiên việc thiết kế kiểm thử hoàn toàn độc lập này có thể lại không giúp phát hiện các vấn đề với thiết kế. Do đó trên thực tế chúng ta cần cân đối sử dụng một mức độ thông tin trong thiết kế và cài đặt để thiết kế ca kiểm thử. Mức độ này là bao nhiêu phụ thuộc rất nhiều vào tính chất của từng dự án và kinh nghiệm của người thiết kế kiểm thử. Cuối cùng kiểm thử hệ thống có thể sử dụng lại các ca kiểm thử mức thấp hơn như kiểm thử tích hợp, kiểm thử đơn vị.

Một khía cạnh khác nữa với kiểm thử hệ thống là việc tổ chức thực hiện. Chúng ta để một đội vừa phát triển vừa thực hiện kiểm thử hệ thống hay để hai đội độc lập thực hiện hai việc riêng biệt? Khi thực hiện độc lập thì khối lượng công việc tăng lên vì đội thực hiện phải đọc lại tài liệu và hiểu hệ thống mới có thể thực hiện kiểm thử. Nhưng sự khách quan này thường giúp phát hiện nhiều vấn đề trong hệ thống và kết quả cho phép kiểm thử chấp nhận dễ dàng vượt qua hơn. Nếu đội phát triển thực hiện kiểm thử hệ thống thì yếu tố chủ quan sẽ giúp việc kiểm thử tập trung vào một số điểm yếu của hệ thống, tuy nhiên đội sẽ bị ảnh hưởng bởi những thông tin đã biết về thiết kế và cài đặt của hệ thống nên thiếu tính khách quan trong thiết kế và thực hiện kiểm thử. Để giảm sự ảnh hưởng chủ quan này, trong các dự án phát triển theo quy trình linh hoạt [Mar03] người ta phát triển các ca kiểm thử hệ thống ngay từ đầu, trong quá trình thu thập yêu cầu và trong quá trình xây dựng hệ thống cũng bổ sung các ca kiểm thử hệ thống ngay khi có thể.

Với kiểm thử hệ thống chúng ta cũng cần một khái niệm về mức độ kỹ của bộ kiểm thử so với đặc tả. Việc này có thể thực hiện bằng cách tạo ít nhất một ca kiểm thử cho mỗi tính năng hay hành vi của hệ thống. Hơn nữa, việc này nên được xác định ngay từ lúc thu thập yêu cầu, vì các ca kiểm thử này sẽ làm rõ đặc tả của hệ thống, giúp quá trình thiết kế và lập trình có thể đối chiếu

ngay với các kịch bản kiểm thử. Đây cũng là tư tưởng của quy trình phát triển dựa trên hành vi (Behaviour Driven Development - BDD) [Kar12] hay phát triển dựa trên kiểm thử (Acceptance Test Driven Development – ATDD) [GG12]. Các ca kiểm thử theo BDD thường được viết dưới dạng ngôn ngữ tự nhiên, gần với ngôn ngữ của người đưa ra yêu cầu, tuy nhiên sẽ tuân thủ một số nguyên tắc để giúp công cụ tự động hóa được. Các ca kiểm thử này cũng trở thành thước đo về tiến độ của dự án. Khi các tính năng được cài đặt thì thực hiện các ca kiểm thử sẽ thành công dần cho đến khi không còn ca kiểm thử nào thất bại có nghĩa là hệ thống đã đáp ứng đúng và đủ các yêu cầu đặt ra. Lưu ý là các ca kiểm thử này vẫn có thể liên tục được bổ sung, chỉnh sửa trong quá trình phát triển nếu cần, khi có vấn đề trong cài đặt hoặc vấn đề ngay trong yêu cầu được mô tả dưới các ca kiểm thử.

10.3.2 Kiểm thử chất lượng hệ thống

Bên cạnh các chức năng nghiệp vụ của hệ thống, các tính chất chất lượng (quality attributes/factors) [LB12, KN08] của hệ thống như độ sẵn sàng, hiệu năng, độ tin cậy cũng là các tính chất quan trọng của các phần mềm. Phần mềm có đủ chức năng, chạy đúng, nhưng chạy chậm, hay khó sử dụng, hay không ổn định đều không được người dùng chấp nhận. Nhiều khi các yêu cầu chất lượng này là ngầm định, người dùng không nói ra, không mô tả trong tài liệu nhưng chúng ta vẫn phải thực hiện khi kiểm thử hệ thống. Nhiều các yếu tố chất lượng này chỉ có thể kiểm tra được với hệ thống hoàn chỉnh chứ không thể suy ra từ tính chất của các thành phần của hệ thống.

Một khó khăn của kiểm thử chất lượng là có những tính chất của hệ thống khó có thể chỉ ra chính xác, lượng hóa được. Những tính chất này đôi khi còn phụ thuộc vào môi trường phù hợp mới thể hiện, có thể phát hiện ra. Ví dụ một ứng dụng quản lý cơ sở

dữ liệu sau một hai năm sử dụng số bản ghi tăng lên thì sự chậm chễ mới dễ nhận thấy, khi người dùng luôn phải chờ khi thực hiện thao tác thêm mới hoặc truy vấn cơ sở dữ liệu. Đặc biệt vấn đề an ninh, an toàn rất khó có thể lường trước và đảm bảo không còn nguy cơ an ninh bảo mật nào trong hệ thống. Một ví dụ khác nữa là tính dễ bảo trì hệ thống. Khi hệ thống mới xây dựng xong thì đội phát triển đều còn nhớ và thuộc yêu cầu cũng như thiết kế và mã nguồn. Tuy nhiên sau một thời gian khi họ chuyển sang dự án khác hoặc có người mới tham gia vào dự án thì tính dễ bảo trì của dự án không còn được đánh giá như lúc trước nữa, mặc dù tài liệu và mã nguồn không thay đổi.

Kiểm thử các tính chất của toàn hệ thống cũng cần mô phỏng môi trường thực hiện kiểm thử và việc này đòi hỏi rất nhiều công sức để thiết lập. Ví dụ để tính điểm tổng kết học kỳ cho một lớp học, chúng ta phải nhập dữ liệu điểm đầy đủ các đầu điểm, và mô phỏng đầy đủ các điểm cao thấp để có thể xem việc xét lên lớp có đúng không, các báo cáo tổng hợp có đúng không, việc làm tròn điểm có đúng quy định và giống với tính tay không. Khối lượng công việc này khi làm thực tế là rất nhiều, cả sức người và sức máy.

Có nhiều khía cạnh về chất lượng của một hệ thống phần mềm [LB12]. Trong thực tế, tùy thuộc vào từng bài toán cụ thể, chúng ta sẽ ưu tiên và chọn một số chất lượng quan trọng, cần thiết của hệ thống để kiểm thử. Ví dụ các phần mềm ngân hàng trực tuyến thì an ninh an toàn là quan trọng nhất, tiếp đó là tính sẵn sàng, hay dễ sử dụng, hay tính tương thích trình duyệt. Danh sách sau liệt kê và giải thích một số loại kiểm thử chất lượng phổ biến. Sau đó chúng ta sẽ thảo luận kỹ hơn về hai tính chất chất lượng điển hình là tính dễ dùng và hiệu năng.

- *Kiểm thử cài đặt* (basic) đảm bảo hệ thống có thể cài đặt được, cấu hình được, và đưa vào trạng thái sử dụng được.
- *Kiểm thử tính dễ dùng* (usability) đảm bảo hệ thống làm hài lòng người dùng như dễ học, dễ nhớ, dễ thao tác, ít lỗi.

- *Kiểm thử hiệu năng* (performance) đo các đặc trưng về hiệu năng của hệ thống như thông lượng và thời gian trả lời của hệ thống dưới các điều kiện khác nhau.
- *Kiểm thử tương thích* (interoperability) kiểm tra xem hệ thống có thể hoạt động với các sản phẩm khác hay không.
- *Kiểm thử khả năng mở rộng* (scalibility) xác định các giới hạn mở rộng và thu hẹp hệ thống như số người dùng tăng lên, lượng dữ liệu tăng lên.
- *Kiểm thử áp lực* (stress) đưa hệ thống vào điều kiện áp lực đến khi hệ thống có vấn đề, và từ đó xác định các vấn đề xảy ra.
- *Kiểm thử sức mạnh* (robustness) xác định khả năng hệ thống hồi phục lại từ đầu vào lỗi hoặc các sự cố trục trặc khác.
- *Kiểm thử tải* (load) nhằm kiểm tra hệ thống có ổn định khi chạy với công suất tối đa trong một thời gian dài.
- *Kiểm thử tin cậy* (reliability) đo khả năng hệ thống vẫn tiếp tục hoạt động trong một thời gian dài mà không phát sinh các sự cố.
- *Kiểm thử hồi quy* (regression) kiểm tra xem hệ thống còn ổn định không qua các giai đoạn chỉnh sửa, cải tiến.
- *Kiểm thử tài liệu* (document) đảm bảo các tài liệu hướng dẫn sử dụng là chính xác và dùng được.
- *Kiểm thử quy định* (regulatory) để đảm bảo hệ thống tuân thủ các quy định, chuẩn mực của vùng, miền, đất nước nơi sản phẩm được triển khai.

Sau đây chúng ta xem xét sơ lược về kiểm thử hai tính chất khá điển hình là tính dễ dùng và hiệu năng (performance).

10.3.2.1 Kiểm thử tính dễ dùng

Tính dễ sử dụng (usability) là một trong những yếu tố quan trọng nhất quyết định khả năng thành công của sản phẩm. Tính dễ sử dụng bao gồm một loạt tính chất như dễ học, giúp người dùng hoàn thành công việc dễ dàng, hiệu quả hơn, và làm người dùng dễ chịu khi sử dụng phần mềm. Một số phép đo cụ thể có thể sử dụng để đo các tính chất trên. Ví dụ số thao tác để hoàn thành một tác vụ, số lỗi người dùng mắc phải trong quá trình thao tác. Tuy nhiên quan trọng nhất là cảm nhận tổng thể của người dùng về phần mềm. Trên thực tế tính dễ sử dụng đã đóng vai trò quyết định thành công của một loạt sản phẩm như Microsoft Windows, Microsoft Office hay điện thoại iPhone dù các sản phẩm này đắt tiền và (trước đây) có nhiều vấn đề an ninh, lỗ hổng bảo mật.

Một số chất lượng khác cũng liên quan đến tính dễ sử dụng như độ tin cậy, hiệu năng, an ninh an toàn, v.v. Ví dụ phần mềm Google Docs có các chức năng soạn thảo và bảng tính khá cơ bản và đủ dùng, nhưng đòi hỏi kết nối Internet nhanh nên người dùng ở Việt Nam ít dùng vì tốc độ Internet ở Việt Nam khá chậm và thiếu ổn định.

Quy trình kiểm tra tính dễ sử dụng bao gồm các bước chính sau đây [Rub94]:

- Kiểm tra thiết kế giao diện người dùng với danh sách kiểm tra tính dễ sử dụng (usability checklist).

- Kiểm thử thăm dò bản mẫu phác thảo với người dùng để xem thiết kế có phù hợp mô hình tư duy, trình độ của người dùng, và để phát hiện các cải tiến phù hợp (kiểm thử so sánh) hay khẳng định thiết kế đã phù hợp. Bản mẫu này không nhất thiết phải hoạt động đầy đủ, chỉ là hình ảnh hoặc cao hơn là có giao diện tương tác được nhưng chưa thực sự hoạt động về mặt xử lý bên trong.

- Kiểm thử các phiên bản của phần mềm với cả các chuyên gia về tính dễ sử dụng và với người dùng để theo dõi quá trình sử dụng của họ và phát hiện các vấn đề về tính dễ sử dụng của sản phẩm.

- Kiểm thử hệ thống và kiểm thử chấp nhận với chuyên gia và với người dùng thật, có thể kết hợp so sánh với các sản phẩm cạnh tranh.

Kiểm thử với người dùng là kiểm thử với một số đại diện của người dùng thật sau này của sản phẩm. Việc này đặc biệt quan trọng để khẳng định sản phẩm đã là dùng được tốt chưa và có thể tiến hành sớm, ngay khi có giao diện phác thảo của phần mềm, và mỗi khi có chỉnh sửa, cải tiến về giao diện. Suy cho cùng sự chấp nhận của người dùng cuối cùng vẫn là yếu tố quyết định, nhiều khi vượt qua cả các ý kiến chuyên gia và chuẩn mực trên lý thuyết.

Mục đích của kiểm thử thăm dò là để điều tra các mô hình tư duy của người dùng cuối. Chúng ta tiến hành bằng việc hỏi người dùng về cách họ sẽ tương tác với hệ thống. Thực tế cho thấy nhiều sản phẩm thành công đều có giao diện và cách sử dụng rất đơn giản. Kiểm thử thăm dò thường được thực hiện sớm từ lúc thiết kế, đặc biệt là khi thiết kế sản phẩm cho một nhóm đối tượng mới.

Mục đích của kiểm thử so sánh là để xem xét các lựa chọn khác. Chúng ta sẽ quan sát phản ứng của người dùng với các mẫu giao diện và cách tương tác khác nhau. Đôi khi chúng ta cung cấp cho người dùng nhiều cách để làm cùng một việc, vì có người phù hợp và muốn hoặc quen cách tương tác này, nhưng có người lại thích và chọn cách sử dụng khác, vì kiến thức và kinh nghiệm của họ là khác nhau. Những người chuyên nghiệp, sử dụng thường xuyên sản phẩm thường thích dùng bàn phím, còn nhiều người ít sử dụng, không thạo thì thường hay dùng chuột.

Mục đích của kiểm thử xác nhận là đánh giá tổng hợp tính dễ sử dụng của sản phẩm. Chúng ta cần xác định những khó khăn và

trở ngại mà người dùng gặp phải trong khi tương tác với hệ thống, cũng như các đặc trưng qua đo đạc như tỷ lệ lỗi và thời gian để hoàn thành một việc.

Thiết kế kiểm thử tính dễ dùng gồm việc chọn đại diện người dùng thích hợp và tổ chức các phiên hướng dẫn để có thể triển khai việc dùng thử đồng thời cho phép quan sát hoặc ghi lại các diễn biến để phân tích đánh giá tính dễ sử dụng. Cách tiếp cận phổ biến thường chia quá trình này thành ba giai đoạn: chuẩn bị, thực hiện và phân tích. Ở giai đoạn chuẩn bị, người thiết kế kiểm thử xác định mục tiêu của mỗi phiên, và xác định các mục sẽ kiểm thử, chọn đại diện người dùng phù hợp, và lập kế hoạch cho các hoạt động cần thiết khác. Trong quá trình thực hiện, người dùng được theo dõi các hành động sử dụng hệ thống, trong môi trường có kiểm soát. Giai đoạn phân tích sẽ đánh giá kết quả, xác định những thay đổi cần có cho giao diện phần mềm, và lập kế hoạch kiểm thử lại nếu cần thiết.

Mỗi giai đoạn phải được thực hiện một cách cẩn thận để đảm bảo sự thành công của phiên kiểm thử. Thời gian của người dùng là rất quý giá và hạn chế. Mỗi phiên kiểm thử chúng ta tập trung vào một số mục tiêu. Mục tiêu này không được quá hẹp để tránh lãng phí nguồn lực nhưng cũng không được quá rộng để tránh việc tài nguyên bị rải rác và kết quả khó chính xác. Mỗi phiên nên tập trung vào một số giao diện và tương tác cụ thể thay vì đánh giá toàn bộ phần mềm một lúc.

Khi kiểm thử chấp nhận, người dùng cần thực hiện nhiệm vụ một cách độc lập mà không có sự giúp đỡ hoặc ảnh hưởng từ các đội phát triển và kiểm thử. Các hành động cũng như các bình luận của người dùng được ghi lại và cuối cùng ấn tượng của họ được hỏi sau trong phần hỏi đáp. Hoạt động giám sát có thể rất đơn giản, chẳng hạn như ghi lại hành trình của chuột cùng các lần nhấp chuột và các phím được gõ. Phức tạp hơn chúng ta có thể ghi thêm thời gian, theo dõi mắt và hành động của người dùng qua camera.

Một khía cạnh quan trọng khác của khả năng sử dụng được là khả năng tiếp cận (accessibility) đến sản phẩm của tất cả người dùng [Kru05], gồm cả những người khuyết tật. Kiểm thử tiếp cận là yêu cầu theo luật trong một số lĩnh vực ứng dụng. Ví dụ, cổng giao tiếp của chính phủ điện tử bắt buộc phải cho những người khuyết tật cũng sử dụng được. Tài liệu hướng dẫn tiếp cận nội dung các trang Web (WCAG)[1] do World Wide Web đưa ra đang trở thành một tiêu chuẩn quan trọng.

Hướng dẫn khả năng tiếp cận nội dung Web (WCAG)

- Cung cấp các lựa chọn thay thế tương đương với nội dung âm thanh và hình ảnh mà căn bản vẫn truyền đạt được mục đích và chức năng của nó.
- Đảm bảo rằng văn bản và đồ họa là dễ hiểu khi nhìn mà không có màu sắc.
- Đánh dấu tài liệu với các yếu tố cấu trúc phù hợp, kiểm soát trình bày bằng trang thay vì trình bày bằng các thuộc tính.
- Sử dụng cách đánh dấu bằng phát âm hoặc giải thích văn bản được viết tắt hoặc được viết bằng tiếng nước ngoài.
- Đảm bảo rằng các bảng đã được đánh dấu cần thiết để được biến đổi theo các truy cập vào trình duyệt khác nhau bởi các đại lý sử dụng khác nhau.
- Đảm bảo rằng các trang có thể truy cập ngay cả khi các công nghệ mới hơn không được hỗ trợ hoặc đã bị tắt.
- Đảm bảo rằng chuyển động, nhấp nháy, di chuyển, hoặc tự động cập nhật các đối tượng hoặc các trang có thể được tạm dừng hoặc dừng lại.

[1] http://www.w3.org/TR/WAI-WEBCONTENT

- Đảm bảo rằng giao diện người dùng, bao gồm các yếu tố giao diện người dùng nhúng, sau các nguyên tắc thiết kế truy cập: truy cập thiết bị một cách độc lập với chức năng, khả năng hoạt động của bàn phím, selfvoicing, v.v.

- Sử dụng tính năng cho phép kích hoạt các yếu tố trang thông qua một loạt các thiết bị đầu vào.

- Sử dụng tạm thời khả năng tiếp cận công nghệ để giúp việc và các trình duyệt cũ hơn sẽ hoạt động một cách chính xác.

- Trường hợp các công nghệ bên ngoài của chi tiết kỹ thuật W3C được sử dụng (ví dụ, Flash), cung cấp các phiên bản thay thế để đảm bảo khả năng truy cập chuẩn cho các đại lý người dùng và có công nghệ trợ giúp (ví dụ, màn hình người đọc).

- Cung cấp bối cảnh và định hướng thông tin để giúp người dùng hiểu các trang hoặc các yếu tố phức tạp.

- Cung cấp cơ chế chuyển hướng rõ ràng và nhất quán để tăng khả năng rằng một người sẽ tìm thấy những gì họ đang tìm kiếm ở một trang Web.

- Đảm bảo rằng các tài liệu được rõ ràng và đơn giản vì thế mà nó dễ hiểu hơn.

10.3.2.2 Kiểm thử hiệu năng

Có nhiều định nghĩa về kiểm thử hiệu năng như [MFB+07, Mol09]. Theo [MFB+07], kiểm thử hiệu năng là kiểm thử xác định thời gian phản hồi (repsonsiveness), thông lượng (throughput), mức độ tin cậy (reliability) hoặc khả năng mở rộng (scalability) của hệ thống theo khối lượng công việc (workload).

Kiểm thử nghiệm hiệu năng tìm câu trả lời cho các câu hỏi như: hệ thống có thể phục vụ tối đa bao nhiêu người dùng đồng thời, trong khoảng thời gian bao lâu, khi chạy với số người tối đa như vậy thì thời gian phản hồi và thông lượng là bao nhiêu, sử dụng bao nhiêu phần trăm CPU, bao nhiêu bộ nhớ của máy chủ, v.v.?

Kiểm thử hiệu năng về cơ bản giúp chúng ta khẳng định phần mềm đã chạy đủ nhanh chưa, có chậm quá đến mức làm người dùng không thể dùng được hay không. Ngoài ra kiểm thử hiệu năng còn giúp chúng ta có thêm các thông tin hữu ích như:

- So sánh hiệu năng hiện tại của hệ thống với hiệu năng của hệ thống khác đang làm hài lòng người dùng.
- Khẳng định hiệu năng của hệ thống đã đúng theo yêu cầu của khách hàng.
- Phân tích hành vi của hệ thống ở các mức tải khác nhau, giúp ta đánh giá hiệu năng của hệ thống một cách toàn diện hơn.
- Xác định nguyên nhân ảnh hưởng đến hiệu năng của hệ thống và các điểm tắc nghẽn, nút cổ chai.
- Xác định cấu hình của thiết bị cần mua sắm cho hệ thống thực tế, đáp ứng được hiệu năng mong muốn.
- Khẳng định các giải pháp tốt hơn so với các giải pháp hiện tại về mặt hiệu năng.

Kiểm thử hiệu năng liên quan: Khi thực hiện kiểm thử hiệu năng cho một ứng dụng chúng ta có thể thực hiện một hoặc một vài loại kiểu kiểm thử hiệu năng liên quan tùy theo đặc thù và yêu cầu của ứng dụng đặt ra. Dưới đây sẽ giới thiệu một vài kiểu kiểm thử hay được sử dụng trong kiểm thử hiệu năng.

Kiểm thử cơ sở (baseline test): Kiểm thử cơ sở đánh giá hiệu năng với một người dùng. Kiểm thử cơ sở được thực hiện để kiểm tra tính đúng của kịch bản kiểm thử được phát triển cho kiểm thử hiệu năng. Kịch bản kiểm thử có thể được tạo ra với thời gian nghĩ (think time) trong thực tế và những cài đặt khác giống sử dụng thực tế. Thông tin về thời gian phản hồi, các số liệu sử dụng tài nguyên máy chủ được thu thập và lưu lại để tính toán hiệu năng của hệ thống ở điều kiện tải khác nhau.

Kiểm thử chuẩn (benchmark test): Kiểm thử chuẩn để đo hiệu năng của ứng dụng trong điều kiện tải thấp, thường chỉ khoảng 15-20% mức tải dự kiến. Kiểm thử chuẩn còn có mục đích kiểm tra tính đúng đắn của kịch bản kiểm thử và tính sẵn sàng của hệ thống trước khi chuyển sang điều kiện tải cao. Nó giúp chúng ta nhận ra các thành phần khác nhau của hệ thống kết hợp với nhau theo thiết kế có đáp ứng mức dịch vụ cung cấp.

Thời gian phản hồi của hệ thống và số liệu về mức sử dụng tài nguyên máy chủ sẽ giúp chúng ta phân tích hệ thống có thể đứng vững với kiểm thử tải người dùng cao. Các số liệu thu thập trong kiểm thử chuẩn sẽ được xem như là kết quả hiệu năng tốt nhất của hệ thống. Chúng ta có thể đánh giá sự giảm sút hiệu năng của hệ thống bằng cách so sánh với hiệu năng hệ thống trong kiểm thử chuẩn.

Kiểm thử tải (load test): Kiểm thử tải đo hiệu năng hệ thống với điều kiện tải cao, nhiều người dùng đồng thời như trong thực tế. Nó giúp ta kiểm tra tải dự kiến có đạt được hay không với điều kiện thiết bị, môi trường đã có. Ngoài ra kiểm thử tải giúp đánh giá hiệu năng hệ thống trong những điều kiện tải khác nhau như trong điều kiện tải bình thường và tải cao điểm. Ví dụ khi cao điểm thì người dùng phải chờ lâu hơn, tuy nhiên thời gian chờ là bao nhiêu, có chấp nhận được không, hay có gây ra những lỗi gì không, như từ chối dịch vụ (DoS). Ví dụ một ứng dụng Web cho phép tối đa 1000 người dùng đồng thời. Chúng ta giả lập cho hệ thống chạy ở

đúng mức 1000 người dùng liên tục trong hai ngày. Hay chúng ta cho ban đầu là 800 người dùng đồng thời, rồi mỗi phút tăng thêm hai người nữa, đến khi đạt 1000 người và để ở tải này tiếp trong hai ngày. Sau đó chúng ta xem các báo cáo sử dụng CPU, RAM, thời gian phản hồi của hệ thống, thông lượng.

Kiểm thử quá tải (stress test): Kiểm thử quá tải kiểm tra hệ thống trong điều kiện tải bất hợp lý để xác định điểm ngoặt (break-point) của hệ thống. Kiểm thử này được coi là kiểm thử tiêu cực vì hệ thống phải chịu một tải quá thực tế yêu cầu. Chúng ta cần kiểm thử này để biết tác hại có thể xảy ra nếu hệ thống phải chịu quá tải. Nếu kết quả tích cực, chúng ta có thể dành tài nguyên của máy móc cho việc khác hoặc sẵn sàng mở rộng thêm trong tương lai.

Ví dụ một ứng dụng web có thể đáp ứng tối đa là 1000 người dùng đồng thời. Chúng ta cho mô phỏng cho 1000 người dùng đồng thời rồi tăng thêm 100 người đồng thời quan sát kết quả xem 100 người bổ sung này bị từ chối hay không, máy chủ có bị khởi động lại không, hay có bị treo không, v.v. Từ đó đưa ra kết luận tình trạng của hệ thống khi bị quá tải.

Kiểm thử đột biến (spike test): Kiểm thử này giống kiểm thử quá tải (stress test) nhưng khác ở điểm hệ thống được đặt ở tải cực cao trong một thời gian gian ngắn để xem những thời điểm bất thường này có gây ra sự cố gì không.

Kiểm thử độ bền (endurance test): Kiểm thử độ bền đánh giá hiệu năng của hệ thống với mức tải bình thường nhưng trong thời gian kéo dài. Kiểm thử này giúp phát hiện những vấn đề phải qua thời gian dài mới bộc lộ như dò bộ nhớ, tràn bộ đệm, hết đĩa cứng. Ngoài ra nó còn giúp đánh giá tính sẵn sàng (availability) của hệ thống. Kiểm thử độ bền thường chạy với 70%- 80% của tải tối đa.

10.4 Kiểm thử chấp nhận

Mặc dù có phần trùng lặp với với kiểm thử hệ thống, mục đích chính của kiểm thử chấp nhận là để quyết định xem sản phẩm có đúng mong muốn của người dùng chưa, có thể đưa vào sử dụng được chưa. Quyết định này có thể dựa vào các *số đo* của sản phẩm hoặc quy trình. Các số đo của sản phẩm thường được suy ra từ kết quả kiểm thử thống kê. Số đo quy trình thường dựa trên kinh nghiệm của người đánh giá so với sản phẩm trước đó. Các số đo này đánh giá mức độ tin cậy của phần mềm để làm cơ sở cho việc quyết định triển khai cho người dùng thật.

Các số đo về độ tin cậy, độ sẵn sàng, thời gian hỏng hóc trung bình là các số đo mang tính thống kê vì chúng ta rút ra các số đo này dựa trên kết quả kiểm thử của một số mẫu. Ví dụ nếu hệ thống có xử lý các giao dịch và mỗi giao dịch có một số thao tác với thì tỷ lệ thất bại của giao dịch khác hoàn toàn tỷ lệ thất bại của thao tác. Hơn nữa tỷ lệ thất bại giao dịch cũng khác nhau khi thực hiện chúng vào cùng một thời điểm hay rải rác đều trong một ngày. Do đó chúng ta cần dựa trên một mô hình thống kê để xây dựng mô hình sử dụng và kết quả của kiểm thử thống kê là tương đối với mô hình đó.

Mô hình thống kê sử dụng, hay còn gọi là hồ sơ vận hành (operational profiles), có thể thu được từ các số đo của các hệ thống tương tự trước đó. Ví dụ, chiếc điện thoại cầm tay đang được sử dụng như thế nào sẽ là mô hình cho một mẫu điện thoại mới sẽ được sử dụng. Tuy nhiên việc sử dụng mô hình kiểu này cũng không dễ cho kết quả chính xác vì trên thực tế có rất nhiều tham số khác ảnh hưởng đến mô hình.

Một vấn đề khác của kiểm thử thống kê, đặc biệt với độ tin cậy của phần mềm, là cần kiểm thử rất kỹ mới đạt được mức độ tin cậy mong muốn. Ví dụ với độ tin cậy 99,9999, tức là trong một triệu lần mới có một lần trục trặc, và giả sử mỗi lần chạy mất một

giây thì trong một năm hệ thống chỉ gây vấn đề 31 lần. Sẽ không dễ dàng để khẳng định độ tin cậy này bằng kiểm thử, và đôi khi là không thể thực hiện nếu trục trặc này gây thiệt hại lớn, ví dụ như tính mạng con người.

Một cách tiếp cận không hình thức được sử dụng nhiều trên thực tế là kiểm thử với người dùng thật. Với cách này chúng ta sẽ chọn một nhóm người cho dùng thử sản phẩm để phát hiện các vấn đề và phản hồi của người dùng về hệ thống. Các đợt kiểm thử này thường được gọi là kiểm thử al-pha hay bê-ta. Kiểm thử al-pha thường thực hiện bởi đội phát triển là chính và ở ngay tại đơn vị phát triển. Kiểm thử be-ta sẽ do người dùng thật dùng thử và đánh giá ở tại máy của người dùng và nơi làm việc, môi trường thật.

10.5 Kiểm thử hồi quy

10.5.1 Giới thiệu

Khi xây dựng một phiên bản mới của hệ thống (ví dụ sau khi sửa lỗi, thay đổi hoặc thêm bớt một số chức năng, v.v.), chúng ta có thể vô tình gây ra các lỗi mới (vô tình cấy thêm các lỗi mới). Nhiều khi thay đổi rất nhỏ cũng gây những vấn đề lớn ngoài sức tưởng tượng của đội phát triển. Để tránh những việc đáng tiếc này xảy ra chúng ta cần kiểm thử lại phần mềm để đảm bảo các chức năng đã chạy tốt vẫn tiếp tục chạy tốt.

Khi phiên bản mới của phần mềm không còn hoạt động như trước chúng ta nói là phiên bản mới *hồi quy* với bản trước đó. Phiên bản mới không hồi quy tức là nó vẫn giữ được các tính năng là một yêu cầu chất lượng cơ bản. Các hoạt động kiểm thử tập trung vào vấn đề hồi quy được gọi là kiểm thử hồi quy. Đúng ra chúng ta cần kiểm tra phiên bản mới không hồi quy về phiên bản cũ, nhưng thuật ngữ kiểm thử hồi quy đã phổ biến nên chúng ta dùng thuật ngữ này.

Phương pháp đơn giản để kiểm thử hồi quy là chạy lại tất cả các ca kiểm thử của phiên bản trước. Tuy nhiên việc kiểm lại toàn bộ này rất tốn kém và không tầm thường. Các ca kiểm thử trước đó có thể không chạy lại được với phiên bản mới của phần mềm mà không sửa đổi gì. Chạy lại toàn bộ các ca kiểm thử là rất tốn kém và nhiều phần trong đó là không cần thiết. Một bộ kiểm thử chất lượng tốt phải được duy trì xuyên suốt các phiên bản của hệ thống.

Thay đổi trong phiên bản phần mềm mới có thể tác động tới khuôn dạng của đầu vào và đầu ra, và các ca kiểm thử có thể cần các thay đổi tương ứng để chạy được. Ngay cả việc sửa đổi một cách đơn giản của cấu trúc dữ liệu, chẳng hạn như việc bổ sung các trường hay sự thay đổi nhỏ của các loại dữ liệu, có thể làm các ca kiểm thử trước đây không chạy được nữa. Hơn nữa, một số ca kiểm thử có thể bị lỗi thời, khi các tính năng của phần mềm đã thay đổi hoặc bị loại bỏ khỏi phiên bản mới.

Các khung hỗ trợ để dịch đặc tả kiểm thử thành ca kiểm thử sẽ giúp giảm ảnh hưởng của thay đổi định dạng của đầu vào và đầu ra. Các đặc tả ca kiểm thử và các mệnh đề về kết quả mong đợi mà phản ánh tính đúng của ca kiểm thử và tránh các chi tiết cụ thể sẽ giảm một phần lớn công sức kiểm thử khi có thay đổi nhỏ.

Các bộ kiểm thử chất lượng cao có thể được duy trì qua các phiên bản của phần mềm bằng việc xác định và loại bỏ các ca kiểm thử lỗi thời, và bằng việc phát hiện và đánh dấu thích hợp các ca kiểm thử dư thừa. Ca kiểm thử đi cùng một đường đi trong chương trình là dư thừa với tiêu chuẩn kiểm thử cấu trúc, nhưng ca kiểm thử cùng một phân hoạch lại là dư thừa với kiểm thử dựa trên lớp tương đương. Việc dư thừa là do nhiều người cùng làm hoặc do chương trình bị thay đổi gây ra. Các ca kiểm thử dư thừa không ảnh hưởng đến tính đúng, sai, mà chỉ ảnh hưởng đến chi phí kiểm thử. Chúng không phát hiện được lỗi nhưng làm tăng chi phí kiểm thử. Các ca kiểm thử lỗi thời cần phải loại bỏ, vì chúng ta không

còn cần chúng còn những ca kiểm thử dư thừa chúng ta có thể giữ, vì chúng giúp phần mềm phát triển trong tương lai đảm bảo chất lượng hơn, tuy nhiên nếu chúng gây ra chi phí kiểm thử lớn quá mức cần thiết thì có thể loại bỏ.

10.5.2 Kỹ thuật kiểm thử hồi quy

Ngay cả khi chúng ta có thể xác định và loại trừ ca kiểm thử lỗi thời, số lượng các kiểm thử được thực hiện lại có thể là lớn. Có thể mất cả ngày thậm chí cả tuần để chạy lại tất cả các ca kiểm thử với các sản phẩm lớn. Một số ca kiểm thử lại không tự động được thì kiểm thử hồi quy còn phụ thuộc vào nguồn lực, hoặc một số kiểm thử lại đòi hỏi máy móc đắt đỏ hay điện thoại di động đã lạc hậu thì không dễ mua để thực hiện. Chi phí cho việc thực hiện lại một bộ kiểm thử có thể được giảm bằng cách chọn một tập hợp các ca kiểm thử được thực hiện lại, bỏ qua các ca kiểm thử không liên quan hoặc ưu tiên thực hiện một tập con của bộ kiểm thử liên quan đến thay đổi.

Thứ tự ưu tiên của các ca kiểm thử cũng kéo theo tần suất nó được thực hiện. Do đó chúng ta cần dựa trên đặc tả hoặc mã nguồn để có thể sắp xếp các ca kiểm thử theo một mức độ ưu tiên. Ngoài ra có thể dựa trên nhiều yếu tố khác như lịch sử lỗi của các dự án tương tự, của những người lập trình, tính chất của ngôn ngữ lập trình, v.v. để giúp thứ tự ưu tiên này có thể phát hiện ra các lỗi nhanh hơn, giảm chi phí kiểm thử hồi quy.

Về mặt hình thức, giả sử f là phiên bản cũ của một hàm hoặc một chương trình và f' là phiên bản mới, và giả sử T là bộ kiểm thử cho hàm f. Quy trình cơ bản của kiểm thử hồi quy theo [RH96] gồm các bước sau:

1. Xác định $T' \subseteq T$ để kiểm thử f'.

2. Kiểm thử f' với T_1 để xem f' còn đúng với T_1 không.

3. Nếu cần, xây dựng T_2 gồm các ca kiểm thử mới theo đặc tả chức năng hoặc theo cấu trúc mới của f'.

4. Kiểm thử f' với T_2 để xem f' đúng với T_2 chưa.

5. Tạo bộ kiểm thử T' cho f' từ T, T_1 và T_2.

Quy trình này đặt ra bốn bài toán cho kiểm thử hồi quy. Bước 1 là bài toán chọn ca kiểm thử cần kiểm tra lại, còn gọi là bài toán lựa chọn kiểm thử hồi quy (regression test problem). Bước 2 và bước 4 là bài toán tối ưu để giảm chi phí thực hiện lại kiểm thử. Bước 3 là bài toán đảm bảo chất lượng hoặc tiêu chuẩn bao phủ của bộ kiểm thử. Bước 5 là bài toán cập nhật và lưu trữ thông tin kiểm thử. Mặc dù các bài toán này đều quan trọng, chúng ta sẽ tập trung vào bài toán 1 là lựa chọn các ca kiểm thử cần kiểm tra lại.

Kỹ thuật lựa chọn kiểm thử hồi quy thường dựa trên mã nguồn hoặc đặc tả. Kỹ thuật dựa trên mã nguồn chọn ca kiểm thử mà sẽ chạy qua phần mã nguồn thay đổi. Kỹ thuật dựa trên đặc tả sẽ chọn ca kiểm thử để thực hiện nếu nó liên quan đến phần đặc tả bị thay đổi. Kỹ thuật dựa trên mã nguồn, có thể dễ thực hiện hơn nhờ một số công cụ, và chúng ta có thể thực hiện độc lập với đặc tả. Trái lại, kỹ thuật dựa trên đặc tả dễ mở rộng và dễ áp dụng cho các thay đổi liên quan đến một loạt mô-đun. Tuy nhiên kỹ thuật dựa trên đặc tả lại khó tự động hóa hơn và đòi hỏi đặc tả được quản lý tốt.

Trong các kỹ thuật lựa chọn dựa trên mã nguồn, kỹ thuật kiểm thử dòng điều khiển dựa vào một bản ghi của các phần tử chương trình (ví dụ dòng lệnh) đã được thực hiện cho mỗi ca kiểm thử. Việc xác định bản ghi này có thể thực hiện bằng kỹ thuật chèn mã (instrumentation). Lần kiểm thử sau đó sẽ dựa trên các phần tử chương trình bị thay đổi để chọn các ca kiểm thử sẽ cần thực hiện lại. Với kỹ thuật này chúng ta có thể sử dụng các tiêu chuẩn kiểm thử khác nhau, như dựa trên luồng điều khiển hay luồng dữ liệu.

Các kỹ thuật hồi quy đồ thị luồng điều khiển (CFG) dựa trên sự khác biệt giữa CFG của phiên bản phần mềm mới và phiên bản cũ trước đó. Tập các đường đi của hai đồ thị được so sánh và chúng ta chạy lại tất cả các ca kiểm thử ở các đường đi mới. Các kỹ thuật kiểm thử hồi quy luồng dữ liệu (DF) chọn các ca kiểm thử cho các cặp DU bị thay đổi. Các ca kiểm thử được chọn thực hiện lại là các ca kiểm thử chạy qua các cặp DU của chương trình gốc mà các cặp này bị thay đổi hoặc xóa đi trong phiên bản mới. Các ca kiểm thử chạy qua các lệnh điều kiện mà mệnh đề điều kiện bị thay đổi cũng được chọn vì các thay đổi này có thể thay đổi các cặp DU.

Trái với các kỹ thuật dựa trên mã nguồn, kỹ thuật lựa chọn kiểm thử dựa trên đặc tả không yêu cầu ghi lại các đường đi mà các ca kiểm thử đi qua. Ca kiểm thử hồi quy có thể được xác định từ liên hệ giữa các ca kiểm thử và các mục trong đặc tả. Ví dụ với các ca kiểm thử chức năng sử dụng kỹ thuật lớp tương đương, khi đặc tả thay đổi thì các ca kiểm thử của các lớp này cũng thay đổi theo và chúng ta cần kiểm thử lại chúng và xác định lại các ca kiểm thử nếu cần. Trong một số trường hợp khi chúng ta có thể sinh các ca kiểm thử tự động, như kỹ thuật kiểm thử dựa trên biên thì chúng ta đơn giản chạy lại việc sinh các ca kiểm thử khi đặc tả thay đổi.

10.6 Tổng kết

Chương này đã giới thiệu ba pha kiểm thử vào giai đoạn cuối của quy trình phát triển phần mềm (khi hoàn thành sản phẩm) gồm: kiểm thử hệ thống, kiểm thử chấp thuận và kiểm thử hồi quy. Các hoạt động kiểm thử ở các giai đoạn này có vai trò quyết định với việc phần mềm có được đưa vào sử dụng hay không (sản phẩm có được chấp nhận bởi khách hàng hay không). Đây cũng là các pha kiểm thử tốn kém nhiều công sức và thời gian nhất so với các hoạt động ứng với các mức kiểm thử còn lại. Các xu hướng phát triển

phần mềm hiện đại ngày nay đang tập trung tự động hóa kiểm thử ở các bước này giúp giảm chi phí, công sức, và tăng độ tin cậy của cả nhà phát triển và người sử dụng vào chất lượng sản phẩm.

10.7 Bài tập

1. Hãy áp dụng các kỹ thuật tích hợp cho hệ thống SATM trình bày ở Chương 2?

2. Thảo luận ưu và nhược điểm của tích hợp từ trên xuống và từ dưới lên.

3. Hãy đưa ra các ví dụ minh họa về các loại lỗi giao diện.

4. Hãy tìm thêm và mô tả ba loại lỗi giao diện ngoài những loại đã nêu trong chương và cho ví dụ minh họa.

5. Thảo luận ưu nhược điểm của tích hợp dựa trên cấu trúc mô-đun và dựa trên đồ thị gọi hàm.

6. Phương pháp tích hợp nào giúp khoanh vùng lỗi dễ nhất? Vì sao?

7. Phương pháp tích hợp nào giúp giảm hàm gọi giả nhất?

8. Khi nào và tại sao phải chuyển việc kiểm thử từ đội phát triển sang đội kiểm thử độc lập? Khi nào không nên sử dụng đội kiểm thử độc lập?

9. Tìm một số tính chất không thể kiểm chứng bằng kiểm thử hệ thống, và chỉ ra cách kiểm chứng chúng.

10. Viết phiên bản mới của các chương trình ví dụ trong Chương 2 và áp dụng kỹ thuật kiểm thử hồi quy luồng dữ liệu và luồng điều khiển để xác định các ca kiểm thử cần chạy lại.

11. Với kỹ thuật kiểm thử sử dụng bảng quyết định thì kiểm thử hồi quy sẽ chọn các ca kiểm thử như thế nào?

12. Với mỗi kỹ thuật kiểm thử cấu trúc hãy áp dụng kiểm thử hồi quy khi chương trình được thêm lệnh, xóa bớt lệnh, và sửa lệnh không thay đổi cấu trúc? (xét hai trường hợp thêm/xóa/sửa lệnh điều khiển và lệnh tính toán).

Tài liệu tham khảo

[ADP93] Topper A., Ouellette D., and Jorgensen P., *Structured methods: Merging models, techniques, and case*, McGraw-Hill, 1993.

[AJ83] Clarke Lori A. and Richardson Debra J., *The application of error-sensitive testing strategies to debugging*, SIGSOFT Softw. Eng. Notes **8** (1983), no. 4, 45–52.

[AJ84] _____, *A reply to foster's "comment on 'the application of error-sensitive testing strategies to debugging'"*, SIGSOFT Softw. Eng. Notes **9** (1984), no. 1, 24–28.

[AJ00] Abdurazik Aynur and Offutt Jeff, *Using uml collaboration diagrams for static checking and test generation*, Proceedings of the 3rd international conference on The unified modeling language: advancing the standard (Berlin, Heidelberg), UML'00, Springer-Verlag, 2000, pp. 383–395.

[AK04a] Hartman A. and Nagin K., *The agedis tools for model based testing*, Proceedings of the 2004 ACM SIGSOFT international symposium on Software testing

and analysis (New York, NY, USA), ISSTA '04, ACM, 2004, pp. 129–132.

[AK04b] _____, *The agedis tools for model based testing*, SIGSOFT Softw. Eng. Notes **29** (2004), no. 4, 129–132.

[AW93] K. Agrawal and James A. Whittaker, *Experiences in applying statistical testing to a real-time, embedded software system*, the Pacific Northwest Software Quality Conference, 1993.

[BBH05] F. Belli, C.J. Budnik, and A. Hollman, *A holistic approach to testing of interactive systems using statecharts*, 2nd South-East European Workshop on Formal Methods (SEEFM 05), Los Angeles, 2005, pp. 1–15.

[Bec02] Beck, *Test driven development: By example*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[BFM04] Legeard Bruno, Peureux Fabien, and Utting Mark, *Controlling test case explosion in test generation from b formal models: Research articles*, Softw. Test. Verif. Reliab. **14** (2004), no. 2, 81–103.

[Bil88] Hetzel Bill, *The complete guide to software testing*, 2nd ed., QED Information Sciences, Inc., Wellesley, MA, USA, 1988.

[BL75] J. R. Brown and M. Lipov, *Testing for software reliability*, the International Symposium on Reliable Software, Los Angeles, 1975, pp. 518–527.

[Bor84] Beizer Boris, *Software system testing and quality assurance*, Van Nostrand Reinhold Co., New York, NY, USA, 1984.

[Bor95] ———, *Black-box testing: techniques for functional testing of software and systems*, John Wiley & Sons, Inc., New York, NY, USA, 1995.

[BP84] Victor R. Basili and Barry T. Perricone, *Software errors and complexity: An empirical investigation.*, Commun. ACM **27** (1984), no. 1, 42–52.

[C.79] Huang J. C., *Detection of data flow anomaly through program instrumentation*, IEEE Trans. Softw. Eng. **5** (1979), no. 3, 226–236.

[D.95] Adams D., *The hitchhiker's guide to the galaxy*, San Val, 1995.

[Dav88] Harel David, *On visual formalisms*, Commun. ACM **31** (1988), no. 5, 514–530.

[DJ76] Fosdick Lloyd D. and Osterweil Leon J., *Data flow analysis in software reliability*, ACM Comput. Surv. **8** (1976), no. 3, 305–330.

[fS91] International Organization for Standardization, *Data elements and interchange formats—information interchange—representation of dates and times*, International standard iso 8601:1988, technical corrigendum 1, switzerland, International Organization for Standardization, 1991.

[G.95] Leveson Nancy G., *Safeware: system safety and computers*, ACM, New York, NY, USA, 1995.

[GG12] Markus Grtner and Markus Grtner, *Atdd by example: A practical guide to acceptance test-driven development*, 1st ed., Addison-Wesley Professional, 2012.

[GJ88] Frankl P. G. and Weyuker E. J., *An applicable family of data flow testing criteria*, IEEE Trans. Softw. Eng. **14** (1988), no. 10, 1483–1498.

[GOA05] Mats Grindal, Jeff Offutt, and Sten F. Andler, *Combination testing strategies: a survey*, Softw. Test., Verif. Reliab. **15** (2005), no. 3, 167–199.

[Gru73] F. Gruenberger, *Program testing: The historical perspective*, Program Test Methods, Prentice-Hall, 1973, pp. 11–14.

[IEE93] Computer Society IEEE, *Ieee standard classification for software anomalies*, Ieee standard 1044-1993, IEEE Computer Society, 1993.

[Ing61] Stuart J. Inglis, *Planets, stars, and galaxies*, 4th ed., Wiley and Sons, New York, NY, USA, 1961.

[Jor13] Paul C. Jorgensen, *Software testing: A craftman's approach*, 4th ed., CRC Press, Inc., Boca Raton, FL, USA, 2013.

[Kar12] Michael Kart, *Behavior-driven development: conference tutorial*, J. Comput. Sci. Coll. **27** (2012), no. 4, 75–75.

[KJ02] El-Far I. K. and Whittaker J.A., *Model-based software testing*, Encyclopedia of Software Engineering (2002), 825—-837.

[KN08] Priyadarshi Tripathy Kshirasagar Naik, *Software testing and quality assurance: Theory and practice*, John Wiley and Sons, Inc., 2008.

[Kru05] Steve Krug, *Don't make me think: A common sense approach to the web (2nd edition)*, New Riders Publishing, Thousand Oaks, CA, USA, 2005.

[KWG04] D. Richard Kuhn, Dolores R. Wallace, and Albert M. Gallo, *Software fault interactions and implications for software testing*, IEEE Trans. Software Eng. **30** (2004), no. 6, 418–421.

[LB12] Rick Kazman Len Bass, Paul Clements, *Software architecture in practice (3rd edition)*, Addison-Wesley Professional, 2012.

[Lee03] Copeland Lee, *A practitioner's guide to software test design*, Artech House, Inc., Norwood, MA, USA, 2003.

[Mal87] Chellappa Mallika, *Nontraversible paths in a program*, IEEE Trans. Softw. Eng. **13** (1987), no. 6, 751–756.

[Mar81] Weiser Mark, *Program slicing*, Proceedings of the 5th international conference on Software engineering (Piscataway, NJ, USA), ICSE '81, IEEE Press, 1981, pp. 439–449.

[Mar84] ———, *Program slicing*, IEEE Trans. Softw. Eng. **10** (1984), no. 4, 352–357.

[Mar03] Robert Cecil Martin, *Agile software development: Principles, patterns, and practices*, Prentice Hall PTR, Upper Saddle River, NJ, USA, 2003.

[McC76a] Thomas J. McCabe, *A complexity measure*, Proceedings of the 2Nd International Conference on Software Engineering (Los Alamitos, CA, USA), ICSE '76, IEEE Computer Society Press, 1976, pp. 407–.

[McC76b] T.J. McCabe, *A complexity measure*, IEEE Transactions on Software Engineering **2** (1976), no. 4, 308–320.

[MFB+07] J. Meier, Carlos Farre, Prashant Bansode, Scott Barber, and Dennis Rea, *Performance testing guidance for web applications: patterns & practices*, Microsoft Press, Redmond, WA, USA, 2007.

[MG04] J. Mayer and R. Guderlei, *Test oracles using statistical methods*, the First International Workshop on Software Quality, 2004, pp. 179–189.

[Mil91] Edward F. Miller, *Automated software testing: A technical perspective*, American Programmer **4** (1991), no. 4, 38–43.

[MMA02] Vanmali M., Last M., and Kandel A., *Using a neural network in the software testing process*, International Journal of Intelligent Systems **17** (2002), no. 1, 45–62.

[MMA04] Last M., Friendman M., and Kandel A., *Using data mining for automated software testing*, International Journal of Software Engineering and Knowledge Engineering **14** (2004), no. 4, 369–393.

[Mol09] Ian Molyneaux, *The art of application performance testing: Help for programmers and quality assurance*, 1st ed., O'Reilly Media, Inc., 2009.

[MRA04] Blackburn M., Busser R., and Nauman A., *Why model-based test automation is different and what you should know to get started*, Proceedings of International Conference on Practical Software Quality and Testing, PSQT'04, 2004.

[Mye75] Glenford J. Myers, *The art of software testing*, Wiley Interscience, New York, NY, USA, 1975.

[oST02a] National Institute of Standards and Technology, *The economic impacts of inadequate infrastructure for software testing*, Nist planning report, National Institute of Standards and Technology, 2002.

[oST02b] ———, *Software errors cost u.s. economy $59.5 bullion annually*, Nist news release, National Institute of Standards and Technology, 2002.

[PE85] D. E. Perry and W. M. Evangelist, *An empirical study of software interface faults*, Proceedings of the Twentieth Annual Hawaii International Conference on Systems Sciences, January 1987, Volume II, 1985, pp. 113–126.

[Pos90] Robert M. Poston, *Automated software testing*, Workshop Programming Environments (NJ, USA), Inc. Tinton Falls, 1990.

[Pos91] ———, *A complete toolkit for the software tester*, American Programmer **4** (1991), no. 4, 28–37.

[PZKH06] Hu Peifeng, Zhang Zhenyu, Chan W. K., and Tse T. H., *An empirical comparison between direct and indirect test result checking approaches*, Proceedings of the 3rd international workshop on Software quality

assurance (New York, NY, USA), SOQUA '06, ACM, 2006, pp. 6–13.

[RH96] Gregg Rothermel and Mary Jean Harrold, *Analyzing regression test selection techniques*, IEEE Trans. Softw. Eng. **22** (1996), no. 8, 529–551.

[Rob85] Mandl Robert, *Orthogonal latin squares: an application of experiment design to compiler testing*, Commun. ACM **28** (1985), no. 10, 1054–1058.

[Rub94] Jeffrey Rubin, *Handbook of usability testing: How to plan, design, and conduct effective tests*, John Wiley & Sons, Inc., New York, NY, USA, 1994.

[S.82] Pressman Roger S., *Software engineering: A practitioner's approach*, McGraw-Hill, Inc., New York, NY, USA, 1982.

[Shr89] Phadke Madhav Shridhar, *Quality engineering using robust design*, Englewood Cliffs, N.J. Prentice Hall, 1989.

[SJ85] Rapps Sandra and Weyuker Elaine J., *Selecting software test data using data flow information*, IEEE Trans. Softw. Eng. **11** (1985), no. 4, 367–375.

[Som10] Ian Sommerville, *Software engineering*, 9 ed., Addison-Wesley, 2010.

[SvBGF+91] Fujiwara Susumu, von Bochmann Gregor, Khendek Ferhat, Amalou Mokhtar, and Ghedamsi Abderrazak, *Test selection based on finite state models*, IEEE Trans. Softw. Eng. **17** (1991), no. 6, 591–603.

[TL02] Kuo-Chung Tai and Yu Lei, *A test generation strategy for pairwise testing*, IEEE Trans. Software Eng. **28** (2002), no. 1, 109–111.

[WM96] Arthur H. Watson and Thomas J. McCabe, *Structured testing: A testing methodology using the cyclomatic complexity metric*, NIST Special Publication 500-235, National Institute of Standards and Technology, Computer Systems Laboratory, NIST, Gaithersburg, MD 20899-0001, 1996.

Sơ lược về các tác giả

TS. Phạm Ngọc Hùng

TS. Phạm Ngọc Hùng[1] tốt nghiệp đại học ngành Công nghệ Thông tin tại Khoa Công nghệ (nay là Trường Đại học Công nghệ), Đại học Quốc gia Hà Nội (ĐHQGHN) năm 2002, tốt nghiệp Thạc sĩ chuyên ngành Công nghệ Phần mềm tại Viện Khoa học và Công nghệ tiên tiến Nhật Bản năm 2006, bảo vệ thành công học vị Tiến sĩ chuyên ngành Công nghệ Phần mềm tại Viện Khoa học và Công nghệ tiên tiến Nhật Bản năm 2009. TS. Phạm Ngọc Hùng bắt đầu sự nghiệp nghiên cứu từ năm 2002 tại Khoa Công nghệ, ĐHQGHN. Từ năm 2009 đến nay, Tiến sĩ là giảng viên tại Bộ môn Công nghệ Phần mềm, Khoa Công nghệ Thông tin, Trường Đại học Công nghệ, ĐHQGHN. Các hướng nghiên cứu quan tâm của TS. Phạm Ngọc Hùng gồm các phương pháp hình thức cho phát triển phần mềm, mô hình hoá, đặc tả và kiểm chứng phần mềm, kiểm chứng đảm bảo giả định (assume-guarantee verification), kiểm thử tự động và tiến hóa phần mềm. TS. Phạm Ngọc Hùng đã công bố hơn 30 bài báo trong lĩnh vực chuyên môn của mình trong các tạp chí và kỷ yếu của các hội nghị quốc tế và trong nước. Ngoài các hoạt động chuyên môn, TS. Phạm Ngọc Hùng rất tích cực trong các hoạt động

[1] http://uet.vnu.edu.vn/~hungpn/

xã hội nhằm quảng bá việc học tập, nghiên cứu và ứng dụng Công nghệ Thông tin cho giới trẻ. TS đã vinh dự nhận được nhiều giải thưởng cao quý trong lĩnh vực Công nghệ Thông tin.

TS. Trương Anh Hoàng

TS. Trương Anh Hoàng[1] tốt nghiệp cử nhân tin học năm 1994 và thạc sỹ năm 2001 tại khoa Toán-Cơ-Tin, trường Đại học Tổng hợp Hà Nội (nay là Trường Khoa học Tự nhiên, Đại học Quốc gia Hà Nội). Sau khi tốt nghiệp đại học ông đã tham gia nhiều dự án phát triển phần mềm của nhiều công ty trong nước và của nước ngoài qua đó ông thu được nhiều kinh nghiệm thực tế làm phần mềm.

Năm 2002 ông sang Na Uy làm nghiên cứu sinh và năm 2006 bảo vệ thành công học vị Tiến sĩ khoa học ngành Tin học tại Trường Đại học Bergen, Na Uy. Sau khi về nước năm 2006 ông tiếp tục làm cho một công ty chuyên về phần mềm cho điện thoại di động. Từ năm 2008 đến nay ông là giảng viên Khoa Công nghệ Thông tin, Trường Đại học Công nghệ, Đại học Quốc gia Hà Nội. Các môn giảng dạy chính của ông là Công nghệ Phần mềm và Kiểm thử Phần mềm. Ông quan tâm nghiên cứu về phân tích chương trình, kiểm chứng phần mềm và kiểm thử tự động và quy trình phần mềm.

TS. Đặng Văn Hưng

TS. Đặng Văn Hưng[2] tốt nghiệp ngành Toán học tính toán tại Khoa Toán-Cơ, Trường Đại học Tổng hợp Hà Nội (nay thuộc Trường đại học Khoa học Tự nhiên, Đại học Quốc gia Hà Nội) năm 1977, bảo vệ thành công học vị Tiến sĩ ngành Khoa học máy tính tại

[1] http://uet.vnu.edu.vn/~hoangta/
[2] http://uet.vnu.edu.vn/~dvh/

Viện Khoa học Máy tính và Tự động hóa, Viện Hàn lâm Khoa học Hungary tháng 2 năm 1988. TS. Đặng Văn Hưng khởi đầu sự nghiệp chuyên môn của mình là một nghiên cứu viên tại viện Công nghệ Thông tin, Viện Khoa học và Công nghệ Việt Nam từ năm 1978, và làm nghiên cứu viên của Viện Quốc tế về Kỹ nghệ Phần mềm thuộc Trường đại học Liên hiệp quốc từ năm 1995 đến năm 2007. Từ năm 2008 đến nay TS. Đặng Văn Hưng là giảng viên cao cấp của Khoa Công nghệ Thông tin, Trường Đại học Công nghệ, Đại học Quốc gia Hà Nội.

TS. Đặng Văn Hưng đã tham gia tích cực vào một số hoạt động khoa học như giảng dạy tại một số trường hè quốc tế, tham gia các tiểu ban chương trình của nhiều hội nghị quốc tế về lĩnh vực công nghệ thông tin, hiệu đính sách và tạp chí quốc tế cho nhà xuất bản Springer. Lĩnh vực nghiên cứu mà TS. Đặng Văn Hưng quan tâm bao gồm các phương pháp hình thức, mô hình hoá, đặc tả và kiểm chứng phần mềm, các hệ tính toán song song và phân tán, kỹ thuật phát triển phần mềm dựa trên thành phần. TS. Đặng Văn Hưng đã công bố hơn 80 bài báo trong lĩnh vực chuyên môn của mình trong các tạp chí và kỷ yếu của các hội nghị quốc tế và trong nước. TS. Đặng Văn Hưng đã từng là giáo sư mời của Đại học Sư phạm Hoa Đông, Thượng Hải, Trung Quốc, và hiện nay là Phó Tổng biên tập Chuyên san Công nghệ Thông tin và Truyền thông của Đại học Quốc gia Hà Nội.