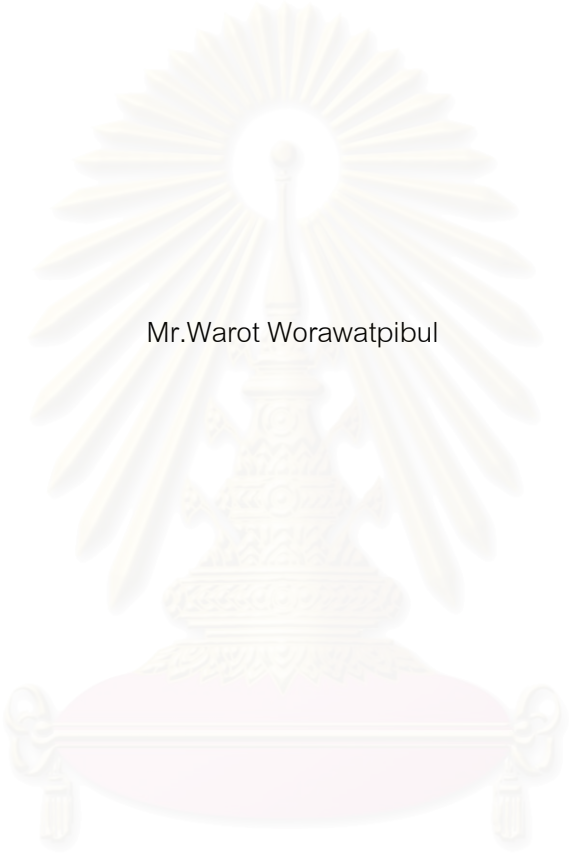
การวิเคราะห์มิวเทชันแบบหลายข้อผิดพลาด

นายวรท วรวัฒน์พิบูลย์

วิทยานิพนธ์นี้เป็นส่วนหนึ่งของการศึกษาตามหลักสูตรปริญญาวิศวกรรมศาสตรมหาบัณฑิต
สาขาวิชาวิศวกรรมคอมพิวเตอร์ ภาควิชาวิศวกรรมคอมพิวเตอร์
คณะวิศวกรรมศาสตร์ จุฬาลงกรณ์มหาวิทยาลัย
ปีการศึกษา 2546
ISBN 974-17-5815-4

MULTIPLE-FAULT MUTATION ANALYSIS

Mr.Warot Worawatpibul



A Thesis Submitted in Partial Fulfillment of the Requirements

for the Degree of Master of Engineering in Computer Engineering

Department of Computer Engineering

Faculty of Engineering

Chulalongkorn University

Academic Year 2003

ISBN 974-17-5815-4

หัวข้อวิทยานิพนธ์	การวิเคราะห์มิวเมชันแบบหลายข้อผิดพลาด
โดย	นายวรท วรวัฒน์พิบูลย์
สาขาวิชา	วิศวกรรมคอมพิวเตอร์
อาจารย์ที่ปรึกษา	อาจารย์ ดร. อรรถสิทธิ์ สุรฤกษ์

คณะวิศวกรรมศาสตร์ จุฬาลงกรณ์มหาวิทยาลัย อนุมัติให้นับวิทยานิพนธ์ฉบับนี้เป็นส่วนหนึ่งของการศึกษาตามหลักสูตรปริญญามหาบัณฑิต

…….. คณบดีคณะวิศวกรรมศาสตร์
(ศาสตราจารย์ ดร.ดิเรก ลาวัณย์ศิริ)

คณะกรรมการสอบวิทยานิพนธ์

…… ประธานกรรมการ
(รองศาสตราจารย์ ดร.ประภาส จงสถิตย์วัฒนา)

…….. อาจารย์ที่ปรึกษา
(อาจารย์ ดร.อรรถสิทธิ์ สุรฤกษ์)

…… กรรมการ
(ผู้ช่วยศาสตราจารย์ ดร.ธาราทิพย์ สุวรรณศาสตร์)

…… กรรมการ
(ผู้ช่วยศาสตราจารย์ ดร.อานนท์ รุ่งสว่าง)

วรท วรวัฒน์พิบูลย์ : การวิเคราะห์มิวเทชันแบบหลายข้อผิดพลาด. (MULTIPLE-FAULT MUTATION ANALYSIS) อ. ที่ปรึกษา: อาจารย์ ดร.อรรถสิทธิ์ สุรฤกษ์, 115 หน้า. ISBN 974-17-5815-4.

งานวิจัยชิ้นนี้กล่าวถึงการปรับปรุงประสิทธิภาพการวิเคราะห์มิวเทชันซึ่งเป็นวิธีการทดสอบซอฟต์แวร์แบบหน่วยเดียว ข้อเสียของการวิเคราะห์มิวเทชันคือใช้เวลาในการคำนวณที่สูง เนื่องจากจำนวนโปรแกรมมิวแทนท์ที่ต้องนำมาทดสอบมีจำนวนมาก จุดมุ่งหมายสำหรับงานวิจัยชิ้นนี้คือ การลดจำนวนของโปรแกรมมิวแทนท์และลดจำนวนครั้งในการทดสอบโปรแกรมมิวแทนท์ แต่ยังรักษาประสิทธิผลของการทดสอบไม่ให้ลดลง

การวิเคราะห์มิวเทชันแบบหลายข้อผิดพลาดเป็นวิธีการที่ลดจำนวนของโปรแกรมมิวแทนท์ลง โดยการให้โปรแกรมมิวแทนท์หนึ่งๆ สามารถเป็นตัวแทนให้กับหลายโปรแกรมมิวแทนท์ได้ ในงานวิจัยชิ้นนี้ได้เสนออัลกอริธึมในการสร้างโปรแกรมมิวแทนท์แบบหลายข้อผิดพลาด และอัลกอริธึมในการกำจัดโปรแกรมมิวแทนท์แบบหลายข้อผิดพลาด งานวิจัยชิ้นนี้ยังได้พิสูจน์และทำการทดลองเพื่อยืนยันถึงจำนวนของโปรแกรมมิวแทนท์ที่ลดลงและการรักษาประสิทธิผลของกรณีทดสอบที่ได้รับ

นอกจากปัญหาที่กล่าวถึงข้างต้น ในการวิเคราะห์มิวเทชันแบบเดิมนั้นกรณีทดสอบจำเป็นต้องทดสอบกับโปรแกรมมิวแทนท์ที่ไม่สามารถถูกกำจัดได้ด้วยกรณีทดสอบนั้นๆ งานวิจัยนี้จึงเสนออัลกอริธึมในการแบ่งกลุ่มโปรแกรมมิวแทนท์ตามข้อบังคับการไปถึงเพื่อลดจำนวนครั้งของการทดสอบที่สูญเสียไป พร้อมทั้งพิสูจน์ถึงจำนวนครั้งในการทดสอบโปรแกรมมิวแทนท์ที่สามารถลดลงได้เมื่อทำการแบ่งกลุ่ม

งานวิจัยชิ้นนี้ยังได้พิจารณาถึงปัจจัยที่สำคัญสำหรับการนำเอาการแบ่งกลุ่มโปรแกรมมิวแทนท์มาปรับปรุงการวิเคราะห์มิวเทชันแบบหลายข้อผิดพลาด

ภาควิชา.....วิศวกรรมคอมพิวเตอร์..... ลายมือชื่อนิสิต.....
สาขาวิชา.....วิศวกรรมคอมพิวเตอร์..... ลายมือชื่ออาจารย์ที่ปรึกษา.....
ปีการศึกษา...2546...

4570513721 : MAJOR COMPUTER ENGINEER

KEY WORD: SOFTWARE TESTING / MUTATION ANALYSIS / PROGRAM SLICING / CONSTRAINT-BASED TESTING / TEST CASE GENERATION

WAROT WORAWATPIBUL: MULTIPLE-FAULT MUTATION ANALYSIS, THESIS ADVISOR: ATHASIT SURARERKS, Ph.D., 115 pp. ISBN 974-17-5815-4.

This research concerns improving the efficiency of mutation analysis technique, which is an effective software unit testing. The main disadvantage of mutation analysis is to use high computation because of a large number of mutant programs to be tested. The aims of this research are not only to reduce the number of mutants and the amount of testing mutant programs, but also to maintain the effective level of the testing.

Multiple-Fault mutation is a technique to reduce the number of mutants by using a mutant that can represent several mutants. This research proposes an algorithm for creating multiple-fault mutants and an algorithm for killing such mutants. We also prove theorems and carry out experiments to assure that the number of mutants is reduced and the effectiveness of test cases is preserved.

In additional to the mentioned problem, in a classical approach, many test cases have to run against the mutants which cannot be killed by these test cases. This research proposes an algorithm for grouping mutants with respect to the reachability constraint. We prove that grouping mutants helps reduce the number of testing mutants.

This research also considers factors of improving multiple-fault mutation using the grouping technique.

Department….Computer Engineering…….Student's signature.....
Field of study….Computer Engineering…..Advisor's signature.....
Academic year ...2003.....

กิตติกรรมประกาศ

ข้าพเจ้าใคร่ขอกราบขอบพระคุณอาจารย์ ดร.อรรถสิทธิ์ สุรฤกษ์ อาจารย์ที่ปรึกษาวิทยานิพนธ์ของข้าพเจ้า ที่กรุณาแนะนำให้ความรู้ คำปรึกษา ความช่วยเหลือต่าง ๆ ตลอดจนคอยดูแลการทำวิทยานิพนธ์ของข้าพเจ้าจนสำเร็จลุล่วงลงได้ด้วยดี

ขอกราบขอบพระคุณ รองศาสตราจารย์ ดร.ประภาส จงสถิตย์วัฒนา ซึ่งเป็นประธานกรรมการสอบวิทยานิพนธ์ ผู้ช่วยศาสตราจารย์ ดร.ธาราทิพย์ สุวรรณศาสตร์ และ ผู้ช่วยศาสตราจารย์ ดร.อานนท์ รุ่งสว่าง ซึ่งเป็นกรรมการสอบวิทยานิพนธ์ ที่ได้สละเวลาและให้คำแนะนำต่าง ๆ ที่เป็นประโยชน์อย่างยิ่งต่อการจัดทำวิทยานิพนธ์ฉบับนี้

ขอขอบคุณอาจารย์ทุกท่าน ที่ได้ประสิทธิ์ประสาทวิชาให้กับข้าพเจ้า รวมถึงชี้แนะสิ่งดี ๆ ตลอดเวลาที่ข้าพเจ้าได้ศึกษาเล่าเรียนในระดับมหาบัณฑิต

ท้ายที่สุด ข้าพเจ้าใคร่ขอกราบขอบพระคุณบิดา มารดาของข้าพเจ้า ที่คอยให้กำลังใจและสนับสนุนด้านการเงินแก่ข้าพเจ้าเสมอมา

สารบัญ

หน้า

หน้า

สารบัญ (ต่อ)

หน้า

สารบัญตาราง

หน้า

สารบัญภาพ

หน้า

บทที่ 1
บทนำ

1.1 ความเป็นมาและความสำคัญของปัญหา

ขั้นตอนการทดสอบซอฟต์แวร์ (Software testing) เป็นกระบวนการที่สำคัญในการรับประกันความถูกต้องของซอฟต์แวร์ที่ได้รับการพัฒนา วัตถุประสงค์ของการทดสอบคือ การหาข้อผิดพลาดที่แฝงเร้นอยู่ ถึงอย่างไรก็ตามการทดสอบซอฟต์แวร์โดยส่วนใหญ่นั้นอาจทำได้ไม่สมบูรณ์มากนัก ด้วยสาเหตุมาจากข้อจำกัดทางด้านเวลาที่เสียไปในการทดสอบ ส่งผลให้มีหลายงานวิจัยนำเสนอเทคนิคและอัลกอริธึมเพื่อช่วยลดเวลาในการทดสอบลง ประสิทธิผลของการทดสอบนั้นขึ้นอยู่กับชุดทดสอบ (Test data) ที่ได้รับ ซึ่งชุดทดสอบใดๆ นั้น ประกอบขึ้นจากชุดของกรณีทดสอบ (Set of test cases)

การวิเคราะห์มิวเทชัน (Mutation Analysis) [5] เป็นเทคนิคประเภทหนึ่งของการทดสอบโดยการใส่ข้อผิดพลาด (Fault-based Testing) ที่เป็นการทดสอบแบบหน่วยเดียว (Unit Testing) หลักการคือ พยายามเปลี่ยนแปลงโครงสร้างของโปรแกรมหรือใส่ข้อผิดพลาดเข้าไปในโปรแกรมที่นำมาทดสอบ (Test program) เพื่อสร้างเป็นโปรแกรมมิวแทนท์ (Mutant programs) ต่างๆ ซึ่งเปรียบเสมือนการจำลองข้อผิดพลาดที่อาจเกิดขึ้นกับโปรแกรมทดสอบ โดยรูปแบบของข้อผิดพลาดที่ใส่ไปนั้นถูกนิยามไว้ในตัวดำเนินการมิวเทชัน (Mutation Operator) เป้าหมายของการวิเคราะห์มิวเทชันคือ การสร้างชุดทดสอบที่มีประสิทธิผล (Effectiveness) ในการหาข้อผิดพลาด โดยสามารถแยกความแตกต่างของผลลัพธ์ (Output) ของการประมวลผลด้วยชุดทดสอบเดียวกันระหว่างโปรแกรมมิวแทนท์และโปรแกรมที่นำมาทดสอบ การวิเคราะห์มิวเทชันนั้นได้มีการนำไปประยุกต์ใช้ในการวิจัยอื่นๆ ซึ่งสามารถศึกษารายละเอียดได้จาก [1, 2, 4, 11]

การวิเคราะห์มิวเทชันเป็นเทคนิคที่มีความละเอียดในการทดสอบมาก แต่ยังไม่ได้เป็นที่นิยมอย่างแพร่หลายเนื่องมาจากเสียเวลามากในการประมวลผลแต่ละโปรแกรมมิวแทนท์ งานวิจัยส่วนใหญ่ที่เสนอทางออกในการลดจำนวนโปรแกรมมิวแทนท์นั้นจำเป็นต้องลดประสิทธิผลของชุดทดสอบลงด้วย ข้อแตกต่างที่สำคัญของวิทยานิพนธ์ชิ้นนี้เมื่อเทียบกับวิธีการที่ผ่านมาคือ ศึกษาการลดจำนวนโปรแกรมมิวแทนท์ลงโดยที่ยังคงรักษาประสิทธิผลของการทดสอบไว้ได้เท่าเดิม

1.2 วัตถุประสงค์ของการวิจัย

เพื่อปรับปรุงกรรมวิธีการทำการวิเคราะห์มิวเทชัน ให้ใช้จำนวนครั้งในการทดสอบโปรแกรมมิวแทนท์น้อยลง ด้วยวิธีการดังต่อไปนี้

1.3.1 การสร้างโปรแกรมมิวแทนท์แบบหลายข้อผิดพลาด

1.3.2 การแบ่งกลุ่มโปรแกรมมิวแทนท์

1.3.3 พิจารณาการรวมกันของการสร้างโปรแกรมมิวแทนท์แบบหลายข้อผิดพลาดและการแบ่งกลุ่มโปรแกรมมิวแทนท์

1.3 ขอบเขตของการวิจัย

1.3.1 วิทยานิพนธ์ชิ้นนี้เน้นไปที่การปรับปรุงกระบวนการวิเคราะห์มิวเทชัน โดยมิได้เน้นเรื่องของการสร้างกรณีทดสอบ

1.3.2 การทดสอบเป็นการทดสอบแบบหน่วยเดียว

1.3.3 การวิเคราะห์มิวเทชันเป็นแบบแข็ง (Strong Mutation)

1.3.4 กรณีทดสอบที่สร้างขึ้นมาทดสอบโปรแกรมมีชนิดของข้อมูลเป็นจำนวนเต็มหรือบูเลียนเท่านั้น

1.3.5 ผลลัพธ์ของโปรแกรมมีชนิดข้อมูลเป็นจำนวนเต็ม บูเลียน หรือสายอักขระเท่านั้น

1.3.6 งานวิจัยชิ้นนี้ไม่สนใจเรื่องของเส้นทางที่ไม่สามารถได้รับการประมวลผล (Infeasible Path)

1.3.7 หลักการสร้างกรณีทดสอบนั้นอาศัยหลักการของการทดสอบโดยอิงข้อบังคับ (Constraint-based testing)

1.3.8 โปรแกรมที่นำมาทดสอบต้องอยู่ในรูปของโปรแกรมภาษาปาสคาล ในมาตรฐาน ANSI

1.3.9 ขนาดของโปรแกรมที่นำมาทดสอบมีขนาดไม่เกิน 50 บรรทัด จำนวน 5 โปรแกรม โดยแต่ละโปรแกรมจะมีผลลัพธ์มากกว่า 1 ผลลัพธ์

1.3.10 ตัวดำเนินการมิวเทชันที่เลือกมาประยุกต์ใช้ในการสร้างโปรแกรมมิวแทนท์ เป็นตัวดำเนินการมิวเทชันที่เลือกมาด้วยหลักการของอีซีเล็คทีฟ ที่ประกอบไปด้วย UOI : Unary Operator Insertion, ROR : Relational Operator Replacement, AOR : Arithmetic Operator Replacement, LCR : Logical Connection Replacement, ABS : Absolute Value Insertion เท่านั้น

1.3.11 การตัดส่วนโปรแกรมที่นำมาใช้เป็นวิธีแบบอพลวัตในลักษณะย้อนกลับ (Static Backward Slicing) โดยอาศัยหลักการสร้างกราฟการขึ้นต่อกัน (Program

Dependence Graph) เพื่อวิเคราะห์ความขึ้นต่อกันทั้ง 2 ประเภท คือ การขึ้นต่อกันแบบควบคุม และการขึ้นต่อกันแบบตัวแปร

1.3.12 การแบ่งกลุ่มโปรแกรมมิวแทนท์อาศัยข้อบังคับการไปถึงเท่านั้น ในทางทฤษฎีสามารถนำเอาทั้งข้อบังคับการไปถึงและข้อบังคับที่จำเป็นมาเป็นเงื่อนไขในการแบ่งกลุ่ม แต่ด้วยวิธีการนี้จำเป็นต้องวิเคราะห์ข้อบังคับที่จำเป็นของทุกโปรแกรมมิวแทนท์ที่ถูกสร้างขึ้นมา ทั้งยังต้องหาอัลกอริธึมในการแบ่งกลุ่มที่เหมาะสมอีกด้วย ทำให้เวลาที่เสียไปจริงนั้นอาจไม่ลดลงไปจากเดิม

1.3.13 การสร้างโปรแกรมมิวแทนท์นั้น ข้อผิดพลาดที่ใส่ให้กับคำสั่งที่ไม่ใช่คำสั่งร่วมกันเท่านั้นที่สามารถนำมาสร้างเป็นโปรแกรมมิวแทนท์แบบหลายข้อผิดพลาด

1.4 ขั้นตอนในการวิจัย

1.4.1 ศึกษาและทำความเข้าใจการวิเคราะห์มิวเทชัน

1.4.2 ศึกษางานวิจัยที่เกี่ยวข้องกับการประยุกต์ใช้การวิเคราะห์มิวเทชันและการปรับปรุงประสิทธิภาพของการวิเคราะห์มิวเทชัน

1.4.3 ศึกษาการทดสอบโปรแกรมโดยอิงข้อบังคับ

1.4.4 ศึกษางานวิจัยที่เกี่ยวข้องในการตัดส่วนโปรแกรม

1.4.5 ศึกษางานวิจัยที่นำเอาการตัดส่วนโปรแกรมมาประยุกต์ใช้ในการวิเคราะห์มิวเทชัน

1.4.6 ทำการทดลองและพิสูจน์ทฤษฎีที่เกี่ยวข้องในการทำงานวิจัยชิ้นนี้

1.4.7 สรุปผลการวิจัย และจัดทำรายงานวิทยานิพนธ์

1.5 ประโยชน์ที่คาดว่าจะได้รับ

ประโยชน์ที่คาดว่าจะได้รับจากงานวิจัยชิ้นนี้คือ การลดจำนวนครั้งในการทำการทดสอบโปรแกรมด้วยวิธีการวิเคราะห์มิวเทชัน โดยการลดในเรื่องของจำนวนโปรแกรมมิวแทนท์ที่ถูกสร้างขึ้นและในเรื่องของจำนวนครั้งในการทดสอบโปรแกรมมิวแทนท์ที่ไม่จำเป็น

บทที่ 2
ทฤษฎีและงานวิจัยที่เกี่ยวข้อง

2.1 ทฤษฎีที่เกี่ยวข้อง

2.1.1 การวิเคราะห์มิวเทชัน (Mutation Analysis)

วิธีการทดสอบโปรแกรมนั้นมีอยู่มากมายหลายวิธี ซึ่งการทดสอบโปรแกรมด้วยหลักการวิเคราะห์มิวเทชันซึ่งได้รับการเสนอโดย Demillo [5] นั้นอาศัยการจำลองข้อผิดพลาดที่อาจเกิดขึ้นให้กับโปรแกรมที่นำมาทดสอบ ข้อผิดพลาดหนึ่งๆ ที่ใส่เข้าไปนั้นเปรียบเสมือนแต่ละโปรแกรมที่มีการทำงานที่ไม่ถูกต้อง โดยวัตถุประสงค์ที่แท้จริงของกระบวนการนี้คือ การปรับปรุงคุณภาพของชุดทดสอบให้มีประสิทธิผลที่ดียิ่งขึ้นเพื่อสามารถตรวจจับผลกระทบของข้อผิดพลาดที่ถูกใส่เข้าไป ชนิดของข้อผิดพลาดนั้นถูกกำหนดไว้ในตัวดำเนินการมิวเทชัน ซึ่งข้อผิดพลาดเหล่านี้จะต้องเป็นข้อผิดพลาดที่ไม่ผิดวากยสัมพันธ์ (Syntax) ของภาษาที่ใช้เขียนโปรแกรม นิยามพื้นฐานที่เกี่ยวข้องกับการทำการวิเคราะห์โปรแกรมมิวแทนท์นั้นมีดังต่อไปนี้

นิยามที่ 1
ข้อผิดพลาด (Fault) คือ สิ่งที่แสดงถึงการมีอยู่ของความผิดพลาด (Error)

นิยามที่ 2
ผลกระทบของข้อผิดพลาด (Incident) เป็นผลลัพธ์ที่ได้รับจากการมีอยู่ของข้อผิดพลาด ซึ่งผลกระทบนี้สามารถแสดงออกมาให้ผู้ใช้หรือผู้ทดสอบเห็นได้

นิยามที่ 3
โปรแกรม M ถูกเรียกว่า โปรแกรมมิวแทนท์ *ที่เกิดจากโปรแกรม P หาก M เกิดจากการเปลี่ยนแปลงโครงสร้างของโปรแกรม P โดยรูปแบบของการเปลี่ยนแปลงนั้นได้รับการนิยามไว้ในตัวดำเนินการมิวเทชัน*

ตัวดำเนินการมิวเทชันใช้สำหรับนิยามข้อผิดพลาดที่จะใส่ให้กับโปรแกรมที่นำมาทดสอบเพื่อให้โปรแกรมมิวแทนท์นั้นมีพฤติกรรมที่แตกต่างไปจากโปรแกรมทดสอบ ชนิดของตัวดำเนินการมิวเทชันที่ได้แสดงไว้ในตารางที่ 2.1 นั้นอยู่ในงานวิจัย [15] ในตัวอย่างที่ 2.1 แสดงถึงโปรแกรมมิวแทนท์ที่ได้รับจากการใส่ข้อผิดพลาดให้กับโปรแกรมที่นำมาทดสอบ

ตัวอย่างที่ 2.1 *โปรแกรมมิวแทนท์ที่ได้รับจากการใส่ข้อผิดพลาดให้กับโปรแกรมที่นำมาทดสอบ*

โปรแกรมที่นำมาทดสอบ

```
001  Program FindMax;
002  var n, m, max : integer;
003  begin
004          readln(n);
005          readln(m);
006          max := m;
007          if (n>m) then max := n;
008          writeln(max);
009  end.
```

ตัวอย่างของโปรแกรมมิวแทนท์ที่ได้รับการใส่ข้อผิดพลาดให้กับโปรแกรมที่นำมาทดสอบ

โปรแกรมมิวแทนท์ที่ 1

```
001  Program FindMax;
002  var n, m, max : integer;
003  begin
004          readln(n);
005          readln(m);
006          max := m;
007          if (n<m) then max := n;
008          writeln(max);
009  end.
```

ในโปรแกรมมิวแทนท์ที่ 1 นั้น ได้มีการใส่ข้อผิดพลาดโดยการเปลี่ยนจากคำสั่ง if (n>m) then max :=n; ซึ่งอยู่ในโปรแกรมที่นำมาทดสอบ เป็นคำสั่ง if (n<m) then max=n;

โปรแกรมมิวแทนท์ที่ 2

```
001  Program FindMax;
002  var n, m, max : integer;
003  begin
004          readln(n);
005          readln(m);
006          max := m;
007          if (n>=m) then max := n;
008          writeln(max);
009  end.
```

ในโปรแกรมมิวแทนท์ที่ 2 นั้น ได้มีการใส่ข้อผิดพลาดโดยการเปลี่ยนจากคำสั่ง if (n>m) then max :=n; ซึ่งอยู่ในโปรแกรมที่นำมาทดสอบ เป็นคำสั่ง if (n>=m) then max:=n;

หมายเหตุ ทุกโปรแกรมในตัวอย่างของงานวิจัยชิ้นนี้เป็นไปตามมาตรฐาน ANSI/ISO 7185

ตารางที่ 2.1 ตารางแสดงตัวดำเนินการมิวเทชัน [15]

Mutation Operator	Description
AAR	array reference for array reference replacement
ABS	absolute value insertion
ACR	array reference for constant replacement
AOR	arithmetic operator replacement
ASR	array reference for scalar variable replacement
CAR	constant for array reference replacement
CNR	comparable array name replacement
CRP	constant replacement
CSR	constant for scalar variable replacement
DER	DO statement and replacement
DSA	DATA statement alterations
GLR	GOTO label replacement
LCR	logical connector replacement
ROR	relational operator replacement
RSR	RETURN statement replacement
SAN	statement analysis
SAR	scalar variable for array reference replacement
SCR	scalar for constant replacement
SDL	statement deletion
SRC	source constant replacement
SVR	scalar variable replacement
UOI	unary operator insertion

การทดสอบโปรแกรมด้วยวิธีการวิเคราะห์มิวเทชันจำเป็นต้องมีการสร้างชุดทดสอบเพื่อแยกความแตกต่างของผลลัพธ์ที่เกิดจากโปรแกรมมิวแทนท์และโปรแกรมทดสอบ

นิยามที่ 4

โปรแกรมมิวแทนท์ M จะถูก กำจัด(Killed) *ด้วยกรณีทดสอบ t ถ้า M(t) ซึ่งเป็นผลลัพธ์จากการประมวลผลโปรแกรมมิวแทนท์ M ด้วยกรณีทดสอบ t ให้ผลลัพธ์ที่ต่างจาก P(t) ซึ่งเป็นผลลัพธ์จากการประมวลผลโปรแกรมที่นำมาทดสอบด้วยกรณีทดสอบ t*

นิยามที่ 5

โปรแกรมมิวแทนท์ที่ให้ผลลัพธ์จากการประมวลผลเหมือนกับโปรแกรมทดสอบ ไม่ว่าจะใช้ชุดทดสอบใดๆ มาทดสอบจึงทำให้ไม่สามารถแยกความแตกต่างของผลลัพธ์ได้ โปรแกรมมิวแทนท์เหล่านี้ถูกเรียกว่า โปรแกรมมิวแทนท์สมมูล (Equivalence Mutant)

ในตัวอย่างที่ 2.2 แสดงถึงโปรแกรมมิวแทนท์สมมูลและการกำจัดโปรแกรมมิวแทนท์ของตัวอย่างที่ 2.1

ตัวอย่างที่ 2.2 *การกำจัดโปรแกรมมิวแทนท์ในตัวอย่างที่ 2.1 โดยอาศัยกรณีทดสอบที่มีค่า val1=3 และ val2=2 มาทดสอบกับโปรแกรมมิวแทนท์ โดยผลลัพธ์ที่ได้จากการทดสอบถูกแสดงไว้ในตารางที่ 2.2*

ตารางที่ 2.2 ตารางแสดงผลลัพธ์ที่ได้รับจากการทดสอบโปรแกรมมิวแทนท์

โปรแกรม	**ผลลัพธ์ที่ได้รับ**	**สถานะ**
FindMax	max = 3	-
โปรแกรมมิวแทนท์1	max = 2	ถูกกำจัด
โปรแกรมมิวแทนท์2	max = 3	ไม่ถูกกำจัด

ผลลัพธ์ที่ได้รับจากโปรแกรม FindMax ซึ่งเป็นโปรแกรมที่นำมาทดสอบจะถูกเก็บไว้เป็น Expected Output ในโปรแกรมมิวแทนท์1 นั้นถูกกำจัดเนื่องจากค่าของผลลัพธ์ต่างไปจากค่าของผลลัพธ์ของโปรแกรมที่นำมาทดสอบ ส่วนในโปรแกรมมิวแทนท์2 นั้นให้ผลลัพธ์ที่ไม่ต่างไปจากเดิม ดังนั้นโปรแกรมมิวแทนท์2 จะไม่ถูกกำจัด ทั้งนี้เนื่องจากโปรแกรมมิวแทนท์2 นั้นเป็นโปรแกรมมิวแทนท์สมมูล ซึ่งไม่ว่าจะใช้กรณีทดสอบใดๆ มาทดสอบจะไม่สามารถกำจัดโปรแกรมมิวแทนท์2 ได้

นิยามที่ 6

ชุดทดสอบ T ซึ่งประกอบด้วยกรณีทดสอบต่างๆ จะถูกเรียกว่า การมิวเทชันที่เพียงพอ (Mutation-adequate) สำหรับโปรแกรมทดสอบ P ถ้าทุกโปรแกรมมิวแทนท์ถูกกำจัดด้วยกรณีทดสอบที่มีอยู่ในชุดทดสอบ T

ในการวัดประสิทธิผลของชุดทดสอบใดๆนั้น จะถูกแสดงออกมาอยู่ในรูปของคะแนนมิวเทชัน *(Mutation Score)* ซึ่งมีการนิยามไว้ดังนี้

$$MS\ (P,T)\ =\ \frac{K \times 100}{M\ -E}$$

โดยที่ MS คือ คะแนนมิวเทชัน

P คือ โปรแกรมทดสอบ

T คือ ชุดทดสอบ

K คือ จำนวนโปรแกรมมิวแทนท์ที่ถูกกำจัด

M คือ จำนวนโปรแกรมมิวแทนท์ทั้งหมด

E คือ จำนวนโปรแกรมมิวแทนท์สมมูล

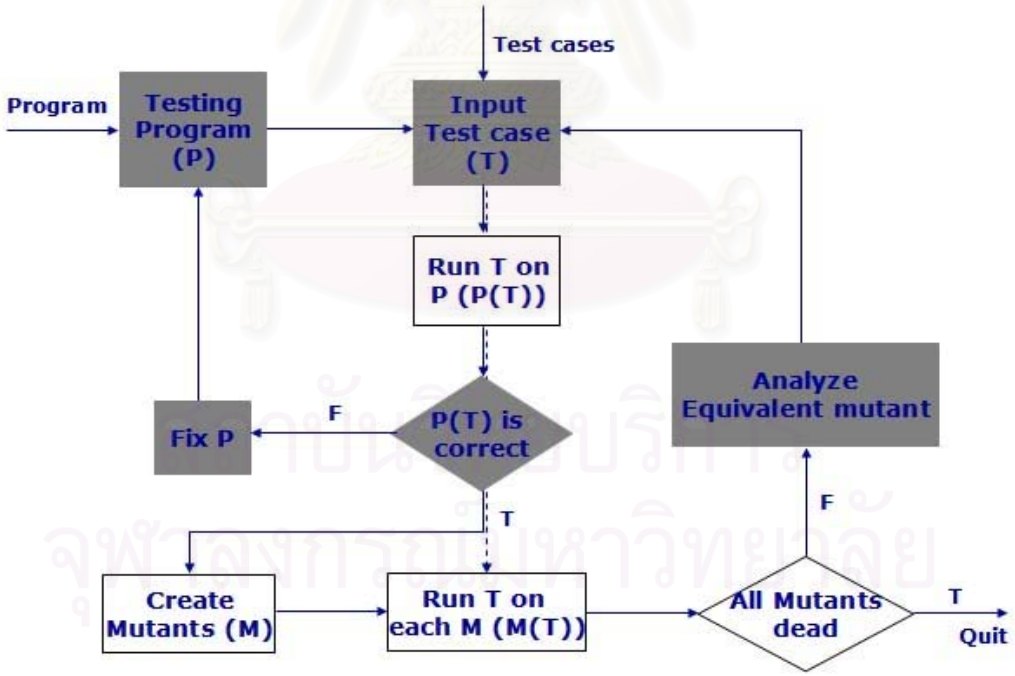
ประสิทธิภาพ (Efficiency) ของการทดสอบ สามารถวัดได้จากทรัพยากรที่สูญเสียไปในการทดสอบซอฟต์แวร์แบบต่างๆ เช่น เวลาที่เสียไป เป็นต้น

ในงานวิจัยชิ้นนี้จะพิจารณาจำนวนครั้งในการทดสอบเป็นสำคัญ โดยประสิทธิภาพของการวิเคราะห์มิวเทชันสามารถวัดได้จากจำนวนครั้งในการทดสอบแต่ละโปรแกรมมิวแทนท์ ดังนั้นประสิทธิภาพของการวิเคราะห์มิวเทชันจึงเป็นสัดส่วนผกผันกับจำนวนโปรแกรมมิวแทนท์และจำนวนครั้งของการทดสอบกับโปรแกรมมิวแทนท์

ขั้นตอนในการทำการวิเคราะห์มิวเทชัน [14] เริ่มต้นจากการนำเอาโปรแกรมที่ต้องการทดสอบ P มาทดสอบกับกรณีทดสอบ t ที่ได้รับการสร้างมาจากกรรมวิธีต่างๆ เพื่อตรวจสอบว่าผลลัพธ์จากการทำงานนั้นถูกต้องหรือไม่ หากโปรแกรมที่นำมาทดสอบมีการทำงานที่ไม่ถูกต้องให้แก้ไขโปรแกรมที่นำมาทดสอบนี้ให้มีการทำงานที่ถูกต้อง ในทางตรงกันข้ามหากโปรแกรมที่นำมาทดสอบมีผลลัพธ์ในการทำงานถูกต้องจะรับประกันได้อย่างไรว่าโปรแกรมที่นำมาทดสอบนั้นปราศจากข้อผิดพลาดที่แฝงเร้นอยู่ ดังนั้นการวิเคราะห์มิวเทชันจึงถูกเลือกขึ้นมาเพื่อปรับปรุงประสิทธิผลของกรณีทดสอบที่มีอยู่เดิมนี้ โดยอาศัยการใส่ข้อผิดพลาดที่ได้รับการนิยามไว้ในตัวดำเนินการมิวเทชันเพื่อสร้างเป็นโปรแกรมมิวแทนท์ M ต่างๆ ซึ่งผลลัพธ์ของโปรแกรมที่นำมาทดสอบจะถูกเก็บไว้เป็นผลลัพธ์ที่คาดหวัง (Expected output) จากนั้นจะนำกรณีทดสอบไปทดสอบกับโปรแกรมมิวแทนท์ต่างๆ ว่าให้ผลลัพธ์ที่ตรงกับผลลัพธ์ที่คาดหวังหรือไม่ หากไม่ตรงกันแสดงว่ากรณีทดสอบนี้มีความสามารถเพียงพอในการตรวจจับผลกระทบของข้อผิดพลาดที่เกิดจากโปรแกรมมิวแทนท์ตัวนั้นๆ โดยจะเรียกว่าโปรแกรมมิวแทนท์นี้ว่าถูกกำจัดและจะไม่นำมาพิจารณาอีกต่อไป จากนั้นตรวจสอบว่าทุกๆ โปรแกรมมิวแทนท์ถูกกำจัดได้ด้วยกรณีทดสอบนี้

หรือไม่ หากถูกกำจัดทั้งหมดแล้วเป็นอันสิ้นสุดกระบวนการ แต่ถ้ามีบางโปรแกรมมิวแทนท์ยังไม่ถูกกำจัดจะนำโปรแกรมมิวแทนท์เหล่านี้มาวิเคราะห์ต่อไปอีกว่าเป็นโปรแกรมมิวแทนท์สมมูลหรือไม่ ถ้าเป็นโปรแกรมมิวแทนท์สมมูลจะไม่นำมาพิจารณาในการทำการวิเคราะห์มิวเทชันเนื่องมาจากไม่ว่าใช้กรณีทดสอบใดๆ มาทดสอบจะไม่สามารถกำจัดโปรแกรมมิวแทนท์ชนิดนี้ได้ ต่อจากนั้นจะเข้าสู่ขั้นตอนการสร้างกรณีทดสอบตัวอื่นๆ เพื่อมาทดสอบและพยายามกำจัดโปรแกรมมิวแทนท์ที่เหลือนี้ให้ได้หมด แผนภาพแสดงการทำงานของการวิเคราะห์มิวเทชันเป็นดังรูปที่ 2.1

เนื่องจากการทดสอบเริ่มต้นโดยนำกรณีทดสอบมาทีละกรณีทดสอบ นำไปทดสอบกับโปรแกรมมิวแทนท์ทั้งหมดเพื่อดูว่า โปรแกรมมิวแทนท์ใดสามารถถูกกำจัดได้ด้วยกรณีทดสอบนี้ โปรแกรมมิวแทนท์ที่ไม่สามารถถูกกำจัดได้จะถูกนำไปทดสอบในรอบถัดไปกับกรณีทดสอบกรณีต่อไปจนกระทั่ง โปรแกรมมิวแทนท์ทั้งหมดถูกกำจัด ดังนั้นจำนวนครั้งของการทดสอบที่สนใจ คือ ผลรวมของจำนวนโปรแกรมมิวแทนท์ที่ถูกทดสอบด้วยกรณีทดสอบแต่ละกรณี



รูปที่ 2.1 แผนภาพแสดงขั้นตอนการทำงานของการวิเคราะห์มิวเทชัน [14]

2.1.2 การทดสอบโดยอิงข้อบังคับ (Constraint-Based Testing)

ในการทำการทดสอบซอฟต์แวร์นั้น สิ่งที่ขาดไม่ได้คือกรณีทดสอบ ซึ่งกรรมวิธีในการสร้างกรณีทดสอบนั้นมีให้เลือกใช้มากมายตามความเหมาะสมของสถานการณ์ การทดสอบโดยอิงข้อบังคับ [7, 8] นั้นเป็นการสร้างกรณีทดสอบมาเพื่อกำจัดโปรแกรมมิวแทนท์ โดยอาศัยกฎข้อบังคับในการกำจัดโปรแกรมมิวแทนท์ซึ่งอยู่ในรูปของนิพจน์ทางคณิตศาสตร์มาวิเคราะห์ว่ากรณีทดสอบควรมีคุณสมบัติเป็นเช่นไรจึงจะสามารถกำจัดโปรแกรมมิวแทนท์นั้นๆ ได้ กฎข้อบังคับในการกำจัดโปรแกรมมิวแทนท์แบ่งออกเป็น 3 ข้อ ดังต่อไปนี้

2.1.2.1 ข้อบังคับการไปถึง (Reachability Constraint) เป็นข้อบังคับที่กำหนดให้คำสั่งที่ได้รับการใส่ข้อผิดพลาดจำเป็นต้องได้รับการประมวลผล

2.1.2.2 ข้อบังคับที่จำเป็น (Necessity Constraint) เป็นข้อบังคับที่กล่าวว่า คำสั่งที่ได้รับการใส่ข้อผิดพลาดเมื่อได้รับการประมวลผลแล้วผลลัพธ์ที่ได้จากคำสั่งนั้นต้องต่างไปจากค่าเดิมในโปรแกรมที่นำมาทดสอบ

2.1.2.3 ข้อบังคับที่เพียงพอ (Sufficient Constraint) ถึงแม้ว่าชุดทดสอบจะรองรับข้อบังคับที่จำเป็นแล้วก็ตาม ผลลัพธ์สุดท้ายของการประมวลผลโปรแกรมอาจไม่แตกต่างไปจากเดิม ดังนั้นในข้อบังคับข้อนี้จึงบังคับให้ผลลัพธ์สุดท้ายของโปรแกรมมิวแทนท์ต่างไปจากของโปรแกรมที่นำมาทดสอบ

ตัวอย่างที่ 2.3 แสดงให้เห็นถึงข้อบังคับทั้ง 3 ข้อบังคับ ของโปรแกรมมิวแทนท์ที่เกิดจากการใส่ข้อผิดพลาดให้กับโปรแกรมที่นำมาทดสอบ

ตัวอย่างที่ 2.3 *จากตัวอย่างของโปรแกรมที่นำมาทดสอบในตัวอย่างที่ 2.1 มีคำสั่งดังต่อไปนี้*

```
001  Program FindMax;
002  var n, m, max : integer;
003  begin
004          readln(n);
005          readln(m);
006          max := m;
007          if (n>=m) then max := n;
008          writeln(max);
009  end.
```

โดยสมมุติให้คำสั่งในแถวที่ 007 ได้รับการเปลี่ยนแปลงเป็น if (n>m) then max := -n; ข้อบังคับต่างๆ สำหรับคำสั่งที่ได้รับการใส่ข้อผิดพลาดเป็นดังต่อไปนี้

1. ข้อบังคับการไปถึง มีค่าเป็น n > m
2. ข้อบังคับที่จำเป็น มีค่าเป็น n <> 0 เนื่องจากค่า -n และค่า n ให้ค่าเดียวกันที่ n=0
3. ข้อบังคับที่เพียงพอ มีค่าเป็น n > m และ n <> 0 เพื่อให้ผลกระทบของข้อผิดพลาดสามารถถ่ายทอดไปสู่ผลลัพธ์สุดท้ายได้

การทดสอบโดยอิงข้อบังคับจะอาศัยเพียงข้อบังคับการไปถึงและข้อบังคับที่จำเป็น มาสร้างเป็นข้อบังคับในการสร้างกรณีทดสอบ เนื่องจากการนำเอาข้อบังคับที่เพียงพอมาวิเคราะห์ด้วยนั้นจะทำให้เกิดความซับซ้อนในการวิเคราะห์นิพจน์ทางคณิตศาสตร์มาก

2.1.3 กรรมวิธีการตัดส่วนโปรแกรม (Program Slicing)

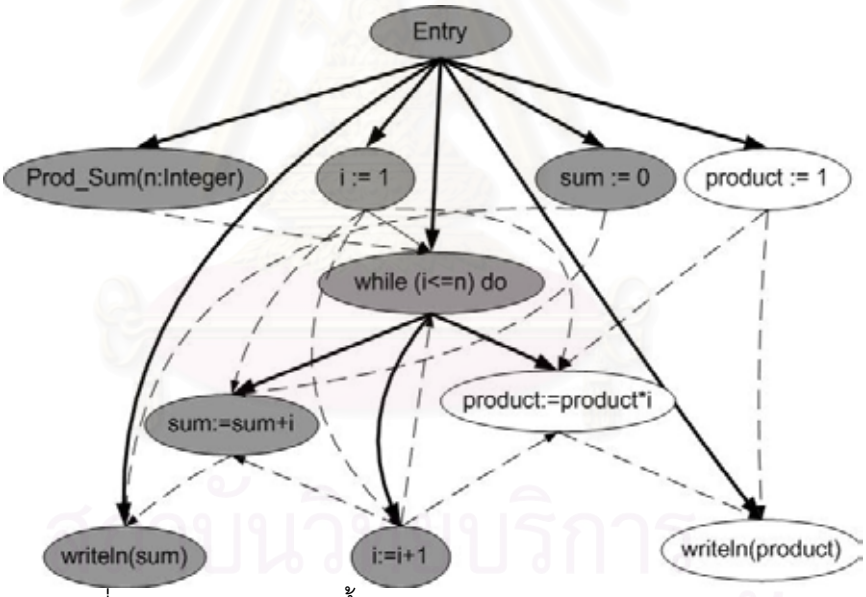
กรรมวิธีการตัดส่วนโปรแกรมได้ถูกเสนอโดย Weiser [20] ซึ่งได้ประยุกต์ใช้ในการวิเคราะห์เพื่อทำความเข้าใจโปรแกรมให้ง่ายขึ้น (Program Comprehension) ทั้งยังในเรื่องของการดีบัก (Debugging) วัตถุประสงค์หลักคือการตัดโปรแกรมเอาเฉพาะส่วนโปรแกรมที่เกี่ยวข้องกับจุดที่สนใจ โดยจุดที่สนใจนั้นได้นิยามไว้ใน บรรทัดฐานการตัด (Slicing Criterion) ที่อยู่ในรูปของคู่ลำดับ (V, n) โดย V นั้นเป็นชุดของตัวแปรที่สนใจ ส่วน n นั้นเป็นตำแหน่งของคำสั่งในโปรแกรม Ottenstein [18] ได้เสนออัลกอริธึมในการตัดส่วนโปรแกรมโดยอาศัยการท่องไปตามบัพ (node) ต่างๆ ในกราฟการขึ้นต่อกัน (Program Dependence Graph) โดยบัพของกราฟแสดงคำสั่งในโปรแกรม ส่วนเส้นที่เชื่อมแต่ละบัพแสดงถึง การขึ้นต่อกันแบบตัวแปร (Data Dependence) ที่แสดงถึงความสัมพันธ์ของการกำหนดค่าให้แก่ตัวแปรและอ้างถึงตัวแปรนั้น และการขึ้นต่อกันแบบควบคุม (Control Dependence) ที่แสดงถึงความสัมพันธ์ของการควบคุมของแต่ละคำสั่ง การตัดส่วนโปรแกรมด้วยวิธีการนี้ถูกเรียกว่า การตัดส่วนแบบอพลวัตในลักษณะย้อนกลับ (Static Backward Slicing) ตัวอย่างของการตัดส่วนโปรแกรมโดยอาศัยการท่องไปตามบัพต่างๆ ของกราฟการขึ้นต่อกันถูกแสดงไว้ในตัวอย่างที่ 2.4

ตัวอย่างที่ 2.4 *กำหนดให้ส่วนของโปรแกรม Prod_Sum.มีคำสั่งดังต่อไปนี้*

```
001  Procedure Prod_Sum(n: Integer);
002  var i,n, sum, product : Integer;
003  begin
004            i := 1;
```

```
005         sum := 0;
006         product := 1;
007         while (i<=n) do
008         begin
009                 sum := sum + i;
010                 product := product * i;
011                 i := i + 1;
012         end;
013         writeln(sum);
014         writeln(product);
015 end;
```

กราฟการขึ้นต่อกันของโปรแกรมข้างต้นนี้ถูกแสดงไว้ในรูปที่ 2.2 โดยเส้นสีเข้มนั้นแสดงถึงการขึ้นต่อกันแบบควบคุม ส่วนเส้นประนั้นแสดงถึงการขึ้นต่อกันแบบตัวแปร ส่วนจุดยอดที่มีสีเข้มนั้นแสดงถึงคำสั่งที่ส่งผลกระทบต่อตัวแปรในบรรทัดฐานการตัดซึ่งในที่นี้คือ ({sum},013)



รูปที่ 2.2 แสดงกราฟการขึ้นต่อกันของโปรแกรม Prod_Sum(int n)

ส่วนของโปรแกรมที่ได้รับการตัดส่วนที่ไม่เกี่ยวข้องออกไปเป็นดังนี้

```
001 Procedure Prod_Sum(n: Integer);
002 var i,n, sum, product : Integer;
003 begin
004         i := 1;
005         sum := 0;
007         while (i<=n) do
```

```
008         begin
009                 sum := sum + i;
011                 i := i + 1;
012         end;
013         writeln(sum);
014 end;
```

โดยทุกๆ คำสั่งที่เหลืออยู่ในส่วนของโปรแกรมภายหลังจากการตัดส่วนจะส่งผลกระทบต่อบรรทัดฐานในการตัดที่ตัวแปร sum ในคำสั่งที่ 13

2.2 งานวิจัยที่เกี่ยวข้อง

2.2.1 งานวิจัย An Experimental Determination of Sufficient Mutant Operators [15]

ในการทดสอบโปรแกรมด้วยวิธีการวิเคราะห์มิวเทชันนั้น จำนวนของโปรแกรมมิวแทนท์เป็นปัญหาสำคัญในการทดสอบ ดังนั้นในงานวิจัยชิ้นนี้ได้วิเคราะห์ประสิทธิภาพของการทำการวิเคราะห์มิวเทชันให้อยู่ในรูปของจำนวนโปรแกรมมิวแทนท์ที่ถูกสร้างขึ้นจากการใส่ข้อผิดพลาด พร้อมทั้งได้นำเสนอแนวทางในการลดจำนวนของโปรแกรมมิวแทนท์ลงโดยการตัดตัวดำเนินการมิวเทชันบางชนิดออกไปจากตารางที่ 2.1 งานวิจัยชิ้นนี้ได้ทำการทดลองและพบว่าตัวดำเนินการมิวเทชันที่สามารถสร้างข้อผิดพลาดที่เกิดกับเอ็กซ์เพรสชัน (Expression) ของแต่ละคำสั่งในโปรแกรม ซึ่งเรียกว่าตัวดำเนินการแบบอีซีเล็คทีฟ (E-Selective Operator) นั้น เป็นตัวดำเนินการที่เพียงพอสำหรับการสร้างโปรแกรมมิวแทนท์ที่จะนำมาใช้ในการวิเคราะห์มิวเทชัน โดยตัวดำเนินการที่สร้างข้อผิดพลาดให้กับเอ็กซ์เพรสชันมีดังต่อไปนี้

1. ABS (Absolute Value Insertion) เป็นตัวดำเนินการที่บังคับให้แต่ละนิพจน์ทางคณิตศาสตร์มีค่าเป็นบวกเสมอ ลบเสมอ หรือมีค่าเป็นศูนย์
2. AOR (Arithmetic Operator Replacement) เป็นตัวดำเนินการที่เปลี่ยนตัวดำเนินการทางคณิตศาสตร์ ตัวเดิมไปเป็นตัวดำเนินการทางคณิตศาสตร์ตัวใหม่
3. LCR (Logical Connector Replacement) เป็นตัวดำเนินการที่เปลี่ยนตัวเชื่อมทางตรรกศาสตร์ ตัวเดิมไปเป็นตัวเชื่อมตัวใหม่
4. ROR (Relational Operator Replacement) เป็นตัวดำเนินการที่เปลี่ยนตัวดำเนินการทางความสัมพันธ์ ตัวเดิมไปเป็นตัวดำเนินการทางความสัมพันธ์ตัวใหม่
5. UOI (Unary Operator Insertion) เป็นตัวดำเนินการที่ทำการใส่ตัวดำเนินการแบบเดี่ยว เช่น -, +, ++, -- ที่ด้านหน้าเอ็กซ์เพรสชัน

ในการทดลองของงานวิจัยชิ้นนี้ยังแสดงให้เห็นต่อไปอีกว่า ชุดของกรณีทดสอบที่ถูกสร้างขึ้นมาจากการทำการวิเคราะห์มิวเทชันโดยอาศัยตัวดำเนินการแบบอีซีเล็คทีฟ นั้นมีความสามารถเพียงพอที่จะกำจัดโปรแกรมมิวแทนท์โดยเฉลี่ย 99.5% ของโปรแกรมมิวแทนท์ที่ได้รับมาจากวิธีการเดิม

2.2.2 งานวิจัย Automatically Detecting Equivalent Mutants and Infeasible Paths [16]

ปัญหาที่สำคัญรองลงมาจากเรื่องของจำนวนโปรแกรมมิวแทนท์ในการทำการวิเคราะห์มิวเทชันคือ การที่ผู้ทดสอบจำเป็นต้องตรวจสอบว่าโปรแกรมมิวแทนท์ที่สร้างขึ้นมานั้นเป็นโปรแกรมมิวแทนท์สมมูลหรือไม่ ซึ่งในการตรวจสอบนั้นจำเป็นต้องอาศัยผู้ทดสอบเป็นคนดำเนินการ ดังนั้นงานวิจัยชิ้นนี้จึงเสนอแนวทางในการตรวจสอบโปรแกรมมิวแทนท์สมมูลแบบอัตโนมัติ ซึ่งอาศัยหลักการของการทดสอบโดยอิงข้อบังคับที่กล่าวว่า ในการกำจัดโปรแกรมมิวแทนท์ใดๆ นั้นกรณีทดสอบควรรองรับข้อบังคับ 2 ประการคือ ข้อบังคับการไปถึง และ ข้อบังคับที่จำเป็น ในทางตรงกันข้ามหากโปรแกรมมิวแทนท์จะเป็นโปรแกรมมิวแทนท์แบบสมมูลได้นั้นจะต้องไม่สามารถหากรณีทดสอบที่รองรับกฎข้อบังคับในการกำจัดโปรแกรมมิวแทนท์ได้ ซึ่งอาจเกิดมาได้จากกรณีดังต่อไปนี้คือ

1. การที่ข้อบังคับการไปถึงไม่มีทางเป็นจริงได้
2. การที่ข้อบังคับที่จำเป็นไม่มีทางเป็นจริงได้
3. การที่ข้อบังคับการไปถึงและข้อบังคับที่จำเป็นเกิดความขัดแย้งระหว่างกัน

จากการทดลองกับโปรแกรมทั้งหมด 11 โปรแกรมพบว่า โปรแกรมมิวแทนท์สมมูลของ 7 โปรแกรมใน 11 โปรแกรมที่นำมาทดสอบนั้นถูกตรวจพบถึง 65 เปอร์เซ็นต์ โดยที่ค่าเฉลี่ยของทั้ง11 โปรแกรมอยู่ในระดับ 45 เปอร์เซ็นต์

2.2.3 งานวิจัย Using Program Slicing to Assist in the Detection of Equivalent Mutants [9]

ในงานวิจัยชิ้นนี้เป็นการนำเอากรรมวิธีการตัดส่วนโปรแกรมมาประยุกต์ใช้ในการตรวจสอบโปรแกรมมิวแทนท์สมมูล โดยชนิดของกรรมวิธีการตัดส่วนโปรแกรมที่เลือกใช้นั้นคือการตัดส่วนโปรแกรมแบบ อสัณฐาน (Amorphous Slicing) ซึ่งจะทำให้โปรแกรมที่ได้รับการตัดส่วนนั้นอยู่ในรูปแบบที่ง่ายที่สุด ขั้นตอนของการตรวจสอบโปรแกรมมิวแทนท์สมมูลเริ่มต้นโดยการ

ประกาศตัวแปรหนึ่งๆ (z) ที่มีชนิดเป็นตรรกะ (Boolean) ไว้ที่ตำแหน่งเริ่มต้นของโปรแกรม โดยกำหนดค่าเริ่มต้นเป็นจริง พร้อมทั้งใส่คำสั่งที่มีการกำหนดค่า z เป็น (z and (Necessity Condition)) ไว้ก่อนหน้าคำสั่งที่ได้รับการใส่ข้อผิดพลาด จากนั้นทำการตัวส่วนของโปรแกรมโดยกำหนดบรรทัดฐานของการตัดไว้ที่ตัวแปร z ในคำสั่งที่ใส่เข้าไปก่อนหน้าคำสั่งที่ใส่ข้อผิดพลาด ในการทำการตัดส่วนของโปรแกรมด้วยวิธีอสัณฐานนั้นจะประกอบด้วยขั้นตอนย่อยเพิ่มจากการตัดส่วนโปรแกรมธรรมดา 3 ขั้นตอนคือ ขั้นตอนที่1 คือ การตีความเงื่อนไข (Condition Consideration) ขั้นตอนที่ 2 คือ การส่งผ่านค่าคงที่ (Constant Propagation) ส่วนขั้นตอนสุดท้ายคือ การคำนวณค่าคงที่ (Constant Evaluation) หลังจากทำการตัดส่วนโปรแกรมเรียบร้อยแล้วหากค่าของตัวแปร z ไม่ต่างไปจากเดิมนั่นคือเป็นจริงอยู่ โปรแกรมมิวแทนท์ตัวนั้นจะเป็นโปรแกรมมิวแทนท์สมมูล งานวิจัยชิ้นนี้แสดงให้เห็นว่าการตรวจสอบโปรแกรมมิวแทนท์สมมูลในงานวิจัยที่ 3.2 นั้น สามารถกระทำได้ด้วยวิธีการเช่นเดียวกันนี้ พร้อมทั้งแสดงให้เห็นถึงกรณีที่การวิเคราะห์กฎข้อบังคับตามงานวิจัยที่ 3.2 ไม่สามารถทำได้แต่สามารถทำได้ด้วยวิธีนี้ ในการทำการตรวจสอบโปรแกรมมิวแทนท์สมมูลโดยอาศัยหลักการตัดส่วนโปรแกรมมาช่วยนั้นมีข้อดีอีก 2 ประการคือ ข้อแรกขนาดของโปรแกรมที่พิจารณาลดลงเนื่องมาจากใช้การตัดส่วนโปรแกรม ข้อถัดมาคือ การเปรียบเทียบผลลัพธ์จะอยู่ในรูปของตัวแปรที่เป็นตัวแปรทางตรรกะเพียงตัวเดียวเท่านั้น

2.2.4 งานวิจัย The Relationship between Program Dependence and Mutation Analysis [10]

งานวิจัยชิ้นนี้ได้นำเสนอความเกี่ยวเนื่องกันระหว่างการวิเคราะห์การขึ้นต่อกัน (Dependence Analysis) ด้วยหลักการของกราฟการขึ้นต่อกัน และการทำการวิเคราะห์มิวเทชันโดยเฉพาะอย่างยิ่งการนำเอาการวิเคราะห์การขึ้นต่อกันมาช่วยในการแก้ปัญหา 2 ข้อ ดังต่อไปนี้

1. ทำอย่างไรให้หลีกเลี่ยงการสร้างโปรแกรมมิวแทนท์สมมูล
2. ทำอย่างไรให้สามารถสร้างกรณีทดสอบที่เหมาะสมในการกำจัดโปรแกรมมิวแทนท์ที่ไม่ใช่โปรแกรมมิวแทนท์สมมูล

ในการนำเอากราฟการขึ้นต่อกันมาวิเคราะห์ความเป็นโปรแกรมมิวแทนท์สมมูลนั้นในงานวิจัยชิ้นนี้ได้เสนอว่า หากมีการกำหนดค่าตัวแปรใหม่ (Re-Initialize) หลังจากที่มีการใส่ข้อผิดพลาดให้กับตัวแปรตัวนั้น ทำให้โปรแกรมมิวแทนท์ที่ได้เป็นโปรแกรมมิวแทนท์สมมูล ซึ่งกราฟการขึ้นต่อกันนั้นสามารถประยุกต์ใช้เพื่อบอกถึงคำสั่งที่ไม่มีทางส่งผลต่อผลลัพธ์สุดท้ายของโปรแกรมนั่นคือ คำสั่งที่มีการกำหนดค่าตัวแปรก่อนหน้าการกำหนดค่าตัวแปรตัวนั้นใหม่

กราฟการขึ้นต่อกันสามารถนำมาพิจารณาควบคู่ไปกับการสร้างกรณีทดสอบได้ โดยหากนำกราฟการขึ้นต่อกันมาวิเคราะห์ว่าตัวแปรใดในโปรแกรมส่งผลต่อผลลัพธ์ของโปรแกรมที่สนใจ ความเป็นไปได้ในการสร้างกรณีทดสอบทั้งหมดจะลดลงเนื่องมาจากสามารถตัดตัวแปรบางตัวออกไปจากการพิจารณาการสร้างกรณีทดสอบได้

2.2.5 งานวิจัย A Technique for Mutation of Java Objects [1]

ในปัจจุบันแนวความคิดทางด้านการเขียนโปรแกรมเชิงวัตถุเป็นที่นิยมเป็นอย่างมาก จาวาเป็นภาษาหนึ่งที่อาศัยหลักการนี้ งานวิจัยชิ้นนี้ได้เสนอการนำเอาการวิเคราะห์มิวเทชันมาประยุกต์ใช้กับภาษาจาวา โดยนิยามตัวดำเนินการมิวเทชันที่จำเป็นสำหรับการสร้างข้อผิดพลาดที่อาจเกิดขึ้นในการเขียนโปรแกรมเชิงวัตถุพร้อมทั้งพัฒนาระบบในการใส่ข้อผิดพลาดที่ได้รับการนิยามไว้ในตัวดำเนินการมิวเทชัน ในงานวิจัยนี้ได้ศึกษาตัวดำเนินการมิวเทชันที่มีการนำไปประยุกต์ใช้ในอินเตอร์เฟสดังต่อไปนี้ คือ

1. ชนิดคอนเทนเนอร์ (Container Types) ที่ถูกนิยามไว้ในอินเตอร์เฟสคอลเล็คชัน (Collection) และรายการ (List) ตัวอย่างของตัวดำเนินการประเภทนี้ได้แก่ การล้างค่าที่บรรจุในคอลเล็คชัน (Collection) การลบสมาชิกบางตัวในคอลเล็คชัน การเพิ่มสมาชิกบางตัวให้กับคอลเล็คชัน
2. การวนรอบ (Iterators) ที่ถูกนิยามไว้ในอินเตอร์เฟสของการวนรอบ ตัวอย่างของตัวดำเนินการประเภทนี้ได้แก่ การกระโดดข้ามเป็นจำนวนรอบ

อินพุตสตรีม (InputStream) ที่ถูกนิยามไว้ในโครงสร้างของคลาสอินพุตสตรีม ตัวอย่างของตัวดำเนินการประเภทนี้ได้แก่ การกระโดดข้ามเป็นจำนวนไบต์ของข้อมูลที่รับเข้ามา

2.2.6 งานวิจัย A methodology for Validating Digital Circuits with Mutation Testing [19]

ในงานวิจัยชิ้นนี้ได้นำเสนอแนวทางในการนำเอาการวิเคราะห์มิวเทชันมาประยุกต์ใช้ร่วมกับภาษาสำหรับการออกแบบวงจรต่างๆ เช่น VHDL และ Verilog และได้สร้างเครื่องมือในการจำลองข้อผิดพลาดให้กับโปรแกรมภาษา VHDL เพื่อใช้วัดคุณภาพของกรณีทดสอบที่ถูกสร้างขึ้นมา ทั้งยังออกแบบตัวดำเนินการมิวเทชันที่เหมาะสมกับการทดสอบทางด้านฮาร์ดแวร์ เนื่องจากตัวดำเนินการมิวเทชันที่มีอยู่เดิมนั้นสร้างโปรแกรมมิวแทนท์ที่ซ้ำซ้อนและประกอบด้วยโปรแกรมมิวแทนท์สมมูลที่เป็นปัญหาสำหรับการทดสอบเป็นจำนวนมาก ตัวอย่างของตัว

ดำเนินการมิวเทชันที่งานวิจัยนี้เสนอได้แก่ การทดสอบขอบเขตของค่าต่างๆ โดยการเพิ่มค่าของตัวแปรอีกเล็กน้อย ตัวดำเนินการในการเปลี่ยนค่าที่เก็บอยู่ในอาร์เรย์เพื่อจำลองเหตุการณ์ที่อาจเกิดขึ้นกับข้อมูลในรีจิสเตอร์ การสร้างการวนรอบที่มีจำนวนรอบเป็นค่าอนันต์ เป็นต้น

2.2.7 งานวิจัย An Extended Overview of the Mothra Software Testing Environment [6]

งานวิจัยชิ้นนี้เป็นงานวิจัยชิ้นแรกที่ได้นำหลักการวิเคราะห์มิวเทชันมาสร้างเป็นระบบที่สามารถใช้งานได้จริง โดยโปรแกรมที่นำมาทดสอบนั้นจะอยู่ในรูปของภาษา Fortran ซึ่งในระบบนี้ได้รวมการสร้างกรณีทดสอบด้วยหลักการของการทดสอบโดยอิงข้อบังคับ พร้อมทั้งได้ทำการทดลองเปรียบเทียบอัตราการกำจัดโปรแกรมมิวแทนท์ด้วยวิธีการทดสอบโดยอิงข้อบังคับกับการสร้างกรณีทดสอบแบบอื่นๆ ดังต่อไปนี้ การวิเคราะห์ตามคำสั่ง (Statement Analysis) การวิเคราะห์ตามข้อกำหนด (Specification Analysis) การวิเคราะห์ตามทางแยก (Branch Coverage) การวิเคราะห์ตามขอบเขต (Domain Analysis) จากผลการทดลองพบว่าการสร้างกรณีทดสอบด้วยหลักการทดสอบโดยอิงข้อบังคับให้คะแนนมิวเทชัน 99.7 เปอร์เซ็นต์ ซึ่งเป็นค่าที่มากที่สุดเมื่อเทียบกับการกำจัดโปรแกรมมิวแทนท์ของการสร้างกรณีทดสอบด้วยหลักการแบบอื่น ถึงอย่างไรก็ตามจำนวนของกรณีทดสอบที่ได้รับจากวิธีการทดสอบโดยอิงข้อบังคับมีจำนวนมากกว่าวิธีแบบอื่นๆ

2.2.8 งานวิจัย An Empirical Evaluation of Weak Mutation [13]

การปรับปรุงการวิเคราะห์มิวเทชันด้วยการวิเคราะห์มิวเทชันแบบอ่อนนั้นอาศัยหลักการที่ว่า ขอให้สถานะ (State) ของค่าต่างๆ ภายหลังจากการประมวลผลคำสั่งที่ได้รับการใส่ข้อผิดพลาดเข้าไปแตกต่างไปจากสถานะของคำสั่งเดียวกันในโปรแกรมที่นำมาทดสอบ ผลที่ตามมาคือโปรแกรมมิวแทนท์ต่างๆ สามารถถูกกำจัดได้ง่ายกว่าเดิม ข้อเสียของวิธีนี้คือกรณีทดสอบที่ได้รับนั้นอาจจะมีประสิทธิผลลดลงเมื่อเทียบกับกรณีทดสอบที่ได้รับจากการวิเคราะห์มิวเทชันแบบเดิม ดังนั้นในงานวิจัยชินนี้ได้ทำการทดลองและเปรียบเทียบประสิทธิผลของกรณีทดสอบที่ได้รับจากการวิเคราะห์มิวเทชันแบบอ่อนโดยการนำเอากรณีทดสอบที่ได้รับมาจากการวิเคราะห์มิวเทชันแบบอ่อนนี้ไปทดสอบกับโปรแกรมมิวแทนท์ด้วยหลักการวิเคราะห์มิวเทชันแบบเดิม จากการทดลองพบว่ากรณีทดสอบที่ได้รับจากการวิเคราะห์มิวเทชันแบบอ่อนที่มีคะแนนมิวเทชันเป็น 100 % สำหรับการวิเคราะห์มิวเทชันแบบอ่อนสามารถเป็นตัวแทนของกรณีทดสอบที่ได้รับจากการวิเคราะห์มิวเทชันแบบแข็งได้

2.2.9 งานวิจัย Mutation of Model Checker Specifications for Test Generation and Evaluation [3]

ในวิจัยชิ้นนี้ได้นำเสนอการนำเอาการวิเคราะห์มิวเทชันมาประยุกต์ใช้ในการตรวจสอบข้อกำหนดทางซอฟต์แวร์ (Specification-based mutation analysis) ซึ่งการทำการวิเคราะห์มิวเทชันนั้นจำเป็นต้องมีการแยกความแตกต่างระหว่างข้อกำหนดที่แท้จริงกับข้อกำหนดที่ได้รับการใส่ข้อผิดพลาดเข้าไป การตรวจสอบแบบจำลอง (Model checking) เป็นวิธีการหนึ่งที่ใช้ในการเปรียบเทียบข้อกำหนดต่างๆ ว่ามีความสอดคล้องกันหรือไม่ โดยอาศัยหลักการของการเปลี่ยนแปลงของสถานะ หากข้อกำหนดที่นำมาตรวจสอบไม่สอดคล้องกันแล้วการตรวจสอบแบบจำลองจะแสดงตัวอย่างที่ทำให้เกิดความขัดแย้งกันขึ้น ดังนั้นในงานวิจัยชิ้นนี้ได้นำเอาการตรวจสอบแบบจำลองที่มีชื่อว่า SMV มาเป็นเครื่องมือในการแยกความแตกต่างระหว่างข้อกำหนดที่แท้จริงและข้อกำหนดที่ได้รับการใส่ข้อผิดพลาด จุดประสงค์หลักของงานวิจัยชิ้นนี้เพื่อวิเคราะห์ความสามารถในการตรวจจับผลกระทบของข้อผิดพลาดแต่ละประเภทที่ถูกสร้างขึ้นมาจากตัวดำเนินการมิวเทชัน โดยความสัมพันธ์ของความสามารถในการตรวจจับผลกระทบของข้อผิดพลาดที่ได้รับมาจากการทดลองในงานวิจัยชิ้นนี้ คือ หากสามารถตรวจจับผลกระทบของข้อผิดพลาดที่เกิดจากตัวดำเนินการมิวเทชันประเภทการเปลี่ยนตัวถูกดำเนินการ (Operand Replacement Operator) ได้แล้วจะสามารถเตรวจจับผลกระทบของข้อผิดพลาดที่เกิดจากตัวดำเนินการมิวเทชันที่กลับค่าความจริงของเอ็กซ์เพรสชันแบบง่าย (Simple Expression Negation) และหากสามารถตรวจจับผลกระทบของข้อผิดพลาดที่เกิดจากตัวดำเนินการมิวเทชันที่กลับค่าความจริงของเอ็กซ์เพรสชันแบบง่ายได้แล้วจะสามารถตรวจจับผลกระทบของข้อผิดพลาดที่เกิดจากตัวดำเนินการมิวเทชันที่กลับค่าความจริงของเอ็กซ์เพรสชันทั่วไปได้ด้วย (Expression Negation) ดังนั้นหากใช้ตัวดำเนินการประเภทการเปลี่ยนตัวถูกดำเนินการในการสร้างข้อผิดพลาดแล้วจะสามารถละข้อผิดพลาดที่เกิดจากตัวดำเนินการที่กลับค่าความจริงของเอ็กซ์เพรสชันแบบต่างๆ ได้

บทที่ 3
การวิเคราะห์มิวเทชันแบบหลายข้อผิดพลาด

Demillo [5] ได้เสนอทฤษฎี คอปปลิงเอฟเฟ็ค (Coupling Effect Theorem) ซึ่งกล่าวว่า ชุดทดสอบใดๆ ก็ตามที่สามารถหาข้อผิดพลาดที่มีอยู่ในโปรแกรมเพียงที่เดียวได้แล้วจะมีความสามารถเพียงพอในการหาข้อผิดพลาดที่มีหลายที่ได้ โดยการวิเคราะห์มิวเทชันนั้นอาศัยทฤษฎีข้อนี้ ทำให้แต่ละโปรแกรมมิวแทนท์เกิดจากข้อผิดพลาดเพียงที่เดียวและไม่สนใจโปรแกรมมิวแทนท์ที่เกิดจากข้อผิดพลาดหลายที่ โดย Offutt ได้ทำการทดลองเพื่อยืนยันทฤษฎีนี้ใน [12] ถึงอย่างไรก็ตามโปรแกรมมิวแทนท์ที่เกิดจากข้อผิดพลาดเดียวยังคงมีจำนวนมากอยู่ ดังนั้นจึงเป็นประเด็นให้หลายงานวิจัยมุ่งเน้นในการลดจำนวนโปรแกรมมิวแทนท์ลง แต่ส่วนใหญ่แล้วประสิทธิผลในการทดสอบลดลงตามไปด้วย

ในงานวิจัยที่ผ่านมาที่เกี่ยวข้องกับการนำเอาการตัดส่วนโปรแกรมมาช่วยในการวิเคราะห์มิวเทชันนั้นมีในงานวิจัยที่ [9] ซึ่งเป็นการนำเอาการตัดส่วนโปรแกรมมาช่วยตรวจสอบโปรแกรมมิวแทนท์สมมูล และในงานวิจัยที่ [10] เป็นการลดจำนวนกรณีทดสอบที่ต้องนำมาทดสอบแต่ละโปรแกรมมิวแทนท์ลง ตามที่ได้กล่าวรายละเอียดมาแล้ว

ในบทนี้จะเสนอแนวทางในการลดจำนวนโปรแกรมมิวแทนท์ในการวิเคราะห์มิวเทชัน โดยการใช้การวิเคราะห์มิวเทชันแบบหลายข้อผิดพลาด ซึ่งวิธีนี้สามารถรักษาระดับประสิทธิผลของกรณีทดสอบไว้ได้

3.1 การสร้างโปรแกรมมิวแทนท์แบบหลายข้อผิดพลาด

วิทยานิพนธ์ชิ้นนี้ได้ประยุกต์ใช้การตัดส่วนโปรแกรมมาวิเคราะห์ว่าคำสั่งใดบ้างที่ส่งผลกระทบต่อบรรทัดฐานการตัด โดยตัวแปรในบรรทัดฐานการตัดคือ แต่ละผลลัพธ์ของโปรแกรม เมื่อทราบว่าคำสั่งใดกระทบต่อผลลัพธ์ที่สนใจแล้ว จะสามารถแบ่งข้อผิดพลาดออกเป็นกลุ่มตามแต่ละผลลัพธ์ของโปรแกรม โดยผลกระทบของข้อผิดพลาดภายในกลุ่มเดียวกันจะส่งผลกระทบต่อผลลัพธ์ตัวเดียวกันเท่านั้น เพื่อนำมาสร้างเป็นโปรแกรมมิวแทนท์แบบหลายข้อผิดพลาดต่อไป

3.1.1 นิยามของโปรแกรมมิวแทนท์แบบหลายข้อผิดพลาด

โปรแกรมมิวแทนท์แบบหลายข้อผิดพลาดนี้เกิดจากการใส่ข้อผิดพลาดในหลายตำแหน่ง เพื่อให้โปรแกรมเพียงโปรแกรมเดียวสามารถเป็นตัวแทนของหลายโปรแกรมมิวแทนท์ โดยได้มีการนิยามโปรแกรมมิวแทนท์แบบหลายข้อผิดพลาดไว้ดังต่อไปนี้

นิยามที่ 7

กำหนดให้ P เป็นโปรแกรมทดสอบที่ประกอบด้วยคำสั่ง s_1, s_2, s_3,…,s_n *และ C เป็นโปรแกรมมิวแทนท์แบบหลายข้อผิดพลาด ถ้ามีจำนวนเต็มบวก k ใดๆ ที่*

$$P \overset{u_1}{\underset{s_1}{\mapsto}} M_1 \overset{u_2}{\underset{s_2}{\mapsto}} M_2 \overset{u_3}{\underset{s_3}{\mapsto}} M_3 \ldots \overset{u_k}{\underset{s_k}{\mapsto}} M_k = C$$

โดย $P \overset{u_j}{\underset{s_j}{\mapsto}} M$ *เป็นการใส่ข้อผิดพลาด* u_j *ที่คำสั่ง* s_j

3.1.2 อัลกอริธึมในการสร้างโปรแกรมมิวแทนท์แบบหลายข้อผิดพลาด

อัลกอริธึมที่ใช้ในการสร้างโปรแกรมมิวแทนท์แบบหลายข้อผิดพลาด กำหนดให้โปรแกรมทดสอบ P มี n ผลลัพธ์ คือ o_1, o_2, o_3, …, o_n

ขั้นตอนที่ 1 สร้างข้อผิดพลาดที่จะนำไปใส่ให้กับโปรแกรมที่นำมาทดสอบ โดยชนิดของข้อผิดพลาดจะถูกนิยามไว้ในตัวดำเนินการมิวเทชัน ซึ่งในงานวิจัยนี้เลือกตัวดำเนินการมิวเทชันเป็นแบบอีซีเล็คทีฟ

ขั้นตอนที่ 2 ทำการตัดส่วนของโปรแกรม P ออกเป็น n โปรแกรมย่อย ซึ่งเรียกว่า P_1, P_2, P_3, …, P_n โดยที่แต่ละ P_j ที่$1 \leq j \leq n$ นั้นประกอบด้วยคำสั่งที่สามารถส่งผลกระทบต่อ o_j สำหรับแต่ละ P_i และ P_j ที่ $i \neq j$ นั้น ให้ตัดคำสั่งที่เป็นคำสั่งร่วมกันซึ่งส่งผลกระทบต่อทั้ง o_i และ o_j จาก P_i และ P_j

ขั้นตอนที่ 3 วิเคราะห์แต่ละข้อผิดพลาดที่จะใส่ให้กับโปรแกรมที่นำมาทดสอบว่าเมื่อใส่ข้อผิดพลาดนี้แล้วโปรแกรมมิวแทนท์ที่เกิดขึ้นเป็นโปรแกรมมิวแทนท์สมมูลหรือไม่ หากเป็นโปรแกรมมิวแทนท์สมมูลเราจะไม่พิจารณาข้อผิดพลาดนี้

ขั้นตอนที่ 4 จากโปรแกรมที่ได้รับมาจากการตัดส่วน P_i ถึง P_n เราสามารถแบ่งกลุ่มข้อผิดพลาดที่จะใส่ให้กับ P ออกเป็น n กลุ่ม โดยกำหนดให้ข้อผิดพลาดกลุ่มที่ i (SM_i) มีจำนวนข้อผิดพลาดที่เป็นสมาชิกอยู่ทั้งหมด $|SM_i|$

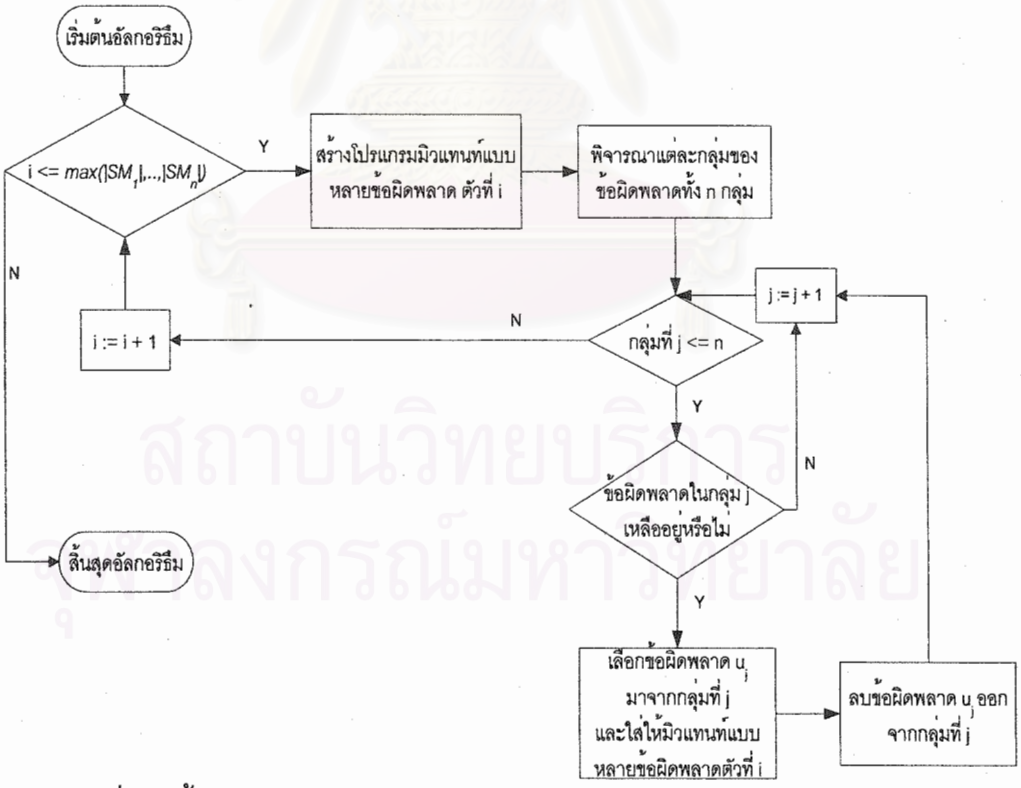
ขั้นตอนที่ 5 สร้างโปรแกรมมิวแทนท์แบบหลายข้อผิดพลาดตามอัลกอริธึมด้านล่างนี้ โดยการเลือกข้อผิดพลาดจากแต่ละกลุ่มมาสร้างเป็นโปรแกรมมิวแทนท์แบบหลายข้อผิดพลาดจะทำจนกระทั่งไม่มีข้อผิดพลาดใดๆ เหลืออยู่ในทุกๆ กลุ่ม

for i=1 to max(|SM_1|,|SM_2|,...,|SM_n|) do
M= P
for j=1 to n do
if |SM_j|<> 0 then
M' = insert u_j at statement s_j to M //Select and insert fault from SM_j
SM_j=SM_j\{u_j}
endif;
endfor;
C_i = M
endfor;
output $Mutant_1$ = {C_1,C_2,..., C_n}

ขั้นตอนที่ 6 รวมโปรแกรมมิวแทนท์ทั้งหมด ซึ่งเกิดจากข้อผิดพลาดที่ใส่ให้กับคำสั่งร่วมกัน (ที่ตัดออกในขั้นตอนที่ 2) $Mutant_2$={M_1, M_2, ...,M_k} เข้ากับ $Mutant_1$= {C_1,C_2,..., C_n}

ผลลัพธ์ที่ต้องการ คือ ชุดของโปรแกรมมิวแทนท์สุดท้ายที่เกิดจากชุดของโปรแกรมมิวแทนท์แบบหลายข้อผิดพลาดรวมกับชุดของโปรแกรมมิวแทนท์ที่เกิดจากการใส่ข้อผิดพลาดที่คำสั่งร่วมกัน

รูปที่ 3.1 แสดงถึงขั้นตอนการทำงานของอัลกอริธึมในขั้นตอนที่ 5



รูปที่ 3.1 ขั้นตอนในการสร้างโปรแกรมมิวแทนท์แบบหลายข้อผิดพลาด

เหตุผลสำคัญที่โปรแกรมมิวแทนท์ที่ถูกรวมอยู่ในโปรแกรมมิวแทนท์แบบหลายข้อผิดพลาดเป็นอิสระจากกันเมื่อทำตามวิธีการสร้างโปรแกรมมิวแทนท์แบบหลายข้อผิดพลาดข้างต้น มีอยู่ด้วยกัน 2 ประการคือ

1. การใช้วิธีการตัดส่วนโปรแกรมมาวิเคราะห์ว่าคำสั่งใดส่งผลกระทบต่อผลลัพธ์ตัวใดบ้าง ข้อผิดพลาดที่ใส่ให้คำสั่งที่ส่งผลกระทบต่อผลลัพธ์ที่ต่างกันจึงเป็นอิสระจากกัน
2. ชนิดของตัวดำเนินการมิวเทชันที่เลือกมาใช้นั้นเป็นแบบอีซีเล็คทีฟ ซึ่งไม่สามารถสร้างข้อผิดพลาดที่ทำให้คำสั่งที่ถูกใส่ข้อผิดพลาดไปแล้วนั้นกระทบต่อผลลัพธ์ที่ต่างไปจากเดิม

ตัวอย่างของโปรแกรมมิวแทนท์แบบหลายข้อผิดพลาดถูกแสดงไว้ในตัวอย่างที่ 3.1

ตัวอย่างที่ 3.1 *จากตัวอย่างที่ 2.4 ในเรื่องของการตัดส่วนโปรแกรม โดยกำหนดให้โปรแกรม Prod_Sum(int n) ถูกตัดส่วนเป็นสองโปรแกรมตามบรรทัดฐานการตัดคือ* S_s*=({sum},012) (ในตัวอย่างที่ 2.4) และ* S_p*=({product,013)ซึ่งแสดงไว้ดังต่อไปนี้*

```
001  Procedure Prod_Sum(n: Integer);
002  var i,n, sum, product : Integer;
003  begin
004          i := 1;
006          product := 1;
007          while (i<=n) do
008          begin
010                  product := product * i;
011                  i := i + 1;
012          end;
014          writeln(product);
015  end;
```

ทั้ง S_s และ S_p มีคำสั่งที่ร่วมกันคือคำสั่งในแถวที่ 003, 006 และ 010 ข้อผิดพลาดที่ใส่ให้คำสั่งที่ไม่ใช่คำสั่งร่วมกันของทั้ง S_s และ S_p นั้นสามารถรวมไว้ในโปรแกรมเดียวกันได้ เพื่อสร้างเป็นโปรแกรมมิวแทนท์แบบหลายข้อผิดพลาด ดังต่อไปนี้

โปรแกรมมิวแทนท์ M_1 โดยเปลี่ยนที่คำสั่งที่ 8 จาก sum = sum + i เป็น sum = sum * i;

```
001  Procedure Prod_Sum(n: Integer);
002  var i,n, sum, product : Integer;
003  begin
```

```
004         i := 1;
005         sum := 0;
006         product := 1;
007         while (i<=n) do
008         begin
009                 sum := sum * i; // Arithmetic Operator Replacement
010                 product := product * i;
011                 i := i + 1;
012         end;
013         writeln(sum);
014         writeln(product);
015 end;
```

โปรแกรมมิวแทนท์ M_2 โดยการเปลี่ยนที่คำสั่งที่ 9 จาก product = product * i; เป็น product = product + i;

```
001 Procedure Prod_Sum(n: Integer);
002 var i,n, sum, product : Integer;
003 begin
004         i := 1;
005         sum := 0;
006         product := 1;
007         while (i<=n) do
008         begin
009                 sum := sum + i;
010                 product := product + i; // Arithmetic Operator Replacement
011                 i := i + 1;
012         end;
013         writeln(sum);
014         writeln(product);
015 end;
```

โปรแกรมมิวแทนท์แบบหลายข้อผิดพลาด C โดยการเปลี่ยนที่คำสั่งที่ 8 จาก sum = sum + i; เป็น sum = sum * i; และเปลี่ยนที่คำสั่งที่ 9 จาก product = product * i; เป็น product = product + i;

```
001 Procedure Prod_Sum(n: Integer);
002 var i,n, sum, product : Integer;
003 begin
004         i := 1;
005         sum := 0;
```

```
006         product := 1;
007         while (i<=n) do
008         begin
009                 sum := sum * i; // Arithmetic Operator Replacement
010                 product := product + i; // Arithmetic Operator Replacement
011                 i := i + 1;
012         end;
013         writeln(sum);
014         writeln(product);
015 end;
```

สังเกตได้ว่าโปรแกรมมิวแทนท์แบบหลายข้อผิดพลาด C เกิดจากข้อผิดพลาดของทั้ง M_1 และ M_2

3.1.3 การกำจัดโปรแกรมมิวแทนท์แบบหลายข้อผิดพลาด

กรณีทดสอบถูกใช้สำหรับการประมวลผลโปรแกรมมิวแทนท์ ถ้าผลลัพธ์สุดท้ายที่ได้รับแตกต่างจากผลลัพธ์ที่คาดหวัง โปรแกรมมิวแทนท์ตัวนั้นจะถูกกำจัด เนื่องจากโปรแกรมมิวแทนท์แบบหลายข้อผิดพลาดเกิดจากการใส่ข้อผิดพลาดเข้าไปมากกว่าหนึ่งข้อผิดพลาด ในการกำจัดโปรแกรมมิวแทนท์แบบหลายข้อผิดพลาดนี้จำเป็นต้องตรวจจับผลกระทบของทุกข้อผิดพลาดที่ใส่ให้กับโปรแกรมที่นำมาทดสอบเพื่อสร้างเป็นโปรแกรมมิวแทนท์แบบหลายข้อผิดพลาด การที่โปรแกรมมิวแทนท์แบบหลายข้อผิดพลาดจะถูกกำจัดได้ด้วยกรณีทดสอบเดียวก็ต่อเมื่อ กฎข้อบังคับในการตรวจจับผลกระทบของแต่ละข้อผิดพลาดที่ใส่ให้นั้นไม่มีข้อขัดแย้งระหว่างกัน

นิยามที่ 8

กำหนดให้ P เป็นโปรแกรมทดสอบที่ประกอบด้วยผลลัพธ์จำนวน n ผลลัพธ์ สำหรับแต่ละจำนวนเต็มบวก k ที่ $k \leq n$ *ใดๆ ที่* $1 \leq j \leq k$

$$P \overset{u_j}{\underset{s_j}{\mapsto}} N_j$$

และมีกรณีทดสอบ t ใดๆ ที่ โปรแกรมมิวแทนท์ N_j *ถูกประมวลผลด้วยกรณีทดสอบ t (* $N_j(t)$ *) แล้วให้ผลลัพธ์ที่ต่างจาก P(t) เฉพาะผลลัพธ์ตัวที่ j เท่านั้น โดยโปรแกรมมิวแทนท์แบบหลายข้อผิดพลาด C ใดๆ ที่*

$$P \overset{u_1}{\underset{s_1}{\mapsto}} M_1 \overset{u_2}{\underset{s_2}{\mapsto}} M_2 \overset{u_3}{\underset{s_3}{\mapsto}} M_3 \ldots \overset{u_k}{\underset{s_k}{\mapsto}} M_k = C$$

จะถูกกำจัดด้วยชุดของกรณีทดสอบ $T = \{t_1, t_2, \ldots, t_m\}$ *ถ้าทุกๆ j ใดๆ ที่* $1 \leq j \leq k$ *จะมี* t_i *ใดๆ ที่เมื่อ* $C(t_i)$ *ให้ผลลัพธ์ที่แตกต่างจาก* $P(t_i)$ *ณ ผลลัพธ์ j*

จากตัวอย่างที่ 3.1 หากกำหนดให้กรณีทดสอบ t = (n=5) แล้วผลลัพธ์สุดท้ายของโปรแกรมมิวแทนท์ต่างๆ เป็นดังตารางที่ 3.1

ตารางที่ 3.1 ผลลัพธ์สุดท้ายของแต่ละโปรแกรมมิวแทนท์

ผลลัพธ์ เมื่อ n=5	sum	product
P	15	120
M_1	0	120
M_2	15	16
Multiple-faults mutant C	0	16

สังเกตเห็นว่าข้อผิดพลาดของทั้ง M_1 และ M_2 สามารถถูกแทนด้วยโปรแกรมมิวแทนท์แบบหลายข้อผิดพลาด C ซึ่งสามารถกำจัดได้ด้วยกรณีทดสอบเพียงครั้งเดียว ดังนั้นจำนวนครั้งในการทดสอบโปรแกรมมิวแทนท์สามารถลดลงได้ดังในตัวอย่าง

3.1.4 อัลกอริธึมในการกำจัดโปรแกรมมิวแทนท์แบบหลายข้อผิดพลาด

กำหนดให้ชุดของโปรแกรมมิวแทนท์แบบหลายข้อผิดพลาด $M_c = \{C_1,.., C_n\}$ อัลกอริธึมในการกำจัดโปรแกรมมิวแทนท์แบบหลายข้อผิดพลาดเป็นดังต่อไปนี้

ขั้นตอนที่ 1 สร้างกรณีทดสอบ t_i เพื่อมากำจัดโปรแกรมมิวแทนท์แบบหลายข้อผิดพลาด (ในการสร้างกรณีทดสอบนั้นสามารถใช้เทคนิคต่างๆ เช่น การทดสอบโดยอิงข้อบังคับ การจำกัดขอบเขตแบบพลวัต (Dynamic-Domain Reduction) [17] หรือการทดสอบแบบสุ่ม)

ขั้นตอนที่ 2 สำหรับแต่ละ C_i ที่ถูกสร้างมาจากการใส่ข้อผิดพลาด F_1 ถึง F_k

ขั้นตอนที่ 2.1 หาก t_i สามารถตรวจจับผลกระทบของข้อผิดพลาด F_1 ถึง F_j โดยที่ $j < k$ แล้วเก็บกรณีทดสอบ t_i ไว้ พร้อมทั้งติดฉลากไว้ว่าผลกระทบของ F_1 ถึง F_j ของ C_i ถูกตรวจจับเรียบร้อยแล้ว

ขั้นตอนที่ 2.2 หาก t_i สามารถตรวจจับผลกระทบของข้อผิดพลาด F_1 ถึง F_j โดยที่ $j = k$ แล้ว (t_i สามารถตรวจจับทุกผลกระทบของทุกข้อผิดพลาดที่ใส่ให้กับโปรแกรมมิวแทนท์แบบหลายข้อผิดพลาด C_i) ลบ C_i ออกจากชุดของโปรแกรมมิวแทนท์แบบหลายข้อผิดพลาด M_c

ขั้นตอนที่ 3 นำ t_i ไปทดสอบกับโปรแกรมมิวแทนท์แบบหลายข้อผิดพลาดตัวอื่น C_{i+1} จากนั้นกลับไปทำยังขั้นตอนที่ 2

ขั้นตอนที่ 4 หาก $M_c = Ø$ เป็นการสิ้นสุดอัลกอริธึม แต่ถ้าหาก $M_c \neq Ø$ ให้กลับไปทำยังขั้นตอนที่ 1 เพื่อสร้างกรณีทดสอบตัวถัดไป t_{i+1} เพื่อมากำจัดโปรแกรมมิวแทนท์แบบหลายข้อผิดพลาดที่เหลืออยู่ (โดยการตรวจจับข้อผิดพลาดที่ได้รับการติดฉลากไว้ในรอบก่อนหน้านี้)

ผลลัพธ์ที่ต้องการ คือ โปรแกรมมิวแทนท์แบบหลายข้อผิดพลาดถูกกำจัดออกไปจนหมด พร้อมทั้งได้รับกรณีทดสอบที่มีความสามารถในการตรวจจับผลกระทบของข้อผิดพลาดที่สร้างขึ้นได้

ปัญหาสำคัญที่เกิดขึ้นในการทำการวิเคราะห์มิวเทชันแบบหลายข้อผิดพลาด คือ เมื่อใส่ข้อผิดพลาดต่างๆ ให้กับโปรแกรมที่นำมาทดสอบแล้ว อาจเกิดกรณีที่ข้อบังคับในการตรวจจับผลกระทบของข้อผิดพลาดต่างๆ มีความขัดแย้งกัน ส่งผลให้การกำจัดโปรแกรมมิวแทนท์แบบหลายข้อผิดพลาดชนิดนี้ต้องอาศัยกรณีทดสอบจำนวนมากในการกำจัดโปรแกรมมิวแทนท์แบบหลายข้อผิดพลาด

ปัญหาข้อถัดมาสำหรับการสร้างโปรแกรมมิวแทนท์แบบหลายข้อผิดพลาดคือ หากโปรแกรมประกอบด้วยผลลัพธ์เพียงผลลัพธ์เดียวจะไม่สามารถลดจำนวนโปรแกรมมิวแทนท์ได้

3.2 ทฤษฎีและการพิสูจน์การวิเคราะห์มิวเทชันแบบหลายข้อผิดพลาด

ในการสร้างโปรแกรมมิวแทนท์แบบหลายข้อผิดพลาดเพื่อลดจำนวนของโปรแกรมมิวแทนท์ งานวิจัยชิ้นนี้ได้เสนอแนวทางในการพิสูจน์พร้อมทั้งการออกแบบการทดลองเพื่อยืนยันแนวความคิดในการลดจำนวนโปรแกรมมิวแทนท์โดยอาศัยหลักการสร้างโปรแกรมมิวแทนท์แบบหลายข้อผิดพลาด ดังต่อไปนี้

3.2.1 ผลกระทบของข้อผิดพลาดที่ใส่ให้โปรแกรมมิวแทนท์แบบหลายข้อผิดพลาด

การสร้างโปรแกรมมิวแทนท์แบบหลายข้อผิดพลาดโดยการใส่ข้อผิดพลาดหลายข้อผิดพลาดให้กับโปรแกรมที่นำมาทดสอบ ข้อผิดพลาดที่ใส่เข้าไปนั้นจำเป็นต้องเป็นอิสระจากกัน โดยผลกระทบของข้อผิดพลาดต่างๆ ที่ใส่เข้าไปในโปรแกรมที่นำมาทดสอบจำเป็นต้องกระทบต่อผลลัพธ์ของโปรแกรมที่นำมาทดสอบที่ต่างกัน ในหัวข้อนี้จะแสดงถึงการพิสูจน์ว่าตามอัลกอริธึมการสร้างโปรแกรมมิวแทนท์แบบหลายข้อผิดพลาดในหัวข้อที่ 3.1.2 นั้นสามารถสร้างโปรแกรมมิวแทนท์ที่เกิดจากการใส่ข้อผิดพลาดที่เป็นอิสระจากกัน

ทฤษฎีบทที่ 1

กำหนดให้ P เป็นโปรแกรมที่นำมาทดสอบ และ C เป็นโปรแกรมมิวแทนท์แบบหลายข้อผิดพลาดที่ได้รับมาจากอัลกอริธึมในการสร้างโปรแกรมมิวแทนท์แบบหลายข้อผิดพลาด

$$P \underset{s_1}{\overset{u_1}{\Rightarrow}} M_1 \underset{s_2}{\overset{u_2}{\Rightarrow}} M_2 \underset{s_3}{\overset{u_3}{\Rightarrow}} M_3 \ldots \underset{s_j}{\overset{u_j}{\Rightarrow}} M_j = C$$

โดย u_j ใดๆ คือ ข้อผิดพลาดที่ถูกใส่ให้กับคำสั่งที่ S_j และผลกระทบของข้อผิดพลาด u_j ส่งผลกระทบต่อผลลัพธ์ตัวที่ j (O_j) เท่านั้น

หากกรณีทดสอบ t ใดๆ สามารถแยกความแตกต่างของผลลัพธ์ตัวที่ j (O_j) ของโปรแกรมที่นำมาทดสอบเมื่อได้รับการทดสอบกับกรณีทดสอบ t (P(t)) จากโปรแกรมมิวแทนท์เมื่อได้รับการทดสอบด้วยกรณีทดสอบ t เดียวกัน ($N_j(t)$) โดยโปรแกรมมิวแทนท์นี้เกิดจาก

$$P \underset{s_j}{\overset{u_j}{\Rightarrow}} N_j \text{ , } 1 \le j \le k$$

แล้วโปรแกรมมิวแทนท์แบบหลายข้อผิดพลาดเมื่อได้รับการทดสอบกับกรณีทดสอบ t C(t) จะต้องให้ผลลัพธ์ที่แตกต่างจาก P(t) ที่ผลลัพธ์ตัวที่ j เท่านั้น

พิสูจน์

สมมุติให้กรณีทดสอบ t สามารถแยกความแตกต่างของผลลัพธ์ O_j ของ P(t) จาก $N_j(t)$ ได้ โดยการพิสูจน์แบบขัดแย้งสมมุติให้ C(t) ให้ผลลัพธ์ที่ไม่แตกต่างไปจาก P(t) ที่ผลลัพธ์ O_j จะต้องพิสูจน์ให้ได้ว่าเกิดข้อขัดแย้งขึ้น

เนื่องจากโปรแกรมมิวแทนท์ N_j ต่างจากโปรแกรมที่นำมาทดสอบ P ที่คำสั่ง S_j ด้วยการใส่ข้อผิดพลาด u_j และข้อผิดพลาด u_j นี้ยังได้ถูกใส่ให้กับโปรแกรมมิวแทนท์แบบหลายข้อผิดพลาด C

จากสมมุติฐานที่กำหนดว่า สมมุติให้ C(t) ให้ผลลัพธ์ที่ไม่แตกต่างไปจาก P(t) ที่ผลลัพธ์ O_j ดังนั้นเราสามารถบอกได้ว่า C จะต้องมีคำสั่งที่ได้รับการใส่ข้อผิดพลาดคำสั่งหนึ่ง (S_0) ที่ $S_0 \neq S_j$ โดยที่ผลกระทบของข้อผิดพลาดที่คำสั่ง S_0 กระทบต่อ O_j เช่นเดียวกับข้อผิดพลาดที่ใส่ที่ S_j

สรุปได้ว่า S_0 และ S_j จะต้องอยู่ในส่วนของโปรแกรมที่ได้รับการตัดส่วนเดียวกันซึ่งมีการนิยามบรรทัดฐานในการตัดไว้ที่ตัวแปร O_j ทำให้ขัดแย้งกับอัลกอริธึมในการสร้างโปรแกรมมิวแทนท์แบบหลายข้อผิดพลาดที่กล่าวว่าให้เลือกข้อผิดพลาดหนึ่งข้อผิดพลาดมาจากแต่ละกลุ่ม

3.2.2 ประสิทธิผลของกรณีทดสอบ

ประสิทธิผลของกรณีทดสอบในการทำการวิเคราะห์มิวเทชันนั้น ขึ้นอยู่กับกรณีทดสอบที่สร้างขึ้นสามารถกำจัดโปรแกรมมิวแทนท์ได้มากหรือไม่ มีหลายงานวิจัยที่เสนอแนวทางในการปรับปรุงประสิทธิภาพของการวิเคราะห์มิวเทชันโดยการลดจำนวนโปรแกรมมิวแทนท์ลงไป ด้วยวิธีการนี้ส่วนมากทำให้ประสิทธิผลของชุดของกรณีทดสอบที่ได้มีประสิทธิผลที่ลดลง

ในหัวข้อนี้เป็นการพิสูจน์ถึงประสิทธิผลของชุดกรณีทดสอบที่ได้รับมาจากการวิเคราะห์มิวเทชันแบบหลายข้อผิดพลาดว่ามีประสิทธิผลที่ไม่ลดลง

ทฤษฎีบทที่ 2

ชุดของกรณีทดสอบ T ใดๆ ที่สามารถกำจัดโปรแกรมมิวแทนท์ในวิธีการแบบเดิมได้แล้ว จะมีความสามารถเพียงพอที่จะกำจัดโปรแกรมมิวแทนท์แบบหลายข้อผิดพลาด

พิสูจน์

กำหนดให้

- F = {u_1, u_2, u_3,…,u_n} เป็นชุดของข้อผิดพลาดที่ใส่ให้กับโปรแกรมที่นำมาทดสอบ P
- C เป็นโปรแกรมมิวแทนท์แบบหลายข้อผิดพลาดใดๆ ก็ตามที่สร้างมาจากการใส่ข้อผิดพลาดที่มีอยู่ใน F หลายข้อผิดพลาดให้กับ P
- T = {t_1, t_2, t_3, …,t_m} เป็นชุดของกรณีทดสอบที่สามารถตรวจจับผลกระทบของทุกข้อผิดพลาดที่มีอยู่ใน F

เราสามารถพิสูจน์ทฤษฎีบทที่ 2 ได้โดยสมมุติว่าเมื่อเรานำทุกๆ t_i ใน T มาทดสอบกับ C แล้วยังมีผลลัพธ์ที่ O_j ของ P(t_i) เท่ากับ O_j ของ C(t_i)

ถ้า $u_j \in F$ เป็นข้อผิดพลาดที่ส่งผลกระทบต่อ O_j และถูกใส่ให้กับ C จากข้อสมมุติที่ตั้งไว้ คือ ทุกๆ t_i ใน T เมื่อนำมาทดสอบกับ C แล้วยังมี ผลลัพธ์ที่ O_j ของ P(t_i) เท่ากับ O_j ของ C(t_i) และจาก ทฤษฎีบทที่ 1 ที่กล่าวว่า ti จะต้องตรวจจับ uj ได้ที่ Oj เท่านั้น

จะได้ว่าว่าทุกๆ t_i ใน T ไม่สามารถตรวจจับผลกระทบของข้อผิดพลาด u_j ซึ่งขัดแย้งกับข้อกำหนดที่ว่า T เป็นชุดทดสอบที่สามารถตรวจจับผลกระทบของข้อผิดพลาดได้ทุกข้อผิดพลาดใน F

3.3 การทดลองเพื่อวัดเปอร์เซ็นต์การลดลงของจำนวนโปรแกรมมิวแทนท์

ในงานวิจัยชิ้นนี้ได้เสนอการทดลองเพื่อวัดประสิทธิภาพของการสร้างโปรแกรมมิวแทนท์แบบหลายข้อผิดพลาดโดยในการทดลองนี้เราได้เลือกโปรแกรมภาษาปาสคาลที่แต่ละโปรแกรมนั้นประกอบด้วยผลลัพธ์ที่มากกว่าหนึ่งผลลัพธ์ โปรแกรมที่ถูกเลือกมามีดังในตารางที่ 3.2

ตารางที่ 3.2 ตัวอย่างโปรแกรมภาษาปาสคาล

โปรแกรม	จำนวนผลลัพธ์	คำบรรยาย
SinCos	2	คำนวณค่า Sin และ Cos
MulSumMatrix	2	คำนวณค่าการคูณและบวกเมตริกซ์
MulSumFraction	2	คำนวณค่าการคูณและบวกเศษส่วน
Mean	3	คำนวณค่า Harmonic Mean, Mean และรากที่สองของผลบวกกำลังสอง
MaxMinAvg	3	หาค่า max, min และ average ของจำนวน 5 ค่า

ขั้นตอนในการทำการทดลองเริ่มต้นจากการนิยามตัวดำเนินการมิวเทชัน ซึ่งในงานวิจัยชิ้นนี้ได้เลือกตัวดำเนินการมิวเทชันเป็นแบบอีซีเล็คทีฟ เท่านั้น

ขั้นตอนถัดมาคือวิเคราะห์ข้อผิดพลาดว่ามีข้อผิดพลาดใดที่ทำให้เกิดโปรแกรมมิวแทนท์สมมูล ข้อผิดพลาดเหล่านี้จะไม่นำมาสร้างเป็นโปรแกรมมิวแทนท์ ดังนั้นจำนวนข้อผิดพลาดที่เหลืออยู่จะเป็นจำนวนโปรแกรมมิวแทนท์ในวิธีการวิเคราะห์มิวเทชันแบบเดิม

จากนั้นสร้างโปรแกรมมิวแทนท์แบบหลายข้อผิดพลาดโดยอาศัยอัลกอริธึมในการสร้างโปรแกรมมิวแทนท์ในหัวข้อที่ 3.1.2 แล้วเปรียบเทียบจำนวนโปรแกรมมิวแทนท์แบบหลายข้อผิดพลาดที่ได้รับกับจำนวนโปรแกรมมิวแทนท์ในวิธีการแบบเดิมเพื่อวัดเปอร์เซ็นต์ที่ลดลง (Percentage Saved)

การคำนวณเปอร์เซ็นต์ที่ลดลงของจำนวนมิวแทนท์สามารถคำนวณได้จากสูตรดังต่อไปนี้

$$\text{เปอร์เซ็นต์ที่ลดลง} = \frac{M - E - C}{M - E} \times 100$$

M คือ จำนวนมิวแทนท์ที่ได้รับจากวิธีการวิเคราะห์มิวเทชันแบบเดิม

E คือ จำนวนมิวแทนท์สมมูล

C คือ จำนวนมิวแทนท์แบบหลายข้อผิดพลาด

ผลลัพธ์ที่ได้จากการสร้างโปรแกรมมิวแทนท์แบบหลายข้อผิดพลาดนี้ถูกแสดงไว้ในตารางที่ 3.3

ตารางที่ 3.3 เปอร์เซ็นต์การลดลงของจำนวนโปรแกรมมิวแทนท์

โปรแกรมที่นำมาทดสอบ	มิวแทนท์ในวิธีการเดิม	มิวแทนท์สมมูล	มิวแทนท์แบบหลายข้อผิดพลาด	เปอร์เซ็นต์การลดลง
SinCos	148	8	75	46.43
MulSumMatrix	81	14	42	37.31
MulSumFraction	231	78	81	47.06
Mean	83	6	46	40.26
MaxMinAvg	71	7	28	56.25
Average				45.46

จากผลการทดลองที่ได้รับชี้ให้เห็นว่าจำนวนของโปรแกรมมิวแทนท์ลดลงได้มากที่สุดถึง 56.25 % ในโปรแกรม MaxMinAvg โดยค่าเฉลี่ยของเปอร์เซ็นต์ที่ลดลงของทั้ง 5 โปรแกรมเท่ากับ 45.46 %

เปอร์เซ็นต์ที่ลดลงของจำนวนโปรแกรมมิวแทนท์ โดยอาศัยการสร้างโปรแกรมมิวแทนท์แบบหลายข้อผิดพลาดนี้ขึ้นกับจำนวนของข้อผิดพลาดที่จะใส่ให้กับคำสั่งที่ไม่ใช่คำสั่งร่วมกัน หากข้อผิดพลาดที่ใส่ให้กับคำสั่งที่ไม่ใช่คำสั่งร่วมกันมีจำนวนมาก เปอร์เซ็นต์ที่ลดลงของโปรแกรมมิวแทนท์ในวิธีการสร้างโปรแกรมมิวแทนท์แบบหลายข้อผิดพลาดจะมีเปอร์เซ็นต์สูง

ในโปรแกรม MulSumMatrix ที่ประกอบด้วยผลลัพธ์ 2 ผลลัพธ์คือ ผลบวกของเมตริกซ์และผลคูณของเมตริกซ์ มีจำนวนของข้อผิดพลาดที่ใส่ให้กับคำสั่งที่ไม่ใช่คำสั่งร่วมกันของผลบวกเมตริกซ์น้อย ทำให้มีข้อผิดพลาดที่นำมาสร้างเป็นโปรแกรมมิวแทนท์แบบหลายข้อผิดพลาดได้น้อย ดังนั้นโปรแกรม MulSumMatrix มีการลดลงของจำนวนโปรแกรมมิวแทนท์น้อยที่สุด

ปัจจัยสำคัญอีกประการในการเพิ่มขึ้นของเปอร์เซ็นต์ที่ลดลงคือ จำนวนผลลัพธ์ของโปรแกรม หากจำนวนผลลัพธ์ของโปรแกรมมีจำนวนมาก ดังนั้นโอกาสที่โปรแกรมมิวแทนท์แบบหลายข้อผิดพลาดตัวหนึ่งๆ สามารถเป็นตัวแทนให้กับโปรแกรมมิวแทนท์หลายโปรแกรมมีมากขึ้น

3.4 การทดลองเพื่อวัดจำนวนครั้งในการวิเคราะห์มิวเทชันแบบหลายข้อผิดพลาด

ในการทดลองเพื่อแสดงถึงการลดลงของจำนวนครั้งในการทดสอบโปรแกรมมิวแทนท์แบบหลายข้อผิดพลาดเมื่อเทียบกับวิธีการเดิมนั้น ได้นำโปรแกรมมิวแทนท์มาทดสอบกับกรณีทดสอบต่างๆ ที่ผู้ทดสอบสร้างขึ้น จากนั้นเปรียบเทียบจำนวนครั้งที่ใช้ไปในการทดสอบโปรแกรมมิวแทนท์

แบบหลายข้อผิดพลาดและจำนวนครั้งที่ใช้ไปในการทดสอบโปรแกรมมิวแทนท์ในวิธีการเดิม พร้อมทั้งคำนวณเปอร์เซ็นต์ที่ลดลงของจำนวนครั้งในการทดสอบโปรแกรมมิวแทนท์แบบหลายข้อผิดพลาด ผลลัพธ์ที่ได้จากการทดลองถูกแสดงไว้ในตารางที่ 3.4

ตารางที่ 3.4 เปอร์เซ็นต์การลดลงของจำนวนครั้งในการทดสอบโปรแกรมมิวแทนท์

โปรแกรมที่นำมาทดสอบ	**จำนวนกรณีทดสอบทั้งหมด**	**จำนวนครั้งในวิธีการแบบเดิม**	**จำนวนครั้งในวิธีการแบบหลายข้อผิดพลาด**	**เปอร์เซ็นต์การลดลงของจำนวนครั้ง**
SinCos	2	141	81	42.55%
MulSumMatrix	1	67	43	35.82%
MulSumFraction	4	218	129	40.83%
Mean	3	89	57	35.96%
MaxMinAvg	4	89	50	43.82%
Average				39.80%

จากการทดองพบว่าเปอร์เซ็นต์การลดลงของจำนวนครั้งในการทดสอบมากที่สุดถึง 43.82% ในโปรแกรม MaxMinAvg ซึ่งค่าเฉลี่ยของทั้ง 5 โปรแกรมเท่ากับ 39.80%

ในโปรแกรม MulSumMatrix นั้นให้เปอร์เซ็นต์การลดลงของจำนวนครั้งในการทดสอบที่น้อยที่สุดเนื่องมาจาก เปอร์เซ็นต์การลดลงของจำนวนโปรแกรมมิวแทนท์ (ในตารางที่ 3.3) มีค่าน้อยที่สุดเมื่อเทียบกับทั้ง 5 โปรแกรม

ข้อสังเกตประการหนึ่งที่ส่งผลต่อเปอร์เซ็นต์การลดลงของจำนวนครั้งในการทดสอบคือ ประสิทธิผลของกรณีทดสอบและลำดับในการทดสอบของกรณีทดสอบ หากกรณีทดสอบที่นำมาใช้ทดสอบเป็นอันดับต้นๆ มีความสามารถในการตรวจจับผลกระทบของข้อผิดพลาดต่างๆ ที่ใส่ให้กับโปรแกรมมิวแทนท์แบบหลายข้อผิดพลาดใดๆ ได้จำนวนมาก จะทำให้เปอร์เซ็นต์การลดลงของจำนวนครั้งในการทดสอบมีค่ามาก ซึ่งเหตุการณ์เช่นนี้จะเกิดขึ้นได้ก็ต่อเมื่อข้อผิดพลาดที่ใส่ให้กับโปรแกรมมิวแทนท์ตัวหนึ่งๆ นั้นมีเงื่อนไขในการตรวจจับผลกระทบของข้อผิดพลาดที่ไม่ขัดแย้งกัน

บทที่ 4
การแบ่งกลุ่มโปรแกรมมิวแทนท์

4.1 การแบ่งกลุ่มโปรแกรมมิวแทนท์

ปัญหาสำคัญสำหรับการวิเคราะห์มิวเทชันคือ จำนวนครั้งที่กรณีทดสอบตัวหนึ่งใช้ในการทดสอบกับแต่ละโปรแกรมมิวแทนท์ โดยส่วนใหญ่แล้วกรณีทดสอบจำเป็นต้องทดสอบกับโปรแกรมมิวแทนท์ที่กรณีทดสอบนี้ไม่สามารถกำจัดได้ ดังนั้นหากสามารถแบ่งกลุ่มของโปรแกรมมิวแทนท์ออกเป็นกลุ่มย่อยๆ โดยแต่ละกลุ่มนั้นประกอบด้วยโปรแกรมมิวแทนท์ที่มีโอกาสถูกกำจัดได้ด้วยกรณีทดสอบที่มีข้อบังคับที่ใกล้เคียงกันจะทำให้จำนวนครั้งที่แต่ละกรณีทดสอบใช้ในการทดสอบโปรแกรมมิวแทนท์ลดลง

4.1.1 หลักการที่นำมาใช้ในการแบ่งกลุ่ม

ในงานวิจัยชิ้นนี้ได้เสนอหลักการสำหรับการแบ่งกลุ่มโปรแกรมมิวแทนท์ออกเป็นกลุ่มย่อยโดยอาศัยข้อบังคับของการไปถึงมาเป็นตัวจำแนกกลุ่มของโปรแกรมมิวแทนท์ สาเหตุที่เลือกข้อบังคับในการไปถึงเพียงข้อเดียวมาใช้ในการแบ่งกลุ่ม เนื่องมาจากหากนำข้อบังคับอื่นมาใช้ด้วยจะทำให้เสียเวลาเป็นอย่างมากในการวิเคราะห์การขัดแย้งกันของข้อบังคับที่นำมาใช้ โดยตัวอย่างของการวิเคราะห์ข้อบังคับการไปถึงของแต่ละคำสั่งในโปรแกรมถูกแสดงไว้ในตัวอย่างที่ 4.1

ตัวอย่างที่ 4.1 *กำหนดให้ส่วนของโปรแกรมมีดังนี้*

```
000   Procedure Test (a, b: Integer);
001   var i : Integer;
002   begin
003       for i:=1 to 5 do
004       begin
005           if a >= b then
006           begin
007               a := a + 1;
008           end
009           else
010           begin
011               b := b + 1;
012           end;
013       end;
014   end;
```

จากส่วนของโปรแกรมในตัวอย่างนี้สามารถวิเคราะห์ข้อบังคับการไปถึงได้ดังในตารางที่ 4.1

ตารางที่ 4.1 ข้อบังคับการไปถึง

คำสั่งที่	ข้อบังคับการไปถึง
003	True
005	(i <= 5)
007	(i <= 5) and (a >= b)
009	(i <= 5)
011	(i <= 5) and (a < b)

จากตารางที่ 4.1 พบว่ามีคำสั่งที่ 007 และ 011 ที่มีข้อบังคับการไปถึงที่ขัดแย้งกัน ดังนั้น กรณีทดสอบที่สามารถกำจัดข้อผิดพลาดที่ได้รับการใส่ข้อผิดพลาดที่คำสั่งที่ 007 นั้น จะไม่สามารถกำจัดโปรแกรมมิวแทนท์ที่เกิดจากการใส่ข้อผิดพลาดที่คำสั่งที่ 011 ได้ เนื่องมาจากข้อบังคับของการไปถึงของทั้งสองคำสั่งนั้นขัดแย้งกัน ดังนั้นสามารถแบ่งกลุ่มของโปรแกรมมิวแทนท์ออกเป็น 2 กลุ่ม ตามข้อบังคับ (i <= 5) and (a >= b) และ (i <= 5) and (a < b) ในส่วนของโปรแกรมมิวแทนท์ที่เกิดจากคำสั่งอื่นๆ นั้นจะถูกกระจายไปในแต่ละกลุ่มตามวิธีการด้านล่าง

4.1.2 อัลกอริธึมในการแบ่งกลุ่มโปรแกรมมิวแทนท์

กำหนดให้ชุดของโปรแกรมมิวแทนท์ M = Ø อัลกอริธึมในการแบ่งกลุ่มโปรแกรมมิวแทนท์เป็นดังต่อไปนี้

ขั้นตอนที่ 1 พิจารณาแต่ละคำสั่งในโปรแกรมที่นำมาทดสอบ เพื่อสร้างกราฟแสดงการไหลของการควบคุม (Control Flow Graph) พร้อมทั้งสร้างข้อผิดพลาดที่จะใส่ให้กับแต่ละคำสั่งจนครบทุกคำสั่ง

ขั้นตอนที่ 2 พิจารณากลุ่มของโปรแกรมมิวแทนท์ทุกกลุ่มที่มีอยู่ในขณะนั้นควบคู่กับเส้นทางในการทำงานของโปรแกรมที่นำมาทดสอบตามกราฟแสดงการไหลของการควบคุม

ขั้นตอนที่ 2.1 หากเส้นทางในการทำงานเป็นลักษณะเส้นตรง ให้สร้างและรวมโปรแกรมมิวแทนท์ที่ได้จากคำสั่งถัดมา (M') เข้าไว้เป็นกลุ่มเดียวกัน (M = M U M')

ขั้นตอนที่ 2.2 หากเส้นทางในการทำงานแบ่งออกเป็นทางแยกสองทางแยก ให้สร้างโปรแกรมมิวแทนท์ในแต่ละทางแยก

ขั้นตอนที่ 2.3 หากเส้นทางในการทำงานเกิดจากการรวมกันของสองทางแยก

1 กำหนดให้ n_1 และ n_2 เป็นจำนวนโปรแกรมมิวแทนท์ในทางแยกที่ 1 และทางแยกที่ 2

2 กำหนดให้ n เป็นจำนวนกลุ่มของโปรแกรมมิวแทนท์ก่อนเข้าทางแยก โดยกลุ่มที่ i มีจำนวนมิวแทนท์เท่ากับ M_i

3 ทางแยกที่ 1 แบ่งโปรแกรมมิวแทนท์ n_1 จำนวนให้เป็น n กลุ่มที่เท่ากัน คือ g_{11}, g_{12}, …,g_{1n}

4 ทางแยกที่ 2 แบ่งโปรแกรมมิวแทนท์ n_2 จำนวนให้เป็น n กลุ่มที่เท่ากัน คือ g_{21}, g_{22}, …,g_{2n}

5 นำโปรแกรมมิวแทนท์ก่อนเข้าทางแยกในกลุ่ม i มาแบ่งออกเป็นสองกลุ่ม กลุ่มที่หนึ่งจำนวน $\frac{n_2 \times M_i}{n_1 + n_2}$ โปรแกรม ไปรวมที่กลุ่ม g_{i1} และที่เหลือกลุ่มที่สองไปรวมในกลุ่ม g_{i2} ในกรณีที่ข้อบังคับของการไปถึงไม่ขัดแย้งกัน มิเช่นนั้น ให้รวมโปรแกรมมิวแทนท์ทั้ง M_i โปรแกรมเข้ากับกลุ่มใดกลุ่มหนึ่งที่ข้อบังคับของการไปถึงไม่ขัดแย้งกัน

หมายเหตุ *1. ทางแยกในโปรแกรมหมายถึงคำสั่งที่ทำให้เกิดการทำงานในโปรแกรมได้สองแบบที่แตกต่างกัน เช่นคำสั่ง if เป็นต้น*

2. หากในโปรแกรมมีการวนรอบหลายๆ การวนรอบติดกัน หรือ การที่มีคำสั่ง If โดยไม่มีทางเลือกอื่น (Else, Else if) ติดต่อกัน จะไม่มีการแบ่งกลุ่มโปรแกรมมิวแทนท์เปรียบเสมือนเป็นการทำงานแบบเป็นเส้นตรง

ขั้นตอนที่ 3 พิจารณาแต่ละคำสั่งจนสิ้นสุดโปรแกรม โดยอาศัยหลักการแบ่งกลุ่มในขั้นตอนที่ 2 ซึ่งโปรแกรมมิวแทนท์ที่ได้รับการแบ่งกลุ่มไปเรียบร้อยแล้วจะไม่มีการนำกลับมารวมกันใหม่อีกครั้ง ถึงแม้ว่าเส้นทางในการทำงานจะเป็นลักษณะเส้นตรง

ผลลัพธ์ที่ต้องการ โปรแกรมมิวแทนท์จะถูกแบ่งออกเป็นกลุ่มๆ ซึ่งในแต่ละกลุ่มจะมีโปรแกรมมิวแทนท์ที่มีข้อบังคับการไปถึงที่ไม่ขัดแย้งกัน

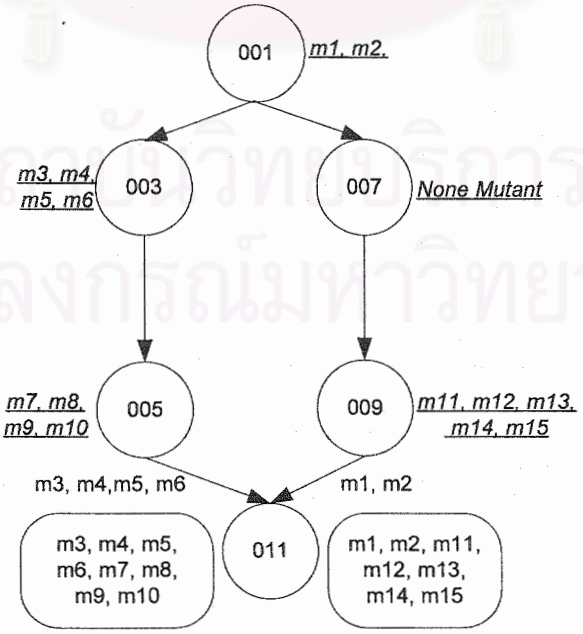
การแบ่งกลุ่มโปรแกรมมิวแทนท์ออกเป็นหลายกลุ่มมีประโยชน์ คือ กรณีทดสอบตัวหนึ่งๆ ไม่จำเป็นต้องทดสอบกับโปรแกรมมิวแทนท์ที่เกิดจากคำสั่งที่กรณีทดสอบนี้ไม่สามารถไปถึงได้ เวลาที่เสียไปในการวิเคราะห์ข้อบังคับการไปถึงเป็นส่วนหนึ่งของการสร้างกรณีทดสอบ ดังนั้นเวลาที่เสียไปจริงๆ ในการทำงานคือ ขั้นตอนในการแบ่งกลุ่มของโปรแกรมมิวแทนท์เท่านั้น โดยตัวอย่างของการแบ่งกลุ่มโปรแกรมมิวแทนท์ถูกแสดงไว้ในตัวอย่างที่ 4.2

ตัวอย่างที่ 4.2 *จากส่วนของโปรแกรมในตัวอย่างที่ 4.1 หากมีการใส่ข้อผิดพลาดให้กับแต่ละคำสั่ง ตัวอย่างของโปรแกรมมิวแทนท์ที่จะได้รับจะเป็นดังในตารางที่ 4.2*

ตารางที่ 4.2 ตัวอย่างมิวแทนท์ที่เกิดจากการใส่ข้อผิดพลาดให้กับส่วนของโปรแกรม

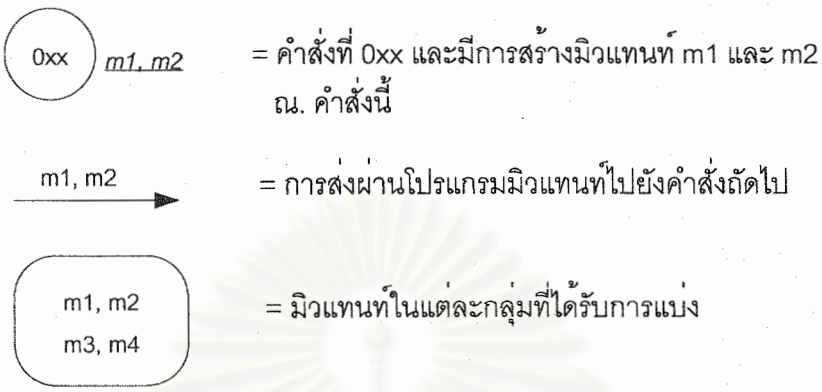
ลำดับที่	บรรทัด	ข้อผิดพลาด	ข้อบังคับการไปถึง
m1	003	i := -1 to 5 do	True
m2	003	i := 1 to -5 do	True
m3	005	a > b	(I <= 5)
m4	005	a < b	(I <= 5)
m5	005	a = b	(I <= 5)
m6	005	a <> b	(I <= 5)
m7	007	a := a – 1;	(i <= 5) and (a >= b)
m8	007	a := a * 1;	(i <= 5) and (a >= b)
m9	007	a := a / 1;	(i <= 5) and (a >= b)
m10	007	a := a MOD 1;	(i <= 5) and (a >= b)
m11	011	b := b – 1;	(i <= 5) and (a < b)
m12	011	b := b * 1;	(i <= 5) and (a < b)
m13	011	b := b / 1;	(i <= 5) and (a < b)
m14	011	b := b MOD 1;	(i <= 5) and (a < b)
m15	011	b := b DIV 1;	(i <= 5) and (a < b)

ส่วนของโปรแกรมในตัวอย่างที่ 4.1 นั้น สามารถเขียนแผนภาพแสดงการไหลและการแบ่งกลุ่มของโปรแกรมมิวแทนท์ได้ดังในรูปที่ 4.1



รูปที่ 4.1 แผนภาพแสดงการไหลและการแบ่งกลุ่มของโปรแกรมมิวแทนท์

ในรูปที่ 4.1 นั้น แสดงการแบ่งกลุ่มของโปรแกรมมิวแทนท์ตามอัลกอริธึมในการแบ่งกลุ่ม ซึ่งในแผนภาพประกอบด้วยสัญลักษณ์ต่างๆ ดังในรูปที่ 4.2



รูปที่ 4.2 สัญลักษณ์ที่ใช้ในการแบ่งโปรแกรมมิวแทนท์

โปรแกรมมิวแทนท์ที่ได้รับการขีดเส้นใต้ที่อยู่ด้านของข้างแต่ละคำสั่งนั้นแสดงถึงโปรแกรมมิวแทนท์ที่เกิดจากคำสั่งนั้นๆ ส่วนโปรแกรมมิวแทนท์ที่อยู่ด้านข้างของเส้นทางการทำงานแสดงถึงจำนวนโปรแกรมมิวแทนท์ที่ไหนไปในเส้นทางนั้นๆ

เมื่อพิจารณาข้อบังคับการไปถึงของแต่ละโปรแกรมมิวแทนท์และแบ่งกลุ่มโปรแกรมผันตามวิธีการแบ่งกลุ่มข้างต้น ทำให้กลุ่มของโปรแกรมมิวแทนท์ที่มีอยู่เดิมนั้นสามารถแบ่งออกได้เป็น 2 กลุ่มด้วยกัน

กลุ่มที่ 1 จะประกอบด้วยโปรแกรมมิวแทนท์ m3, m4, m5, m6, m7, m8, m9, m10

กลุ่มที่ 2 จะประกอบด้วยโปรแกรมมิวแทนท์ m1, m2, m11, m12, m13, m14, m15

เมื่อเปรียบเทียบจำนวนครั้งในการทดสอบโปรแกรมด้วยชุดทดสอบหนึ่งๆ เมื่อเราสร้างกรณีทดสอบ t_1 = (a=5, b= 5) สำหรับโปรแกรมมิวแทนท์ในกลุ่มที่ 1 และกรณีทดสอบ t_2 = (a=3, b=5) สำหรับโปรแกรมมิวแทนท์ในกลุ่มที่ 2 และผลลัพธ์ที่คาดหวังของแต่ละกรณีทดสอบเป็นดังต่อไปนี้คือ (a = 10, b = 5) สำหรับกรณีทดสอบ t_1 และ (a = 3, b=10) สำหรับกรณีทดสอบ t_2

ในวิธีการแบบเดิม ในขั้นแรกต้องนำกรณีทดสอบ t_1 ไปทดสอบกับทุกๆ โปรแกรมมิวแทนท์ ซึ่งต้องใช้จำนวนครั้งในการทดสอบทั้งหมด 15 ครั้งในการทดสอบ และสามารถกำจัดโปรแกรมมิวแทนท์ออกไปได้ 10 โปรแกรมมิวแทนท์ จากนั้นนำ t_2 มาทดสอบกับโปรแกรมมิวแทนท์ที่เหลือซึ่งใช้จำนวนครั้งในการทดสอบอีก 5 ครั้ง

ในวิธีการแบ่งกลุ่มโปรแกรมมิวแทนท์นั้น สามารถแบ่งกลุ่มโปรแกรมมิวแทนท์ออกเป็น 2 กลุ่ม โดยจะนำกรณีทดสอบ t_1 ไปทดสอบกับโปรแกรมมิวแทนท์ในกลุ่มที่ 1 เท่านั้น เนื่องจากกรณีทดสอบ t_1 นั้นไม่รองรับข้อบังคับการไปถึงของมิวแทนท์ส่วนใหญ่ในกลุ่มที่ 2 และ t_2 นำไปทดสอบกับโปรแกรมมิวแทนท์ในกลุ่มที่ 2 เท่านั้น ดังนั้นเมื่อนำ t_1 ไปทดสอบจะใช้จำนวนครั้งในการทดสอบไป 8 ครั้ง และนำกรณีทดสอบ t_2 ไปทดสอบจะใช้จำนวนครั้งในการทดสอบไป 7 ครั้ง โดยตารางที่ 4.3 แสดงการเปรียบเทียบจำนวนครั้งในการทดสอบแต่ละโปรแกรมมิวแทนท์ของแต่ละวิธี

ตารางที่ 4.3 ตารางแสดงการเปรียบเทียบจำนวนครั้งในการทดสอบแต่ละโปรแกรมมิวแทนท์

กรณีทดสอบ	จำนวนครั้งในวิธีการแบบเดิม	จำนวนครั้งในวิธีการแบ่งกลุ่ม
t_1	15	8
t_2	5	7
Total	20	15

สังเกตได้ว่าจำนวนครั้งในการทดสอบโปรแกรมมิวแทนท์ด้วยวิธีการแบ่งกลุ่มนั้นสามารถลดลงได้เมื่อเทียบกับวิธีการแบบเดิม

4.2 ทฤษฎีและการพิสูจน์การแบ่งกลุ่มโปรแกรมมิวแทนท์

ในการแบ่งกลุ่มของโปรแกรมมิวแทนท์เพื่อลดจำนวนครั้งในการนำกรณีทดสอบไปทดสอบกับโปรแกรมมิวแทนท์ที่กรณีทดสอบนั้นไม่สามารถกำจัดได้ งานวิจัยชิ้นนี้ได้เสนอแนวทางในการพิสูจน์เรื่องของจำนวนครั้งที่ลดลงไปไว้ดังต่อไปนี้

นิยามที่ 9

กำหนดให้

- β เป็นจำนวนของโปรแกรมมิวแทนท์ทั้งหมด
- γ เป็นจำนวนกรณีทดสอบในชุดของกรณีทดสอบ W
- α_j เป็นจำนวนโปรแกรมมิวแทนท์ที่ถูกกำจัดด้วยกรณีทดสอบ t_j ที่เป็นสมาชิกของ W

จำนวนครั้งที่ใช้ในการทดสอบโปรแกรมมิวแทนท์ต่างๆ ในวิธีการแบบเดิม (T) ด้วยทุกๆ กรณีทดสอบ t_j ที่เป็นสมาชิกของ W เป็นดังต่อไปนี้

$$T = \beta + (\beta - \alpha_1) + (\beta - \alpha_1 - \alpha_2) + ... + (\beta - \alpha_1 - \alpha_2 - ... - \alpha_{\gamma-1})$$

ดังนั้น

$$T = \beta\gamma - \sum_{i=1}^{\gamma-1}(\gamma - i)\alpha_i$$

ถ้าเราประมาณความน่าจะเป็นของโปรแกรมมิวแทนท์ที่ถูกกำจัดด้วยกรณีทดสอบที่สร้างขึ้นเป็น α ดังนั้นจำนวนครั้งที่ใช้ในการทดสอบเป็น

$$\begin{aligned} T &= \beta + \beta(1-\alpha) + \beta(1-\alpha)^2 + .. + \beta(1-\alpha)^\gamma \\ &= \beta\sum_{i=0}^{\gamma}(1-\alpha)^i \end{aligned}$$

การลดจำนวนครั้งที่ใช้ในการทดสอบโปรแกรมมิวแทนท์นั้นสามารถทำได้โดยการแบ่งกลุ่มของโปรแกรมมิวแทนท์ออกเป็นกลุ่มย่อยๆ พร้อมทั้งสร้างกรณีทดสอบสำหรับแต่ละกลุ่ม ดังที่ได้กล่าวไปแล้วในหัวข้อที่ 3.2

หลักการสำคัญที่นำมาใช้ในการแบ่งกลุ่มโปรแกรมมิวแทนท์คือ ทุกๆ โปรแกรมมิวแทนท์ที่อยู่ในกลุ่มเดียวกันควรมีข้อบังคับการไปถึงที่ไม่ขัดแย้งกัน เพื่อให้ความน่าจะเป็นในการกำจัดโปรแกรมมิวแทนท์ที่อยู่ในกลุ่มเดียวกันมีสูงขึ้น

4.2.1 การลดลงของจำนวนครั้งในการทดสอบด้วยวิธีการแบ่งกลุ่ม

ทฤษฎีบทที่ 3

จำนวนครั้งในการทดสอบโปรแกรมมิวแทนท์ด้วยวิธีการแบ่งกลุ่มนั้นจะลดลงเมื่อเทียบกับวิธีการแบบเดิม ในกรณีที่ความน่าจะเป็นที่มิวแทนท์ในแต่ละกลุ่มจะถูกกำจัด (α') มากกว่าหรือเท่ากับ α

พิสูจน์

กำหนดให้

- M เป็นชุดของโปรแกรมมิวแทนท์ที่เกิดจากการใส่ข้อผิดพลาดให้กับโปรแกรมทดสอบ P
- ชุดโปรแกรมมิวแทนท์ M สามารถแบ่งได้ออกเป็น g กลุ่มดังนี้ $M_1, M_2, M_3,\ldots,M_g$
- ความน่าจะเป็นของโปรแกรมมิวแทนท์ในกลุ่มต่างๆ ที่ถูกกำจัดเป็น α'
- μ เป็นจำนวนที่มากที่สุดของกรณีทดสอบในแต่ละกลุ่ม
- β_j เป็นจำนวนของโปรแกรมมิวแทนท์ในกลุ่ม M_j ใดๆ สำหรับค่า $1 \le j \le g$

จำนวนครั้งในการทดสอบโปรแกรมมิวแทนท์ในกลุ่ม M_j ใดๆ มีค่าเท่ากับ

$$T'(M_j)=\beta_j+\beta_j(1-\alpha')+\beta_j(1-\alpha')^2+..+\beta_j(1-\alpha')^{\mu_j}$$

$$T'(M_j)=\beta_j\sum_{i=0}^{\mu_j}(1-\alpha')^i$$

โดยที่ μ_j เป็นจำนวนของกรณีทดสอบสำหรับกลุ่ม M_j ที่ $1 \le j \le g$ นั่นคือ $\mu = \max(\mu_j)$ ดังนั้นจำนวนครั้งที่ใช้ในการทดสอบทั้งหมดเป็นผลรวมของจำนวนครั้งในการทดสอบแต่ละกลุ่ม

$$T'=\beta_1\sum_{i=0}^{\mu_1}(1-\alpha')^i+\beta_2\sum_{i=0}^{\mu_2}(1-\alpha')^i+...+\beta_g\sum_{i=0}^{\mu_g}(1-\alpha')^i$$

เมื่อเราประมาณค่ากรณีแย่ที่สุดคือ $\mu = \mu_j$ ใดๆ ดังนั้นจำนวนครั้งที่ใช้ในการทำงานทั้งหมดเป็น

$$T' \le (\sum_{j=1}^{g}\beta_j)(\sum_{i=0}^{\mu}(1-\alpha')^i)=\beta(\sum_{i=0}^{\mu}(1-\alpha')^i)$$

เนื่องจากค่า $\mu \le \gamma$ และ $\alpha' \ge \alpha$ จึงทำให้จำนวนครั้งในการทดสอบด้วยวิธีการวิเคราะห์มิวเทชันแบบแบ่งกลุ่มลดลง

4.2.2 ความน่าจะเป็นในการกำจัดโปรแกรมมิวแทนท์ในแต่ละกลุ่ม

ทฤษฎีบทที่ 4

ด้วยวิธีการแบ่งกลุ่มโปรแกรมมิวแทนท์นั้น ความน่าจะเป็นของโปรแกรมมิวแทนท์ที่จะถูกกำจัดในแต่ละกลุ่มจะไม่ลดลง

พิสูจน์

กำหนดให้

- β เป็นจำนวนของโปรแกรมมิวแทนท์ทั้งหมด
- กรณีทดสอบ t ใดๆ มี k โปรแกรมมิวแทนท์ที่ถูกกำจัดด้วยวิธีการแบบเดิม ความน่าจะเป็นของโปรแกรมมิวแทนท์ที่ถูกกำจัดเป็น

$$\alpha=\frac{k}{\beta}$$

- $M_1, M_2, M_3,\ldots, M_g$ เป็นกลุ่มที่เกิดจากการแบ่งโปรแกรมมิวแทนท์ M
- $k_1,k_2,k_3,\ldots, k_g$ เป็นจำนวนของโปรแกรมมิวแทนท์ในแต่ละกลุ่มที่ถูกกำจัดด้วยกรณีทดสอบ t

เมื่อพิจารณาที่กลุ่ม M_j ใดๆ ที่ $1 \le j \le g$ กรณีทดสอบ t_j ที่สร้างมาสำหรับโปรแกรมมิวแทนท์ในกลุ่ม M_j สามารถกำจัดโปรแกรมมิวแทนท์ในกลุ่ม M_j อย่างน้อย k_j (เนื่องมาจาก t เดิมสามารถกำจัดได้ k_j มิวแทนท์อยู่แล้ว) ดังนั้นความน่าจะเป็นของโปรแกรมมิวแทนท์ที่ถูกกำจัดในกลุ่ม M_j ด้วยชุดของกรณีทดสอบ t_j มีค่าเท่ากับ

$$\alpha_j \ge \frac{k_j}{\beta_j}$$

โดยที่ β_j เป็นจำนวนของโปรแกรมมิวแทนท์ในกลุ่ม M_j ความน่าจะเป็นของโปรแกรมมิวแทนท์ที่ถูกกำจัด (α') เป็น

$$\alpha' = \frac{\alpha_1\beta_1 + \alpha_2\beta_2 + \alpha_3\beta_3 + .. + \alpha_g\beta_g}{\beta_1 + \beta_2 + \beta_3 + .. + \beta_g} = \frac{\sum_{j=1}^{g} \alpha_j\beta_j}{\sum_{j=1}^{g} \beta_j}$$

$$\alpha' = \frac{\sum_{j=1}^{g} \alpha_j\beta_j}{\sum_{j=1}^{g} \beta_j} \ge \frac{\sum_{j=1}^{g} k_j}{\sum_{j=1}^{g} \beta_j} = \alpha$$

สาเหตุที่ $\alpha' \ge \alpha$ เนื่องมาจากค่า $\alpha_j \beta_j \ge k_j$

ตัวอย่างที่ 4.3 *กำหนดให้จำนวนโปรแกรมมิวแทนท์ทั้งหมดเป็น 2_m ความน่าจะเป็นของโปรแกรมมิวแทนท์ที่ถูกกำจัดด้วยกรณีทดสอบเป็น 0.5 และสามารถแบ่งกลุ่มมิวแทนท์เป็น 2_g กลุ่ม*

ดังนั้น จำนวนโปรแกรมมิวแทนท์ในแต่ละกลุ่มเป็น 2^{m-g}

จำนวนกรณีทดสอบในแต่ละกลุ่ม μ มีค่าเท่ากับ m-g+1

จำนวนกรณีทดสอบทั้งหมด γ มีค่าเท่ากับ m+1

จำนวนครั้งที่ใช้ในการทดสอบโปรแกรมมิวแทนท์ในวิธีการแบบเดิมเป็น

$$T = \beta\sum_{i=0}^{\gamma}(1-\alpha)^i$$

$$= \beta\sum_{i=0}^{m+1}(1-\alpha)^i = 2^m(1+\frac{1}{2}+\frac{1}{2^2}+...+\frac{1}{2^{m+1}})$$

จำนวนครั้งที่ใช้ในการทดสอบโปรแกรมมิวแทนท์ในวิธีการแบบแบ่งกลุ่มเป็น

$$T' = \beta\sum_{i=0}^{\mu}(1-\alpha)^i$$
$$= \beta\sum_{i=0}^{m-g+1}(1-\alpha)^i = 2^m(1+\frac{1}{2}+\frac{1}{2^2}+...+\frac{1}{2^{m-g+1}})$$

จำนวนครั้งที่ใช้ในการทดสอบโปรแกรมมิวแทนท์ในวิธีการแบบแบ่งกลุ่มลดลงไปจากวิธีการเดิมเป็น

$$T - T' = 2^m(\frac{1}{2^{m-g+2}}+\frac{1}{2^{m-g+3}}+..+\frac{1}{2^{m+1}})$$

4.3 การทดลองเพื่อวัดจำนวนครั้งในการทดสอบโปรแกรมมิวแทนท์ด้วยวิธีการแบ่งกลุ่ม

ในงานวิจัยชิ้นนี้ได้ทำการทดลองเพื่อเปรียบเทียบจำนวนครั้งที่ใช้ในการทดสอบโปรแกรมมิวแทนท์ในวิธีการวิเคราะห์มิวเทชันแบบเดิมและวิธีการแบ่งกลุ่มโปรแกรมมิวแทนท์ โดยเลือกโปรแกรมที่นำมาทดสอบจำนวน 5 โปรแกรม ผลลัพธ์ที่ได้รับจากการทดลองเป็นดังในตารางที่ 4.4 โดยรายละเอียดของแต่ละโปรแกรมอยู่ในภาคผนวก ก.

ตารางที่ 4.4 ตารางแสดงจำนวนครั้งในการทดสอบโปรแกรมมิวแทนท์ด้วยวิธีการแบ่งกลุ่ม

โปรแกรม	กรณีทดสอบ	จำนวนครั้งในวิธีการแบบเดิม	จำนวนครั้งในวิธีการแบ่งกลุ่ม	เปอร์เซ็นต์การลดลง
IncDec	5	128	68	46.88
Triangle	22	1181	627	46.91
ProdSum	4	100	82	18.00
IncWithCond	4	68	58	14.71
SinCos	2	145	145	0.00
Average				25.30

โปรแกรม Triangle ในรูปที่ 4.3 ใช้ในการวิเคราะห์ว่าด้านทั้งสามด้านของสามเหลี่ยมทำให้เกิดสามเหลี่ยมประเภทใด พร้อมทั้งคำนวณหาความยาวของเส้นรอบรูปของสามเหลี่ยม

```
Program TriAngle;
var sumSide,side1,side2,side3 : integer;
begin
    write('Please input side 1 ranging [1,200] of triangle =');
    readln(side1);
    write('Please input side 2 ranging [1,200] of triangle =');
    readln(side2);
    write('Please input side 3 ranging [1,200] of triangle =');
    readln(side3);
    if ((side1>=1)and(side1<=200))and((side2>=1)and(side2<=200))
    and((side3>=1)and(side3<=200)) then
    begin
        if (side1 < (side2 + side3)) and (side2 < (side1 + side3)) and
        (side3 < (side1 + side2)) then
        begin
            if (side1 = side2) and (side2 = side3) then
            begin
                sumSide := side1*3;
                writeln('The summation of Equilateral triangle''s sides is',sumSide);
            end
            else if ((side1=side2)and(side1<>side3))or((side1=side3)and(side1<>side2))
                or ((side2=side3)and(side1<>side3))then
            begin
                sumSide := side1 + side2 + side3;
                writeln('The summation of Isoscales triangle''s sides  is',sumSide);
            end
            else
            begin
                sumSide := side1 + side2 + side3;
                 writeln('The summation of Scales triangle''s sides  is',sumSide);
```

รูปที่ 4.3 โปรแกรม Triangle

```
                end;
            end
            else
            begin
                sumSide := side1 + side2 + side3;
                writeln('This is not Triangle, but summation of sides is',sumSide);
            end;
        end
        else
        begin
            sumSide := side1+side2+side3;
             writeln('Over range, but summation = ',sumSide);
        end;
    end.
```

รูปที่ 4.3 โปรแกรม Triangle (ต่อ)

ในการสร้างโปรแกรมมิวแทนท์โดยการใส่ข้อผิดพลาดให้กับคำสั่งต่างๆ ของโปรแกรม Triangle ให้จำนวนโปรแกรมมิวแทนท์ทั้งหมด 310 โปรแกรมมิวแทนท์ ซึ่งเป็นจำนวนโปรแกรมมิวแทนท์สมมูลทั้งสิ้น 56 โปรแกรมมิวแทนท์ เมื่อแบ่งกลุ่มโปรแกรมมิวแทนท์ออกเป็นกลุ่มย่อยๆ โดยอาศัยอัลกอริธึมในการแบ่งกลุ่ม ทำให้สามารถแบ่งกลุ่มของโปรแกรมมิวแทนท์ออกได้เป็น 5 กลุ่ม การแบ่งโปรแกรมมิวแทนท์ไปให้ในแต่ละกลุ่มจะแบ่งตามทางแยกในโปรแกรม

การทดสอบโปรแกรมมิวแทนท์นั้น ในขั้นต้นได้สร้างกรณีทดสอบที่สอดคล้องกับข้อบังคับการไปถึงของแต่ละกลุ่ม แต่อาจมีบางโปรแกรมมิวแทนท์ที่มีข้อบังคับการไปถึงที่ไม่ขัดแย้งแต่มีข้อบังคับที่จำเป็นขัดแย้ง ทำให้ต้องอาศัยกรณีทดสอบของกลุ่มอื่นมาใช้

ปัจจัยสำคัญที่ส่งผลต่อการเพิ่มขึ้นของเปอร์เซ็นต์การลดลงของจำนวนครั้งในการทดสอบโปรแกรมมิวแทนท์ด้วยวิธีการแบ่งกลุ่มคือ จำนวนกลุ่มของโปรแกรมมิวแทนท์ที่แบ่งได้ โดยเปอร์เซ็นต์การลดลงของจำนวนครั้งในการทดสอบโปรแกรมมิวแทนท์ของโปรแกรม Triangle มีค่ามากถึง 46.91 % เนื่องมาจากสามารถแบ่งกลุ่มโปรแกรมมิวแทนท์ได้ 5 กลุ่ม

ในโปรแกรม SinCos นั้น จำนวนครั้งในการทดสอบโปรแกรมมิวแทนท์ในวิธีการแบบเดิมมีจำนวนเท่ากับวิธีการแบ่งกลุ่มเนื่องมาจากว่าโปรแกรม SinCos เป็นโปรแกรมที่ไม่มีทางเลือก ดังนั้นจึงไม่สามารถแบ่งกลุ่มโปรแกรมมิวแทนท์ได้

ถึงแม้ว่าโปรแกรมที่นำมาทดสอบจะสามารถแบ่งกลุ่มของโปรแกรมมิวแทนท์ได้หลายกลุ่ม หากจำนวนโปรแกรมมิวแทนท์ที่มีเป็นสมาชิกของแต่ละกลุ่มก่อนที่จะได้รับการแบ่งโปรแกรมมิวแทนท์มาจากคำสั่งอื่นมีจำนวนน้อย ส่งผลให้เปอร์เซ็นต์การลดลงของจำนวนครั้งการทดสอบโปรแกรมมิวแทนท์ไม่ได้เพิ่มจากเดิมมากนัก

บทที่ 5
การวิเคราะห์มิวเทชันแบบหลายข้อผิดพลาดโดยการประยุกต์ใช้การแบ่งกลุ่มโปรแกรมมิวแทนท์

ในบทนี้จะกล่าวถึงการพิจารณานำเอาการแบ่งกลุ่มโปรแกรมมิวแทนท์ มาประยุกต์ใช้กับการสร้างโปรแกรมมิวแทนท์แบบหลายข้อผิดพลาด เพื่อเพิ่มประสิทธิภาพในการลดจำนวนครั้งในการทดสอบโปรแกรม ทั้งนี้เนื่องจากการแบ่งกลุ่มโปรแกรมมิวแทนท์นั้น ถึงแม้จะลดจำนวนครั้งของการทดสอบลงได้ แต่จำนวนโปรแกรมมิวแทนท์นั้นมิได้ลดลงไปด้วย จำนวนครั้งของการทดสอบที่น้อยที่สุดที่เป็นไปได้ยังคงเท่ากับจำนวนของโปรแกรมมิวแทนท์ทั้งหมดนั่นเอง สำหรับโปรแกรมมิวแทนท์แบบหลายข้อผิดพลาดนั้น นอกจากมีข้อจำกัดในกรณีที่โปรแกรมทดสอบมีจำนวนผลลัพธ์น้อยหรือในกรณีที่ผลลัพธ์ไม่เป็นอิสระจากกันมีผลต่อจำนวนการลดลงของโปรแกรมมิวแทนท์แล้ว ในการกำจัดโปรแกรมมิวแทนท์แบบหลายข้อผิดพลาดแต่ละโปรแกรม อาจต้องใช้จำนวนกรณีทดสอบมากกว่าหนึ่งกรณีทดสอบ ทำให้กรณีที่เลวร้ายที่สุดของการทดสอบต้องใช้จำนวนครั้งของการทดสอบเท่ากับจำนวนครั้งของการทดสอบในวิธีเดิม

5.1 แนวคิดของการวิเคราะห์มิวเทชันแบบหลายข้อผิดพลาดโดยการประยุกต์ใช้การแบ่งกลุ่มโปรแกรมมิวแทนท์

จากปัญหาที่พบในการวิเคราะห์มิวเทชันโดยหลักการสร้างโปรแกรมมิวแทนท์แบบหลายข้อผิดพลาด 2 ประการ คือ กรณีที่ข้อบังคับในการตรวจจับผลกระทบของข้อผิดพลาดต่างๆ ที่ใส่ให้กับโปรแกรมมิวแทนท์แบบหลายข้อผิดพลาดตัวเดียวกันมีความขัดแย้งกัน ทำให้จำเป็นต้องใช้กรณีทดสอบจำนวนมากในการกำจัดโปรแกรมมิวแทนท์แบบหลายข้อผิดพลาดนี้ ปัญหาข้อถัดมาคือ หากโปรแกรมที่นำมาทดสอบนั้นมีผลลัพธ์เพียงผลลัพธ์เดียวทำให้ไม่สามารถรวมโปรแกรมมิวแทนท์ต่างๆ เข้าเป็นโปรแกรมมิวแทนท์แบบหลายข้อผิดพลาดได้ ทำให้ไม่สามารถลดเวลาในการทำการวิเคราะห์มิวเทชันได้

ประเด็นที่น่าสนใจในการประยุกต์คือนำการแบ่งกลุ่มของโปรแกรมมิวแทนท์มาช่วยในการสร้างมิวแทนท์แบบหลายข้อผิดพลาดเพื่อลดจำนวนกรณีทดสอบที่ใช้ในการกำจัดมิวแทนท์แบบหลายข้อผิดพลาด

ในหัวข้อนี้ได้พิจารณาการนำเอาการแบ่งกลุ่มโปรแกรมมิวแทนท์มาช่วยในการแก้ปัญหาเหล่านี้ โดยหลังจากที่ได้ทำการแบ่งโปรแกรมมิวแทนท์ออกเป็นกลุ่มต่างๆ ตามวิธีในการแบ่งกลุ่ม

เรียบร้อยแล้ว ให้ทำการวิเคราะห์มิวเทชันโดยหลักการวิเคราะห์มิวเทชันแบบหลายข้อผิดพลาดภายในแต่ละกลุ่มที่แบ่งไว้

ข้อดีของการอาศัยการแบ่งกลุ่มโปรแกรมมิวแทนท์มาช่วยในการสร้างโปรแกรมมิวแทนท์แบบหลายข้อผิดพลาดคือ ทำให้กรณีที่โปรแกรมมิวแทนท์ที่ยุบรวมกันเป็นโปรแกรมมิวแทนท์แบบหลายข้อผิดพลาดมีข้อบังคับการไปถึงขัดแย้งกันนั้นหมดไป เนื่องจากว่าโปรแกรมมิวแทนท์ในแต่ละกลุ่มจะมีข้อบังคับของการไปถึงที่สอดคล้องกันภายในกลุ่มเดียวกัน การรวมโปรแกรมมิวแทนท์เหล่านี้ให้เป็นโปรแกรมมิวแทนท์แบบหลายข้อผิดพลาดจะเป็นการลดจำนวนโปรแกรมมิวแทนท์ลง และนำไปสู่การลดจำนวนครั้งของการทดสอบ เพราะโปรแกรมมิวแทนท์แบบหลายข้อผิดพลาดอาจสามารถถูกกำจัดได้ด้วยกรณีทดสอบเดียวนั่นเอง

สำหรับกรณีที่โปรแกรมที่นำมาทดสอบมีผลลัพธ์เพียงผลลัพธ์เดียว แล้วโดยวิธีการนี้จะสามารถลดเวลาในการทำการวิเคราะห์มิวเทชันได้ถ้าสามารแบ่งกลุ่มได้

5.2 ขั้นตอนในการมิวเทชันแบบหลายข้อผิดพลาดโดยประยุกต์ใช้การแบ่งกลุ่มโปรแกรมมิวแทนท์

ขั้นตอนในการประยุกต์การสร้างโปรแกรมมิวแทนท์แบบหลายข้อผิดพลาดโดยอาศัยการแบ่งกลุ่มโปรแกรมมิวแทนท์สามารถแบ่งออกได้เป็นขั้นตอนดังนี้

ขั้นตอนที่ 1 สร้างข้อผิดพลาดทั้งหมดจากโปรแกรมที่ต้องการทดสอบ

ขั้นตอนที่ 2 นำข้อผิดพลาดที่ได้จากขั้นตอนที่ 1 มาทำการแบ่งกลุ่ม ตามอัลกอริธึมของการแบ่งกลุ่มโปรแกรมมิวแทนท์ในบทที่ 4 โดยมิต้องสร้างโปรแกรมมิวแทน

ขั้นตอนที่ 3 ทำการตัดส่วนโปรแกรมที่ต้องการทดสอบออกเป็นโปรแกรมตัดส่วนตามแต่ละผลลัพธ์ของโปรแกรมที่ต้องการทดสอบ

ขั้นตอนที่ 4 พิจารณาแต่ละกลุ่มของโปรแกรมมิวแทนท์

1. แบ่งข้อผิดพลาดออกเป็นกลุ่มย่อย โดยพิจารณา ว่าข้อผิดพลาดถูกใส่ในตำแหน่งที่ปรากฏในโปรแกรมตัดส่วนเดียวกันหรือไม่ ถ้าปรากฏในโปรแกรมตัดส่วนเดียวกันให้จัดอยู่ในกลุ่มย่อยเดียวกัน ในกรณีที่ข้อผิดพลาดนั้นปรากฏในโปรแกรมตัดส่วนมากกว่าหนึ่งโปรแกรม ให้แยกข้อผิดพลาดนั้นออกจากการพิจารณา (เนื่องจากเป็นข้อผิดพลาดที่ไม่สามารถนำมาสร้างเป็นมิวแทนท์แบบหลายข้อผิดพลาดได้)

2. สร้างโปรแกรมมิวแทนท์แบบหลายข้อผิดพลาด จากการรวมข้อผิดพลาดจากกลุ่มย่อยที่ต่างกัน โดยข้อผิดพลาดที่รวมกันได้ต้องมาจากกลุ่มย่อยที่ต่างกันเท่านั้น

ขั้นตอนที่ 5 สร้างโปรแกรมมิวแทนท์จากข้อผิดพลาดที่แยกออกมาจากข้อ 1 ในขั้นตอนที่ 4 โดยหนึ่งข้อผิดพลาดสร้างได้หนึ่งมิวแทนท์

ขั้นตอนที่ 6 รวมโปรแกรมมิวแทนท์จากขั้นตอนที่ 4 และขั้นตอนที่ 5

5.3 กรณีตัวอย่างของการปรับปรุงมิวแทนท์แบบหลายข้อผิดพลาดโดยการแบ่งกลุ่ม

ในงานวิจัยชิ้นนี้ได้เสนอกรณีตัวอย่างในการปรับปรุงการวิเคราะห์มิวเทชันแบบหลายข้อผิดพลาดโดยการนำเอาการแบ่งกลุ่มโปรแกรมมิวแทนท์นั้นมาประยุกต์ใช้ โดยโปรแกรมที่นำมาทดสอบคือโปรแกรม IncWithCond ซึ่งได้แสดงไว้ดังรูปที่ 5.1

```
Program IncWithCond;
var v1,v2,a : integer;
begin
        readln(a);
        if a <= 0 then
        begin
                v1 := a + 1;
        end;
        if a > 10 then
        begin
                v2 := a + 1;
        end
        else
        begin
                v2 := a + 5;
        end;
        writeln(v1);
        writeln(v2);
end.
```

รูปที่ 5.1 โปรแกรม IncWithCond

โปรแกรมที่นำมาทดสอบ IncWithCond ประกอบด้วยผลลัพธ์จำนวน 2 ผลลัพธ์ และเมื่อทำการใส่ข้อผิดพลาดให้กับโปรแกรม IncWithCond จำนวนโปรแกรมมิวแทนท์ที่ได้รับมีทั้งหมด 44 โปรแกรมและเป็นโปรแกรมมิวแทนท์สมมูล 2 โปรแกรม

ในการทดสอบโปรแกรม IncWithCond นี้ ได้ทำการวัดจำนวนครั้งในการทดสอบโปรแกรมมิวแทนท์โดยวิธีการวิเคราะห์มิวเทชัน 4 วิธีคือ

1. วิธีการวิเคราะห์มิวเทชันแบบเดิม
2. วิธีการวิเคราะห์มิวเทชันแบบหลายข้อผิดพลาด
3. วิธีการวิเคราะห์มิวเทชันแบบแบ่งกลุ่มโปรแกรมมิวแทนท์
4. วิธีการวิเคราะห์มิวเทชันแบบหลายข้อผิดพลาดโดยการประยุกต์ใช้การแบ่งกลุ่มโปรแกรมมิวแทนท์

ผลลัพธ์ที่ได้รับจากการทดสอบโปรแกรมที่นำมาทดสอบโดยวิธีการดังกล่าว เป็นดังในตารางที่ 5.1

ตารางที่ 5.1 ตารางเปรียบเทียบจำนวนครั้งในการทดสอบโดยวิธีแบบต่างๆ

วิธีการวิเคราะห์มิวเทชัน	**มิวแทนท์**	**จำนวนครั้ง**
การวิเคราะห์มิวเทชันแบบเดิม	44	68
การวิเคราะห์มิวเทชันแบบหลายข้อผิดพลาด	26	52
การวิเคราะห์มิวเทชันแบบแบ่งกลุ่มโปรแกรมมิวแทนท์	26 + 18	58
การวิเคราะห์มิวเทชันแบบหลายข้อผิดพลาดโดยการประยุกต์ใช้การแบ่งกลุ่ม	18 + 9	40

ในคอลัมน์มิวแทนท์แสดงถึงจำนวนของโปรแกรมมิวแทนท์ที่ได้รับมาจากแต่ละวิธีการวิเคราะห์มิวเทชัน ซึ่งหากเป็นตัวเลขบวกกัน ตัวเลขแต่ละตัวแสดงถึงจำนวนโปรแกรมมิวแทนท์ในแต่ละกลุ่ม

จากตารางที่ 5.1 แสดงให้เห็นว่าวิธีการมิวเทชันแบบหลายข้อผิดพลาดและวิธีการแบ่งกลุ่มโปรแกรมมิวแทนท์สามารถลดจำนวนครั้งในการทดสอบได้ แต่ในการวิเคราะห์มิวเทชันแบบหลายข้อผิดพลาดธรรมดานั้นยังมีบางโปรแกรมมิวแทนท์แบบหลายข้อผิดพลาดสร้างมาจากข้อผิดพลาดที่มีข้อบังคับในการไปถึงที่ขัดแย้งกัน ดังนั้นในงานวิจัยชิ้นนี้จึงได้นำเอาการแบ่งกลุ่มโปรแกรมมิวแทนท์มาประยุกต์ใช้เพื่อแบ่งกลุ่มของโปรแกรมมิวแทนท์ตามข้อบังคับการไปถึงและสร้างโปรแกรมมิวแทนท์แบบหลายข้อผิดพลาดจากโปรแกรมมิวแทนท์ภายในกลุ่ม

การวิเคราะห์มิวเทชันแบบหลายข้อผิดพลาดโดยการประยุกต์ใช้การแบ่งกลุ่มโปรแกรมมิวแทนท์นั้น ใช้จำนวนครั้งในการทดสอบที่น้อยที่สุดเนื่องมาจากโปรแกรมมิวแทนท์แบบหลายข้อผิดพลาดที่ได้รับมาจากวิธีการนี้มีโอกาสที่จะถูกกำจัดได้ด้วยกรณีทดสอบเพียงกรณีเดียวมากกว่าวิธีการวิเคราะห์มิวเทชันแบบหลายข้อผิดพลาดธรรมดา

ปัจจัยของการลดลงของจำนวนครั้งในการทดสอบโดยวิธีการวิเคราะห์มิวเทชันแบบหลายข้อผิดพลาดโดยการประยุกต์ใช้การแบ่งกลุ่มนี้ ไม่ได้ขึ้นอยู่กับจำนวนโปรแกรมมิวแทนท์ที่สามารถลดได้มากกว่าวิธีการวิเคราะห์มิวเทชันแบบหลายข้อผิดพลาดธรรมดา แต่ขึ้นกับจำนวนโปรแกรมมิวแทนท์แบบหลายข้อผิดพลาดที่มีข้อบังคับการไปถึงที่ไม่ขัดแย้งกัน ตัวอย่างเช่นในตารางที่ 5.1 ถึงแม้ว่าจำนวนโปรแกรมมิวแทนท์แบบหลายข้อผิดพลาดที่ได้รับจากวิธีการวิเคราะห์มิวเทชันแบบหลายข้อผิดพลาดโดยการประยุกต์ใช้การแบ่งกลุ่มมีค่ามากกว่าของการวิเคราะห์มิวเทชันแบบหลายข้อผิดพลาดธรรมดา แต่จำนวนครั้งในการทดสอบโปรแกรมมิวแทนท์นั้นน้อยกว่าการวิเคราะห์มิวเทชันแบบหลายข้อผิดพลาดธรรมดา

ในการสร้างโปรแกรมมิวแทนท์แบบหลายข้อผิดพลาดโดยการประยุกต์ใช้การแบ่งกลุ่มนั้นอาจจำเป็นต้องใช้โปรแกรมมิวแทนท์ของกลุ่มการแบ่งกลุ่มอื่นที่มีข้อบังคับการไปถึงที่ไม่ขัดแย้งมาสร้างเป็นโปรแกรมมิวแทนท์แบบหลายข้อผิดพลาด

5.4 บทพิสูจน์

จากการรวมกันของสองแนวทางเพื่อลดจำนวนครั้งของการทดสอบโปรแกรมโดยวิธีการวิเคราะห์มิวเทชัน สามารถสรุปได้ตามทฤษฎีบทสองบทต่อไปนี้

ทฤษฎีบทที่ 5

จำนวนครั้งของการทดสอบโปรแกรมโดยการแบ่งกลุ่มโปรแกรมมิวแทนท์สามารถลดลงได้เมื่อเปรียบเทียบกับจำนวนครั้งการทดสอบโปรแกรมโดยสร้างโปรแกรมมิวแทนท์แบบหลายข้อผิดพลาดโดยประยุกต์ใช้การแบ่งกลุ่มโปรแกรมมิวแทนท์

พิสูจน์

เนื่องจากกรณีทดสอบที่ใช้ในการทดสอบจะแยกจากกันในแต่ละกลุ่มของโปรแกรมมิวแทนท์ ดังนั้นถ้าจำนวนครั้งของการทดสอบในแต่ละกลุ่มสามารถลดลงได้ จะนำไปสู่การลดลงของจำนวนครั้งของการทดสอบรวม เมื่อพิจารณาการนับจำนวนครั้งของการทดสอบของแต่ละกลุ่มจะ

เห็นว่า จำนวนโปรแกรมมิวแทนท์สามารถลดลงได้โดยการสร้างโปรแกรมมิวแทนท์แบบหลายข้อผิดพลาด แต่จำนวนกรณีทดสอบที่สามารถนำมากำจัดโปรแกรมมิวแทนท์ของแต่ละกลุ่มยังคงเดิม (เพราะข้อผิดพลาดของแต่ละกลุ่มคงเดิม) ดังนั้นจำนวนครั้งของการทดสอบสามารถลดลงได้จริง

ทฤษฎีบทที่ 6

จำนวนครั้งของการทดสอบโปรแกรมโดยการสร้างโปรแกรมมิวแทนท์แบบหลายข้อผิดพลาดสามารถลดลงได้ เมื่อเปรียบเทียบกับจำนวนครั้งการทดสอบโปรแกรมโดยสร้างโปรแกรมมิวแทนท์แบบหลายข้อผิดพลาดโดยอาศัยการแบ่งกลุ่มโปรแกรมมิวแทนท์

พิสูจน์

จำนวนครั้งของการทดสอบโดยวิธีการสร้างโปรแกรมมิงแทนท์แบบหลายข้อผิดพลาดขึ้นกับจำนวนโปรแกรมมิวแทนท์และจำนวนกรณีทดสอบที่ใช้ในการกำจัดโปรแกรมมิวแทนท์แบบหลายข้อผิดพลาดด้วย ในวิธีการสร้างโปรแกรมมิวแทนท์แบบหลายข้อผิดพลาดโดยประยุกต์ใช้การแบ่งกลุ่มมีโอกาสทำให้จำนวนโปรแกรมมิวแทนท์รวมทั้งหมดสูงขึ้น จำนวนที่สูงขึ้นนี้มาจากการที่ข้อผิดพลาดที่มีข้อบังคับการไปถึงขัดแย้งกันมิได้ถูกนำไปสร้างเป็นโปรแกรมมิวแทนท์แบบหลายข้อผิดพลาดได้ เมื่อเป็นเช่นนี้ โปรแกรมมิวแทนท์แบบหลายข้อผิดพลาดโดยวิธีการแบบธรรมดาที่ไม่สามารถสร้างได้โดยวิธีการประยุกต์ใช้การแบ่งกลุ่มโปรแกรมมิวแทนท์ ต้องการกรณีทดสอบมากกว่าหนึ่งกรณีทดสอบมากำจัด จำนวนกรณีทดสอบที่ใช้ต้องขึ้นกับว่าข้อผิดพลาดทั้งหลายที่มารวมกันมีการขัดแย้งกันได้จำนวนกรณีเป็นเท่าใด จำนวนนี้จะเท่ากับจำนวนกลุ่มย่อยในวิธีการประยุกต์ใช้การแบ่งกลุ่มที่ข้อผิดพลาดเหล่านี้กระจายอยู่ ดังนั้นจำนวนครั้งของการทดสอบจะคงเดิม แต่เนื่องจากวิธีที่สองมีการแบ่งโปรแกรมมิวแทนท์ออกเป็นกลุ่มย่อย ทำให้จำนวนครั้งของการทดสอบลดลงแน่นอนถ้ามีกลุ่มของโปรแกรมมิวแทนท์มากกว่าหนึ่งกลุ่ม ดังนั้นเราสามารถสรุปได้ว่าจำนวนครั้งของการทดสอบสามารถลดลงได้จริง

บทที่ 6
สรุปผลการวิจัยและข้อเสนอแนะ

6.1 สรุปผลการวิจัย

การวิเคราะห์มิวเทชันเป็นการทดสอบซอฟต์แวร์ที่มีความละเอียดในการทดสอบโปรแกรมมาก กรณีทดสอบที่ได้รับมาจากการทำการวิเคราะห์มิวเทชันมีประสิทธิผลในการตรวจจับผลกระทบของข้อผิดพลาดสูงกว่ากรณีทดสอบที่ได้รับการสร้างมาจากวิธีการอื่น ถึงอย่างไรก็ตาม ปัญหาสำคัญของการวิเคราะห์มิวเทชันคือ จำนวนครั้งในการทดสอบแต่ละโปรแกรมมิวแทนท์ซึ่งมีจำนวนมาก

งานวิจัยชิ้นนี้ได้เสนอแนวทางในการปรับปรุงประสิทธิภาพของการวิเคราะห์มิวเทชัน โดยการลดจำนวนของโปรแกรมมิวแทนท์ลงโดยการสร้างโปรแกรมมิวแทนท์แบบหลายข้อผิดพลาดซึ่งสามารถใช้เป็นตัวแทนให้กับโปรแกรมมิวแทนท์หลายๆ โปรแกรมได้

ข้อดีของการสร้างโปรแกรมมิวแทนท์แบบหลายข้อผิดพลาดเพื่อลดจำนวนโปรแกรมมิวแทนท์ที่ต่างจากหลายๆ วิธีที่ผ่านมาคือประสิทธิผลของชุดของกรณีทดสอบที่ได้รับจะมีประสิทธิผลที่ไม่ลดลงไปจากเดิมตามทฤษฎีบทที่ 2

จากการทดลองพบว่าเปอร์เซ็นต์การลดลงของจำนวนโปรแกรมมิวแทนท์โดยเฉลี่ยของโปรแกรมที่นำมาทดสอบทั้ง 5 โปรแกรมมีค่า 45.46%

ปัจจัยที่ส่งผลต่อการลดลงของจำนวนโปรแกรมมิวแทนท์เมื่อใช้วิธีการสร้างโปรแกรมมิวแทนท์แบบหลายข้อผิดพลาดมีด้วยกัน 2 ปัจจัยคือ

1. จำนวนผลลัพธ์ของโปรแกรมที่นำมาทดสอบ โดยหากโปรแกรมที่นำมาทดสอบนั้นมีจำนวนผลลัพธ์มากแล้ว โอกาสที่โปรแกรมมิวแทนท์โปรแกรมหนึ่งๆ สามารถเป็นตัวแทนให้กับโปรแกรมมิวแทนท์หลายๆ โปรแกรมมีมากขึ้น

2. จำนวนข้อผิดพลาดที่จะใส่ให้กับคำสั่งที่ไม่ใช่คำสั่งร่วมกัน โดยหากโปรแกรมที่ได้รับการตัดส่วนมาจากโปรแกรมที่นำทดสอบมีคำสั่งที่ไม่ใช่คำสั่งร่วมกันและข้อผิดพลาดที่จะใส่ให้กับคำสั่งเหล่านี้จำนวนมาก การลดลงของจำนวนโปรแกรมมิวแทนท์จะมีมากเนื่องมาจากสามารถสร้างโปรแกรมมิวแทนท์แบบหลายข้อผิดพลาดได้มาก

ปัญหาในการสร้างโปรแกรมมิวแทนท์แบบหลายข้อผิดพลาด คือ หากโปรแกรมที่นำมาทดสอบประกอบด้วยผลลัพธ์เพียงผลลัพธ์เดียวแล้ว จำนวนของโปรแกรมมิวแทนท์จะไม่ลดลงไปจากเดิม และหากโปรแกรมมิวแทนท์แบบหลายข้อผิดพลาดได้รับการใส่ข้อผิดพลาดที่มีข้อบังคับในการตรวจจับผลกระทบของข้อผิดพลาดที่ขัดแย้งกันแล้ว เราจำเป็นต้องใช้จำนวนของกรณีทดสอบจำนวนมากมากำจัดโปรแกรมมิวแทนท์แบบหลายข้อผิดพลาดนี้

นอกจากจำนวนโปรแกรมมิวแทนท์จะเป็นปัจจัยสำคัญของจำนวนครั้งในการทดสอบโปรแกรมมิวแทนท์แล้ว วิธีในการทดสอบโปรแกรมมิวแทนท์ยังเป็นอีกปัจจัยที่ส่งผลให้จำนวนครั้งในการทดสอบโปรแกรมมิวแทนท์เพิ่มขึ้น เนื่องมาจากวิธีในการทดสอบโปรแกรมมิวแทนท์แบบเดิมนั้นมีการทดสอบโปรแกรมมิวแทนท์ที่ไม่สามารถกำจัดได้ด้วยกรณีทดสอบที่นำไปทดสอบแต่สามารถกำจัดได้ด้วยกรณีทดสอบถัดมา ดังนั้นในงานวิจัยชิ้นนี้ได้เสนอวิธีในการทดสอบโปรแกรมมิวแทนท์แบบใหม่โดยการแบ่งกลุ่มโปรแกรมมิวแทนท์

จากทฤษฎีบทที่ 3 ของการแบ่งกลุ่มโปรแกรมมิวแทนท์นั้น จำนวนครั้งในการทดสอบโปรแกรมมิวแทนท์จะลดลงได้ก็ต่อเมื่อ กลุ่มของโปรแกรมมิวแทนท์ที่ได้รับการแบ่งเรียบร้อยแล้ว แต่ละมิวแทนท์ภายในกลุ่มมีความน่าจะเป็นในการถูกกำจัด (α') มากกว่าความน่าจะเป็นของการกำจัดโปรแกรมมิวแทนท์ก่อนการแบ่งกลุ่ม (α) ซึ่งจากทฤษฎีบทที่ 4 นั้นแสดงให้เห็นว่าอัลกอริธึมในการแบ่งกลุ่มโปรแกรมมิวแทนท์โดยอาศัยข้อบังคับการไปถึงมาเป็นเกณฑ์ในการแบ่ง จะสามารถแบ่งกลุ่มของโปรแกรมมิวแทนท์ที่แต่ละโปรแกรมมิวแทนท์ภายในกลุ่มมีความน่าจะเป็นในการถูกกำจัดไม่ลดลงไปจากก่อนการแบ่งกลุ่ม ดังนั้นการแบ่งกลุ่มด้วยอัลกอริธึมที่เสนอในงานวิจัยชิ้นนี้จะสามารถลดจำนวนครั้งในการทดสอบโปรแกรมมิวแทนท์ได้

จากการทดลองพบว่าเปอร์เซ็นต์การลดลงของจำนวนครั้งในการทดสอบโปรแกรมมิวแทนท์โดยเฉลี่ย 25.30% ของวิธีการแบ่งกลุ่ม

ในทางทฤษฎีนั้นหากเราแบ่งกลุ่มโปรแกรมมิวแทนท์โดยอาศัยข้อบังคับการไปถึงและข้อบังคับที่จำเป็นมาเป็นเกณฑ์ในการแบ่งกลุ่ม จำนวนกลุ่มที่ได้รับการแบ่งจะมีมากกว่าเดิมอีกทั้งโปรแกรมมิวแทนท์ของแต่ละกลุ่มมีความน่าจะเป็นในการถูกกำจัดที่มากกว่าการแบ่งกลุ่มโดยอาศัยข้อบังคับการไปถึงมาเป็นเกณฑ์ในการแบ่ง ถึงอย่างไรก็ตามความซับซ้อนในการวิเคราะห์ความขัดแย้งกันของข้อบังคับการไปถึงและข้อบังคับที่จำเป็นนั้นมีความซับซ้อนมาก ทำให้เวลาที่เสียไปในทางปฏิบัติจริงอาจไม่ลดลงไปจากเดิม

จากปัญหาในการวิเคราะห์มิวเทชันแบบหลายข้อผิดพลาดดังที่ได้กล่าวมา ในงานวิจัยชิ้นนี้ยังได้พิจารณาถึงการนำเอาการแบ่งกลุ่มโปรแกรมมิวแทนท์มาประยุกต์ใช้กับการวิเคราะห์มิวเทชันแบบหลายข้อผิดพลาด วัตถุประสงค์เพื่อแก้ปัญหาของการวิเคราะห์มิวเทชันแบบหลายข้อผิดพลาด โดยการแบ่งกลุ่มโปรแกรมมิวแทนท์ตามอัลกอริธึมในการแบ่งกลุ่มโปรแกรมมิวแทนท์ จากนั้นทำการวิเคราะห์มิวเทชันแบบหลายข้อผิดพลาดจากโปรแกรมมิวแทนท์ในแต่ละกลุ่ม

ข้อได้เปรียบของการวิเคราะห์มิวเทชันแบบหลายข้อผิดพลาดที่มีการประยุกต์ใช้การแบ่งกลุ่มโปรแกรมมิวแทนท์ที่เหนือกว่าการวิเคราะห์มิวเทชันแบบหลายข้อผิดพลาดแบบธรรมดา ไม่ใช่จำนวนของโปรแกรมมิวแทนท์แบบหลายข้อผิดพลาดที่ได้รับมีจำนวนน้อยกว่า แต่เป็นจำนวนครั้งในการทดสอบโปรแกรมมิวแทนท์แบบหลายข้อผิดพลาด มีจำนวนครั้งน้อยกว่าจำนวนครั้งในการทดสอบโปรแกรมมิวแทนท์แบบหลายข้อผิดพลาดที่ได้รับมาจากการวิเคราะห์มิวเทชันแบบหลายข้อผิดพลาดแบบธรรมดา

ปัจจัยสำคัญของการประยุกต์ใช้การแบ่งกลุ่มนี้คือ โปรแกรมที่นำมาทดสอบควรประกอบด้วยผลลัพธ์ที่มากกว่าหนึ่งผลลัพธ์และมีทางแยกเพื่อให้สามารถแบ่งกลุ่มได้ตามอัลกอริธึมในการแบ่งกลุ่มโปรแกรมมิวแทนท์ ซึ่งคำสั่งที่อยู่ในแต่ละทางแยกนั้นควรเป็นคำสั่งที่ไม่ใช่คำสั่งร่วมกันและสามารถสร้างเป็นโปรแกรมมิวแทนท์ได้จำนวนมาก

ปัญหาสำคัญในการปรับปรุงด้วยวิธีการนี้ คือ เมื่อเราตัดส่วนโปรแกรมเรียบร้อยแล้ว โปรแกรมมิวแทนท์ในกลุ่มที่ได้รับจากการแบ่งตามบรรทัดฐานการตัดมีจำนวนน้อย ทำให้สร้างเป็นโปรแกรมมิวแทนท์แบบหลายข้อผิดพลาดได้น้อย ดังนั้นในงานวิจัยชิ้นนี้จึงได้เสนอให้เลือกโปรแกรมมิวแทนท์ที่เหลือจากกลุ่มของโปรแกรมมิวแทนท์กลุ่มอื่น เพื่อสร้างเป็นโปรแกรมมิวแทนท์แบบหลายข้อผิดพลาดได้ แต่ข้อผิดพลาดที่นำมาจากคนละกลุ่มกันนั้นต้องมีข้อบังคับในการไปถึงที่ไม่ขัดแย้งกัน

6.2 ปัญหาและอุปสรรค

การวิเคราะห์ว่าโปรแกรมมิวแทนท์ใดเป็นโปรแกรมมิวแทนท์สมมูล ผู้ใช้ต้องเป็นผู้ระบุว่าโปรแกรมมิวแทนท์นั้นเป็นโปรแกรมมิวแทนท์สมมูลหรือไม่ ถึงแม้ว่ามีบางงานวิจัยที่ได้นำเสนอแนวทางในการตรวจสอบโปรแกรมมิวแทนท์สมมูลแบบอัตโนมัติแต่ยังไม่สามารถตรวจสอบได้ถูกต้องทุกโปรแกรมมิวแทนท์ อีกทั้งความซับซ้อนของการวิเคราะห์ข้อบังคับในการกำจัดโปรแกรมมิวแทนท์มีสูงมาก

ในการทดสอบโปรแกรมมิวแทนท์แบบหลายข้อผิดพลาดนั้น มีบางโปรแกรมมิวแทนท์ให้ผลลัพธ์โดยการแจ้งว่ามีการทำงานไม่สมบูรณ์ (โปรแกรมหยุดการทำงานก่อนจบโปรแกรม) ในโปรแกรมที่นำมาทดสอบ เช่น การอ้างถึงข้อมูลที่อยู่ในตำแหน่งที่อยู่นอกเหนือขอบเขต ซึ่งผู้ทดสอบไม่สามารถทราบได้ว่าการทำงานที่ผิดพลาดนี้เกิดจากข้อผิดพลาดที่ใส่เข้าไปตัวใด ดังนั้นแนวทางในการแก้ปัญหานี้คือต้องแยกข้อผิดพลาดจากโปรแกรมมิวแทนท์แบบหลายข้อผิดพลาดออกมาแล้วตรวจสอบดูว่าข้อผิดพลาดใดที่ทำให้เกิดการทำงานที่ผิดพลาด ส่วนข้อผิดพลาดที่เหลือนั้นสามารถนำกลับมารวมกันเป็นโปรแกรมมิวแทนท์แบบหลายข้อผิดพลาดได้เหมือนเดิม

เนื่องจากข้อจำกัดในการวิเคราะห์ข้อบังคับในการกำจัดโปรแกรมมิวแทนท์ ทำให้การสร้างกรณีทดสอบในงานวิจัยชิ้นนี้ ผู้ใช้ต้องเป็นผู้สร้างกรณีทดสอบเอง

6.3 แนวทางในการประยุกต์ใช้ร่วมกับงานวิจัยอื่น ๆ

ในงานวิจัยที่ [9] ซึ่งเป็นการนำเอาการตัดส่วนโปรแกรมมาใช้ในการตรวจสอบโปรแกรมมิวแทนท์ว่าเป็นโปรแกรมมิวแทนท์แบบสมมูลหรือไม่และในงานวิจัยที่ [10] ซึ่งเป็นการนำเอากราฟการขึ้นต่อกันมาวิเคราะห์ความขึ้นต่อกันของแต่ละตัวแปรในโปรแกรม เพื่อลดความเป็นไปได้ในการสร้างกรณีทดสอบมาทดสอบกับโปรแกรม

งานวิจัยชิ้นนี้อาศัยการตัดส่วนโปรแกรมมาช่วยในการสร้างโปรแกรมมิวแทนท์แบบหลายข้อผิดพลาดดังนั้นจึงเป็นไปได้ที่นำไปประยุกต์ใช้ร่วมกันงานวิจัยทั้งสองข้างต้น เพื่อให้สามารถตรวจสอบโปรแกรมมิวแทนท์สมมูล, ลดจำนวนกรณีทดสอบที่นำมาทดสอบโปรแกรม และลดจำนวนโปรแกรมมิวแทนท์ ไปพร้อมกัน

6.4 ข้อเสนอแนะในการพัฒนาเพิ่มเติม

แนวทางในการปรับปรุงการวิเคราะห์มิวเทชันแบบหลายข้อผิดพลาดเพิ่มเติม มีด้วยกัน 2 แนวทางคือ

1. ในงานวิจัยชิ้นนี้ได้เสนอให้สร้างโปรแกรมมิวแทนท์ด้วยตัวดำเนินการมิวเทชันแบบอีซีเล็คทีฟ ซึ่งสร้างข้อผิดพลาดที่ไม่ส่งผลกระทบต่อผลลัพธ์ที่เปลี่ยนไปจากเดิม แต่ถึงอย่างไรก็ตามยังมีตัวดำเนินการชนิดอื่นที่สามารถสร้างข้อผิดพลาดเช่นเดียวกันนี้ เช่น การเปลี่ยนแปลงค่าคงที่ (Constant Replacement Operator) ดังนั้นจึงเป็นแนวทางให้ศึกษา

ต่อไปว่ามีตัวดำเนินการอะไรอีกบ้างที่สร้างข้อผิดพลาดที่สามารถนำมาสร้างเป็นโปรแกรมมิวแทนท์แบบหลายข้อผิดพลาด

2. การสร้างโปรแกรมมิวแทนท์แบบหลายข้อผิดพลาดที่เกิดจากข้อผิดพลาดที่ไม่ขัดแย้งกันในข้อบังคับที่จำเป็นและข้อบังคับที่เพียงพอ

แนวทางในการนำไปประยุกต์ใช้ต่อในอนาคต คือ การนำเอาการสร้างโปรแกรมมิวแทนท์แบบหลายข้อผิดพลาดไปประยุกต์ใช้กับการวิเคราะห์มิวเทชันในระดับของข้อกำหนดทางซอฟต์แวร์ พร้อมทั้งปรับปรุงการวิเคราะห์มิวเทชันแบบหลายข้อผิดพลาดให้สอดคล้องกับตัวดำเนินการมิวเทชันต่างๆ ในระดับของข้อกำหนดทางซอฟต์แวร์

6.5 ผลงานที่เกี่ยวข้องกับงานวิจัย

ส่วนหนึ่งของวิทยานิพนธ์ชิ้นนี้ได้รับการคัดเลือกให้นำไปเสนอผลงานทางวิชาการและตีพิมพ์ในเอกสาร

Proceeding of the IASTED International Conference on Software Engineering (SE-2004) ในหัวข้อ “Composite Mutant: An Innovative Approach to Mutation Testing” ในระหว่างวันที่ 17-19 กุมภาพันธ์ 2547 โดยรายละเอียดอยู่ในภาคผนวก ข

Proceeding of the 7th National Computer Science and Engineering Conference (NCSEC 2003) ในหัวข้อ “An Improvement of Mutation Analysis by Multiple-Fault Mutant” ในระหว่างวันที่ 28-30 ตุลาคม 2546 โดยรายละเอียดอยู่ในภาคผนวก ค

Proceeding of the 7th National Computer Science and Engineering Conference (NCSEC 2003) ในหัวข้อ “Time Reducing in Mutation Analysis by Mutant Classification” ในระหว่างวันที่ 28-30 ตุลาคม 2546 โดยรายละเอียดอยู่ในภาคผนวก ง

รายการอ้างอิง

[1] James M. Bieman, Sudipto Ghosh, Roger T. Alexander. “A Technique for Mutation of Java Objects”, *Proceedings Automated Software Engineering Conference (ASE 2001)*, 2001: 337-340.

[2] P. E. Black, V. Okun, Y. Yesha. “Mutation Operators for specification”, *IEEE International Conference on Automated Software Engineering*, 2000: 81.

[3] P. E. Black, V. Okun, Y. Yesha. “Mutation of Model Checker Specifications for Test Generation and Evaluation”, *Mutation Testing for the New Century*, 2000: 14-20.

[4] P. Chevalley. “Applying Mutation Analysis for Object-Oriented Programs Using a Reflective Approach”, *Asia-Pacific Software Engineering Conference,* 2001: 267.

[5] R. A. Demillo, R. J. Lipton, and F.G. Sayward. “Hints on Test Data Selection: Help for the Practicing Programmer”, *IEEE Computer*, 11(4), 1978: 34-41.

[6] R. A. Demillo , D. S. Guindi , W. M. McCracken, A.J. Offutt, K. N. King. “An Extended Overview of the Mothra Software Testing Environment”, *Second Workshop on Software Testing, Verification, and Analysis*, 1988: 142-151.

[7] R. A. Demillo , A. J. Offutt . “Constraint-Based Automatic Test Data Generation”, *IEEE Transaction on Software Engineering*, 1991: 900-910.

[8] R. A. Demillo, A. J. Offutt. “Experimental Result from an Automatic Test Case Generator”, *ACM Transaction of Software Engineering and Methodology*, 1993: 109-127.

[9] Rob Hierons, Mark Harman and Sebastian Danicic. “Using Program slicing to Assist in the Detection of Equivalent Mutants”, *Software Testing, Verification and Reliability*, 1999: 233-262.

[10] Rob Hierons, Mark Harman and Sebastian Danicic. “The Relationship between Program Dependence and Mutation Analysis”, *Mutation Testing for the New Century*, 2000: 5-13.

[11] S. C. Lee, A. J. Offutt. “Generating Test Cases for XML-Based Web Component Interactions Using Mutation Analysis”, *The Twelfth IEEE International Symposium on Software Reliability Engineering (ISSRE ‘01)*, 2001: 200-209.

[12] A. J. Offutt. “Investigations of the Software Testing Coupling Effect”, *ACM Transaction on Software Engineering and Methodology*, 1992: 5-20.

[13] A. J. Offutt, S. D. Lee. “An Empirical Evaluation of Weak Mutation”, *IEEE Transaction on Software Engineering*, 1994: 337-344.

[14] A. J. Offutt. “A Practical System for Mutation Testing: Help for the Common Programmer”, *International Conference on Testing Computer Software*, 1995: 99-109.

[15] A. J. Offutt, G. Rothermel, R. H. Untch, and C. Zapf. “An Experimental Determination of Sufficient Mutant Operators”, *ACM Transaction of Software Engineering Methodology*, 1996: 99-118

[16] A. J. Offutt, J. Pan. “Automatically Detecting Equivalent Mutants and Infeasible Paths”, *The journal of Software Testing, Verification and Reliability*, 1997: 165-192.

[17] A. J. Offutt, Zhenyi Jin, Jie Pan. “The Dynamic Domain Reduction Procedure for Test Data Generation”, *Software Practice and Experience,* 1997: 167-193.

[18] K. Ottenstein, and L Ottenstein. “The program dependence graph in a software development environment”, *ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, 1984: 177-184.

[19] P. Vado, Y. Savaria, Y. Zoccarato and C. Robach. “A methodology for validating digital circuits with mutation testing”, *The 2000 IEEE International Symposium on, Volume 1*, 2000: 343-346.

[20] M. Weiser. “Program slicing”, *IEEE Transaction on Software Engineering*, 1984: 352-357.

ภาคผนวก

ภาคผนวก ก

โปรแกรมที่นำมาทดสอบ

โปรแกรมที่นำมาทดสอบในงานวิจัยชิ้นนี้มีด้วยกัน 9 โปรแกรมดังต่อไปนี้

1. โปรแกรม SinCos ได้รับการแสดงไว้ในรูปที่ ก.1

```
Program SinCos;
var     i,x : integer;
        cos,sin,term : real;
begin
        readln(x);
        i := 1;
        sin := x;
        term := x;
        while i <= 21 do
        begin
                term := term*(x*x*(-1))/((i+2)*(i+1));
                sin := sin + term;
                i := i + 2;
        end;
        i := 1;
        cos := 1;
        term := 1;
        while i <= 20 do
        begin
                term := term*(x*x*(-1))/((i+1)*(i));
                cos := cos + term;
                i := i + 2;
        end;
        writeln(sin);
        writeln(cos)
end.
```

รูปที่ ก.1 โปรแกรม SinCos

2. โปรแกรม MulSumMatrix ได้รับการแสดงไว้ในรูปที่ ก.2

```
Program MulSumMatrix;
var     a,b,mul,sum : array [1..2,1..2] of integer;
        i, j, k : integer;
begin
    write('input Matrix A[1,1]=');  readln(a[1,1]);
    write('input Matrix A[1,2]=');  readln(a[1,2]);
    write('input Matrix A[2,1]=');  readln(a[2,1]);
    write('input Matrix A[2,2]=');  readln(a[2,2]);
    write('input Matrix B[1,1]=');  readln(b[1,1]);
    write('input Matrix B[1,2]=');  readln(b[1,2]);
    write('input Matrix B[2,1]=');  readln(b[2,1]);
    write('input Matrix B[2,2]=');  readln(b[2,2]);
    for i:=1 to 2 do
    begin
        for j:=1 to 2 do
        begin
                sum[i,j] := a[i,j]+b[i,j];
                for k:=1 to 2 do
                begin
                        mul[i,j] :=  mul[i,j]+ a[i,k]*b[k,j];
                end;
        end;
    end;
    for i:=1 to 2 do
    begin
        for j:=1 to 2 do
        begin
                writeln('output SUM[',i,',',j,']=',sum[i,j]);
        end;
    end;
```

รูปที่ ก.2 โปรแกรม MulSumMatrix

```
    for i:=1 to 2 do
    begin
        for j:=1 to 2 do
        begin
        writeln('output MUL[',i,',',j,']=',mul[i,j]);
        end;
    end;
end.
```

รูปที่ ก.2 โปรแกรม MulSumMatrix (ต่อ)

3. โปรแกรม MulSumFraction ได้รับการแสดงไว้ในรูปที่ ก.3

```
Program MulSumFraction;
uses InttoStr;
var     firstRe,firstDe,secondRe,secondDe : integer;
        MulpRe,MulpDe,sumRe,sumDe,maxDe,minsum : integer;
        gcd,lcm,i : integer;
        resultSum,resultMul,first,second,tmp : String;
begin
 write('Please input a first remainder :');
 readln(firstRe);
 write('Please input a first devider :');
 readln(firstDe);
 write('Please input a second remainder :');
 readln(secondRe);
 write('Please input a second devider :');
 readln(secondDe);
 MulpRe := firstRe*secondRe;
 MulpDe := firstDe*secondDe;
```

รูปที่ ก.3 โปรแกรม MulSumFraction

```
if abs(MulpRe) > abs(MulpDe) then
begin
        minsum := MulpDe;
end
else
begin
        minsum := MulpRe;
end;
for i:=abs(minsum) downto 1 do
begin
        if (MulpRe mod i = 0) and (MulpDe mod i = 0) then
        begin
                gcd := i;
                break;
        end;
end;
MulpRe := MulpRe div gcd;
MulpDe := MulpDe div gcd;
if abs(firstDe) > abs(secondDe) then
begin
         maxDe := abs(firstDe);
end
else
begin
        maxDe := abs(secondDe);
end;
while (maxDe <> 0) do
begin
        if (maxDe mod firstDe = 0) and (maxDe mod secondDe = 0) then
        begin
                lcm := maxDe;
                break;
        end;
```

รูปที่ ก.3 โปรแกรม MulSumFraction (ต่อ)

```
            maxDe := maxDe + 1;
  end;
  sumRe := (firstRe*(lcm div firstDe)) + (secondRe*(lcm div secondDe));
  sumDe := lcm;
  first := IntStr(MulpRe);
  second := IntStr(MulpDe);
  resultMul := first+'/'+second;
  first := IntStr(sumRe);
  second := IntStr(sumDe);
  resultSum := first+'/'+second;
  writeln('The result of multiplication fraction is ',resultMul);
  writeln('The result of summation fraction is ',resultSum);
end.
```

รูปที่ ก.3 โปรแกรม MulSumFraction (ต่อ)

4. โปรแกรม Mean ได้รับการแสดงไว้ในรูปที่ ก.4

```
Program Mean;
var       HmMean, sqrOfSqSum, Avg : Real;
          multiple, sum, sumSq,i : integer;
          Temp : array [1..5] of integer;
begin
  for i :=1 to 5 do
  begin
          write('Please input number ',i,' = ');
          readln(Temp[i]);
  end;
  sum := 0;
  multiple := 1;
  sumSq := 0;
```

รูปที่ ก.4 โปรแกรม Mean

```
for i:=1 to 5 do
 begin
            sum := sum + Temp[i];
            multiple := multiple * Temp[i];
            sumSq := sumSq + Temp[i]*Temp[i];
 end;
 HmMean := sum/(multiple*5);
 sqrOfSqSum := sqrt(sumSq);
 Avg := sum/5;
 writeln(HmMean);
 writeln(sqrOfSqSum);
 writeln(Avg);
end.
```

รูปที่ ก.4 โปรแกรม Mean (ต่อ)

5. โปรแกรม MaxMinAvg ในรูปที่ ก.5

```
Program MaxMin;
var       i, max, min : integer;
            avg : real;
            Temp : array[1..5] of integer;
begin
 for i:=1 to 5 do
 begin
            write('Please input number',i,' = ');
            readln(Temp[i]);
 end;
 max := Temp[1];
 min := Temp[1];
 avg := Temp[1];
```

รูปที่ ก.5 โปรแกรม MaxMinAvg

```
for i:=2 to 5 do
 begin
        if Temp[i] >= max then
        begin
                max := Temp[i];
        end
        else if Temp[i] <= min then
        begin
                min := Temp[i];
        end;
        avg := avg+Temp[i];
 end;
 avg := avg / 5;
 writeln(max);
 writeln(min);
 writeln(avg);
end.
```

รูปที่ ก.5 โปรแกรม MaxMinAvg (ต่อ)

6. โปรแกรม IncDec ในรูปที่ ก.6

```
Program IncDecVal;
var max,min,v1,v2 : integer;
begin
 readln(v1);
 readln(v2);
 if v1 >= v2 then
 begin
        v2 := v2 + 10;
        v1 := v1 - 5;
 end
```

รูปที่ ก.6 โปรแกรม IncDec

```
else
begin
        v1 := v1 + 10;
        v2 := v2 - 5;
end;
 writeln(v1);
 writeln(v2);
end.
```

รูปที่ ก.6 โปรแกรม IncDec (ต่อ)

7. โปรแกรม Triangle ในรูปที่ 4.3

8. โปรแกรม ProdSum ในรูปที่ ก.7

```
Program ProductSum;
var product,sum,n,i : integer;
begin
 readln(n);
 i := 1;
 sum := 0;
 product := 1;
 while (i <= n) do
 begin
        if n > 5 then
        begin
                sum := sum + 1;
                product := product * 2;
        end
        else
        begin
                sum := sum + 2;
                product := product * 3;
```

รูปที่ ก.7 โปรแกรม ProdSum

```
        end;
        i := i + 1;
  end;
  writeln(product);
  writeln(sum);
end.
```

รูปที่ ก.7 โปรแกรม ProdSum (ต่อ)

9. โปรแกรม IncWithCond ในรูปที่ 5.1

ภาคผนวก ข

ผลงานตีพิมพ์ในงาน SE-2004

Composite Mutant: An Innovative Approach to Mutation Testing

W. Worawatpibul and A. Surarerks
Centre of Excellence in Software Engineering,
Department of Computer Engineering,
Faculty of Engineering, Chulalongkorn University, Thailand
warot.w@student.chula.ac.th, athasit@cp.eng.chula.ac.th

Abstract

This research concerns software testing using mutation analysis, a powerful technique for unit testing. The number of generated mutants can have a profound influence on the execution time. Our aim is to reduce the number of mutants without an effect on the effectiveness of testing. We propose an algorithm for generating composite mutants from independent mutants identified by program slicing technique. All of mutants are grouped with respect to slicing criterion into composite mutants. The experiments demonstrate that our method can save the number of mutants. Accordingly, the execution time to kill mutants can be reduced with this approach. We also prove that the number of test cases to kill composite mutants is not greater than the one to kill classic mutants.

Key Words

Software testing, Mutation testing, Constraint-based testing, Program slicing

1. Introduction

Software testing, one of the most important activities to guarantee the correctness of software, has an objective to demonstrate the presence of errors. Unfortunately, software testing is very labour-intensive and requires high computation. There are literally hundreds of techniques for reducing the cost of testing. The performance of testing depends on the quality of test data. The test data is a subset of input domain

satisfying testing criterion and composes of several test cases. The phrase “Fault-based testing” has been applied to the techniques, such as mutation, in generating test data to indicate the presence of specific faults during testing process.

Mutation analysis has been first proposed in [5]. This method is designed to substantiate the correctness of a program unit. It is realized by introducing a syntactical change into the original program, thereby creating a set of *mutant* programs. Each mutant represents a fault in program, and the goal of tester is to construct a set of test cases that distinguishes the output of the original program from that of all mutants. Mutation analysis was applied in [1, 2, 3, 13]. Regrettably, the problem of conventional method is the waste of time for interpreting many slightly different mutants. The focus of this paper is not only to reduce the number of mutants but also to preserve the effectiveness of testing.

This paper is organized as follows. In Section 2, some definitions, notations, and basic techniques are clarified. A novel approach to form composite mutants using a slicing technique is established in Section 3. Section 4 provides empirical results from implementation. Finally, Section 5 concludes with the direction for future work.

2. Background

We start with the notions of mutation analysis and test data generation. Then, the technique of program slicing is explained.

2.1 Mutation Analysis

Mutation analysis is a fault-based testing technique that measures the effectiveness of test data [15]. A tester will determine the adequacy and enhance the quality of test data set. To produce mutants, specific types of faults are introduced as the syntactical changes to the original program.

Example 1, given program FindMax(m,n)

```
001   int FindMax(int m, int n)
002   {
003             max = m;
004             if (n>m) max = n; □max = m;
005             return(max)
006   }
```

In Example 1, we insert a fault by changing the statement max = n to max = m and then generate a test case which can propagate this fault to the output of program. First, the test case is run against the original program, and the tester examines the output. If the test case discovers the fault, the program must be corrected. On the contrary, save its output as *Expected output.* Next, the test case is run on each mutant and the output of each mutant is compared with the expected output.

Definition 1

Let P be the original program. A program M is said to be a mutant *of P if M is obtained by modifying P with a small syntactical change. Such a change is modelled in mutation operations.*

These mutation operators are the operators used for generating the mutants. Some mutation operators are described in [15]. To perform mutation testing, a set of test cases is constructed in order to distinguish the output of all mutant programs from that of the original program.

Definition 2

A mutant program M is said to be killed *by a test case t if M, executed with the test case t, denoted by M(t), gives the output which differs from the output of the original program P, executed with the test case t, P(t).*

In Example 1, the expected output (max = 2) is obtained from the original program with a test case (n=2, m=1). The mutant with (max = m) gives the output (max = 1) by the

same test case. Since the output of mutant differs from expected output, this test case is said to kill the mutant.

Each mutant is executed on every test case until it is killed or passed all test cases in the test data set. The test data that kills all mutants is adequate for that set of mutants. A program which is successfully tested with an adequate test data is correct; otherwise, it may contain faults that have not been represented by the mutants. Unfortunately, the number of all mutants is very large, so an execution of all mutants is time-consuming.

Definition 3

A mutant M is functionally equivalent *to its original program P if M cannot be killed by any test data.*

Usually, equivalent mutants are detected by hand, which makes it very expensive and time-consuming. Some researches propose the techniques that automatically detect equivalent mutants [10, 17].

E-selective mutation, originated by Offutt et al [16], is an approximation technique that selects only mutants which are created by expression modification. The basic idea stems from two observations: (1) the classical mutation is too inefficient in term of performance, and (2) this inefficiency happens as a result of redundancy in the used mutants. The operators in E-Selective method are shown as the following:

1. UOI : Unary Operator Insertion
2. ROR: Relational Operator Replacement
3. AOR: Arithmetic Operator Replacement
4. LCR: Logical Connection Replacement
5. ABS: Absolute Value Insertion

In their experiments, E-Selective provides almost the same coverage as the classical mutation, with high reduction in cost.

2.2 Constraint-Based Technique

A mutant will be killed by an *effective test case* represented in killing constraint [7] described by the three conditions. The first is Reachability Constraint. A mutated statement is represented as an executable syntactic change but the other statements are equivalent to the original program. The minimum requirement for a test case to kill a mutant is that the mutated statement must be executed. The next condition is *Necessity Constraint*. This condition means that a state of a mutant must differ from the state of its original program after the mutated statement has been executed. The last condition is *Sufficient Constraint*. It is the strongest condition that the final state of the mutant differs from that of the original program.

The constraints of mutants are able to detect equivalent mutants shown by Offutt and Pan in [17]. *Constraint-based testing* (CBT) [7, 8] uses the first two conditions as simple algebraic constraints to produce the test cases. If a test case kills the mutants of the original program, the result of a substituted constraint will be true. This technique was fully implemented in a tool called Godzilla Test Data Generator, integrated within the *Mothra* software testing system [4, 6]. CBT has been extended to *Dynamic Domain Reduction* (DDR) [18] that is capable in handling loop and array.

2.3 Program Slicing

Program slicing, introduced by Mark Weiser [20], has been found to be useful in program analysis, debugging and reengineering. In order to analyze an interesting point in a program, other parts which do not have an influence on the point are eliminated. A slicing criterion, denoted by (*V*, *n*), is composed of a set of variables *V*, and a program point *n*. More precisely, all statements of the program that do not affect any variables in *V* at point *n* are removed to form slicing programs. Ottenstein [19] was the first to define slicing as the reachable problem in a dependence graph representation of a program. They used a program dependence graph (PDG) [9] for static slicing of a procedure of a

program. Statements and expressions of a program are represented by vertices of a PDG, and edges correspond to data dependence and control dependence. Data dependence represents data flow relationship of a program, while control dependence is introduced to represent only the essential control flow relationship of a program. In this paper, we are interested in slicing programs that are executable. The slicing technique in Example 2 is said to be a *static backward* slicing which does not really execute program and use backward traversal on PDG.

Example 2, given program Prod_Sum(int n) .

```
001  void Prod_Sum(int n)
002  {
003          i = 1;
004          sum = 0;
005          product = 1;
006          while (i<=n)
007          {
008                  sum = sum + i;
009                  product = product * i;
010                  i = i + 1;
011          }
012          printf("%d", sum);
013          printf("%d", product);
014  }
```

The program dependence graph is shown in Figure 1. The thick edges represent control dependencies and dash lines represent data dependencies. Shading is used for indicating the statements that affect the slicing criterion, ({sum},12).

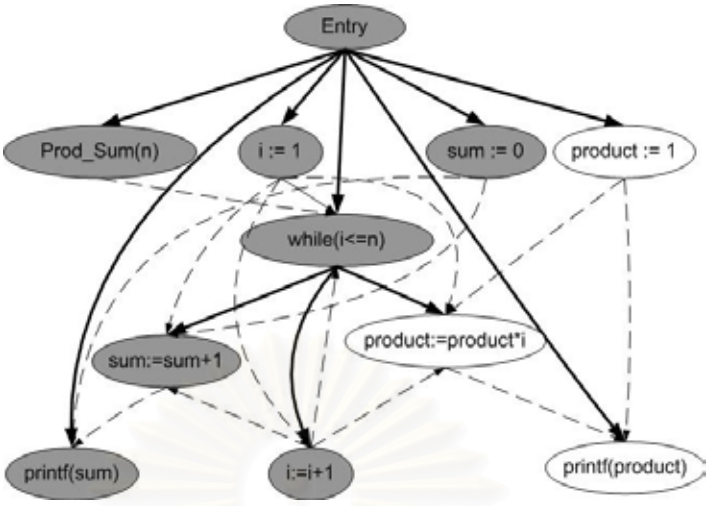


Figure 1: The program dependence graph of program Prod_Sum

The slicing program ({sum},012) is shown as follow:

```
001   void Prod_Sum(int n)
002   {
003             i = 1;
004             sum = 0;
006             while (i<=n)
007             {
008                       sum = sum + i;
010                       i = i + 1;
011             }
012             printf("%d", sum);
014   }
```

Line 005, 009 and 013 are eliminated because they do not have any effect on variable "sum" at line 012. □

3. Composite Mutation

Since mutation testing uses high computation to execute the entire mutants, Demillo proposed the coupling effect theorem [5] states that the test data which can detect a single fault are effective in the detection of multiple faults. From this theorem, the mutation aims to inject only one fault into the original program and to ignore the multiple faults, for details refer to [14]. Even though the using Coupling Effect Theorem can

ignore multiple-faults mutants, the amount of them is still large. The majority of the previous dissertations about the reduction of execution time have to decrease the effectiveness of the testing. This problem is solved by our approach.

To improve the performance of the mutation technique, program slicing is applied to the detection of equivalent mutant in [10] and the reduction of test cases in [11]. This section focuses on the composite mutation, an efficient technique in reducing the number of mutants, and the advantage of program slicing.

3.1 Definition of Composite Mutant

By inserting several faults into a program, we obtain a composite mutant which represents a group of mutants.

Definition 5

Let P be the original program containing statements s_1, s_2, s_3,…,s_n . Program C is said to be a composite mutant *if there exists a positive integer k such that*

$$P \underset{s_1}{\overset{u_1}{\Rightarrow}} M_1 \underset{s_2}{\overset{u_2}{\Rightarrow}} M_2 \underset{s_3}{\overset{u_3}{\Rightarrow}} M_3 \ldots \underset{s_k}{\overset{u_k}{\Rightarrow}} M_k = C ,$$

where $P \underset{s_j}{\overset{u_j}{\Rightarrow}} M$ *is an insertion of fault u_i at statement s_i.*

3.2 Generating Algorithm

Because a composite mutant is generated by injection of multiple faults, it is possible that an injected fault obstructs the propagation of the other faults to the output. Slicing is the way that we choose to solve this problem. In addition, this work also proposes the generating algorithm for creating the composite mutant. Program slicing is used to analyze statements which affect the variable in the slicing criterion. We set the slicing criterion for each output of the original program. From the results of program slicing, we can separate the mutants into independent groups with respect to the outputs of program. Afterward, a mutant from each group must be selected and combined together. The generating algorithm is shown as the following.

Given a program P with n outputs, o_1, o_2, o_3, ..., o_n,

Step1. The program P is sliced into n sliced programs, P_1, P_2, P_3, …, P_n for each $1 \leq j \leq n$, P_j contains all statements that affect the output o_j.

Step2. For each P_j, form the set SM_j which is the set of all mutants generated from P_j.

Step3. For any P_i and P_j, $i \neq j$, remove all mutants generated from common statements that appear in both slicing programs, from P_i and P_j.

Step4. Remove all equivalent mutants from SM_j.

Step5. Generate the composite mutants from the following algorithm:

```
for i=1 to max(|SM1|,|SM2|,..,,|SMn|) do
begin // |SMj| means the number of elements of SMj.
    M= P
    for j=1 to n do
    begin
        if |SMj|<> 0 then
        begin
          M' = insert fault uj at statement sj to M
          SMj=SMj\{uj}
        end;
    end;
    Ci = M
end;
output Mutant1 = {C1,C2,..., Cn}
```

Step6. Form the set of all mutants generated from the common statement, $Mutant_2 = \{M_1, M_2, \ldots, M_k\}$. The mutants used in our approach are from both of $Mutant_1$ and $Mutant_2$.

The result of generating algorithm

This algorithm produces the set of composite mutants that can represent several independent mutants.

In fact, there are two reasons for which the composed mutant, a part of a composite mutant, is independent of the other. The first is slicing technique used for classifying the mutants, according to the outputs, into the independent groups. The other one is the types of faults, seeded into the independent statement. These faults are modelled by the E-Selective operators creating anomalies that affect only the output defined in the slicing criterion.

Example 3, *program Prod_Sum(int n) is sliced into two programs, S_s=({sum},012)(see in Example 2.) and S_p=({product,013) as follow:*

```
001   void Prod_Sum(int n)
002   {
003             i = 1;
005             product = 1;
006             while (i<=n) do
007             {
009                   product = product*i;
010                   i = i + 1;
011             }
013             printf("%d",product);
014   }
```

Both S_s and S_p have three common statements, 003, 006 and 010. The mutants from S_s and S_p can be combined together unless they are mutated at the common statements. For instance, the composite mutant C_1 is obtained by combining both faults from M_1 and M_2, shown in Table 1.

Table 1: Fault insertion in S_s and S_p

Mutant	Line No.	Original	Mutate
M_1	004	sum = 0	sum = 5
M_2	005	product = 1	product = 5
C_1(Composite)	004	sum = 0	sum = 5
	005	product = 1	product = 5

Proposition 1

Let C be a composite mutant obtained from the algorithm:

$$P \underset{s_1}{\overset{u_1}{\Rightarrow}} M_1 \underset{s_2}{\overset{u_2}{\Rightarrow}} M_2 \underset{s_3}{\overset{u_3}{\Rightarrow}} M_3 \ldots \underset{s_k}{\overset{u_k}{\Rightarrow}} M_k = C,$$

where u_j is the fault, inserted at statement s_j, which affects the output o_j. If a test case t can distinguish the output o_j of P(t) from $N_j(t)$ which

$$P \underset{s_j}{\overset{u_j}{\Rightarrow}} N_j,\ 1 \le j \le k,$$

then C(t) differs from P(t) at the output o_j.

Proof: Suppose that a test case t can distinguish the output o_j of P(t) from that of N_j(t), inserted the fault at line s_j. We assume that C(t) does not differ from P(t) at the output o_j. Since N_j is differed from P by u_j, and u_j is also inserted into C, we conclude that C has another fault statement s_0 that affects o_j and prevents the propagation of the fault u_j. That is s_0 and s_j are in the same slicing program with respect to o_j. This contradicts the generating algorithm choosing a mutant from each slicing program. □

3.3 Killing the Composite Mutant

A mutant in the classical approach is killed if it has at least one output that is not equal to the expected output. Killing in the composite mutation, however, requires all outputs that depend on fault statements must differ from the same outputs of the original program.

Several mutants can be killed with one test case if they do not have the contradiction between their constraints [7]. In the same way, composite mutant will be killed with one time of execution if all mutants composed in a composite mutant do not have any contradictions in the constraints of the others.

Definition 6

Let P be a program with n outputs and C be a composite mutant such that

$$P \underset{s_1}{\overset{u_1}{\Rightarrow}} M_1 \underset{s_2}{\overset{u_2}{\Rightarrow}} M_2 \underset{s_3}{\overset{u_3}{\Rightarrow}} M_3 \ldots \underset{s_k}{\overset{u_k}{\Rightarrow}} M_k = C,$$

where u_j is the fault, inserted at statement s_j, which affects the output o_j. C is said to be killed *by the set of test cases T = {t_1, t_2, …, t_m} if for all $1 \leq j \leq k$, there exists t_i that $C(t_i)$ differs from $P(t_i)$ at the output o_j.*

Algorithm for killing composite mutant

Given a set of composite mutants $C_s = \{C_1,\ldots, C_n\}$, we will describe the algorithm in killing it as follow.

Step1. A test case t_i will be generated to kill mutants. The techniques of generation can be Constraint-base testing, Dynamic-Domain Reduction or Random generation etc.

Step2. For any C_i generated from the mutants M_1 to M_k, if t_i kills the mutants M_1 to M_j, where $j < k$, keep the test case t_i and label that M_1 to M_j of C_i are killed.

Step3. If t_i can kill the mutants M_1 to M_j where $j = k$, remove C_i from the composite mutants set C_s.

Step4. Test case t_i is used to execute the other composite mutants C_{i+1} (Go to step 2, 3). If $C_s = \emptyset$, exit this algorithm; otherwise, go to step 1 for generating other test cases.

The result of killing algorithm

This killing algorithm produces the set of test cases that has enough effectiveness in the detection of the faults in mutant programs.

Proposition 2

The set of test data T that kills all mutants in the classical approach is effective enough to kill all composite mutants.

Proof: Let $F=\{u_1, u_2, u_3, ..,u_n\}$ be the set of faults inserted into the program P.

Let C be the composite mutant obtained by introducing some faults in F to P by generating algorithm. Given a set of test data $T=\{t_1, t_2, t_3, ..,t_n\}$ that can detect all faults in F.

We prove the proposition by assuming that for any t_i, there exist out_j such that out_j of $C(t_i)$ = out_j of $P(t_i)$.

Let a fault u_j affecting out_j is introduced to C, and all t_i in T cannot detect u_j, this contradicts that T can detect all faults in F. We infer that none of the faults affecting out_j is introduced to C.

Any faults introduced to C can be detected by some t in T, so a composite mutant C can be killed by T. □

From the previous example, given the test case t (n=5), the results are shown in Table 2:

Table 2: The result of execution

Program with n=5	Sum	product
Program P	15	120
Mutant M_1	20	120
Mutant M_2	15	600
Composite mutant C	20	600

It is clear that mutant M_1 and M_2 can be represented by a composite mutant C that uses only one time of execution to be killed. From this concept, the execution time of the mutants is reduced.

4. Experimentation

We evaluated the composite mutation empirically. Pascal program units were chosen for the experiment. These programs have more than one output and are described in Table 3.

Table 3: The experimental programs

Program	Description
SinCos	Calculate Sin and Cos function in Taylor Theorem
MulSumMatrix	Calculate multiple and summation of Matrix
MulSumFraction	Calculate multiple and summation of fraction
Mean	Find Harmonic, Geometric and Statistic Mean
MaxMinAvg	Find the max, min and average of 5 values

Our experiment began by observing with selective mutation. For each program, we first created E-Selective mutants, and then compared them with the number of composite mutants. For the experiment, we have developed a tool for seeding faults and creating the composite mutants, from the generating algorithm.

Table 4 illustrates the number of mutants obtained by the composite mutation. To compute the "percentage saved" column, we defined the *saved mutant* to be the number of mutants that is a part of composite mutant having at least two faults. Percentage saved was computed by subtracting the saved mutants from the number of the selective mutants and dividing the difference by the number of the selective mutants.

Table 4: The result of experimentation

Program	Selective Mutants	Composite mutants	Percentage Saved
SinCos	192	101	47.40
MulSumMatrix	151	117	22.52
MulSumFraction	295	162	45.08
Mean	86	65	24.42
MaxMinAvg	89	53	40.45
Average			35.97

The result shows that the number of mutants is decreased 47.40% in SinCos program with an average over five programs of 35.97%. The percentage saved of composite mutants depends on the independence of each output, which is the proportion of non-common statement to all statements. If outputs of a program have a great independence, the percentage saved will increase as shown in Figure 2. For example, Mean program which computes harmonic, geometric and statistic means has only one independent statement for each output. The saved mutant of this program is 34 from the total 65 mutants (24.42%).

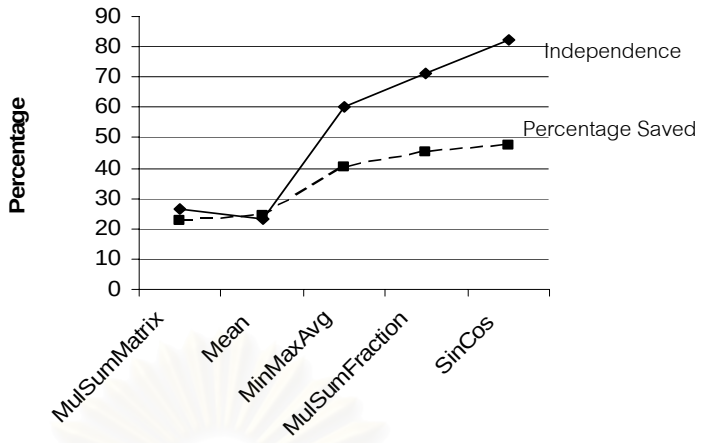


Figure 2: The relation between Percentage Saved and Independence Percentage

Note that MulSumMatrix has the lowest percentage saved because there are a few mutants generated from its independent statements. The amount of slicing program (P_i) is also a factor in reducing the number of mutants. If the original program can be sliced into numerous slicing programs, an occasion for composing several mutants will increase.

5. Conclusion

This paper introduces a novel approach to mutation testing which suffered from a serious problem that is the computational cost in execution of all mutants. Composite mutation, combining several mutants to form a composite mutant, is an economical way to perform mutation technique. The mutants could be composed if each of them does not prevent the propagation of the other faults. Program slicing is used for making an analysis of the dependency of the program statement.

This work is unusual for the previous researches that preserve the single-fault assumption. The most obvious advantage over the several techniques is that composite mutation could maintain the effectiveness of mutation testing. The test cases, therefore, that kill all composite mutants are effective enough for the classical approach. Moreover, our approach could also be applied in the other techniques.

The time used for slicing program is not greater than the execution time of all mutants because of two reasons:

1. Program can have a loop that uses n iterations in execution but excludes from analysing the dependence of variables.
2. The number of mutant programs is large; accordingly, the used time, in the execution of all mutants, is significantly more than slicing time.

Further research will focus on generating composite mutants which can be killed by one test case. Constraint of mutants should be considered.

6. Reference

[1] James M. Bieman, Sudipto Ghosh, Roger T. Alexander, “A Technique for Mutation of Java Objects”, *Proceedings Automated Software Engineering Conference (ASE 2001)*, 2001, 337-340.

[2] P. E. Black, V. Okun, Yaacov Yesha, “Mutation Operators for specification”, *IEEE International Conference on Automated Software Engineering*, 2000, 81.

[3] P. Chevalley, “Applying Mutation Analysis for Object-Oriented Programs Using a Reflective Approach”, *Asia-Pacific Software Engineering Conference,* 2001, 267.

[4] B. J. Choi, R. A. Demillo, E. W. Krauser, R. J. Martin, A. P. Muthur, A. J. Offutt , H. Pan, E. H. Spafford, “The Mothra Tool Set”, *Hawaii International Conference on System Sciences* , 1989, 275-254.

[5] R. A. Demillo, R. J. Lipton, and F.G. Sayward, “Hints on Test Data Selection: Help for the Practicing Programmer”, *IEEE Computer*, 11(4), 1978, 34-41.

[6] R. A. Demillo , D. S. Guindi , W. M. McCracken, A.J. Offutt, K. N. King , “An Extended Overview of the Mothra Software Testing Environment”, *Second Workshop on Software Testing, Verification, and Analysis* , 1988, 142-151.

[7] R. A. Demillo , A. J. Offutt , “Constraint-Based Automatic Test Data Generation”, *IEEE Transaction on Software Engineering* , 1991, 900-910.

[8] R. A. Demillo , A. J. Offutt , “Experimental Result from an Automatic Test Case Generator”, *ACM Transaction of Software Engineering and Methodology*, 1993, 109-127.

[9] J. Ferrante , K. Ottenstein, and J. Warren, “The program dependence graph and its use in optimization”, *ACM Transaction on Programming Languages and System*, 1987, 131-145.

[10] Rob Hierons, Mark Harman and Sebastian Danicic, “Using Program slicing to Assist in the Detection of Equivalent Mutants”, *Software Testing, Verification and Reliability*, 1999, 233-262.

[11] Rob Hierons, Mark Harman and Sebastian Danicic, “The Relationship between Program Dependence and Mutation Analysis”, *Mutation Testing for the new Century*, 2000, 5-13.

[12] W. E. Howden, “Weak Mutation Testing and Completeness of Test Sets”, *IEEE Transactions of Software Engineering,* 8(4), 1982, 371-379.

[13] S. C. Lee, A. J. Offutt, “Generating Test Cases for XML-Based Web Component Interactions Using Mutation Analysis”, *The Twelfth IEEE International Symposium on Software Reliability Engineering (ISSRE ‘01)*, 2001, 200-209.

[14] A. J. Offutt, “Investigations of the Software Testing Coupling Effect”, *ACM Transaction on Software Engineering and Methodology*, 1992, 5-20.

[15] A. J. Offutt, “A Practical System for Mutation Testing: Help for the Common Programmer”, *International Conference on Testing Computer Software*, 1995, 99-109.

[16] A. J. Offutt, G. Rothermel, R. H. Untch, and C. Zapf, “An Experimental Determination of Sufficient Mutant Operators”, *ACM Transaction of Software Engineering Methodology*, 1996, 99-118

[17] A. J. Offutt, J. Pan, “Automatically Detecting Equivalent Mutants and Infeasible Paths”, *The journal of Software Testing, Verification and Reliability*, 1997, 165-192.

[18] A. J. Offutt, Zhenyi Jin, Jie Pan, “The Dynamic Domain Reduction Procedure for Test Data Generation”, *Software Practice and Experience,* 1997, 167-193.

[19] K. Ottenstein, and L Ottenstein , “The program dependence graph in a software development environment”, *ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments* , 1984, 177-184.

[20] M. Weiser, “Program slicing”, *IEEE Transaction on Software Engineering*, 1984, 352-357.

ภาคผนวก ค

ผลงานตีพิมพ์ในงาน NCSEC 2003 ชิ้นที่ 1

การปรับปรุงการวิเคราะห์การผันแปรด้วยโปรแกรมผันแปรแบบหลายข้อผิดพลาด

An Improvement of Mutation Analysis by Multiple-Fault Mutant

วรท วรวัฒน์พิบูลย์ และ อรรถสิทธิ์ สุรฤกษ์
ภาควิชา วิศวกรรมคอมพิวเตอร์ จุฬาลงกรณ์มหาวิทยาลัย
ถนน พญาไท เขต ปทุมวัน กรุงเทพฯ 10330
warot.w@student.chula.ac.th, athasit@cp.eng.chula.ac.th

บทคัดย่อ

งานวิจัยชิ้นนี้เกี่ยวข้องกับการทดสอบโปรแกรมโดยอาศัยการวิเคราะห์การผันแปรที่มีประสิทธิผลในการทดสอบแบบหน่วยเดียว ข้อเสียคือใช้เวลาในการคำนวณที่สูง เนื่องจากจำนวนโปรแกรมผันแปรเป็นปัจจัยหลักที่ส่งผลกระทบต่อเวลาที่สูญเสียไปในการทดสอบโปรแกรม จุดมุ่งหมายสำหรับงานวิจัยชิ้นนี้คือ การลดจำนวนโปรแกรมผันแปรลง ทั้งยังคงระดับประสิทธิผลในการทดสอบ โดยเสนออัลกอริทึมในการสร้างโปรแกรมผันแปรแบบหลายข้อผิดพลาดที่อาศัยการใส่ข้อผิดพลาดต่างๆ ซึ่งแต่ละข้อผิดพลาดนั้นไม่ได้ส่งผลกระทบต่อผลลัพธ์ตัวเดียวกัน โดยนำเอาเทคนิคการตัดส่วนโปรแกรมมาวิเคราะห์ความเป็นอิสระของข้อผิดพลาด ในการทดลองชี้ให้เห็นว่าวิธีที่ปรับปรุงใหม่นี้สามารถลดจำนวนโปรแกรมผันแปรได้จริง

Abstract

This research concerns software testing using mutation analysis, a powerful technique of unit testing, but use high computation. The number of generated mutants can have a profound influence on the execution time and on the size of implementation. We focus on how the number of mutants can be reduced without effect to the effectiveness of the testing. We propose an algorithm for generating a multiple-fault mutant, made up from independent mutants described by the program slicing technique. A multiple-faults mutant is described by insertion faults that do not effect to the same output. An empirical study shows that our approach can save number of mutants.

Key-Word: *Software Engineering, Software Testing, Mutation analysis, Program slicing*

1. บทนำ

กระบวนการทดสอบโปรแกรมเป็นหัวใจสำคัญในการรับประกันความถูกต้องของโปรแกรม วัตถุประสงค์ของการทดสอบคือ การหาข้อผิดพลาดที่แฝงเร้นอยู่ ถึงอย่างไรก็ตามเวลาส่วนใหญ่สูญเสียไปในการทดสอบ ส่งผลให้มีหลายงานวิจัยนำเสนอเทคนิคและอัลกอริทึมเพื่อช่วยลดเวลาในการทดสอบลง ประสิทธิผลของการทดสอบนั้นขึ้นอยู่กับ*ชุดทดสอบ (Test data)* ที่ได้ ซึ่งชุดทดสอบใดๆ นั้น ประกอบขึ้นจากชุดของ*กรณีทดสอบ (a set of test cases)*

การวิเคราะห์การผันแปร (Mutation Analysis) [12] นั้นเป็นเทคนิคประเภทหนึ่งในหลักการของ*การทดสอบโดยการใส่ข้อผิดพลาด (Fault-based Testing)* ที่เป็น*การทดสอบแบบหน่วยเดียว (Unit Testing)* หลักการคือ พยายามเปลี่ยนแปลงโครงสร้างของโปรแกรมหรือใส่ข้อผิดพลาดเข้าไปใน*โปรแกรมทดสอบ (Testing program)* เพื่อสร้างเป็น*โปรแกรมผันแปร (Mutant programs) ต่างๆ* ซึ่งเปรียบเสมือนการจำลองข้อผิดพลาดที่อาจเกิดขึ้นกับโปรแกรมทดสอบ โดยรูปแบบของข้อผิดพลาดที่ใส่ไปนั้นถูกนิยามไว้ใน*ตัวดำเนินการผันแปร (Mutation Operator)* วัตถุประสงค์ของการวิเคราะห์การผันแปรคือ การสร้างชุดทดสอบที่มี*ประสิทธิผล (Effectiveness)* ในการหาข้อผิดพลาด หรือสามารถแยกความแตกต่างของ*ผลลัพธ์(Output)* ของการประมวลผลด้วยชุดทดสอบเดียวกันระหว่างโปรแกรมผันแปรและโปรแกรมทดสอบ การวิเคราะห์การผันแปรนั้นได้มีการนำไปประยุกต์ใช้ในการวิจัยอื่นๆ อีกเช่น [1, 2, 3, 10] การวิเคราะห์การผันแปรนั้นเป็นเทคนิคที่มีประสิทธิผล แต่ยังไม่ได้เป็นที่นิยมอย่างแพร่หลาย เนื่องมาจากเสียเวลามากในการประมวลผลแต่ละโปรแกรมผันแปร

งานวิจัยชิ้นนี้ได้เสนอวิธีการปรับปรุงการทดสอบด้วยวิธีการวิเคราะห์การผันแปรให้มี*ประสิทธิภาพ (Efficiency)* ดีขึ้น โดยได้จัดลำดับของแต่ละหัวข้อเป็นดังต่อไปนี้ หัวข้อที่ 2 เป็นนิยาม, สัญลักษณ์ และเทคนิคพื้นฐานต่างๆ ส่วนของกรรมวิธีในการปรับปรุงโดยอาศัยโปรแกรมผันแปรแบบหลายข้อผิดพลาดนั้นอยู่ในหัวข้อที่ 3 และ หัวข้อที่ 4 เป็นผลที่ได้จากการทดลอง ส่วนหัวข้อสุดท้ายเป็นการสรุปผล

2. บทนิยาม

ในหัวข้อนี้เป็นส่วนของนิยามต่างๆ ที่มีส่วนเกี่ยวข้องกับงานวิจัย ดังต่อไปนี้

2.1 การวิเคราะห์การผันแปร

หลักการวิเคราะห์การผันแปรนั้นเริ่มแรกได้ถูกเสนอโดย R. A. Demillo [4]

นิยามที่ 1

โปรแกรม m ถูกเรียกว่า โปรแกรมผันแปร *ที่เกิดจากโปรแกรม P หาก m เกิดจากการเปลี่ยนแปลงโครงสร้างของโปรแกรม P โดยรูปแบบของการเปลี่ยนแปลงนั้นได้รับการนิยามไว้ในตัวดำเนินการผันแปร*

ตัวดำเนินการผันแปรใช้สำหรับนิยามข้อผิดพลาดที่จะใส่ให้กับโปรแกรมทดสอบ เพื่อให้โปรแกรมผันแปรนั้นมีพฤติกรรมที่แตกต่างไปจากโปรแกรมทดสอบ ชนิดของตัวดำเนินการผันแปรถูกแสดงไว้ในงานวิจัย [13]

ในการทดสอบโปรแกรมด้วยวิธีการวิเคราะห์การผันแปรจำเป็นต้องมีการสร้างชุดทดสอบเพื่อแยกความแตกต่างของผลลัพธ์ที่เกิดจากโปรแกรมผันแปรและโปรแกรมทดสอบ

นิยามที่ 2

*โปรแกรมผันแปร m จะถูก*กำจัด(Kill) *ด้วยกรณีทดสอบ t ถ้า m(t) ซึ่งเป็นผลลัพธ์จากการประมวลผลโปรแกรมผันแปร m ด้วยกรณีทดสอบ t ให้ผลลัพธ์ที่ต่างจาก P(t) ซึ่งเป็นผลลัพธ์จากการประมวลผลโปรแกรมทดสอบด้วยกรณีทดสอบ t*

นิยามที่ 3

โปรแกรมผันแปรที่มีการทำงานเหมือนกับโปรแกรมทดสอบ ทำให้ไม่ว่าใช้ชุดทดสอบใดๆ มาทดสอบจะไม่สามารถแยกความแตกต่างของผลลัพธ์ได้ โปรแกรมผันแปรเหล่านี้ถูกเรียกว่า โปรแกรมผันแปรเทียบเท่า (Equivalence Mutant)

นิยามที่ 4

ชุดทดสอบ T ถูกเรียกว่า เพียงพอต่อการผันแปร (Mutation-adequate) *สำหรับโปรแกรมทดสอบ P ถ้าทุกโปรแกรมผันแปรถูกกำจัดด้วยกรณีทดสอบที่มีอยู่ในชุดทดสอบ T*

Howden [9] เสนอ *การวิเคราะห์การผันแปรแบบอ่อน (Weak Mutation)* โดย เงื่อนไขในการที่โปรแกรมผันแปรจะถูกกำจัด คือ ขอเพียงแค่คำสั่งที่มีการใส่ข้อผิดพลาดให้ผลลัพธ์ต่างกับคำสั่งเดียวกันในโปรแกรมทดสอบ แต่ประสิทธิผลของชุดทดสอบที่ได้รับจะลดลง

Offutt ได้เสนอตัวดำเนินการแบบ *E-Selective* ไว้ใน [13] โดยตัวดำเนินการในกลุ่มนี้จะนิยามเฉพาะข้อผิดพลาดที่ประกอบไปด้วย

6. UOI : Unary Operator Replacement
7. ROR: Relational Operator Replacement
8. AOR: Arithmetic Operator Replacement
9. LCR: Logical Connection Replacement
10. ABS: Absolute Value Insertion

2.2 กฎข้อบังคับในการกำจัดโปรแกรมผันแปร

การวัดประสิทธิผลของกรณีทดสอบใดๆ สามารถดูได้จากการกำจัดโปรแกรมผันแปร โดย [5] ได้เสนอ*กฎข้อบังคับ (Constraint)* ในการกำจัดโปรแกรมผันแปรที่อยู่ในรูปนิพจน์ทางคณิตศาสตร์ ซึ่งแบ่งออกเป็น 3 เงื่อนไขหลักด้วยกัน โดยในเงื่อนไขแรกคือ การที่พยายามทำให้คำสั่งที่ได้รับการใส่ข้อผิดพลาดได้รับการประมวลผล ซึ่งเรียกว่า *ข้อบังคับการไปถึง (Reachability Constraint)* ในเงื่อนไขที่สองคือ การให้ผลลัพธ์ของคำสั่งที่มีการใส่ข้อผิดพลาดแตกต่างไปจากคำสั่งเดียวกันในโปรแกรมทดสอบ ซึ่งเรียกว่า *ข้อบังคับที่จำเป็น (Necessity Constraint)* ในเงื่อนไขข้อสุดท้าย เรียกว่า*ข้อบังคับที่เพียงพอ (Sufficient Constraint)* ที่กล่าวว่าผลลัพธ์สุดท้ายของโปรแกรมผันแปรและโปรแกรมทดสอบต้องแตกต่างกัน

ตัวอย่างที่ 1 *กำหนดให้ส่วนของโปรแกรมเป็นดังต่อไปนี้*

```
001  …
00n  if (x+c>=y) …
999  …
```

สมมุติโปรแกรมผันแปรที่เกิดจากการเปลี่ยนค่า “x” เป็น ค่า “4” ดังนั้นข้อบังคับที่จำเป็นสำหรับโปรแกรมผันแปรนี้คือ ($x \neq 4$) หากเรานำกรณีทดสอบ t (x=8, y=10, c=8) ซึ่งรองรับข้อบังคับที่จำเป็นมาทดสอบจะเห็นได้ว่าผลลัพธ์ของเงื่อนไขไม่ต่างไปจากเดิม คือ 8+c และ 4+c

ต่างทำให้เงื่อนไขเป็นจริง ดังนั้นจึงต้องนำข้อบังคับที่เพียงพอมากำหนดเพิ่มเป็น (x+c>=y) ≠ (4+c>=y) สังเกตได้ว่าตัวทดสอบเดิมนั้นไม่รองรับข้อบังคับที่เพียงพอ

กฎข้อบังคับ ที่กล่าวมานี้ได้นำไปประยุกต์ใช้ในงานด้านต่างๆ เช่น การสร้างชุดทดสอบที่เรียกว่า *การทดสอบโดยอิงข้อบังคับ (Constraint-Based Testing)* [5, 6] ที่สร้างชุดทดสอบที่รองรับข้อบังคับเพื่อให้สามารถกำจัดโปรแกรมผันแปรได้ ในงาน [14] เป็นการนำเอากฎข้อบังคับไปตรวจจับโปรแกรมผันแปรเทียบเท่าแบบอัตโนมัติ

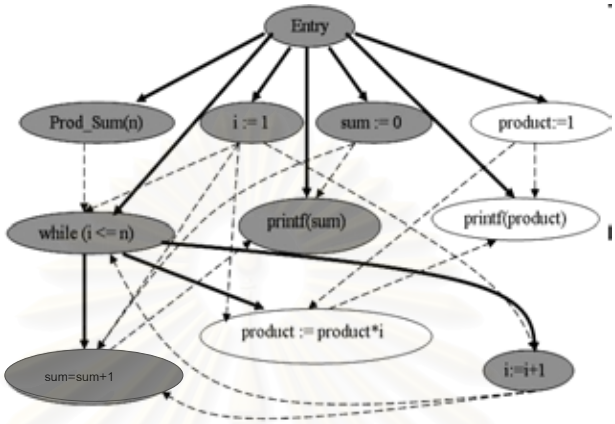
2.3 กรรมวิธีการตัดส่วนโปรแกรม

กรรมวิธีการตัดส่วนโปรแกรม *(Program Slicing)* ถูกเสนอโดย Weiser [16] ได้ประยุกต์ใช้ในการวิเคราะห์เพื่อทำความเข้าใจโปรแกรมให้ง่ายขึ้น *(Program Comprehension)* ทั้งยังในเรื่องของการค้นหาตำแหน่งข้อผิดพลาด *(Debugging)* วัตถุประสงค์หลักคือการตัดโปรแกรมเอาเฉพาะส่วนโปรแกรมที่เกี่ยวข้องกับจุดที่สนใจ โดยจุดที่สนใจนั้นได้นิยามไว้ใน บรรทัดฐานการตัด *(Slicing Criterion)* ที่อยู่ในรูปของคู่ลำดับ (V,n) โดย V เป็นชุดของตัวแปรที่สนใจ ส่วน n เป็นตำแหน่งของคำสั่งในโปรแกรม Ottenstein [15] ได้เสนออัลกอริทึมในการตัดส่วนโปรแกรมโดยอาศัยการท่องไปตามบัพต่างๆ ในกราฟการขึ้นต่อกัน *(Program Dependence Graph)* โดยจุดยอดของกราฟแสดงคำสั่งในโปรแกรม ส่วนเส้นที่เชื่อมจุดยอดแสดงถึง การขึ้นต่อกันแบบตัวแปร *(Data Dependence)* และการขึ้นต่อกันแบบควบคุม *(Control Dependence)* ในงานวิจัยชิ้นนี้ได้นำเอาวิธีการตัดส่วนโปรแกรมแบบกราฟการขึ้นต่อกันนี้มาประยุกต์ใช้ โดยนิยามบรรทัดฐานการตัดไว้ที่ผลลัพธ์สุดท้ายของโปรแกรมทดสอบ

ตัวอย่างที่ 2 *กำหนดให้โปรแกรม Prod_Sum(int n).*

```
001  void Prod_Sum(int n)
002  {
003          i = 1;
004          sum = 0;
005          product = 1;
006          while (i<=n)
007          {
008                  sum = sum + 1;
009                  product = product * i;
010                  i = i + 1;
011          }
012          printf("%d", sum);
```

กราฟการขึ้นต่อกันของโปรแกรมข้างต้นนี้ถูกแสดงไว้ในรูปที่ 1 โดยเส้นสีเข้มนั้นแสดงถึงการขึ้นต่อกันแบบควบคุม ส่วนเส้นประนั้นแสดงถึงการขึ้นต่อกันแบบตัวแปร ส่วนจุดยอดที่มีสีเข้มนั้นแสดงถึงคำสั่งที่ส่งผลกระทบต่อตัวแปรในบรรทัดฐานการตัดซึ่งในที่นี้คือ ({sum},012)



รูปที่ 1 แสดงกราฟการขึ้นต่อกันของโปรแกรม Prod_Sum(int n)

โปรแกรมที่ได้รับการตัดส่วนที่ไม่เกี่ยวข้องออกไปเป็นดังนี้

```
001  void Prod_Sum(int n)
002  {
003          i = 1;
004          sum = 0;
006          while (i<=n)
007          {
008                  sum = sum + 1;
010                  i = i + 1;
011          }
012          printf("%d", sum);
014  }
```

3. โปรแกรมผันแปรแบบหลายข้อผิดพลาด

Demillo [4, 11] ได้เสนอทฤษฎี คอปปลิงเอฟเฟ็ค (Coupling Effect Theorem) ซึ่งกล่าวว่า ชุดทดสอบใดๆ ก็ตามที่สามารถหาข้อผิดพลาดที่มีอยู่ในโปรแกรมเพียงที่เดียวได้แล้วจะมีความสามารถเพียงพอในการหาข้อผิดพลาดที่มีหลายที่ได้ โดยการวิเคราะห์การผันแปรนั้นอาศัยทฤษฎีข้อนี้ ทำให้แต่ละโปรแกรมผันแปรนั้นเกิดจากข้อผิดพลาดเพียงที่เดียวและไม่สนใจโปรแกรมผันแปรที่เกิดจากหลายข้อผิดพลาด ถึงอย่างไรก็ตามโปรแกรมผันแปรที่เกิดจากข้อผิดพลาดเดียว

ยังคงมีจำนวนมากอยู่ ดังนั้นจึงเป็นประเด็นให้หลายงานวิจัยมุ่งเน้นในการลดจำนวนโปรแกรมผันแปรลง แต่ส่วนใหญ่แล้วประสิทธิผลในการทดสอบลดลงตามไปด้วย

ในงานวิจัยที่ผ่านมาที่เกี่ยวข้องกับการนำเอาการตัดส่วนโปรแกรมมาช่วยในการวิเคราะห์การผันแปรนั้นมี [7] ซึ่งเป็นการนำเอาการตัดส่วนโปรแกรมมาช่วยตรวจจับโปรแกรมผันแปรเทียบเท่า และใน [8] เป็นการลดจำนวนกรณีทดสอบที่ต้องนำมาทดสอบแต่ละโปรแกรมผันแปรลง

งานวิจัยชิ้นนี้ได้นำเอาการตัดส่วนโปรแกรมมาวิเคราะห์ว่าคำสั่งใดบ้างที่ส่งผลกระทบต่อบรรทัดฐานการตัด โดยตัวแปรในบรรทัดฐานการตัดคือ แต่ละผลลัพธ์ของโปรแกรม เมื่อทราบว่าคำสั่งใดกระทบต่อผลลัพธ์ที่สนใจแล้ว จะสามารถแบ่งโปรแกรมผันแปรออกเป็นกลุ่มตามแต่ละผลลัพธ์ของโปรแกรม โดยโปรแกรมผันแปรในแต่ละกลุ่มจะไม่ส่งผลกระทบต่อผลลัพธ์ตัวอื่นๆ

3.1 นิยามของโปรแกรมผันแปรแบบหลายข้อผิดพลาด

โปรแกรมผันแปรแบบหลายข้อผิดพลาดนี้เกิดจากการใส่ข้อผิดพลาดในหลายตำแหน่ง เพื่อให้โปรแกรมเพียงโปรแกรมเดียวสามารถเป็นตัวแทนของหลายโปรแกรมผันแปร โดยได้มีการนิยามโปรแกรมผันแปรแบบหลายข้อผิดพลาดไว้ดังต่อไปนี้

นิยามที่ 5

กำหนดให้ P เป็นโปรแกรมทดสอบที่ประกอบด้วยคำสั่ง $s_1, s_2, s_3, \ldots, s_n$ และ C เป็นโปรแกรมผันแปรแบบหลายข้อผิดพลาด ถ้ามีจำนวนเต็มบวก k ใดๆ ที่

$$P \underset{s_1}{\overset{u_1}{\mapsto}} M_1 \underset{s_2}{\overset{u_2}{\mapsto}} M_2 \underset{s_3}{\overset{u_3}{\mapsto}} M_3 \ldots \underset{s_k}{\overset{u_k}{\mapsto}} M_k = C$$

โดย $P \underset{s_j}{\overset{u_j}{\mapsto}} M$ เป็นการใส่ข้อผิดพลาด u_j ที่คำสั่ง s_j

อัลกอริทึมที่ใช้สร้างโปรแกรมผันแปรแบบหลายข้อผิดพลาด

อัลกอริทึมที่ใช้ในการสร้างโปรแกรมผันแปรแบบหลายข้อผิดพลาด กำหนดให้โปรแกรมทดสอบ P มี n ผลลัพธ์ คือ $o_1, o_2, o_3, \ldots, o_n$

ขั้นตอนที่ 1 ทำการตัดส่วนของโปรแกรม P ออกเป็น n โปรแกรมย่อย ซึ่งเรียกว่า $P_1, P_2, P_3, \ldots, P_n$ โดยที่แต่ละ P_j ที่ $1 \le j \le n$ นั้นประกอบด้วยคำสั่งที่สามารถส่งผลกระทบต่อ o_j

สำหรับแต่ละ P_i และ P_j ที่ $i \ne j$ นั้น ให้ตัดคำสั่งที่เป็นคำสั่งร่วมกันซึ่งส่งผลกระทบต่อทั้ง o_i และ o_j จาก P_i และ P_j

ขั้นตอนที่ 2 สำหรับแต่ละ P_j ให้สร้างชุดของโปรแกรมผันแปร SM_j ที่เกิดจากการใส่ข้อผิดพลาดเพียงที่เดียวให้กับโปรแกรมทดสอบ P_j

ขั้นตอนที่ 3 สร้างโปรแกรมผันแปรแบบหลายข้อผิดพลาดตามอัลกอริทึมด้านล่างนี้ โดยกำหนดให้ $|SM_j|$ คือ จำนวนโปรแกรมผันแปรใน SM_j

for i=1 to max(|SM$_1$|,|SM$_2$|,...,|SM$_n$|) do
M= P
for j=1 to n do
if |SM$_j$|<> 0 then
M' = insert u$_j$ at statement s$_j$ to M
SM$_j$=SM$_j$\{u$_j$}
endif;
endfor;
C$_i$ = M
endfor;
output Mutant$_1$ = {C$_1$,C$_2$,..., C$_n$}

ขั้นตอนที่ 4 รวมโปรแกรมผันแปรทั้งหมด ซึ่งเกิดจากคำสั่งร่วมกัน(ที่ตัดออกในขั้นตอนที่ 1) *Mutant$_2$*={*M$_1$, M$_2$, ...,M$_k$*} เข้ากับ *Mutant$_1$*

ผลลัพธ์จากอัลกอริทึม

ผลลัพธ์ที่ได้ คือ ชุดของโปรแกรมผันแปรแบบหลายข้อผิดพลาดที่เกิดจากการใส่ข้อผิดพลาดที่เป็นอิสระต่อกันไว้

ข้อผิดพลาดที่นำมาสร้างเป็นโปรแกรมผันแปรแบบหลายข้อผิดพลาดนี้เป็นอิสระต่อกัน เนื่องมาจากการใช้วิธีการตัดส่วนโปรแกรมมาวิเคราะห์ว่าคำสั่งใดส่งผลกระทบต่อผลลัพธ์ตัวใดบ้าง ข้อผิดพลาดที่ใส่ให้คำสั่งที่ส่งผลกระทบต่อผลลัพธ์ที่ต่างกันจะเป็นอิสระจากกันและตัวดำเนินการผันแปรที่เลือกมาใช้เป็นแบบ E-Selective จึงไม่สามารถสร้างข้อผิดพลาดที่ทำให้คำสั่งเดิมไปส่งผลต่อผลลัพธ์ตัวอื่นได้

ตัวอย่างที่ 4 *กำหนดให้โปรแกรม Prod_Sum(int n) ถูกตัดส่วนออกเป็นสองโปรแกรมตามบรรทัดฐานการตัดคือ S$_s$=({sum},012)(ในตัวอย่างที่ 2) และ S$_p$=({product,013)ซึ่งแสดงไว้ดังต่อไปนี้*

```
001         void Prod_Sum(int n)
002         {
003                 i = 1;
005                 product = 1;
006                 while (i<=n) do
007                 {
009                 product = product*i;
010                 i = i + 1;
```

```
011                 }
013                 printf("%d",product);
014         }
```

ทั้ง S_s และ S_p มีคำสั่งที่ร่วมกันคือคำสั่งในแถวที่ 003, 006 และ 010 ข้อผิดพลาดที่ใส่ให้คำสั่งที่ไม่ใช่คำสั่งร่วมกันของทั้ง S_s และ S_p นั้นสามารถรวมไว้ในโปรแกรมเดียวกันได้ เพื่อสร้างเป็นโปรแกรมผันแปรแบบหลายข้อผิดลพาด ดังต่อไปนี้

ตารางที่ 1 แสดงข้อผิดพลาดของโปรแกรมผันแปรต่างๆ

โปรแกรมผันแปร	**บรรทัด**	**ต้นแบบ**	**ข้อผิดพลาด**
M_1	004	sum = 0	sum = 5
M_2	005	product = 1	product = 5
C_1(Multi-Fault)	004	sum = 0	sum = 5
	005	product = 1	product = 5

สังเกตได้ว่าโปรแกรมผันแปรแบบหลายข้อผิดพลาด (C_1) เกิดจากข้อผิดพลาดของทั้ง M_1 และ M_2

3.2 การกำจัดโปรแกรมผันแปรแบบหลายข้อผิดพลาด

กรณีทดสอบถูกใช้สำหรับการประมวลผลโปรแกรมผันแปร ถ้าผลลัพธ์สุดท้ายที่ได้รับแตกต่างจาก*ผลลัพธ์ที่คาดหวัง (Expected Output)* โปรแกรมผันแปรตัวนั้นจะถูกกำจัด เนื่องจากโปรแกรมผันแปรแบบหลายข้อผิดพลาดเกิดจากการใส่ข้อผิดพลาดเข้าไปมากกว่าหนึ่งข้อผิดพลาด ในการกำจัดโปรแกรมผันแปรแบบหลายข้อผิดพลาดนี้จำเป็นต้องกำจัดทุกโปรแกรมผันแปรที่ประกอบขึ้นเป็นโปรแกรมผันแปรแบบหลายข้อผิดพลาด การที่โปรแกรมผันแปรแบบหลายข้อผิดพลาดจะถูกกำจัดได้ด้วยกรณีทดสอบเดียวก็ต่อเมื่อ กฎข้อบังคับของแต่ละโปรแกรมผันแปรที่ประกอบกันขึ้นเป็นโปรแกรมผันแปรแบบหลายข้อผิดพลาดนั้นไม่มีข้อขัดแย้งระหว่างกัน

นิยามที่ 6

กำหนดให้ P เป็นโปรแกรมทดสอบที่ประกอบด้วย n ผลลัพธ์ สำหรับแต่ละจำนวนเต็มบวก k ใดๆ ที่ $1 \leq j \leq k$

$$P \underset{s_j}{\overset{u_j}{\mapsto}} N_j$$

และมีกรณีทดสอบ t ใดๆ ที่ โปรแกรมผันแปร N_j ถูกประมวลผลด้วยกรณีทดสอบ t ($N_j(t)$) แล้วให้ผลลัพธ์ที่ต่างจาก P(t) เฉพาะผลลัพธ์ตัวที่ j เท่านั้น โดยโปรแกรมผันแปรแบบหลายข้อผิดพลาด C ใดๆ ที่

$$P \underset{s_1}{\overset{u_1}{\mapsto}} M_1 \underset{s_2}{\overset{u_2}{\mapsto}} M_2 \underset{s_3}{\overset{u_3}{\mapsto}} M_3 \ldots \underset{s_k}{\overset{u_k}{\mapsto}} M_k = C$$

จะถูกกำจัดด้วยชุดของกรณีทดสอบ $T= \{t_1, t_2, \ldots, t_m\}$ ถ้าทุกๆ j ใดๆ ที่ $1 \leq j \leq k$ จะมี t_i ใดๆ ที่เมื่อ $C(t_i)$ ให้ผลลัพธ์ที่แตกต่างจาก $P(t_i)$ ณ ผลลัพธ์ j

จากตัวอย่างที่ผ่านมา หากกำหนดให้กรณีทดสอบ t = (n=5) แล้วผลลัพธ์สุดท้ายของโปรแกรมผันแปรต่างๆ เป็นดังตารางต่อไปนี้

ตารางที่ 2 ผลลัพธ์สุดท้ายของแต่ละโปรแกรมผันแปร

โปรแกรม เมื่อ n=5	sum	product
P	15	120
M_1	20	120
M_2	15	600
Multiple-faults mutant C	20	600

สังเกตเห็นว่าข้อผิดพลาดของทั้ง M_1 และ M_2 สามารถถูกแทนด้วยโปรแกรมผันแปรแบบหลายข้อผิดพลาด C ซึ่งสามารถกำจัดได้ด้วยกรณีทดสอบเพียงครั้งเดียว ดังนั้นเราสามารถลดเวลาที่สูญเสียไปได้ดังในตัวอย่าง

4. การทดลอง

ในงานวิจัยชิ้นนี้ได้ทำการทดลองโดยเลือกเอาโปรแกรมภาษาปาสคาลมาเป็นโปรแกรมทดสอบ ซึ่งทุกๆ โปรแกรมจะประกอบด้วยผลลัพธ์ที่มากกว่าหนึ่งผลลัพธ์ และเนื่องจากขนาดของโปรแกรมไม่ได้เป็นปัจจัยของการลดจำนวนโปรแกรมผันแปร ในที่นี้จึงขอยกตัวอย่างโปรแกรมที่สะดวกต่อการทำความเข้าใจดังต่อไปนี้

ตารางที่ 3 โปรแกรมที่นำมาทดสอบ

โปรแกรม	**คำบรรยาย**
SinCos	คำนวณค่า Sin และ Cos ตามสูตรของเทย์เลอร์
MulSumMatrix	คำนวณค่าการคูณและบวกเมตริกซ์
MulSumFraction	คำนวณค่าการคูณและบวกเศษส่วน
Mean	คำนวณค่า Harmonic, Geometric และ Statistic Mean
MaxMinAvg	หาค่า max, min และ average ของจำนวน 5 ค่า

ตัวดำเนินการผันแปรที่ใช้เป็นแบบ E-Selective เพื่อสร้างเป็นโปรแกรมผันแปรและโปรแกรมผันแปรแบบหลายข้อผิดพลาด โดยในตารางที่ 4 แสดงจำนวนโปรแกรมผันแปรที่ได้รับ ซึ่งในคอลัมน์เปอร์เซ็นต์ที่ลดลง *(Percentage Saved)* เป็นเปอร์เซ็นต์ที่ลดลงเมื่อเทียบกับวิธีเดิม

ตารางที่ 4 ผลลัพธ์ที่ได้จากการทดลอง

Program	Primitive Mutants	Multiple-faults mutants	Percentage Saved
SinCos	192	101	47.40
MulSumMatrix	151	117	22.52
MulSumFraction	295	162	45.08
Mean	86	65	24.42
MaxMinAvg	89	53	40.45
Average			35.97

จากผลการทดลองชี้ให้เห็นว่าจำนวนของโปรแกรมผันแปรสามารถลดลงได้ถึง 47.40% ในโปรแกรม SinCos ซึ่งค่าเฉลี่ยการลดลงของโปรแกรมทั้งหมดเป็น 35.97% โดยเปอร์เซ็นต์การลดลงนั้นแปรผันโดยตรงกับความเป็นอิสระกันของแต่ละผลลัพธ์ ถ้าความเป็นอิสระของแต่ละผลลัพธ์มากเปอร์เซ็นต์การลดลงจะมากตามไปด้วย ตัวอย่างเช่นในโปรแกรม Mean มีเพียง 1 คำสั่งเท่านั้นที่ไม่เป็นคำสั่งร่วมกัน ทำให้โปรแกรม Mean มีเปอร์เซ็นต์การลดลงเพียง 24.42%

จำนวนโปรแกรมที่ได้รับการตัดส่วน (P_i) เป็นอีกปัจจัยที่ส่งผลต่อการลดลงของโปรแกรมผันแปร ถ้าจำนวนโปรแกรมที่ได้รับการตัดส่วนมีมาก โอกาสที่โปรแกรมผันแปรจำนวนมากจะถูกรวมเป็นโปรแกรมผันแปรตัวเดียวมีมากขึ้น

5. สรุป

งานวิจัยชิ้นนี้ได้นำเสนอกรรมวิธีในการปรับปรุงการทดสอบโปรแกรมด้วยการวิเคราะห์การผันแปร โดยอาศัยหลักการในการสร้างโปรแกรมผันแปรที่สามารถเป็นตัวแทนของโปรแกรมผันแปรหลายๆ โปรแกรม ซึ่งการวิเคราะห์ความเป็นอิสระจากกันของแต่ละคำสั่งในโปรแกรมนั้นอาศัยหลักการของการตัดส่วนของโปรแกรม โดยที่จำนวนส่วนของโปรแกรมที่ได้รับการตัดส่วนแล้วจะเท่ากับจำนวนผลลัพธ์ของโปรแกรม งานวิจัยชิ้นนี้สามารถรักษาประสิทธิผลในการวิเคราะห์การผันแปรเอาไว้ซึ่งต่างจากวิธีการปรับปรุงหลายๆ วิธีที่ผ่านมา แนวทางในการพัฒนาต่อไปในอนาคตคือการสร้างโปรแกรมผันแปรแบบหลายข้อผิดพลาดที่สามารถถูกกำจัดได้ด้วยกรณีทดสอบเดียว

6. เอกสารอ้างอิง

[1] James M. Bieman, Sudipto Ghosh, Roger T. Alexander, “A Technique for Mutation of Java Objects”, *Proceedings Automated Software Engineering Conference (ASE 2001)*, 2001.

[2] Paul E. Black, V. Okun, Yaacov Yesha, “Mutation Operators for specification”, *IEEE International Conference on Automated Software Engineering*, 2000.

[3] Philippe Chevalley, “Applying Mutation Analysis for Object-Oriented Program Using a Reflective Approach”, *Asia-Pacific Software Engineering Conference*, 2001.

[4] R. A. Demillo, R. J. Lipton, and F.G. Sayward, “Hints on Test Data Selection: Help for the Practicing Programmer”, *IEEE Computer*, 1978, pp. 34-41.

[5] R. A. Demillo , A. Jefferson Offutt , “Constraint-Based Automatic Test Data Generation”, *IEEE Transaction on Software Engineering* , 1991.

[6] R. A. Demillo , A. Jefferson Offutt , “Experimental Result from an Automatic Test Case Generator”, *ACM Transaction of Software Engineering and Methodology*, 1993, pp. 109-127.

[7] Rob Hierons, Mark Harman and Sebastian Danicic, “Using Program slicing to Assist in the Detection of Equivalent Mutants”, *Software Testing, Verification and Reliability*, 1999, pp. 233-262.

[8] Rob Hierons, Mark Harman and Sebastian Danicic, “The Relationship between Program Dependence and Mutation Analysis”, *Mutation Testing for the new Century*, 2000, pp. 5-13.

[9] W. E. Howden, “Weak Mutation Testing and Completeness of Test Sets”, *IEEE Transactions of Software Engineering*, 8(4), 1982, pp. 371-379.

[10] Suet Chun Lee, Jeff Offutt, “Generating Test Cases for XML-Based Web Component Interactions Using Mutation Analysis”, *The Twelfth IEEE International Symposium on Software Reliability Engineering (ISSRE ‘01),* 2001, pp. 200-209.

[11] A. J. Offutt, “Investigations of the Software Testing Coupling Effect”, *ACM Transaction on Software Engineering and Methodology*, 1992, pp. 5-20.

[12] A. J. Offutt, “A Practical System for Mutation Testing: Help for the Common Programmer”, *International Conference on Testing Computer Software*, 1995, pp. 99-109.

[13] A. J. Offutt, G. Rothermel, R. H. Untch, and C. Zapf, “An Experimental Determination of Sufficient Mutant Operators”, *ACM Transaction of Software Engineering Methodology*, 1996, pp. 99-118

[14] A. J. Offutt, and J. Pan, “Automatically Detecting Equivalent Mutants and Infeasible Paths”, *The journal of Software Testing, Verification and Reliability*, 1997, pp. 165-192.

[15] K. Ottenstein, and L Ottenstein , “The program dependence graph in a software development environment”, *ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments* , 1984, pp.177-184.

[16] M. Weiser, “Program slicing”, *IEEE Transaction on Software Engineering*, 1984, pp. 352-357.

ภาคผนวก ง

ผลงานตีพิมพ์ในงาน NCSEC 2003 ชิ้นที่ 2

การลดเวลาในการวิเคราะห์การผันแปรด้วยวิธีการแบ่งกลุ่ม

Time Reducing in Mutation Analysis by Mutant Classification

วรท วรวัฒน์พิบูลย์ และ อรรถสิทธิ์ สุรฤกษ์
ภาควิชา วิศวกรรมคอมพิวเตอร์ จุฬาลงกรณ์มหาวิทยาลัย
ถนน พญาไท เขต ปทุมวัน กรุงเทพฯ 10330
warot.w@student.chula.ac.th, athasit@cp.eng.chula.ac.th

บทคัดย่อ

งานวิจัยชิ้นนี้นำเสนอแนวทางในการปรับปรุงการทดสอบโปรแกรมด้วยกรรมวิธีการวิเคราะห์การผันแปร ซึ่งมุ่งประเด็นในเรื่องของการลดเวลาที่เสียไปในการทดสอบโปรแกรมผันแปรลง หลักการที่ถูกนำมาใช้เพื่อลดเวลาการทำงานคือ การแบ่งโปรแกรมผันแปรออกเป็นกลุ่ม โดยอาศัยหลักการในเรื่องของข้อบังคับในการที่โปรแกรมผันแปรหนึ่งๆ จะถูกกำจัดหรือเปิดเผยข้อผิดพลาดมาเป็นเกณฑ์ในการแบ่ง ซึ่งวิธีการนี้สามารถพิสูจน์ได้ว่า ไม่ส่งผลกระทบต่อประสิทธิผลในการทำการวิเคราะห์การผันแปรเช่นวิธีที่ผ่านมา

Abstract

This research proposes an approach of testing software improvement using mutation analysis. We focus on how to reduce the execution time of the analysis. The used time reducing technique is separation mutants into several groups classified by killing constraints. We prove that our approach does not effect on the effectiveness of the testing.

Key-Word: *Software Engineering, Software Testing, Mutation analysis, Constraint-based testing*

1. บทนำ

กระบวนการทดสอบโปรแกรมถือเป็นกระบวนการที่มีความสำคัญในการพัฒนาโปรแกรม เนื่องจากเวลาส่วนใหญ่สูญเสียไปในการทดสอบโปรแกรม จึงเป็นประเด็นให้งานวิจัยส่วนใหญ่คิดค้นเทคนิคและกรรมวิธีที่ช่วยลดความสิ้นเปลืองต่างๆ ในส่วนของการทดสอบโปรแกรมลง

ประสิทธิผลของการทดสอบโปรแกรมนั้นขึ้นอยู่กับการสร้าง *ชุดทดสอบ (Test data)* ซึ่งประกอบด้วย*กรณีทดสอบต่างๆ (A set of test cases)* เพื่อให้การทดสอบนั้นเป็นไปตาม*บรรทัดฐานของการทดสอบ (Testing criterion)*

การทดสอบโปรแกรมด้วยวิธีใส่ข้อผิดพลาด(Fault-based testing) เป็นเทคนิคหนึ่งที่ใช้สร้างชุดทดสอบให้สามารถหาข้อผิดพลาดที่มีอยู่ในโปรแกรม โดยอาศัยหลักการจำลองข้อผิดพลาดไปใส่ในโปรแกรมที่ต้องการทดสอบโดยมีบรรทัดฐานของการทดสอบคือ การให้ชุดทดสอบสามารถหาข้อผิดพลาดที่ใส่เข้าไปนั้นให้ได้

การวิเคราะห์การผันแปร (Mutation Analysis) [10] เป็นเทคนิคประเภทหนึ่งที่จัดอยู่ในกลุ่มของการทดสอบโปรแกรมด้วยการใส่ข้อผิดพลาด หลักการของการวิเคราะห์การผันแปรคือ ใส่ข้อผิดพลาดที่ได้รับมาจาก*ตัวดำเนินการผันแปร (Mutation Operator)* ให้กับโปรแกรมที่ต้องการทดสอบเพื่อสร้างเป็นโปรแกรมใหม่ที่เรียกว่า *โปรแกรมผันแปร (Mutant Program)* โดยแต่ละโปรแกรมผันแปรจะแสดงตัวเสมือนเป็นโปรแกรมที่มีข้อผิดพลาดอยู่ ดังนั้นเป้าหมายของผู้ทดสอบคือ การสร้างชุดทดสอบขึ้นมาเพื่อที่จะแยกความแตกต่างของผลลัพธ์ที่ได้จากการทำงานของโปรแกรมผันแปรและ โปรแกรมทดสอบ การวิเคราะห์กระบวนการผันแปรนั้นมีการนำไปประยุกต์ใช้ในงานด้านต่างๆ [1, 2 , 3, 8]

ปัจจัยสำคัญที่ทำให้การวิเคราะห์การผันแปรนั้นยังมิได้เป็นที่นิยมในการนำมาประยุกต์ใช้ในการพัฒนาโปรแกรม คือ จำนวนของโปรแกรมผันแปรที่มาก ส่งผลให้เวลาส่วนใหญ่ที่ใช้ในการทำงานเสียไปกับการประมวลผลโปรแกรมผันแปร

งานวิจัยที่ผ่านมาที่มุ่งเน้นในเรื่องของการลดเวลาที่เสียไปนั้นส่วนใหญ่อาศัยหลักการที่ว่าหากจำนวนโปรแกรมผันแปรลดลงแล้วเวลาที่ใช้ในการประมวลผลย่อมลดลงด้วย ผลที่ตามมาคือประสิทธิผลของชุดทดสอบที่ได้ลดลงเพราะไม่ครอบคลุมทุกกรณีที่อาจเกิดขึ้นจากโปรแกรมผันแปร

ในงานวิจัยนี้ได้นำเสนอกรรมวิธีใหม่ที่ช่วยลดเวลาที่ใช้ในการทดสอบโปรแกรมผันแปรลงพร้อมทั้งคงระดับประสิทธิผลของชุดทดสอบไว้ โดยได้มีการจัดลำดับหัวข้อดังต่อไปนี้ หัวข้อที่ 2 เป็นการแสดง นิยาม ,สัญลักษณ์ และเทคนิคพื้นฐานที่นำมาใช้ ส่วนของการวิเคราะห์การผันแปรโดยอาศัยหลักการแบ่งกลุ่มนั้นบรรยายไว้ในหัวข้อที่ 3 และหัวข้อที่ 4 เป็นตัวอย่างของการ

แบ่งกลุ่มการวิเคราะห์การผันแปรพร้อมทั้งแสดงเวลาที่ลดลงในการทำงาน ในหัวข้อสุดท้ายเป็นส่วนของการสรุปผล

2. บทนิยาม

ในหัวข้อนี้เป็นส่วนของการนิยามต่างๆ ที่นำมาใช้ในงานนี้ โดยจะเริ่มจากนิยามของการวิเคราะห์การผันแปร และการสร้างชุดทดสอบ

2.1 การวิเคราะห์การผันแปร

หลักการของการวิเคราะห์การผันแปรนั้นได้ถูกเสนอโดย R.A. Demillo ในปี 1978 [4] ซึ่งอาศัยการใส่ข้อผิดพลาดที่ถูกนิยามไว้ในตัวดำเนินการผันแปรเข้าไปในโปรแกรม ตัวอย่างของตัวดำเนินการผันแปรเช่น การแทนตัวดำเนินการในการคำนวณด้วยตัวดำเนินการในการคำนวณตัวอื่น (Arithmetic Operator Replacement) เช่น การแทน x = x + 5 เป็น x = x - 5 เป็นต้น ชนิดของตัวดำเนินการผันแปรต่างๆได้แสดงไว้ในงานวิจัย [7]

นิยามที่ 1
โปรแกรม m จะถูกเรียกว่า โปรแกรมผันแปร *ของโปรแกรม P ถ้า m เกิดจากการเปลี่ยนแปลงโครงสร้างของโปรแกรม P โดยรูปแบบของการเปลี่ยนเปลี่ยนแปลงนั้นได้รับการนิยามไว้ในส่วนของตัวดำเนินการผันแปร*

ในการทดสอบโปรแกรมด้วยวิธีวิเคราะห์การผันแปรนั้นต้องมีการสร้างชุดทดสอบเพื่อแยกความแตกต่างของผลลัพธ์ที่เกิดจากโปรแกรมผันแปรและโปรแกรมทดสอบ

นิยามที่ 2
โปรแกรมผันแปร m จะถูก กำจัด(Kill) *ด้วยกรณีทดสอบ(test case) t ถ้า m(t) ซึ่งเป็นผลลัพธ์จากการประมวลผลโปรแกรมผันแปร m ด้วยกรณีทดสอบ t ให้ผลลัพธ์ที่แตกต่างจาก P(t) ซึ่งเป็นผลลัพธ์จากการประมวลผลโปรแกรมทดสอบด้วยกรณีทดสอบ t*

นิยามที่ 3

โปรแกรมผันแปรที่มีการทำงานเหมือนกับโปรแกรมทดสอบ ทำให้ไม่ว่าจะใช้ชุดทดสอบใดๆ ก็ตามมาทดสอบจะไม่สามารถแยกความแตกต่างของผลลัพธ์ได้ โปรแกรมผันแปรเหล่านี้ถูกเรียกว่า โปรแกรมผันแปรเทียบเท่า (Equivalence Mutant)

นิยามที่ 4

ชุดทดสอบ T ถูกเรียกว่า เพียงพอต่อการผันแปร (Mutation-adequate) *สำหรับโปรแกรมทดสอบ P ถ้าทุกโปรแกรมผันแปรถูกกำจัดด้วยกรณีทดสอบที่อยู่ในชุดทดสอบ T*

ประสิทธิผลของชุดทดสอบสามารถแสดงออกมาอยู่ในรูปของคะแนนความผันแปรซึ่งมีการนิยามไว้ดังนี้

$$MS\ (P,T)\ =\ \frac{K\ \times 100}{M\ -\ E}$$

โดยที่ MS คือ คะแนนความผันแปร

P คือ โปรแกรมทดสอบ

T คือ ชุดทดสอบ

K คือ จำนวนโปรแกรมผันแปรที่ถูกกำจัด

M คือ จำนวนโปรแกรมผันแปรทั้งหมด

E คือ จำนวนโปรแกรมผันแปรเทียบเท่า

ชุดทดสอบ T เพียงพอต่อการผันแปรหากมีคะแนนความผันแปรเป็น 100 % เนื่องจากการวิเคราะห์การผันแปรมีข้อเสียในด้านของเวลาที่สูญเสียไป Howden [6] ได้เสนอวิธีการปรับปรุงที่มีชื่อว่า *การวิเคราะห์การผันแปรแบบอ่อน (Weak Mutation Testing)* ที่ลดระยะเวลาในการทำงานลง หลักการของการวิเคราะห์โปรแกรมผันแปรแบบอ่อน คือ เงื่อนไขในการที่โปรแกรมผันแปรจะถูกกำจัดเพียงแค่คำสั่งที่มีการใส่ข้อผิดพลาดให้ผลลัพธ์ต่างกับคำสั่งเดียวกันในโปรแกรมทดสอบ โดย Offutt และ Lee [9] ได้ทำการทดลองเพื่อแสดงให้เห็นว่าชุดทดสอบที่ได้รับมาจากการวิเคราะห์การผันแปรแบบอ่อนนั้นมีประสิทธิผลเพียงพอสำหรับการวิเคราะห์การผันแปรทั่วไป

การทดสอบโดยอิงข้อบังคับ

ระบบที่ใช้สำหรับการสร้างชุดทดสอบเพื่อรองรับการวิเคราะห์การผันแปรนั้นมีมากมาย ซึ่งการทดสอบโดยอิงข้อบังคับ (Constraint-based Testing) [5] เป็นวิธีหนึ่งที่ใช้ในการสร้างชุดทดสอบ

การทดสอบโดยอิงข้อบังคับ ได้นิยามค่าที่เป็นไปได้ของ*ข้อมูลนำเข้า (Input)* ด้วย*ข้อบังคับ (Constraints)* ที่อยู่ในรูปของนิพจน์ทางคณิตศาสตร์ ค่าของข้อมูลนำเข้าที่รองรับข้อบังคับจะนำมาสร้างเป็นชุดทดสอบ

ข้อบังคับสำหรับกรรมวิธีนี้ ตั้งอยู่บนพื้นฐานของการกำจัดโปรแกรมผันแปรหนึ่งๆ กล่าวคือชุดทดสอบใดก็ตามที่รองรับข้อบังคับที่กำหนดไว้จะมีความสามารถในการกำจัดโปรแกรมผันแปรนั้นได้

ข้อบังคับในการกำจัดโปรแกรมผันแปรใดๆสามารถ แบ่งออกเป็น 3 ประเภท ได้ดังต่อไปนี้

1 **ข้อบังคับการไปถึง** (Reachability Constraint) เป็นข้อบังคับที่กำหนดให้ คำสั่งที่ได้รับการใส่ข้อผิดพลาดจำเป็นต้องได้รับการประมวลผล

2 **ข้อบังคับที่จำเป็น** (Necessity Constraint) เป็นข้อบังคับที่กล่าวว่า คำสั่งที่ได้รับการใส่ข้อผิดพลาดเมื่อได้รับการประมวลผลแล้วผลลัพธ์ที่ได้จากคำสั่งนั้นต้องต่างไปจากค่าเดิมในโปรแกรมทดสอบ

ตัวอย่างที่ 1

กำหนดให้โปรแกรม FindMax(m,n) มีคำสั่งดังต่อไปนี้

```
001  int FindMax(int m, int n)
002  {
003          max = m;
004          if (n>m) max = n;
005          return(max)
006  }
```

โดยสมมุติให้คำสั่งในแถวที่ 004 ได้รับการเปลี่ยนแปลงเป็น if (n>=m) max = n; ข้อบังคับที่จำเป็นสำหรับในกรณีนี้คือนิพจน์ (n > m) ไม่ควรเท่ากับนิพจน์ (n >= m) ดังนั้นข้อบังคับที่ได้มีค่าเป็น (n = m)

3 **ข้อบังคับที่เพียงพอ** (Sufficient Constraint) ถึงแม้ว่าชุดทดสอบจะรองรับข้อบังคับที่จำเป็นแล้วก็ตาม ผลลัพธ์สุดท้ายของการประมวลผลโปรแกรมอาจไม่แตกต่างไปจากเดิม ดังนั้นในข้อบังคับข้อนี้จึงบังคับให้ผลลัพธ์สุดท้ายของโปรแกรมผันแปรต่างไปจากของโปรแกรมทดสอบ

ตัวอย่างที่ 2

กำหนดให้โปรแกรมทดสอบมีคำสั่งดังต่อไปนี้

```
001  …
00n  if (x+c>=y) …
999  …
```

หากโปรแกรมผันแปรเกิดจากการแทนค่าตัวแปร x ด้วยค่า 4 ดังนั้นข้อบังคับที่จำเป็นมีค่า ($x \neq 4$) เมื่อพิจารณากรณีทดสอบ T=<x=8,y=10,c=8> ซึ่งรองรับข้อบังคับที่จำเป็น แต่ผลลัพธ์ของเงื่อนไขไม่ต่างจากเดิม คือ ทั้ง 8+c และ 4+c ต่างทำให้เงื่อนไขเป็นจริง ในกรณีเช่นนี้ต้องมีการใช้ข้อบังคับที่เพียงพอมากำหนดเพิ่ม ซึ่งมีค่าเป็น (x+c>=y) ≠ (4+c>=y) สังเกตได้ว่ากรณีทดสอบตัวเดิมนั้นไม่รองรับข้อบังคับที่เพียงพอ (8+8>=10) ≠ (4+8>=10)

3. การแบ่งกลุ่มโปรแกรมผันแปร

กำหนดให้ M เป็นชุดของโปรแกรมผันแปรที่เป็นไปได้ทั้งหมดที่ได้จากการใส่ข้อผิดพลาดให้โปรแกรมทดสอบ P กรณีทดสอบ t ถูกสร้างจากหลักการการทดสอบโดยอิงข้อบังคับเพื่อมากำจัดโปรแกรมผันแปรใดๆ ใน M เมื่อโปรแกรมผันแปรใดๆ ถูกกำจัดด้วย t แล้ว ตัวโปรแกรมผันแปรนั้นจะถูกนำออกจากชุดของโปรแกรมผันแปร กระบวนการนี้จะดำเนินต่อไปจนทุกโปรแกรมผันแปรใน M ถูกกำจัด โดยเวลาที่ใช้ในการกำจัดทุกๆ โปรแกรมผันแปรถูกประมาณด้วยจำนวนครั้งในการประมวลผลโปรแกรมผันแปร

กำหนดให้ β เป็นจำนวนของโปรแกรมผันแปรทั้งหมดและ γ เป็นจำนวนกรณีทดสอบในชุดทดสอบ T และ α_j เป็นจำนวนโปรแกรมผันแปรที่ถูกกำจัดด้วยกรณีทดสอบ t_j เวลาที่ใช้ในการทำงานจะประกอบด้วยเวลาที่ใช้ในการทดสอบทั้งหมดของแต่ละกรณีทดสอบ t_j ดังนี้

$$T_{exe} = \beta + (\beta - \alpha_1) + (\beta - \alpha_1 - \alpha_2) + ... + (\beta - \alpha_1 - \alpha_2 - ... - \alpha_{\gamma-1})$$

ดังนั้น

$$T_{exe} = \beta\gamma - \sum_{i=1}^{\gamma-1} (\gamma - i)\alpha_i$$

ถ้าเราประมาณความน่าจะเป็นของโปรแกรมผันแปรที่ถูกกำจัดด้วยกรณีทดสอบที่สร้างขึ้นเป็น α ดังนั้นเวลาที่ใช้ในการทำงานเท่ากับ

$$\begin{aligned} T_{exe} &= \beta + \beta(1-\alpha) + \beta(1-\alpha)^2 + .. + \beta(1-\alpha)^{\gamma} \\ &= \beta \sum_{i=0}^{\gamma} (1-\alpha)^i \end{aligned}$$

การลดเวลาที่ใช้ในการทำงานสามารถทำได้ด้วยการแยกกลุ่มโปรแกรมผันแปรออกเป็นหลายกลุ่ม และสร้างชุดทดสอบสำหรับแต่ละกลุ่ม

หลักการสำคัญที่ใช้ในการแบ่งกลุ่มโปรแกรมผันแปรคือ ทุกๆ โปรแกรมผันแปรที่อยู่ในกลุ่มเดียวกันควรกำจัดได้ด้วยชุดทดสอบเดียวกัน

ทฤษฎีบทย่อยที่ 1

กำหนดให้ M เป็นชุดของโปรแกรมผันแปรที่เกิดจากโปรแกรมทดสอบ P แล้ว M_1, M_2, M_3,…,M_g เป็นกลุ่มที่เกิดจากการแบ่งชุดโปรแกรมผันแปร M โดยความน่าจะเป็นของโปรแกรมผันแปรที่ถูกกำจัดเป็น α และให้ μ เป็นจำนวนที่มากที่สุดของกรณีทดสอบในแต่ละกลุ่ม ดังนั้นเวลาที่ใช้ในการทำงานของการวิเคราะห์การผันแปรเป็น

$$T_{exe} \le \beta \sum_{i=1}^{\mu} (1-\alpha)^i$$

พิสูจน์ กำหนดให้ β_j เป็นจำนวนของโปรแกรมผันแปรใน M_j สำหรับ $1 \le j \le g$ ดังนั้นเวลาที่ใช้ในการทำงานในกลุ่ม M_j เท่ากับ

$$T_{exe}(M_j) = \beta_j + \beta_j(1-\alpha) + \beta_j(1-\alpha)^2 + ..+0$$

$$T_{exe}(M_j) = \beta_j \sum_{i=0}^{\mu_j} (1-\alpha)^i$$

โดยที่ μ_j เป็นจำนวนของกรณีทดสอบสำหรับ M_j ที่ $1 \le j \le g$ นั่นคือ $\mu = \max(\mu_j)$ ดังนั้นเวลาที่ใช้ในการทำงานทั้งหมดเป็น

$$T_{exe} \le (\sum_{j=1}^{g} \beta_j)(\sum_{i=0}^{\mu} (1-\alpha)^i) = \beta(\sum_{i=0}^{\mu} (1-\alpha)^i) \qquad \blacksquare$$

เนื่องจากค่า $\mu < \gamma$ จึงทำให้เวลาที่สูญเสียสำหรับวิธีการวิเคราะห์การผันแปรแบบแบ่งกลุ่มลดลง

ทฤษฎีบทย่อยที่ 2

การทดสอบโปรแกรมด้วยหลักการ การวิเคราะห์การผันแปรด้วยการแบ่งกลุ่มโปรแกรมผันแปรจะไม่ส่งผลกระทบต่อความน่าจะเป็นของโปรแกรมผันแปรที่ถูกกำจัดในแต่ละกลุ่ม

พิสูจน์ สมมุติให้ β เป็นจำนวนของโปรแกรมผันแปรทั้งหมด โดยในกรณีทดสอบ t ใดๆ มี k โปรแกรมผันแปรที่ถูกกำจัด ความน่าจะเป็นของโปรแกรมผันแปรที่ถูกกำจัดเป็น

$$\alpha = k/\beta$$

กำหนดให้ M_1, M_2, M_3, …, M_g เป็นกลุ่มที่เกิดจากการแบ่งโปรแกรมผันแปร M ที่ k_1, k_2, k_3, …, k_g เป็นจำนวนของโปรแกรมผันแปรที่ถูกกำจัดด้วยกรณีทดสอบ t สำหรับแต่ละกลุ่ม เมื่อพิจารณากลุ่ม M_j ใดๆ ที่ $1<j<g$ กรณีทดสอบ t ที่สร้างมาจากโปรแกรมผันแปรทั้งหมดสามารถกำจัดโปรแกรมผันแปรในกลุ่ม M_j อย่างน้อย k_j ดังนั้นความน่าจะเป็นของโปรแกรมผันแปรที่ถูกกำจัดในกลุ่ม M_j ด้วยกรณีทดสอบ t มีค่าเท่ากับ

$$\alpha_j \ge k_j/\beta_j$$

โดยที่ β_j เป็นจำนวนของโปรแกรมผันแปรในกลุ่ม M_j ความน่าจะเป็นของโปรแกรมผันแปรที่ถูกกำจัด (new α) เป็น

$$new\ \alpha = \frac{\sum_{j=1}^{g} \alpha_j \beta_j}{\sum_{j=1}^{g} \beta_j} \geq \frac{\sum_{j=1}^{g} k_j}{\sum_{j=1}^{g} \beta_j} = \alpha \qquad \blacksquare$$

ทฤษฎีบทที่ 1

เวลาที่ใช้ในการทำงานของการวิเคราะห์การผันแปรสามารถลดได้เมื่อจำนวนของโปรแกรมผันแปรถูกแบ่งออกเป็นกลุ่มย่อย และความน่าจะเป็นของโปรแกรมผันแปรที่ถูกกำจัดด้วยกรณีทดสอบที่สร้างขึ้น ในแต่ละกลุ่ม (α_j)ไม่ลดลง ($\alpha_j \geq \alpha$)

พิสูจน์ การพิสูจน์สามารถทำได้โดยอาศัยทฤษฎีบทย่อยที่1 และทฤษฎีบทย่อยที่ 2 ตามลำดับ ■

ตัวอย่างที่ 3

กำหนดให้จำนวนโปรแกรมผันแปรทั้งหมดเป็น 2^m โดยการทดสอบโดยอิงข้อบังคับสามารถสร้างชุดทดสอบที่มีความน่าจะเป็นของโปรแกรมผันแปรที่ถูกกำจัดเป็น 0.5 และสามารถแบ่งกลุ่มโปรแกรมผันแปรออกเป็น 2^g กลุ่ม

ดังนั้น จำนวนโปรแกรมผันแปรในแต่ละกลุ่มเป็น 2^{m-g} และ จำนวนกรณีทดสอบในแต่ละกลุ่ม μ มีค่าเท่ากับ m-g+1 และ จำนวนกรณีทดสอบทั้งหมด γ มีค่าเท่ากับ m+1 ดังนั้น

$$newT_{exe} = \beta\sum_{i=0}^{\mu}(1-\alpha)^i = 2^m(1+\frac{1}{2}+\frac{1}{2^2}+...+\frac{1}{2^{m-g+1}})$$

โดยวิธีเดิมนั้นใช้เวลาในการคำนวณเป็น

$$T_{exe} = \beta\sum_{i=0}^{m+1}(1-\alpha)^i = 2^m(1+\frac{1}{2}+\frac{1}{2^2}+...+\frac{1}{2^{m+1}})$$

ดังนั้นเวลาที่ลดได้ด้วยวิธีการแบ่งกลุ่มเป็น

$$T_{exe} - newT_{exe} = 2^m(\frac{1}{2^{m-g+2}}+\frac{1}{2^{m-g+3}}+..+\frac{1}{2^{m+1}})$$

หลักการที่นำมาใช้ในการแบ่งกลุ่ม

ในงานวิจัยชิ้นนี้ได้เสนอหลักการสำหรับการแบ่งกลุ่มโปรแกรมผันแปรออกเป็นกลุ่มย่อย โดยอาศัยข้อบังคับการไปถึงและข้อบังคับที่จำเป็นมาเป็นตัวจำแนกกลุ่มของโปรแกรมผันแปร เพื่อความเข้าใจขอยกโปรแกรมในตัวอย่างที่ 1 หากมีการเปลี่ยนแปลงเงื่อนไขในบรรทัดที่ 003 จาก

เดิม max=m เป็น max=n และเงื่อนไขในบรรทัดที่ 004 จากเดิม if (n>m) max = n; เป็น (n ≠ m) และ (n>=m) โดยการเปลี่ยนแปลงนี้สามารถสร้างเป็นโปรแกรมผันแปรที่ 1, 2 และ 3 ตามลำดับ

เนื่องจากข้อบังคับการไปถึงของทั้งสามโปรแกรมผันแปรนั้นเป็นจริงเสมอจึงไม่จำเป็นต้องนำมาพิจารณา ส่วนข้อบังคับที่จำเป็นสำหรับโปรแกรมผันแปรแต่ละตัวนั้น คือ (n ≠ m), (n<m) และ (n=m) ตามลำดับ

ข้อบังคับของโปรแกรมผันแปรโปรแกรมที่ 1 และ 2 นั้นมีส่วนที่สามารถร่วมกันได้คือ (n<m) แต่ข้อบังคับของโปรแกรมที่ 3 นั้นมีข้อขัดแย้งกับข้อบังคับของโปรแกรมที่ 1 และ 2 ดังนั้นเราจึงแบ่งกลุ่มของโปรแกรมผันแปรออกเป็น 2 กลุ่ม คือ ในกลุ่มแรกมีโปรแกรมผันแปร1 และ 2 ส่วนในกลุ่มที่สองนั้นมีโปรแกรมผันแปร 3 เป็นต้น

อัลกอริทึมสำหรับการแบ่งกลุ่ม

พิจารณาแต่ละคำสั่งในโปรแกรมทดสอบเรียงตามลำดับ

1. ใส่ข้อผิดพลาดที่ได้รับการนิยามไว้ในตัวดำเนินการผันแปร เพื่อสร้างเป็นโปรแกรมผันแปร
2. วิเคราะห์ข้อบังคับที่ได้จากโปรแกรมผันแปรแต่ละตัว
   2.1 ถ้าข้อบังคับของโปรแกรมผันแปรไม่ขัดแย้งข้อบังคับของโปรแกรมผันแปรในกลุ่มที่มีอยู่เดิม ให้รวมโปรแกรมผันแปรนั้นเข้าไปไว้ในกลุ่ม จากนั้นเพิ่มข้อบังคับของโปรแกรมผันแปรนี้ให้กับข้อบังคับของทั้งกลุ่ม
   2.2 ในทางตรงกันข้าม ถ้าข้อบังคับของโปรแกรมผันแปรมีการขัดแย้ง สร้างเป็นกลุ่มใหม่ขึ้นมา สำหรับโปรแกรมผันแปรนี้

ผลลัพธ์ของอัลกอริทึม

โปรแกรมผันแปรที่อยู่ในต่างกลุ่มกันจะไม่สามารถถูกกำจัดได้ด้วยกรณีทดสอบเดียวกัน

การแบ่งกลุ่มของโปรแกรมผันแปรออกเป็นหลายกลุ่มที่เป็นอิสระต่อกันนั้นเพื่อให้สามารถสร้างชุดทดสอบสำหรับโปรแกรมผันแปรในแต่ละกลุ่มได้ โดยชุดทดสอบใดๆ จะไม่สามารถนำไปทดสอบโปรแกรมผันแปรในกลุ่มอื่นได้ เมื่อเปรียบเทียบในแง่จำนวนกรณีทดสอบของการวิเคราะห์การผันแปรด้วยวิธีการแบ่งกลุ่มนั้น พบว่ากรณีทดสอบจะไม่มากไปกว่าเดิม เนื่องจากในวิธีการเดิมนั้นชุดทดสอบที่ได้อาจมีกรณีทดสอบบางกรณีที่สามารถใช้แทนหลายๆ กรณีทดสอบ ได้ เมื่อแบ่งกลุ่มโปรแกรมผันแปรตามข้อบังคับการไปถึงและข้อบังคับที่จำเป็นแล้วเลือกกรณีทดสอบมาจากแต่ละกลุ่ม จะช่วยให้ลดกรณีทดสอบที่ไม่จำเป็นลงได้ เช่น กรณีทดสอบที่ได้จากข้อบังคับ x > 5 สามารถใช้แทนกรณีทดสอบที่ได้จากข้อบังคับ x > 0ได้

เวลาที่สูญเสียไปในการวิเคราะห์ข้อบังคับต่างๆ ของโปรแกรมผันแปรนั้นขึ้นอยู่กับขนาดของโปรแกรมที่นำมาทดสอบ ซึ่งข้อบังคับเหล่านี้ได้มาจากการสร้างชุดทดสอบด้วยวิธีการทดสอบโดยอาศัยข้อบังคับ ดังนั้นเวลาที่สูญเสียจริงในการทำงานคือขั้นตอนการแบ่งกลุ่มโปรแกรมผันแปร ซึ่งขึ้นกับจำนวนโปรแกรมผันแปรของโปรแกรมทดสอบเท่านั้น

4. การวิเคราะห์การผันแปรด้วยการแบ่งกลุ่ม

การวิเคราะห์การผันแปรด้วยการแบ่งกลุ่มนั้น มีการสร้างชุดทดสอบสำหรับโปรแกรมผันแปรในแต่ละกลุ่มโดยอาศัย ทฤษฎีบทย่อยที่2 กล่าวว่าความน่าจะเป็นของโปรแกรมผันแปรที่ถูกกำจัดจะไม่ลดลงเมื่อมีการแบ่งกลุ่ม

ในหัวข้อนี้เป็นการยกตัวอย่างการวิเคราะห์การผันแปรโดยวิธีการแบ่งกลุ่มมาช่วยเพิ่มประสิทธิภาพการทำงาน ดังต่อไปนี้

ตัวอย่างที่ 4 โปรแกรมผันแปรที่เกิดจากการใส่ข้อผิดพลาดให้กับโปรแกรมในตัวอย่างที่ 1 มีข้อบังคับดังต่อไปนี้

ตารางที่ 1 แสดงข้อบังคับของโปรแกรมผันแปรในตัวอย่างที่ 1

ลำดับที่	บรรทัด	โปรแกรมผันแปร	ข้อบังคับ
m1	003	max = abs(m)	m < 0
m2	003	max = -abs(m)	m > 0
m3	004	if n >= m	n = m
m4	004	if n = m	n > m
m5	004	if n ≠ m	n < m
m6	004	if n < m	n ≠ m
m7	004	if n <= m	n ≠ m
m8	004	max = abs(n)	(n > m) &&(n < 0)
M9	004	max = -abs(n)	(n > m) &&(n > 0)

จากตารางที่ 1 เราสามารถแบ่งกลุ่มของโปรแกรมผันแปรออกเป็นกลุ่มย่อยๆ ตามอัลกอริทึมที่กล่าวไว้ ได้ดังต่อไปนี้

1. 0 > n = m มีโปรแกรมผันแปร m1, m3
2. n > m > 0 มีโปรแกรมผันแปรที่ m2, m4, m6, m7, m9
3. n < m มีโปรแกรมผันแปรที่ m5

4. 0 > n > m มีโปรแกรมผันแปรที่ m8

โดยเราสามารถสร้างชุดทดสอบจากโปรแกรมผันแปรในแต่ละกลุ่มได้ดังต่อไปนี้

t_1มีค่า <n=-2, m=-2> , t_2 มีค่า <n=2, m=1> ,

t_3 มีค่า <n= 2, m= 4> , t_4 มีค่า <n=-2, m=-4>

จำนวนครั้งที่แต่ละกรณีทดสอบถูกใช้ในการทดสอบเป็นดังต่อไปนี้ (เนื่องจากโปรแกรมผันแปรที่ 3 เป็นโปรแกรมผันแปรเทียบเท่าจึงไม่นำมาพิจารณา)

ตารางที่ 2 เปรียบเทียบจำนวนครั้งในการทดสอบ

กรณีทดสอบ	เวลาในวิธีการเดิม	เวลาในวิธีการแบ่งกลุ่ม	จำนวนที่ถูกกำจัด
t_1	9	2	1
t_2	7	5	4
t_3	3	1	1
t_4	2	1	1
Total	19	9	7

จากตารางที่ 2 เห็นได้ว่าจำนวนครั้งในการทดสอบด้วยวิธีการเดิมมากกว่าวิธีการแบ่งกลุ่ม เนื่องมาจากแต่ละกรณีทดสอบต้องทดสอบโปรแกรมผันแปรภายนอกกลุ่มซึ่งไม่มีความจำเป็น โดยโปรแกรมผันแปร m_2 นั้นเป็นโปรแกรมผันแปรที่เหลืออยู่ด้วยสาเหตุมาจากข้อบังคับที่ใช้สร้างชุดทดสอบตามหลักการทดสอบโดยอิงข้อบังคับยังไม่ดีพอ

5. สรุป

งานวิจัยชิ้นนี้ได้เสนอทางแก้ปัญหาสำหรับการทดสอบโปรแกรมแบบการวิเคราะห์การผันแปร ด้วยการแยกกลุ่มโปรแกรมผันแปรออกเป็นหลายๆ กลุ่ม เพื่อลดเวลาในการกำจัดโปรแกรมผันแปร โดยนำกรณีทดสอบที่ไม่สามารถกำจัดโปรแกรมผันแปรได้ออกจากการทดสอบ ในการแบ่งกลุ่มอาศัยหลักการของข้อบังคับในการกำจัดโปรแกรมผันแปร 2 ข้อคือ ข้อบังคับการไปถึง และ ข้อบังคับที่จำเป็น มาเป็นเกณฑ์ในการแบ่งกลุ่ม

ข้อได้เปรียบที่เหนือกว่ากรรมวิธีส่วนใหญ่ที่เกี่ยวข้องกับการปรับปรุงประสิทธิภาพ คือ การรักษาระดับประสิทธิผลของการวิเคราะห์การผันแปรซึ่งหมายถึงชุดทดสอบที่ได้รับจากกระบวนการวิเคราะห์การผันแปรนั้นจะเป็นชุดทดสอบที่มีประสิทธิผลเทียบเท่ากับวิธีเดิม

จำนวนกรณีทดสอบที่ได้จากวิธีการแบ่งกลุ่มนั้นจะไม่มากไปกว่าวิธีเดิมเนื่องจากเราเลือกกรณีทดสอบมาสำหรับแต่ละกลุ่มทำให้ลดกรณีทดสอบที่ไม่จำเป็นลงได้ ทั้งนี้จำนวนกรณีทดสอบที่ได้รับนี้ขึ้นกับประสิทธิภาพของวิธีที่นำมาใช้ในการแบ่งกลุ่มโปรแกรมผันแปร ดังนั้นแนวทางในการพัฒนาต่อคือการหากรรมวิธีในการแบ่งกลุ่มได้ดีที่สุดมาประยุกต์ใช้งาน

6. เอกสารอ้างอิง

[1] James M. Bieman, Sudipto Ghosh, Roger T. Alexander, “A Technique for Mutation of Java Objects”, *Proceedings Automated Software Engineering Conference(ASE 2001)*, 2001.

[2] Paul E. Black, V. Okun, Yaacov Yesha, “Mutation Operators for specification”, *IEEE International Conference on Automated Software Engineering*, 2000 .

[3] Philippe Chevalley, “Applying Mutation Analysis for Object-Oriented Program Using a Reflective Approach”, *Asia-Pacific Software Engineering Conference* , 2001.

[4] R. A. Demillo, R. J. Lipton, and F.G. Sayward, “Hints on Test Data Selection: Help for the Practicing Programmer”, *IEEE Computer*, 11(4), 1978, pp. 34-41.

[5] R. A. Demillo , A. Jefferson Offutt , “Constraint-Based Automatic Test Data Generation”, *IEEE Transaction on Software Engineering* , 1991.

[6] W. E. Howden, “Weak Mutation Testing and Completeness of Test Sets”, *IEEE Transactions of Software Engineering* , 8(4), 1982, pp. 371-379.

[7] A. J. Offutt, G. Rothermel, and C. Zapf, “An Experimental Evaluation of Selective Mutation”, *Internal Conference on Software Engineering*, 1993, pp.100-107.

[8] Suet Chun Lee, A. J. Offutt, “Generating Test Cases for XML-Based Web Component Interactions Using Mutation Analysis”, *The Twelfth IEEE International Symposium on Software Reliability Engineering (ISSRE ‘01)*, 2001, pp. 200-209.

[9] A. J. Offutt, and Ss. D. Lee, “An Empirical Evaluation of Weak Mutation”, *IEEE Transactions on Software Engineering*, 20(5), 1994, pp. 337-344.

[10] A. J. Offutt, “A Practical System for Mutation Testing: Help for the Common Programmer”, *International Conference on Testing Computer Software*, 1995, pp. 99-109.

ประวัติผู้เขียนวิทยานิพนธ์

นายวรท วรวัฒน์พิบูลย์ เกิดเมื่อวันที่ 25 สิงหาคม พ.ศ. 2524 ที่จังหวัดกรุงเทพ สำเร็จการศึกษาระดับปริญญาบัณฑิต หลักสูตรวิศวกรรมศาสตรบัณฑิต (วศ. บ.) สาขาวิศวกรรมคอมพิวเตอร์ จากเกษตรศาสตร์มหาวิทยาลัย เมื่อ พ.ศ. 2545 และได้เข้าศึกษาต่อในหลักสูตรวิศวกรรมศาสตรมหาบัณฑิต (วศ. ม.) สาขาวิศวกรรมคอมพิวเตอร์ ณ จุฬาลงกรณ์มหาวิทยาลัย เมื่อปี พ.ศ. 2545