การสร้างกรณีทดสอบจากแผนภาพ use case และแผนภาพกิจกรรม บนพื้นฐานของไวยากรณ์

Syntax-based Test Cases Generation from UML Use Case and Activity Diagrams

กาญจนี เพชรทะนันท์

Kanjanee Pechtanun

วิทยานิพนธ์นี้เป็นส่วนหนึ่งของการศึกษาตามหลักสูตรปริญญา
วิทยาศาสตรมหาบัณฑิต สาขาวิชาวิทยาการคอมพิวเตอร์
มหาวิทยาลัยสงขลานครินทร์

A Thesis Submitted in Partial Fulfillment of the Requirements
for the Degree of Master of Science in Computer Science
Prince of Songkla University
2556

การสร้างกรณีทดสอบจากแผนภาพ use case และแผนภาพกิจกรรม บนพื้นฐานของไวยากรณ์

Syntax-based Test Cases Generation from UML Use Case and Activity Diagrams

กาญจนี เพชรทะนันท์

Kanjanee Pechtanun

วิทยานิพนธ์นี้เป็นส่วนหนึ่งของการศึกษาตามหลักสูตรปริญญา
วิทยาศาสตรมหาบัณฑิต สาขาวิชาวิทยาการคอมพิวเตอร์
มหาวิทยาลัยสงขลานครินทร์

A Thesis Submitted in Partial Fulfillment of the Requirements for the Degree of
Master of Science in Computer Science
Prince of Songkla University
2556

ชื่อวิทยานิพนธ์ การสร้างกรณีทดสอบจากแผนภาพ use case และแผนภาพกิจกรรมบนพื้นฐานของไวยากรณ์

ผู้เขียน นางสาวกาญจนี เพชรทะนันท์

สาขาวิชา วิทยาการคอมพิวเตอร์

อาจารย์ที่ปรึกษาวิทยานิพนธ์หลัก	**คณะกรรมการสอบ**
…….....	……..ประธานกรรมการ
(ดร.สุภาภรณ์ กานต์สมเกียรติ)	(ผู้ช่วยศาสตราจารย์ ดร.กฤดาภัทร สีหารี)
	…….....กรรมการ
	(ดร.สุภาภรณ์ กานต์สมเกียรติ)
	…….....กรรมการ
	(ผู้ช่วยศาสตราจารย์ ดร.อำนาจ เปาะทอง)

บัณฑิตวิทยาลัย มหาวิทยาลัยสงขลานครินทร์ อนุมัติให้นับวิทยานิพนธ์ฉบับนี้เป็นส่วนหนึ่งของการศึกษา ตามหลักสูตรปริญญาวิทยาศาสตรมหาบัณฑิต สาขาวิชาวิทยาการคอมพิวเตอร์

……..……

(รองศาสตราจารย์ ดร.ธีระพล ศรีชนะ)

คณบดีบัณฑิตวิทยาลัย

ขอรับรองว่า ผลงานวิจัยนี้เป็นผลมาจากการศึกษาวิจัยของนักศึกษาเอง และขอแสดงความขอบคุณบุคคลที่มีส่วนเกี่ยวข้อง

ลงชื่อ.....
(ดร. สุภาภรณ์ กานต์สมเกียรติ)
อาจารย์ที่ปรึกษาวิทยานิพนธ์

ลงชื่อ.....
(นางสาวกาญจนี เพชรทะนันท์)
นักศึกษา

ข้าพเจ้าขอรับรองว่า ผลงานวิจัยนี้ไม่เคยเป็นส่วนหนึ่งในการอนุมัติปริญญาในระดับใดมาก่อน และไม่ได้ถูกใช้ในการยื่นขออนุมัติปริญญาในขณะนี้

ลงชื่อ.....
(นางสาวกาญจนี เพชรทะนันท์)
นักศึกษา

ชื่อวิทยานิพนธ์ การสร้างกรณีทดสอบจากแผนภาพ use case และแผนภาพกิจกรรมบนพื้นฐานของไวยากรณ์
ผู้เขียน นางสาวกาญจนี เพชรทะนันท์
สาขาวิชา วิทยาการคอมพิวเตอร์
ปีการศึกษา 2555

บทคัดย่อ

การทดสอบซอฟต์แวร์ เป็นขั้นตอนหนึ่งที่สำคัญในกระบวนการผลิตซอฟต์แวร์ ซึ่งช่วยให้ซอฟต์แวร์มีความถูกต้องและมีความน่าเชื่อถือมากขึ้น ในปัจจุบันการออกแบบซอฟต์แวร์มักถูกออกแบบโดยใช้ภาษาการออกแบบ ภาษาการออกแบบที่นิยมใช้คือภาษาการออกแบบเชิงโมเดล (Unified Modeling Language: UML) ซึ่งเป็นภาษาที่แสดงโครงสร้างและรายละเอียดของระบบเชิงวัตถุ แผนภาพ use case (Use case diagram) เป็นแผนภาพที่แสดงภาพรวมของกระบวนการทำงานทั้งหมดของระบบ แผนภาพกิจกรรม (Activity Diagram) เป็นแผนภาพที่แสดงพฤติกรรมของระบบ งานวิจัยนี้เสนอการสร้างกรณีทดสอบจากแผนภาพ use case และแผนภาพกิจกรรมบนพื้นฐานของไวยากรณ์ โดยการวิเคราะห์แผนภาพ use case เพื่อให้ได้รายการ use-case ทั้งหมดที่จะต้องนำมาดำเนินการในการทดสอบระบบ จากนั้นจึงนำแผนภาพกิจกรรมที่สอดคล้องกับแต่ละ use case มาเตรียมชุดข้อมูลเบื้องต้น และสร้างตารางการขึ้นต่อกันและตารางเงื่อนไขทางตรรกะจากชุดข้อมูลเบื้องต้น จากนั้นจึงนำทั้งสองตารางมาพิจารณาร่วมกันเพื่อสร้างไวยากรณ์ และสร้างกรณีทดสอบจากไวยากรณ์ งานวิจัยนี้ถูกนำไปประยุกต์ใช้กับ 6 กรณีศึกษา ซึ่งพบว่ากรณีทดสอบที่สร้างได้มีประสิทธิภาพในการตรวจจับข้อผิดพลาดได้ดี ทั้งยังสามารถเริ่มการทดสอบได้ตั้งแต่ช่วงต้นของกระบวนการผลิตซอฟต์แวร์ ส่งผลให้ค่าใช้จ่ายในกระบวนการผลิตซอฟต์แวร์ลดลง

Thesis Title	Syntax-based Test Cases Generation from UML Use Case and Activity Diagrams
Author	Miss Kanjanee Pechtanun
Major Program	Computer Science
Academic Year	2012

ABSTRACT

Software testing is an important process in the software development life-cycle. It increases accuracy and make software more reliable. Nowadays, software designs are often designed by using software design language. The Unified Modeling Language (UML) is the most popular standard software design language that shows the object-oriented system structure and details. In UML, use case diagram shows an overview of a system and activity diagram shows the behavior of a system. This research proposes the syntax-based test cases generation from UML use case and activity diagrams. First, use case diagram is analyzed to create a list of use cases for system testing. Second, the activity diagram corresponding to each use case is used to create a set of data. The data set is used to created dependency and condition tables. Then, these two tables are used to built a grammar. Finally, test cases are generated from the grammar. This method was applied to six case studies. The experimental result shows that the generated test cases can effectively detect errors. Moreover, the test can be started early in a software development life-cycle that reducing the cost of software development.

กิตติกรรมประกาศ

วิทยานิพนธ์ฉบับนี้สำเร็จได้ด้วยความช่วยเหลือและสนับสนุนจากบุคคลหลายฝ่าย ผู้วิจัยรู้สึกซาบซึ้งและขอกราบขอบพระคุณอย่างสูง คือ

ดร.สุภาภรณ์ กานต์สมเกียรติ อาจารย์ที่ปรึกษาวิทยานิพนธ์ ที่กรุณาให้ความรู้ คำแนะนำ และช่วยเหลือในการแก้ปัญหาต่างๆ ให้แก่ผู้วิจัยเสมอมา พร้อมทั้งตรวจทานและแก้ไขวิทยานิพนธ์ให้แก่ผู้วิจัย

ผู้ช่วยศาสตราจารย์ ดร.กฤดาภัทร สีหารี ประธานกรรมการสอบวิทยานิพนธ์ ที่กรุณาให้ข้อเสนอแนะและช่วยตรวจทานแก้ไขวิทยานิพนธ์ให้มีความถูกต้อง สมบูรณ์

ผู้ช่วยศาสตราจารย์ ดร.อำนาจ เปาะทอง กรรมการในการสอบวิทยานิพนธ์ ที่กรุณาให้ข้อเสนอแนะและช่วยตรวจทานแก้ไขวิทยานิพนธ์ให้มีความถูกต้อง สมบูรณ์

คณะวิทยาศาสตร์ มหาวิทยาลัยสงขลานครินทร์ ที่ช่วยสนับสนุนทุนการศึกษาทุนผู้ช่วยวิจัยคณะวิทยาศาสตร์

อาจารย์ภาควิชาวิทยาการคอมพิวเตอร์ คณะวิทยาศาสตร์ มหาวิทยาลัยสงขลานครินทร์ทุกท่านที่ให้ความรู้ทางด้านวิชาการ และคำแนะนำในการทำวิทยานิพนธ์

เจ้าหน้าที่ภาควิชาวิทยาการคอมพิวเตอร์ และเจ้าหน้าที่บัณฑิตวิทยาลัย ทุกท่านที่ให้ความช่วยเหลือ และอำนวยความสะดวกเกี่ยวกับเอกสารและอุปกรณ์ต่างๆ

เพื่อนๆ พี่ๆ และน้องๆ ภาควิชาวิทยาการคอมพิวเตอร์ คณะวิทยาศาสตร์ ที่ให้คำปรึกษา และกำลังใจในการทำวิทยานิพนธ์

คุณพ่อ คุณแม่ และครอบครัว ที่สนับสนุนและให้กำลังใจแก่ผู้วิจัยเสมอมา

ผู้วิจัยขอขอบคุณทุกท่านเป็นอย่างสูงมา ณ โอกาสนี้

กาญจนี เพชรทะนันท์

สารบัญ

หน้า

สารบัญ (ต่อ)

หน้า

รายการตาราง

รายการภาพประกอบ

รายการภาพประกอบ (ต่อ)

ภาพประกอบ **หน้า**

รายการภาพประกอบ (ต่อ)

ภาพประกอบ **หน้า**

บทที่ 1

บทนำ

เนื้อหาในบทนี้กล่าวถึงความสำคัญและที่มาของงานวิจัย งานวิจัยที่เกี่ยวข้อง วัตถุประสงค์ของงานวิจัย ขอบเขตการดำเนินงานของการวิจัย ขั้นตอนและระยะเวลาการดำเนินงาน สถานที่และเครื่องมือที่ใช้ในการดำเนินการวิจัย และประโยชน์ที่คาดว่าจะได้รับจากงานวิจัย

1.1 ความสำคัญและที่มาของงานวิจัย

การทดสอบซอฟต์แวร์ ถือเป็นขั้นตอนหนึ่งที่มีความสำคัญในกระบวนการผลิตซอฟต์แวร์ โดยใช้ความรู้ทางด้านเทคนิค เพื่อระบุและค้นหาข้อผิดพลาดหรือข้อบกพร่องของซอฟต์แวร์ การทดสอบทำให้ซอฟต์แวร์มีความถูกต้อง ความสมบูรณ์ คุณภาพ และความน่าเชื่อถือเพิ่มขึ้น แต่อย่างไรก็ตาม ซอฟต์แวร์ในปัจจุบันมักมีขนาดใหญ่ และมีความซับซ้อนมากขึ้น ส่งผลให้ค่าใช้จ่ายในการทดสอบซอฟต์แวร์เพิ่มมากขึ้นตามไปด้วย การสร้างกรณีทดสอบได้โดยอัตโนมัติ และกระทำได้ตั้งแต่ช่วงต้นของการพัฒนาซอฟต์แวร์ จะช่วยให้การทดสอบและตรวจจับข้อผิดพลาดหรือข้อบกพร่องต่างๆได้ก่อนขั้นตอนการพัฒนาซอฟต์แวร์ และช่วยให้ค่าใช้จ่ายในการทดสอบซอฟต์แวร์ลดลง ส่งผลให้ค่าใช้จ่ายในกระบวนการผลิตซอฟต์แวร์ลดลงได้อีกด้วย

ในปัจจุบัน นักเขียนโปรแกรมนิยมพัฒนาซอฟต์แวร์เชิงวัตถุ เนื่องจากสามารถพัฒนาซอฟต์แวร์ที่ซับซ้อนและมีขนาดใหญ่ได้สะดวก และรวดเร็ว ทั้งยังสามารถนำชุดโปรแกรมกลับมาแก้ไข หรือนำไปประยุกต์ใช้กับซอฟต์แวร์อื่นได้ จึงมีการสร้างเครื่องมือเพื่อช่วยในการวิเคราะห์และออกแบบซอฟต์แวร์เชิงวัตถุ เช่น Unified Modeling Language (UML) มาช่วยในการวิเคราะห์และออกแบบระบบเชิงวัตถุ โดยแสดงโครงสร้างและรายละเอียดต่างๆของระบบ ซึ่งประกอบด้วยการออกแบบหลักๆอยู่ 3 กลุ่ม คือ การออกแบบเชิงโครงสร้าง (Structure) การออกแบบเชิงพฤติกรรม (Behavior) และ การออกแบบเชิงปฏิสัมพันธ์ (Interaction) UML ประกอบด้วยแผนภาพหลายชนิด ซึ่งแผนภาพ use case (Use case Diagram) และแผนภาพกิจกรรม (Activity Diagram) เป็นแผนภาพที่อยู่ในแผนภาพ UML เช่นกัน ในส่วนของแผนภาพ use case จะแสดงภาพรวมของกระบวนการทำงานทั้งหมดของ

ระบบ และแผนภาพกิจกรรมจะแสดงพฤติกรรมของซอฟต์แวร์โดยอธิบายลำดับการทำงานที่เกิดขึ้นในซอฟต์แวร์ตั้งแต่เริ่มต้นจนสิ้นสุด

อภิภาษา (Metalanguage) เป็นภาษาหนึ่งที่ใช้ในการอธิบายรูปแบบของภาษาจุดหมาย (Object Language) ซึ่งไวยากรณ์ (Syntax) ถือเป็นรูปแบบหนึ่งของอภิภาษาที่นำมาใช้ในการอธิบายรูปแบบของภาษาการโปรแกรม ในปัจจุบันมีภาษาการโปรแกรมเป็นจำนวนมากและล้วนแต่สามารถอธิบายด้วยไวยากรณ์ได้ทั้งสิ้น ดังนั้นไวยากรณ์เปรียบเสมือนภาษากลางที่สามารถอธิบายภาษาการโปรแกรมอื่นๆได้ และด้วยปัจจุบัน ระบบที่มีขนาดใหญ่และมีความซับซ้อน เช่น ระบบธนาคาร ระบบโรงพยาบาล ระบบมหาวิทยาลัย มักถูกออกแบบโดยใช้แผนภาพ UML มากกว่าหนึ่งแผนภาพ เช่น ระบบธนาคาร ถูกออกแบบภาพรวมด้วยแผนภาพ use case ออกแบบลำดับขั้นตอนของกระบวนการทำงานต่างๆด้วยแผนภาพกิจกรรม หรือออกแบบสถานะภายในของกระบวนการทำงานโดยใช้แผนภาพสถานะ ส่งผลให้การทดสอบซอฟต์แวร์ที่มีขนาดใหญ่และมีความซับซ้อนจำเป็นต้องนำแผนภาพการออกแบบต่างๆมาใช้ร่วมกันในการสร้างกรณีทดสอบ เพื่อให้ได้กรณีทดสอบที่ครอบคลุมและสมบูรณ์ ด้วยเหตุนี้ ผู้วิจัยจึงนำเสนอการอธิบายแผนภาพต่างๆด้วยไวยากรณ์ โดยไวยากรณ์จะทำหน้าที่เป็นภาษากลางในการอธิบายแผนภาพต่างๆและเป็นเครื่องมือสำหรับสร้างกรณีทดสอบได้อีกด้วย โดยนำเสนอการสร้างกรณีทดสอบจากแผนภาพ use case และแผนภาพกิจกรรมบนพื้นฐานของไวยากรณ์

1.2 งานวิจัยที่เกี่ยวข้อง

A Transformation-based Approach to Generating Scenario-oriented Test Cases from UML Activity Diagram for Concurrent Application (Sun, 2008)

งานวิจัยนี้ เสนอการสร้างกรณีทดสอบจากแผนภาพกิจกรรม โดยแปลงแผนภาพกิจกรรมให้อยู่ในรูปตาราง

- ตาราง Extracted activities for the Activity Diagram Specification
- ตาราง Transformed Nodes in the Intermediate Representation

จากนั้นนำตารางมาสร้างเป็นโครงสร้างต้นไม้และสร้างลำดับขั้นตอนของการทดสอบโดยการท่องเข้าไปตามแนวกว้างของโครงสร้างต้นไม้ แล้วจึงสร้างกรณีทดสอบจากลำดับขั้นตอนของการทดสอบ

Test Case Generation from UML Subactivity and Activity Diagram (Fan, *et al.,* 2009)

งานวิจัยนี้ เสนอการสร้างกรณีทดสอบจากแผนภาพกิจกรรมและแผนภาพกิจกรรมย่อย โดยสร้าง Activity Diagram Composition Tree (ADCT) จากแผนภาพกิจกรรมและแผนภาพกิจกรรมย่อย จากนั้นสร้างกรณีทดสอบโดยพิจารณาโครงสร้างต้นไม้จากล่างขึ้นบน ซึ่งแบ่งการสร้างกรณีทดสอบออกเป็น 3 ส่วนคือ

- กรณีทดสอบสำหรับ leaf node
- กรณีทดสอบสำหรับ parent node
- กรณีทดสอบสำหรับ root node

จากนั้นรวมกรณีทดสอบ โดยที่ทุกกรณีทดสอบต้องเป็นเส้นทางจาก root node ไปจนถึง leaf node

Generation Test Cases from UML Activity Diagram using the Condition-Classification Tree Method (Kansomkeat, *et al.,* 2010)

งานวิจัยนี้ เสนอการสร้างกรณีทดสอบโดยแบบจำลองต้นไม้การจำแนกแบบมีเงื่อนไขจากแผนภาพกิจกรรม โดยแปลงแผนภาพกิจกรรมให้อยู่ในรูปของต้นไม้การจำแนกแบบมีเงื่อนไขโดย

1. พิจารณาจุดตัดสินใจจากแผนภาพกิจกรรม และสร้าง category node ของจุดตัดสินใจนั้นลงในต้นไม้การจำแนก
2. พิจารณาแต่ละ transition จากจุดตัดสินใจ เพื่อนำมาสร้างเป็น child node ของ category node ซึ่งเรียกว่า option node
3. ระบุเงื่อนไขก่อนหน้าให้กับต้นไม้การจำแนกโดยระบุบนต้นไม้การจำแนก

จากนั้นสร้างตารางกรณีทดสอบโดยการนำต้นไม้การจำแนกมาวางเรียงลำดับตามจำนวนเงื่อนไขจากน้อยไปมาก หากจำนวนเงื่อนไขเท่ากัน สามารถวางก่อนหรือหลังก็ได้ แล้วจึงสร้างกรณีทดสอบ โดยทำสัญลักษณ์ลงบนเส้นแถวของตาราง ณ ตำแหน่งที่โหนดลูกสามารถจับคู่กันได้จากเงื่อนไขที่ระบุ

Automatic Test Case Generation from UML Models (Sarma and Mall, 2007)

งานวิจัยนี้ เสนอการสร้างกรณีทดสอบจากแผนภาพ use case และแผนภาพ sequence โดยแปลงแผนภาพ use case ให้อยู่ในรูปของกราฟชื่อ Use case Diagram Graph (UDG) และแปลงแผนภาพ sequence ให้อยู่ในรูปของกราฟชื่อ Sequence Diagram Graph (SDG) จากนั้นจึงทำการรวมกราฟ UDG และ SDG เข้าด้วยกันเป็นกราฟ System Testing Graph (STG) แล้วจึงสร้างกรณีทดสอบโดยการท่องเข้าไปในกราฟโดยใช้เกณฑ์การครอบคลุมทุก transition

Test Case Generation Based on State and Activity Models (Swain, *et al.*, 2010)

งานวิจัยนี้ เสนอการสร้างกรณีทดสอบจากแผนภาพสถานะและแผนภาพกิจกรรม โดยนำแผนภาพสถานะและแผนภาพกิจกรรมมาสร้างเป็น State-Activity Diagram (SAD) ซึ่งอยู่ในรูปแบบของกราฟที่ประกอบด้วย

- state-activity node
- AND-OR node
- edges

จากนั้นสร้าง basis path โดยพิจารณาเกณฑ์การครอบคลุมและใช้ UTCG algorithm แล้วจึงสร้างแผนการทดสอบ และสร้างกรณีทดสอบจากแผนการทดสอบที่ได้

รายละเอียดของงานวิจัยที่เกี่ยวข้องสรุปดังตารางที่ 1.1

ตารางที่ 1.1 ตารางเปรียบเทียบงานวิจัยที่เกี่ยวข้อง

ปี	ผู้วิจัย	งานวิจัย	แผนภาพ UML				โครงสร้าง			เกณฑ์การครอบคลุม	
			Use case	Activity	Sequence	State-chart	กราฟ	ต้นไม้	ตาราง	Branch	Path
2007	M.Sarma และ R.Mall	Automatic Test Case Generation from UML Models	✓		✓		✓			✓	
2008	Chan-ai Sun	A Transformation-based Approach to Generating Scenario-oriented Test Cases from UML Activity Diagram for Concurrent Application		✓				✓	✓		✓
2009	Xin Fan และคณะ	Test Case Generation from UML Subactivity and Activity Diagram		✓				✓			✓
2010	Supaporn Kansomkeat และคณะ	Generation Test Cases from UML Activity Diagram using the Condition-Classification Tree Method		✓				✓	✓		
2010	Santosh Kumar Swain และคณะ	Test Case Generation Based on State and Activity Models		✓		✓	✓				✓

1.3 วัตถุประสงค์ของงานวิจัย

1. เพื่อเสนอวิธีการสร้างกรณีทดสอบจากแผนภาพ use case และแผนภาพกิจกรรมบนพื้นฐานของไวยากรณ์สำหรับการทดสอบระบบ
2. เพื่อพัฒนาเครื่องมือสำหรับสร้างกรณีทดสอบจากแผนภาพ use case และแผนภาพกิจกรรมบนพื้นฐานของไวยากรณ์ที่นำเสนอ

1.4 ขอบเขตการดำเนินงานของการวิจัย

1. เสนอวิธีการสร้างกรณีทดสอบจากแผนภาพ use case และแผนภาพกิจกรรมบนพื้นฐานของไวยากรณ์สำหรับการทดสอบระบบ
2. พัฒนาเครื่องมือสำหรับสร้างกรณีทดสอบจากแผนภาพ use case และแผนภาพกิจกรรมบนพื้นฐานของไวยากรณ์ที่นำเสนอ

1.5 ขั้นตอนและระยะเวลาการดำเนินงาน

1.5.1 ขั้นตอนการดำเนินงาน

1) ศึกษางานวิจัยและเอกสารที่เกี่ยวข้อง

2) วิเคราะห์และออกแบบไวยากรณ์

3) พัฒนาและปรับปรุงเครื่องมือ

4) ประเมินประสิทธิภาพของกรณีทดสอบ

5) เขียนบทความวิจัยและเผยแพร่

6) จัดทำรายงานการวิจัย

1.5.2 ระยะเวลาการดำเนินงาน

มกราคม 2555 – มีนาคม 2556

1.5.3 แผนการดำเนินการวิจัย

ระยะเวลาการดำเนินการวิจัยแสดงดังตารางที่ 1.2

ตารางที่ 1.2 แผนการดำเนินการวิจัย

ขั้นตอนการดำเนินงาน	2555												2556		
	1	2	3	4	5	6	7	8	9	10	11	12	1	2	3
1. ศึกษางานวิจัยและเอกสารที่เกี่ยวข้อง	■	■	■	■	■	■	■								
2. วิเคราะห์และออกแบบไวยากรณ์			■	■	■	■	■	■							
3. พัฒนาและปรับปรุงเครื่องมือ							■	■	■	■	■				
4. ประเมินประสิทธิภาพของกรณีทดสอบที่ได้					■	■	■	■	■	■	■	■			
5. เขียนบทความวิจัยและเผยแพร่					■	■	■	■	■	■	■	■	■	■	■
6. จัดทำรายงานการวิจัย										■	■	■	■	■	■

1.6 สถานที่และเครื่องมือที่ใช้

1.6.1 สถานที่

ห้องปฏิบัติการวิจัยวิศวกรรมซอฟต์แวร์และการประยุกต์ CS 209 ภาควิชาวิทยาการคอมพิวเตอร์ คณะวิทยาศาสตร์ มหาวิทยาลัยสงขลานครินทร์ วิทยาเขตหาดใหญ่

1.6.2 เครื่องมือที่ใช้

1) ด้านฮาร์ดแวร์

เครื่องคอมพิวเตอร์ส่วนบุคคล

- CPU : Intel core 2 Quad 2.40 GHz
- Hard disk : 320 GB
- RAM : 2 GB

2) ด้านซอฟต์แวร์

- Windows XP Professional Service Pack 3
- Java
- Microsoft Office visio 2003
- Visual studio 2003

1.7 ประโยชน์ที่คาดว่าจะได้รับ

1 ได้วิธีการสร้างกรณีทดสอบจากแผนภาพ use case และแผนภาพกิจกรรมบนพื้นฐานของไวยากรณ์สำหรับการทดสอบระบบ

2 ได้เครื่องมือเพื่อใช้ในการสร้างกรณีทดสอบจากแผนภาพ use case และแผนภาพกิจกรรมบนพื้นฐานของไวยากรณ์ที่นำเสนอ

บทที่ 2

ทฤษฎีที่เกี่ยวข้อง

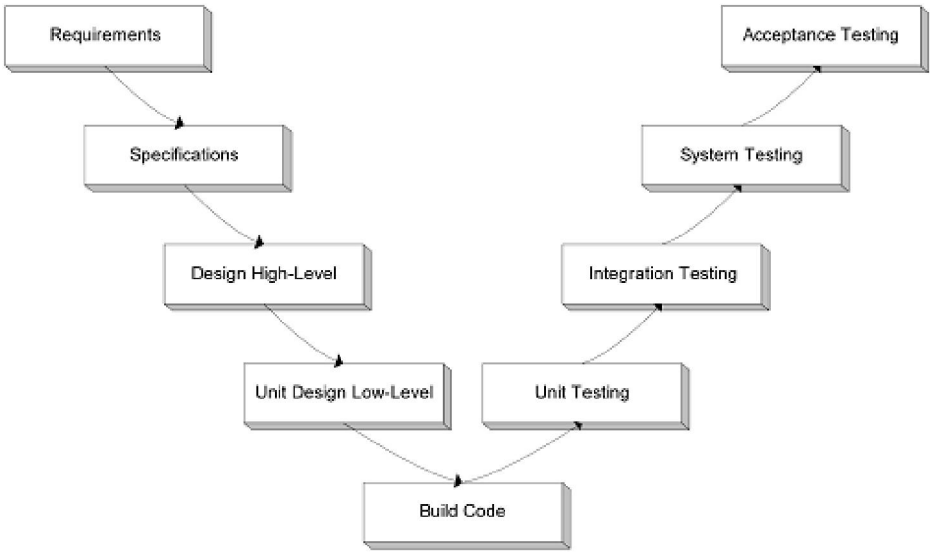
เนื้อหาในบทนี้กล่าวถึงทฤษฎีและหลักการต่าง ๆ โดยส่วนแรกกล่าวถึงการทดสอบซอฟต์แวร์ ระดับการทดสอบซอฟต์แวร์ และเทคนิคการทดสอบซอฟต์แวร์ ส่วนที่สองกล่าวถึงแผนภาพ use case ส่วนที่สามเกี่ยวกับแผนภาพกิจกรรม ส่วนที่สี่เป็นเนื้อหาเกี่ยวกับไวยากรณ์บีเอ็นเอฟ และส่วนที่ห้ากล่าวถึง mutation testing

2.1 การทดสอบซอฟต์แวร์ (Software Testing)

การทดสอบซอฟต์แวร์ (Ammann and Offutt, 2008) เป็นขั้นตอนหนึ่งที่มีความสำคัญในกระบวนการผลิตซอฟต์แวร์ ซึ่งทำควบคู่ไปกับขั้นตอนต่าง ๆของการพัฒนาซอฟต์แวร์ โดยใช้ความรู้ทางด้านเทคนิคเพื่อระบุและค้นหาข้อผิดพลาดหรือข้อบกพร่องของซอฟต์แวร์ และทำการแก้ไขปรับปรุงข้อผิดพลาดหรือข้อบกพร่องนั้น ๆเพื่อให้ซอฟต์แวร์มีความถูกต้อง สมบูรณ์ มีคุณภาพดี และมีความน่าเชื่อถือ

2.1.1 ระดับการทดสอบซอฟต์แวร์

การทดสอบซอฟต์แวร์แบ่งออกเป็น 5 ระดับซึ่งทำควบคู่กับขั้นตอนในการพัฒนาซอฟต์แวร์ แต่ละระดับของการทดสอบจะเกี่ยวข้องกับแต่ละขั้นตอนในการพัฒนาซอฟต์แวร์ อธิบายโดยใช้แผนภาพวี (V-Model) แสดงดังภาพประกอบที่ 2.1



ภาพประกอบที่ 2.1 แผนภาพวี (V-Model) (FHWA, 2004)

1) การทดสอบระดับหน่วย (Unit Testing) เป็นการทดสอบหน่วยย่อยที่สุดของซอฟต์แวร์อย่างอิสระต่อกัน เพื่อค้นหาข้อผิดพลาดของแต่ละหน่วยย่อย

2) การทดสอบระดับรวมหน่วย (Integration Testing) เป็นการทดสอบการทำงานของกลุ่มโปรแกรมหรือส่วนประกอบย่อยที่ถูกประสานเข้าด้วยกัน เพื่อทำงานหน้าที่ใดหน้าที่หนึ่งร่วมกัน ถึงแม้ว่าแต่ละโปรแกรมย่อยจะผ่านการทดสอบมาแล้วก็ตาม แต่เมื่อต้องทำงานร่วมกันอาจเกิดข้อผิดพลาดบางประการได้ จึงต้องมีการทดสอบการรวมกันเพื่อค้นหาข้อผิดพลาดดังกล่าว

3) การทดสอบระบบ (System Testing) เป็นการทดสอบการทำงานของระบบโดยรวมซึ่งเกิดจากการรวมกันขององค์ประกอบต่างๆ

4) การทดสอบการยอมรับ (Acceptance Testing) เป็นการทดสอบเพื่อประเมินประสิทธิภาพของระบบว่าสามารถทำงานได้ถูกต้องตามความต้องการและเป็นที่พึงพอใจของผู้ใช้หรือไม่

2.1.2 เทคนิคการทดสอบซอฟต์แวร์

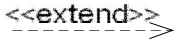
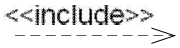
1) Black-box testing หรือการทดสอบแบบกล่องดำ เป็นการทดสอบโดยไม่คำนึงถึงกลไกภายในของระบบ ซึ่งจะทดสอบการทำงานต่าง ๆของระบบให้มีความถูกต้องตรงตามความต้องการ

2) White-box testing หรือการทดสอบแบบกล่องขาว เป็นการทดสอบโดยพิจารณากลไกภายในของระบบ โดยดูโครงสร้างการทำงานต่าง ๆของระบบ ซึ่งสามารถพิจารณาการทดสอบโดย ทดสอบทุกเส้นทางในกระบวนการทำงาน ทดสอบทุกการตัดสินใจทางตรรกะ ทดสอบการทำงานภายในลูป และ ทดสอบโครงสร้างข้อมูลภายใน

2.2 แผนภาพ use case (Use Case Diagram)

แผนภาพ use case (Booch, *et al.,* 1998) เป็นแผนภาพที่แสดงภาพรวมของกระบวนการทำงานทั้งหมดของระบบจากมุมมองของผู้ใช้ภายนอก ซึ่งจะแสดงการทำงานของผู้ใช้ระบบและแสดงความสัมพันธ์ระหว่างระบบย่อย โดยที่ผู้ใช้ระบบจะถูกกำหนดเป็น actor และ ระบบย่อยถูกกำหนดเป็น use case สัญลักษณ์พื้นฐานของแผนภาพ use case แสดงดังตารางที่ 2.1

ตารางที่ 2.1 สัญลักษณ์พื้นฐานของแผนภาพ use case

สัญลักษณ์	ชื่อสัญลักษณ์	คำอธิบาย
	Use case	กระบวนการทำงานต่าง ๆของระบบ
	Actor	ผู้ใช้ระบบ
	System Boundary	กรอบการทำงานของระบบ
	Association	เส้นที่เชื่อม use case ไปยังองค์ประกอบอื่น
	Extend	ความสัมพันธ์จาก extension use case ไปยัง base use case
	Include	ความสัมพันธ์จาก base use case ไปยัง inclusion use case

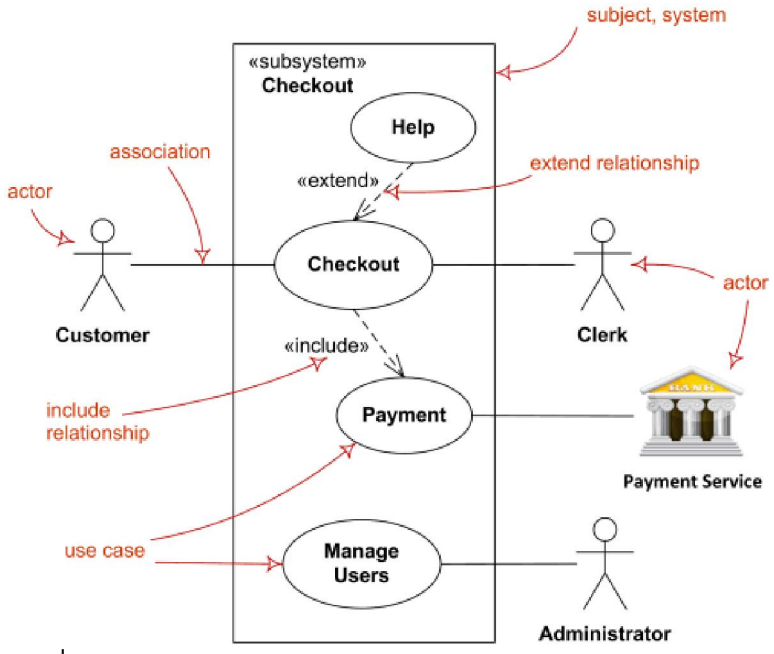
2.2.1 ความสัมพันธ์ระหว่าง use case

ความสัมพันธ์ของแต่ละ use case ภายในระบบ สามารถแบ่งออกได้ 2 แบบ คือ Extends และ Include

- ความสัมพันธ์แบบ Extend คือ use case หนึ่งมีผลต่อการทำงานตามปกติของอีก use case หนึ่งอย่างมีเงื่อนไข นั่นคือ use case ที่ถูก extend หรือ base use case จะมีกิจกรรมที่เปลี่ยนแปลงไปหากมีการระบุเงื่อนไขไปยัง use case ที่ extend
- ความสัมพันธ์แบบ Include คือ use case หนึ่ง เรียกใช้งานอีก use case หนึ่ง ลักษณะเช่นเดียวกันกับการเรียกใช้งาน program ย่อยโดย program หลัก

2.2.2 ตัวอย่าง

ตัวอย่างแผนภาพ use case สำหรับการ Checkout ซึ่งประกอบด้วยผู้ใช้งาน คือ ลูกค้า (Customer) เสมียน (Clerk) บริการการจ่ายเงิน (Payment Service) และผู้ดูแลระบบ (Administrator) โดยที่ผู้ดูแลระบบสามารถจัดการข้อมูลผู้ใช้งานระบบได้ ลูกค้าและเสมียนสามารถทำการ Checkout ได้ ซึ่งในการ Checkout จะต้องทำการชำระเงินผ่านบริการการจ่ายเงิน และในขณะทำการ Checkout หากต้องการความช่วยเหลือสามารถเรียกใช้ Help ได้ ดังแสดงในภาพประกอบที่ 2.2

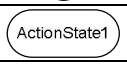
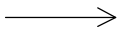
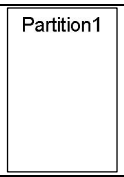
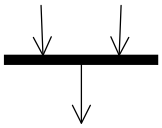
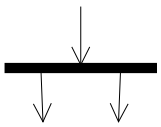


ภาพประกอบที่ 2.2 แผนภาพ use case สำหรับการ Checkout (Galen, *et al.,* 2010)

2.3 แผนภาพกิจกรรม (Activity Diagram)

แผนภาพกิจกรรม (Booch, *et al.,* 1998) เป็นแผนภาพที่ใช้ในการแสดงพฤติกรรมของซอฟต์แวร์ โดยอธิบายลำดับการทำงานที่เกิดขึ้นในซอฟต์แวร์ตั้งแต่เริ่มต้นจนสิ้นสุด สัญลักษณ์พื้นฐานของแผนภาพกิจกรรมแสดงดังตารางที่ 2.2

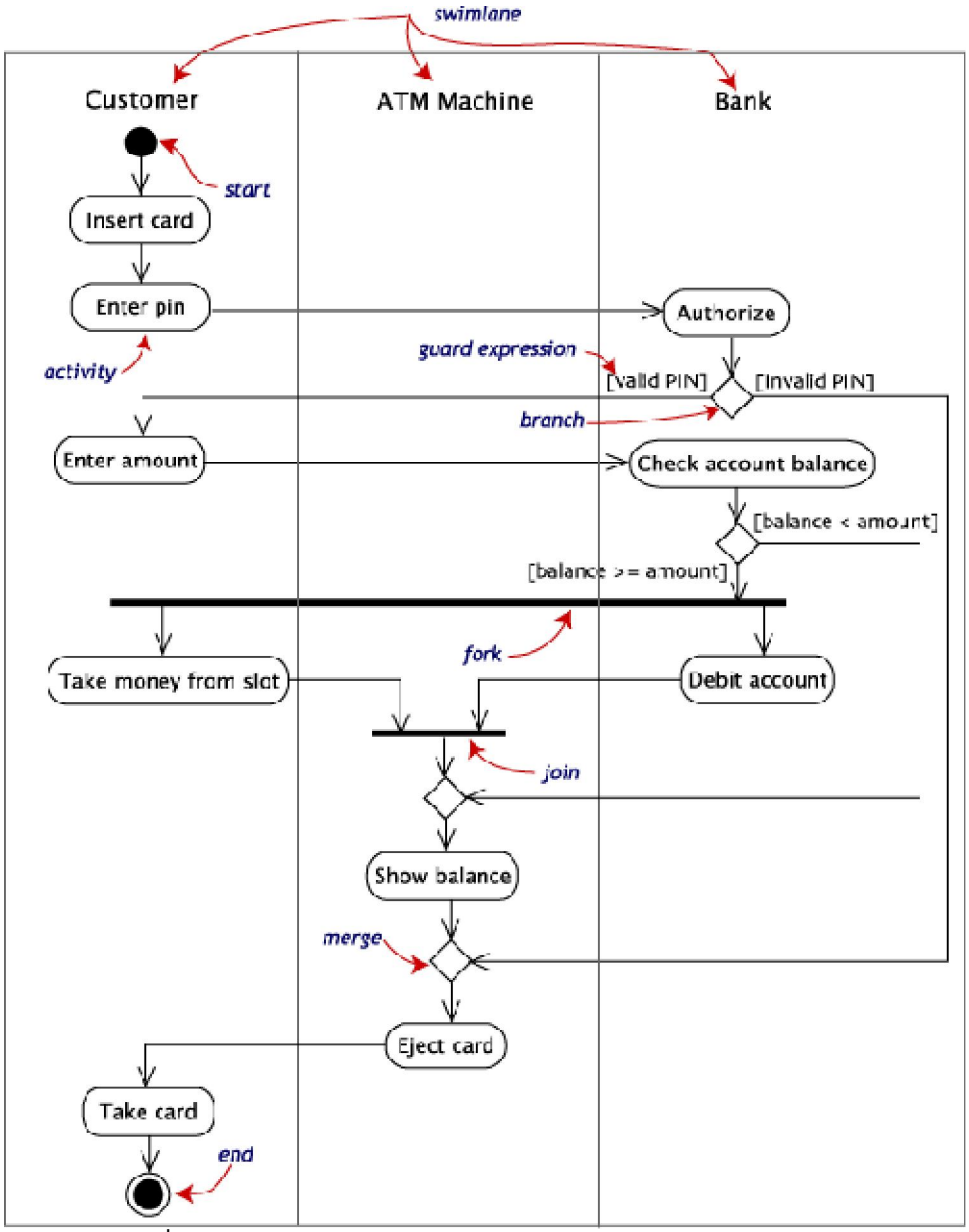
ตารางที่ 2.2 สัญลักษณ์พื้นฐานของแผนภาพกิจกรรม

สัญลักษณ์	ชื่อสัญลักษณ์	คำอธิบาย
	Start activity	จุดเริ่มต้น
	End activity	จุดสิ้นสุด
	Activity	กิจกรรมการทำงาน
	Branch, Merge	ตัวตัดสินใจหรือการรวมกันของการทำงาน - ตัวตัดสินใจ จะมีการระบุเงื่อนไขทางตรรกะ - การรวมกันของการทำงาน จะไม่มีการระบุเงื่อนไขทางตรรกะ
	Transition	การเปลี่ยนแปลงสถานะของกิจกรรม
	Swimlane	แบ่งส่วนหน้าที่การทำงาน
	Join	รวมการทำงานจากหลายกิจกรรม
	Fork	แยกการทำงานออกเป็นหลายกิจกรรม

2.3.1 ตัวอย่าง

ตัวอย่างแผนภาพกิจกรรมสำหรับการถอนเงินจากเครื่องถอนเงินอัตโนมัติ (Automatic Teller Machine: ATM) ซึ่งการทำงานแบ่งออกเป็น 3 ส่วน คือ ส่วนลูกค้า ส่วนของ ATM และส่วนธนาคาร โดยกระบวนการทำงานเริ่มต้นจากลูกค้าใส่บัตรและกรอกรหัส จากนั้นธนาคารทำการตรวจสอบความถูกต้องของรหัส ในกรณีที่รหัสไม่ถูกต้อง ธนาคารจะส่งคำสั่งไปที่ตู้ ATM ให้คืนบัตรให้กับลูกค้า แต่หากรหัสถูกต้อง ธนาคารให้ลูกค้ากรอกจำนวนเงินที่ต้องการ

ถอน เมื่อลูกค้ากรอกจำนวนเงินเรียบร้อยแล้ว ธนาคารจะทำการตรวจสอบยอดเงินที่ต้องการถอนกับยอดเงินคงเหลือในบัญชีของลูกค้า ในกรณีที่จำนวนเงินคงเหลือน้อยกว่าจำนวนเงินที่ต้องการถอน ธนาคารจะส่งคำสั่งไปที่ตู้ ATM ให้แสดงยอดเงินคงเหลือและคืนบัตรให้กับลูกค้า แต่หากจำนวนเงินคงเหลือมากกว่าจำนวนเงินที่ต้องการถอน ธนาคารจะอนุมัติการถอนเงินและทำการหักยอดบัญชีในขณะเดียวกัน หลังจากนั้นตู้ ATM จะแสดงยอดเงินคงเหลือและคืนบัตรให้กับลูกค้า ดังแสดงในภาพประกอบที่ 2.3



ภาพประกอบที่ 2.3 แผนภาพกิจกรรมสำหรับการถอนเงินจากตู้ ATM (Miller, 1994)

2.4 ไวยากรณ์บีเอ็นเอฟ

ไวยากรณ์บีเอ็นเอฟ (Backus Naur Form: BNF) (Sebesta, 2010) เป็น metalanguage ที่ใช้อธิบายไวยากรณ์ของภาษาการโปรแกรม ซึ่งถือเป็นส่วนสำคัญในการสร้าง compiler ไวยากรณ์บีเอ็นเอฟประกอบด้วยกฎต่าง ๆ ที่อยู่ในรูปโปรดักชัน (Production) โดยแต่ละโปรดักชันมีรูปแบบดังนี้

< nonterminal> ::= sequence of terminal or <nonterminal>

แต่ละโปรดักชันจะประกอบด้วยสัญลักษณ์ที่อยู่ทางด้านซ้าย (Left-Hand Side: LHS) และสัญลักษณ์ที่อยู่ทางด้านขวา (Right-Hand Side: RHS) ของสัญลักษณ์ ::= (defined as) โดยที่สัญลักษณ์ด้านซ้ายจะเป็นสัญลักษณ์ไม่สิ้นสุด (Nonterminal Symbol) นั่นคือสัญลักษณ์ที่อยู่ในเครื่องหมาย < และ > ส่วนสัญลักษณ์ด้านขวาจะประกอบด้วยลำดับของสัญลักษณ์สิ้นสุด (Terminal) หรือสัญลักษณ์ไม่สิ้นสุดก็ได้ ซึ่งสัญลักษณ์ด้านซ้ายจะถูกสร้างเป็นสัญลักษณ์ด้านขวา หากสัญลักษณ์ไม่สิ้นสุดด้านซ้ายนั้นสามารถสร้างได้มากกว่า 1 รูปแบบจะแยกแต่ละรูปแบบด้วยสัญลักษณ์หรือ (|) ภาพประกอบที่ 2.4 แสดงตัวอย่างไวยากรณ์บีเอ็นเอฟ

```
< number> ::= <digit> | <number><digit>
<digit>      ::=  0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

ภาพประกอบที่ 2.4 ตัวอย่างไวยากรณ์บีเอ็นเอฟ

จากตัวอย่างในภาพประกอบที่ 2.4 ประกอบด้วย 2 โปรดักชัน โปรดักชันแรกประกอบด้วยสัญลักษณ์ด้านซ้ายคือ สัญลักษณ์ไม่สิ้นสุด number นิยามเป็นด้านขวาคือ สัญลักษณ์ไม่สิ้นสุด digit หรือสัญลักษณ์ไม่สิ้นสุด number ตามด้วยสัญลักษณ์ไม่สิ้นสุด digit โปรดักชันที่สองประกอบด้วยสัญลักษณ์ด้านซ้ายคือ สัญลักษณ์ไม่สิ้นสุด digit นิยามด้วยสัญลักษณ์สิ้นสุด 0 หรือ 1 หรือ 2 หรือ 3 หรือ 4 หรือ 5 หรือ 6 หรือ 7 หรือ 8 หรือ 9 ตัวอย่างการนิยามสัญลักษณ์ไม่สิ้นสุด number แสดงดังตารางที่ 2.3

ตารางที่ 2.3 ตัวอย่างการนิยามสัญลักษณ์ไม่สิ้นสุด number

ตัวอย่างที่ 1	ตัวอย่างที่ 2
< number> ::= <digit> ::= 5	< number> ::= <number><digit> ::= <digit><digit> ::= 8<digit> ::= 82

2.5 Mutation Testing

Mutation Testing (Ammann and Offutt, 2008) เป็นวิธีการหนึ่งในการทดสอบซอฟต์แวร์ โดยปรับเปลี่ยนโปรแกรมให้ต่างไปจากโปรแกรมต้นฉบับ โปรแกรมที่ได้จากการปรับเปลี่ยนนี้เรียกว่าโปรแกรม mutant ในการปรับเปลี่ยนโปรแกรมแต่ละครั้งจะทำเพียง 1 ตำแหน่งเพื่อสร้าง 1 โปรแกรม mutant โดยการปรับเปลี่ยนจะถูกกระทำตามตัวดำเนินการที่เรียกว่า mutation operator ตัวอย่างตัวดำเนินการ เช่น ตัวดำเนินการการแทนที่ ตัวอย่างโปรแกรมต้นฉบับและโปรแกรม mutant แสดงดังตารางที่ 2.4

ตารางที่ 2.4 ตัวอย่างโปรแกรมต้นฉบับและโปรแกรม mutant

โปรแกรมต้นฉบับ	โปรแกรม Mutant
sameValue() { int A = 1; int B = 1; if ((A - B)==0) { printf(“Yes!!!!”); } }	sameValue() { int A = 1; int B = 1; Δ if ((A + B)==0) { printf(“Yes!!!!”); } }

จากตัวอย่าง โปรแกรมต้นฉบับสามารถสร้างโปรแกรม mutant ด้วยตัวดำเนินการแทนที่โดยการแทนที่เครื่องหมายลบด้วยเครื่องหมายบวก

หลังจากได้โปรแกรม mutant จะทำการทดสอบโปรแกรมต้นฉบับและโปรแกรม mutant ด้วยกรณีทดสอบชุดเดียวกัน เพื่อเปรียบเทียบผลลัพธ์ที่ได้ หากผลลัพธ์ที่ได้จาก

โปรแกรมต้นฉบับเหมือนกับผลลัพธ์ที่ได้จากโปรแกรม mutant เรียกว่า live mutant แต่หากผลลัพธ์ที่ได้จากโปรแกรมต้นฉบับไม่เหมือนกับผลลัพธ์ที่ได้จากโปรแกรม mutant หรือกล่าวได้ว่าโปรแกรม mutant นี้ถูกฆ่า (Killed Mutant) จะเรียก mutant นั้นว่า dead mutant ซึ่งแสดงว่ากรณีทดสอบสามารถตรวจจับข้อผิดพลาดในโปรแกรม mutant ได้ โดยประสิทธิภาพของกรณีทดสอบดูได้จากค่า Mutation Score ซึ่งเป็นอัตราส่วนของจำนวน dead mutant ต่อจำนวนโปรแกรม mutant ลบด้วยจำนวน equivalent mutant หากค่า Mutation score เข้าใกล้ 1 หมายถึงกรณีทดสอบนั้นมีประสิทธิภาพ โดย equivalent mutant จะหมายถึงโปรแกรม mutant ที่ไม่สามารถหากรณีทดสอบใดๆมาฆ่าได้ ค่า Mutation Score คำนวณได้จากสูตรด้านล่างนี้

$$\text{Mutation score} = \frac{\#\text{dead mutants}}{\#\text{total mutants} - \#\text{equivalent mutants}}$$

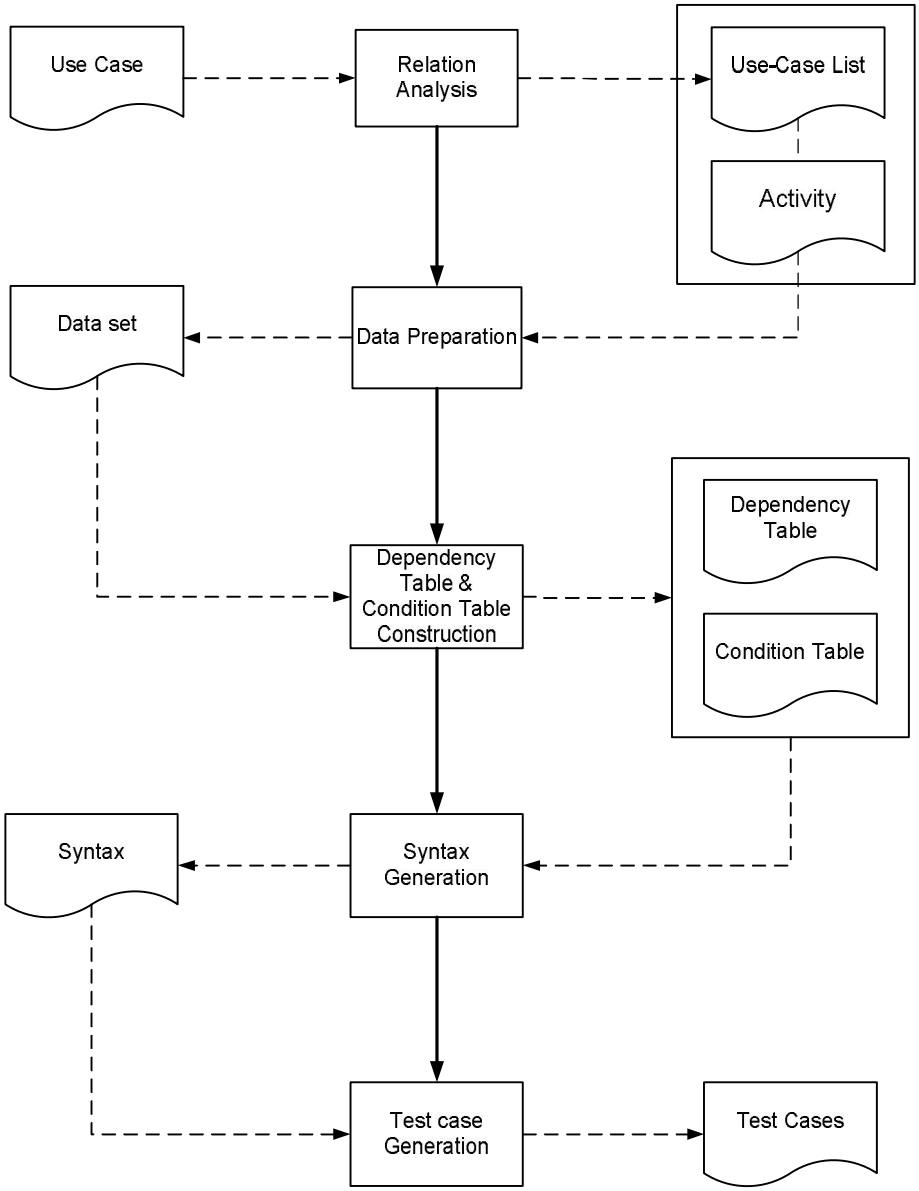
บทที่ 3

การสร้างกรณีทดสอบจากแผนภาพ use case และแผนภาพกิจกรรม บนพื้นฐานของไวยากรณ์

เนื้อหาในบทนี้กล่าวถึงการสร้างกรณีทดสอบจากแผนภาพ use case และแผนภาพกิจกรรมบนพื้นฐานของไวยากรณ์ ซึ่งส่วนแรกกล่าวถึงสถาปัตยกรรมของวิธีการที่นำเสนอ ส่วนที่สองกล่าวถึงรายละเอียดของขั้นตอนต่าง ๆในการสร้างกรณีทดสอบพร้อมตัวอย่าง และส่วนที่สามกล่าวถึงการประเมินประสิทธิภาพของกรณีทดสอบ

3.1 สถาปัตยกรรม

สถาปัตยกรรมที่นำเสนอเป็นการสร้างกรณีทดสอบโดยการวิเคราะห์แผนภาพ use case เพื่อสร้างรายการ use-case ทั้งหมดที่จะต้องนำมาดำเนินการในการทดสอบระบบ จากนั้นจึงนำแผนภาพกิจกรรมที่สอดคล้องกับแต่ละ use case ในรายการ use-case มาเตรียมชุดข้อมูล (Data Set) เบื้องต้นต่อจากนั้นจะสร้างตารางการขึ้นต่อกัน (Dependency Table) และตารางเงื่อนไขทางตรรกะ (Condition Table) จากชุดข้อมูลเบื้องต้น จากนั้นจึงนำทั้งสองตารางมาพิจารณาร่วมกันเพื่อสร้างไวยากรณ์ และใช้ไวยากรณ์นี้สร้างกรณีทดสอบจากไวยากรณ์ สถาปัตยกรรมของวิธีการที่นำเสนอแสดงดังภาพประกอบที่ 3.1



ภาพประกอบที่ 3.1 สถาปัตยกรรมของวิธีการที่นำเสนอ

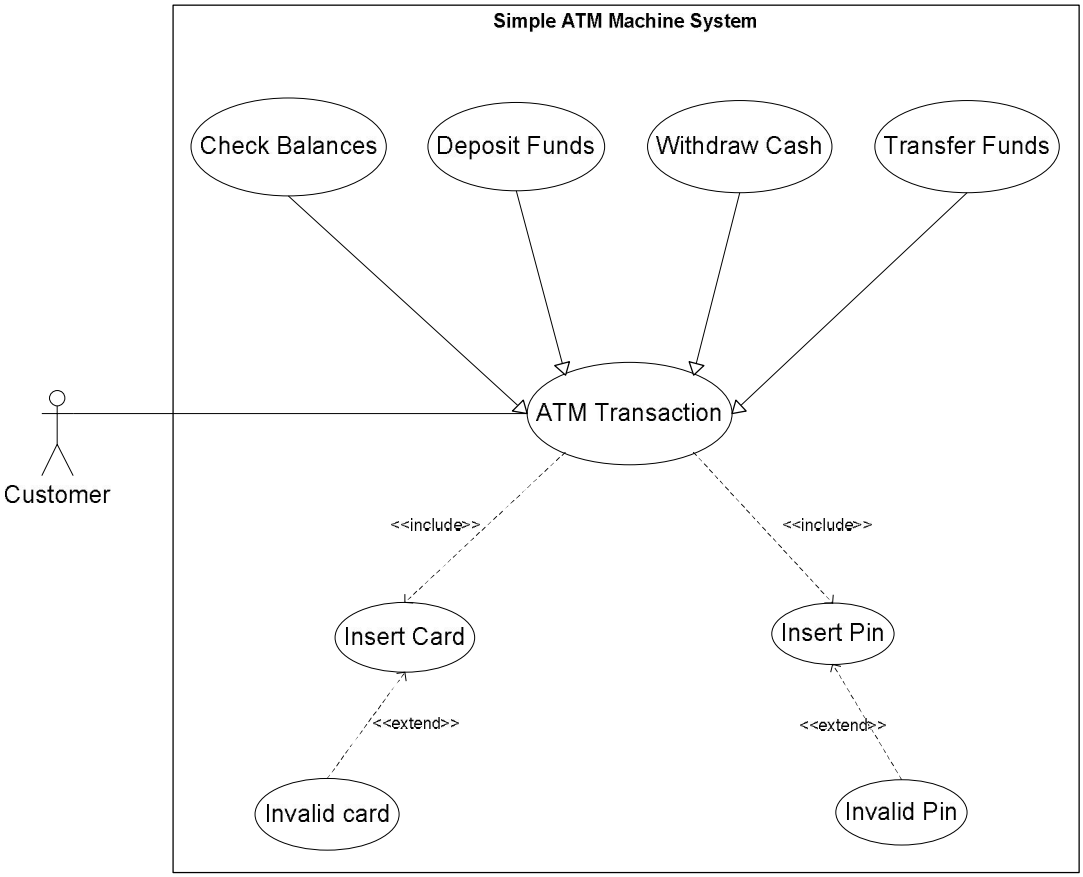
3.2 ขั้นตอนต่าง ๆในการสร้างกรณีทดสอบ

การสร้างกรณีทดสอบจากแผนภาพ use case และแผนภาพกิจกรรมบนพื้นฐานของไวยากรณ์ มีขั้นตอนและรายละเอียดดังต่อไปนี้

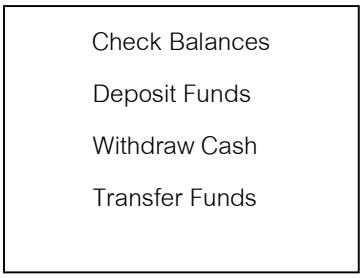
ขั้นตอนที่ 1 วิเคราะห์ความสัมพันธ์ (Relation Analysis)

ทำการวิเคราะห์ความสัมพันธ์จากแผนภาพ use case ในการวิเคราะห์ความสัมพันธ์จะพิจารณาทุกๆ actor ในระบบ โดยพิจารณาแต่ละ actor แล้วทำการระบุชื่อ use case ที่มีความเกี่ยวข้องกับ actor นั้นลงในรายการ use-case ซึ่งแบ่งการพิจารณาออกเป็น 3 กรณี ดังนี้

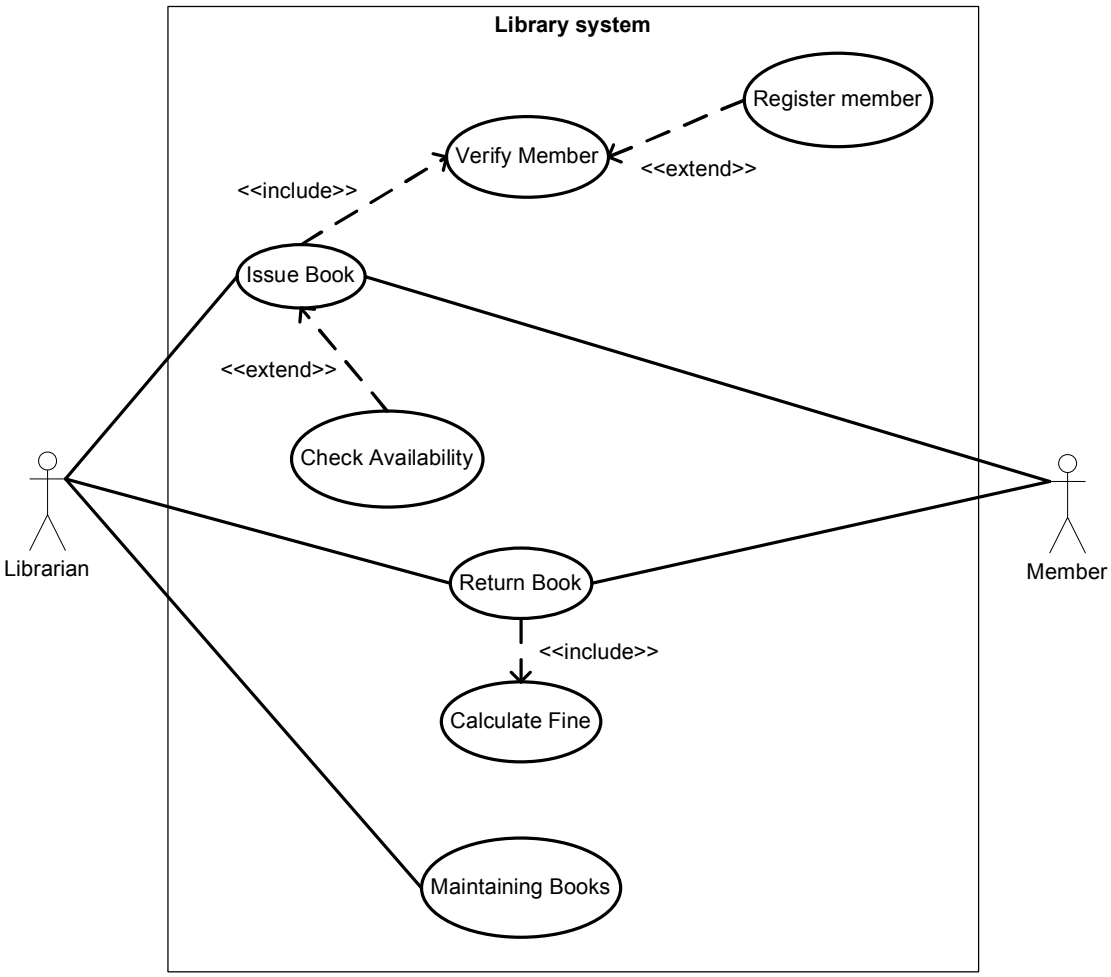
กรณีที่ 1: กรณีที่ use case นั้นเป็น abstract use case ที่มีความสัมพันธ์เป็น generalizations จะนำแต่ละ use case ที่มีลักษณะเป็น generalizations นั้น แทนที่ในตำแหน่งของ abstract use case แล้วจึงระบุชื่อของ use case เหล่านั้นลงในรายการ use-case จากภาพประกอบที่ 3.2 พบว่า ATM Transaction เป็น abstract use case และมีลักษณะความสัมพันธ์แบบ generalizations กับ 4 use case นั่นคือ Check Balances use case, Deposit Funds use case, Withdraw Cash use case และ Transfer Funds use case ดังนั้นในการระบุชื่อ use case ลงในรายการ use-case จะทำการระบุชื่อทั้ง 4 use case ข้างต้นแทนที่ ATM Transaction use case ดังรายการ use-case ในภาพประกอบที่ 3.3



ภาพประกอบที่ 3.2 แผนภาพ use case สำหรับระบบ ATM (MCA, 2010)



ภาพประกอบที่ 3.3 รายการ use-case สำหรับระบบ ATM

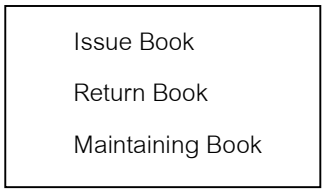


ภาพประกอบที่ 3.4 แผนภาพ use case สำหรับระบบห้องสมุด (MCA, 2010)

กรณีที่ 2: กรณีที่ use case นั้นมีความสัมพันธ์กับ actor มากกว่า 1 actor จะทำการระบุชื่อ use case นั้นลงในรายการ use-case โดยจะกระทำเพียง 1 ครั้งเท่านั้น ตัวอย่างจากภาพประกอบที่ 3.4 พบว่า Issue Book และ Return Book เป็น use case ที่มีความสัมพันธ์กับ actor มากกว่า 1 actor นั่นคือ Librarian และ Member ดังนั้นในการระบุชื่อ Issue Book use case และ Return Book use case จะถูกกระทำเพียงครั้งเดียวเท่านั้น

กรณีที่ 3: กรณีอื่นๆที่ไม่ตรงกับกรณีที่ 1 และ กรณีที่ 2 ที่กล่าวข้างต้นสามารถระบุชื่อของ use case นั้น ลงในรายการ use-case ได้ทันที ตัวอย่างจากภาพประกอบที่ 3.4 พบว่า Maintaining Book use case เป็น use case ที่ไม่เป็นกรณีที่ 1 และ 2 ดังนั้นจึงสามารถระบุชื่อ Maintaining Book ลงในรายการ use-case ได้ทันที

จากภาพประกอบที่ 3.4 พบว่าระบบห้องสมุดประกอบด้วย 2 actor คือ Librarian และ Member เมื่อทำการวิเคราะห์ความสัมพันธ์ตามกรณีข้างต้นแล้ว จะได้รายการ use-case ดังแสดงในภาพประกอบที่ 3.5



ภาพประกอบที่ 3.5 รายการ use-case สำหรับระบบห้องสมุด

รายการ use-case ที่ได้จากการวิเคราะห์ความสัมพันธ์ข้างต้นแสดงให้เห็นถึง use case ที่ต้องพิจารณาเพื่อทำการทดสอบระบบ โดยแต่ละ use case ในรายการ use-case จะถูกนำไปกระทำตามขั้นตอนที่ 2-5 เพื่อสร้างกรณีทดสอบสำหรับ use case นั้น ซึ่งกรณีทดสอบทั้งหมดของทุก use case รวมกันจะเป็นกรณีทดสอบที่ใช้ในการทดสอบระบบ

ขั้นตอนที่ 2 เตรียมข้อมูล (Data Preparation)

ในขั้นตอนการเตรียมข้อมูล รายการ use-case จะถูกนำมาพิจารณาเพื่อหาแผนภาพกิจกรรมที่สอดคล้องกับ use case แต่ละรายการโดยแต่ละแผนภาพกิจกรรมจะถูกนำมาใช้เพื่อเตรียมข้อมูล ซึ่งผลลัพธ์ที่ได้คือชุดข้อมูล (Data Set) เบื้องต้น 1 ชุด ขั้นตอนการเตรียมข้อมูล เริ่มจากการกำหนดชื่อสัญลักษณ์ (Symbol Name) ให้กับองค์ประกอบต่างๆ ในแผนภาพกิจกรรม โดยใช้ตัวอักษรภาษาอังกฤษพิมพ์ใหญ่เรียงตามลำดับ ต่อจากนั้น จะรวบรวมข้อมูล โดยแยกข้อมูลออกเป็น 10 ชุดข้อมูลย่อยดังต่อไปนี้

ชุดข้อมูลย่อยที่ 1: ชุดข้อมูลจุดเริ่มต้น (Start) เก็บ symbol name ขององค์ประกอบที่เป็น start activity

ชุดข้อมูลย่อยที่ 2: ชุดข้อมูลกิจกรรม (Activity) เก็บ symbol name ขององค์ประกอบที่เป็น activity ทั้งหมด (ยกเว้น start activity และ end activity) และคั่นแต่ละ symbol name ด้วยเครื่องหมายจุลภาค (,)

ชุดข้อมูลย่อยที่ 3: ชุดข้อมูลเส้นการเปลี่ยนแปลงสถานะของกิจกรรม (Transition) เก็บ symbol name ของ transition ทั้งหมด ในลักษณะของชุด transition ซึ่งแต่ละชุดจะประกอบด้วย symbol name ขององค์ประกอบที่หัวลูกศรของเส้น transition ตามด้วยเครื่องหมายทวิภาค (:) จากนั้นตามด้วย symbol name ขององค์ประกอบที่ปลายหัวลูกศรของเส้น transition และคั่นแต่ละชุด transition ด้วยเครื่องหมายจุลภาค

ชุดข้อมูลย่อยที่ 4: ชุดข้อมูลตัวตัดสินใจ (Branch) เก็บ symbol name ขององค์ประกอบที่เป็น branch ทั้งหมด และคั่นแต่ละ symbol name ด้วยเครื่องหมายจุลภาค

ชุดข้อมูลย่อยที่ 5: ชุดข้อมูลเงื่อนไขทางตรรกะ (Condition) เก็บเงื่อนไขทางตรรกะบนเส้น transition ที่ออกจาก branch ทั้งหมด ในลักษณะของชุด condition ซึ่งแต่ละชุดจะประกอบด้วย symbol name ของ branch นั้น ตามด้วย เครื่องหมายทวิภาค จากนั้นตามด้วยเงื่อนไขบนเส้น transition และเครื่องหมายทวิภาค สุดท้ายตามด้วย symbol name ขององค์ประกอบที่ปลายเส้น transition และคั่นแต่ละชุด condition ด้วยเครื่องหมายจุลภาค

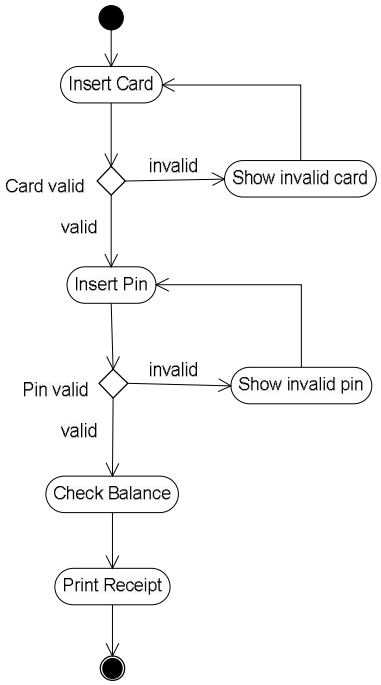
ชุดข้อมูลย่อยที่ 6: ชุดข้อมูลการรวมกันของการทำงาน (Merge) เก็บ symbol name ขององค์ประกอบที่เป็น merge ทั้งหมด และคั่นแต่ละ symbol name ด้วยเครื่องหมายจุลภาค

ชุดข้อมูลย่อยที่ 7: ชุดข้อมูลองค์ประกอบที่ทำให้เกิดการวนซ้ำ (Loop) เก็บ symbol name ขององค์ประกอบที่ปลายลูกศรของเส้น transition ที่ทำให้เกิดการวนซ้ำทั้งหมด และคั่นแต่ละ symbol name ด้วยเครื่องหมายจุลภาค

ชุดข้อมูลย่อยที่ 8: ชุดข้อมูลของการแยกการทำงานออกเป็นหลายการทำงาน (Fork) เก็บ symbol name ขององค์ประกอบที่เป็น fork ทั้งหมด และคั่นแต่ละ symbol name ด้วยเครื่องหมายจุลภาค

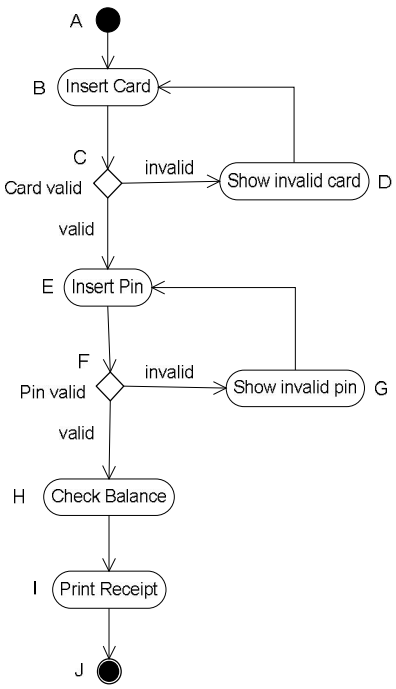
ชุดข้อมูลย่อยที่ 9: ชุดข้อมูลของการรวมการทำงานจากหลายการทำงาน (Join) เก็บ symbol name ขององค์ประกอบที่เป็น join ทั้งหมด และคั่นแต่ละ symbol name ด้วยเครื่องหมายจุลภาค

ชุดข้อมูลย่อยที่ 10: ชุดข้อมูลจุดสิ้นสุด (End) เก็บ symbol name ขององค์ประกอบที่เป็น end activity



ภาพประกอบที่ 3.6 แผนภาพกิจกรรมที่สอดคล้องกับ Check Balance use case (MCA, 2010)

ตัวอย่างการเตรียมชุดข้อมูลเบื้องต้น สำหรับแผนภาพกิจกรรม จากภาพประกอบที่ 3.6 โดยเริ่มจากกำหนดชื่อสัญลักษณ์ (Symbol Name) ให้กับทุกองค์ประกอบของแผนภาพกิจกรรม ได้ดังภาพประกอบที่ 3.7



ภาพประกอบที่ 3.7 symbol name สำหรับแผนภาพกิจกรรม Check Balance

จากภาพประกอบที่ 3.7 สามารถรวบรวมชุดข้อมูลเบื้องต้นดังภาพประกอบที่ 3.8

Check Balance
Start = {A}
Activity = {B,D,E,G,H,I}
Transition = {A:B,B:C,C:D,D:B,C:E,E:F,F:G,G:E,F:H,H:I,I:J}
Branch = {C,F}
Condition = {C:invalid:D,C:valid:E,F:invalid:G,F:valid:H}
Merge = { }
Loop = {D,G}
Fork = { }
Join = { }
End = {J}

ภาพประกอบที่ 3.8 ชุดข้อมูลเบื้องต้นสำหรับแผนภาพกิจกรรม Check Balance

ขั้นตอนที่ 3 สร้างตารางการขึ้นต่อกันและตารางเงื่อนไขทางตรรกะ (Dependency Table and Condition Table Construction)

- ขั้นตอนการสร้างตารางการขึ้นต่อกัน

การสร้างตารางการขึ้นต่อกัน จะพิจารณาจากชุดข้อมูลเบื้องต้นที่จัดเตรียมไว้ในขั้นตอนที่ 2 การสร้างตารางการขึ้นต่อกันกระทำตามขั้นตอนด้านล่างนี้

```
Input:
    Start
    Activity[1..m]
    Branch[1..n]
    Merge[1..o]
    Loop[1..p]
    Fork[1..q]
    Join[1..r]
    Transition[1..s]
    End
Create Dependency Table
    Point = Start
    For i=1 to s
        For j=0 to 1
            Point = Transition[i+1]
            For l=1 to p
                If Point = Loop[p] then
                For a=1 to m
                        If Point = Activity[m] then save data loop Activity[m]
                        For b=1 to n
                            If Point = Branch[n] then save data loop Branch[n]
```

```
                    For m=1 to o
                        If Point = Merge[o] then save data loop Merge[o]
                        For f=1 to q
                            If Point = Fork[q] then save data loop Fork[q]
                            For j=1 to r
                                If Point = Join[r] then save data loop Join[r]
Else For a=1 to m
        If Point = Activity[m] then save data Activity[m]
        For b=1 to n
            If Point = Branch[n] then save data Branch[n]
            For m=1 to o
                If Point = Merge[o] then save data Merge[o]
                For f=1 to q
                    If Point = Fork[q] then save data Fork[q]
                    For j=1 to r
                        If Point = Join[r] then save data Join[r]
                        if Point = End then save data End
```

ตารางการขึ้นต่อกันที่ได้จากขั้นตอนข้างต้นประกอบด้วย 5 คอลัมน์ต่อไปนี้

- Defined Name: ชื่อที่กำหนดให้กับกิจกรรม (Activity) ต่างๆของชุดข้อมูลกิจกรรม โดยใช้ตัวอักษรภาษาอังกฤษพิมพ์เล็ก
- Symbol Name: ชื่อสัญลักษณ์ที่กำหนดให้กับองค์ประกอบต่างๆ ตามที่ระบุไว้ในข้อมูลเบื้องต้น
- Symbol Type: ประเภทขององค์ประกอบต่างๆ ตามที่ระบุไว้ในชุดข้อมูลเบื้องต้น
- Loop Type: ระบุว่าเป็นจุดที่ทำให้เกิดการวนซ้ำ โดยการระบุคำว่า “Loop”
- Dependency: symbol name ที่มีผลต่อองค์ประกอบที่กำลังพิจารณาอยู่

ตารางการขึ้นต่อกันสำหรับแผนภาพกิจกรรม Check Balance แสดงดังตารางที่ 3.1

ตารางที่ 3.1 ตารางการขึ้นต่อกันสำหรับ Check Balance

Defined Name	Symbol Name	Symbol Type	Loop Type	Dependency
b	B	Activity	-	A
-	C	Branch	-	B
d	D	Activity	Loop	C
b	B	Activity	-	D
e	E	Activity	-	C
-	F	Branch	-	E
g	G	Activity	Loop	F
e	E	Activity	-	G
h	H	Branch	-	F
i	I	Activity	-	H
-	J	End	-	I

- ขั้นตอนการสร้างตารางเงื่อนไขทางตรรกะ

การสร้างตารางเงื่อนไขทางตรรกะ จะพิจารณาจากชุดข้อมูลเบื้องต้นเฉพาะชุดข้อมูลย่อย condition ดังขั้นตอนวิธีด้านล่างนี้

```
Input: Condition[1..m]
Create Condition Table
for i=1 to m
    for j=1 to 3
        Save data condition[j]
```

ตารางเงื่อนไขทางตรรกะที่ได้จากขั้นตอนข้างต้น ประกอบด้วย 3 คอลัมน์ต่อไปนี้

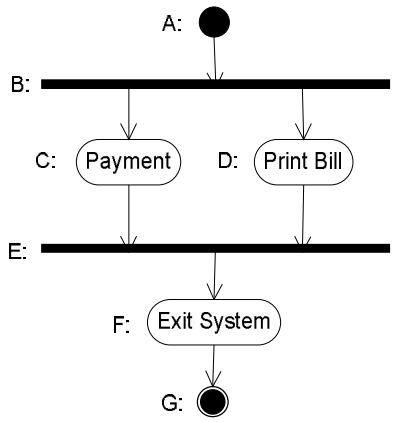
- Source: symbol name ขององค์ประกอบต้นทาง
- Condition: เงื่อนไขทางตรรกะ
- Destination: symbol name ขององค์ประกอบปลายทาง

ตารางเงื่อนไขทางตรรกะสำหรับแผนภาพกิจกรรม Check Balance แสดงดังตารางที่ 3.2

ตารางที่ 3.2 ตารางเงื่อนไขทางตรรกะสำหรับ Check Balance

Source	Condition	Destination
C	invalid	D
C	valid	E
F	invalid	G
F	valid	H

ตัวอย่างแผนภาพกิจกรรมที่ใช้สำหรับพิจารณาสร้างตารางการขึ้นต่อกันของ fork และ join แสดงดังภาพประกอบที่ 3.9 ซึ่งจะได้ ตารางการขึ้นต่อกันของ fork และ join แสดงดังตารางที่ 3.3



ภาพประกอบที่ 3.9 แผนภาพกิจกรรมตัวอย่าง fork และ join

ตารางที่ 3.3 ตัวอย่างตารางการขึ้นต่อกันของ fork และ join

Defined Name	Symbol Name	Symbol Type	Loop type	Dependency
-	B	Fork	-	A
c	C	Activity	-	B
d	D	Activity	-	B
-	E	Join	-	C
-	E	Join	-	D
f	F	Activity	-	E
-	G	End	-	F

ขั้นตอนที่ 4 สร้างไวยากรณ์ (Syntax Generation)

ไวยากรณ์จะถูกสร้างขึ้นโดยพิจารณาตารางการขึ้นต่อกันและตารางเงื่อนไขทางตรรกะ โดยไวยากรณ์จะประกอบด้วยโปรดักชั่นต่างๆ แต่ละโปรดักชั่นแบ่งออกเป็น 2 ส่วนคือ ส่วนซ้าย (Left Hand Side: LHS) และส่วนขวา (Right Hand Side: RHS) ของสัญลักษณ์ ::= (Defined as) การสร้างไวยากรณ์แบ่งออกเป็น 2 ขั้นตอนหลักๆ ดังนี้

ขั้นตอนที่ 1 การสร้างโปรดักชั่นแรก จะสร้างจากแถวแรกของตารางการขึ้นต่อกัน โดยแทนส่วน LHS ด้วยค่าในคอลัมน์ Dependency ภายในเครื่องหมาย < และ > และแทนส่วน RHS ด้วยค่าในคอลัมน์ Symbol Name ภายในเครื่องหมาย < และ >

ขั้นตอนที่ 2 การสร้างโปรดักชั่นถัดไป จะสร้างจากกำหนดแถวถัดไปที่ยังไม่ถูกพิจารณา เป็นแถว “กำลังพิจารณา” และค้นหาแถวอื่นๆที่มีค่าในคอลัมน์ Dependency ตรงกับค่าในคอลัมน์ Dependency ของแถวที่กำลังพิจารณา หากพบจะกำหนดให้แถวที่ค้นหาได้เป็นแถว “กำลังพิจารณา” ด้วย ตัวอย่างจากตารางที่ 3.3 กำหนดให้แถวที่สอง เป็นแถวที่กำลังพิจารณาและมีค่าในคอลัมน์ Dependency เป็น B ซึ่งจะพบว่าแถวที่สาม มีค่าในคอลัมน์ Dependency เป็น B เช่นกัน ดังนั้น แถวที่สามจะถูกกำหนดให้เป็นแถวกำลังพิจารณาพร้อมกับแถวที่สอง

ขั้นตอนที่ 2.1 พิจารณาค่าในคอลัมน์ Symbol Type ของแถวที่กำลังพิจารณา โดยแบ่งออกเป็น 3 กรณี ดังนี้

กรณีที่ 1 ค่าในคอลัมน์ Symbol Type เป็น End จะกระทำตามขั้นตอนที่ 2.2 และขั้นตอนที่ 2.5 และหยุดกระบวนการสร้างไวยากรณ์

กรณีที่ 2 ค่าในคอลัมน์ Symbol Type เป็น Join จะกำหนดสัญลักษณ์ไม่สิ้นสุดเป็น { และ }

กรณีที่ 3 ค่าในคอลัมน์ Symbol Type อื่นๆ จะกำหนดสัญลักษณ์ไม่สิ้นสุดเป็น < และ >

ขั้นตอนที่ 2.2 สร้าง LHS ของโปรดักชั่น โดยแทนด้วยค่าในคอลัมน์ Dependency ของแถวที่กำลังพิจารณา ภายในเครื่องหมาย < และ >

ขั้นตอนที่ 2.3 สร้าง RHS ของโปรดักชั่น โดยค้นหาแถวที่มีค่าในคอลัมน์ Symbol Name ตรงกับค่าในคอลัมน์ Dependency ของแถวที่กำลังพิจารณา และนำค่าในคอลัมน์ Symbol Type ของแถวที่ค้นได้มาพิจารณา โดยแบ่งออกเป็น 5 ประเภทดังต่อไปนี้

- Activity: แทนส่วน RHS ด้วยชื่อของกิจกรรมที่กำหนดไว้ในคอลัมน์ Defined Name ของแถวที่ค้นหาได้ตามด้วยค่าในคอลัมน์ Symbol Name ของแถวที่กำลังพิจารณา ภายในเครื่องหมายสัญลักษณ์ไม่สิ้นสุดที่กำหนดไว้ ตัวอย่าง จากตารางที่ 3.1 กำหนดให้แถวที่กำลังพิจารณาคือแถวที่ 2 พบว่าค่าในคอลัมน์ Dependency เท่ากับ B และ ค่า

ในคอลัมน์ Symbol Type เป็น Branch เมื่อพิจารณาคอลัมน์ Symbol Name ที่มีค่าเท่ากับ B พบว่า Symbol Type เป็น Activity ดังนั้นจะได้ RHS เป็น b<C>

- Branch: แทนที่ส่วน RHS โดยแบ่งการพิจารณาออกเป็น 2 ขั้นตอน ดังนี้

ขั้นตอนที่ 1: พิจารณาตารางเงื่อนไขทางตรรกะที่ค่าในคอลัมน์ Source ตรงกับค่าในคอลัมน์ Dependency ของแถวที่กำลังพิจารณา เพื่อระบุแถวต่าง ๆในตารางเงื่อนไขทางตรรกะที่ต้องดำเนินการ โดยในแต่ละแถวที่ต้องดำเนินการจะถูกดำเนินการตามขั้นตอนที่ 2

ขั้นตอนที่ 2: นำค่าในคอลัมน์ Destination ไปตรวจสอบกับค่าในคอลัมน์ Loop Type ในตารางการขึ้นต่อกัน ซึ่งแบ่งออกเป็น 2 กรณี

กรณีที่ 1 : เป็นประเภท Loop จะแทนส่วน RHS ด้วยค่าในคอลัมน์ Condition ตามด้วยค่าในคอลัมน์ Destination จากตารางเงื่อนไขทางตรรกะภายในเครื่องหมาย [และ]

กรณีที่ 2 : ไม่เป็นประเภท Loop จะแทนส่วน RHS ด้วยค่าในคอลัมน์ Condition ตามด้วยค่าในคอลัมน์ Destination จากตารางเงื่อนไขทางตรรกะภายในเครื่องหมายสัญลักษณ์ไม่สิ้นสุดที่กำหนดไว้

เมื่อแทนแต่ละแถวที่ต้องดำเนินการตามขั้นตอนที่ 2 เรียบร้อยแล้ว จะนำ RHS ของแต่ละแถวที่แทนได้เหล่านั้นมาเรียงกัน และคั่นด้วยสัญลักษณ์หรือ (|) ตัวอย่างจากตารางที่ 3.1 กำหนดให้แถวที่กำลังพิจารณาคือแถวที่ 3 พบว่าค่าในคอลัมน์ Dependency เท่ากับ C และค่าในคอลัมน์ Symbol Type เป็น Activity เมื่อพิจารณาคอลัมน์ Symbol Name ที่มีค่าเท่ากับ C พบว่า Symbol Type เป็น Branch จึงทำการตรวจสอบตามขั้นตอนที่ 1 พบว่ามี 2 แถวในตารางเงื่อนไขทางตรรกะที่ต้องดำเนินการ คือ แถวที่ 1 และ 2 จากนั้นนำแถวที่ 1 และ แถวที่ 2 ไปดำเนินการตามขั้นตอนที่ 2 ซึ่งมีค่าในคอลัมน์ Destination คือ D และ E ตามลำดับ เมื่อตรวจสอบพบว่า D มีการระบุ Loop ในคอลัมน์ Loop Type ดังนั้น จะเข้ากรณีที่ 1 ทำให้ได้ RHS เป็น invalid[D] ส่วน E ไม่มีการระบุ Loop ในคอลัมน์ Loop Type จะเข้ากรณีที่ 2 ทำให้ได้ RHS เป็น valid<E> ดังนั้นสุดท้ายจะได้ RHS เป็น invalid[D] | valid<E>

- Merge: แทนที่ส่วน RHS ด้วย ค่าในคอลัมน์ Symbol Name ของแถวที่กำลังพิจารณา ภายในเครื่องหมายสัญลักษณ์ไม่สิ้นสุดที่กำหนดไว้
- Fork: แทนที่ส่วน RHS ด้วยค่าในคอลัมน์ Symbol Name ของแถวที่กำลังพิจารณา ภายในเครื่องหมายสัญลักษณ์ไม่สิ้นสุดที่กำหนดไว้ ในกรณีที่ตรวจพบมากกว่า 1 แถว จะนำการแทนแต่ละแถวมาเรียงต่อกันจนครบทุกแถว โดยเรียงลำดับใน 2 ลักษณะ คือ

เรียงจากซ้ายไปขวา และ เรียงจากขวาไปซ้าย และคั่นระหว่างการเรียง 2 ลักษณะด้วยสัญลักษณ์หรือ (|) พิจารณาตารางที่ 3.3 ตารางการขึ้นต่อกันสำหรับแผนภาพกิจกรรมตัวอย่างสำหรับ fork และ join ดังแสดงในภาพประกอบที่ 3.9 จากตารางที่ 3.3 กำหนดให้แถวที่กำลังพิจารณาคือแถวที่ 2 และ แถวที่ 3 พบว่าค่าในคอลัมน์ Dependency เท่ากับ B และค่าในคอลัมน์ Symbol Type เป็น Activity ทั้งสองแถว เมื่อพิจารณาคอลัมน์ Symbol Name ที่มีค่าเท่ากับ B พบว่า Symbol Type เป็น Fork ดังนั้นจะได้ RHS เป็น <C><D>|<D><C>

- Join: แทนที่ส่วน RHS ด้วย ค่าในคอลัมน์ Symbol Name ของแถวที่กำลังพิจารณา ภายในเครื่องหมายสัญลักษณ์ไม่สิ้นสุดที่กำหนดไว้ ตัวอย่าง จากตารางที่ 3.3 กำหนดให้แถวที่กำลังพิจารณาคือแถวที่ 6 พบว่าค่าในคอลัมน์ Dependency เท่ากับ E และค่าในคอลัมน์ Symbol Type เป็น Activity เมื่อพิจารณาคอลัมน์ Symbol Name ที่มีค่าเท่ากับ E พบว่า Symbol Type เป็น Join ดังนั้นจะได้ RHS เป็น <F>

ขั้นตอนที่ 2.4 กำหนดให้แถวที่กำลังพิจารณา เป็นแถวที่ถูกพิจารณาแล้ว และกลับไปทำตามขั้นตอนที่ 2

ขั้นตอนที่ 2.5 สร้าง RHS ของโปรดักชั่น โดยค้นหาแถวที่มีค่าในคอลัมน์ Symbol Name ตรงกับค่าในคอลัมน์ Dependency ของแถวที่กำลังพิจารณา และนำค่าในคอลัมน์ Symbol Type ของแถวที่ค้นได้มาพิจารณา โดยแบ่งออกเป็น 3 ประเภทดังต่อไปนี้

- Activity: แทนส่วน RHS ด้วยค่าในคอลัมน์ Defined Name
- Merge: แทนส่วน RHS ด้วยคำว่า “END”
- Join: แทนส่วน RHS ด้วยคำว่า “END”

จากตารางที่ 3.1 และ 3.2 สามารถสร้างไวยากรณ์ดังแสดงในภาพประกอบที่ 3.10

```
<A> ::= <B>
<B> ::= b<C>
<C> ::= invalid[D] | valid<E>
<D> ::= d<B>
<E> ::= e<F>
<F> ::= invalid[G] | valid<H>
<G> ::= g<E>
<H> ::= h<I>
<I> ::= i
```

ภาพประกอบที่ 3.10 ไวยากรณ์สำหรับ Check Balance

ขั้นตอนที่ 5 สร้างกรณีทดสอบ (Test Case Generation)

ไวยากรณ์ที่สร้างในขั้นตอนที่ 4 จะถูกใช้เพื่อสร้างกรณีทดสอบ โดยอักษรที่อยู่ในเครื่องหมาย < และ >, เครื่องหมาย [และ] และเครื่องหมาย { และ } จะเป็นสัญลักษณ์ไม่สิ้นสุด ในการสร้างกรณีทดสอบจะพิจารณาจาก RHS ของไวยากรณ์ โดยเริ่มต้นจากเพิ่ม RHS ของโปรดักชั่นแรกเข้าไปในกรณีทดสอบ จากตัวอย่างไวยากรณ์ภาพประกอบที่ 3.10 พบว่า RHS ของโปรดักชั่นแรกคือ <B> ดังนั้น <B> จะถูกเพิ่มเข้าไปในกรณีทดสอบ จากนั้นจะพิจารณาว่าในกรณีทดสอบมีสัญลักษณ์ไม่สิ้นสุดหรือไม่ หากมีจะแทนสัญลักษณ์ไม่สิ้นสุดนั้นโดยค้นหาโปรดักชั่นที่ LHS มีสัญลักษณ์ไม่สิ้นสุดตรงกันกับสัญลักษณ์ไม่สิ้นสุดที่พบในกรณีทดสอบ โดยจะนำ RHS ของโปรดักชั่นที่ค้นหาได้นั้นมาแทนที่สัญลักษณ์ไม่สิ้นสุดในกรณีทดสอบ ตัวอย่างเช่น กรณีทดสอบคือ <B> ซึ่ง <B> คือสัญลักษณ์ไม่สิ้นสุด ดังนั้น เราจะพิจารณาหาโปรดักชั่นที่ LHS ตรงกับ <B> จากภาพประกอบที่ 3.10 โปรดักชั่นที่ค้นหาได้คือ <B> ::= b<C> ดังนั้นจึงนำ RHS ของโปรดักชั่นนี้ นั่นคือ b<C> มาแทนเข้าไปที่สัญลักษณ์ไม่สิ้นสุด <B> ในกรณีทดสอบ ดังนั้นจะได้กรณีทดสอบเป็น b<C> กระบวนการค้นหาและแทนที่นี้จะกระทำไปเรื่อยๆ โดยจะสิ้นสุดกระบวนการใน 2 ลักษณะคือ 1) RHS ของโปรดักชั่นที่ค้นหาได้ ระบุเป็น “END” จะทำการตัดสัญลักษณ์ไม่สิ้นสุดในกรณีทดสอบนั้นออกจากกรณีทดสอบ และ 2) ในกรณีทดสอบไม่ปรากฏสัญลักษณ์ไม่สิ้นสุด และหากในกรณีทดสอบปรากฏสัญลักษณ์ไม่สิ้นสุดมากกว่า 1 ตำแหน่ง จะทำการแทนเป็นลำดับจากซ้ายไปขวา

ในขั้นตอนการนำ RHS มาแทนที่ หากพบสัญลักษณ์หรือ (|) สัญลักษณ์วงเล็บก้ามปู ([ ]) หรือ สัญลักษณ์วงเล็บปีกกา ({ }) จะพิจารณาตามสัญลักษณ์เหล่านั้นดังนี้

สัญลักษณ์หรือ (|) : หากพบว่าโปรดักชั่นที่ได้จากการค้นหาปรากฏสัญลักษณ์หรือด้าน RHS จะแทนสัญลักษณ์ไม่สิ้นสุดในกรณีทดสอบด้วยรูปแบบต่างๆที่คั่นด้วยสัญลักษณ์หรือของ RHS โดยแต่ละรูปแบบจะต้องถูกนำมาแทน 1 ครั้ง โดยจะแยกกรณีทดสอบตามจำนวนของรูปแบบที่แทน จากตัวอย่างไวยากรณ์ภาพประกอบที่ 3.10 สมมติให้กรณีทดสอบคือ b<C> สัญลักษณ์ไม่สิ้นสุดในกรณีทดสอบคือ <C> เมื่อพิจารณาหาโปรดักชั่นที่ LHS ตรงกับ <C> นั่นคือ <C>::=invalid[D] | valid<E> พบว่า RHS ปรากฏสัญลักษณ์หรือ ซึ่งมี 2 รูปแบบคือ invalid[D] และ valid<E> ดังนั้นจะนำทั้ง 2 รูปแบบแทนที่ <C> จะได้ 2 กรณีทดสอบคือ b invalid[D] และ b valid<E>

สัญลักษณ์วงเล็บก้ามปู ([ ]) : หากพบว่าโปรดักชั่นที่ได้จากการค้นหาปรากฏสัญลักษณ์วงเล็บก้ามปู ซึ่งเป็นการระบุว่าเป็นการวนซ้ำ จะแทนสัญลักษณ์ไม่สิ้นสุดในกรณีทดสอบโดยพิจารณาเป็น 2 กรณีดังนี้

กรณีที่ 1: กรณีที่ไม่เคยแทนรูปแบบนั้นลงในสัญลักษณ์ไม่สิ้นสุดในกรณีทดสอบ จะนำรูปแบบนั้นแทนลงไปในสัญลักษณ์ไม่สิ้นสุดในกรณีทดสอบ

กรณีที่ 2: กรณีที่เคยแทนรูปแบบนั้นลงไปในสัญลักษณ์ไม่สิ้นสุดในกรณีทดสอบแล้ว จะนำรูปแบบลำดับถัดไปมาแทนให้กับสัญลักษณ์ไม่สิ้นสุดในกรณีทดสอบนั้น

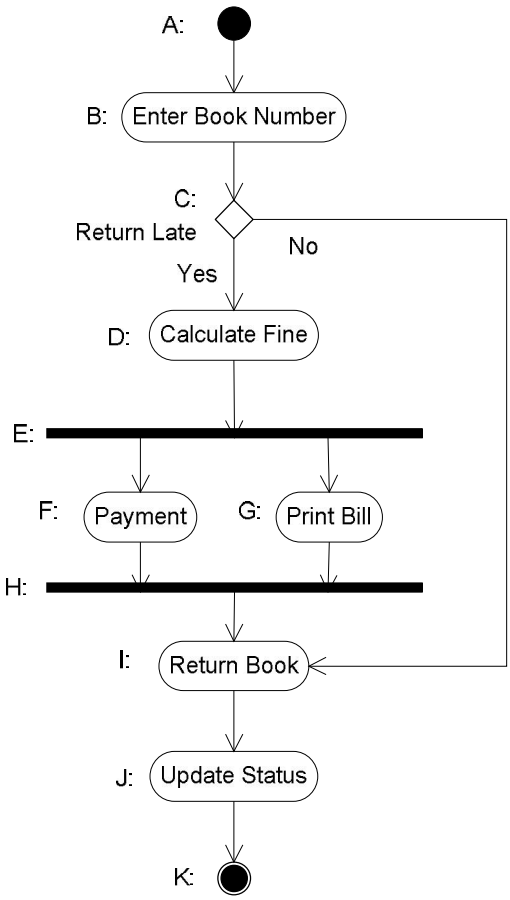
ตัวอย่างจากไวยากรณ์ภาพประกอบที่ 3.10 เมื่อกำหนดให้กรณีทดสอบคือ b<C> และโปรดักชั่นที่ถูกเลือกใช้คือ <C>::= invalid[D] | valid<E> ซึ่งมี LHS ตรงกับ <C> ดังนั้นจึงนำ invalid[D] และ valid<E> ไปแทน <C> จะได้กรณีทดสอบเป็น b invalid[D] และ b valid<E> เมื่อพิจารณาจะพบว่ากรณีทดสอบ b invalid[D] มีสัญลักษณ์การวนซ้ำที่ [D] และ [D] ยังไม่เคยถูกแทนที่ จึงเข้ากรณีที่ 1 จึงทำการแทนต่อเป็น b invalid d<B> และแทน <B> ต่อด้วยโปรดักชั่น <B> ::= b<C> จะได้กรณีทดสอบเป็น b invalid d b <C> จากนั้นแทน <C> ต่อ ซึ่งจะพบว่าเคยแทนรูปแบบ invalid[D] มาแล้ว ทำให้เข้ากรณีที่ 2 เราจึงนำรูปแบบลำดับถัดไปนั่นคือ valid<E> เข้ามาแทนสัญลักษณ์ไม่สิ้นสุด <C> ดังนั้นสุดท้ายจะได้กรณีทดสอบเป็น b invalid d b valid<E> และ b valid<E>

สัญลักษณ์วงเล็บปีกกา ({ }) : หากพบว่าโปรดักชั่นที่ได้จากการค้นหาปรากฏสัญลักษณ์วงเล็บปีกกา จะนำไปแทนที่สัญลักษณ์ไม่สิ้นสุดในกรณีทดสอบ แล้วทำการตรวจสอบก่อนว่าในกรณีทดสอบยังปรากฏสัญลักษณ์ไม่สิ้นสุดอื่นต่อท้ายอีกหรือไม่ หากในกรณีทดสอบพบสัญลักษณ์ไม่สิ้นสุดอื่นอีก จะตัดวงเล็บปีกกานั้นออก แล้วทำการแทนที่สัญลักษณ์ไม่สิ้นสุดในกรณีทดสอบลำดับต่อไปแทน แต่หากในกรณีทดสอบไม่ปรากฏสัญลักษณ์ไม่สิ้นสุดอื่นๆอีก จะทำการแทนที่วงเล็บปีกกาโดยการค้นหาโปรดักชั่นที่มี LHS ที่ตรงกันตามขั้นตอนปกติ พิจารณาแผนภาพกิจกรรม Return Book ในภาพประกอบที่ 3.11 สามารถสร้างไวยากรณ์ได้ดังภาพประกอบที่ 3.12 และภาพประกอบที่ 3.13 แสดงขั้นตอนการสร้างกรณีทดสอบ

สัญลักษณ์แสดงสัญลักษณ์ไม่สิ้นสุดของไวยากรณ์แสดงดังตารางที่ 3.4

ตารางที่ 3.4 สัญลักษณ์แสดงสัญลักษณ์ไม่สิ้นสุดของไวยากรณ์

สัญลักษณ์	ความหมาย
< >	สัญลักษณ์ไม่สิ้นสุด
[ ]	สัญลักษณ์ไม่สิ้นสุด แสดงการทำซ้ำ 1 ครั้ง
{ }	สัญลักษณ์ไม่สิ้นสุด แสดงการแทนหรือไม่แทน
|	สัญลักษณ์แสดงการแทนหลายรูปแบบ



ภาพประกอบที่ 3.11 แผนภาพกิจกรรม Return Book (MCA, 2010)

<A> ::= <B>

<B> ::= b<C>

<C> ::= Yes <D> | No<I>

<D> ::= d<E>

<E> ::= <F><G> | <G><F>

<F> ::= f{H}

<G> ::= g{H}

<H> ::= <I>

<I> ::= i<J>

<J> ::= j

ภาพประกอบที่ 3.12 ตัวอย่างไวยากรณ์สำหรับ Return Book

Test Case	: <B>	(1)
	: b<C>	(2)
	: b-Yes <D>	(3)
	: b-Yes-d <E>	(4)
	: b-Yes-d <F><G>	(5)
	: b-Yes-d-f {H} <G>	(6)
	: b-Yes-d-f-g {H}	(7)
	: b-Yes-d-f-g <I>	(8)
	: b-Yes-d-f-g-i <J>	(9)
	: b-Yes-d-f-g-i-j	(10)

ภาพประกอบที่ 3.13 ขั้นตอนการสร้างกรณีทดสอบสำหรับ Return Book

จากภาพประกอบที่ 3.13

บรรทัดที่ (6) ตรวจสอบแล้วพบว่าในกรณียังมีสัญลักษณ์ไม่สิ้นสุด <G> ต่อท้ายอีก จึงทำการตัดวงเล็บปีกกาออก แล้วแทนที่สัญลักษณ์ไม่สิ้นสุด <G> ต่อไป

บรรทัดที่ (7) ตรวจสอบแล้วไม่พบสัญลักษณ์ไม่สิ้นสุดใดต่อท้ายในกรณีทดสอบอีก จึงทำการแทน {H} ตามขั้นตอนปกติ

จากไวยากรณ์ภาพประกอบที่ 3.12 สามารถสร้างกรณีทดสอบได้ 3 กรณีทดสอบดังแสดงในภาพประกอบที่ 3.14

Test Case 1 : b-Yes-d-f-g-i-j

Test Case 2 : b-Yes-d-g-f-i-j

Test Case 3 : b-No-i-j

ภาพประกอบที่ 3.14 กรณีทดสอบสำหรับ Return Book

จากไวยากรณ์ภาพประกอบที่ 3.10 สามารถสร้างกรณีทดสอบได้ 4 กรณีทดสอบดังแสดงในภาพประกอบที่ 3.15

Test Case 1 : b-valid-e-valid-h-i

Test Case 2 : b-invalid-d-b-valid-e-valid-h-i

Test Case 3 : b-valid-e-invalid-g-e-valid-h-i

Test Case 4 : b-invalid-d-b-valid-e-invalid-g-e-valid-h-i

ภาพประกอบที่ 3.15 กรณีทดสอบสำหรับ Check Balance

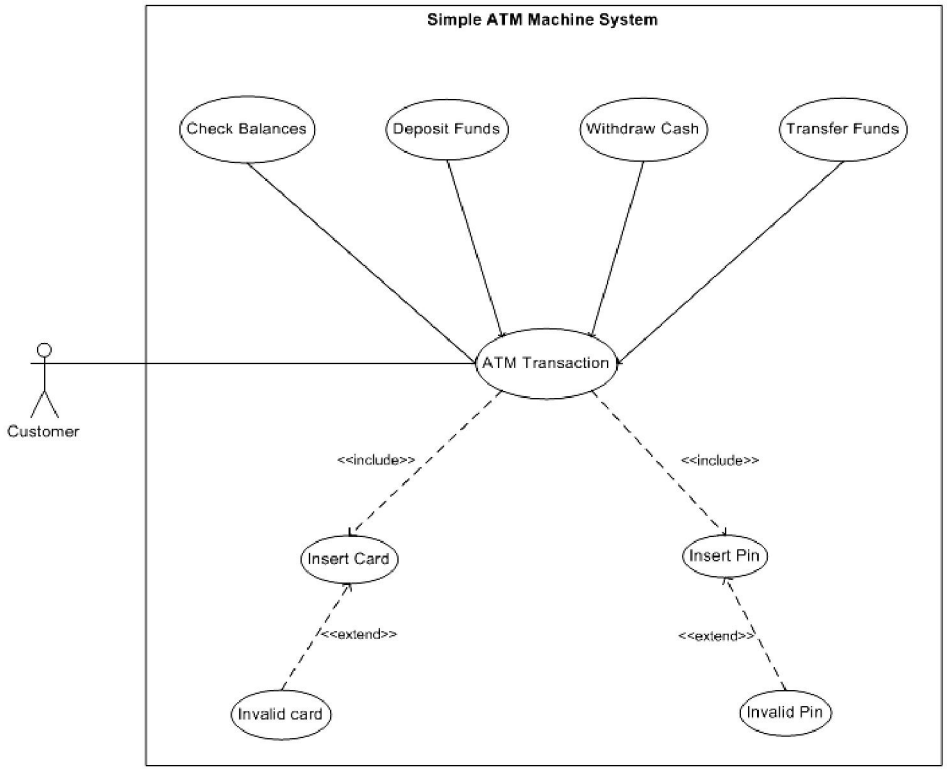
3.3 การประเมินประสิทธิภาพของกรณีทดสอบ

การประเมินประสิทธิภาพกรณีทดสอบในงานวิจัยนี้ ใช้วิธีการ mutation testing ซึ่งประกอบด้วย 3 ขั้นตอนดังนี้

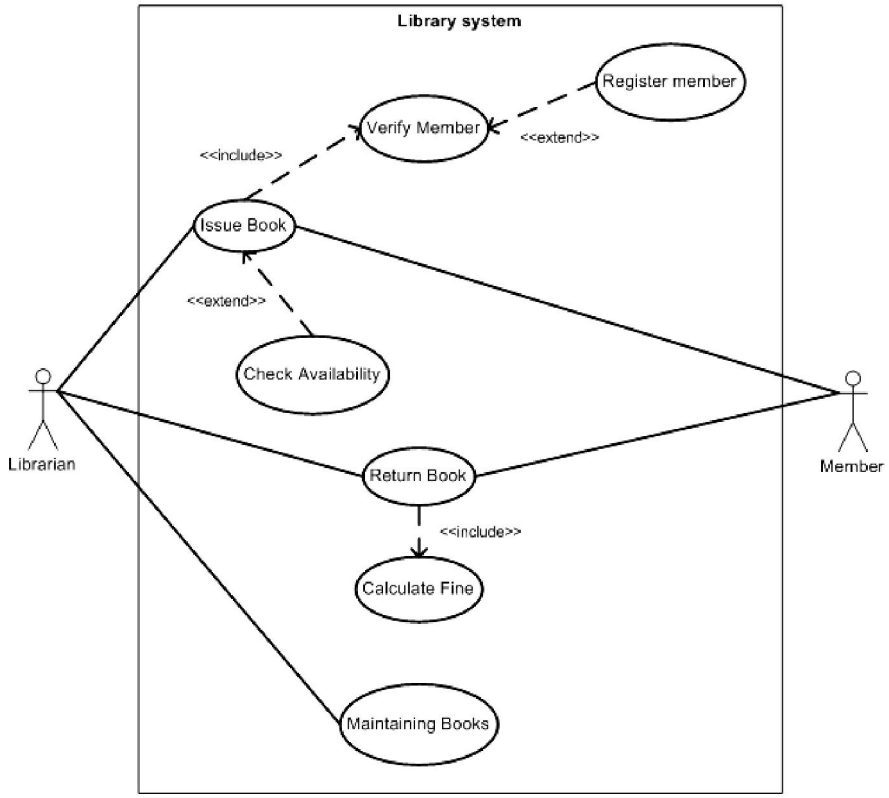
1. สร้างกรณีทดสอบตามวิธีการที่นำเสนอ
2. สร้างโปรแกรม mutant จากโปรแกรมต้นฉบับ
3. ทดสอบโปรแกรมต้นฉบับและโปรแกรม mutant ด้วยกรณีทดสอบที่ได้จากขั้นตอนที่ 1

ขั้นตอนที่ 1 สร้างกรณีทดสอบตามวิธีการที่นำเสนอ

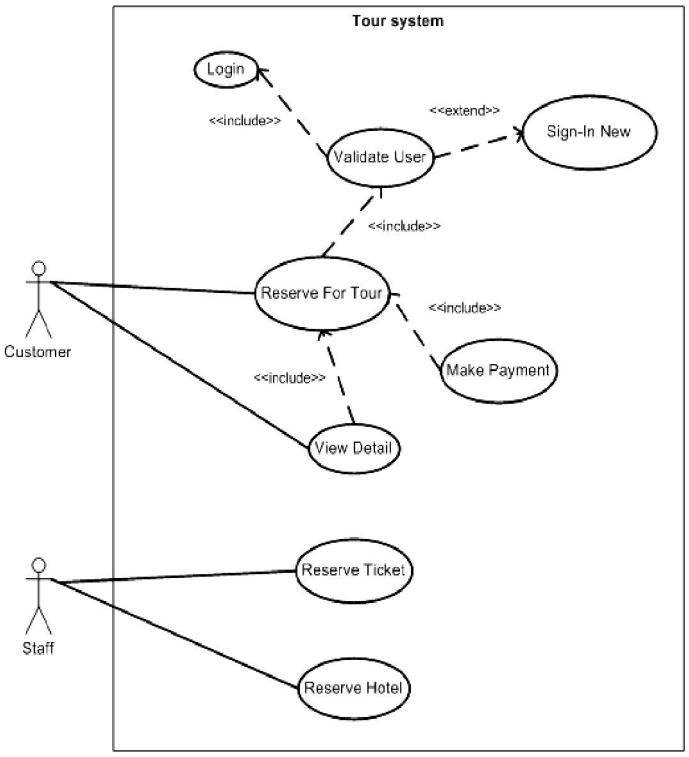
ในการประเมินประสิทธิภาพของกรณีทดสอบที่สร้างตามวิธีการที่นำเสนอ ได้เลือกใช้กรณีศึกษาทั้งหมด 6 ระบบ (MCA, 2010) ได้แก่ 1) ระบบ ATM 2) ระบบห้องสมุด 3) ระบบทัวร์ 4) ระบบจองตั๋วรถไฟ 5) ระบบการขาย และ 6) ระบบโรงพยาบาล แผนภาพ use case ของทั้ง 6 ระบบ แสดงในภาพประกอบที่ 3.16 – ภาพประกอบที่ 3.21 ตามลำดับ ตารางที่ 3.5 แสดงจำนวนกรณีทดสอบของกรณีศึกษาที่สร้างจากวิธีการที่นำเสนอ



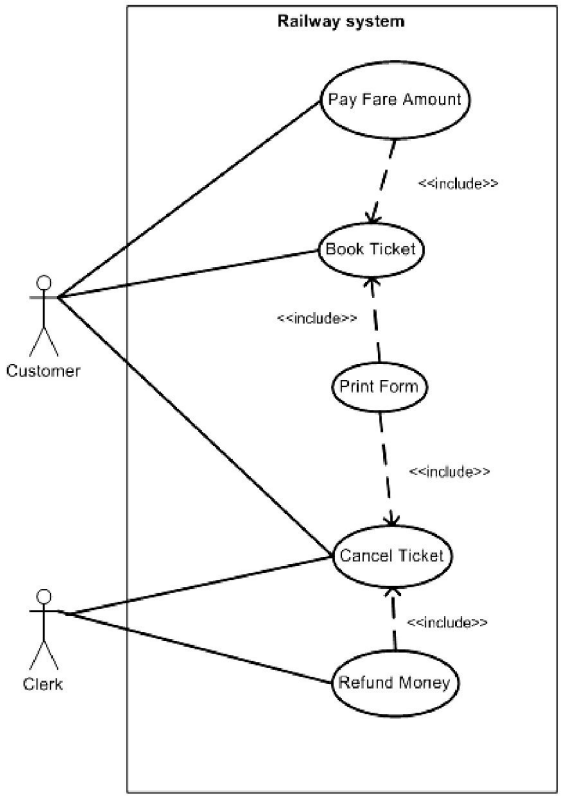
ภาพประกอบที่ 3.16 แผนภาพ use case ระบบ ATM (MCA, 2010)



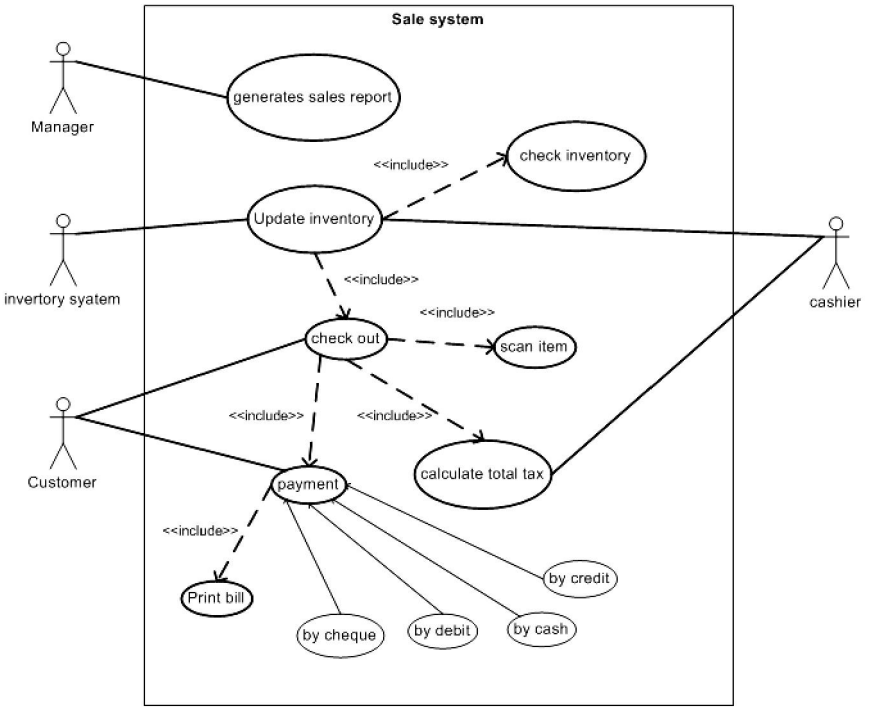
ภาพประกอบที่ 3.17 แผนภาพ use case ระบบห้องสมุด (MCA, 2010)



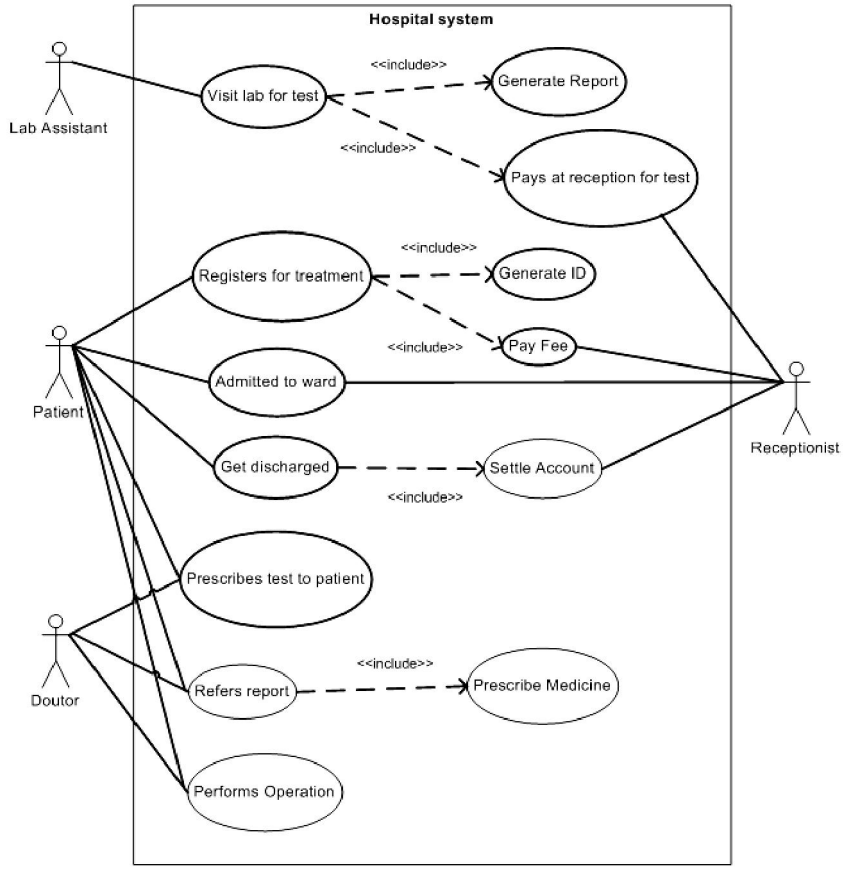
ภาพประกอบที่ 3.18 แผนภาพ use case ระบบทัวร์ (MCA, 2010)



ภาพประกอบที่ 3.19 แผนภาพ use case ระบบจองตั๋วรถไฟ (MCA, 2010)



ภาพประกอบที่ 3.20 แผนภาพ use case ระบบการขาย (MCA, 2010)



ภาพประกอบที่ 3.21 แผนภาพ use case ระบบโรงพยาบาล (MCA, 2010)

ตารางที่ 3.5 จำนวนกรณีทดสอบของกรณีศึกษาที่สร้างจากวิธีการที่นำเสนอ

กรณีศึกษา	จำนวนกรณีทดสอบ
ระบบ ATM	44
ระบบห้องสมุด	7
ระบบทัวร์	18
ระบบจองตั๋วรถไฟ	7
ระบบการขาย	20
ระบบโรงพยาบาล	18

ขั้นตอนที่ 2 สร้างโปรแกรม mutant จากโปรแกรมต้นฉบับ

การสร้างโปรแกรม mutant จะสร้างโดยการปรับเปลี่ยนโปรแกรมให้ต่างไปจากโปรแกรมต้นฉบับ โดยการปรับเปลี่ยนโปรแกรมแต่ละครั้งจะกระทำเพียง 1 ตำแหน่งเท่านั้น โปรแกรมที่ได้จากการปรับเปลี่ยนเรียกว่าโปรแกรม mutant ซึ่งในงานวิจัยนี้จะปรับเปลี่ยนโปรแกรมต้นฉบับโดยอาศัยตัวดำเนินการที่เรียกว่า mutation operator ซึ่งถูกนำเสนอโดย Offutt และคณะ ในบทความเรื่อง “An Experimental Determination of Sufficient Mutant Operators” (Offutt, *et al.,* 1996) ซึ่งในบทความดังกล่าวได้กล่าวถึง mutation operator ทั้งหมด 22 operator ดังแสดงในตารางที่ 3.6

ตารางที่ 3.6 รายละเอียด mutation operator (Offutt, *et al.,* 1996)

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 end replacement
DSA	DATA statement alteration
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

จากโปรแกรมต้นฉบับของกรณีศึกษาทั้ง 6 ระบบ สามารถสร้างโปรแกรม mutant ได้ดังตารางที่ 3.7

ตารางที่ 3.7 จำนวนโปรแกรม mutant ของกรณีศึกษาทั้ง 6 ระบบ

กรณีศึกษา	จำนวนโปรแกรม mutant
ระบบ ATM	620
ระบบห้องสมุด	159
ระบบทัวร์	1155
ระบบจองตั๋วรถไฟ	385
ระบบการขาย	1206
ระบบโรงพยาบาล	453

ตัวอย่างการสร้างโปรแกรม mutant โดยใช้ mutation operator ที่แตกต่างกันแสดงได้ดังนี้

- ตัวอย่างที่ 1 การสร้างโปรแกรม mutant โดยใช้ AOR operator ซึ่งแทน arithmetic operator (–) ด้วย arithmetic operator (+) แสดงดังภาพประกอบที่ 3.22

โปรแกรมต้นฉบับ	โปรแกรม mutant
int amount = 500; int count = 100; if (amount > count) { amount **-=** count; System.out.println(amount); }	int amount = 500; int count = 100; if (amount > count) { amount **+=** count; System.out.println(amount); }

ภาพประกอบที่ 3.22 ตัวอย่างการสร้างโปรแกรม mutant โดยใช้ AOR operator

- ตัวอย่างที่ 2 การสร้างโปรแกรม mutant โดยใช้ CRP operator ซึ่งแทน constant (Penalty = 3) ด้วย constant (Penalty = 10) แสดงดังภาพประกอบที่ 3.23

โปรแกรมต้นฉบับ	โปรแกรม mutant
Constant{ public static final int Penalty = **3**; } int amount = 100; int date = 2; if(status == 1) { amount = date * Constant.Penalty; } System.out.println(amount);	Constant{ public static final int Penalty = **10**; } int amount = 100; int date = 2; if(status == 1) { amount = date * Constant.Penalty; } System.out.println(amount);

ภาพประกอบที่ 3.23 ตัวอย่างการสร้างโปรแกรม mutant โดยใช้ CRP operator

- ตัวอย่างที่ 3 การสร้างโปรแกรม mutant โดยใช้ ROR operator ซึ่งแทน relational operator (>) ด้วย relational operator (<) แสดงดังภาพประกอบที่ 3.24

โปรแกรมต้นฉบับ	โปรแกรม mutant
int amount = 500; int count = 100; if (amount **>** count) { amount -= count; System.out.println(amount); }	int amount = 500; int count = 100; if (amount **<** count) { amount -= count; System.out.println(amount); }

ภาพประกอบที่ 3.24 ตัวอย่างการสร้างโปรแกรม mutant โดยใช้ ROR operator

- ตัวอย่างที่ 4 การสร้างโปรแกรม mutant โดยใช้ SDL operator ซึ่งลบ statement (amount -= count) ออก แสดงดังภาพประกอบที่ 3.25

โปรแกรมต้นฉบับ	โปรแกรม mutant
int amount = 500; int count = 100; if (amount > count) { **amount -= count;** System.out.println(amount); }	int amount = 500; int count = 100; if (amount > count) { **//amount -= count;** System.out.println(amount); }

ภาพประกอบที่ 3.25 ตัวอย่างการสร้างโปรแกรม mutant โดยใช้ SDL operator

- ตัวอย่างที่ 5 การสร้างโปรแกรม mutant โดยใช้ UOI operator ซึ่งเพิ่ม unary operator (++) หน้าตัวแปร count แสดงดังภาพประกอบที่ 3.26

โปรแกรมต้นฉบับ	โปรแกรม mutant
int amount = 500; int count = 100; if (amount > count) { amount -= **count**; System.out.println(amount); }	int amount = 500; int count = 100; if (amount > count) { amount -= **(++count)**; System.out.println(amount); }

ภาพประกอบที่ 3.26 ตัวอย่างการสร้างโปรแกรม mutant โดยใช้ UOI operator

ขั้นตอนที่ 3 ทดสอบโปรแกรมต้นฉบับและโปรแกรม mutant ด้วยกรณีทดสอบที่ได้จากขั้นตอนที่ 1

แต่ละโปรแกรม mutant และโปรแกรมต้นฉบับถูกนำมาปฏิบัติโดยกรณีทดสอบต่างๆที่ได้จากขั้นตอนที่ 1 ทำการเปรียบเทียบผลลัพธ์ที่ได้จากการปฏิบัติ โดยการเปรียบเทียบผลลัพธ์จากแต่ละโปรแกรม mutant และผลลัพธ์จากโปรแกรมต้นฉบับ หากผลลัพธ์ต่างกันแสดงว่ากรณีทดสอบที่สร้างนั้นสามารถตรวจจับข้อผิดพลาดได้ นั่นคือ สามารถทำการฆ่า mutant นี้ได้ ซึ่งประสิทธิภาพของกรณีทดสอบดูได้จากค่า mutation score หากค่า mutation score เข้าใกล้ 1 หมายถึงกรณีทดสอบมีประสิทธิภาพ ซึ่งผลจากการประเมินประสิทธิภาพของกรณีทดสอบที่สร้างตามวิธีการที่นำเสนอ โดยการทำ mutation testing แสดงดังตารางที่ 3.8

จากผลการประเมินประสิทธิภาพของกรณีทดสอบทั้ง 6 กรณีศึกษา คือระบบ ATM, ระบบห้องสมุด, ระบบทัวร์, ระบบจองตั๋วรถไฟ, ระบบการขาย และระบบโรงพยาบาล พบว่าที่ค่า mutation score เท่ากับ 0.97, 0.97, 0.98, 0.97, 0.98 และ 0.95 ตามลำดับ ซึ่งค่า mutation score ของทั้ง 6 กรณีศึกษาเป็นค่าที่เข้าใกล้ 1 แสดงให้เห็นว่ากรณีทดสอบที่สร้างตามวิธีการที่นำเสนอมีประสิทธิภาพในการตรวจจับข้อผิดพลาด

ตารางที่ 3.8 ผลการประเมินประสิทธิภาพของกรณีทดสอบ โดย mutation testing

กรณีศึกษา	จำนวนกรณีทดสอบ	จำนวนโปรแกรม mutant	จำนวนโปรแกรม mutant ที่ถูกฆ่า	Mutation Score
ระบบ ATM	44	620	600	0.97
ระบบห้องสมุด	7	159	155	0.97
ระบบทัวร์	18	1155	1130	0.98
ระบบจองตั๋วรถไฟ	7	385	373	0.97
ระบบการขาย	20	1206	1178	0.98
ระบบโรงพยาบาล	18	453	429	0.95

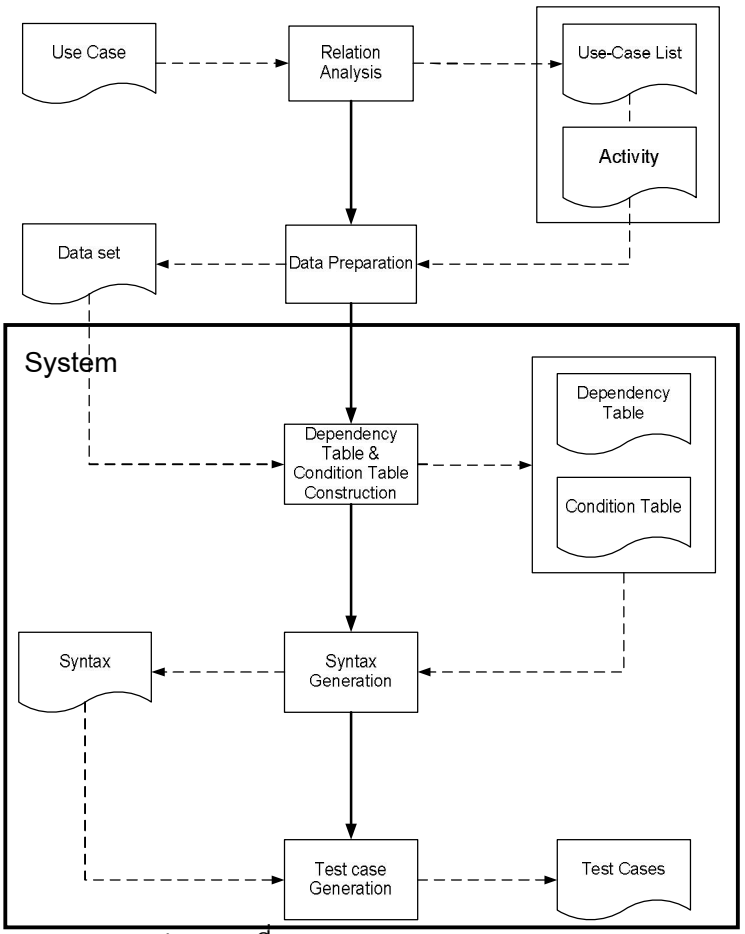
บทที่ 4

การออกแบบและพัฒนาเครื่องมือต้นแบบ

เนื้อหาในบทนี้กล่าวถึงการออกแบบและพัฒนาเครื่องมือต้นแบบ ซึ่งพัฒนาขึ้นเพื่อสนับสนุนการสร้างกรณีทดสอบจากแผนภาพ use case และแผนภาพกิจกรรมบนพื้นฐานของไวยากรณ์ โดยใช้ภาษา Visual C# ในการพัฒนา

4.1 กรอบการทำงานของระบบ

เครื่องมือต้นแบบมีส่วนการทำงานหลักๆ 3 ส่วน ดังนี้ 1) Dependency Table & Condition Table Construction 2) Syntax Generation และ 3) Test case Generation กรอบการทำงานของระบบ แสดงดังภาพประกอบที่ 4.1



ภาพประกอบที่ 4.1 กรอบการทำงานของระบบ

4.2 ขั้นตอนการดำเนินงานของระบบ

ขั้นตอนการดำเนินงานของระบบประกอบด้วย 3 ขั้นตอนดังนี้

1. การป้อนข้อมูลเข้าสู่ระบบ
2. ระบบสร้างตารางการขึ้นต่อกันและตารางเงื่อนไขทางตรรกะ จากนั้นสร้างไวยากรณ์
3. ระบบสร้างกรณีทดสอบ โดยพิจารณาจากไวยากรณ์

ขั้นตอนที่ 1 การป้อนข้อมูลเข้าสู่ระบบ ผู้ใช้จะต้องป้อนชื่อของแผนภาพกิจกรรมและชุดข้อมูลเบื้องต้นที่พิจารณาจากแผนภาพกิจกรรมเข้าสู่ระบบ ซึ่ง 1 ชุดข้อมูลเบื้องต้นจะประกอบด้วยชุดข้อมูลย่อยทั้งหมด 10 ชุดข้อมูลย่อยดังนี้

Data Set {

Start

Activity[1..m]

Transition[1..n]

Branch[1..o]

Condition[1..p]

Merge[1..q]

Loop[1..r]

Fork[1..s]

Join[1..t]

End

}

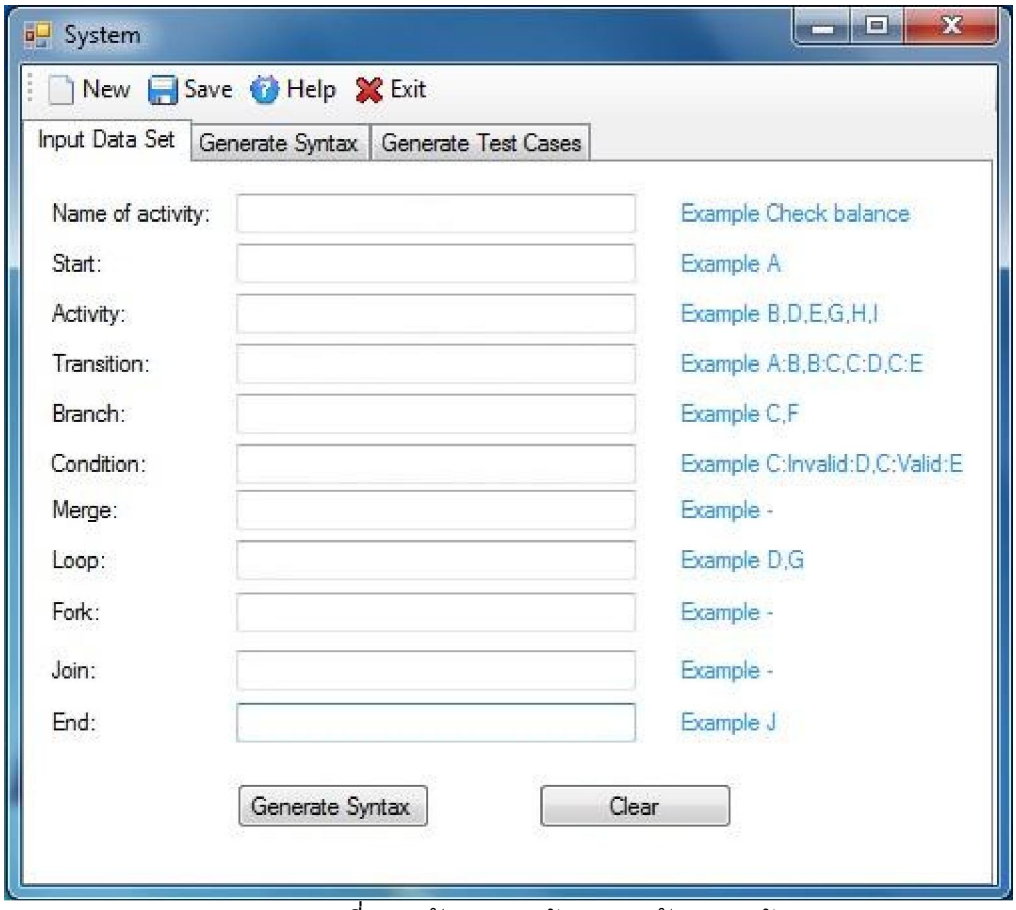
ขั้นตอนที่ 2 ระบบจะนำข้อมูลที่ป้อนเข้ามา ไปสร้างตารางการขึ้นต่อกันและตารางเงื่อนไขทางตรรกะ จากนั้นจึงนำทั้ง 2 ตารางมาพิจารณาร่วมกันเพื่อสร้างไวยากรณ์

ขั้นตอนที่ 3 ระบบทำการสร้างกรณีทดสอบ โดยพิจารณาจากไวยากรณ์

4.3 การออกแบบส่วนติดต่อกับผู้ใช้

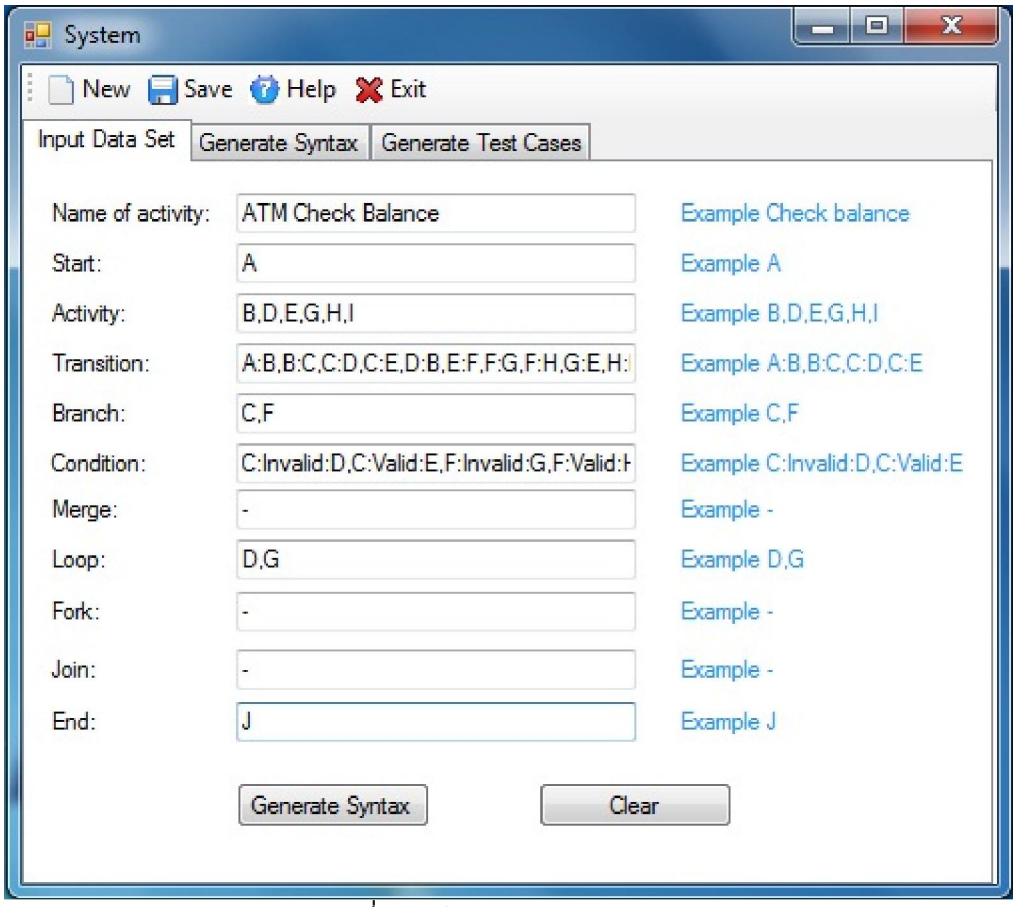
เครื่องมือต้นแบบมีส่วนติดต่อผู้ใช้งานทั้งหมด 3 ส่วน ดังนี้

ส่วนที่ 1 การรับข้อมูลเข้าสู่ระบบ โดยผู้ใช้ระบุชื่อของแผนภาพกิจกรรมและชุดข้อมูลเบื้องต้น โดยการกดปุ่ม “New” บนแถบเมนู ระบบจะแสดงแถบ Input Data Set ดังแสดงในภาพประกอบที่ 4.2



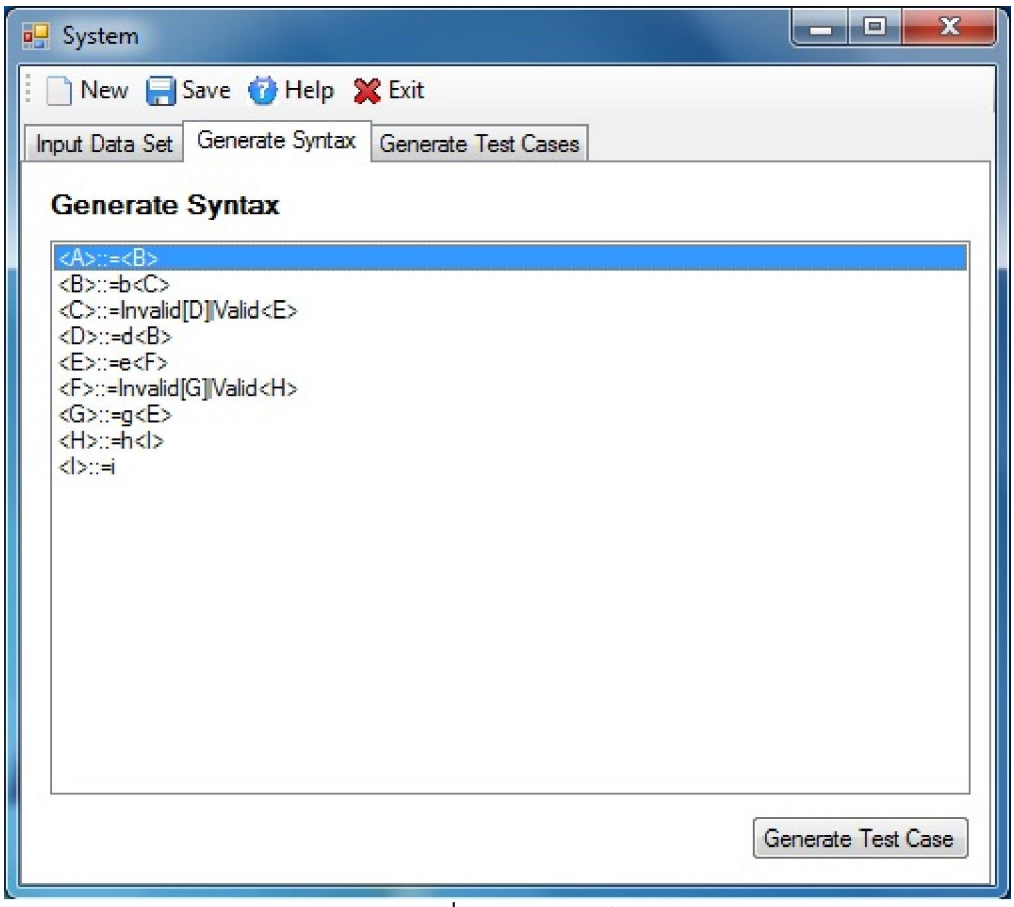
ภาพประกอบที่ 4.2 ตัวอย่างหน้าจอสำหรับกรอกข้อมูล

ผู้ใช้ทำการป้อนชื่อแผนภาพกิจกรรมลงในกล่องข้อความ Name of activity: และป้อนชุดข้อมูลย่อยต่างๆ ลงในแต่ละกล่องข้อความที่ตรงกับชุดข้อมูลย่อยนั้นๆ กรณีที่ชุดข้อมูลย่อยใดไม่มีข้อมูล ให้ป้อนเครื่องหมายยัติภังค์ (-) ลงในกล่องข้อความนั้น ตัวอย่างการป้อนข้อมูล แสดงดังภาพประกอบที่ 4.3



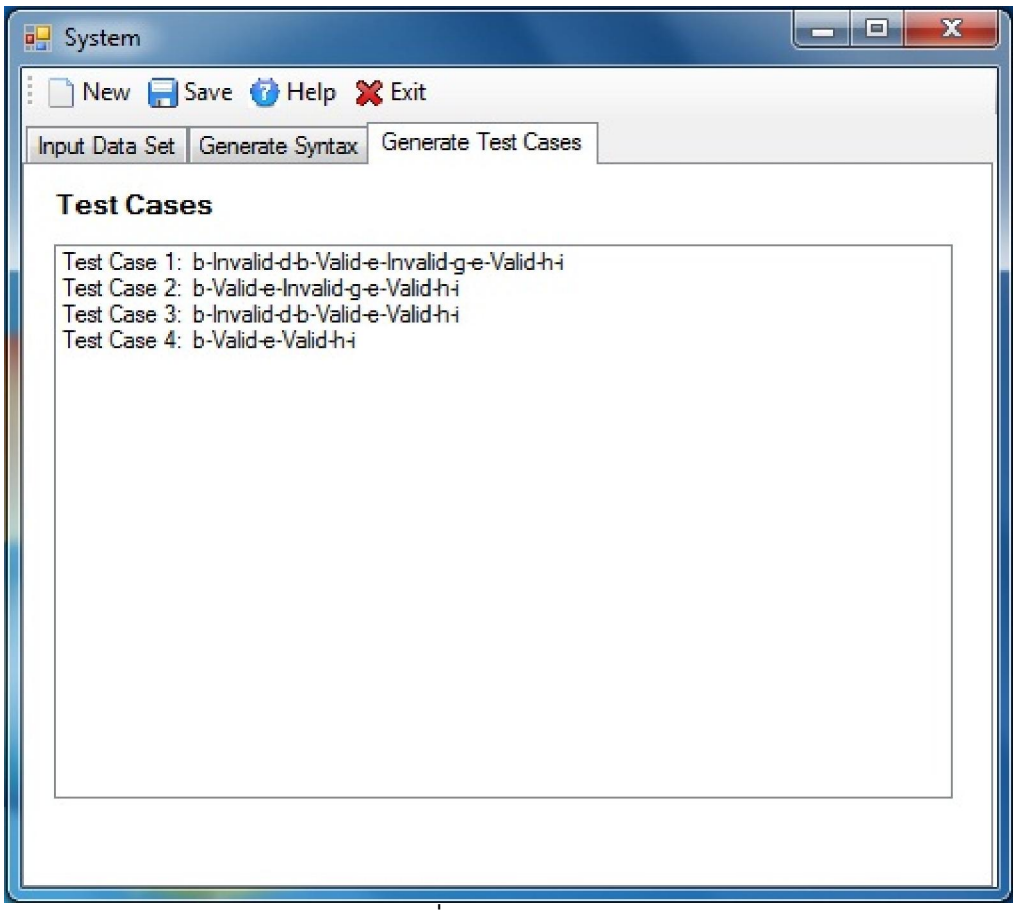
ภาพประกอบที่ 4.3 ตัวอย่างการป้อนข้อมูลเข้าระบบ

ส่วนที่ 2 เมื่อผู้ใช้ป้อนข้อมูลทั้งหมดครบถ้วนแล้ว ผู้ใช้สามารถสร้างไวยากรณ์โดยการกดปุ่ม “Generate Syntax” ที่ปรากฏด้านล่างในแถบ Input Data Set ระบบจะทำการสร้างตารางการขึ้นต่อกันและตารางเงื่อนไขทางตรรกะ จากนั้นระบบจะนำทั้งสองตารางมาพิจารณาเพื่อสร้างไวยากรณ์ ซึ่งไวยากรณ์ที่สร้างได้จะถูกแสดงในแถบ Generate Syntax ตัวอย่างไวยากรณ์ดังแสดงในภาพประกอบที่ 4.4



ภาพประกอบที่ 4.4 ตัวอย่างไวยากรณ์

ส่วนที่ 3 ผู้ใช้สามารถสร้างกรณีทดสอบโดยการกดปุ่ม “Generate Test Case” ที่ปรากฏด้านล่างของแถบ Generate Syntax ระบบจะทำการสร้างกรณีทดสอบโดยพิจารณาจากไวยากรณ์ที่สร้างได้ และแสดงกรณีทดสอบที่สร้างได้ในแถบ Generate Test Cases ตัวอย่างกรณีทดสอบแสดงดังภาพประกอบที่ 4.5



ภาพประกอบที่ 4.5 ตัวอย่างกรณีทดสอบ

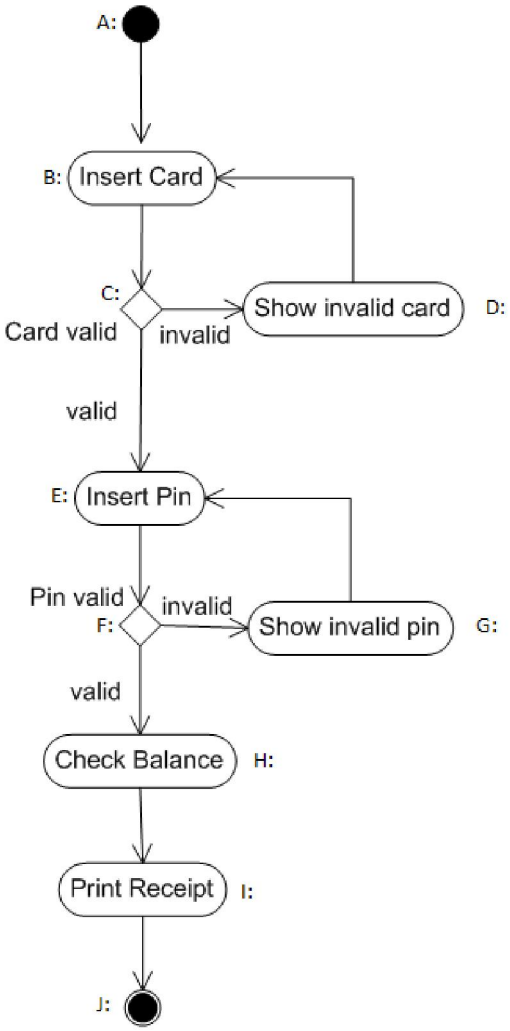
4.4 การทดสอบประสิทธิภาพของเครื่องมือต้นแบบ

เครื่องมือต้นแบบที่พัฒนาขึ้นเพื่อสนับสนุนวิธีการที่นำเสนอนี้ ถูกนำมาใช้สร้างกรณีทดสอบสำหรับ 6 กรณีศึกษา (MCA, 2010) คือ 1) ระบบ ATM 2) ระบบห้องสมุด 3) ระบบทัวร์ 4) ระบบจองตั๋วรถไฟ 5) ระบบการขาย และ 6) ระบบโรงพยาบาล จากการสร้างกรณีทดสอบพบว่าเครื่องมือต้นแบบนี้สามารถสร้างกรณีทดสอบได้อย่างถูกต้อง

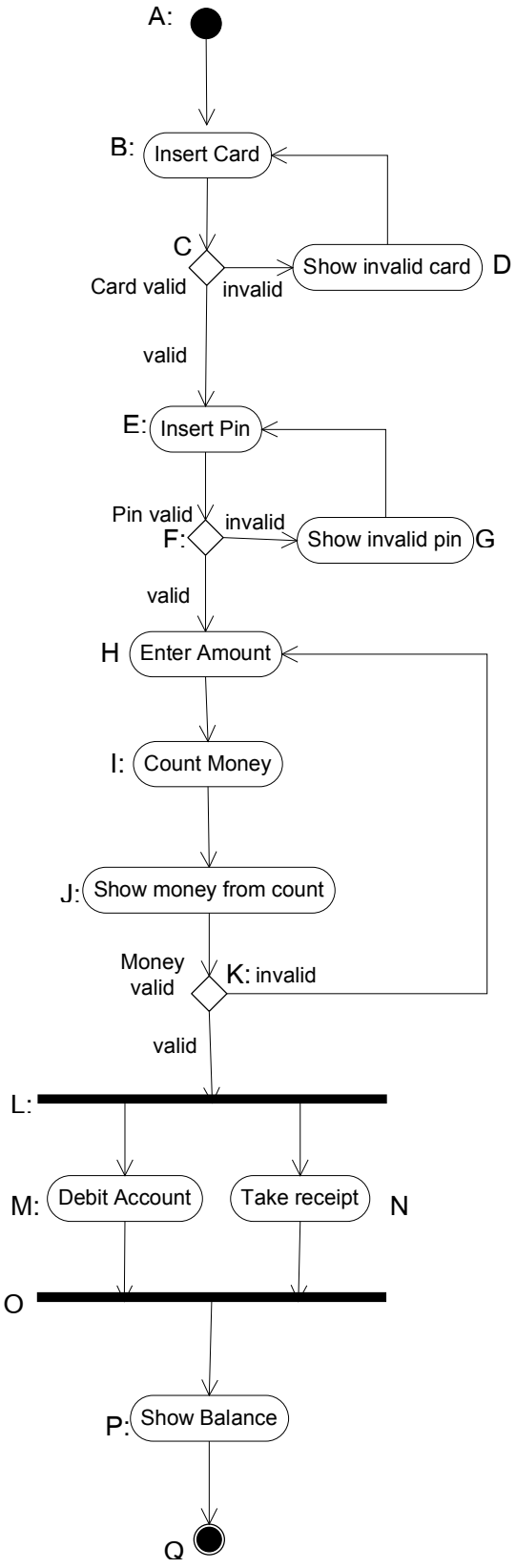
ตัวอย่างการใช้เครื่องมือต้นแบบ แสดงได้ดังต่อไปนี้

ตัวอย่างที่ 1 การสร้างกรณีทดสอบสำหรับระบบ ATM

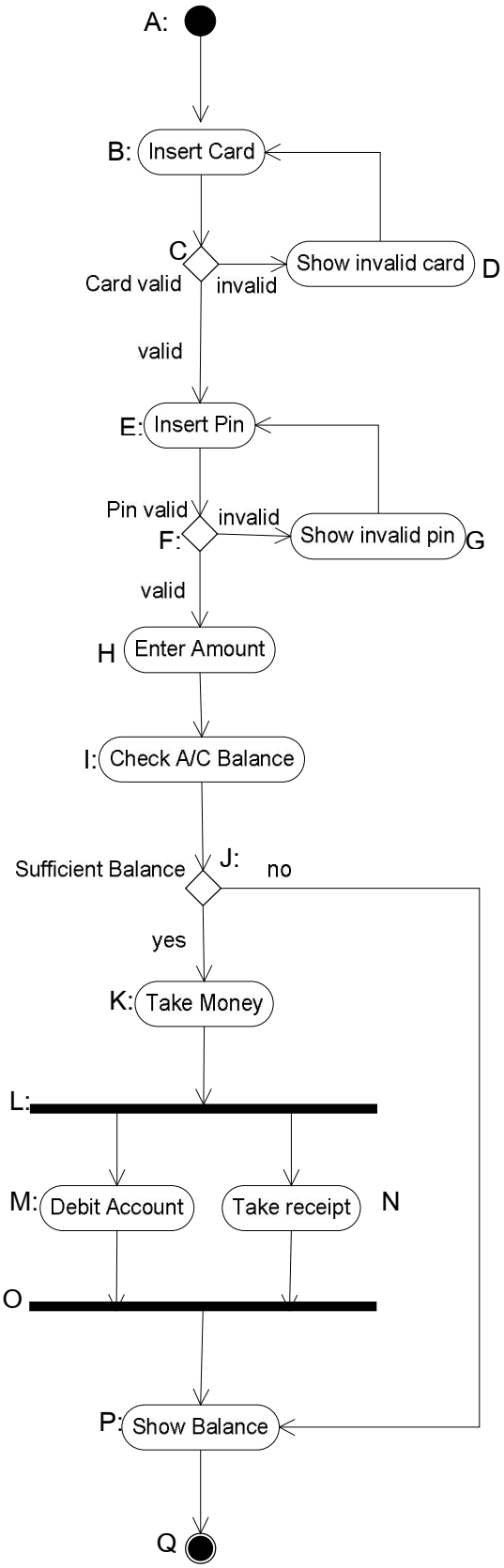
ระบบ ATM ประกอบด้วย use case ที่ต้องทำการทดสอบทั้งหมด 4 use case คือ Check Balances use case, Deposit Funds use case, Withdraw Cash use case และ Transfer Funds use case ซึ่งแผนภาพกิจกรรมที่สอดคล้องกับทั้ง 4 use case แสดงดังภาพประกอบที่ 4.6 – ภาพประกอบที่ 4.9 ตามลำดับ



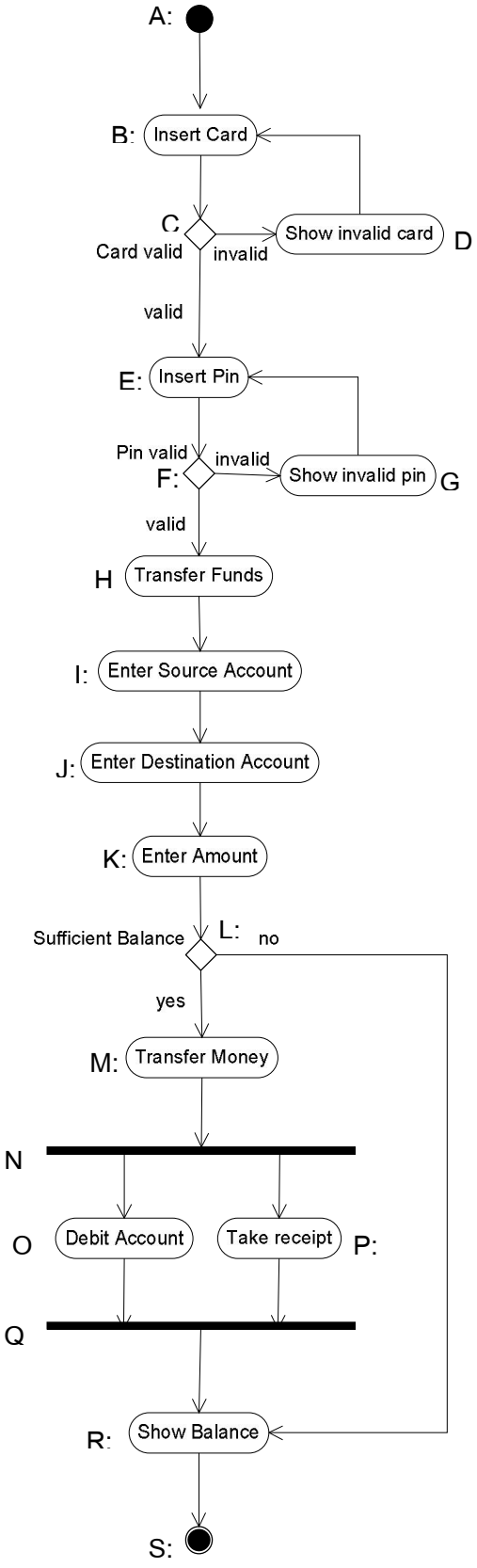
ภาพประกอบที่ 4.6 แผนภาพกิจกรรม Check Balances (MCA, 2010)



ภาพประกอบที่ 4.7 แผนภาพกิจกรรม Deposit Funds (MCA, 2010)



ภาพประกอบที่ 4.8 แผนภาพกิจกรรม Withdraw Cash (MCA, 2010)

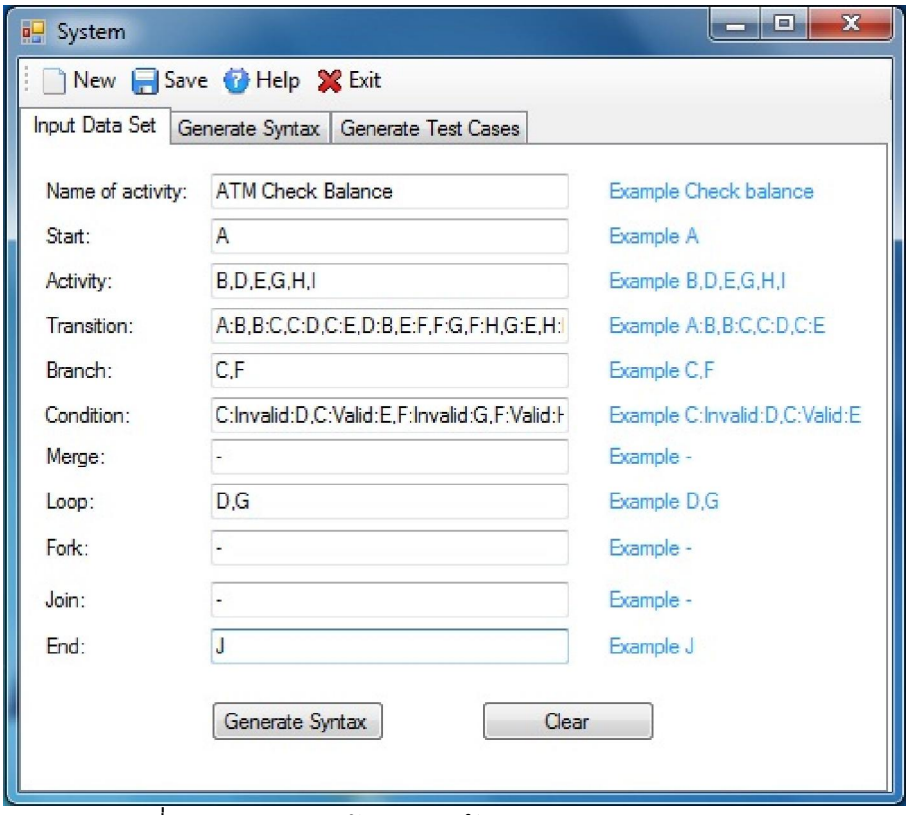


ภาพประกอบที่ 4.9 แผนภาพกิจกรรม Transfer Funds (MCA, 2010)

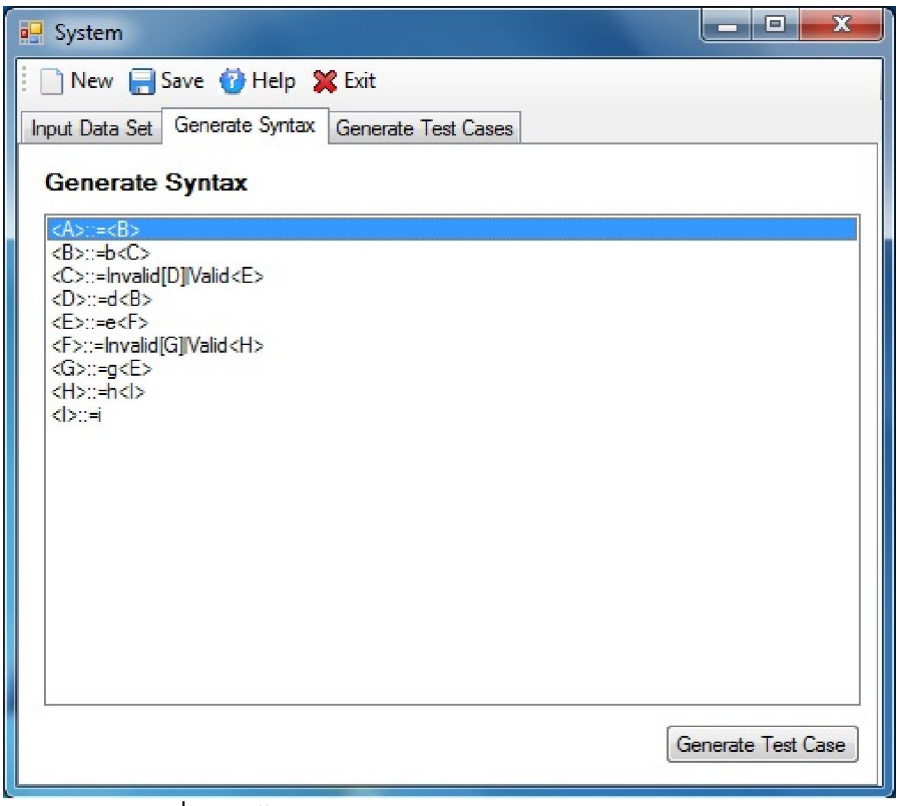
- ตัวอย่างการใช้เครื่องมือต้นแบบในการสร้างกรณีทดสอบสำหรับแผนภาพกิจกรรม Check Balances จากแผนภาพกิจกรรมภาพประกอบที่ 4.6 สามารถเตรียมชุดข้อมูลเบื้องต้นได้ดังภาพประกอบที่ 4.10 จากนั้นทำการป้อนข้อมูลเข้าสู่ระบบ ดังแสดงในภาพประกอบที่ 4.11 ระบบทำการสร้างไวยากรณ์และกรณีทดสอบได้ดังภาพประกอบที่ 4.12 และ 4.13 ตามลำดับ

ATM Check Balance
Start = {A}
Activity = {B,D,E,G,H,I}
Transition = {A:B,B:C,C:D,C:E,D:B,E:F,F:G,F:H,G:E,H:I,I:J}
Branch = {C,F}
Condition = {C:Invalid:D,C:Valid:E,F:Invalid:G,F:Valid:H}
Merge = { }
Loop = {D,G}
Fork = { }
Join = { }
End = {J}

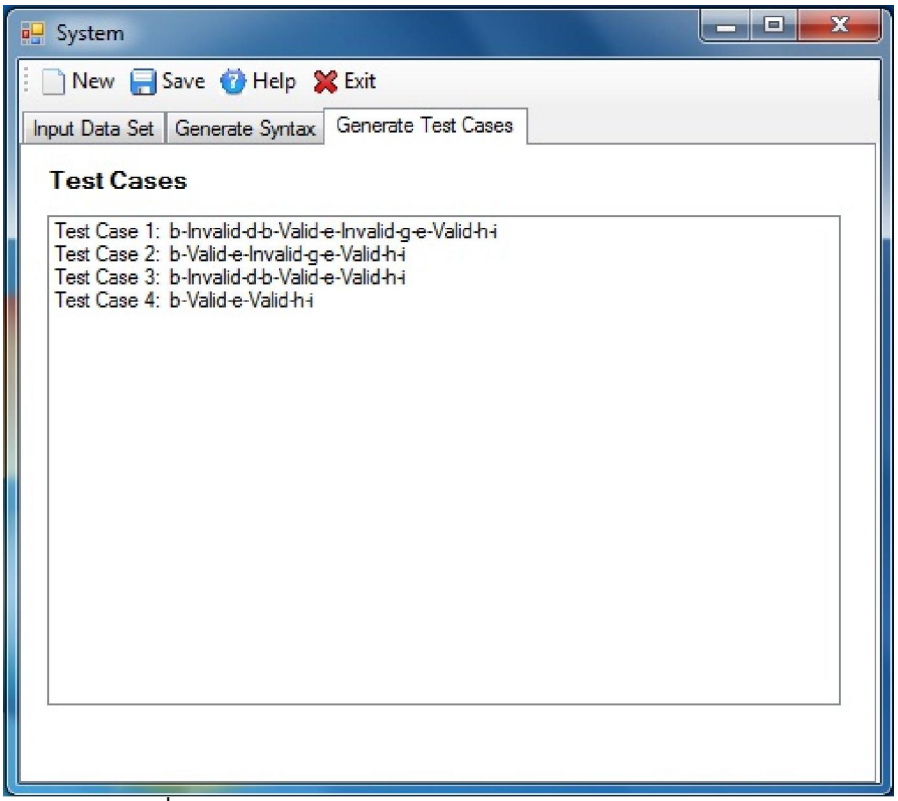
ภาพประกอบที่ 4.10 ชุดข้อมูลเบื้องต้นสำหรับแผนภาพกิจกรรม Check Balances



ภาพประกอบที่ 4.11 การป้อนข้อมูลสำหรับแผนภาพกิจกรรม Check Balances



ภาพประกอบที่ 4.12 ไวยากรณ์สำหรับแผนภาพกิจกรรม Check Balances



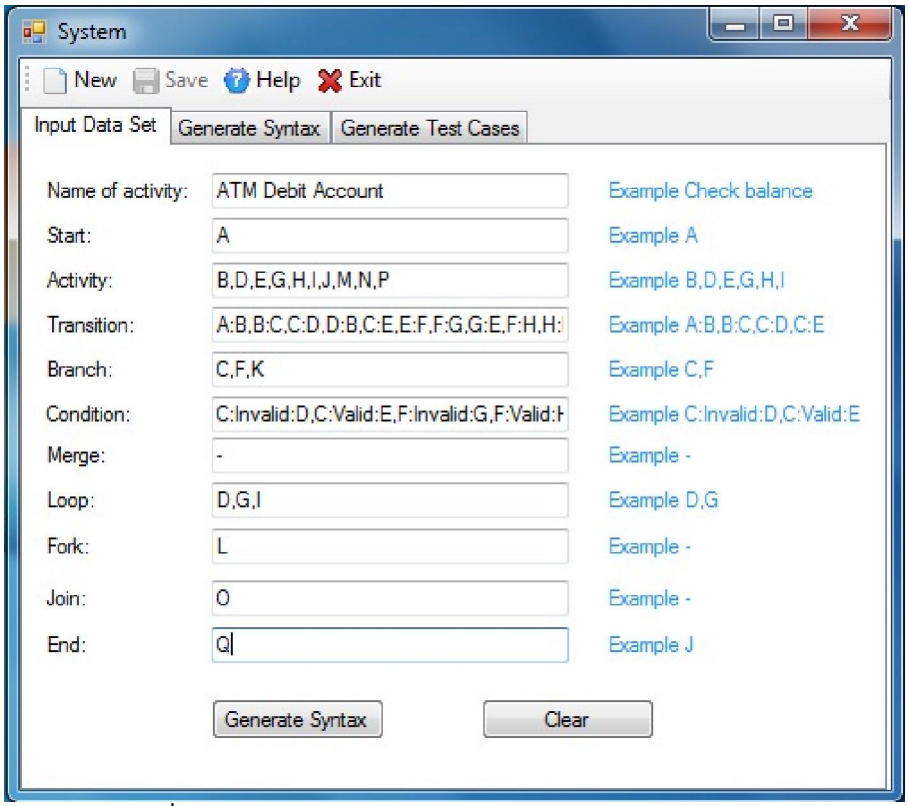
ภาพประกอบที่ 4.13 กรณีทดสอบสำหรับแผนภาพกิจกรรม Check Balances

- ตัวอย่างการใช้เครื่องมือต้นแบบในการสร้างกรณีทดสอบสำหรับแผนภาพ

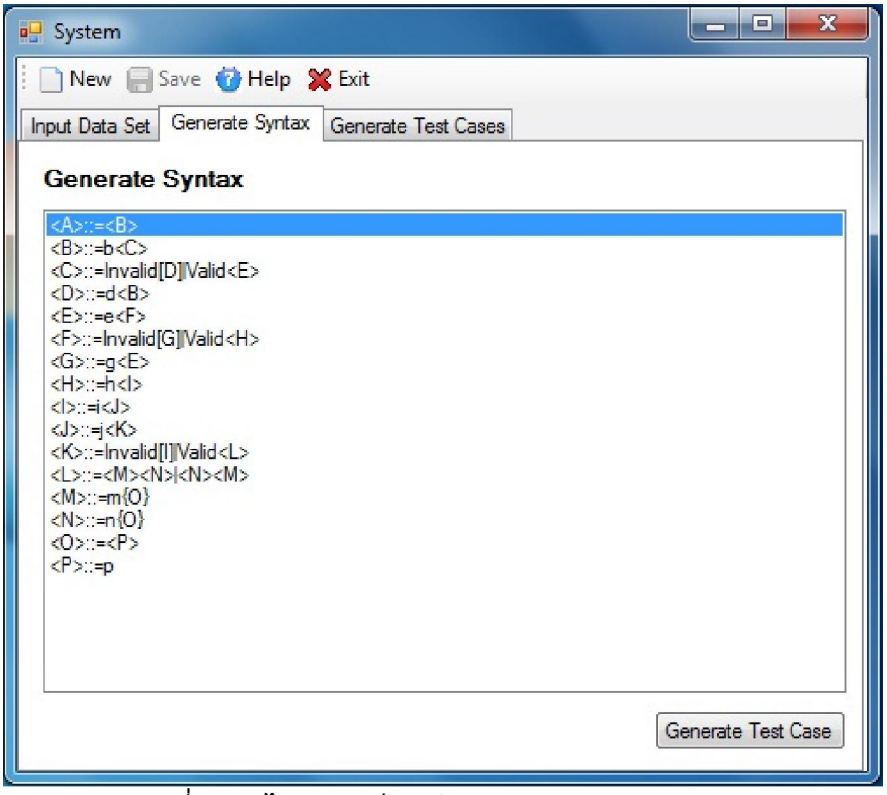
กิจกรรม Deposit Funds จากแผนภาพกิจกรรมภาพประกอบที่ 4.7 สามารถเตรียมชุดข้อมูลเบื้องต้นได้ดังภาพประกอบที่ 4.14 จากนั้นทำการป้อนข้อมูลเข้าสู่ระบบ ดังแสดงในภาพประกอบที่ 4.15 ระบบทำการสร้างไวยากรณ์และกรณีทดสอบได้ดังภาพประกอบที่ 4.16 และ 4.17 ตามลำดับ

ATM Deposit Funds
Start = {A}
Activity = {B,D,E,G,H,I,J,M,N,P}
Transition = {A:B,B:C,C:D,D:B,C:E,E:F,F:G,G:E,F:H,H:I,I:J,J:K,K:I,K:L,L:M,L:N,M:O,N:O,O:P,P:Q}
Branch = {C,F,K}
Condition = {C:Invalid:D,C:Valid:E,F:Invalid:G,F:Valid:H,K:Invalid:I,K:Valid:L}
Merge = { }
Loop = {D,G,I}
Fork = {L}
Join = {O}
End = {Q}

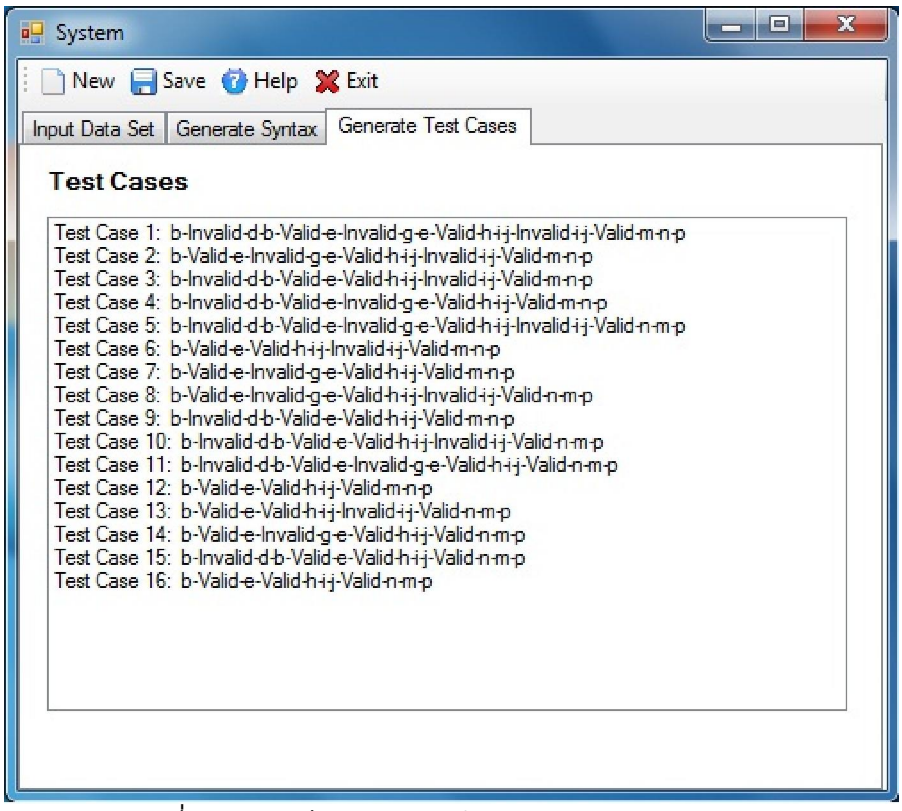
ภาพประกอบที่ 4.14 ชุดข้อมูลเบื้องต้นสำหรับแผนภาพกิจกรรม Deposit Funds



ภาพประกอบที่ 4.15 การป้อนข้อมูลสำหรับแผนภาพกิจกรรม Deposit Funds



ภาพประกอบที่ 4.16 ไวยากรณ์สำหรับแผนภาพกิจกรรม Deposit Funds



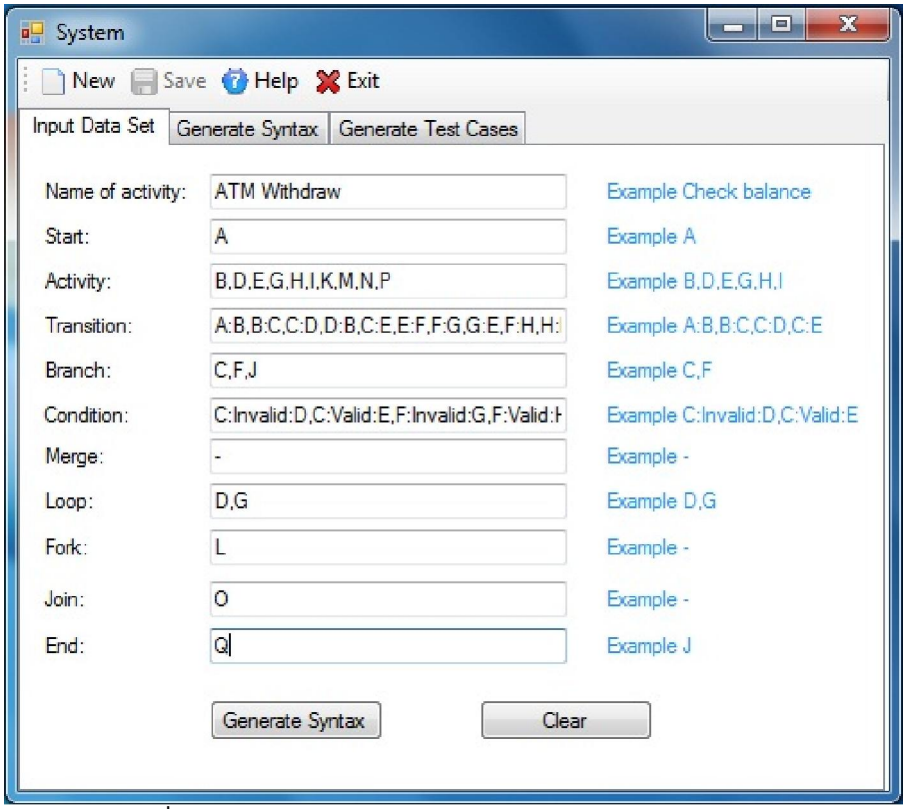
ภาพประกอบที่ 4.17 กรณีทดสอบสำหรับแผนภาพกิจกรรม Deposit Funds

- ตัวอย่างการใช้เครื่องมือต้นแบบในการสร้างกรณีทดสอบสำหรับแผนภาพ

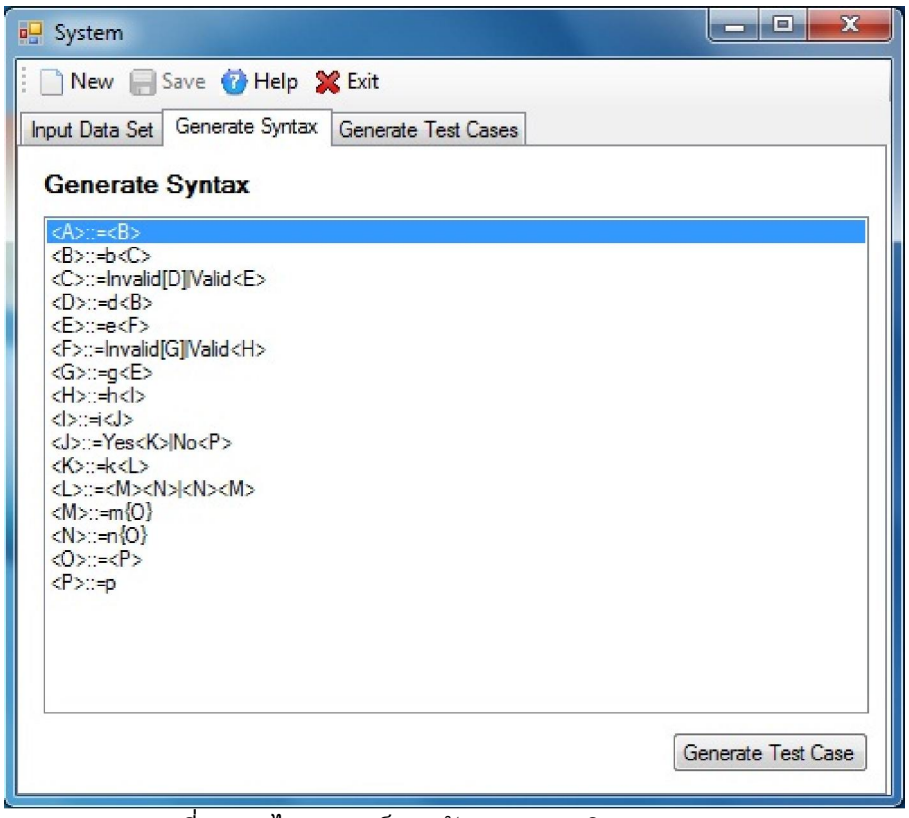
กิจกรรม Withdraw Cash จากแผนภาพกิจกรรมภาพประกอบที่ 4.8 สามารถเตรียมชุดข้อมูลเบื้องต้นได้ดังภาพประกอบที่ 4.18 จากนั้นทำการป้อนข้อมูลเข้าสู่ระบบ ดังแสดงในภาพประกอบที่ 4.19 ระบบทำการสร้างไวยากรณ์และกรณีทดสอบได้ดังภาพประกอบที่ 4.20 และ 4.21 ตามลำดับ

ATM Withdraw
Start = {A}
Activity = {B,D,E,G,H,I,K,M,N,P}
Transition = {A:B,B:C,C:D,D:B,C:E,E:F,F:G,G:E,F:H,H:I,I:J,J:K,K:L,L:M,L:N,M:O,N:O,J:P,O:P,P:Q}
Branch = {C,F,J}
Condition = {C:Invalid:D,C:Valid:E,F:Invalid:G,F:Valid:H,J:Yes:K,J:No:P}
Merge = { }
Loop = {D,G}
Fork = {L}
Join = {O}
End = {Q}

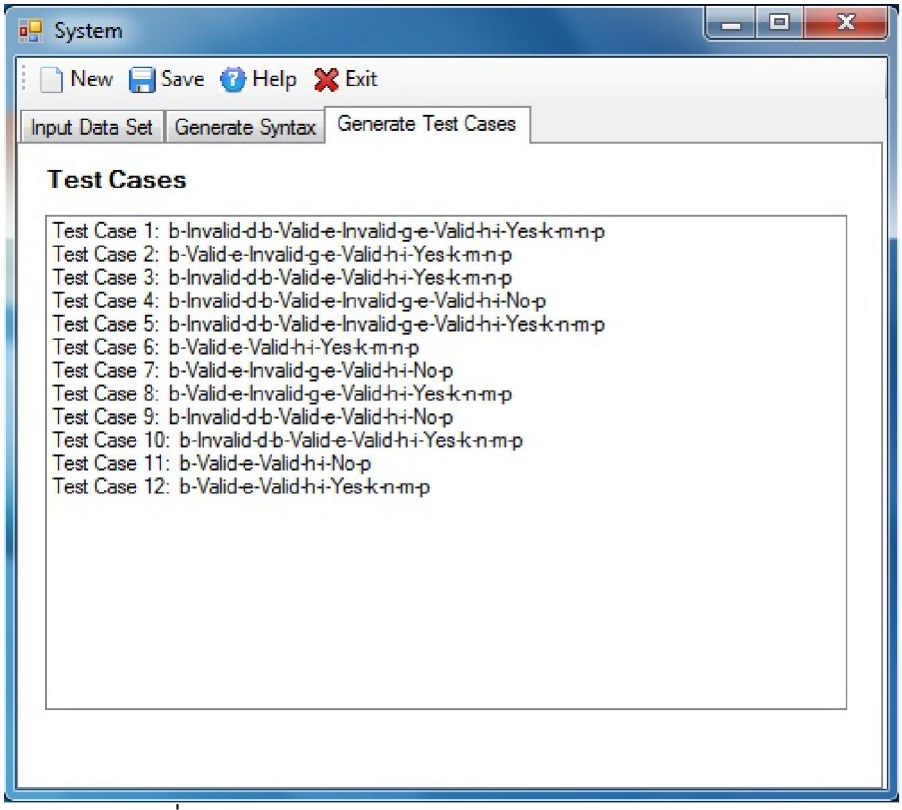
ภาพประกอบที่ 4.18 ชุดข้อมูลเบื้องต้นสำหรับแผนภาพกิจกรรม Withdraw Cash



ภาพประกอบที่ 4.19 การป้อนข้อมูลสำหรับแผนภาพกิจกรรม Withdraw Cash



ภาพประกอบที่ 4.20 ไวยากรณ์สำหรับแผนภาพกิจกรรม Withdraw Cash



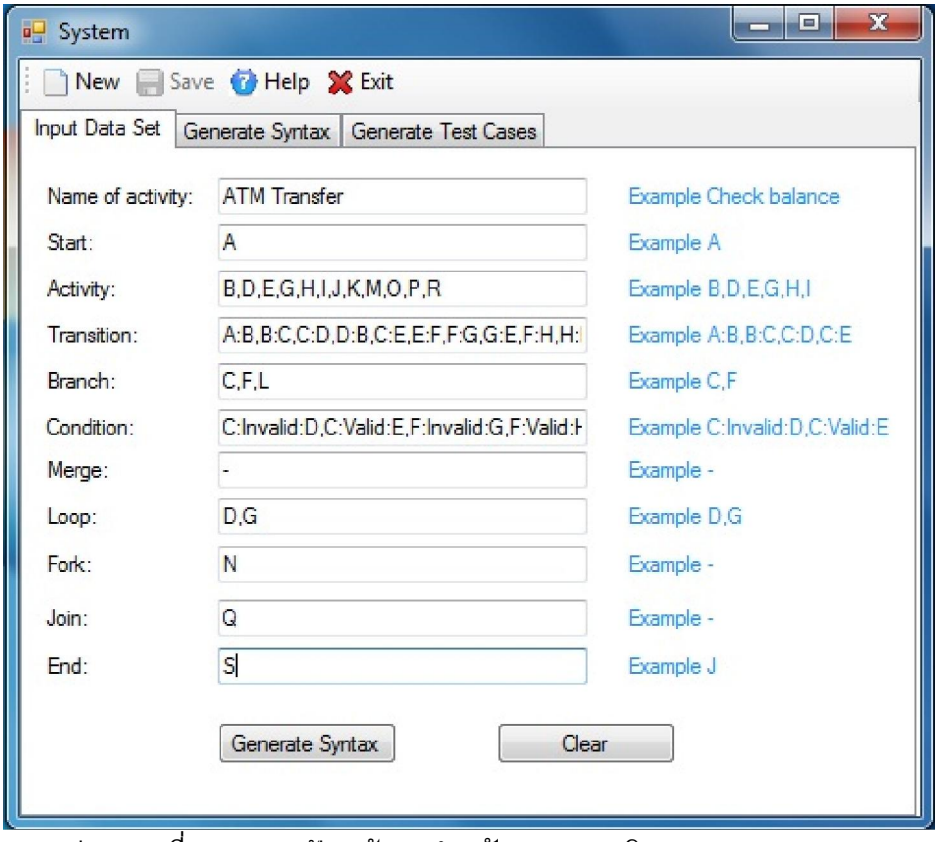
ภาพประกอบที่ 4.21 กรณีทดสอบสำหรับแผนภาพกิจกรรม Withdraw Cash

- ตัวอย่างการใช้เครื่องมือต้นแบบในการสร้างกรณีทดสอบสำหรับแผนภาพ

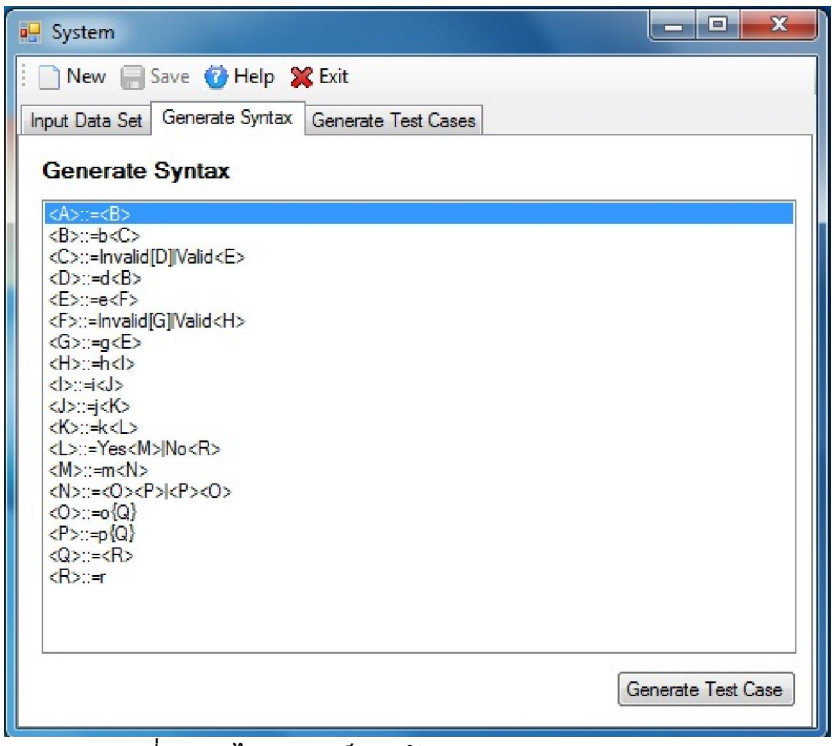
กิจกรรม Transfer Funds จากแผนภาพกิจกรรมภาพประกอบที่ 4.9 สามารถเตรียมชุดข้อมูลเบื้องต้นได้ดังภาพประกอบที่ 4.22 จากนั้นทำการป้อนข้อมูลเข้าสู่ระบบ ดังแสดงในภาพประกอบที่ 4.23 ระบบทำการสร้างไวยากรณ์และกรณีทดสอบได้ดังภาพประกอบที่ 4.24 และ 4.25 ตามลำดับ

ATM Transfer
Start = {A}
Activity = {B,D,E,G,H,I,J,K,M,O,P,R}
Transition = {A:B,B:C,C:D,D:B,C:E,E:F,F:G,G:E,F:H,H:I,I:J,J:K,K:L,L:M,L:R,M:N,N:O,N:P,O:Q,P:Q,Q:R,R:S}
Branch = {C,F,L}
Condition = {C:Invalid:D,C:Valid:E,F:Invalid:G,F:Valid:H,L:Yes:M,L:No:R}
Merge = { }
Loop = {D,G}
Fork = {N}
Join = {Q}
End = {S}

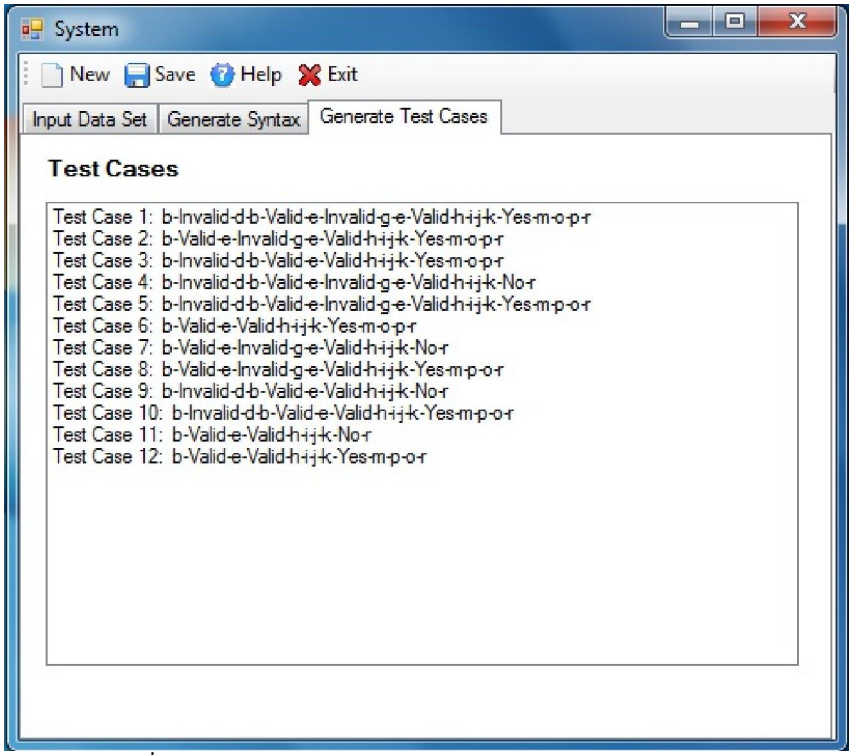
ภาพประกอบที่ 4.22 ชุดข้อมูลเบื้องต้นสำหรับแผนภาพกิจกรรม Transfer Funds



ภาพประกอบที่ 4.23 การป้อนข้อมูลสำหรับแผนภาพกิจกรรม Transfer Funds



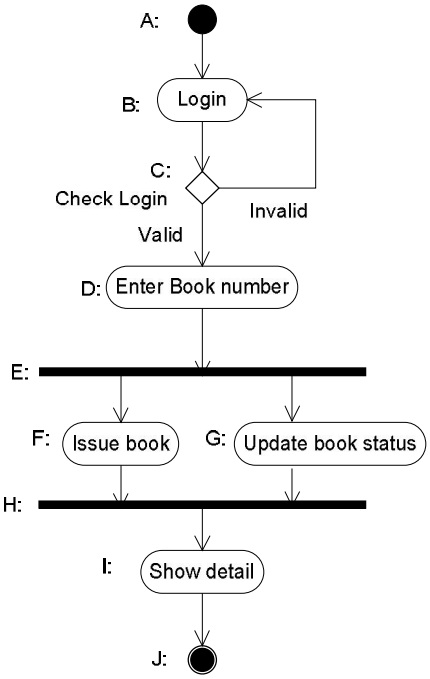
ภาพประกอบที่ 4.24 ไวยากรณ์สำหรับแผนภาพกิจกรรม Transfer Funds



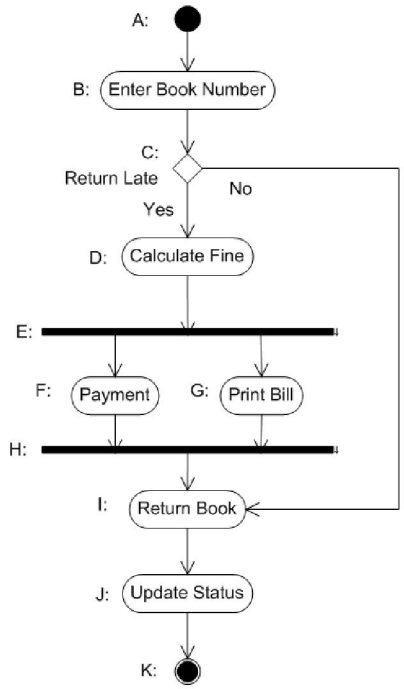
ภาพประกอบที่ 4.25 กรณีทดสอบสำหรับแผนภาพกิจกรรม Transfer Funds

กรณีทดสอบที่สร้างได้ดังภาพประกอบที่ 4.13, 4.17, 4.21 และ 4.25 เป็นกรณีทดสอบทั้งหมดที่จะนำไปใช้ในการทดสอบระบบ ATM

ตัวอย่างที่ 2 การสร้างกรณีทดสอบสำหรับระบบห้องสมุด ซึ่งประกอบด้วย use case ที่ต้องทำการทดสอบทั้งหมด 2 use case คือ Issue Book และ Return Book ซึ่งแผนภาพกิจกรรมที่สอดคล้องกับทั้ง 2 use case แสดงดังภาพประกอบที่ 4.26 และ 4.27 ตามลำดับ



ภาพประกอบที่ 4.26 แผนภาพกิจกรรม Issue Book (MCA, 2010)

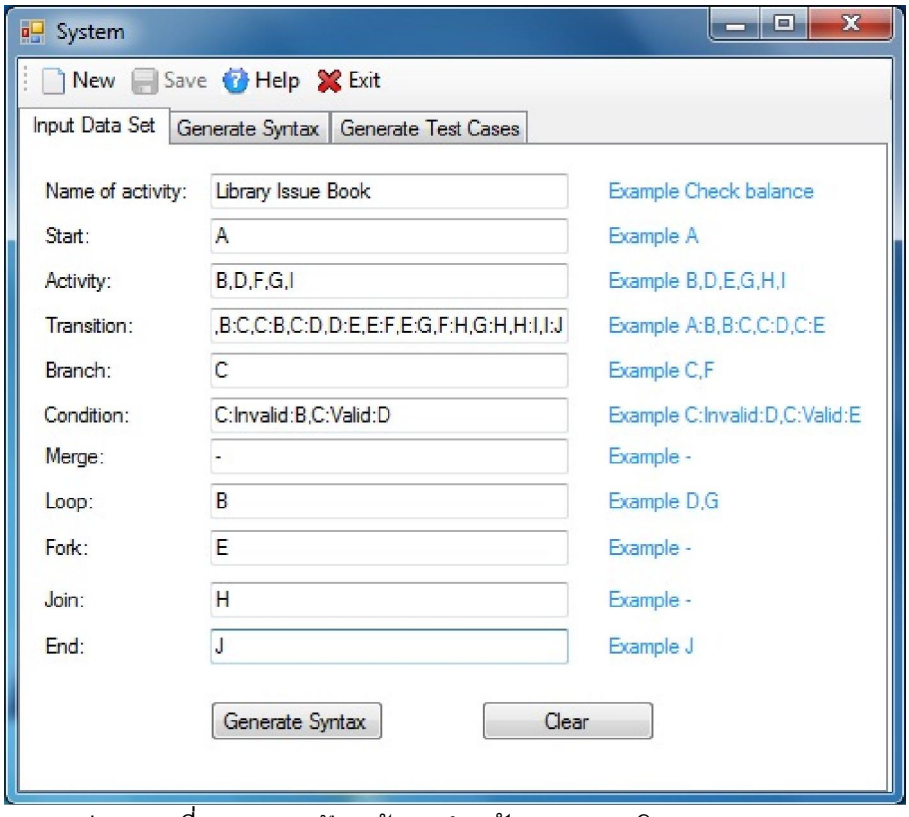


ภาพประกอบที่ 4.27 แผนภาพกิจกรรม Return Book (MCA, 2010)

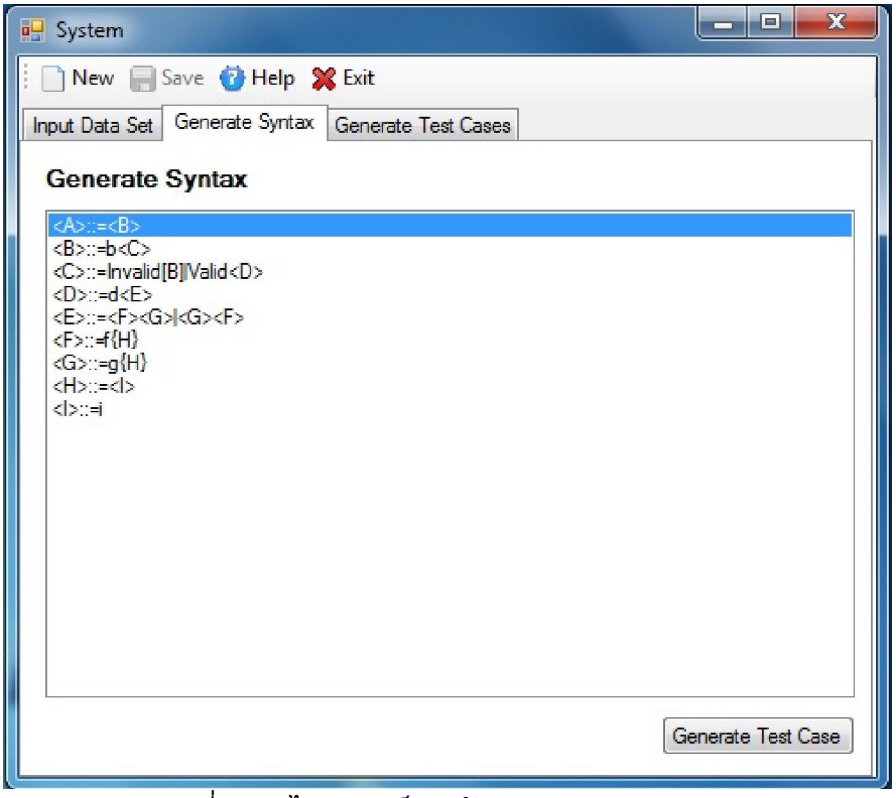
- ตัวอย่างการใช้เครื่องมือต้นแบบในการสร้างกรณีทดสอบสำหรับแผนภาพกิจกรรม Issue Book จากแผนภาพกิจกรรมภาพประกอบที่ 4.26 สามารถเตรียมชุดข้อมูลเบื้องต้นได้ดังภาพประกอบที่ 4.28 จากนั้นทำการป้อนข้อมูลเข้าสู่ระบบ ดังแสดงในภาพประกอบที่ 4.29 ระบบทำการสร้างไวยากรณ์และกรณีทดสอบได้ดังภาพประกอบที่ 4.30 และ 4.31 ตามลำดับ

Library Issue Book
Start = {A}
Activity = {B,D,F,G,I}
Transition = {A:B,B:C,C:B,C:D,D:E,E:F,E:G,F:H,G:H,H:I,I:J}
Branch = {C}
Condition = {C:Invalid:B,C:Valid:D}
Merge = { }
Loop = {B}
Fork = {E}
Join = {H}
End = {J}

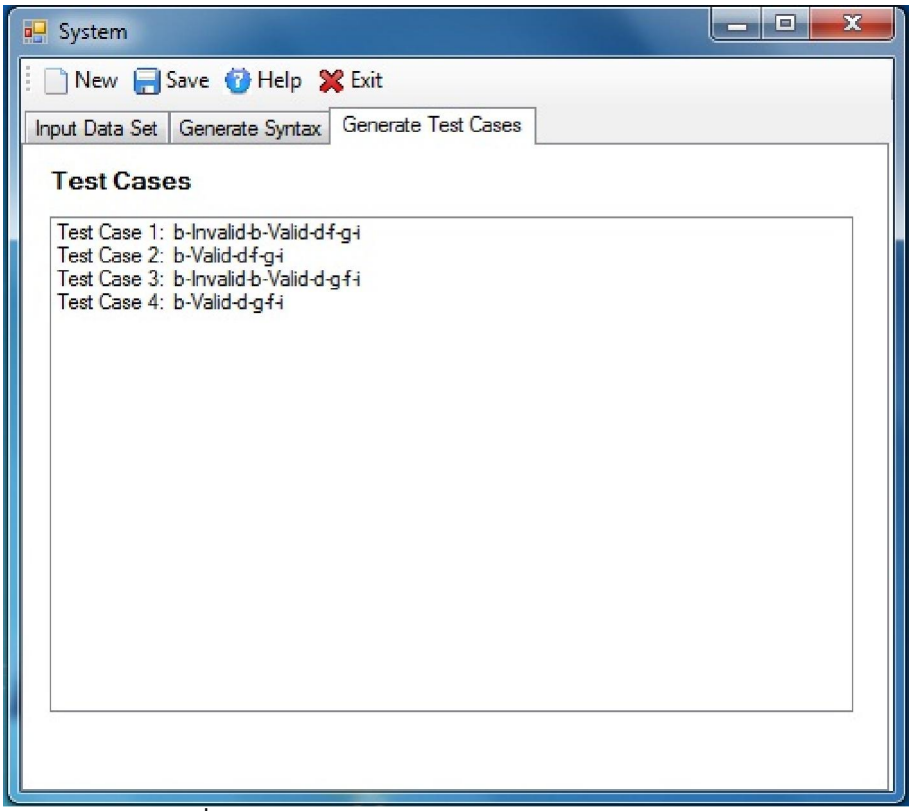
ภาพประกอบที่ 4.28 ชุดข้อมูลเบื้องต้นสำหรับแผนภาพกิจกรรม Issue Book



ภาพประกอบที่ 4.29 การป้อนข้อมูลสำหรับแผนภาพกิจกรรม Issue Book



ภาพประกอบที่ 4.30 ไวยากรณ์สำหรับแผนภาพกิจกรรม Issue Book

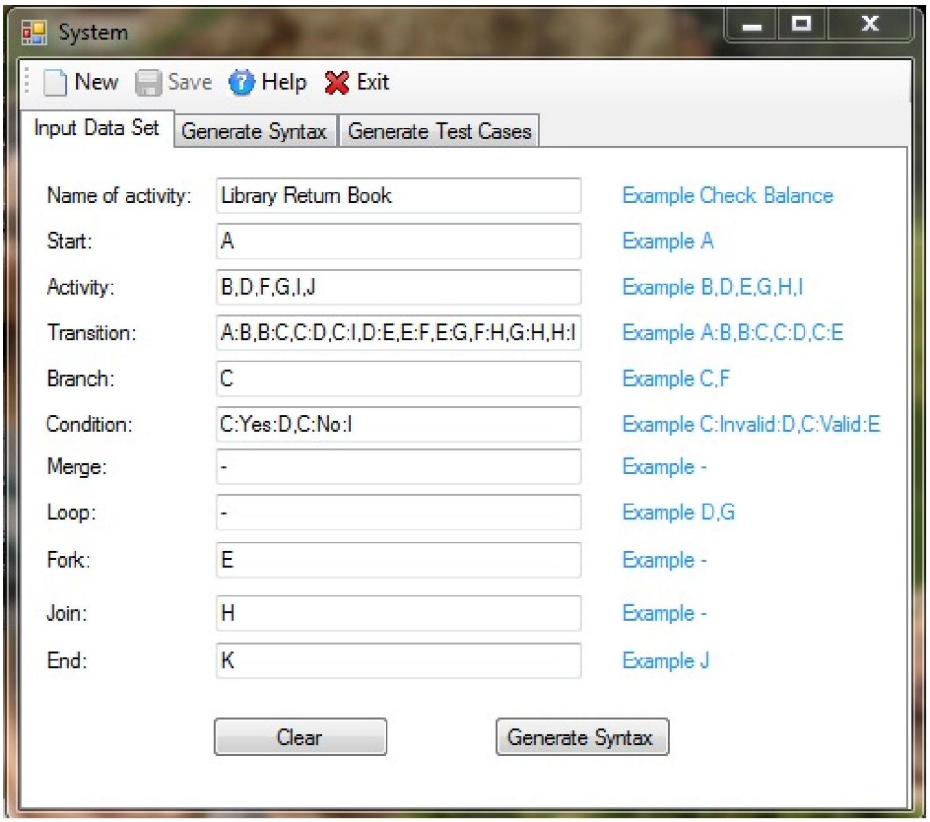


ภาพประกอบที่ 4.31 กรณีทดสอบสำหรับแผนภาพกิจกรรม Issue Book

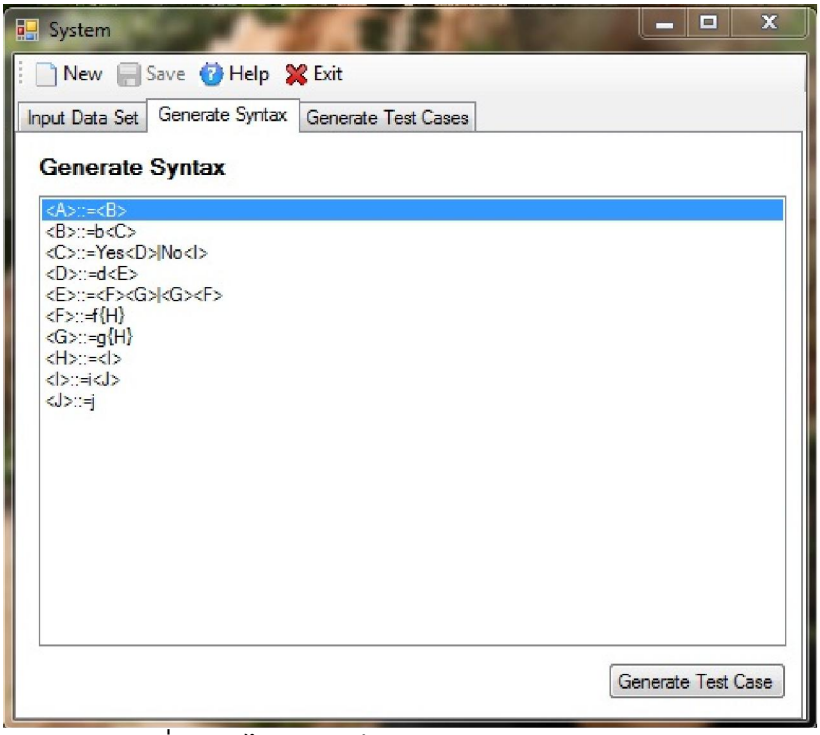
- ตัวอย่างการใช้เครื่องมือต้นแบบในการสร้างกรณีทดสอบสำหรับแผนภาพกิจกรรม Return Book จากแผนภาพกิจกรรมภาพประกอบที่ 4.27 สามารถเตรียมชุดข้อมูลเบื้องต้นได้ดังภาพประกอบที่ 4.32 จากนั้นทำการป้อนข้อมูลเข้าสู่ระบบ ดังแสดงในภาพประกอบที่ 4.33 ระบบทำการสร้างไวยากรณ์และกรณีทดสอบได้ดังภาพประกอบที่ 4.34 และ 4.35 ตามลำดับ

Library Return Book
Start = {A}
Activity = {B,D,E,G,I,J}
Transition = {A:B,B:C,C:D,C:I,D:E,E:F,E:G,F:H,G:H,H:I,I:J,J:K}
Branch = {C}
Condition = {C:Yes:D,C:No:I}
Merge = { }
Loop = { }
Fork = {E}
Join = {H}
End = {K}

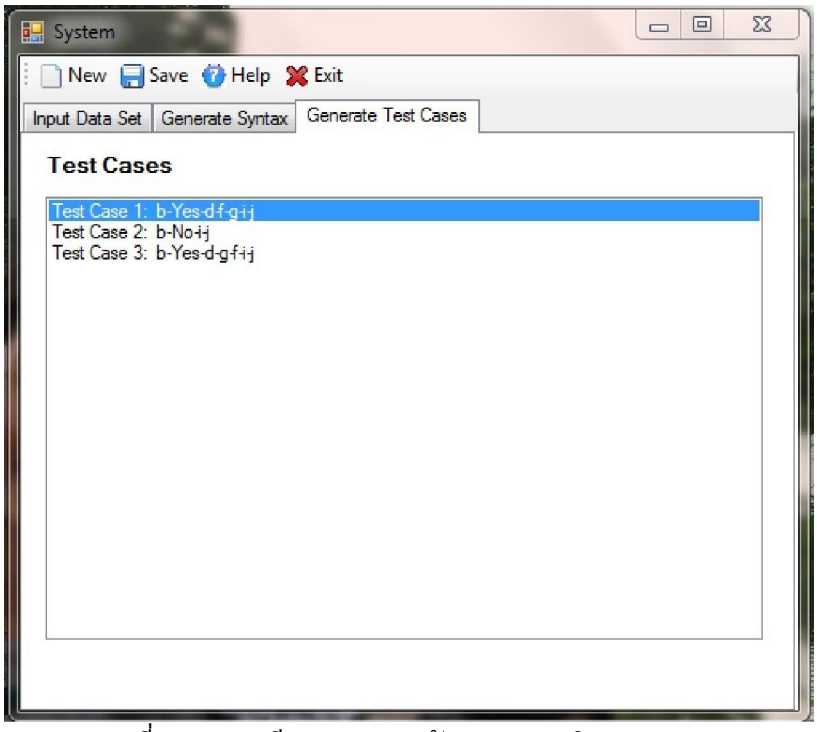
ภาพประกอบที่ 4.32 ชุดข้อมูลเบื้องต้นสำหรับแผนภาพกิจกรรม Return Book



ภาพประกอบที่ 4.33 การป้อนข้อมูลสำหรับแผนภาพกิจกรรม Return Book



ภาพประกอบที่ 4.34 ไวยากรณ์สำหรับแผนภาพกิจกรรม Return Book



ภาพประกอบที่ 4.35 กรณีทดสอบสำหรับแผนภาพกิจกรรม Return Book

กรณีทดสอบที่สร้างได้ดังภาพประกอบที่ 4.31 และ 4.35 เป็นกรณีทดสอบทั้งหมดที่จะนำไปใช้ในการทดสอบระบบห้องสมุด

บทที่ 5

บทสรุปและข้อเสนอแนะ

จากเอกสารในบทที่ผ่านมา ได้กล่าวถึงความสำคัญและที่มาของงานวิจัย การศึกษาทฤษฎีและงานวิจัยที่เกี่ยวข้อง และหลักการที่ได้นำเสนอ จนถึงการพัฒนาเครื่องมือต้นแบบและการทดสอบการใช้งานเครื่องมือต้นแบบ ในบทนี้จะกล่าวถึงบทสรุปและข้อเสนอแนะ

5.1 บทสรุป

การทดสอบซอฟต์แวร์ถือเป็นขั้นตอนหนึ่งที่มีความสำคัญในกระบวนการผลิตซอฟต์แวร์ ซึ่งการทดสอบทำให้ซอฟต์แวร์มีความถูกต้อง ความสมบูรณ์ คุณภาพ และความน่าเชื่อถือเพิ่มขึ้น แต่เนื่องจากซอฟต์แวร์ในปัจจุบันมักมีขนาดใหญ่ และมีความซับซ้อนมากยิ่งขึ้น ส่งผลให้ค่าใช้จ่ายในการทดสอบซอฟต์แวร์เพิ่มมากขึ้นตามไปด้วย ดังนั้น การสร้างกรณีทดสอบได้โดยอัตโนมัติ และกระทำได้ตั้งแต่ช่วงต้นของการพัฒนาซอฟต์แวร์ จะช่วยให้ค่าใช้จ่ายในการทดสอบซอฟต์แวร์ลดลง และส่งผลให้ค่าใช้จ่ายในกระบวนการผลิตซอฟต์แวร์ลดลงได้อีกด้วย

วิทยานิพนธ์นี้ได้นำเสนอการสร้างกรณีทดสอบจากแผนภาพ use case และแผนภาพกิจกรรมบนพื้นฐานของไวยากรณ์ โดยพิจารณาความสัมพันธ์ระหว่าง actor และ use case ทั้งหมดของระบบ เพื่อให้ได้รายการ use-case ทั้งหมดที่จะต้องนำมาดำเนินการในการทดสอบระบบ จากนั้นจึงนำแผนภาพกิจกรรมที่สอดคล้องกับ use case ต่าง ๆนั้นมาเตรียมชุดข้อมูลเบื้องต้นเพื่อสร้างเป็นกรณีทดสอบ โดยการนำชุดข้อมูลเบื้องต้นมาสร้างเป็นตารางการขึ้นต่อกันและตารางเงื่อนไขทางตรรกะ เมื่อได้ทั้งสองตารางแล้ว จึงนำตารางทั้งสองเหล่านี้มาพิจารณาร่วมกันเพื่อสร้างไวยากรณ์ จากนั้นจึงทำการสร้างกรณีทดสอบจากไวยากรณ์ที่ได้

ในงานวิจัยนี้ได้ทำการประเมินประสิทธิภาพของกรณีทดสอบที่ได้จากวิธีการที่นำเสนอ โดยใช้เทคนิค Mutation Testing ซึ่งผลจากการประเมินพบว่า Mutation Score มีค่าค่อนข้างสูง แสดงให้เห็นว่ากรณีทดสอบที่ได้จากวิธีการที่นำเสนอมีประสิทธิภาพในการตรวจจับข้อผิดพลาด นอกจากนี้วิทยานิพนธ์นี้ได้พัฒนาเครื่องมือต้นแบบเพื่อสนับสนุนวิธีการที่นำเสนอ ซึ่งจากการใช้เครื่องมือต้นแบบในการสร้างกรณีทดสอบพบว่า กรณีทดสอบที่ได้มีความถูกต้อง

กรณีทดสอบที่ได้จากวิธีการที่นำเสนอ สามารถช่วยให้นักวิเคราะห์ระบบทำการสร้างกรณีทดสอบเบื้องต้นจากแผนภาพ use case และแผนภาพกิจกรรม โดยกรณีทดสอบนี้จะเป็นประโยชน์ในการนำไปขยายเป็นข้อมูลที่ใช้สำหรับทดสอบระบบ

5.2 ข้อเสนอแนะ

1. วิทยานิพนธ์นี้มีการดำเนินการสร้างกรณีทดสอบสำหรับสัญลักษณ์ fork ในแผนภาพกิจกรรมเพียง 2 รูปแบบ คือการเรียงกิจกรรมจากซ้ายไปขวา และจากขวาไปซ้ายของแผนภาพกิจกรรม ซึ่งอาจทำให้กรณีทดสอบที่ได้ยังไม่ครอบคลุมเท่าที่ควร ในอนาคตจึงควรทำการสร้างกรณีทดสอบสำหรับสัญลักษณ์ fork ในทุกรูปแบบ เพื่อความครอบคลุมยิ่งขึ้น

2. วิทยานิพนธ์นี้ สามารถดำเนินการได้กับกิจกรรมที่อยู่ระหว่างสัญลักษณ์ fork และ สัญลักษณ์ join ที่เป็นเพียงกิจกรรมเดี่ยวๆเท่านั้น ดังนั้นในอนาคตควรขยายขอบเขตให้สามารถดำเนินการกับกิจกรรมการทำงานที่อยู่ระหว่างสัญลักษณ์ fork และ สัญลักษณ์ join ในลักษณะที่เป็นกลุ่มการทำงานได้

3. เครื่องมือต้นแบบที่นำเสนอต้องอาศัยผู้ทดสอบในการเตรียมข้อมูลจากแผนภาพกิจกรรม ซึ่งหากผู้ทดสอบเตรียมชุดข้อมูลผิดพลาด จะส่งผลให้เครื่องมือทำงานผิดพลาดด้วย ดังนั้นในอนาคตควรพัฒนาเครื่องมือที่สามารถสร้างกรณีทดสอบได้โดยอัตโนมัติจากแผนภาพกิจกรรม

4. จากการออกแบบซอฟต์แวร์โดยทั่วไป ไม่จำเป็นต้องออกแบบโดยใช้แผนภาพ use case และแผนภาพกิจกรรมเท่านั้น ดังนั้นในอนาคตควรขยายการดำเนินการโดยนำแนวคิดที่ได้จากวิธีการที่นำเสนอ ไปประยุกต์ใช้เพื่อสร้างกรณีทดสอบบนพื้นฐานของไวยากรณ์จากแผนภาพอื่นๆ

บรรณานุกรม

P.Ammann and J.Offutt. 2008. Introduction to Software Testing. Cambridge University Press. USA.

G. Booch, J. Rumbaugh, and I. Jacobson. 1998. The Unified Modeling Language User Guide. Object Technology Series. Addison-Wesley Longman.

R.W. Sebesta. 2010. Concepts of Programming Language. 9th edition. Pearson Addison Wesley.

C. Sun. 2008. A Transformation-based Approach to Generating Scenario-oriented Test Cases from UML Activity Diagram for Concurrent Applications. Proceeding of 32nd. Annual IEEE International Computer Software and Application Conference. 2008. pp. 160-167.

X. Fan, J. Shu, L. Liu and Q.J. Liang. 2009. Test Case Generation from UML Subactivity and Activity Diagram. Proceeding of 2nd. IEEE International Symposium on Electronic Commerce and Security. 2009. pp. 244-248.

Supaporn Kansomkeat, Phachayanee Thiket and Jeff Offutt. 2010. Generating Test Cases from UML Activity Diagram using the Condition-Classification Tree Method. Proceeding of 2nd. IEEE International Conference on Software Technology and Engineering (ICSTE). 2010. pp.V1-62 – V1-66

M. Sarma and R. Mall. 2007. Automatic Test Case Generation from UML Models. Proceeding of 10th. International Conference on Information Technology. IEEE computer society. 2007. pp. 196-201.

Santosh Kumar Swain, Durga Prasad Mohapatra and Rajib Mall. 2010. Test Case Generatio Based on State and Activity Models. Journal of Object Technology. 2010. vol. 9, no. 5, pp. 1-27.

A.J. Offutt, A. Lee, G. Rothermel, R. Untch, and C. Zapf. 1996. An Experimental Determination of Sufficient Mutant Operators. ACM Transaction on Software Engineering Methodology. 1996. pp. 99-118

FHWA. 2004. V-Model. http://www.fhwa.dot.gov/publications/research/safety/04080/02.cfm (accessed 5/7/2012).

Randy Miller. 1994. Activity Diagram. http://edn.embarcadero.com/article/31863 (accessed 9/6/2012).

Willem Van Galen. 2010. Use case Diagram. http://www.uml-diagrams.org/use-case-diagrams.html (accessed 9/6/2012)

MCA. 2010. UML Diagram. http://www.programsformca.com/2012/03 (accessed 15/9/2012)

ภาคผนวก

ภาคผนวก ก.

ผลงานตีพิมพ์

เรื่อง	การสร้างกรณีทดสอบบนพื้นฐานของไวยากรณ์ AbS โดยใช้แผนภาพกิจกรรม
งานประชุมวิชาการ	The 9th International Joint Conference on Computer Science and Software Engineering (JCSSE 2012)
สถานที่	กรุงเทพมหานคร ประเทศไทย
วันที่	30 พฤษภาคม – 1 มิถุนายน 2555

การสร้างกรณีทดสอบบนพื้นฐานของไวยากรณ์ AbS โดยใช้แผนภาพกิจกรรม

Generation Test case Based on AbS grammar from UML Activity Diagram

กาญจนี เพชรทะนันท์ [1] และ สุภาภรณ์ กานต์สมเกียรติ[2]
ภาควิชาวิทยาการคอมพิวเตอร์ คณะวิทยาศาสตร์ มหาวิทยาลัยสงขลานครินทร์
Emails: nunimopa@hotmail.com[1], supaporn.k@psu.ac.th[2]

บทคัดย่อ

การทดสอบซอฟต์แวร์ เป็นขั้นตอนหนึ่งที่สำคัญในกระบวนการผลิตซอฟต์แวร์ ซึ่งช่วยให้ซอฟต์แวร์มีความถูกต้องและมีความน่าเชื่อถือมากขึ้น ในปัจจุบันการออกแบบซอฟต์แวร์มักถูกออกแบบโดยใช้ภาษาการออกแบบ ภาษาการออกแบบที่นิยมใช้คือภาษาการออกแบบเชิงโมเดล (Unified ModelingLanguage: UML) ซึ่งเป็นภาษาที่แสดงโครงสร้างและรายละเอียดของระบบเชิงวัตถุ แผนภาพกิจกรรม (Activity Diagram) เป็นแผนภาพประเภทหนึ่งใน UML ซึ่งถูกใช้เพื่อแสดงพฤติกรรมของซอฟต์แวร์ บทความนี้เสนอการสร้างกรณีทดสอบจากแผนภาพกิจกรรม โดยการสร้างกรณีทดสอบมีขั้นตอนเริ่มจากการแปลงแผนภาพกิจกรรมให้อยู่ในรูปแบบของไวยากรณ์ที่มีชื่อว่า Activity-based Syntax (AbS) หลังจากนั้นไวยากรณ์ AbS จะถูกใช้เพื่อสร้างกรณีทดสอบ ขั้นตอนวิธีการสร้างกรณีทดสอบที่ได้นำเสนอในบทความนี้ ได้ถูกนำไปประยุกต์ใช้กับ 3 กรณีศึกษา ซึ่งผลลัพธ์ที่ได้แสดงให้เห็นว่าการทดสอบมีประสิทธิภาพในการตรวจจับข้อผิดพลาดได้ดี

คำสำคัญ: การทดสอบซอฟต์แวร์, ภาษาการออกแบบเชิงโมเดล, แผนภาพกิจกรรม

Abstract

Software testing is an important part in software development lift-cycle. It makes software more accurate and reliable. Nowadays, the software is designed by using the design language, widely used in Unified Modeling Language (UML). UML shows the structure and details of object-oriented systems. Activity diagram is UML diagrams used to represent the behavior of the software. In this paper, we proposed a method to generate test cases from UML activity diagram. Initially, an activity diagram is transformed into grammar, called Activity-based Syntax (AbS). Then, the AbS grammar is used to generate test cases. The proposed method was applied on three case studies. The result shows that our tests have ability in fault detection.

Keywords: software testing, UML based testing, activity diagram

1. บทนำ

การทดสอบซอฟต์แวร์ [1] เป็นขั้นตอนที่มีความสำคัญในกระบวนการผลิตซอฟต์แวร์ ซึ่งช่วยให้ซอฟต์แวร์มีความถูกต้อง สมบูรณ์ มีคุณภาพดี และมีความน่าเชื่อถือ โดยใช้ความรู้ทางด้านเทคนิค เพื่อระบุและค้นหาข้อผิดพลาดของซอฟต์แวร์ หากสามารถทดสอบและตรวจจับข้อผิดพลาดได้ก่อนขั้นตอนการพัฒนาซอฟต์แวร์ จะช่วยให้ค่าใช้จ่ายในการกระบวนการพัฒนาซอฟต์แวร์ลดลงได้

ภาษาการออกแบบเชิงโมเดล (Unified Modeling Language: UML) [2] เป็นภาษามาตรฐานที่ช่วยในการวิเคราะห์และออกแบบระบบเชิงวัตถุ UML โดยแสดงโครงสร้างและรายละเอียดต่างๆของระบบ UML ประกอบด้วยแผนภาพหลายชนิดและแผนภาพกิจกรรม (Activity Diagram) เป็นแผนภาพหนึ่งที่ได้รับความนิยมในการนำมาใช้สร้างกรณีทดสอบ ซึ่งแสดงพฤติกรรมของซอฟต์แวร์โดยอธิบายลำดับการทำงานที่เกิดขึ้นในซอฟต์แวร์ตั้งแต่เริ่มต้นจนสิ้นสุด

บทความนี้เสนอการสร้างกรณีทดสอบจากแผนภาพกิจกรรมที่ถูกสร้างขึ้นในขั้นตอนการออกแบบซอฟต์แวร์ โดยในขั้นตอนแรกจะแปลงแผนภาพกิจกรรมให้อยู่ในรูปแบบของไวยากรณ์ที่มีชื่อว่า Activity-based Syntax (AbS) ขั้นตอนถัดไปไวยากรณ์ AbS จะถูกใช้เพื่อสร้างกรณีทดสอบ

บทความนี้ประกอบด้วยหัวข้อต่างๆดังต่อไปนี้ หัวข้อที่ 2 แนะนำทฤษฎีและงานวิจัยที่เกี่ยวข้อง หัวข้อที่ 3 เสนอการสร้างกรณีทดสอบจากไวยากรณ์ AbS หัวข้อที่ 4 แสดงการตรวจสอบประสิทธิภาพของกรณีทดสอบ และหัวข้อที่ 5 บทสรุปและงานในอนาคต

2. ทฤษฎีและงานวิจัยที่เกี่ยวข้อง

ในส่วนนี้นำเสนอทฤษฎีพื้นฐานที่ใช้ ซึ่งประกอบด้วย 3 ส่วนย่อย ส่วนแรกอธิบายไวยากรณ์บีเอ็นเอฟ (Backus-Naur Form: BNF) ส่วนที่สองอธิบายแผนภาพกิจกรรม และส่วนที่สามกล่าวถึงงานวิจัยที่เกี่ยวข้อง

2.1 ไวยากรณ์ BNF (Backus-Naur Form)

BNF (Backus-Naur Form) [3] เป็นอภิภาษา (Metalanguage) ที่ใช้อธิบายไวยากรณ์ของภาษาการโปรแกรม ซึ่งถือเป็นส่วนสำคัญของการสร้าง Compiler ไวยากรณ์ BNF ประกอบด้วยกฎต่างๆ ที่อยู่ในรูปโปรดักชั่น (Production) โดยแต่ละโปรดักชั่นมีรูปแบบดังนี้

<single nonterminal> ::= sequence of terminal or <nonterminal>

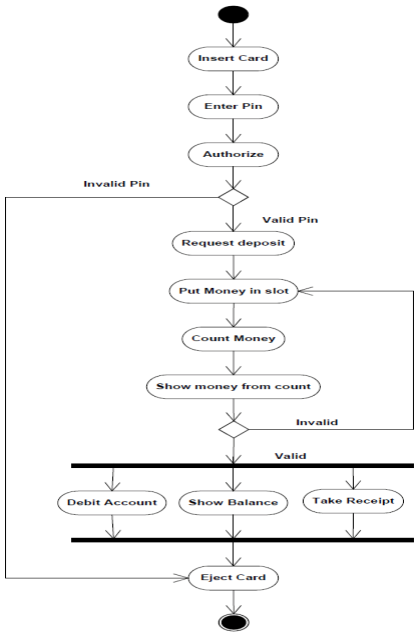
แต่ละโปรดักชั่นจะประกอบด้วยสัญลักษณ์ที่อยู่ทางด้านซ้าย (Left-Hand Side: LHS) และสัญลักษณ์ที่อยู่ทางด้านขวา (Right-Hand Side: RHS) ของสัญลักษณ์ ::= (defined as) โดยที่ LHS จะเป็นสัญลักษณ์ไม่สิ้นสุด นั่นคือสัญลักษณ์ที่อยู่ในเครื่องหมาย < > และ RHS เป็นลำดับของสัญลักษณ์สิ้นสุดหรือสัญลักษณ์ไม่สิ้นสุดก็ได้ รูปที่ 1 แสดงตัวอย่างไวยากรณ์ BNF

```
< number> ::= <digit> | <number><digit>
<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

รูปที่ 1 ตัวอย่างไวยากรณ์ BNF

2.2 แผนภาพกิจกรรม

แผนภาพกิจกรรมแสดงพฤติกรรมของซอฟต์แวร์โดยอธิบายลำดับการทำงานที่เกิดขึ้นในซอฟต์แวร์ตั้งแต่เริ่มต้นจนสิ้นสุด แผนภาพกิจกรรมจะมีจุดเริ่มต้นและจุดสิ้นสุดเพียงจุดเดียว โดยจุดเริ่มต้นจะแสดงด้วยสัญลักษณ์ ● เรียกว่า Start activity และจุดสิ้นสุดจะแสดงด้วยสัญลักษณ์ ◉ เรียกว่า Final activity กิจกรรมการทำงานแสดงด้วยสัญลักษณ์ (ActionState1) เรียกว่า Activity และระหว่างกิจกรรมจะเชื่อมด้วยลูกศรมีทิศทาง ⟶ เรียกว่า transition ส่วนเงื่อนไขการตัดสินใจใช้สัญลักษณ์ ◇ เรียกว่า Branch ซึ่งมี transition เข้าอย่างน้อยหนึ่ง transition และมี transition ออกอย่างน้อยสอง transition โดยบน transition ออกจะถูกระบุด้วยข้อความเงื่อนไขทางตรรกะ และสัญลักษณ์ ◇ ยังใช้แสดงการรวมหรือ Merge อีกด้วย การแบ่งส่วนการทำงานใช้กรอบสี่เหลี่ยมเรียกว่า Swim lane ส่วนบนสุดของแต่ละกรอบสี่เหลี่ยมจะถูกระบุด้วยชื่อของส่วนการทำงานนั้นๆ การแตกกิจกรรมที่ทำงานขนานกันใช้สัญลักษณ์ ┴┴┴ เรียกว่า Fork โดยมี transition เข้า synchronization bar อย่างน้อยหนึ่ง transition และออกอย่างน้อยสอง transition การรวมกิจกรรมจากการทำงานขนานกันใช้สัญลักษณ์ ┬┬┬ เรียกว่า Join โดยมี transition เข้า synchronization bar อย่างน้อยสอง transition และออกอย่างน้อยหนึ่ง transition



รูปที่ 2 แผนภาพกิจกรรมของเครื่องเบิกถอนเงินอัตโนมัติ

รูปที่ 2 แสดงแผนภาพกิจกรรมของเครื่องรับฝากเงินอัตโนมัติ (Cash Deposit Machine: CDM) การดำเนินการเริ่มต้นจากการใส่บัตรและผู้ใช้กรอกรหัส ระบบตรวจสอบความถูกต้องรหัส หากรหัสไม่ถูกต้อง ระบบจะคืนบัตรให้ผู้ใช้ แต่เมื่อรหัสถูกต้อง ผู้ใช้จะส่งคำร้องขอฝากเงิน จากนั้นระบบจะเปิดช่องรับเงิน ผู้ใช้ใส่เงินที่ต้องการฝากเข้าไปในช่องรับเงิน ระบบนับจำนวนเงินที่ในช่องรับเงิน และแสดงจำนวนเงินที่ได้จากการนับ ผู้ใช้ตรวจสอบความถูกต้องของจำนวนเงิน หากจำนวนเงินไม่ถูกต้อง ระบบจะเปิดช่องรับเงินเพื่อคืนเงินให้กับผู้ใช้และให้ผู้ใช้ใส่จำนวนเงินเข้าไปใหม่อีกครั้ง ระบบตรวจนับเงินและแสดงจำนวนเงินจากการนับ ผู้ใช้ตรวจสอบความถูกต้องของจำนวนเงิน หากจำนวนเงินไม่ถูกต้อง ระบบจะทำตามกระบวนการเดิม แต่เมื่อจำนวนเงินถูกต้อง ระบบจะเพิ่มยอมเงินในบัญชี แสดงยอดเงินคงเหลือทั้งหมด และออกใบเสร็จรับเงิน โดยเรียงลำดับการดำเนินการส่วนนี้อย่างไรก็ได้ หลังจากนั้นระบบจะคืนบัตรให้กับผู้ใช้

2.3 งานวิจัยที่เกี่ยวข้อง

แผนภาพกิจกรรมเป็นแผนภาพหนึ่งของ UML และเป็นแผนภาพที่ได้รับความนิยมในการนำมาสร้างกรณีทดสอบ เช่น Sapna P.G. and H. Mohanty [4] เสนองานวิจัยเรื่อง “Prioritization of Scenarios based on UML Activity Diagrams” เพื่อหาลำดับความสำคัญที่ดีที่สุดของชุดทดสอบในแผนการทดสอบ โดยแปลงแผนภาพกิจกรรมให้อยู่ใน

รูปแบบโครงสร้างต้นไม้ และกำหนดค่าน้ำหนักให้กับโหนดและเส้นทั้งหมดของต้นไม้ จากนั้นคำนวณค่าน้ำหนักของเส้นทางต่างๆ และจัดลำดับความสำคัญของแต่ละเส้นทางทดสอบโดยเรียงจากผลรวมของค่าน้ำหนักของเส้นทางจากน้อยไปมาก Chang-ai Sun [5] เสนองานวิจัยเรื่อง “A Transformation-based Approach to Generating Scenario-oriented Test Case from UML Activity Diagram for Concurrent Applications” เพื่อสร้างกรณีทดสอบจากแผนภาพกิจกรรม โดยแปลงแผนภาพกิจกรรมให้อยู่ในรูปตาราง activity extracted และตาราง intermediate จากนั้นนำตารางมาสร้างเป็นโครงสร้างต้นไม้และสร้างแผนการทดสอบ โดยการท่องเข้าไปตามแนวกว้างของโครงสร้างต้นไม้ สุดท้ายสร้างกรณีทดสอบจากแผนการทดสอบ จากงานวิจัยที่กล่าวมาข้างต้น พบว่าใช้พื้นฐานกราฟและไม่พิจารณาลำดับของกิจกรรมที่ดำเนินการขนานกัน ดังนั้นบทความนี้จึงเสนอการสร้างกรณีทดสอบจากแผนภาพกิจกรรม โดยให้ความสำคัญกับลำดับของกิจกรรมที่ดำเนินการขนานกัน

3. การสร้างกรณีทดสอบบนพื้นฐานของไวยากรณ์ AbS โดยใช้แผนภาพกิจกรรม

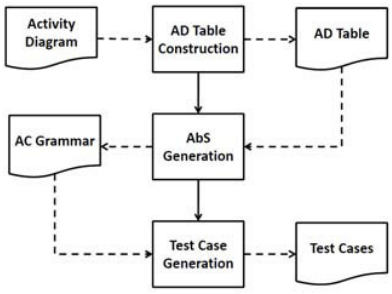
วิธีการที่นำเสนอแสดงในรูปที่ 3 ประกอบด้วยขั้นตอนหลักๆ สามขั้นตอนดังต่อไปนี้

1) สร้างตารางการขึ้นต่อกันของกิจกรรม (Activity Dependency – AD Table Construction) : กำหนดชื่อสัญลักษณ์ให้กับกิจกรรมและกำหนดกิจกรรมที่ขึ้นต่อกัน

2) สร้างไวยากรณ์บนพื้นฐานของแผนภาพกิจกรรม (Activity-based Syntax – AbS Generation) : วิเคราะห์แผนภาพกิจกรรมและสร้างไวยากรณ์ AbS

3) สร้างกรณีทดสอบ (Test Case Generation) : สร้างกรณีทดสอบโดยใช้ไวยากรณ์ AbS ที่สร้างจากขั้นตอนก่อนหน้า

ในงานวิจัยนี้ใช้วิธีการ Mutation Testing ในการตรวจสอบประสิทธิภาพของกรณีทดสอบที่ได้จากวิธีที่นำเสนอ



รูปที่ 3 สถาปัตยกรรมของวิธีการที่นำเสนอ

3.1 สร้างตารางการขึ้นต่อกันของกิจกรรม (Activity Dependency – AD Table Construction)

ในขั้นตอนนี้ตาราง AD จะถูกสร้างขึ้นโดยพิจารณาข้อมูลต่างๆ จากแผนภาพกิจกรรม ตาราง AD แสดงชื่อสัญลักษณ์ที่ใช้อ้างถึงกิจกรรมต่างๆ และระบุการขึ้นต่อกันของกิจกรรมเหล่านั้น โดยตาราง AD ประกอบด้วย 4 คอลัมน์ต่อไปนี้

- *Defined Name:* ชื่อที่กำหนดให้กับกิจกรรม (activity) ต่างๆที่ปรากฏในแผนภาพกิจกรรม โดยใช้ตัวอักษรภาษาอังกฤษพิมพ์เล็กเรียงตามลำดับ
- *Symbol Name:* ชื่อที่กำหนดให้กับสัญลักษณ์ต่างๆที่ปรากฏในแผนภาพกิจกรรม โดยใช้ตัวอักษรภาษาอังกฤษพิมพ์ใหญ่
- *Symbol:* สัญลักษณ์ต่างๆที่ปรากฏในแผนภาพกิจกรรม
- *Dependency:* ชื่อของสัญลักษณ์ต่างๆที่มีผลต่อสัญลักษณ์ที่กำลังพิจารณาอยู่

ตารางที่ 1 แสดงตาราง AD ที่ได้จากแผนภาพกิจกรรมของเครื่องรับฝากเงินอัตโนมัติที่อธิบายในหัวข้อ 2.2

ตาราง 1 ตาราง AD สำหรับแผนภาพกิจกรรมของเครื่องรับฝากเงินอัตโนมัติ

Defined Name	Symbol Name	Symbol	Dependency
a	A	Insert Card	Start
b	B	Enter Pin	A
c	C	Authorize	B
-	D	Branch (1)	C
e	E	Request deposit	D
f	F	Put money in slot	E
g	G	Count Money	F
h	H	Show money from count	G
-	I	Branch (2)	H
-	J	Fork (1)	I
k	K	Debit Account	J
l	L	Show Balance	J
m	M	Take Receipt	J
-	N	Join (1)	K,L,M
o	O	Eject Card	N

3.2 สร้างไวยากรณ์บนพื้นฐานของแผนภาพกิจกรรม (Activity-based Syntax – AbS Generation)

ไวยากรณ์ AbS ถูกสร้างขึ้นโดยการวิเคราะห์แผนภาพกิจกรรม โดยไวยากรณ์ AbS ประกอบด้วยโปรดักชั่นต่างๆ ซึ่งในการสร้างโปรดักชั่น จะพิจารณาแต่ละองค์ประกอบในแผนภาพกิจกรรม โดยแต่ละโปรดักชั่นแบ่งออกเป็นสองส่วน คือ ส่วนซ้าย (Left Hand side: LHS) และส่วนขวา (Right Hand Side: RHS) ของสัญลักษณ์ ::= (defined

as) ในการสร้างไวยากรณ์ AbS ประกอบด้วยขั้นตอนหลัก 2 ขั้นตอนคือ 1) สร้าง โปรดักชั่นและ 2) ระบุกิจกรรมที่ทำให้เกิดการวนซ้ำ

ขั้นตอนที่ 1 ตาราง AD ที่ถูกสร้างขึ้นจากขั้นตอนก่อนหน้าจะถูกใช้ในขั้นตอนการสร้างโปรดักชั่น โดยพิจารณาแต่ละองค์ประกอบในแผนภาพกิจกรรม เพื่อสร้างเป็นโปรดักชั่น โดยจะแทนส่วน LHS ของโปรดักชั่นด้วยชื่อของสัญลักษณ์ที่ปรากฏในคอลัมน์ Dependency ของตาราง AD ขององค์ประกอบที่กำลังพิจารณาในเครื่องหมาย < และ > และแทนส่วน RHS โดยพิจารณาตามประเภทขององค์ประกอบที่กำลังพิจารณา ดังต่อไปนี้

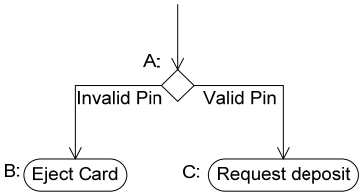
- Activity State: แทนส่วน RHS ด้วยชื่อของกิจกรรมที่ปรากฏในคอลัมน์ Defined Name และตามด้วยชื่อของสัญลักษณ์ลำดับถัดไปที่ปรากฏในคอลัมน์ Symbol Name ภายในเครื่องหมาย < และ >
- Branch: แทนส่วน RHS โดยพิจารณาทุก transition ที่ออกจาก branch โดยแต่ละ transition จะแทนด้วยเงื่อนไขหรือนิพจน์ทางตรรกะที่สอดคล้องกับ transition นั้น และตามด้วย ชื่อของสัญลักษณ์ที่ปรากฏที่ปลายของ transition ภายในเครื่องหมาย < และ > เมื่อทุก transition ถูกแทนครบเรียบร้อยแล้ว จะนำส่วนการแทนเหล่านั้นมาเรียงกันและกั้นด้วยสัญลักษณ์เส้นตรงแนวตั้ง (|) รูปที่ 4 แสดงตัวอย่างของ branch ซึ่งสามารถแทนได้เป็น *Invalid Pin*<B> | *Valid Pin*<C>
- Fork และ Join: แทนส่วน RHS โดยแต่ละกิจกรรมที่ดำเนินการขนานกันอยู่ระหว่าง fork และ join จะถูกแทนด้วยชื่อของสัญลักษณ์จากตาราง AD ภายในเครื่องหมาย < และ > และแต่ละกิจกรรมระหว่าง fork และ join ถูกแทนเรียบร้อยแล้ว จะนำมาเรียงลำดับในสองลักษณะคือ เรียงจากด้านซ้ายไปขวา และเรียงจากด้านขวาไปซ้าย และกั้นระหว่างการเรียงกันทั้งสองลักษณะด้วยสัญลักษณ์เส้นตรงแนวตั้ง รูปที่ 5 แสดงตัวอย่างของ fork และ join ซึ่งสามารถแทนได้เป็น <A><B><C> | <C><B><A>

ขั้นตอนที่ 2 การระบุกิจกรรมที่ทำให้เกิดการวนซ้ำ โดยพิจารณาทุกองค์ประกอบที่พบในแผนภาพกิจกรรมไปทีละองค์ประกอบ เริ่มจากจุดเริ่มต้นจนถึงจุดสิ้นสุด โดยแยกพิจารณาเป็น 2 กรณีดังนี้

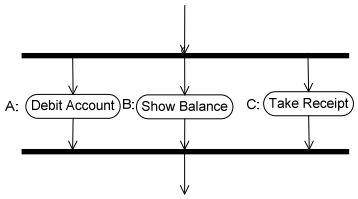
กรณีที่ 1: เกิดการวงซ้ำที่ Branch จะพิจารณา transition ที่ออกจาก branch แล้วทำให้เกิดการวนซ้ำ โดยดูว่า transition นั้นชี้ไปที่องค์ประกอบใด จะทำการเปลี่ยนสัญลักษณ์ < > ขององค์ประกอบนั้นในโปรดักชั่นไปเป็นสัญลักษณ์วงเล็บก้ามปู ([ ]) รูปที่ 6 แสดงตัวอย่างการเกิดการวนซ้ำที่ Branch ทำให้ได้โปรดักชั่น *Invalid Pin*[A] | *Valid Pin*<D> ซึ่งถูกปรับเปลี่ยนมาจากโปรดักชั่น *Invalid Pin*<A> | *Valid Pin*<D>

กรณีที่ 2: เกิดการวนซ้ำที่องค์ประกอบอื่นที่ไม่ใช่ branch จะหาองค์ประกอบก่อนหน้าจากองค์ประกอบที่เกิดการรวนซ้ำไปทีละองค์ประกอบ จนกว่าจะพบ branch จากนั้นจะกำหนด branch ตำแหน่งนั้นเป็นตำแหน่งที่เกิดการวนซ้ำ และดำเนินการเช่นเดียวกับกรณีที่ 1 รูปที่ 7 แสดงตัวอย่างการเกิดการวนซ้ำที่องค์ประกอบอื่นที่ไม่ใช่ branch โดยสามารถแทนได้เป็นโปรดักชั่น Invalid *Pin* [D] | *Valid Pin*<E> ซึ่งเปลี่ยนมาจากโปรดักชั่น Invalid *Pin*<D> | *Valid Pin*<E>

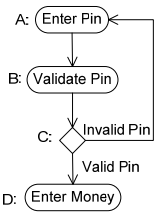
ในบทความนี้ กำหนดให้ข้อความภายในเครื่องหมาย < และ > ในไวยากรณ์ AbS เรียกว่า **AbS state**



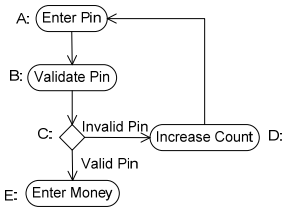
รูปที่ 4 ตัวอย่างของ branch



รูปที่ 5 ตัวอย่าง Fork และ Join

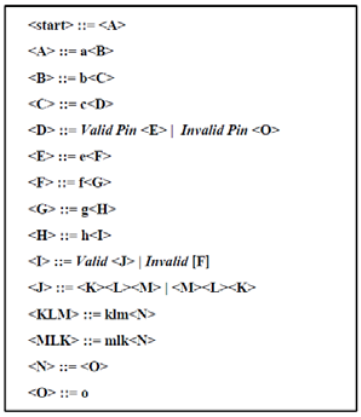


รูปที่ 6 ตัวอย่างการเกิดการวนซ้ำที่ Branch



รูปที่ 7 ตัวอย่างการเกิดการวนซ้ำที่องค์ประกอบอื่นที่ไม่ใช่ branch

จากตัวอย่างแผนภาพกิจกรรมของเครื่องรับฝากเงินอัตโนมัติซึ่งได้กล่าวในหัวข้อ 2.2 สามารถแทนด้วยไวยากรณ์ AbS ดังแสดงในรูปที่ 8



รูปที่ 8 ไวยากรณ์ AbS สำหรับเครื่องรับฝากเงินอัตโนมัติ

3.3 สร้างกรณีทดสอบ

การสร้างกรณีทดสอบจะพิจารณาจาก RHS ของไวยากรณ์ AbS โดยเริ่มต้นจากเพิ่ม RHS ของ <start> โปรดักชั่นเข้าไปในกรณีทดสอบ จากตัวอย่างไวยากรณ์ AbS ของเครื่องรับฝากเงินอัตโนมัติพบว่า RHS ของ <start> โปรดักชั่นคือ <A> ดังนั้น <A> จะถูกเพิ่มเข้าไปในกรณีทดสอบ จากนั้นจะพิจารณาว่าในกรณีทดสอบมีสัญลักษณ์ไม่สิ้นสุดหรือไม่ หากมีจะแทนสัญลักษณ์ไม่สิ้นสุดนั้น โดยค้นหาโปรดักชั่นที่ LHS มีสัญลักษณ์ไม่สิ้นสุดตรงกันกับสัญลักษณ์ไม่สิ้นสุดที่พบในกรณีทดสอบ โดยจะนำ RHS ของโปรดักชั่นที่ค้นหาได้นั้นมาแทนที่สัญลักษณ์ไม่สิ้นสุดในกรณีทดสอบ ตัวอย่างเช่น กรณีทดสอบคือ a<B> ซึ่งสัญลักษณ์ไม่สิ้นสุดที่พบในกรณีทดสอบคือ <B> เราจะพิจารณาหาโปรดักชั่นที่ LHS ตรงกับ <B> สมมุติให้โปรดักชั่นที่ค้นหาได้คือ <B> ::= b<C> ดังนั้นจึงนำ RHS ของโปรดักชั่นนี้ นั่นคือ b<C> มาแทนเข้าไปที่สัญลักษณ์ไม่สิ้นสุด <B> ในกรณีทดสอบ ดังนั้นจะได้กรณีทดสอบเป็น a b <C> เราจะดำเนินกระบวนการค้นหาและแทนที่นี้ไปเรื่อยๆจนกว่าในกรณีทดสอบเป็นสัญลักษณ์สิ้นสุดทั้งหมด

ในการค้นหาโปรดักชั่น หากพบว่าโปรดักชั่นที่ได้จากการค้นหาปรากฏสัญลักษณ์เส้นตรงแนวตั้ง (|) ด้าน RHS จะแทนสัญลักษณ์ไม่สิ้นสุดในกรณีทดสอบด้วยและรูปแบบของ RHS โดยแต่ละรูปแบบจะต้องถูกนำมาแทนอย่างน้อย 1 ครั้ง และหากพบสัญลักษณ์วงเล็บก้ามปู ([ ]) ซึ่งเป็นการระบุว่าเป็นการวนซ้ำ จะแทนสัญลักษณ์ไม่สิ้นสุดในกรณีทดสอบโดยพิจารณาเป็น 2 กรณีดังนี้ กรณีที่ 1 หากยังเคยแทนรูปแบบนั้นลงในสัญลักษณ์ไม่สิ้นสุดในกรณีทดสอบ จะนำรูปแบบนั้นแทนลงไปในสัญลักษณ์ไม่สิ้นสุดในกรณีทดสอบ และกรณีที่ 2 หากพบว่าเคยแทนรูปแบบนั้นลงไปในสัญลักษณ์ไม่สิ้นสุดในกรณีทดสอบแล้ว จะนำรูปแบบลำดับถัดไปมาแทนให้กับสัญลักษณ์ไม่สิ้นสุดในกรณีทดสอบนั้น ตัวอย่างจากรูปที่ 7 ซึ่งทำให้ได้โปรดักชั่นต่างๆดังนี้

<A> ::= a <B>

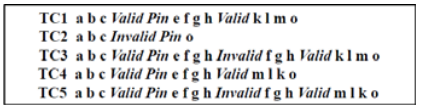
<B> ::= b <C>

<C> ::= *Invalid Pin* [A] | *Valid Pin* <D>

<D> ::= d

เมื่อกำหนดให้กรณีทดสอบคือ a b <C> และโปรดักชั่นที่ถูกเลือกใช้คือ <C> ::= *Invalid Pin* [A] | *Valid Pin* <D> ซึ่งมี LHS ตรงกับ <C> ดังนั้นจึงนำ *Invalid Pin* [A] และ *Valid Pin* <D> ไปแทน <C> จะได้กรณีทดสอบเป็น a b *Invalid Pin* [A] และ a b *Valid Pin* <D> เมื่อพิจารณาจะพบว่ากรณีทดสอบ a b *Invalid Pin* [A] มีสัญลักษณ์การวนซ้ำที่ [A] จึงทำการแทนต่อเป็น ab *Invalid Pin* a <B> และแทน <B> ต่อด้วยโปรดักชั่น <B> ::= b <C> จะได้กรณีทดสอบเป็น a b *Invalid Pin* a b <C> จากนั้นแทน <C> ต่อ ซึ่งจะพบว่าเคยแทนรูปแบบ *Invalid Pin* [A] มาแล้ว เราจึงนำรูปแบบลำดับถัดไปนั่นคือ *Valid Pin* <D> เข้ามาแทนสัญลักษณ์ไม่สิ้นสุด <C> ดังนั้นสุดท้ายจะได้กรณีทดสอบเป็น a b *Invalid Pin* a b *Valid Pin* <D> และ a b *Valid Pin* <D>

รูปที่ 9 แสดงกรณีทดสอบที่ได้สำหรับแผนภาพกิจกรรมของเครื่องรับฝากเงินอัตโนมัติที่อธิบายในหัวข้อ 2.2



รูปที่ 9 กรณีทดสอบของเครื่องรับฝากเงินอัตโนมัติ

4. การตรวจสอบประสิทธิภาพของกรณีทดสอบ

งานวิจัยนี้ใช้ Mutation Testing [1] เพื่อตรวจสอบประสิทธิภาพของกรณีทดสอบที่สร้างจากวิธีการที่นำเสนอ Mutation Testing เป็นการทดสอบโดยปรับเปลี่ยนโปรแกรมให้ต่างไปจากโปรแกรมต้นฉบับ โดยการปรับเปลี่ยนโปรแกรมแต่ละครั้งจะทำเพียง 1 ตำแหน่งเท่านั้น โปรแกรมที่ได้จากการปรับเปลี่ยนเรียกว่าโปรแกรม Mutant ซึ่งในงานวิจัยนี้จะปรับเปลี่ยนโปรแกรมโดยอาศัยตัวดำเนินการที่เรียกว่า Mutation Operator ซึ่งได้นำเสนอโดย Offutt และคณะในบทความเรื่อง “An Experimental Determination of Sufficient Mutant Operators” [6] ในงานวิจัยนี้เราใช้ 3 โปรแกรมเป็นกรณีศึกษาประกอบด้วย 1) ATM depositing 2) Vacation plan และ 3) ระบบงานรังสีวิทยา ซึ่งทั้ง 3 โปรแกรมใช้แผนภาพกิจกรรมในการออกแบบและพัฒนาด้วยภาษา Java และมีจำนวนกรณีทดสอบเป็น 5, 24 และ 37 กรณีทดสอบตามลำดับ

การตรวจสอบประสิทธิภาพของกรณีทดสอบโดยใช้ Mutation Testing มีขั้นตอนหลัก 2 ขั้นตอนดังนี้

ขั้นตอนที่ 1 สร้างโปรแกรม Mutant โดยปรับเปลี่ยนโปรแกรมต้นฉบับตาม Mutation Operator ที่เสนอในบทความ [6] ซึ่งประกอบด้วย operators 7 ประเภท คือ absolute value insertion (ABS), arithmetic operator replacement (AOR), constant replacement (CRP), logical connector replacement (LCR), RETURN statement replacement (RSR), statement deletion (SDL), และ unary operator insertion (UOI) จากโปรแกรมต้นฉบับพบว่ามี operator ที่สอดคล้องกับโปรแกรม ATM depositing 6 operators คือ ABS, AOR, CRP, LCR, SDL, และ UOI operator สำหรับโปรแกรม Vacation plan มี 7 operators ที่สอดคล้อง คือ ABS, AOR, CRP, LCR, RSR, SDL และ UOI สำหรับ operator ที่สอดคล้องกับโปรแกรมระบบงานรังสีวิทยามี 6 operators คือ ABS, AOR, CRP, LCR, SDL และ UOI โดยจำนวนโปรแกรม Mutant สำหรับโปรแกรม ATM depositing, Vacation plan และระบบงานรังสีวิทยาคือ 82, 183 และ 237 ตามลำดับ

ขั้นตอนที่ 2 นำกรณีทดสอบที่ได้จากวิธีการที่นำเสนอมาดำเนินการกับโปรแกรมต้นฉบับและโปรแกรม Mutant เพื่อเปรียบเทียบผลลัพธ์ หากผลลัพธ์ที่ได้จากโปรแกรม Mutant ต่างจากผลลัพธ์ที่ได้จากโปรแกรมต้นฉบับ แสดงว่ากรณีทดสอบที่นำมาใช้ในการดำเนินการนั้นสามารถตรวจจับข้อผิดพลาดในโปรแกรม Mutant ได้จะเรียกว่า Killed mutant ซึ่งประสิทธิภาพของกรณีทดสอบดูได้จากค่า Mutation Score ซึ่งเป็นอัตราส่วนของจำนวน Killed mutant ต่อจำนวนโปรแกรม Mutant หากค่า Mutation score เข้าใกล้ 1 หมายถึงกรณีทดสอบนั้นมีประสิทธิภาพ ตารางที่ 2 แสดงผลการประเมินประสิทธิภาพของกรณีทดสอบที่สร้างจากวิธีการที่นำเสนอโดยใช้วิธี Mutation Testing

จากผลการประเมินประสิทธิภาพของกรณีทดสอบที่สร้างจากวิธีการที่นำเสนอโดยใช้วิธีการ Mutation Testing พบว่า สามารถตรวจจับข้อผิดพลาดได้อย่างมีประสิทธิภาพ ซึ่งค่า Killed Mutant ของโปรแกรม ATM depositing โปรแกรม Vacation plan และระบบงานรังสีวิทยาได้ 72 Mutants, 168 Mutant และ 227 Mutant ตามลำดับ ซึ่งคิดเป็น Mutation Score ได้ 0.87, 0.92 และ 0.96 ตามลำดับ

5. สรุปผลและงานในอนาคต

บทความนี้เสนอการสร้างกรณีทดสอบจากแผนภาพกิจกรรม โดยมีขั้นตอนหลักๆ สามขั้นตอน คือ 1) สร้างตาราง AD 2) สร้างไวยากรณ์ AbS และ 3) สร้างกรณีทดสอบ และในงานวิจัยนี้ได้ใช้วิธี Mutation Testing เพื่อประเมินประสิทธิภาพของกรณีทดสอบที่ได้จากวิธีการที่นำเสนอ โดยนำไปประยุกต์ใช้กับสามกรณีศึกษา และพบว่ากรณีทดสอบที่สร้างจากวิธีการที่นำเสนอมีประสิทธิภาพในการตรวจจับข้อผิดพลาดที่เกิดขึ้นได้ดี

ในอนาคตได้วางแผนที่จะพัฒนาเครื่องมือเพื่อสร้างกรณีทดสอบโดยอัตโนมัติตามวิธีการที่นำเสนอ และจะนำไปปรับใช้ให้ครอบคลุมทุกๆลำดับของกิจกรรมที่ทำงานขนานกันที่พบในแผนภาพกิจกรรม

เอกสารอ้างอิง

[1] P. Amman and J. Offutt, *Introduction to Software testing*, Cambridge University Press, USA, 2008.

[2] G. Booch, J. Rumbaugh, and I. Jacobso, *The Unified Modeling Language User Guide*, Object Technology Series, Addison-Wesley Longman, Inc, 1998.

[3] R.W. Sebesta, *Concepts of Programming Language,* 9th edition, Pearson Addison Wesley, 2010.

[4] Sapra P.G. and Mohanty H., "Prioritization of Scenarios based on UML Activity Diagrams," *Proceeding of 3rt*, International Conference on Computational Intelligence, IEEE Communication Systems and Networks, pp. 271-276, 2009.

[5] C. Sun, "A Transformation-based Approach to Generating Scenario-oriented Test Cases from UML Activity Diagram for Concurrent Applications," *Proceeding of 32nd*, Annual IEEE International Computer Software and Application Conference, pp. 160-167, 2008.

[6] A.J. Offutt, A. Lee, G. Rothermel, R. Untch, and C. Zapf, "An Experimental Determination of Sufficient Mutant Operators," ACM Transaction on Software Engineering Methodology, pp. 99-118, 1996

ตาราง 2 ผลการประเมินประสิทธิภาพของกรณีทดสอบที่สร้างจากวิธีการที่นำเสนอโดยใช้วิธี Mutation Testing

Operator	ATM depositing		Vacation plan		ระบบงานรังสีวิทยา	
	Total Mutant	Killed Mutant	Total Mutant	Killed Mutant	Total Mutant	Killed Mutant
ABS	2	0	-	-	3	0
AOR	3	3	3	3	3	3
CRP	21	18	45	36	61	59
LCR	10	10	60	60	70	70
RSR	-	-	3	3	-	-
SDL	28	23	46	40	58	53
UOI	18	18	26	26	42	42
Total	82	72	183	168	237	227
Mutation Score	0.87		0.92		0.96	

ภาคผนวก ข.

ผลงานตีพิมพ์

เรื่อง	Generation Test case from UML Activity Diagram Based on AC grammar
งานประชุมวิชาการ	International Conference on Computer and Information Sciences (ICCIS)
สถานที่	Kuala Lumpur, Malaysia
วันที่	12 – 14 June 2012

Generation Test Case from UML Activity Diagram Based on AC Grammar

Kanjanee Pechtanun[1] and Supaporn Kansomkeat[2]
Department of Computer Science
Faculty of Science, Prince of Songkla University
Hat Yai, Songkhla, 90112, Thailand
Email:[1]nunimopa@hotmail.com ,[2]supaporn.k@psu.ac.th

***Abstract*- Software testing is an essential part of the software development life-cycle. The test case generation from design specifications is an important work in testing phase. The Unified Modeling Language (UML) is the most dominant standard language used in modeling the requirement and considered an important source of information for test case design. In this paper, we proposed a method to generate test cases from UML activity diagram. Firstly, an activity diagram is transformed into grammar, called Activity Convert (AC) grammar. Then, the AC grammar is used to generate test cases. The proposed method was applied on four case studies. The result shows that our tests achieve all paths coverage and have ability in fault detection.**

Keywords - ***software testing, UML based testing, activity diagram***

I. INTRODUCTION

Testing is the most important part of quality assurance in software development life-cycle. As the complexity and size of software products grow, the time and effort required to do effective testing increase. Studies indicate that more than 50% of software development cost is devoted to testing [1]. If the tests are in the process before implementation, costs of software development will be reduced.

Unified Modeling Language (UML) [2] is the most dominant standard language used in modeling the requirement and considered an important source of information for test case design. Several researchers use different UML models to generate test cases [3, 4, 5, 6, 7, 8]. An activity diagram is one of the UML diagrams that is used to model software behavior. It describes the sequential or concurrent control flow of activities.

This paper uses UML activity diagram as design specifications. Our approach first transforms an activity diagram into grammar, called Activity Convert (AC) grammar, and then the AC grammar is used to generate test cases. Test cases derived from the design stage can help to detect errors in early software development process and reduce the cost of software development.

The rest of the paper is organized as follows. Section II introduces the background. Section III presents our approach test cases generation base on AC grammar. Section IV provides the validation of test cases. Finally, Section V presents conclusions and future work

978-1-4673-1938-6/12/$31.00 ©2012 IEEE

II. BACKGROUND

In this section, we present basic concepts that are used in this paper. First, we describe UML activity diagrams. Next, we describe the control flow graph coverage used for analysis the presented method.

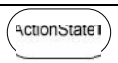
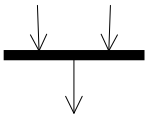
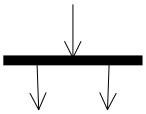
A. UML Activity Diagram

UML activity diagrams [2] are used typically for workflow representation. The diagrams describe behavior by modeling the sequence of activities performed. Activity diagrams are similar to procedural flow charts. But the difference between them is that activity diagrams support parallel activities. Activity diagrams commonly contain such basic symbols as activity state, transition, branch, merge, fork, join and swim lane. Each activity diagram has one start activity and one final activity. The start activity is indicated by a solid circle and the final activity by a bull's eye. Activity state is indicated by round corner boxes. Transitions are drawn as directed arrows to show control flow among activities. Condition behavior is described is a branch and a merge that are shown as diamonds. A branch has one input transition and multiple output transitions. In branches, each output transition may be labeled with a boolean expression to be satisfied to choose the branch. A merge has multiple input transitions but only one output transition. Fork is represented by one input transition to a synchronization bar and multiple output transitions. Join is represented by multiple transitions to a synchronization bar and only one output transition. Swim lanes represent the responsibilities of a particular class. Table1 shows these activity diagram symbols.

Figure 1 shows the activity diagram of ATM withdrawing. It describes the functionality of withdrawing money from the ATM. The ATM withdrawing starts with insert card. Next, the user enters pin of card and the system authorizes the pin, in case of invalid pin will go to eject card. If the pin is right, the user will enter amount of money that wants to withdraw. After that, the system checks A/C balance of account, in case of insufficient balance will go to show balance and eject card. But, if the balance is more than money in the account, the system will take money from slot

along with debit account and take receipt. Finally, the system shows balance and ejects card.

TABLE 1. BASIC ACTIVITY DIAGRAM SYMBOLS

Symbols	Names
	Start activity
	Final activity
	Activity
	Branch, Merge
	Transition
	Swim lane
	Join
	Fork

B. Control Flow Graph Coverage

The most widely used graph coverage criteria are defined on source code. The most common graph is called a Control Flow Graph (CFG). The CFG associates an edge with each possible branch in the program, and a node with sequences of statements [9].

The basic graph coverage criteria defined on source code are given below:

- Statement coverage: All nodes in the CFG must be coverage.
- Branch coverage: All edges in the CFG must be coverage.
- Path coverage: All paths in the CFG must be coverage.

For example, Figure 2 shows a CFG and Table 2 shows test paths of Statement coverage, Branch coverage and Path coverage.

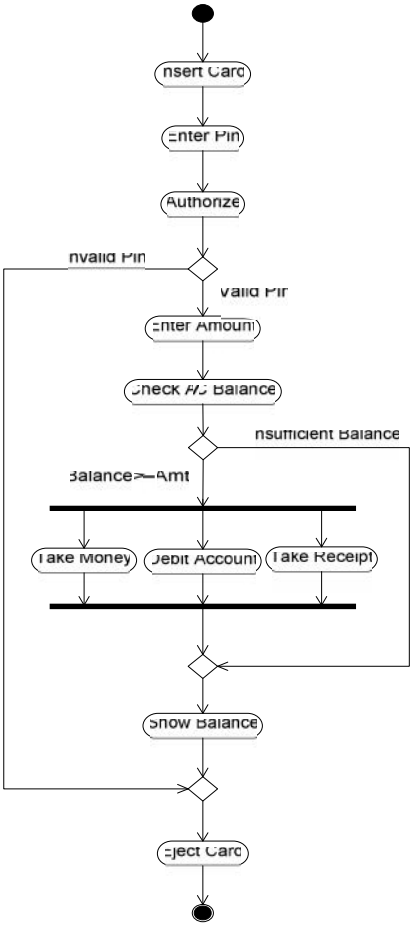


Figure 1. An example of ATM withdrawing

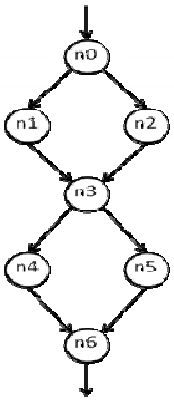


Figure 2. An example of CFG

TABLE 2. COVERAGE CRITERIA OF CFG

Coverage criteria	Test paths
Statement coverage	n0,n1,n3,n5,n6 n0,n2,n3,n4,n6
Branch coverage	n0,n1,n3,n4,n6 n0,n2,n3,n5,n6
Path coverage	n0,n1,n3,n5,n6 n0,n2,n3,n4,n6 n0,n1,n3,n4,n6 n0,n2,n3,n5,n6

III. GENERATING TEST CASES FROM UML ACTIVITY DIAGRAMS BASED ON AC GRAMMAR

The generating test cases from UML activity diagrams based on AC grammar mainly includes three steps as follows: 1) constructing the Activity Dependence (AD) table, 2) generating the AC grammar and 3) generating test cases. In the AD table construction step, the symbols are assigned to activities and the dependence of those activities is indicated. In generation of AC grammar step, an activity diagram is analyzed to model the grammar called Activity Convert (AC) grammar. In test case generation step, the AC grammar is used to generate test cases. The generated test cases are validated against the coverage analysis [1].

The proposed architecture is shown in Figure 3.

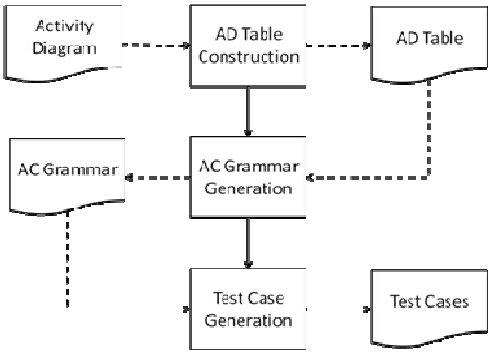


Figure 3. The proposed architecture

A. AD table Construction

In this step, we construct the Activity Dependency (AD) table for an activity diagram. The AD table shows the symbols given for each activity and shows the dependency between activities. The AD table has three columns as follows

- Symbol Name: The alphabetic letter for each activity.
- Activity Name: The name of the activity.
- Dependency: The symbol names of activities that the current activity depends on.

Table 3 shows the AD table of an ATM withdrawing activity diagram explained in section II

TABLE3. THE AD TABLE OF AN ATM WITHDRAWING ACTIVITY DIAGRAM

Symbol Name	Activity Name	Dependency
Start	Start	-
A	Insert Card	Start
B	Enter Pin	A
C	Authorize	B
D	Enter Amount	C
E	Check A/C Balance	D
F	Take Money	E
G	Debit Account	E
H	Take Receipt	E
I	Show Balance	F,G,H
J	Eject Card	I
Stop	Stop	J

B. AC grammar Generation

In this step, we generate a AC grammar from the considered activity diagram. The AC grammar consists of productions. Each production has two parts, the Left Hand Side (LHS) and the Right Hand Side (RHS) of the arrow. Productions are created by considering all components of the used activity diagram. The symbol names in AD table are used in this step. For creating a production of each component, LHS is replaced by the symbol name (dependency column of AD table) of activity that the current activity depends on. The RHS is replaced by considering each type of components:

- **Activity:** An activity is replaced by a symbol name (symbol name column of AD table) of activity state insides angle brackets (< >).
- **Branch:** Each activity in the end of output transitions is replaced by a star (*) and a symbol name of the activity insides angle brackets. All replaced items of activities are separated by a vertical bar (|). Every boolean expression on transitions of this branch is used to construct the Decision Dependency (DD) table. For example, the branch shown in Figure 4 is replaced by *<A>|*<B>. The DD table of this branch is Table 4.
- **Fork and Join:** Each activity between the fork and join is replaced by a symbol name of activity insides angle brackets. All replaced items of activities are listed in forward and backward orders. These two ordered lists are separated by a vertical bar. For example, the fork and join shown in Figure 5 are replaced by <A><B><C>|<C><B><A>.

The string inside angle brackets in AC grammar is called ***AC state***. Table 5 shows the RHS of AC grammars for the basic symbols of activity diagram.

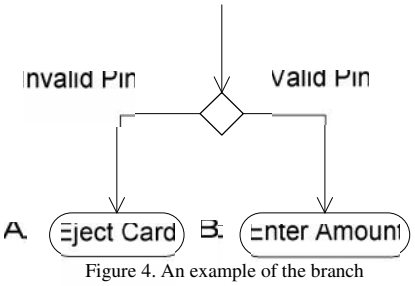


Figure 4. An example of the branch

TABLE 4. DECISION DEPENDENCE TABLE

Symbol Name	Decision Dependency
A	Invalid Pin
B	Valid Pin

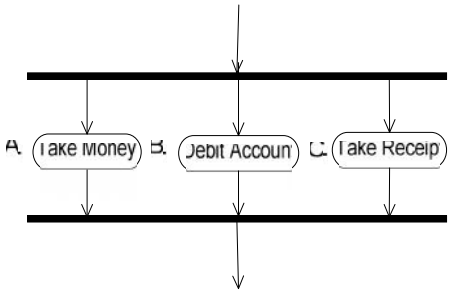
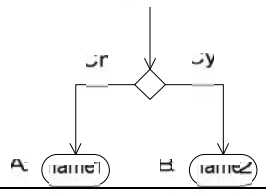
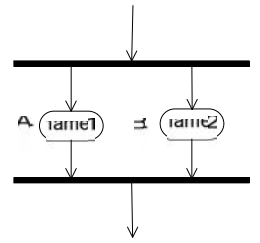


Figure 5. An example of the fork and join

TABLE 5. RHS OF AC GRAMMAR FOR BASIC SYMBOLS

Symbol	RHS of AC grammar
	<A>
	*<A>|*<B>
	<A><B> | <B><A>

For the ATM withdrawing activity diagram explained in section II, the AC grammar and DD table are shown in Figure 6 and Table 6, respectively.

<start> → <A>
<A> → <B>
<B> → <C>
<C> → *<D> | *<J>
<D> → <E>
<E> → *<F><G><H> |
*<H><G><F> |
*<I>
<F,G,H> → <I>
<H,G,F> → <I>
<I> → <J>
<J> → <stop>

Figure 6. The AC grammar for ATM withdrawing

TABLE 6. THE DD TABLE FOR ATM WITHDRAWING

Symbol Name	Decision Dependency
D	Valid Pin
F	Balance >= Amt
H	Balance >= Amt
I	Insufficient Balance
J	Invalid Pin

C. *Test Case Generation*

In this step, a test case is represented as the sequence of RHS of the AC grammar. First, the RHS of the <start> production is added into a test case. Using the ATM withdrawing example, <A> is added into the test case. Then, we consider the production that the LHS matches with AC states in the test case. The RHS of this production is added into the test case. For example, considering the AC state <A>, the production <A>→<B> is used to add <B> into the test case, that is <A><B> now. We will follow this process repeatedly until it reaches a production that has <stop> in RHS.

In case that vertical bar (|) symbols appear in RHS of the used production, the considered test case is duplicated to N test cases. By N is a number of items separated by vertical bars in RHS. For example, the test case is <X> and the matched production is <X>→<Y>|<Z>. Therefore, the two test cases generated are <X><Y> and <X><Z>.

In case that star (*) symbol appears in front of an AC state in RHS of the used production, the star is replaced by the decision dependency from the DD table that the symbol name associates with the AC state. For example, the test case is <R> and the matched production is <R>→*<S>. Therefore, the generated test case is <R>*No Pin*<S>.

In case that both vertical bars and star symbols appear in RHS of the used production, test cases are generated by considering star symbols first and then vertical bars. Using the ATM withdrawing example, the test case is <A><B><C> and the matched production is <C>→*<D> | *<J>. Test cases are generated as follows. In the first step, the star symbols are replaced with the decision dependency from Table 6. The star symbols in front of <D> and <J> are replaced by *Valid Pin* and *Invalid Pin*, respectively. In the second step, the test case is duplicated to be two test cases, <A><B><C>*Valid Pin*<D> and <A><B><C>*Invalid Pin*<J>.

Figure 7 shows the generated Test Case of the ATM withdrawing activity diagram explained in section II

TC1 <A><B><C>*InvalidPin*<J>
TC2 <A><B><C>*ValidPin*<D><E>*InsufficientBalance*<I><J>
TC3 <A><B><C>*ValidPin*<D><E>*Balance*>=*Amt*<F><G><H><I><J>
TC4 <A><B><C>*ValidPin*<D><E>*Balance*>=*Amt*<H><G><F><I><J>

Figure 7. The Test Case generation

IV. VALIDATION OF TEST CASE

In this section, four small applications, Flight check-in, Manage order, Depositing money with the ATM and Withdrawing money from the ATM, are used to evaluate the proposed method by using coverage analysis technique. The coverage analysis processes includes three steps: CFG construction, code instrumentation, and coverage analysis.

In CFG construction step, we construct the CFG from the source code and indicate all paths in CFG according to path coverage criteria. In code instrumentation step, the source code is instrumented along paths indicated in CFG construction step to record program execution trace. In coverage analysis step, the instrumented program is executed with the test cases generated by the proposed method. After executing, the program execution trace is analyzed to compute the required coverage. By our experiment, the coverage analysis result shows that the ability of tests generated by the proposed method can execute all statement and achieve all-paths coverage for all applications.

Moreover, to evaluate fault-detection capability of our tests, the tests generated by the path coverage criteria are compared with our tests. We found that the tests generated by path coverage criteria can't detect errors while our tests can detect four errors. For the result, we analyze that all errors detected are the design errors.

V. CONCLUSION

This paper proposed a method for generating test cases from UML activity diagram. The proposed method mainly includes three steps: Activity Dependence (AD) table construction, AC grammar generation and test cases generation. The method was applied on four case studies. The result shows that our tests can exercise all paths and have ability to the error detection faults. In the future, we will improve the test case generation for more complexity activity diagrams. Such as between the fork and join is not only the single activity but also is the group of activity, and the activity diagrams consist of loop. We intend to develop the tools for automatically generating test cases based on our proposed method. More works need to be done in order to combine other UML diagrams in our method.

REFERENCES

[1] R.V. Binder. "Testing Object-Oriented Systems: Models, Patterns and Tools," Object Technology. Addision-Wesley, 2000

[2] G. Booch, J. Rumbaugh, I. Jacobso. "The Unified Modeling Language User Guide," Object Thechnology Series, Addison-Wesley Longman, Inc, 1998.

[3] M. Sarma, D. Kundu, R. Mall. "Automatic Test Case Generation from UML Sequence Diagrams," Proceedings of the 15th International Conference on Advanced Computing and Communications, IEEE Computer Society Washington, DC, USA, 2007.

[4] S.K. Swain, D.P. Mohapatra. "Test Case Generation from Behavioral UML Models," International Journal of Computer Applications (IJCA) 6 (2010).

[5] S. Kansomkeat, P. Thiket. "Generating Test Cases from UML Activity Diagrams using the Condition-Classification Tree Method," Proceedings of 2nd, International conference on Software Technology and Engineering (ICSTE), 2010.

[6] F. Xin, S. Jian, L. LinLan, L. QiJun. "Test Case Generation from UML Subactivity and Activity Diagram," International Symposium on Electronic Commerce and Security, IEEE, 2009

[7] S. Kansomkeat, W. Rivepiboon. "Automated-Generating Test Case Using UML State chart Diagrams," Proceedings of the 2003 annual research conference of the South African Institute of Computer Scientists and Information Technologists on Enablement through Technology (SAICSIT), Republic of South Africa, 2003

[8] M. Chen, P. Mishra, D. Kalita. "Coverage-driven Automatic Test Generation for UML Activity Diagrams," Proceedings of the 18th ACM Great Lakes symposium on VLSI, Orlando, Florida, USA, 2008

[9] P. Ammann and J.Offutt "Introduction to Software Testing",Cambridge University Press, USA, 2008

ประวัติผู้เขียน

ชื่อ สกุล นางสาวกาญจนี เพชรทะนันท์

รหัสประจำตัวนักศึกษา 5410220109

วุฒิการศึกษา

วุฒิ	ชื่อสถาบัน	ปีที่สำเร็จการศึกษา
วท.บ. (วิทยาการคอมพิวเตอร์)	มหาวิทยาลัยสงขลานครินทร์	2553

ทุนการศึกษา

ทุนผู้ช่วยวิจัยคณะวิทยาศาสตร์ คณะวิทยาศาสตร์ มหาวิทยาลัยสงขลานครินทร์ ประจำปี 2554

การตีพิมพ์เผยแพร่ผลงาน

กาญจนี เพชรทะนันท์ และ สุภาภรณ์ กานต์สมเกียรติ. 2555. การสร้างกรณีทดสอบบนพื้นฐานของไวยากรณ์ AbS โดยใช้แผนภาพกิจกรรม. The Ninth International Joint Conference on Computer Science and Software Engineering (JCSSE'12). กรุงเทพมหานคร ประเทศไทย, 30 พฤษภาคม – 1 มิถุนายน 2555. หน้า 42-47.

K. Pechtanun and S. Kansomkeat. 2012. Generation Test case from UML Activity Diagram Based on AC grammar. International Conference on Computer and Information Sciences (ICCIS 2012). Kuala Lumpur, Malaysia. June 12-14, 2012. pp 895-899.