Formal Verification

การทวนสอบเชิงรูปนัย

วิวัฒน์ วัฒนาวุฒิ
ภาควิชาวิศวกรรมคอมพิวเตอร์
คณะวิศวกรรมศาสตร์ จุฬาลงกรณ์มหาวิทยาลัย

Formal Verification

การทวนสอบเชิงรูปนัย

วิวัฒน์ วัฒนาวุฒิ
ภาควิชาวิศวกรรมคอมพิวเตอร์
คณะวิศวกรรมศาสตร์ จุฬาลงกรณ์มหาวิทยาลัย

Formal Verification การทวนสอบเชิงรูปนัย / วิวัฒน์ วัฒนาวุฒิ

1. การทวนสอบ, วิธีเชิงรูปนัย (คอมพิวเตอร์)
2. การสร้างแบบจำลอง (คอมพิวเตอร์)

005.133

ISBN 978-616-474-543-8

พิมพ์ครั้งที่ 1 (มกราคม 2561) จำนวน 100 เล่ม

สงวนลิขสิทธิ์ตาม พ.ร.บ. ลิขสิทธิ์ พ.ศ. 2537/2540
โดย วิวัฒน์ วัฒนาวุฒิ
การผลิตและการลอกเลียนหนังสือเล่มนี้ไม่ว่ารูปแบบใดทั้งสิ้นต้องได้รับ
อนุญาตเป็นลายลักษณ์อักษรจากเจ้าของลิขสิทธิ์

จัดทำโดย
วิวัฒน์ วัฒนาวุฒิ
ภาควิชาวิศวกรรมศาสตร์
คณะวิศวกรรมศาสตร์
จุฬาลงกรณ์มหาวิทยาลัย
พญาไท กรุงเทพฯ 10330
http://www.cp.eng.chula.ac.th

❤ *To My Parents,*

Piyaprapa,

Vippy and Winnie ❤

คำนำ

ตำราเล่มนี้จัดทำขึ้นเพื่อใช้สอนในรายวิชา 2110502 การทวนสอบเชิงรูปนัย (Formal Verification) เป็นวิชาที่น่าสนใจในหลักสูตรวิศวกรรมศาสตร์บัณฑิต สาขาวิชาวิศวกรรมคอมพิวเตอร์ และหลักสูตรวิทยาศาสตร์มหาบัณฑิต สาขาวิชาวิศวกรรมซอฟต์แวร์

เนื้อหาในตำรานี้ครอบคลุมสามส่วนหลัก คือ ส่วนการแนะนำและทบทวนองค์ความรู้พื้นฐานด้านคณิตตรรกศาสตร์ ส่วนการนิยามและยกตัวอย่างองค์ความรู้ด้านการทวนสอบเชิงรูปนัย และส่วนการสรุปองค์ความรู้จากงานวิจัยของผู้เขียนและคณะ ที่เกี่ยวกับการทวนสอบเชิงรูปนัยที่เป็นประโยชน์แทรกอยู่ในเล่ม

องค์ความรู้ด้านการทวนสอบเชิงรูปนัยที่นำเสนอได้เขียนในลักษณะที่อ่านเข้าใจได้ง่าย ด้วยผู้เขียนพยายามหลีกเลี่ยงไม่เขียนเนื้อหาในลักษณะตำราคณิตตรรกศาสตร์ที่ดำเนินเรื่องเป็นนิยาม การพิสูจน์และทฤษฎีบทที่ใช้สัญลักษณ์ทางคณิตตรรกศาสตร์มากจนเกินไป อย่างไรก็ตามผู้เขียนยังคงต้องมีส่วนนิยามและยกตัวอย่างไว้พอสังเขป และชี้ให้เห็นข้อประโยชน์ในการทวนสอบและการลงมือทำ กล่าวได้ว่าในการทวนสอบเชิงรูปนัยจำเป็นต้องมีสองสิ่งที่สำคัญก่อนเสมอคือ แบบจำลองเชิงรูปนัยของระบบและคุณลักษณะที่ต้องการทวนสอบ ทั้งสองสิ่งนี้จะต้องมีการเตรียมพร้อมไว้ก่อนการทวนสอบ เมื่อเราเริ่มดำเนินการทวนสอบเชิงรูปนัยจำเป็นต้องใช้เครื่องมือสนับสนุน ไม่แนะนำให้ทำด้วยมือตั้งแต่ต้นจนจบ เนื่องจากลักษณะพฤติกรรมของระบบ ขนาด และความซับซ้อนของระบบจะทำให้ผู้ทวนสอบมีอุปสรรคอย่างมากถ้าไม่มีเครื่องมือสนับสนุน ซึ่งในตำราเล่มนี้ก็ได้ยกตัวอย่างการใช้เครื่องมือทวนสอบสองชนิด คือ เครื่องมือเซดอีฟ สำหรับการทวนสอบด้วยวิธีพิสูจน์ทฤษฎีบทแบบนิรนัย และเครื่องมือสปิน สำหรับการทวนสอบด้วยโมเดลเช็กกิง อย่างไรก็ตามผู้อ่านจำเป็นต้องอ้างอิงคู่มือการใช้งานเครื่องมือทั้งสองเพิ่มเติมสำหรับการใช้งานจริงด้วย

สำหรับเนื้อหาสรุปงานวิจัยของผู้เขียนที่เพิ่มเติมแทรกในตำราเพื่อให้ผู้อ่านได้เห็นตัวอย่างหรือกรณีศึกษาจริงทั้งเชิงทฤษฎีและเชิงประยุกต์ไปใช้กับการทวนสอบระบบงานจริง เพื่อให้การทวนสอบเป็นที่ยอมรับและแพร่หลายอย่างจริงจัง จำเป็นต้องมีผู้เล็งเห็นประโยชน์และ

ความเข้าใจภาพรวมของกระบวนการด้วยเช่นกัน และในปัจจุบันที่ระบบงานด้านคอมพิวเตอร์ถือว่าเป็นโครงสร้างพื้นฐานที่ขับเคลื่อนธุรกรรมประจำวันรอบตัวเราแทบทุกเรื่อง ตลอดจนการมีชิ้นส่วนอุปกรณ์ที่ควบคุมด้วยซอฟต์แวร์แบบฝังตัว ต่างก็ทำงานแยกแบบกระจายและทำงานพร้อมๆ กัน แต่ต้องมีการประสานกันอย่างลงตัว ทำให้เกิดความซับซ้อนมากเกินกว่าจะใช้การทดสอบระบบแบบเดิม การทวนสอบจึงเป็นทางเลือกที่น่าสนใจ และน่าจะเป็นการลงทุนที่คุ้มค่าด้านการควบคุมคุณภาพ เพื่อจะได้ยืนยันว่าระบบปลอดภัยจากการติดตาย หรือระบบทำงานแบบเข้าถึงได้ครบทุกสถานะและทำงานได้ครบตามที่ต้องการ

การสร้างแบบจำลองเชิงรูปนัยที่กล่าวถึงในตำรานี้แบ่งเป็นการสร้างแบบจำลองด้วยการเขียนข้อกำหนดเชิงรูปนัยด้วยภาษาเชิงรูปนัย เช่น ภาษาเซด ภาษาบี ภาษาคาเฟโอบีเจ เป็นต้น กว่าจะได้แบบจำลองเชิงรูปนัยผู้เขียนจำเป็นต้องมีพื้นฐานในการวิเคราะห์ปัญหา การกำหนดโดเมนปัญหา การระบุการดำเนินการ การออกแบบเงื่อนไขก่อนและเงื่อนไขหลังของแต่ละการดำเนินการ ตลอดจนการวิเคราะห์หาเงื่อนไขค่ายืนยงข้อมูล เป็นต้น

ตำราเล่มนี้จัดทำเพื่อให้อ่านได้ง่ายโดยไม่ต้องมีพื้นฐานทางคณิตตรรกศาสตร์มากนัก จึงจำเป็นต้องมีการบรรยายเชิงพรรณนาในบางช่วงบางตอน และการยกตัวอย่างประกอบเป็นระยะ โดยหวังว่าผู้อ่านสามารถติดตามและเข้าใจเนื้อหาหลักได้ตามวัตถุประสงค์ที่ตั้งไว้ ขอขอบคุณล่วงหน้าในการให้คำแนะนำและชี้ข้อผิดพลาดที่อาจจะมีเพื่อจะได้แก้ไขปรับปรุงให้ตำราเล่มนี้มีความสมบูรณ์ยิ่งขึ้นต่อไป

วิวัฒน์ วัฒนาวุฒิ

27 กรกฎาคม 2560

สารบัญ

1. แนะนำการทวนสอบเชิงรูปนัย

1.1	ความสำคัญของบทนี้	1
1.2	วัตถุประสงค์	1
1.3	การทวนสอบในกระบวนการพัฒนาระบบ	1
1.4	ข้อกำหนดเชิงรูปนัย	5
1.5	แบบจำลองเชิงรูปนัย	7
1.6	การทวนสอบเชิงรูปนัย	15
1.7	ประเภทระบบซอฟต์แวร์และฮาร์ดแวร์ที่นำมาทวนสอบ	23
1.8	เครื่องมือทวนสอบเชิงรูปนัย	24
1.9	แบบฝึกหัด	25

2. ความรู้พื้นฐานด้านคณิตตรรกศาสตร์

2.1	ความสำคัญของบทนี้	27
2.2	วัตถุประสงค์	27
2.3	ความเป็นมาของคณิตตรรกศาสตร์	27
2.4	การพิสูจน์ทางตรรกศาสตร์	31
2.5	ตรรกศาสตร์เชิงประพจน์	33
2.6	ความพอใจและความมีเหตุผล	42
2.7	ตรรกศาสตร์ภาคแสดง	42
2.8	ทฤษฎีเซต	44
2.9	ความสัมพันธ์	49
2.10	พีชคณิตบูลีน	50
2.11	แบบฝึกหัด	51

3. ตรรกศาสตร์เชิงเวลา

3.1	ความสำคัญของบทนี้	53
3.2	วัตถุประสงค์	53
3.3	ตรรกศาสตร์เชิงเวลา	53
3.4	การแปลงเอ็มทีแอลไปเป็นแอลทีแอล	66
3.5	แบบฝึกหัด	71

4. สร้างแบบจำลองเชิงรูปนัยด้วยภาษาเชิงรูปนัย

4.1 ความสำคัญของบทนี้ 73
4.2 วัตถุประสงค์ 73
4.3 กลุ่มภาษาเชิงรูปนัย 73
4.4 ภาษาเซด 75
4.5 ภาษาวีดีเอ็ม-เอสแอล 75
4.6 วิธีเชิงรูปนัยบี 76
4.7 ภาษาคาเฟโอบีเจ 76
4.8 การเขียนข้อกำหนดเชิงรูปนัยเบื้องต้น 77
4.9 ตัวอย่างระบบ Print Spooler 80
4.10 เขียนข้อกำหนดรูปนัยด้วยภาษาเซด 85
4.11 เอเอ็มเอ็น 108
4.12 ภาษาคาเฟโอบีเจ 116
4.13 ภาษาโพรเมลา 137
4.13 แบบฝึกหัด 141

5. สร้างแบบจำลองเชิงรูปนัยด้วยแผนภาพ

5.1 ความสำคัญของบทนี้ 143
5.2 วัตถุประสงค์ 143
5.3 ออโตมาตา 143
5.4 โครงสร้างคริปกี 149
5.5 บูชิออโตมาตา 152
5.6 เพทริเน็ต 156
5.7 แบบฝึกหัด 161

6. การทวนสอบด้วยการพิสูจน์ทฤษฎีบท

6.1 ความสำคัญของบทนี้ 163
6.2 วัตถุประสงค์ 163
6.3 สร้างแบบจำลองระบบด้วยภาษาเซด 163
6.4 การวางแผนทวนสอบ 176
6.5 แบบฝึกหัด 178

7. การทวนสอบด้วยโมเดลเช็กกิง

7.1 ความสำคัญของบทนี้ 179
7.2 วัตถุประสงค์ 180
7.3 โมเดลเช็กกิง 180

7.4 โมเดลเช็กกิงตามทฤษฎีออโตมาตา 191
7.5 งานวิจัยสนับสนุนการทวนสอบด้วยโมเดลเช็กกิง 206
7.6 แบบฝึกหัด 231
บรรณานุกรม 233
ดัชนี 237

1

แนะนำการทวนสอบเชิงรูปนัย

1.1 ความสำคัญของบทนี้

เนื้อหาบทแรกนี้ แนะนำความหมายและความสำคัญของการทวนสอบ ซึ่งเป็นกิจกรรมที่เกี่ยวข้องกับการควบคุมคุณภาพในกระบวนการพัฒนาระบบ การทวนสอบสามารถเลือกทำได้ทั้งแบบอรูปนัยและแบบรูปนัย โดยวิธีทวนสอบที่เราเห็นกันประจำทั่วไป เรียกว่าการทวนสอบเชิงอรูปนัย ซึ่งเน้นวิธีการตรวจหาข้อผิดพลาดของระบบผ่านการตรวจสอบจากคณะผู้เชี่ยวชาญ ส่วนการทวนสอบเชิงรูปนัยซึ่งมีข้อดีมากกว่า สำหรับการตรวจหาข้อผิดพลาดที่อาจจะเกิดขึ้นได้ในระบบวิกฤติขนาดใหญ่และซับซ้อน

วิธีการทวนสอบเชิงรูปนัยนี้ทำได้ทั้งการพิสูจน์ทางคณิตตรรกศาสตร์แบบนิรนัยและแบบการค้นปริภูมิสถานะหาข้อผิดพลาดโดยอัตโนมัติ เนื้อหาบทนี้ครอบคลุมพื้นฐานและองค์ประกอบสำคัญในของการทวนสอบเชิงรูปนัย เช่น อธิบายความหมายของวิธีเชิงรูปนัย ข้อกำหนดเชิงรูปนัย แบบจำลองเชิงรูปนัย และหลักวิธีพื้นฐานในการทวนสอบเชิงรูปนัย เป็นต้น และตอบคำถามผู้ทวนสอบที่เลือกใช้วิธีทวนสอบเชิงรูปนัยว่า ควรจะมีพื้นฐานความรู้ด้านใด และต้องเตรียมการอย่างไร อีกทั้งปัจจุบันมีเครื่องมือสนับสนุนการทวนสอบเชิงรูปนัยอะไรบ้าง

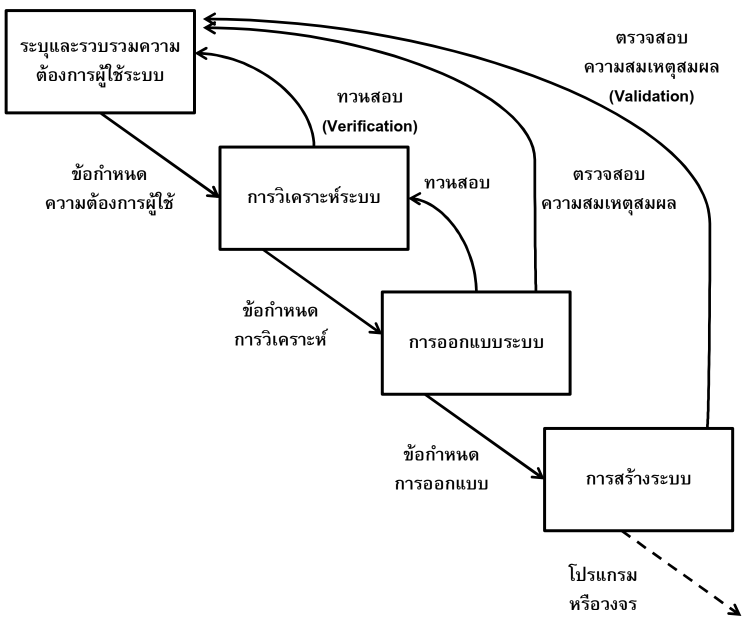
1.2 วัตถุประสงค์

- เพื่อให้ทบทวนภาพรวมของกระบวนการพัฒนาระบบ
- เพื่อให้เข้าใจกิจกรรมการทวนสอบและทางเลือกในการทวนสอบ
- เพื่อให้เข้าใจแนวคิดการใช้วิธีเชิงรูปนัยในการทวนสอบระบบ
- เพื่อให้เข้าใจองค์ประกอบพื้นฐานในการทวนสอบเชิงรูปนัย
- เพื่อแนะนำตัวอย่างเครื่องมือสนับสนุนการทวนสอบเชิงรูปนัย

1.3 การทวนสอบในกระบวนการพัฒนาระบบ

โดยทั่วไปการพัฒนาระบบซอฟต์แวร์หรือฮาร์ดแวร์มักมีกรอบการพัฒนา (development framework) กำหนดไว้แล้วอย่างเหมาะสม เริ่มจากการระบุความ

ต้องการของผู้ใช้ระบบให้ถูกต้องและครบถ้วนเป็นสำคัญ ทั้งนี้เพื่อนำมาเขียนเป็นข้อกำหนดความต้องการ (requirements specifications) จากนั้นให้ทำการวิเคราะห์เพื่อหาข้อกำหนดการวิเคราะห์ (analysis specifications) และนำผลลัพธ์ที่ได้นี้ไปทำการออกแบบต่อ ซึ่งจะได้เป็นข้อกำหนดการออกแบบ (design specifications) เพื่อนำมาใช้ลงมือพัฒนาต่อให้ได้ระบบซอฟต์แวร์ที่เสร็จสมบูรณ์พร้อมทำการติดตั้งและทำงานได้จริง หรือให้ได้ระบบฮาร์ดแวร์ที่มีชิ้นส่วนวงจรทำงานได้ตามข้อกำหนดทั้งหลายที่กล่าวมา เราแสดงตัวอย่างลำดับของกิจกรรมที่มีในกระบวนการพัฒนาระบบในรูปที่ 1.1



รูปที่ 1.1 ตัวอย่างกิจกรรมในกระบวนการพัฒนาระบบ [1]

ข้อกำหนดทั้งหลายที่เกิดขึ้นในแต่ละกิจกรรมของกระบวนการพัฒนาระบบมีความสำคัญมากทีเดียว เพราะต้องมีความถูกต้องและครบถ้วน ครอบคลุมทั้งโครงสร้างและพฤติกรรมของระบบ นอกจากนี้ ยังต้องระบุปัจจัยด้านคุณภาพ (quality factors) ที่ผู้ใช้ต้องการอย่างชัดเจนด้วยเสมอ เพื่อเราจะได้นำข้อกำหนดเหล่าพร้อมปัจจัยด้านคุณภาพนี้เป็นตัวขับเคลื่อนหรือใช้เป็นคู่มืออ้างอิงที่สำคัญระหว่างการพัฒนาระบบ อย่างไรก็ตาม เรามักจะถูกถามคำถามที่เรามักจะตอบไม่ได้อย่างมั่นใจว่า *“ระบบที่ออกแบบและพัฒนาขึ้นมานั้นจะทำงานได้ถูกต้องทุกกรณีหรือไม่”* ข้อแนะนำคือ เราควรเปลี่ยนรูปแบบคำถามเสียใหม่ โดยใช้คำถามนี้แทนว่า *“ระบบจะทำงานไม่ผิดพลาดจากข้อกำหนดทุกข้อที่ให้ไว้หรือไม่”* และ

เราจะตอบคำถามนี้ได้ทันที โดยนำข้อกำหนดทั้งหลายที่มีอยู่มาเป็นเกณฑ์ในการทวนสอบนั่นเอง

การทวนสอบถือเป็นกิจกรรมเสริมที่ถูกกำหนดไว้ในกระบวนการพัฒนาระบบเช่นกัน โดยเป็นกิจกรรมเพื่อการควบคุมคุณภาพ รูปที่ 1.1 แสดงให้เห็นว่าแต่ละกิจกรรมในกระบวนการพัฒนาระบบมักจะได้ผลลัพธ์อย่างใดอย่างหนึ่งเสมอ และเราจะทวนสอบผลลัพธ์ที่ได้จากแต่ละกิจกรรมเทียบกับข้อกำหนดที่นำเข้าของกิจกรรมนั้นด้วยเสมอ ตัวอย่างเช่น เราจะทวนสอบข้อกำหนดการออกแบบเทียบกับข้อกำหนดการวิเคราะห์ โดยตรวจหาให้ได้ว่าข้อกำหนดการออกแบบนั้นไม่มีข้อผิดพลาดเมื่อเทียบกับข้อกำหนดการวิเคราะห์ทุกข้อที่ให้ไว้ ทั้งนี้ ไม่ต้องเทียบย้อนไปถึงข้อกำหนดความต้องการผู้ใช้ด้วย

การทวนสอบที่ปรากฏในกระบวนการพัฒนาระบบ คือ กิจกรรมที่ระบุไว้ในคู่มือวิธีปฏิบัติเพื่อควบคุมคุณภาพ (quality control practices) ใช้เป็นวิธีที่ยืนยันว่า สิ่งที่เป็นผลลัพธ์ออกมาของแต่ละกิจกรรมมีคุณลักษณะเป็นไปตามข้อกำหนดที่นำเข้าทุกข้อที่มีให้ไว้ก่อนหน้าเสมอ [1] ดังนั้น เราจึงควรจัดให้มีการทวนสอบภายหลังแต่ละกิจกรรมสิ้นสุดทันทีทุกครั้งไป เช่น การทวนสอบหลังจบกิจกรรมการรวบรวมความต้องการผู้ใช้ การทวนสอบหลังจบกิจกรรมการวิเคราะห์ระบบ การทวนสอบหลังจบกิจกรรมการออกแบบระบบ และการทวนสอบหลังจบกิจกรรมการสร้างระบบ เป็นต้น

กิจกรรมทวนสอบมักจะได้รับการกล่าวเปรียบเทียบกับกิจกรรมตรวจสอบความสมเหตุสมผล (validation) อยู่บ่อยครั้ง หรือที่เรามักอ้างถึงกิจกรรมทั้งคู่เปรียบเทียบกันเสมอโดยใช้คำว่า “*Verification & Validation*” ข้อปฏิบัติการควบคุมคุณภาพกำหนดให้ต้องมีการดำเนินการตามกิจกรรมทั้งคู่เป็นระยะเสมอ ซึ่งเทคนิควิธีการดำเนินการตามกิจกรรมทั้งคู่จะแตกต่างกัน และเพื่อให้ง่ายในการเข้าใจและจดจำ *Barry Boehm* [1] แนะนำว่า เราควรตั้งคำถามเหล่านี้ถามตนเองเสมอระหว่างกระบวนการพัฒนาระบบ โดยสำหรับการทวนสอบให้ตั้งคำถามว่า *“Are we building the product RIGHT?”* และสำหรับการตรวจสอบความสมเหตุสมผลให้ตั้งถามคำถามว่า *“Are we building the RIGHT product?”*

คำอธิบายสำหรับคำถามของ *Boehm* ข้างต้นคือ การทวนสอบจะเป็นการตอบคำถามว่า เราพัฒนาระบบถูกต้องตามข้อกำหนดที่ให้ไว้ทุกข้อไหม ถ้าข้อกำหนดก่อนหน้าระบุไว้อย่างไรแล้ว เราจะต้องทำตามทุกข้อและให้ครบถ้วนเสมอ จึงจะเรียกว่า เราได้ระบบที่ไม่ผิดพลาดเพราะทวนสอบผ่านทุกข้อ (คำว่า *“RIGHT”* ขยายความคำว่า *“building”*) แต่การตรวจความสมเหตุสมผลมุ่งไปที่ความต้องการที่จริงแท้ของผู้ใช้ โดยให้เน้นว่าเมื่อระบบพัฒนาเสร็จและติดตั้งใช้งานแล้ว จะเป็นที่พอใจและเป็นประโยชน์ต่อผู้ใช้อย่างแท้จริงหรือไม่ (คำว่า

"RIGHT" ขยายความคำว่า *"product"*) สาเหตุที่ Boehm กล่าวเช่นนี้ก็เนื่องจาก นักวิเคราะห์อาจจะไม่มีความเชี่ยวชาญในโดเมนของธุรกิจของลูกค้าหรือเข้าใจปัญหาอย่างคลาดเคลื่อน ทำให้การระบุและรวบรวมความต้องการผู้ใช้ไม่ถูกต้องและครบถ้วนทั้งหมด หลังจากนั้นนำมาเขียนเป็นข้อกำหนดความต้องการส่งต่อมาเรื่อย ๆ นักออกแบบก็ทำการออกแบบตามข้อกำหนดการวิเคราะห์ที่อย่างไม่ผิดพลาดใด ๆ การทวนสอบการออกแบบผ่านไปได้ และในทางปฏิบัตินักออกแบบคงไม่ได้มีโอกาสย้อนกลับไปพูดคุยกับผู้ใช้จริงได้อีกครั้ง ยกเว้นว่าจะมีขั้นตอนการตรวจความสมเหตุสมผลเพิ่มเติมและต้องย้อนกับไปที่ความต้องการที่แท้จริงของผู้ใช้อีกครั้ง

ถ้าเรามีความจำเป็นต้องดำเนินการทวนสอบระบบแล้ว จะต้องทำอย่างไร แนวทางการทวนสอบแบ่งเป็นสองแบบ แบบแรก คือ การทวนสอบเชิงอรูปนัย (informal verification) ซึ่งเป็นแนวทางที่ใช้วิธีปฏิบัติที่ดีที่สุด (best practices) ที่ถูกเขียนไว้เป็นมาตรฐานในคู่มือประกันคุณภาพ เช่น การทวนสอบต้องมีการประชุมร่วมกันของคณะผู้เชี่ยวชาญเพื่อทบทวน และยืนยันผลลัพธ์ว่าเป็นไปตามข้อกำหนดหรือไม่ โดยผู้ทวนสอบสามารถเลือกประเภทของการประชุมร่วมแบบใดก็ได้ตามความเหมาะสม เช่น ประชุมเพื่อพิจารณาทบทวน (review) ประชุมเพื่อการตรวจสอบ (audit) หรือประชุมแบบตรวจตลอด (walkthrough) เป็นต้น ทั้งนี้การทวนสอบเชิงอรูปนัยมักจะถูกดำเนินการและมีการตัดสินผลลัพธ์ด้วยดุลพินิจของมนุษย์ (human judgement) ดังนั้นปัจจัยที่สำคัญ คือ การบริหารจัดการขั้นตอนดำเนินการทวนสอบ และการเลือกผู้เชี่ยวชาญในโดเมนธุรกิจอย่างเหมาะสม

ในขณะที่อีกแบบหนึ่ง คือ การทวนสอบเชิงรูปนัย (formal verification) เป็นการทวนสอบที่ใช้วิธีเชิงรูปนัย (formal methods) และมักจะทำได้โดยการใช้เครื่องมือทางคณิตศาสตร์และตรรกศาสตร์มาช่วยพิสูจน์ การพิสูจน์ที่ว่านี้คือ การพิสูจน์ทางคณิตตรรกศาสตร์ที่เราเรียนรู้กันมาตั้งแต่ชั้นมัธยมศึกษา เพื่อหาทางยืนยันว่า ระบบซอฟต์แวร์และฮาร์ดแวร์ที่เราออกแบบและพัฒนานั้น มีโครงสร้างและพฤติกรรมไม่ผิดพลาดจากข้อกำหนดที่ได้รับมาก่อนหน้า โดยใช้ข้อเท็จจริง สมมุติฐานจากระบบที่เราการออกแบบ และทฤษฎีทางคณิตศาสตร์และตรรกศาสตร์ที่เหมาะสม ตำราเล่มนี้จะเน้นเฉพาะการทวนสอบเชิงรูปนัยของขั้นตอนการออกแบบระบบเป็นสำคัญ เพื่อเป็นการพิสูจน์ยืนยันว่า ผลลัพธ์การออกแบบระบบมีคุณลักษณะตามที่กำหนดไว้ก่อนหน้าในขั้นตอนการวิเคราะห์หรือไม่

วิธีเชิงรูปนัยที่กล่าวถึงข้างต้น และนำมาใช้ในการทวนสอบ คือ การนำคณิตศาสตร์และตรรกศาสตร์มาช่วยพิสูจน์ในกระบวนการพัฒนาระบบซอฟต์แวร์

และฮาร์ดแวร์ วิธีเชิงรูปนัยดังกล่าวนี้จะเกี่ยวข้องกับการสร้างแบบจำลองเชิงรูปนัย (formal model) ที่นำมาใช้ในกระบวนการพิสูจน์เสมอ เช่น แบบจำลองระบบซอฟต์แวร์ที่เขียนอธิบายโครงสร้างและพฤติกรรมขององค์ประกอบของซอฟต์แวร์ (software components) โดยอาจอธิบายได้ด้วยภาษาเชิงรูปนัยหรือการใช้แผนภาพก็ได้ กรณีที่อธิบายด้วยภาษาเชิงรูปนัยมักจะใช้ประโยคทางคณิตตรรกศาสตร์ของเซตและตรรกศาสตร์เชิงประพจน์ และเราพิสูจน์ว่าแบบจำลองนี้มีคุณลักษณะตามที่ต้องการระบุในข้อกำหนดได้ โดยใช้เทคนิคการอนุมานทางตรรกศาสตร์ เป็นต้น

การทวนสอบโดยใช้วิธีเชิงรูปนัยแทนวิธีอรูปนัยแบบเดิมที่ใช้ดุลยพินิจของผู้เชี่ยวชาญ ทำให้การพัฒนาระบบมีต้นทุนด้านเวลาและค่าใช้จ่ายมากกว่าเดิม อีกทั้งยังต้องการบุคลากรที่มีความรู้ความสามารถด้านคณิตศาสตร์และตรรกศาสตร์ แต่ผลลัพธ์ที่ได้ก็คุ้มค่าเสมอ โดยเฉพาะอย่างยิ่งสำหรับระบบวิกฤติ (critical system) ที่ต้องทำงานด้วยความปลอดภัยสูง และทำงานโดยถูกควบคุมด้วยจังหวะของเวลาและต้องสามารถตอบสนองอย่างแม่นยำทันกาล วิธีเชิงรูปนัยเอื้อให้เราสามารถพิสูจน์ว่า การออกแบบระบบวิกฤตินี้มีความผิดพลาดหรือไม่ก่อนลงมือพัฒนาให้เสร็จจริงได้ มีกรณีตัวอย่างที่เกิดขึ้นแล้วว่า ระบบฮาร์ดแวร์ที่มีการออกแบบผิดพลาดแม้ส่วนเล็กน้อยก็ตาม แต่ตรวจหาข้อผิดพลาดด้วยวิธีทวนสอบแบบอรูปนัยเดิมไม่พบ เมื่อนำไปลงมือสร้างชิ้นส่วนจริง แล้วพบข้อผิดพลาดของการทำงานในภายหลัง จะทำให้เกิดความเสียหายต่อผู้ผลิตทั้งเสียเวลาและเงินทุนมากมาย แต่ที่สำคัญยิ่งกว่านั้นคือ ความผิดพลาดที่เสี่ยงต่อชีวิตและทรัพย์สินของผู้ใช้งาน ผู้ผลิตระบบฮาร์ดแวร์ดังกล่าว ได้นำวิธีการทวนสอบเชิงรูปนัยย้อนกลับมาพิสูจน์แบบจำลองเชิงรูปนัยของการออกแบบวงจรที่มีข้อผิดพลาดเดิม และค้นพบข้อผิดพลาดได้ในชั้นการทวนสอบข้อกำหนดการออกแบบวงจรได้

1.4 ข้อกำหนดเชิงรูปนัย

คำว่า “รูปนัย” หมายถึงการนำเสนอหรือแสดงด้วยภาษาที่มีวากยสัมพันธ์ที่จำกัด (restricted syntax) ที่สามารถกำหนดความหมาย (semantic) ได้อย่างชัดเจนโดยแนวคิดทางคณิตศาสตร์และตรรกศาสตร์ จากเอกสาร *RFC2828* [2] ได้นิยามคำว่า “ข้อกำหนดเชิงรูปนัย” คือ ข้อกำหนดของหน้าที่ (functions) และพฤติกรรมการทำงาน (behaviors) ของระบบซอฟต์แวร์หรืออุปกรณ์ฮาร์ดแวร์ที่อธิบายด้วยภาษาทางคณิตศาสตร์ โดยมีวัตถุประสงค์เพื่อใช้ในการพิสูจน์ความถูกต้อง (correctness) ของระบบดังกล่าว และจากเอกสาร *ISO15288-2015* [3] ได้นิยามคำว่า “ข้อกำหนดเชิงรูปนัย” คือ ข้อกำหนดที่เขียนขึ้นด้วยสัญลักษณ์ทาง

คณิตศาสตร์ที่เป็นมาตรฐาน และมักจะถูกใช้ในการพิสูจน์ความถูกต้องของระบบที่ต้องการอธิบาย

ข้อกำหนดเชิงรูปนัยเพื่อการวิเคราะห์ระบบ โดยทั่วไปมักจะถูกเขียนขึ้นเพื่อการบรรยายคุณสมบัติของระบบที่ควรมีอยู่ โดยเน้นระบุถึงคำว่า *"WHAT"* หมายถึง สิ่งใดที่ระบบควรทำได้บ้าง โดยไม่พยายามบรรยายวิธีทำหรืออัลกอริทึม (algorithm) ภายในระบบ ส่วนข้อกำหนดเชิงรูปนัยเพื่อการออกแบบจะเน้นเพิ่มการบรรยายความว่า ระบบจะมีโครงสร้างและพฤติกรรมอย่างไร โครงสร้างของระบบจะต้องประกอบด้วยองค์ประกอบใดบ้าง และระหว่างองค์ประกอบเหล่านั้นมีการเชื่อมต่อกันอย่างไร ตลอดจนพฤติกรรมของแต่ละองค์ประกอบมีการติดต่อและสัมพันธ์ต่อกันอย่างไร รวมถึงระบบจะมีปฏิสัมพันธ์อย่างไรกับผู้ใช้งานและสภาพแวดล้อมรอบนอกหรือระบบที่เกี่ยวข้องภายนอกที่ทำงานร่วมกัน โดยเป็นการระบุถึงคำว่า *"HOW"* นั่นเอง

วิธีการเขียนข้อกำหนดเชิงรูปนัยนี้ มักจะช่วยเพิ่มคุณภาพของประโยคในข้อกำหนดให้มีความถูกต้อง (correctness) และความต้องกัน (consistency) มากขึ้น เมื่ออ่านแล้วจะไม่พบความขัดแย้ง ความผิดพลาด ทั้งยังลดความกำกวมของการบรรยายเนื้อหาได้ เป็นการใช้ภาษาที่บรรยายแล้วไม่เกิดการตีความหมายประโยคที่แตกต่างกันระหว่างผู้เขียนและผู้อ่านข้อกำหนด นอกจากนี้ข้อกำหนดเชิงรูปนัยยังสามารถนำไปใช้ต่อในการทวนสอบหาความผิดพลาดของพฤติกรรมการทำงานของระบบได้ด้วยก่อนจะนำไปพัฒนาสร้างระบบขึ้นจริง ซึ่งเป็นการประหยัดเวลาและค่าใช้จ่าย ตัวอย่างเช่น ระบบขายสินค้าออนไลน์มีข้อกำหนดเขียนไว้ว่า *"ลูกค้าที่เป็นสมาชิกที่มีวงเงินบัตรเครดิตเพียงพอเท่านั้น ที่จะได้รับอนุมัติใบสั่งซื้อสินค้าได้"* อาจจะต้องมีการทำความเข้าใจในรูปประโยค แต่ถ้าเราเขียนข้อกำหนดนี้ด้วยวิธีเชิงรูปนัยโดยใช้สัญลักษณ์และเพรดิเคตทางคณิตตรรกศาสตร์จะเป็น "$member(user) \wedge (credit_line\ (user) \geq amount(order) \Rightarrow approve(order)$" เราพบว่าประโยคในข้อกำหนดจะไม่กำกวม เป็นต้น

เราจำเป็นจะต้องให้ข้อกำหนดเชิงรูปนัยเขียนด้วยภาษาเชิงรูปนัยที่เลือกมาอย่างเหมาะสมกับประเภทและพฤติกรรมของระบบเป็นสำคัญ มิฉะนั้นแล้วการนำไปใช้งานต่อในการทวนสอบจะเกิดอุปสรรคได้ เนื่องจากภาษาเชิงรูปนัยที่ไม่เหมาะสมจะไม่มีตัวอักขระและคำศัพท์ ตลอดจนตัวดำเนินการที่ออกแบบมาสนับสนุนลักษณะพฤติกรรมการทำงานของระบบ ปัจจุบันภาษาเชิงรูปนัยมีหลากหลายชนิดให้เลือกใช้ได้ ตัวอย่างเช่น ภาษาเซด (Z) มีความเหมาะสมให้ใช้เขียนบรรยายระบบที่มีการทำงานแบบเส้นโยงใยเดี่ยว (single thread) ซึ่งจะแตกต่างกับภาษาซีเอสพี (CSP) ที่เหมาะสมกับระบบเชิงพร้อมกัน (concurrent system) ที่มีการทำงานประสานกันระหว่างหลายสายโยงใย (multiple thread)

ส่วนภาษาโพรเมลา (Promela) เหมาะสำหรับอธิบายการทำงานของโพรโทคอล (protocol) ที่ใช้ติดต่อสื่อสารกัน เป็นต้น

1.5 แบบจำลองเชิงรูปนัย

"แบบจำลอง" คือ ตัวแทนเชิงนามธรรม (abstraction representation) ที่ถูกสร้างขึ้นเพื่อใช้อธิบายสิ่งที่เรากำลังสนใจศึกษา ถ้าเรากล่าวถึงแบบจำลองเชิงรูปนัย (formal model) แล้ว เรามักจะหมายถึงแบบจำลองระบบที่ใช้สัญลักษณ์ทางคณิตศาสตร์ พีชคณิตและตรรกศาสตร์อธิบายและสร้างแบบจำลองระบบ

แบบจำลองเชิงรูปนัยประกอบด้วยสองส่วนหลัก ส่วนแรก คือ ข้อกำหนดเชิงรูปนัย (formal specifications) และส่วนที่สอง คือ วิธีการตรวจสอบหรือวิธีพิสูจน์ (proof method) ความถูกต้องสำหรับข้อกำหนดเชิงรูปนัยที่เขียนขึ้นในส่วนแรก ดังนั้น สำหรับผู้ที่ออกแบบภาษาเชิงรูปนัยและพัฒนาเครื่องมือที่ใช้ในการสร้างแบบจำลองเชิงรูปนัยที่มีใช้กันอยู่ในปัจจุบัน ก็จะต้องนิยามวากยสัมพันธ์และความหมายของภาษาที่นำมาใช้ในการเขียนข้อกำหนด พร้อมทั้งต้องกำหนดวิธีพิสูจน์ในภาษาที่ออกแบบขึ้นมาด้วย เช่น ผู้ออกแบบภาษาเซดได้ออกแบบวากยสัมพันธ์และความหมายของสัญกรณ์เซดและชุดคำสั่งในภาษาเซด พร้อมทั้งกำหนดวิธีการพิสูจน์ประโยคที่เขียนด้วยภาษาเซดไว้ให้ด้วย โดยเซดเลือกการพิสูจน์แบบนิรนัย ในขณะเดียวกัน ผู้ออกแบบภาษาเซดพัฒนาเครื่องมือสนับสนุนกระบวนการพิสูจน์ข้อกำหนดที่เขียนด้วยภาษาเซดด้วยเช่นกัน เป็นต้น

การสร้างแบบจำลองเชิงรูปนัยมักทำได้ด้วยการเขียนเป็นข้อกำหนดเชิงรูปนัยด้วยภาษาที่เลือกมาอย่างเหมาะสม โดยผู้เขียนข้อกำหนดเชิงรูปนัยต้องมีความรู้พื้นฐานด้านคณิตตรรกศาสตร์และความเข้าใจโครงสร้างและพฤติกรรมของระบบของตนเองด้วยเช่นกัน นอกจากนี้แล้วยังมีทางเลือกในการสร้างแบบจำลองเชิงรูปนัยด้วยแผนภาพด้วยเช่นกันซึ่งมักจะใช้แผนภาพระบบทรานสิชันอธิบายพฤติกรรมของระบบ

ตัวอย่างการสร้างแบบจำลองด้วยภาษาเชิงรูปนัย

ขอยกตัวอย่าง ข้อกำหนดเชิงรูปนัยภาษาเซดของแบบจำลองเชิงรูปนัยระบบธนาคาร [4] ที่เขียนอธิบายโครงสร้างและพฤติกรรมของระบบไว้โดยสังเขป ส่วนโครงสร้างของระบบประกอบด้วยข้อกำหนดโครงสร้างเซดขององค์ประกอบตัวธนาคารชื่อ *"Bank"* บัญชีเงินฝากชื่อ *"Account"* และระบุส่วนพฤติกรรมของการเปิดบัญชีเงินฝากชื่อ *"OpenAccount"* การฝากเงินเข้าบัญชีเงินฝากชื่อ *"Deposit"* ดังแสดงในรูปที่ 1.2 โครงสร้างเซดของตัวธนาคารจะเก็บข้อมูลหมายเลขบัญชีและตัวบัญชีเงินฝากที่ลูกค้าเปิดไว้ทั้งหมด ในส่วนตัวบัญชีเงินฝาก

จะมีชื่อลูกค้าเจ้าของบัญชีและยอดเงินในบัญชี การเปิดบัญชีเงินฝากครั้งแรกจะเริ่มต้นบัญชีเงินฝากที่ยอดเงินในบัญชีเท่ากับศูนย์เสมอ และเมื่อฝากเงินเข้าบัญชีจะระบุยอดเงินที่ฝากเพิ่ม ซึ่งจะถูกนำไปรวมเพิ่มกับยอดเงินในบัญชีเดิมของบัญชีเสมอ ข้อกำหนดเซดของแบบจำลองระบบธนาคารนี้เป็นตัวอย่างแบบง่ายที่แสดงให้เห็นว่าการใช้วิธีเชิงรูปนัยแทนวิธีอรูปนัยแบบเดิมที่เขียนบรรยายความเป็นประโยคภาษาธรรมชาติทำได้อย่างไร เช่น ประโยคภาษาไทย เป็นต้น

การเรียกใช้งานระบบธนาคารผ่านทางการดำเนินการที่ชื่อ *"Test1"* ที่ทำการเปิดบัญชีและฝากเงินเข้าบัญชีนั่นเอง สำหรับส่วนการพิสูจน์ความถูกต้องของแบบจำลองเชิงรูปนัยระบบธนาคารนี้ทำได้โดยการอนุมานทางตรรกศาสตร์ที่ภาษาเซดกำหนดไว้ อย่างไรก็ตาม มีผู้พัฒนาเครื่องมือที่ช่วยการเขียนภาษาเซดและทำการพิสูจน์แบบจำลองเซดที่มีแบบกึ่งอัตโนมัติ ตัวอย่างข้อกำหนดเซดที่กล่าวมานี้เพียงเพื่อต้องการชี้ให้เห็นตัวอย่างโดยสังเขปของข้อกำหนดเชิงรูปนัยส่วนรายละเอียดเพิ่มเติมเกี่ยวกับภาษาเซดและเครื่องมือจะกล่าวในบทต่อไป

$[ID]$

Bank

$account : ID \rightarrow\!\!\!\!\!+ \; Account$

$accounts : \mathbb{P}\, ID$

$accounts = dom\ account$

Account

$users : \mathbb{P}_1 Client$

$balance : \mathbb{Z}$

OpenAccount

$Account'$

$users? : \mathbb{P}_1 Client$

$users' = users?$

$balance' = 0$

Deposit

$\Delta Account$

$amount? : \mathbb{N}$

$balance' = balance + amount?$

รูปที่ 1.2 ตัวอย่างแบบจำลองเชิงรูปนัยของระบบธนาคาร [4]

นอกจากสร้างแบบจำลองโดยการเขียนข้อกำหนดด้วยภาษาเชิงรูปนัยดังตัวอย่างที่กล่าวมาแล้ว ยังมีทางเลือกอื่นในการสร้างแบบจำลองโดยใช้แผนภาพอธิบายโครงสร้างและพฤติกรรมของระบบ ในที่นี้ขอยกตัวอย่างแผนภาพระบบทรานสิชัน (transition system) เป็นที่นิยมใช้เพื่ออธิบายพฤติกรรมการทำงานของระบบที่ซับซ้อน โดยมีการแจกแจงสถานะและการเปลี่ยนสถานะของระบบ ทำให้ผู้ออกแบบสามารถกำหนดพฤติกรรมการตอบสนองของระบบเมื่อเกิดเหตุการณ์ใดเหตุการณ์หนึ่งได้ การตอบสนองของระบบแสดงออกมาได้ด้วยการกระทำและตามด้วยการเปลี่ยนสถานะ อย่างไรก็ตาม ยังมีแผนภาพอื่นอีกมากมายที่เรานำมาใช้อธิบายพฤติกรรมของระบบได้ เช่น ออโตมาตา กราฟ

สถานะ เพทริเน็ต แผนภาพยูเอ็มแอล เป็นต้น และมีผู้พัฒนาเครื่องมือสนับสนุนในการนำแบบจำลองเชิงรูปนัยที่อธิบายด้วยแผนภาพเหล่านี้ไปใช้ในการทวนสอบหรือการพิสูจน์ด้วยเช่นกัน

ตัวอย่างการสร้างแบบจำลองเชิงรูปนัยด้วยแผนภาพ

นิยามที่ 1-1: ระบบทรานสิชัน

ระบบทรานสิชัน คือ 4-tuple $TS=(S, \Lambda, T, I)$ โดยที่

S คือ เซตของสถานะ

Λ คือ เซตของข้อความกำกับ (label)

T: $S \times \Sigma \longrightarrow S$ คือ ฟังก์ชันทรานสิชัน (transition function) ที่รับคู่ลำดับ (s,l) และได้ผลลัพธ์เป็น u ซึ่ง $s,u \in S$ และ $l \in \Lambda$ โดยที่ s เป็นสถานะปัจจุบัน l เป็นข้อความกำกับ และ u เป็นสถานะถัดไป

I คือ สถานะเริ่มต้น

นิยามที่ 1-2: เส้นทางการกระทำของแผนภาพระบบทรานสิชัน

กำหนดให้ระบบทรานสิชัน $TS=(S,\Lambda,T, I)$ ดังนั้น เส้นทางการกระทำ (execution path) ρ คือ ลำดับของสถานะและเส้นเชื่อมสถานะ $s_1, t_1, s_2, t_2, s_3, t_3, \ldots, s_n$ โดยที่ $s_i \in S$ และ $t_i \in T$ ซึ่ง $s_1=I$ และ $t_i=((s_i,l),s_{i+1})$ ซึ่ง $i=1,2,.., n$ และ $l \in \Lambda$

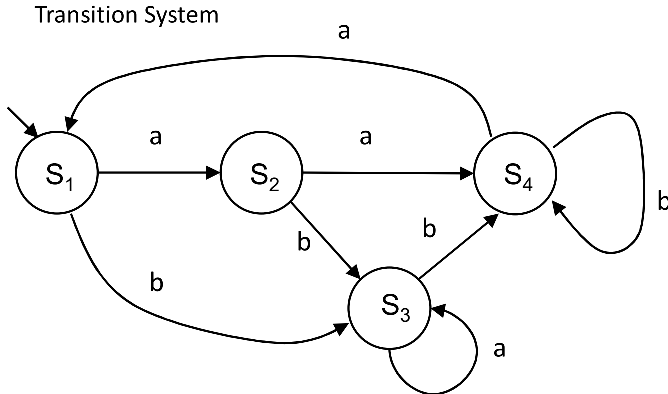
ตัวอย่าง 1-1 แผนภาพระบบทรานสิชันอย่างง่าย

กำหนดให้ $TS=(S, \Lambda, T, I)$ โดยที่

$$S = \{s_1, s_2, s_3, s_4\}$$
$$\Lambda = \{a, b\}$$
$$T = \{((s_1,a), s_2), ((s_1,b), s_3), ((s_2,a), s_4), ((s_2,b), s_3), ((s_3,a), s_3), ((s_3,b), s_4), ((s_4,a), s_1), ((s_4,b), s_4),\}$$
$$I = s_1$$

แผนภาพระบบทรานสิชัน TS พร้อมตัวอย่างเส้นทางการกระทำของแผนภาพวาดแสดงในรูปที่ 1.3 ซึ่งแสดงให้เห็นถึงสถานะทั้งหมดของระบบและการเปลี่ยนแปลงของสถานะที่เกิดขึ้นได้โดยมีข้อความกำกับบนเส้นเชื่อมสถานะ และเพื่อให้สามารถเข้าใจได้ง่าย เราจะมองข้อความกำกับนี้ว่าเป็นการกระทำหรือเหตุการณ์เกิดขึ้นกับระบบได้ เช่น ขณะระบบอยู่ที่สถานะ s_1 และมีการกระทำหรือเหตุการณ์ a จะทำให้เกิดการเปลี่ยนสถานะไปเป็นสถานะ s_2 หรือเมื่อระบบ

อยู่ที่สถานะ s_1 และมีการกระทำหรือเหตุการณ์ b จะทำให้เกิดการเปลี่ยนสถานะไปเป็นสถานะ s_3 เป็นต้น แสดงในรูปที่ 1.3

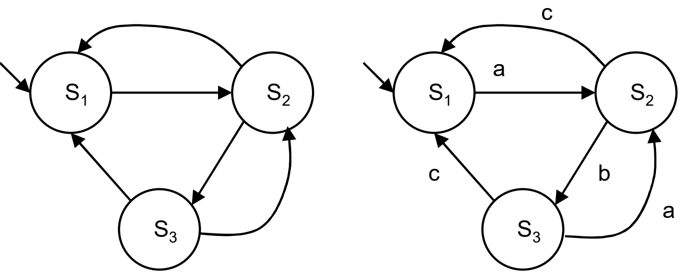


execution path$_1$: $S_1 \xrightarrow{a} S_2 \xrightarrow{b} S_3 \xrightarrow{b} S_4 \xrightarrow{b} S_4 \xrightarrow{b} S_4 \rightarrow \ldots$

execution path$_2$: $S_1 \xrightarrow{b} S_3 \xrightarrow{a} S_3 \xrightarrow{b} S_4 \xrightarrow{a} S_1 \xrightarrow{b} S_3 \rightarrow \ldots$

รูปที่ 1.3 แผนภาพระบบทรานสิชันอย่างง่าย

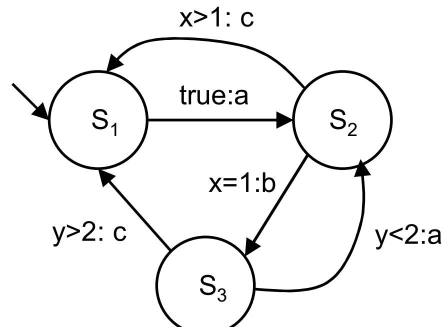
การใช้งานแผนภาพระบบทรานสิชันทำได้หลายวิธี โดยนิยมดัดแปลงตามความเหมาะสม ตัวอย่างเช่น การอธิบายพฤติกรรมของระบบที่ผู้ออกแบบสนใจเฉพาะสถานะและการเปลี่ยนของสถานะเท่านั้น โดยที่เราจะไม่สนใจติดตามค่าตัวแปรใดในระบบ เราจะวาดแผนภาพที่มีแต่สถานะและการกระทำที่แสดงในรูปที่ 1.4 (กรณีที่ไม่มีการระบุการกระทำในแผนภาพด้านซ้ายของรูปที่ 1.4 จะหมายถึงเป็นการกระทำแบบเดียวกัน จึงละข้อความกำกับเส้นไว้เพื่อความง่ายในการอ่านแผนภาพ)



รูปที่ 1.4 แผนภาพระบบทรานสิชันที่ไม่มีการสังเกตข้อมูลตัวแปร

สำหรับผู้ออกแบบที่ต้องการติดตามสังเกตดูค่าตัวแปรที่มีอยู่ในระบบ จะทำให้แบบจำลองของระบบต้องอธิบายพฤติกรรมที่ซับซ้อนมากขึ้น โดยแผนภาพระบบทรานสิชันที่วาดได้ใหม่นี้ จะมีจำนวนสถานะมากขึ้นโดยแปรตามจำนวนตัวแปรและค่าที่เป็นไปได้ของตัวแปรอย่างมีนัยสำคัญ

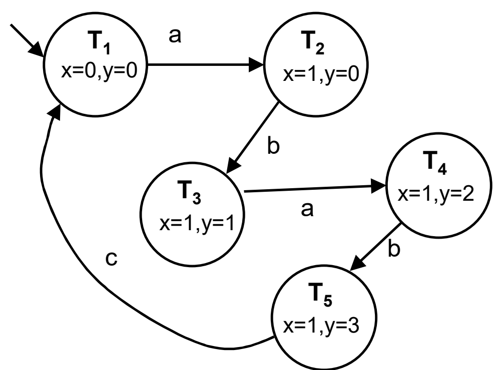
รูปที่ 1.5 แสดงแผนภาพระบบทรานสิชั้นที่มีการเพิ่มส่วนการติดตามสังเกตค่าตัวแปร x, y โดยที่การกระทำ a, b, c ก็จะส่งผลกระทบกับค่าตัวแปรด้วย นอกจากนี้ การเปลี่ยนสถานะก็จะมีเงื่อนไขที่ใช้ค่าตัวแปร x, y นี้มาพิจารณาเพิ่มเติมด้วย เช่น ระหว่างเส้นการเปลี่ยนสถานะจาก S_2 ไปสู่สถานะ S_1 มีกำกับข้อความว่า *"x>1:c"* หมายถึงมีเงื่อนไขว่า ถ้า $x>1$ แล้วจะทำให้เกิดการกระทำ c และเปลี่ยนสถานะถัดไปด้วย



Initialize observation value:
x=0, y=0

Action	***Effect***
a	*x=x+1*
b	*y=y+1*
c	*x=0; y=0*

รูปที่ 1.5 แผนภาพระบบทรานสิชันที่มีการสังเกตข้อมูลตัวแปร x, y



Initialize observation value:
x=0, y=0

Action	***Effect***
a	*x=x+1*
b	*y=y+1*
c	*x=0; y=0*

รูปที่ 1.6 แผนภาพระบบทรานสิชันใหม่ที่แจกแจงค่าตัวแปรที่มีการสังเกตข้อมูลตัวแปร x, y

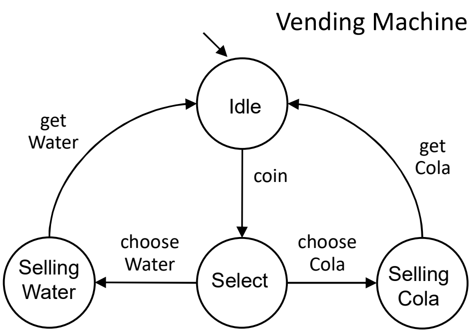
เราเรียกแผนภาพแบบนี้ว่า แผนภาพระบบทรานสิชันแบบขึ้นกับข้อมูล (transition system with data-dependent) และเราต้องวาดแผนภาพแจกแจงการสังเกตค่าตัวแปรเสียใหม่โดยจะได้แผนภาพใหม่แสดงในรูปที่ 1.6 โดยแผนภาพใหม่นี้ จะเริ่มอธิบายตั้งแต่สถานะเริ่มต้นที่ด้วยค่าตัวแปรที่มีค่าเริ่มต้นตามที่กำหนด และแสดงพฤติกรรมที่เกิดจริงตามเงื่อนไขที่ระบุ มีการแจกแจงสถานะเพิ่มใหม่ให้สอดคล้องกับเงื่อนไขของข้อมูลที่กำหนดให้

ตัวอย่าง 1-2 แผนภาพระบบทรานสิชันของระบบเครื่องขายอัตโนมัติ

แบบจำลองเชิงรูปนัยของระบบเครื่องขายอัตโนมัติด้วยแผนภาพ ที่มีข้อกำหนดการวิเคราะห์มาแล้วว่าจะต้องมีการทำงานอะไรบ้าง ดังนี้

"ระบบเครื่องขายอัตโนมัติเมื่อเปิดเครื่องเริ่มต้นจะพร้อมรอรับการหยอดเหรียญอยู่เสมอ และเมื่อมีลูกค้าหยอดเหรียญแล้ว ลูกค้าจะต้องเลือกว่าจะซื้อน้ำเปล่าหรือซื้อโคลา ถ้าลูกค้าเลือกซื้อน้ำเปล่าแล้วระบบจะทำการปล่อยขวดน้ำเปล่ามาที่ช่องรับสินค้า และกลับมารอลูกค้ารายต่อไป ในขณะที่ถ้าลูกค้าเลือกซื้อโคลาแล้วระบบจะทำการปล่อยขวดโคลาที่ช่องรับสินค้า และกลับมารอลูกค้ารายต่อไปเช่นกัน"

เราสามารถตีความข้อกำหนดการวิเคราะห์ข้างต้นเพื่อออกแบบพฤติกรรมของระบบเครื่องขายอัตโนมัติ โดยทำการกำหนดองค์ประกอบของแผนภาพไว้ก่อน คือ เซตของสถานะที่ควรจะมี ซึ่งพบว่าระบบควรมีสถานะ *"Idle"* ในตอนเริ่มต้นเพื่อรอให้บริการลูกค้า จากนั้นเมื่อมีการกระทำ *"coin"* คือการหยอดเหรียญแล้ว ระบบควรจะเปลี่ยนสถานะไปสู่สถานะ *"Select"* คือรอการเลือกชนิดของสินค้า จากนั้นเมื่อมีการกระทำ *"choose Water"* คือ การเลือกซื้อน้ำเปล่า ระบบควรจะเปลี่ยนสถานะไปสู่สถานะ *"Selling Water"* และมีเหตุการณ์ *"get Water"* คือ การที่เครื่องปล่อยขวดน้ำไปช่องรับสินค้า ระบบก็จะเปลี่ยนสถานะกลับไปสู่สถานะ *"Idle"* เดิมเพื่อรอลูกค้าต่อไป ย้อนกลับที่สถานะ *"Select"* ถ้ามีการกระทำ *"choose Cola"* คือ การเลือกซื้อโคลา ระบบก็จะเปลี่ยนสถานะไปสู่สถานะ *"Selling Cola"* และมีเหตุการณ์ *"get Cola"* คือการที่เครื่องปล่อยขวดโคลาไปช่องรับสินค้า จากนั้นระบบจะเปลี่ยนสถานะกลับไปสู่สถานะ *"Idle"* เดิม โดยพฤติกรรมที่อธิบายมาข้างต้นแสดงด้วยแผนภาพระบบทรานสิชันในรูปที่ 1.7 พร้อมด้วยตัวอย่างเส้นทางการกระทำของระบบ เราสามารถตามรอยพฤติกรรมของระบบนี้ได้ตามเส้นทางการกระทำที่หามาได้นั่นเอง



$$\rho_1: Idle \xrightarrow{coin} Select \xrightarrow{choose\ Water} Selling\ Water \xrightarrow{get\ Water} Idle \rightarrow ...$$
$$\rho_2: Idle \xrightarrow{coin} Select \xrightarrow{choose\ Cola} Selling\ Cola \xrightarrow{get\ Cola} Idle \rightarrow ...$$

รูปที่ 1.7 แผนภาพระบบทรานสิชันของระบบเครื่องขายอัตโนมัติ [5]

ตัวอย่าง 1-3 แผนภาพระบบทรานสิชันแบบขึ้นกับข้อมูลของระบบเครื่องขายอัตโนมัติ

แบบจำลองเชิงรูปนัยของระบบเครื่องขายอัตโนมัติด้วยแผนภาพที่มีข้อกำหนดการวิเคราะห์มาแล้วว่าจะต้องมีการทำงานอะไรบ้าง และระบบมีตัวแปรที่ต้องการติดตามสังเกตอยู่ด้วย ดังนี้

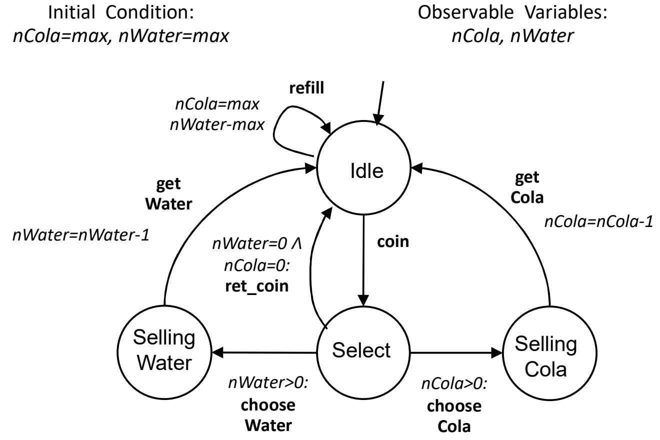
“ระบบเครื่องขายอัตโนมัติเมื่อเปิดเครื่องเริ่มต้นจะพร้อมรอรับการหยอดเหรียญอยู่เสมอ และเมื่อมีลูกค้าหยอดเหรียญแล้ว ลูกค้าจะต้องเลือกว่าจะซื้อน้ำเปล่าหรือซื้อโคลา ถ้าลูกค้าเลือกซื้อน้ำเปล่าแล้วระบบจะทำการปล่อยขวดน้ำเปล่ามาที่ช่องรับสินค้า และกลับมารอลูกค้ารายต่อไป ในขณะที่ถ้าลูกค้าเลือกซื้อโคลาแล้วระบบจะทำการปล่อยขวดโคลาที่ช่องรับสินค้า และกลับมารอลูกค้ารายต่อไปเช่นกัน

โดยระบบมีตัวแปรที่สนใจติดตามสังเกต และเป็นเงื่อนไขในการเปลี่ยนสถานะของระบบด้วย ตัวแปรที่สนใจ คือ $nCola$ แสดงจำนวนขวดโคลาที่เครื่องขายมีอยู่ $nWater$ แสดงจำนวนขวดน้ำเปล่าที่เครื่องขายมีอยู่ max แสดงค่าสูงสุดของจำนวนขวดเครื่องดื่มที่เครื่องขายมีอยู่ โดยการกระทำที่เกิดขึ้นในระบบจะส่งผลต่อค่าในตัวแปรได้ คือ ถ้ามีการขายโคลาแล้วจำนวนขวดโคลาจะลดลงหนึ่งขวด และถ้ามีการขายน้ำเปล่าแล้วจำนวนขวดน้ำเปล่าจะลดลงหนึ่งขวด

นอกจากนี้มีเงื่อนไขและการกระทำที่เพิ่มเติม คือ เงื่อนไขก่อนขายโคลาต้องมีจำนวนขวดโคลามากกว่าศูนย์ เงื่อนไขก่อนขายน้ำเปล่าต้องมีจำนวนขวดน้ำเปล่ามากกว่าศูนย์ และให้เพิ่มการกระทำคืนเหรียญโดยมีเงื่อนไขว่า ถ้า

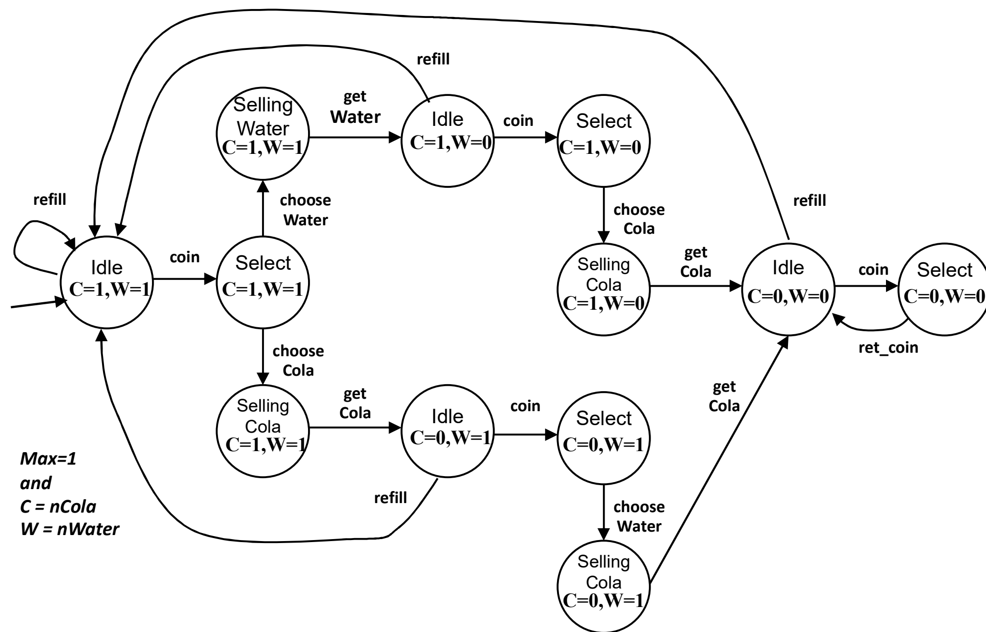
จำนวนขวดน้ำเปล่าและโคลาเท่ากับศูนย์ และเพิ่มการกระทำเติมขวดน้ำสินค้าเพื่อให้จำนวนขวดน้ำเปล่าและโคลามีค่าเท่ากับ max”

เนื่องจากผู้ออกแบบจำเป็นต้องติดตามสังเกตตัวแปรที่กำหนดมาให้ คือ *nCola* และ *nWater* พร้อมทั้งการกระทำต่าง ๆ ในระบบจะส่งผลกับค่าตัวแปรทั้งสองด้วย ทำให้ต้องดัดแปลงมาใช้แผนภาพระบบทรานสิชันแบบขึ้นกับข้อมูล แสดงในรูปที่ 1.8 โดยการเพิ่มการกำกับเงื่อนไขและผลที่มีต่อตัวแปรให้ปรากฏบนเส้นเชื่อมสถานะอย่างครบถ้วน และมีการกำหนดค่าเริ่มต้นระบบโดยให้ค่าตัวแปรมีค่าเท่ากับ *max* ด้วย



รูปที่ 1.8 แผนภาพระบบทรานสิชันแบบขึ้นกับข้อมูลของระบบเครื่องขายอัตโนมัติ [5]

ขั้นตอนต่อมา คือ การแจกแจงสถานะในแผนภาพระบบทรานสิชันแบบขึ้นกับข้อมูลให้มีสถานะครบถ้วนตามที่แสดงในรูปที่ 1.9 เพื่อจะได้มีการแสดงค่าตัวแปรที่ต้องการติดตามสังเกตปรากฏอยู่ด้วยในแผนภาพ วิธีการแจกแจงให้เริ่มกำหนดค่าเริ่มต้นของ *max* และตัวแปร *nCola* และ *nWater* ให้พร้อม หลังจากนั้นให้ทำการเดินติดตามแผนภาพไปทีละสถานะให้ครบถ้วน เมื่อพบสถานะใหม่ให้เขียนเพิ่มเติมไว้ด้วย โดยตอนนี้แต่ละสถานะจะมีองค์ประกอบเป็น (*StateName,nCola,nWater*) ซึ่งก็คือชื่อสถานะและค่าตัวแปรทุกตัวที่ติดตามสังเกตทั้งหมด ถ้าองค์ประกอบมีค่าต่างกันก็ถือได้ว่าเป็นสถานะใหม่ เช่น (*Idle,C=1,W=1*), (*Idle,C=0,W=1*), (*Idle,C=1,W=0*) และ (*Idle,C=0,W=0*) ต่างเป็นสถานะที่มีองค์ประกอบแตกต่างกันและถือว่าเป็นสถานะใหม่ที่เพิ่มขึ้นในแผนภาพ จากเดิมที่มีแค่สถานะเดียวชื่อ *Idle* เท่านั้น เป็นต้น



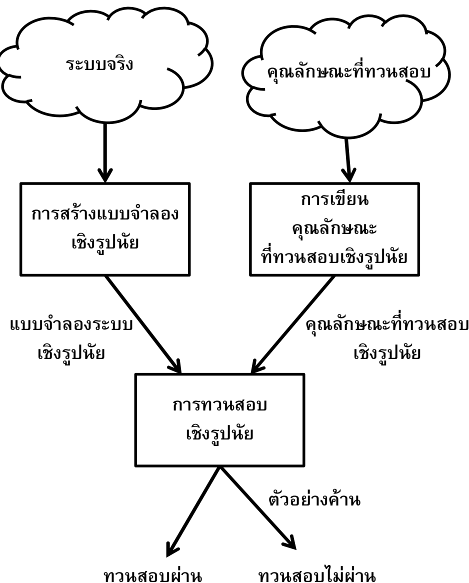
รูปที่ 1.9 แผนภาพระบบทรานสิชันใหม่ที่แจกแจงค่าตัวแปรแบบขึ้นกับข้อมูลของระบบเครื่องขายอัตโนมัติ

1.6 การทวนสอบเชิงรูปนัย

การทวนสอบเชิงรูปนัยจะนำแบบจำลองเชิงรูปนัยมาตรวจสอบว่า แต่ละประโยคที่เขียนขึ้นมานั้นมีข้อผิดพลาดไปจากนิยามของวากยสัมพันธ์ของภาษาที่กำหนดไว้หรือไม่ และเมื่อนำทุกประโยคมารวมกันแล้วจะมีความต้องกันหรือไม่ บางส่วนของแบบจำลองมีลักษณะผิดปกติหรือไม่ เช่น มีประโยคคำสั่งใดที่เข้าไม่ถึงได้ (unreachable statement) หรือมีส่วนใดของแบบจำลองเกิดการติดตาย (deadlock) หรือไม่ เป็นต้น นอกจากนี้การทวนสอบเชิงรูปนัยจะทำการพิสูจน์ความถูกต้องของคุณลักษณะอื่นของระบบที่กำหนดไว้ได้ด้วยเช่นกัน ยกตัวอย่างเช่น คุณลักษณะความปลอดภัย (safety property) คุณลักษณะความดำเนินชีวิต (liveness property) เป็นต้น หรือ คุณลักษณะที่กำหนดไว้ในปัจจัยคุณภาพของระบบ ซึ่งคุณลักษณะที่ถูกนำใช้ในการทวนสอบเหล่านี้จะถูกกล่าวถึงในบทต่อไป

การทวนสอบเชิงรูปนัยทำได้โดยนำข้อกำหนดเชิงรูปนัยของแบบจำลองเชิงรูปนัยของระบบซึ่งเป็นที่ยอมรับในเบื้องต้นว่า ข้อกำหนดดังกล่าวเป็นตัวแทนที่บรรยายโครงสร้างและพฤติกรรมของระบบที่ครบถ้วนเพียงพอ แล้วนำไปพิสูจน์ด้วยวิธีพิสูจน์ที่กำหนดไว้ในประเภทของแบบจำลองเชิงรูปนัยที่เลือกใช้

แนวปฏิบัติทั่วไปในการทวนสอบเชิงรูปนัย จะต้องมีองค์ประกอบตั้งต้นสองส่วน คือ แบบจำลองเชิงรูปนัยที่แสดงพฤติกรรมของระบบ และคุณลักษณะที่ต้องการทวนสอบเชิงรูปนัยตามที่แสดงในรูปที่ 1.10 แบบจำลองระบบอาจจะได้มาจากข้อกำหนดการวิเคราะห์ระบบ การใช้วิธีเชิงรูปนัยในขั้นตอนนี้เป็นช่วงสำคัญมากและทำได้ไม่ง่ายนัก



รูปที่ 1.10 แนวปฏิบัติทั่วไปในการทวนสอบเชิงรูปนัย

ปัจจุบันการนำวิธีทวนสอบแบบนี้ไปใช้จริงมักจะติดปัญหาเรื่องการสร้างแบบจำลองเชิงรูปนัย ทำให้มีงานวิจัยจำนวนไม่น้อยหันมานำเสนอกฎหรือกระบวนวิธีในการสร้างแบบจำลองเชิงรูปนัย เช่น ถ้าเราต้องการออกแบบระบบรางของการเดินรถไฟ เราจะสร้างแบบจำลองเชิงรูปนัยได้อย่างไร โดยที่จะเป็นตัวแทนระบบรางของการเดินรถไฟที่ประกอบด้วย การวางแนวรางรถไฟ ขนาดและความยาวของราว อาณัติสัญญาณที่จำเป็นสำหรับระบบรางและการเดินรถไฟเป็นต้น เพื่อใช้ในการทวนสอบว่าจะไม่เกิดการติดขัด หรือติดตายของการเดินรถไฟในแต่ละวัน ตารางเวลาการเดินรถไฟจะมีปัญหาหรือไม่ ประเภทของรถไฟที่ใช้วิ่งบนระบบรางที่มีความแตกต่างกันจะส่งผลอย่างไรต่อตารางเวลาการเดินรถเป็นต้น การนำการทวนสอบเชิงรูปนัยไปใช้กับงานจริงหลังจากได้แบบจำลองระบบแล้ว มักจะเป็นการใช้เครื่องมือที่พัฒนาขึ้นมาโดยเฉพาะเพื่อสนับสนุนผู้ทวนสอบให้สะดวกและมีความถูกต้องมากขึ้น ซึ่งมักได้ผลลัพธ์จากเครื่องมือการ

ทวนสอบว่าทวนสอบผ่านหรือไม่ ถ้าทวนสอบไม่ผ่านจะมีผลตัวอย่างค้านที่เป็นข้อผิดพลาดแสดงออกมาได้

แนวคิดคุณลักษณะที่ใช้ทวนสอบ

เมื่อเราได้แบบจำลองเชิงรูปนัยที่อธิบายพฤติกรรมของระบบ วิธีกำหนดคุณลักษณะที่ใช้ทวนสอบ ก็คือ การตั้งคำถามที่เกี่ยวกับสิ่งที่ระบบควรจะทำได้หรือไม่ผิดพลาด ส่วนใหญ่คุณลักษณะจะเกี่ยวข้องกับความถูกต้องในการทำงาน ความปลอดภัยจากการทำงานของระบบ ประสิทธิภาพของระบบ คุณลักษณะของระบบที่จำเป็นในระบบวิกฤติ สิ่งที่น่าสังเกต คือ การที่มักจะตั้งคำถามที่เกี่ยวข้องกับเวลาตั้งแต่ระบบเริ่มทำงานและต่อเนื่องไป ต้องมีเงื่อนไขเรื่องเวลาในอนาคตมาเกี่ยวข้องด้วยเสมอ ทำให้เป็นที่มาของแนวคิดของ *Pnueli* ที่ให้ใช้ตัวดำเนินการเชิงเวลาของตรรกศาสตร์เชิงเวลามาร่วมในการกำหนดคุณลักษณะ ในที่นี้ขอไม่กล่าวถึงการใช้สัญลักษณ์ต่าง ๆ ไว้ก่อน โดยเน้นการบรรยายความแทน รายละเอียดเรื่องตรรกศาสตร์เชิงเวลาจะได้กล่าวถึงในบทต่อไป

ตัวอย่างคุณลักษณะที่มักถูกนำมาใช้ทวนสอบเป็นเบื้องต้นสำหรับทุกระบบคือ คุณลักษณะความปลอดภัย (safety) ความดำเนินชีวิต (liveness) และความเป็นธรรม (fairness) จากนั้นเราอาจจะต้องการทราบเพิ่มเติมว่า ระบบที่สนใจทำงานผิดพลาดหรือไม่ เช่น ในระบบฮาร์ดแวร์ เราอาจจะต้องการทราบว่าระบบมีการตอบสนอง (response) ต่อสัญญาณร้องขอ (request) ได้ดีหรือไม่ หรือเมื่อมีการร้องขอแล้วมีการตอบสนองแบบเรียงลำดับความสำคัญก่อนหลัง (precedence of responses) หรือไม่ หรือวงจรการทำงานมีความเสถียรภาพของสัญญาณ (signal stability) หรือไม่ หรือวงจรมีสัญญาณคู่ใดที่มีสหสัมพันธ์กัน (signal correlation) หรือไม่ หรือความไม่เกิดร่วม (mutual exclusion) ของสัญญาณ ตลอดจน เรื่องการเกิดคำร้องขอจนกว่าตอบรับ (request must hold until acknowledge) [6] เป็นต้น รายละเอียดเรื่องคุณลักษณะเหล่านี้จะกล่าวถึงในบทต่อไปเช่นกัน

เทคนิคการทวนสอบเชิงรูปนัย

หลังจากเราได้องค์ประกอบตั้งต้นทั้งสองส่วนมาแล้ว คือ แบบจำลองเชิงรูปนัย และคุณลักษณะที่ใช้ทวนสอบตามที่กล่าวมาแล้วข้างต้น เราเริ่มลงมือทวนสอบได้ เทคนิคการทวนสอบที่นิยมเลือกทำกันมีหลายแบบ อันที่จริงประเภทของระบบและเทคนิคการทวนสอบมักจะย้อนกลับไปเป็นตัวกำหนดว่าแบบจำลองเชิงรูปนัย และคุณลักษณะที่ใช้ทวนสอบที่จะนำมาใช้ควรเป็นอย่างไร โดยเราควรเลือกเทคนิคการทวนสอบเสียก่อนก็จะเป็นการดี

โดยทั่วไปเทคนิคการทวนสอบนิยมเลือกทำกันได้สามแบบ [5] คือ

1) การทวนสอบด้วยการพิสูจน์ทฤษฎีบท (theorem proving) เป็นการใช้เทคนิคการอนุมานแบบนิรนัยของข้อเท็จจริงที่กำหนดให้เบื้องต้นเพื่อพิสูจน์หาข้อยุติเรียกว่าทฤษฎีบท โดยกำหนดให้ข้อกำหนดเชิงรูปนัยของแบบจำลองระบบเป็นข้อเท็จจริงหรือสมมุติฐานเบื้องต้นและให้คุณลักษณะที่ต้องการทวนสอบของเราเป็นข้อยุติที่สงสัยว่าจะเป็นจริงหรือไม่ จากนั้นให้ดำเนินการใช้กฎการอนุมานแบบนิรนัยที่มีอยู่ในคณิตตรรกศาสตร์ เพื่อค้นหาสายโซ่การอนุมานจากจุดเริ่มต้นที่เป็นข้อเท็จจริงไปสู่ข้อยุติสุดท้ายให้ได้ ถ้าเราค้นพบสายโซ่ของการอนุมานดังกล่าวก็แสดงว่า ระบบเรามีคุณลักษณะที่ต้องการทวนสอบจริงเสมอ ปัจจุบันเรามีเครื่องมือแบบกึ่งอัตโนมัติสนับสนุนการทวนสอบด้วยการพิสูจน์ทฤษฎีบท โดยผู้ทวนสอบต้องมีความเข้าใจด้านคณิตตรรกศาสตร์และข้อกำหนดเชิงรูปนัยของระบบเป็นอย่างดีก่อน
2) การทวนสอบด้วยโมเดลเช็กกิง (model checking) เป็นการพิสูจน์แบบอัตโนมัติด้วยการค้นหาว่าสมการ “$M \models \Phi$” เป็นจริงหรือไม่ โดย M คือ พฤติกรรมระบบจากแบบจำลองระบบที่เราสนใจในรูปของสถานะและการเปลี่ยนสถานะ ส่วนเครื่องหมาย $\models$ (entailment) คือ เกิดการพอใจ (satisfaction) และสัญลักษณ์ Φ คือ คุณลักษณะเชิงเวลาที่ต้องการทวนสอบเขียนด้วยสูตรเชิงเวลา การทวนสอบจะพิจารณาสถานะทั้งหมดที่เป็นไปได้ ในปริภูมิสถานะของพฤติกรรมของระบบ M แล้วทำการค้นหาว่า มีลำดับการเปลี่ยนแปลงสถานะที่ทำให้ลักษณะเชิงเวลาที่ต้องการ Φ ว่าเป็นจริงหรือไม่ ถ้าพบก็แสดงว่าเกิดการพอใจคุณลักษณะเชิงเวลา Φ นั้น แต่ถ้าไม่พบแสดงว่ามีจุดที่ผิดพลาดก็จะแสดงผลออกมาเป็นตัวอย่างค้าน โดยจะได้นำมาแก้ไขข้อผิดพลาดนั้น ๆ ได้ในภายหลัง ข้อดีของวิธีนี้ คือ ผู้ทวนสอบจะมีเครื่องมือสนับสนุนแบบอัตโนมัติช่วยเหลือ แต่มีข้อเสียหลัก คือ ขนาดของปริภูมิสถานะที่เกิดขึ้นนั้น มักจะมีขนาดใหญ่มากเสียจนเครื่องมือไม่สามารถจัดการได้ เราเรียกเหตุการณ์นี้ว่า เกิดการระเบิดของปริภูมิสถานะ (state space explosion) อย่างไรก็ตาม มีงานวิจัยอื่น ๆ อีกมากที่นำเสนอวิธีการลดขนาดปริภูมิสถานะจนทำให้โมเดลเช็กกิงได้รับการยอมรับและนำมาใช้งานได้ดีในระดับหนึ่ง ทั้งนี้ระบบที่จะนำมาทวนสอบด้วยวิธีนี้จำเป็นต้องมีสถานะจำกัด (finite state) เท่านั้น ข้อจำกัดของโมเดลเช็กกิงอีกอย่าง คือ เป็นการทวนสอบเฉพาะคุณลักษณะเชิงคุณภาพ (qualitative property) เท่านั้น เช่น

ต้องการรู้ว่าระบบติดตายหรือไม่ หรือระบบทำงานแล้วถ้าเกิดสถานะ p แล้วจะมีสถานะ q ตามมาในที่สุดหรือไม่ เป็นต้น

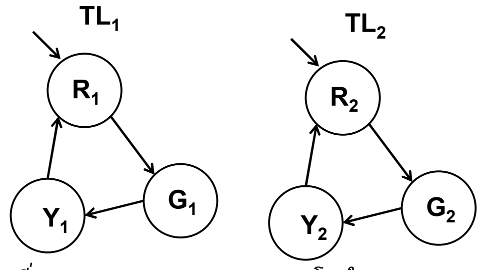
3) การทวนสอบด้วยการตรวจความสมมูล (equivalence checking) ซึ่งพบว่ามักจะใช้ทวนสอบการออกแบบวงจรในระบบฮาร์ดแวร์ นับได้ว่า เป็นกระบวนการตรวจสอบว่า วงจรใหม่ในระบบที่ออกแบบมาเพื่อใช้งานแทนและวงจรต้นฉบับเดิมจะมีพฤติกรรมการทำงานแบบสมมูลกันหรือไม่ การทวนสอบจะสร้างปริภูมิสถานะของการทำงานของทั้งวงจรต้นฉบับและวงจรใหม่เพื่อทำการเปรียบเทียบว่า ถ้ามีสัญญาณนำเข้าวงจรแบบเดียวกันแล้ว จะต้องได้ผลลัพธ์เป็นสถานะของวงจรเหมือนกันเสมอแบบสมมูลกันเท่านั้นจึงจะทวนสอบผ่าน ทำให้เราสามารถใช้วงจรใหม่ทำงานทดแทนวงจรต้นฉบับเดิมได้ ปัญหาที่พบของการทวนสอบแบบการตรวจความสมมูลของวงจร คือ ความจำเป็นต้องมีการตรวจสอบให้มีครอบคลุมทุกกรณีของพฤติกรรมของวงจรเดิมเสมอ ส่งผลให้บางครั้งอาจจะมีปัญหาและอุปสรรคกับวงจรที่มีขนาดใหญ่ และซับซ้อนเนื่องจากสถานะของวงจรต้นฉบับเดิมมีขนาดใหญ่ โดยเฉพาะอย่างยิ่งสำหรับวงจรสมวาร (synchronous circuit) ที่มีการทำงานตามจังหวะของสัญญาณนาฬิกา (clock signal) นั้นพฤติกรรมการทำงานจะต้องสมมูลกันในระดับสัญญาณนาฬิกาด้วยเสมอ อย่างไรก็ตามตำราเล่มนี้จะไม่เน้นการทวนสอบด้วยวิธีนี้

ตัวอย่าง 1-4 การทวนสอบระบบสัญญาณไฟจราจร

ต่อไปนี้เป็นตัวอย่างเพื่อชี้ให้เห็นถึงจำนวนของสถานะและความซับซ้อนของระบบเชิงพร้อมกันที่มีหลายสายโยงใย ระบบสัญญาณไฟจราจรของจุดตัดถนนสี่แยกที่ประกอบด้วยสองเสาสัญญาณไฟ คือ เสาสัญญาณไฟ TL_1 และ TL_2 แต่ละเสาสัญญาณไฟจะถูกควบคุมด้วยโปรแกรมสายโยงใยที่เริ่มต้นด้วยการเปิดไฟแดง ไฟเขียว และไฟเหลือง จากนั้นวนกลับมาเปิดไฟแดงใหม่ วนซ้ำไปไม่สิ้นสุด ตามที่ปรากฏในรูปที่ 1.11 ซึ่งเป็นแผนภาพระบบทรานสิชันของพฤติกรรมการทำงานของสายโยงใยควบคุมการเปิดไฟ โดยในรูปนี้ไม่ได้แสดงการกำกับข้อความที่เส้นเชื่อมสถานะไว้เพื่อความง่ายในการอ่านแผนภาพ

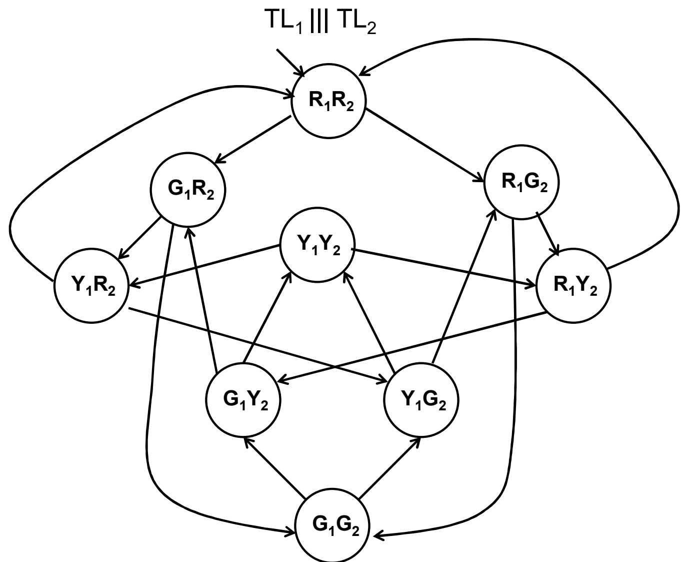
เมื่อพิจารณาในแต่ละสายโยงใยแล้ว จะพบว่าการทำงานไม่มีความซับซ้อนแต่อย่างใด เป็นพฤติกรรมการทำงานแบบลำดับธรรมดา แต่เมื่อเราปรับปรุงระบบให้ทำงานร่วมกันแล้ว เราพบว่าจำนวนสถานะที่พิจารณาได้เบื้องต้นของระบบที่ควบรวมทั้งสองสายโยงใยนี้จะมีจำนวนสถานะเพิ่มขึ้นอย่างมี

นัยสำคัญ โดยแสดงในรูปที่ 1.12 ไม่เพียงแค่เป็นการนำจำนวนสถานะเดิมมาบวกรวมกันแบบง่ายๆ



รูปที่ 1.11 แผนภาพสถานะของสายโยงใย TL1 และ TL2

อย่างไรก็ตามแผนภาพระบบทรานสิชันที่ได้มาเบื้องต้นนี้จะต้องนำมาพิจารณาต่อ เพื่อหาวิธีออกแบบให้ระบบทำงานได้ไม่ผิดพลาด เช่น กรณีที่ทั้งสองเสาสัญญาณไฟเปิดไฟเขียวพร้อมกัน หรือกรณีที่เสาสัญญาณไฟเสาใดเสาหนึ่งไม่เปิดไฟเขียวเลย เป็นต้น



รูปที่ 1.12 แผนภาพระบบทรานสิชันรวมของระบบไฟจราจรประกอบด้วยสายโยงใย TL_1 และ TL_2 ทำงานพร้อมกัน

จากรูปที่ 1.12 เราลองมาหาข้อผิดพลาดเกี่ยวกับคุณลักษณะที่ว่า “สัญญาณไฟเขียวของสองเสาสัญญาณต้องไม่เปิดพร้อมกันอย่างเด็ดขาด” ซึ่งเรา

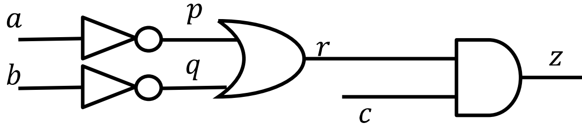
จะทำการทวนสอบโดยการนำเส้นทางการกระทำทุกเส้นทางที่เป็นไปได้มาค้นหาว่ามีกรณีที่สัญญาณไฟเขียวเปิดสว่างทั้งสองเสาสัญญาณหรือไม่ และเราก็พบว่าสถานะชื่อ *"G_1G_2"* ปรากฏในบางเส้นทางการกระทำของระบบนี้ ทำให้เส้นทางที่ผิดพลาดนี้เป็นตัวอย่างค้าน และนำไปปรับปรุงการออกแบบเสียใหม่ได้

ตัวอย่าง 1-5 การทวนสอบแบบการตรวจสอบความสมมูลของวงจรเชิงผสม

การทวนสอบเชิงรูปนัยของการออกแบบระบบฮาร์ดแวร์เริ่มเป็นที่นิยมมากขึ้นกว่าเดิม หลังจากความสำเร็จในการค้นพบข้อผิดพลาดที่การทวนสอบเชิงอรูปนัยแบบเดิมค้นหาไม่พบ และในวงการผลิตชิ้นส่วนฮาร์ดแวร์มีการลงทุนสูงและผลผลิตมีความสูญเสียและสิ้นเปลืองมากกว่า เนื่องจากมีการผลิตชิ้นงานจริงมีการบริหารจัดการคลังสินค้าจริง ในที่นี้ขอยกตัวอย่างภาพรวมของการทวนสอบแผนภาพเค้าร่าง (schematic diagram) ของวงจรเชิงเลข (digital circuit) โดยแผนภาพเค้าร่างมักจะเป็นการออกแบบวงจรด้วยเกตแบบตรรกะเชิงเลข (digital logic gate) ซึ่งแบ่งลักษณะพฤติกรรมการทำงานของวงจรออกเป็นสองแบบ คือ วงจรเชิงผสม (combinational circuit) หรือเรียกว่า วงจรไร้ความจำ และวงจรเชิงลำดับ (sequential circuit) หรือเรียกว่าวงจรมีความจำ

รูปที่ 1.13 แสดงแผนภาพเค้าร่างของวงจรเชิงผสมที่ออกแบบมาให้มีลักษณะเป็นเครือข่ายของเกตที่เชื่อมต่อกัน เราต้องการทวนสอบว่าผลการออกแบบนี้มีพฤติกรรมตามพีชคณิตบูลีน คือ $z = \neg(a \wedge b) \wedge c$ ที่กำหนดมาให้ก่อนหน้าหรือไม่ ดังนั้น การทวนสอบมักจะเป็นการนำแผนภาพเค้าร่างของวงจรนี้มาแปลงเป็นแบบจำลองเชิงรูปนัย และให้นำพีชคณิตบูลีนที่คาดหวังมาเป็นข้อกำหนดคุณลักษณะที่ทวนสอบ

Gate Network (Implementation)



Expected Property

$$z = \neg(a \wedge b) \wedge c$$

รูปที่ 1.13 แผนภาพเค้าร่างวงจรเชิงเลขและพีชคณิตบูลีนเป้าหมาย [12]

กรณีที่เราใช้วิธีทวนสอบเชิงรูปนัยด้วยการตรวจความสมมูล ให้ทำตามขั้นตอนที่แสดงในรูปที่ 1.14 โดยเราก็จะแปลงแผนภาพเครือข่ายของเกตที่เชื่อมต่อกันให้อยู่ในรูปแบบเชิงรูปนัยด้วยเช่นกัน กล่าวคือ เราใช้พีชคณิตบูลีนเช่นเดียวกันได้โดยแจกแจงการเชื่อมต่อกันของเกตแต่ละตัวและได้แบบจำลองเชิงรูปนัยของวงจร ดังนี้

$$p = (\neg a),\ q = (\neg b),\ r = (p \vee q),\ z = (r \wedge c)$$

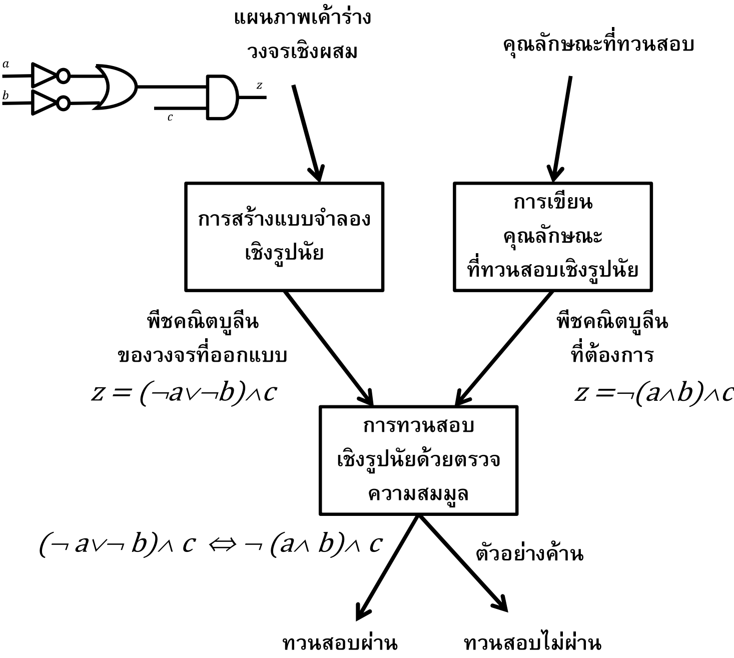
และได้ทำการแทนค่า p, q, r เพิ่มลดรูปสมการ ดังนี้

$z = (\neg a \vee \neg b) \wedge c$ คือแบบจำลองรูปนัยของวงจรแบบที่ได้

จากนั้นเรารู้แล้วว่าคุณลักษณะที่ต้องการทวนสอบคือ $z = \neg (a \wedge b) \wedge c$ เราจึงลงมือทำการทวนสอบด้วยการพิสูจน์ความสมมูลของประพจน์ทั้งสอง

$$(\neg a \vee \neg b) \wedge c \Leftrightarrow \neg (a \wedge b) \wedge c$$

ถ้าผลการพิสูจน์ความสมมูลของประพจน์ข้างต้นเป็นจริง แสดงว่าทวนสอบผ่าน สรุปได้ว่า การออกแบบวงจรเชิงผสมทำงานได้ตามคุณลักษณะที่ทวนสอบได้ แต่ถ้าประพจน์ไม่สมมูลกัน ก็สรุปได้ว่าทวนสอบไม่ผ่าน



รูปที่ 1.14 ตัวอย่างขั้นตอนการทวนสอบวงจรเชิงผสมด้วยการตรวจความสมมูล

1.7 ประเภทระบบซอฟต์แวร์และฮาร์ดแวร์ที่เหมาะกับการทวนสอบ

การทวนสอบเชิงรูปนัยมีต้นทุนด้านเวลาและค่าใช้จ่ายเพิ่มเติมมากกว่าการทวนสอบแบบอรูปนัยเดิม ทำให้เราอาจจะต้องพิจารณาเบื้องต้นว่าจะนำมาใช้ทวนสอบระบบซอฟต์แวร์และฮาร์ดแวร์ประเภทใดจึงจะคุ้มทุน หรือจะยังคงใช้วิธีทวนสอบแบบอรูปนัยเดิมที่ใช้ดุลยพินิจของผู้เชี่ยวชาญ

ระบบซอฟต์แวร์และฮาร์ดแวร์ที่นำมาทวนสอบนั้นมักจะมีลักษณะแบ่งออกได้แตกต่างกันเป็นอย่างน้อยสามประเภท ดังนี้

- ระบบเชิงลำดับ (sequential system)
- ระบบเชิงพร้อมกัน (concurrent system)
- ระบบเชิงปฏิสัมพันธ์ (reactive system)

ระบบเชิงลำดับเป็นระบบที่จะเริ่มทำงานจากจุดเริ่มต้นอย่างต่อเนื่องไปเรื่อย ๆ และสิ้นสุดเมื่อทำงานตามที่กำหนดไว้แล้วเสร็จสิ้น กล่าวคือ มีจุดเริ่มต้นแห่งเดียวและจุดจบของการทำงานหลายแห่งได้ตามแต่กรณีเงื่อนไขที่เกิดขึ้น ซึ่งครอบคลุมลำดับการทำงานทั้งแบบลำดับ แบบเลือกทำ และแบบทำซ้ำเป็นวงวน แต่จะทำงานแบบสายโยงเดี่ยว (single thread) คือ เสมือนทำงานด้วยหน่วยประมวลผลเดียว การติดตามและทดสอบการทำงานของระบบมักไม่ซับซ้อนและสามารถทำได้ด้วยวิธีการทดสอบแบบมาตรฐานทั่วไป ดังนั้นระบบเชิงลำดับจึงมักจะไม่เป็นตัวเลือกที่จะใช้ในการทวนสอบเชิงรูปนัย

ระบบเชิงพร้อมกันเป็นระบบที่ทำงานโดยมีจำนวนสายโยงใยมากกว่าหนึ่งสาย และทำงานไปพร้อมกันได้ (multiple and concurrent threads) ถ้ามองแบบง่าย ๆ คือ การมีสายโยงใยหลาย ๆ สายทำงานพร้อมกันนั่นเอง โดยแต่ละสายโยงใยก็เข้าใจว่า ตนเองทำงานอยู่เพียงโดดเดี่ยวเท่านั้น และมีทรัพยากรของเครื่องคอมพิวเตอร์เป็นของตนเองทั้งหมดไม่ต้องแบ่งกันใช้ แต่อันที่จริงแล้วสิ่งที่ปรากฏจริง คือ แต่ละสายโยงใยจะต้องแบ่งทรัพยากรชุดเดียวกันเพื่อใช้ทำงาน มีการจัดลำดับการเข้าถึงเพื่อใช้หน่วยประมวลผลกลางตัวเดียวกัน ตลอดจนการเข้าถึงเพื่ออ่านและเขียนหน่วยความจำตำแหน่งเดียวกันได้ ทำให้อาจจะเกิดการชนกันหรือการขัดแย้งกันได้บ้าง และสาเหตุหลักที่ทำให้เราไม่เลือกทำการทวนสอบระบบชนิดนี้โดยใช้แบบอรูปนัยเดิม คือ เมื่อเกิดเหตุการณ์การแทรกสลับของการเข้าถึงหน่วยประมวลผลกลาง (interleaving) ในระหว่างการทำงานของแต่ละสายโยงใย ซึ่งการแทรกสลับจะส่งผลให้เกิดความซับซ้อนของเหตุการณ์มากมาย และเมื่อพิจารณาสถานะของระบบเชิงพร้อมกันในองค์รวมที่เป็นไปได้ทั้งหมดแล้วจะพบว่า เกิดการเปลี่ยนสถานะเชิงไม่กำหนด (nondeterministic transitions) ขึ้นในระบบชนิดนี้ ซึ่งส่งผลให้เกิดจำนวนสถานะใหม่เพิ่มมากมายเกินกว่าที่จะใช้ดุลยพินิจของผู้เชี่ยวชาญได้ ระบบประเภทนี้จึงต้องใช้การทวนสอบเชิงรูปนัย

ระบบเชิงปฏิสัมพันธ์เป็นระบบที่ทำงานโดยการเฝ้าสังเกตสภาพแวดล้อมที่กำหนดไว้ให้สนใจ และทำการตอบสนองต่อการเปลี่ยนแปลงของข้อมูลนำเข้าจากสภาพแวดล้อมดังกล่าว โดยที่การนำเข้าข้อมูลอาจจะมีลักษณะแบบอสมวาร (asynchronous stream input) ที่ไม่ต้องทำตามจังหวะของสัญญาณนาฬิกา หรือเป็นแบบสมวารก็ได้ ระบบจะต้องมีการตอบสนองได้ดี (responsive) มีความยืดหยุ่น (resilience) เราอาจจัดให้ระบบเชิงเวลาจริง (real-time system) เป็นระบบเชิงปฏิสัมพันธ์ที่มีพฤติกรรมในการตอบสนองในช่วงเวลาที่กำหนดไว้อย่างชัดเจน และเน้นเรื่องความปลอดภัยของการทำงานตามเงื่อนไขด้านเวลา เราควรใช้การทวนสอบเชิงรูปนัยสำหรับระบบประเภทนี้เป็นอย่างยิ่ง

1.8 เครื่องมือทวนสอบเชิงรูปนัย

เครื่องมือทวนสอบเชิงรูปนัยที่หาใช้ได้ง่าย มักจะเป็นเครื่องมือที่สามารถบรรจุลงมาติดตั้งได้โดยไม่มีค่าใช้จ่าย ส่วนใหญ่เป็นจะผลงานจากห้องปฏิบัติการวิจัยด้านการทวนสอบเชิงรูปนัยของมหาวิทยาลัยและสถาบันวิจัยต่าง ๆ สำหรับบทนี้ขอแนะนำเครื่องมือทวนสอบที่หาได้ง่ายดังต่อไปนี้

เครื่องมือสปิน (SPIN) [7] เป็นเครื่องมือทวนสอบเชิงรูปนัยที่ออกแบบเพื่อใช้ทวนสอบระบบซอฟต์แวร์ที่ทำงานด้วยหลายสายโยงใยพร้อมกันได้ (เครื่องมือสปินไม่ได้ถูกออกแบบมาเพื่อใช้ทวนสอบระบบฮาร์ดแวร์ตั้งแต่แรก เนื่องจากไม่ได้สนับสนุนการจัดการสัญญาณนาฬิกา) โดยสามารถบรรจุลงมาติดตั้งแบบไม่เสียค่าใช้จ่าย พัฒนาแรกเริ่มในช่วงปี ค.ศ. 1980 ที่ห้องปฏิบัติการเบลล์และปรับปรุงต่อยอดเพื่อให้มีขีดความสามารถเพิ่มขึ้นอย่างต่อเนื่อง จนปี ค.ศ. 2002 เครื่องมือสปินได้รับรางวัล *“ACM System Software Award”* มีผู้พัฒนาอื่นเข้ามาร่วมพัฒนาส่วนต่อประสานเพื่อให้เครื่องมือสปินใช้งานได้ง่ายขึ้น เช่น ส่วนต่อประสาน *iSPIN*, *jSPIN* เป็นต้น การใช้เครื่องมือสปินต้องเขียนแบบจำลองเชิงรูปนัยด้วยภาษาโพรเมลาเท่านั้น ซึ่งได้รับการออกแบบมาเพื่อสนับสนุนการทำงานพร้อมกันของหลายสายโยงใยได้ดี สนับสนุนการทำงานที่เกิดการแทรกสลับได้ดี รายละเอียดเพิ่มเติมติดตามได้ที่ *www.spinroot.com*

เครื่องมือเซดอีฟ (Z/EVE) [8] เป็นโปรแกรมที่ใช้เป็นเครื่องมือทวนสอบด้วยวิธีพิสูจน์ทฤษฎีบท ที่ได้รับการพัฒนาขึ้นโดยองค์กร *ORA* ประเทศแคนาดา โปรแกรมนี้สนับสนุนการนำเข้าแฟ้มข้อกำหนดเชิงรูปนัยภาษาเซดด้วย เนื่องจากการเขียนสัญกรณ์ในภาษาเซดต้องเขียนในรูปแบบมาตรฐานลาเท็กซ์ (LaTeX) ซึ่งค่อนข้างยุ่งยาก เครื่องมือเซดอีฟจึงรับข้อกำหนดภาษาเซดได้ทั้งแบบโหมดเชิงตอบโต้ (interactive mode) และแบบโหมดงานแบบกลุ่ม (batch mode) โดยมีความสามารถในการตรวจสอบวายกสัมพันธ์ของข้อกำหนด และตรวจสอบความ

ต้องกันของข้อกำหนดทั้งหมดที่บันทึกเข้ามาได้ นอกจากนี้ผู้ทวนสอบยังสามารถกำหนดเพิ่มข้อสมมุติฐานใหม่ หรือทฤษฎีใหม่เพื่อกำหนดการเปลี่ยนทิศทางของการอนุมานระหว่างการพิสูจน์ผ่านทางหน้าจอได้เช่นกัน รายละเอียดติดตามได้ที่ *www.oracanada.com/z-eves/welcome.html*

เครื่องมือโรดิน (Rodin) [9] พัฒนามาจากโปรแกรม *Eclipse* เพื่อรองรับการเขียนแบบจำลองเชิงรูปนัยด้วยภาษาอีเวนท์บี และเป็นเครื่องมือแกนในการเสริมต่อโปรแกรมเสริม (plug-in program) ที่มีเพิ่มเติมจากผู้พัฒนาร่วมที่ส่งมาได้ เช่น โปรแกรมเสริมกลุ่มที่ใช้ทำภาพเคลื่อนไหว (animation) โปรแกรมเสริมกลุ่มที่ใช้ช่วยนำเสนอให้ง่ายต่อการมองเห็น (visualization) โปรแกรมเสริมกลุ่มที่ใช้ในการทำเอกสาร (documentation) โปรแกรมเสริมกลุ่มที่ใช้ในการพิสูจน์ทฤษฎีบท (theory prover) เป็นต้น รายละเอียดเพิ่มเติมติดตามได้ที่ *www.event-b.org*

เครื่องมือไปป์ (PIPE) [10] พัฒนาเป็นเครื่องมือสนับสนุนการออกแบบและวิเคราะห์เพทริเน็ตแบบสโทแคสติกทั่วไป (generalized stochastic petri nets) โดยมีส่วนต่อประสานผู้ใช้ที่ใช้งานได้สะดวกและง่าย มีส่วนช่วยในการวาดแผนภาพเพทริเน็ตและกำหนดค่ากำกับที่สำคัญต่าง ๆ ทั้งยังใช้จำลองการทำงานเพทริเน็ตได้ด้วย และมีการนำเสนอภาพเคลื่อนไหวช่วยการมองเห็นทีละจังหวะของการเคลื่อนไหวของโทเคนที่มีในแต่ละเพลส ซึ่งเป็นขั้นตอนเพื่อให้เข้าใจได้ง่าย รายละเอียดเพิ่มเติมติดตามได้ที่ *pipe2.sourceforge.net/index.html*

เครื่องมือซีพีเอ็น (CPN tool) [11] พัฒนาขึ้นเป็นเครื่องมือสนับสนุนการวาดภาพและออกแบบคัลเลอร์เพทริเน็ต ซึ่งเป็นเพทริเน็ตประเภทหนึ่งที่ได้รับการขยายขีดความสามารถในการกำหนดให้โทเคนมีชนิดและจัดเก็บค่าไว้ในตัวเองได้ เครื่องมือนี้ยังสนับสนุนการวิเคราะห์ และทวนสอบคัลเลอร์เพทริเน็ตได้โดยตรง โดยจะทำการสร้างปริภูมิสถานะด้วยตนเองโดยอัตโนมัติและจำลองการทำงานหรือทวนสอบได้ รายละเอียดเพิ่มเติมติดตามได้ที่ *cpntools.org*

1.9 แบบฝึกหัด

1. การกำหนดกรอบของกระบวนการพัฒนาระบบซอฟต์แวร์และฮาร์ดแวร์ไว้จะมีประโยชน์อย่างไรสำหรับผู้พัฒนาระบบ?
2. ยกตัวอย่างลำดับของกิจกรรมในกระบวนการพัฒนาระบบทั่วไป?
3. การทวนสอบเป็นกิจกรรมส่วนใดของกระบวนการพัฒนาระบบ?
4. การทวนสอบทำได้กี่แบบ อย่างไรบ้าง ยกตัวอย่างประกอบ?
5. เมื่อเราเลือกทำการทวนสอบระบบแล้ว เราจำเป็นต้องทำการทดสอบความถูกต้องของระบบเมื่อได้โปรแกรมหรือวงจรเสร็จออกมาอีกครั้งหรือไม่ เพราะเหตุใด?

6. การทวนสอบทั่วไปมักจะทำกันอย่างไร?
7. การนำวิธีเชิงรูปนัยมาใช้ในการทวนสอบมีข้อดีและข้อเสียอย่างไร?
8. ผู้การทวนสอบเชิงรูปนัยจะต้องเตรียมการอย่างไร?
9. ข้อกำหนดเชิงรูปนัยต่างจากข้อกำหนดเชิงอรูปนัยอย่างไร?
10. แบบจำลองการออกแบบเชิงรูปนัยคืออะไร ประกอบด้วยองค์ประกอบใดบ้าง และยกตัวอย่างประกอบ?
11. การทวนสอบเชิงรูปนัยของระบบใช้เทคนิคใดได้บ้าง จงอธิบายแต่ละเทคนิคโดยสังเขป?
12. ระบบซอฟต์แวร์และฮาร์ดแวร์ประเภทใดที่มักไม่จำเป็นต้องใช้การทวนสอบเชิงรูปนัย จงอธิบายเหตุผล?
13. เราควรจะทวนสอบคุณลักษณะใดของระบบ?
14. ระบบซอฟต์แวร์ที่มีปัญหาเรื่องการแทรกสลับควรได้รับการทวนสอบอย่างไร?
15. ระบบที่มีหลายสายโยงใยทำงานพร้อมกัน มักจะเกิดปัญหาใด?
16. จงยกตัวอย่างเครื่องมือทวนสอบมาสัก 2 โปรแกรม?

2

ความรู้พื้นฐานด้านคณิตตรรกศาสตร์

2.1 ความสำคัญของบทนี้

เนื้อหาบทนี้ทบทวนความรู้พื้นฐานด้านคณิตตรรกศาสตร์ โดยเริ่มต้นด้วยเรื่องราวความเป็นมาของคณิตตรรกศาสตร์ นิยามของวากยกสัมพันธ์และความหมายของภาษาที่ใช้ระบบคณิตตรรกศาสตร์ ตรรกศาสตร์เชิงประพจน์ การพิสูจน์ทางตรรกศาสตร์ การพิสูจน์ข้อยุติด้วยการใช้ตารางค่าความจริงและด้วยกฎการอนุมานแบบนิรนัย ทฤษฎีเซต ความสัมพันธ์ของเซต และอื่น ๆ ท่านใดที่มีความรู้พื้นฐานดังกล่าวแล้วสามารถข้ามบทนี้ไปได้โดยไม่ต้องเสียเวลาได้

2.2 วัตถุประสงค์

- เพื่อทบทวนความเป็นมาและความสำคัญของคณิตตรรกศาสตร์
- เพื่อทบทวนระบบตรรกศาสตร์เชิงประพจน์
- เพื่อทบทวนวิธีการพิสูจน์ทางคณิตตรรกศาสตร์
- เพื่อให้เข้าใจสัญลักษณ์ที่จำเป็นในการเขียนข้อกำหนดเชิงรูปนัย

2.3 ความเป็นมาของคณิตตรรกศาสตร์

คำภาษาอังกฤษว่า *“Mathematics”* มีรากศัพท์มาจากภาษากรีกโบราณที่แปลว่า *“การเรียนรู้”* ดังนั้นคณิตศาสตร์ คือ ศาสตร์ของการศึกษาเรียนรู้สิ่งต่างๆ โดยเฉพาะอย่างยิ่งเป็นการเรียนรู้เรื่องราวในแง่มุมต่าง ๆ ของสิ่งที่อยู่รอบตัวเราหรือธรรมชาติที่เราสนใจ คือ ปริมาณ (quantity) โครงสร้าง (structure) ปริภูมิ (space) และการเปลี่ยนแปลง (change) นักคณิตศาสตร์ที่สนใจเรื่องของระบบก็มักจะสนใจเรื่องราวของปริมาณของระบบ โครงสร้างของระบบ ปริภูมิของระบบ และการเปลี่ยนแปลงของระบบ เป็นต้น

จากความสนใจและเรียนรู้เกี่ยวกับสิ่งรอบตัวได้ดี นักคณิตศาสตร์มักจะออกแบบ และสร้างเครื่องมือที่มีสัญลักษณ์ทางคณิตศาสตร์ที่นำมาเป็นตัวแทนเชิงนามธรรม เราเรียกว่าเป็นภาษาเชิงคณิตศาสตร์ (mathematical language) นักคณิตศาสตร์มักใช้ภาษาเชิงคณิตศาสตร์นำเสนอแบบจำลองที่เป็นนามธรรมและทำการค้นพบสิ่งใหม่ทางคณิตศาสตร์ของแบบจำลองที่ได้อย่างต่อเนื่องจน

สามารถสรุปผลเพื่อนำไปประยุกต์ใช้งานได้จริงในศาสตร์อื่น เช่น เศรษฐศาสตร์ วิศวกรรมศาสตร์ เป็นต้น กล่าวคือศาสตร์อื่นจะใช้ประโยชน์จากเทคนิคการเรียนรู้แบบคณิตศาสตร์นั่นเอง

โดยทั่วไปนักคณิตศาสตร์พยายามสร้างเครื่องมือทางคณิตศาสตร์หรือออกแบบเทคนิคทางคณิตศาสตร์เพื่อให้นักวิทยาศาสตร์ นักคอมพิวเตอร์ หรือนักวิชาการอื่นนำเครื่องมือหรือเทคนิคทางคณิตศาสตร์ไปใช้ประโยชน์ โดยนำไปแก้ปัญหาที่มีในโดเมนของการประยุกต์ใช้งาน (application domain) ที่แตกต่างกัน และนักคณิตศาสตร์จะต้องเรียนรู้เพื่อที่การค้นพบและพิสูจน์ความจริงทางคณิตศาสตร์ (mathematical facts) ใหม่ เพื่อเป็นต้นทุนหรือต้นทางในการค้นพบความจริงใหม่ ๆ ต่อไป

คณิตศาสตร์ในยุคแรกเริ่ม ยุคซึ่งที่มนุษย์เริ่มรวมกันเป็นสังคมโดยมีการตั้งถิ่นฐานและสร้างอาณานิคม ช่วงเวลานั้นคณิตศาสตร์ถูกนำมาจะใช้ประโยชน์เพื่อทำความเข้าใจรูปแบบของธรรมชาติ เช่น การนับวันและเวลาโดยการสังเกตการเคลื่อนที่ของดวงดาวบนท้องฟ้าในด้านดาราศาสตร์ การสังเกตและนับวันที่มีปรากฏการณ์น้ำขึ้นน้ำลงในทะเล เพื่อประโยชน์ด้านการเกษตรกรรมและศาสนาเป็นสำคัญ ต่อมาการนับและคำนวณพื้นที่และปริมาณผลผลิตก็เพื่อใช้ในระบบการเก็บส่วยและภาษีเพื่อให้เกิดความเป็นธรรมในสังคม เป็นต้น [13] มนุษย์มีการเรียนรู้เกี่ยวกับระบบตัวเลข การนับ การวัด เพื่อหาน้ำหนักและปริมาณ และมีบันทึกอ้างอิงได้ในยุคสุมิเรียน (Sumerian) บาบิโลน (Babylonian) ตั้งแต่แถบเมโสโปเตเมียและข้ามไปแถบลุ่มแม่น้ำไนล์ยุคอียิปต์โบราณ ซึ่งมีอารยธรรมรุ่งเรืองยุคแรกเริ่มของโลก มีการค้นพบหลักฐานจากบันทึกบนกระดาษ ปาปิรุส (papyrus) ว่าสามารถแก้สมการพีชคณิตกำลังสองได้แล้ว มีการบันทึกการสังเกตแบบรูปของช่วงระยะทางจันทรคติ (patterns of lunar phases) ในการกำหนดฤดูเพื่อการเกษตรกรรมและศาสนาเป็นสำคัญ มีการสำรวจพื้นที่และมีการวัดโดยใช้ด้านกว้างของฝ่ามือ หรือข้อศอกถึงปลายนิ้ว เริ่มมีระบบตัวเลขแบบฐานสิบ (decimal numeric system) จากการที่ใช้สิบนิ้วในการนับจำนวน มีหลักหน่วย หลักสิบ หลักร้อย หลักพัน จนกระทั่งหลักล้าน โดยใช้สัญลักษณ์ต่าง ๆ เช่น สัญลักษณ์ขีดแทนหลักหน่วย สัญลักษณ์กระดูกส้นเท้า (heel-bone) แทนหลักสิบ สัญลักษณ์ขดมัดเชือก (coil of rope) แทนหลักร้อย และสัญลักษณ์ต้นบัว (lotus plant) แทนหลักพัน เป็นต้น

คณิตศาสตร์ได้มีการวิวัฒนาการอย่างต่อเนื่องมาเรื่อย ๆ จนครอบคลุมเรื่องของเรขาคณิต พีชคณิตและอื่น ๆ เพื่อใช้ในการคำนวณและออกแบบเพื่อสร้างสถาปัตยกรรมของสิ่งปลูกสร้างที่ยิ่งใหญ่ เช่น พีระมิด วิหารต่าง ๆ และเครื่องมือที่มีชิ้นส่วนและการทำงานที่ซับซ้อน เป็นต้น ดังนั้นคณิตศาสตร์นับได้ว่า

เป็นประโยชน์ต่อโลกมาอย่างต่อเนื่อง คณิตศาสตร์ได้รับการเผยแพร่และพัฒนาในแต่ละยุคสมัยต่อมา

ในขณะเดียวกันอีกด้านหนึ่งเป็นเรื่องราวเกี่ยวกับตรรกศาสตร์ ซึ่งมีผลงานเป็นที่ยอมรับกันอย่างจริงจังและเป็นทางการในยุคกรีกโบราณ (ancient greek) ในช่วงเวลานั้น เมื่อมีข้อโต้แย้งเกิดขึ้นผู้คนไม่สามารถสรุปผลได้ ทำให้มีนักคิดนักปราชญ์ชาวกรีกโบราณในยุคนั้นพยายามคิดค้นหากระบวนการคิด (thinking process) อย่างมีระบบ

เป็นที่กล่าวถึงว่า เริ่มจากกระบวนการตั้งคำถามแบบโสเครติส (Socrates questioning) ซึ่งเชื่อว่าการฝึกตั้งคำถามอย่างรอบคอบและอย่างเป็นระบบจะช่วยให้เราสามารถตรวจสอบความคิดที่มีอยู่ว่าถูกต้องหรือไม่ ในยุคโสเครติสเชื่อว่าการเรียนรู้เกิดจากการพูดคุยเพื่อถามและตอบในเรื่องราวต่าง ๆ กับผู้คนต่าง ๆ นั่นเอง สืบเนื่องต่อมาในยุคเพลโตซึ่งเป็นศิษย์โสเครติส ได้คิดค้นต่อเนื่องแต่เน้นไปทางผลงานด้านปรัชญาและการเมือง จนในที่สุด มาถึงสมัยอริสโตเติล (384-322 B.C.) ซึ่งเป็นศิษย์ของเพลโตและหลานศิษย์ของโสเครติส ได้รับการถ่ายทอดแนวคิดในแนวทางนี้มาตลอดเกี่ยวกับการค้นหากระบวนการคิดอย่างมีระบบ ซึ่งที่สุดแล้ว อริสโตเติลก็สามารถทำได้โดยได้รับการขนานนามว่าเป็นบิดาแห่งตรรกศาสตร์ที่มีแบบแผนและเป็นระบบ โดยได้นำเสนอตรรกบท (syllogism) ที่สำคัญและเป็นตัวอย่างที่ได้รับการกล่าวถึงมากที่สุดในเรื่องการอนุมานแบบนิรนัย (deductive inference) ไว้ว่า

Premise1: All men are mortal
Premise2: Socratis is a man
Conclusion: Therefore, Socratis is mortal

ตรรกบท คือ ข้ออ้างเหตุผลทางตรรก (logical argument) ที่ประกอบด้วยชุดข้อหลักฐาน (premises) ที่มีค่าความจริงเป็นจริงเป็นจุดตั้งต้น และหาเหตุผลแบบนิรนัยจนมีข้อยุติที่เป็นจริงในที่สุด ตรรกศาสตร์ในยุคกรีกโบราณจะเน้นเรื่องตรรกบทที่ใช้ในแนวคิดทางปรัชญา กล่าวคือ การใช้ถ้อยคำหรือประโยคในการอ้างเหตุผลเพื่ออนุมานหาข้อยุติทางแนวคิดทางปรัชญา ตรรกศาสตร์ดั้งเดิมนี้ได้รับการถ่ายทอดและพัฒนาด้านปรัชญาต่อมาในส่วนอื่นของโลกด้วย เช่น ในประเทศอินเดีย จีน และในกลุ่มชาวคริสเตียนและชาวอิสลาม เป็นต้น

เราพบว่ามียุคซบเซาของตรรกศาสตร์ในช่วงศตวรรษที่ 14 ถึงช่วงต้นศตวรรษที่ 19 ซึ่งตรรกศาสตร์ดั้งเดิมไม่ได้รับความสนใจและหมดความนิยมไป ครั้นมาถึงกลางศตวรรษที่ 19 เดิมทีตรรกศาสตร์ไม่ได้ถูกนับรวมว่าเป็นส่วนหนึ่งหรือสาขาหนึ่งของคณิตศาสตร์ เนื่องจากตรรกศาสตร์ดั้งเดิมในระยะแรกช่วงก่อนศตวรรษที่ 19 นั้นเน้นการจัดการเกี่ยวกับประโยคและไม่มีการใช้สัญลักษณ์ หรือ

ตัวแปรมาเกี่ยวข้อง ทำให้ไม่ได้รับการยอมรับให้จัดเป็นส่วนหนึ่งคณิตศาสตร์ได้เลยตั้งแต่ยุคอริสโตเติลเป็นต้น จนเกิดเหตุการณ์การจุดประกายในศตวรรษที่ 17

ในศตวรรษที่ 17 มีนักคณิตศาสตร์ชื่อ *Gottfried Leibniz* ผู้ซึ่งได้รับการยกย่องว่าเป็นผู้คิดค้นหลักการทำแคลคูลัส (calculus) การสร้างสมการฟังก์ชัน การทำแคลคูลัสเชิงอนุพันธ์ (differential calculus) และการทำแคลคูลัสเชิงปริพันธ์ (integral calculus) (คิดค้นได้พร้อมๆ กับ *Sir Isaac Newton*) ช่วงหนึ่งของผลงานของ *Gottfried Leibniz* ได้เคยนำเสนอว่าตรรกศาสตร์ควรจะได้รับการควบรวมเป็นส่วนหนึ่งของคณิตศาสตร์ เนื่องจากตรรกศาสตร์น่าจะมีส่วนช่วยในการพิสูจน์ทางคณิตศาสตร์ได้ และได้พยายามพัฒนาแคลคูลัสเชิงตรรกะ (logical calculus) ในระยะแรกไว้ให้ แต่ก็ไม่ได้รับการยอมรับ ด้วยเหตุนี้เราน่าจะเรียกช่วงศตวรรษที่ 17 ว่าเป็นระยะฟักตัว (embryonic period) ของตรรกศาสตร์ยุคใหม่ (modern logic) ถัดจากตรรกศาสตร์ยุคโบราณได้

ต่อมา เกิดการปรับปรุงครั้งใหญ่ของกระบวนการพิสูจน์ทางคณิตศาสตร์ในช่วงกลางศตวรรษที่ 19 ได้มีการนำเสนอให้ใช้การพิสูจน์ทางตรรกศาสตร์มารวมเป็นทางเลือก และปรับปรุงระบบตรรกศาสตร์ดั้งเดิมให้มีการใช้สัญลักษณ์และตัวแปรโดยให้เรียกเสียใหม่ว่า ตรรกศาสตร์เชิงสัญลักษณ์ (symbolic logic) หรือเรียกกันว่า คณิตตรรกศาสตร์ (mathematical logic) นั่นเองและเป็นการยอมรับให้ตรรกศาสตร์เป็นส่วนหนึ่งของคณิตศาสตร์ในที่สุด นักคณิตศาสตร์ส่วนหนึ่งที่มีบทบาทสำคัญที่ทำให้คณิตตรรกศาสตร์เป็นที่นิยม และนำมาใช้งานต่อมา คือ ในช่วงศตวรรษที่ 19 มี George Boole ผู้คิดค้นพีชคณิตบูลีน (boolean algebra), *Gottlob Frege* และในช่วงศตวรรษที่ 20 *Bertrand Russell*, *David Hilbert* และ *Giuseppe Peano* เป็นต้น

ตรรกศาสตร์เชิงสัญลักษณ์เป็นการนำตรรกศาสตร์ที่แต่เดิมใช้รูปประโยคที่เป็นเนื้อหาที่ตีความได้เลยมาเพิ่มสัญลักษณ์และตัวแปร ดังนั้น ประโยคในตรรกศาสตร์เชิงสัญลักษณ์จะมีสัญลักษณ์และตัวแปรที่ต้องได้รับการแทนค่าก่อนตีความเสมอ จึงเป็นเหตุทำให้ประโยคในตรรกศาสตร์ชนิดนี้มีลักษณะเป็นนิพจน์ทางตรรกศาสตร์เมื่อเทียบเคียงกับนิพจน์ในพีชคณิตที่มีตัวแปรที่ต้องได้รับการแทนค่าด้วยเช่นกัน ตัวอย่างของสัญลักษณ์ที่ใช้ในตรรกศาสตร์เชิงสัญลักษณ์ คือ เครื่องหมายที่ใช้เป็นตัวดำเนินการทางตรรก เช่น เครื่องหมาย $\neg$, เครื่องหมาย $\wedge$, เครื่องหมาย $\vee$, เครื่องหมาย $\Rightarrow$, เครื่องหมาย $\Leftrightarrow$ เป็นต้น หรือเครื่องหมายที่ใช้เป็น ตัวบ่งปริมาณ (quantifier) เช่น เครื่องหมาย $\forall$, เครื่องหมาย $\exists$ เป็นต้น ส่วนตัวอย่างของตัวแปรที่ใช้ คือ ตัวแปรที่เป็นตัวถูกดำเนินการ (operand) x, y, z เป็นต้น ตัวอย่างประโยคหรือนิพจน์ทางตรรกะแบบง่าย คือ “$x \wedge y \Rightarrow z$” ตรรกศาสตร์เชิงสัญลักษณ์เป็นชื่อเรียกกลุ่มระบบตรรกศาสตร์ที่เรารู้จัก จะรวมถึง

ตรรกศาสตร์เชิงประพจน์ ตรรกศาสตร์ลำดับที่หนึ่ง ตรรกศาสตร์ลำดับสูงขึ้น และตรรกศาสตร์อัญรูป (modal logic) และอื่น ๆ

2.4 การพิสูจน์ทางตรรกศาสตร์

การพิสูจน์ทางตรรกศาสตร์คืออะไร และมีประโยชน์อย่างไร กล่าวได้ว่า การพิสูจน์ คือ สายโซ่ของการอนุมานทางตรรกศาสตร์ (chain of logical inference) ซึ่งมีข้อสมมุติฐานพื้นฐานที่กำหนดให้เบื้องต้น คือ ข้อเท็จจริง (axiom) และทฤษฎีบท (theorem) ที่มีอยู่และเป็นจริงอยู่แล้ว เป็นจุดเริ่มต้นของการอนุมานอย่างต่อเนื่องอย่างเหมาะสมจนสามารถลงความเห็นเฉพาะ (deduce) เพื่อให้ได้ประโยคตรรกศาสตร์ (logical statements) ที่เป็นข้อยุติใหม่ (conclusion) หรือเพื่อยืนยันว่าพันธกรณีที่ต้องการพิสูจน์ (proof obligation) ที่ตั้งไว้เป็นจริงเสมอ [14]

ปกติแล้วระบบตรรกศาสตร์จะมีสององค์ประกอบ คือ ส่วนทฤษฎีการจำลองแบบ (model theory) และส่วนทฤษฎีการพิสูจน์ (proof theory) ทฤษฎีการจำลองแบบประกอบด้วยวากยสัมพันธ์และความหมายในการสร้างแบบจำลองตามกรรมวิธีของระบบตรรกศาสตร์นั้น ๆ โดยจะมีการนิยามวากยสัมพันธ์และความหมายที่จัดดีแล้ว (well-formed) ไว้และไม่กำกวม เช่น การสร้างแบบจำลองของระบบสัญญาณไฟจราจร ด้วยระบบตรรกศาสตร์เชิงประพจน์ ตัวแบบจำลองจะต้องบรรยายพฤติกรรมของการเปิดปิดไฟสัญญาณสีแดง สีเหลืองและสีเขียว ด้วยประโยคประพจน์ตามวากยสัมพันธ์ที่กำหนดไว้ และจะมีการตีความ (interpretation) ตามที่กำหนดไว้ในระบบตรรกศาสตร์เชิงประพจน์เช่นกัน ซึ่งจะกล่าวถึงตรรกศาสตร์เชิงประพจน์นี้ในภายหลัง เป็นต้น

ทฤษฎีการพิสูจน์ [6, 15] ประกอบด้วยข้อเท็จจริงและกฎการอนุมาน (inference rules) หรือเรียกอีกอย่างว่ากฎการพิสูจน์ ข้อเท็จจริงนั้น คือ สิ่งที่มีค่าความจริงเป็นจริงเสมอโดยไม่ต้องพิสูจน์ใด ๆ เป็นจุดเริ่มของการพิสูจน์ ดังนั้นเราต้องระมัดระวังในการระบุว่า มีข้อเท็จจริงอะไรบ้างในระบบของเรา ส่วนกฎการอนุมาน คือ กฎที่ใช้ในกระบวนการสร้างประโยคใหม่จากข้อเท็จจริง ประโยคใหม่ที่ได้จากข้อเท็จจริงจะมีค่าความจริงเป็นจริงด้วยเช่นกัน โดยเราจะต้องเลือกกฎการอนุมานที่เหมาะสมและอธิบายกระบวนการอย่างต่อเนื่องจนได้ประโยคใหม่ดังกล่าว จากนั้นเราจะนำประโยคใหม่มารวมกับข้อเท็จจริงแล้วพิสูจน์ต่อเพื่อหาประโยคใหม่ถัดไปอย่างต่อเนื่อง จนพบประโยคที่เป็นข้อยุติที่ต้องการ และเราจะเรียกข้อยุติดังกล่าวว่า ทฤษฎีบท โดยที่ทฤษฎีบทจะมีค่าความจริงเป็นจริงเสมอเช่นกัน เมื่อนำมารวมกับข้อเท็จจริงที่มีอยู่เราสามารถพิสูจน์หาทฤษฎีบทอื่น ๆ ได้ต่อไป

ประโยชน์หลักของการพิสูจน์ทางตรรกศาสตร์ คือ การสรุปว่าระบบซอฟต์แวร์หรือฮาร์ดแวร์ที่เราออกแบบนั้นมีพฤติกรรมตามที่เรากำหนดไว้หรือไม่ โดยไม่ต้องลงมือทำการจำลองพฤติกรรม เช่น เราต้องการพิสูจน์ว่าระบบสัญญาณไฟจราจรที่ออกแบบจะต้องไม่เปิดไฟสัญญาณสีแดงและสีเขียวพร้อมกันอย่างเด็ดขาด เราจะระบุพันธกรณีที่ต้องการพิสูจน์เป็นข้อยุติสุดท้าย และลงมือทำการพิสูจน์ว่าข้อยุตินั้นเป็นจริงหรือไม่ ถ้าผลการพิสูจน์แจ้งว่า ข้อยุติเป็นจริงเสมอ เมื่อเป็นเช่นนั้นก็ถือว่า การออกแบบผ่านการทวนสอบผ่านในพันธกรณีที่ต้องการพิสูจน์ดังกล่าว เป็นต้น เรามีความจำเป็นต้องระบุพันธกรณีที่ต้องพิสูจน์ไว้ให้ครบถ้วนและลงมือทำการพิสูจน์ให้ครบในที่สุด

การพิสูจน์ด้วยกระบวนการอนุมานทำได้สามแบบ [14] ดังนี้

- การอนุมานแบบอุปนัย (inductive inference) เป็นการสรุปหลักเกณฑ์ทั่วไป (generalized statement) ใหม่จากข้อเท็จจริงจำเพาะ (specific facts) เดิมจำนวนหนึ่งที่สังเกตเห็นได้ ผลของการอนุมานจะสรุปได้ว่าหลักเกณฑ์ใหม่นั้นมีโอกาสเป็นค่าจริงได้ (probable) เท่านั้น ไม่สามารถสรุปว่าหลักเกณฑ์ใหม่นี้เป็นจริงแบบสมบูรณ์มีเหตุมีผลได้ (valid) ดังนั้น การใช้วิธีอนุมานแบบอุปนัยอาจผิดพลาดได้ ตัวอย่างของวิธีการอนุมานแบบอุปนัยทำได้โดยการพิสูจน์ว่าหลักเกณฑ์ใหม่นั้นมีค่าจริงเมื่อเวลาแรกเริ่มหรือขั้นตอนแรกเริ่มเสมอ เราเรียกว่าการพิสูจน์กรณีรากฐาน (base case) จากนั้นเราพิสูจน์กรณีขั้นตอนอุปมาน (inductive step) โดยพิสูจน์ว่า ถ้ากำหนดให้หลักเกณฑ์ใหม่มีค่าจริงเมื่อเวลา k แล้วหลักเกณฑ์ใหม่จะมีค่าจริงที่เวลา $k+1$ ด้วยเช่นกัน
- การอนุมานแบบนิรนัย (deductive inference) เป็นการสรุปข้อสรุปจำเพาะจากข้อเท็จจริงที่เป็นหลักเกณฑ์ทั่วไปและข้อเท็จ-จริงจำเพาะ ผลการอนุมานกล่าวได้ว่า ข้อสรุปจำเพาะที่ได้จะเป็นจริงแบบสมบูรณ์มีเหตุมีผลเสมอ และสามารถนำข้อสรุปนี้ไปใช้ต่อเป็นทฤษฎีบทต่อไป ตัวอย่างของวิธีการอนุมานแบบนิรนัยทำได้โดยการกำหนดให้มีข้อเท็จจริงที่เป็นหลักเกณฑ์ทั่วไปอย่างน้อยหนึ่งประโยคพร้อมข้อเท็จจริงจำเพาะจำนวนหนึ่ง จากนั้นหากฎการนิรนัยที่สามารถอธิบายและนำไปสู่ประโยคใหม่ และอนุมานต่อเนื่องไปจนกว่าจะถึงข้อสรุปจำเพาะที่ตั้งเป้าหมายไว้
- การอนุมานแบบตรวจสอบ (abductive inference) เป็นการพยายามหาข้อสรุปจากข้อเท็จจริงเท่าที่มีอยู่ขณะนั้น ดังนั้นผลการสรุปอาจจะไม่จริงก็ได้ในกรณีที่พบว่ามีข้อตั้งเพิ่มเติมขัดแย้ง

ข้อตั้งเดิม โดยทั่วไปเราพบว่ามักจะใช้ในการวินิจฉัยโรคของแพทย์หรือการสืบสวนหาคนร้ายของตำรวจ ซึ่งอาจจะสรุปผิดได้และมีผลแตกต่างถ้าพบข้อตั้งใหม่ เช่น การพบหลักฐานพยานเพิ่มเติมหรือการพบอาการของโรคเพิ่มเติม ที่ทำให้การสรุปก่อนหน้าผิดได้ เป็นต้น

ในการทวนสอบระบบที่สนใจในตำราเล่มนี้ เราจะใช้วิธีการอนุมานแบบนิรนัยเสมอ เนื่องจากความแน่นอนของผลสรุปที่ได้สูงกว่าวิธีอื่นๆ ประโยชน์หลักของการพิสูจน์ทางตรรกศาสตร์ คือ การสรุปว่าระบบซอฟต์แวร์หรือฮาร์ดแวร์ที่เราออกแบบนั้นมีพฤติกรรมตามที่เรากำหนดไว้หรือไม่ โดยไม่ต้องลงมือทำการจำลองพฤติกรรม

2.5 ตรรกศาสตร์เชิงประพจน์

ตรรกศาสตร์เชิงประพจน์ (propositional logic) หรือเรียกอีกชื่อหนึ่งว่าตรรกศาสตร์เชิงประโยค (sentential logic) [14, 15] เนื่องจากระบบตรรกศาสตร์นี้ประกอบด้วยประพจน์และตัวเชื่อมทางตรรกะ (logical connective) ประพจน์คือประโยคที่มีค่าความจริงเป็นจริงหรือเป็นเท็จได้เพียงค่าเดียวเท่านั้น เราสามารถนำประพจน์มากกว่าหนึ่งประพจน์มาเชื่อมต่อกันได้โดยใช้ตัวเชื่อมที่มีอยู่ คือ ตัวเชื่อม *"และ"* ใช้สัญลักษณ์ $\wedge$, ตัวเชื่อม *"หรือ"* ใช้สัญลักษณ์ $\vee$, ตัวเชื่อม *"ไม่"* ใช้สัญลักษณ์ $\neg$, ตัวเชื่อม *"ถ้า…แล้ว"* ใช้สัญลักษณ์ $\Rightarrow$, และ ตัวเชื่อม *"ก็ต่อเมื่อ"* ใช้สัญลักษณ์ $\Leftrightarrow$

โดยทั่วไปเราเรียกประพจน์ที่แบ่งแยกย่อยไม่ได้อีกแล้วว่า ประพจน์เดี่ยว (atomic proposition) และการรวมประพจน์เดี่ยวเข้าด้วยกันโดยใช้ตัวเชื่อมทางตรรกะจะเกิดประพจน์ใหม่ที่เรียกว่า ประพจน์ประกอบ (compound proposition) เช่น *"Peter is a rich man"* $\wedge$ *"It rains today"* ซึ่งประกอบด้วยสองประพจน์เดี่ยวที่เชื่อมกันด้วยสัญลักษณ์ $\wedge$

ตรรกศาสตร์เชิงประพจน์จัดเป็นคณิตตรรกศาสตร์ประเภทหนึ่งที่มีความซับซ้อนน้อยที่สุด นั่นหมายถึงเราสามารถทำความเข้าใจระบบตรรกศาสตร์นี้ได้ง่ายและไม่ซับซ้อน และใช้เป็นพื้นฐานในการเข้าใจระบบตรรกศาสตร์อื่น ๆ ได้ ทุกประพจน์ที่ใช้อธิบายระบบจะได้รับการตีความเป็นค่าจริงหรือค่าเท็จเพียงค่าเดียว กล่าวคือ ประพจน์เดี่ยว *"Peter is a rich man"* จะเป็นค่าจริงหรือเป็นค่าเท็จอย่างใดอย่างหนึ่งเท่านั้น นั่นหมายถึงในระบบที่เราสนใจ *Peter* จะเป็นคนรวยหรือจนอย่างใดอย่างหนึ่ง เราเขียนได้อีกแบบว่า *"Peter is a rich man* $\mapsto$ *True*

หมายถึงประพจน์เดี่ยวนี้ถูกตีความว่าเป็นจริงและตามมาด้วยว่า *Peter* เป็นคนรวยจริง

ตรรกศาสตร์เชิงประพจน์ให้ผู้ใช้กำหนดตัวแปรประพจน์ (propositional variable) เพื่อใช้แทนประพจน์ที่เขียนแบบบรรยายความได้ ทั้งนี้เพื่อให้มีความสะดวกในการนำแต่ละประพจน์ไปเชื่อมต่อกัน เช่น กำหนดให้มีตัวแปรประพจน์ p ใช้แทน *"Peter is a rich man"* และตัวแปรประพจน์ q ใช้แทน *"It rains today"* ผู้ใช้สามารถนำตัวแปรประพจน์ p, q มาใช้แทนเป็น $p \wedge q$ เป็นต้น

ในการสร้างแบบจำลองของระบบที่เราสนใจโดยใช้ตรรกศาสตร์เชิงประพจน์นี้ สิ่งที่ต้องทำก็คือ การกำหนดประพจน์ที่เกี่ยวข้องทั้งหมดให้ครบถ้วน และอาจจะมีการเชื่อมประพจน์เข้าด้วยกันอย่างเหมาะสมเพื่ออธิบายโครงสร้างและพฤติกรรมของระบบเพื่อใช้เป็นข้อเท็จจริงเบื้องต้นที่เราจะใช้ในการพิสูจน์ต่อไป

ตรรกศาสตร์เชิงประพจน์ถือว่าเป็นตรรกศาสตร์อันดับที่ศูนย์ (zeroth-order logic) ซึ่งต่อมาได้รับการขยายขีดความสามารถให้เป็นตรรกศาสตร์อันดับที่หนึ่ง (first-order logic) และปรับขยายต่อไปเป็นอันดับที่สอง และอันดับที่สูงขึ้นเรื่อย โดยเราจะเรียกอันดับที่สูงมากกว่าหนึ่งที่เราเรียกว่าตรรกศาสตร์อันดับสูง (higher order logic)

ตัวอย่างต่อไปนี้แสดงความแตกต่างระหว่างตรรกศาสตร์เชิงประพจน์และตรรกศาสตร์อันดับที่หนึ่ง โดยระบบที่แสดงด้วยตรรกศาสตร์เชิงประพจน์จะต้องแจกแจงด้วยประพจน์ที่เป็นข้อเท็จจริงจำนวนมาก และเมื่อมีข้อเท็จจริงใหม่เกิดขึ้นสิ่งที่ต้องทำ คือ ต้องแจกแจงประโยคเพิ่มให้ครบทุกประโยค ดังนี้

กำหนดให้

Mr.X is father of Mr.Y
Mr.Y is father of Mr.Z
Mr.Z is father of Mr.S

เป็นประพจน์ที่แสดงข้อเท็จจริงเรื่องความสัมพันธ์ *father* ระหว่างบุคคลชื่อ *Mr.X, Mr.Y, Mr.Z, Mr.S* ระบบนี้จะรู้จักความสัมพันธ์ *father* เท่านั้น ถ้ามีคำถามว่าใครเป็น *father* ของ *Mr.Y* เราจะทราบได้ว่า *Mr.X* เป็น *father* ของ *Mr.Y* จากข้อเท็จจริงที่กำหนดให้ กรณีที่ระบบมีข้อเท็จจริงใหม่เรื่องความสัมพันธ์แบบ *grandpa* ก็จะต้องแจกแจงประพจน์เพิ่มเติมเสมอ ดังนี้

Mr.X is grandpa of Mr.Z
Mr.Y is grandpa of Mr.S

เมื่อมีการกำหนดประพจน์เพื่อแสดงความสัมพันธ์ *grandpa* ขึ้นใหม่โดยเป็นความสัมพันธ์ *grandpa* ระหว่าง *Mr.X, Mr.Y, Mr.Z, Mr.S* จะต้องมีการแจก

แจงประพจน์เพิ่มให้ครบ ระบบจึงจะทำงานอย่างถูกต้องครบถ้วนได้ในที่สุด ซึ่งจะเป็นภาระมากในการบำรุงรักษาระบบนี้

ในขณะที่ถ้าเราใช้ตรรกศาสตร์อันดับที่หนึ่งซึ่งจะมีส่วนที่เรียกว่าเพรดิเคต (predicate) ทำหน้าที่คล้ายฟังก์ชันที่รับข้อมูลนำเข้าและประมวลผลทางตรรกะ เพรดิเคตถูกเพิ่มเข้ามาเพื่อช่วยในการเปลี่ยนแปร (derive) ข้อเท็จจริงเดิมเพื่อหาข้อเท็จจริงใหม่ได้โดยไม่ต้องกำหนดแบบแจกแจงให้เพิ่ม ดังนี้

กำหนดให้

Mr.X is father of Mr.Y
Mr.Y is father of Mr.Z
Mr.Z is father of Mr.S
grandpa(a, b) :
a is father of c $\wedge$
c is father of b.

เราพบว่าตรรกศาสตร์อันดับที่หนึ่งมีข้อดีและสะดวกมากกว่าตรรกศาสตร์เชิงประพจน์ โดยมีเพรดิเคตที่สามารถนำมาใช้ในการเปลี่ยนแปรหาข้อเท็จจริงใหม่ที่ไม่ได้ระบุไว้ ในที่นี้ความสัมพันธ์ *grandpa* จะถูกกำหนดให้เป็นเพรดิเคตที่นิยามมาจากความจริงที่ว่า *a* จะเป็น *grandpa* ของ *b* ได้ก็ต่อเมื่อ *a* เป็น *father* ของ *c* ใด ๆ ที่เป็น *father* ของ *b* จากการประมวลผลของเพรดิเคต นี้เราพบว่า *grandpa(Mr.X, Mr.Z)* จะให้ผลลัพธ์เป็นค่าจริง นั่นหมายถึงว่าเพรดิเคตช่วยสนับสนุนข้อเท็จจริงใหม่ว่า *Mr.X* เป็น *grandpa* ของ *Mr.Z* นั่นเอง โดยที่ไม่ต้องทำการแจกแจงประพจน์เพิ่มเติมเข้าไป นอกจากนี้ตรรกศาสตร์อันดับที่หนึ่งยังให้ผู้ใช้กำหนดตัวบ่งปริมาณเพื่อใช้อธิบายขอบเขตเชิงปริมาณของจำนวนสมาชิกในเซตได้

ตามที่ได้กล่าวไว้แล้วว่า ตรรกศาสตร์ประกอบด้วยส่วนทฤษฎีการจำลองแบบและทฤษฎีการพิสูจน์ สำหรับตรรกศาสตร์เชิงประพจน์ก็เช่นกัน ส่วนทฤษฎีการจำลองแบบของตรรกศาสตร์เชิงประพจน์ก็แบ่งออกเป็นสองส่วนด้วยอีกเช่นกัน คือ ส่วนวากยสัมพันธ์และส่วนความหมาย

นิยามวากยสัมพันธ์และความหมายของตรรกศาสตร์เชิงประพจน์

นิยามที่ 2-1: สัญลักษณ์ที่ใช้ในวากยสัมพันธ์ของตรรกศาสตร์เชิงประพจน์ [6]

ภาษาที่ใช้เขียนประโยคในระบบตรรกศาสตร์นี้รวมถึงการเชื่อมประโยคจะประกอบด้วยสัญลักษณ์ ดังนี้

- สัญลักษณ์จากเซต *PROP* ซึ่งเป็นกลุ่มของตัวแปรประพจน์ที่แต่ละตัวแสดงถึงประพจน์เดี่ยวที่เล็กที่สุดไม่สามารถแบ่งเป็นประโยคย่อยอีกได้ เช่น *p, q, r* เป็นประพจน์ที่เล็กที่สุด เป็นต้น

- สัญลักษณ์จากเซตของตัวเชื่อมประพจน์ที่แต่ละตัวใช้เชื่อม หรือตกแต่งประโยคให้เป็นประพจน์ประกอบ ตัวเชื่อมแบบนาวลารี (nullary connective) คือ $\{T, F\}$ โดยสัญลักษณ์ T ใช้แทนค่าจริง และสัญลักษณ์ F ใช้แทนค่าเท็จ ตัวเชื่อมแบบเอกภาค (unary connective) คือ $\{\neg\}$ โดยสัญลักษณ์ $\neg$ ใช้เป็นการทำนิเสธหรือการหาค่าตรงข้าม ตัวเชื่อมแบบทวิภาค (binary connective) คือ $\{\wedge, \vee, \Rightarrow, \Leftrightarrow\}$ โดยสัญลักษณ์ $\wedge$ ใช้แทน *"และ"*, สัญลักษณ์ $\vee$ ใช้แทน *"หรือ"*, สัญลักษณ์ $\Rightarrow$ ใช้แทน *"ถ้า..แล้ว"*, และสัญลักษณ์ $\Leftrightarrow$ ใช้แทน *"ก็ต่อเมื่อ"*
- สัญลักษณ์ ( ) ใช้แทนวงเล็บเพื่อใช้ในการจัดกลุ่มและลดความกำกวมของประพจน์

นิยามที่ 2-2: สูตรเชิงประพจน์ที่จัดดีแล้วในวากยสัมพันธ์ของตรรกศาสตร์เชิงประพจน์ [6]

วิธีการเขียนประพจน์อย่างถูกต้องจะต้องใช้สัญลักษณ์ที่กำหนดไว้ในนิยาม 2-1 ข้างต้นเท่านั้น และเมื่อมีการเชื่อมประพจน์เข้าด้วยกันจะต้องอยู่ในรูปแบบที่ถูกต้องเหมาะสมเรียกว่า สูตรเชิงประพจน์ที่จัดดีแล้ว (well-formed propositional formula: $WFFp$) ด้วยกฎต่อไปนี้

- ตัวแปรประพจน์ถือว่าเป็น $WFFp$
- ตัวเชื่อมแบบนาวลารีที่เป็นค่า T หรือ F ถือว่าเป็น $WFFp$
- ถ้า A เป็น $WFFp$ แล้ว $\neg A$ ถือว่าเป็น $WFFp$ เช่นกัน
- ถ้า A และ B เป็น $WFFp$ แล้ว $(A \wedge B)$, $(A \vee B)$, $(A \Rightarrow B)$, $(A \Leftrightarrow B)$ ต่างก็ถือว่าเป็น $WFFp$ เช่นกัน

ตัวอย่างที่ 2-1: การเขียนประพจน์แบบ WFFp

กำหนดให้ $PROP = \{p, q, r\}$ ประพจน์ต่อไปนี้ถือว่าเป็น $WFFp$ คือ T, $T \wedge p$, $(p \vee q)$, $(p \wedge q) \Rightarrow$ r และการเขียนต่อไปนี้ไม่เป็น $WFFp$ คือ $True \wedge$, $(p\ q)$, $(\ p\wedge) \Rightarrow r$

นิยามที่ 2-3: ส่วนความหมายของตรรกศาสตร์เชิงประพจน์ [6]

การตีความประพจน์ใดว่ามีค่าความจริงเป็นค่าจริงหรือค่าเท็จ เกิดจากการหาค่า $\mathfrak{T}$ ที่ทำให้ประพจน์นั้นมีค่าจริงและเป็นไปตามเงื่อนไขที่กำหนดไว้ข้างล่างนี้

$\mathfrak{T} \models p$	iff	$\mathfrak{T}_{\text{atomic}}(p)=T$ for $p \in PROP$
$\mathfrak{T} \models \neg A$	iff	not $\mathfrak{T} \models A$
$\mathfrak{T} \models A \wedge B$	iff	$\mathfrak{T} \models A$ and $\mathfrak{T} \models B$
$\mathfrak{T} \models A \vee B$	iff	$\mathfrak{T} \models A$ or $\mathfrak{T} \models B$
$\mathfrak{T} \models A \Rightarrow B$	iff	if $\mathfrak{T} \models A$, then $\mathfrak{T} \models B$
$\mathfrak{T} \models A \Leftrightarrow B$	iff	$\mathfrak{T} \models (A \Rightarrow B)$ and $\mathfrak{T} \models (B \Rightarrow A)$

หลายคนอาจจะสงสัยว่าสัญลักษณ์ $\models$ หรือที่เรียกว่า "entailment relation" หมายถึงอะไร และนำมาใช้งานอย่างไร คำว่า *"entail"* แปลได้ว่า *"นำมาซึ่ง"* ถ้าเราเขียนว่า "A *entails* B" หรือ $A \models B$ ก็จะหมายถึง "*A นำมาซึ่ง B*" หรือ "*A leads to B*" นั่นเอง หรือกล่าวได้อีกอย่างว่า *"ถ้า A เป็นจริงแล้ว B จะเป็นจริงเสมอ"*

ประพจน์เดี่ยวทุกตัวที่เป็นสมาชิกของเซต $PROP$ จะมีค่าความจริงอย่างไร กำหนดไว้ด้วยฟังก์ชัน $\mathfrak{T}_{\text{atomic}}(\ )$ และสมการ $\mathfrak{T} \models p$ จะมีค่าจริงก็ต่อเมื่อเราหาค่า $\mathfrak{T}_{\text{atomic}}(p)$ ได้ว่ามีค่าจริงโดยที่ p คือประพจน์เดี่ยวที่เป็นสมาชิกของเซต $PROP$

กำหนดให้ A เป็นประพจน์แบบ $WFFp$ และ A=p แล้วสมการ $\mathfrak{T} \models A$ จะมีค่าจริงด้วย และการตีความว่าประพจน์ $\neg A$ ว่ามีค่าจริงก็ต่อเมื่อค่าความจริงของ A มีค่าเป็นเท็จนั่นเอง

ถ้ากำหนดให้ A, B เป็นประพจน์แบบ $WFFp$ แล้วสำหรับการตีความว่าประพจน์ $A \wedge B$ มีค่าจริงหรือไม่ต้องดูที่สมการ $\mathfrak{T} \models A \wedge B$ ซึ่งจะมีค่าจริงก็ต่อเมื่อค่าความจริงของ A และค่าความจริงของ B เป็นค่าจริงทั้งคู่ด้วย และในทำนองเดียวกันกับการตีความ $A \vee B$, $A \Rightarrow B$, $A \Leftrightarrow B$ ที่เหลือในนิยาม

เรามาทบทวนและทำความเข้าใจกับการตีความอีกครั้ง การตีความ คือการกำหนดความหมายให้กับสัญลักษณ์ที่พบในระบบตรรกศาสตร์ [6] และโชคดีที่ความหมายในระบบตรรกศาสตร์เชิงประพจน์ที่เราสนใจก็มีค่าเพียงสองค่า คือค่าจริงและค่าเท็จนั่นเอง ดังนั้นในที่นี้การตีความจะเป็นการกำหนดความหมายว่าประพจน์ที่เป็น $WFFp$ นั้นมีค่าจริงหรือค่าเท็จ เราอาจจะมองว่าการตีความเป็นฟังก์ชันโดยรับประพจน์ที่เป็น $WFFp$ เป็นตัวแปรนำเข้าและให้ผลออกมาเป็นค่าจริงหรือค่าเท็จก็ได้เช่นกัน

ต่อจากนี้ไปเราจะสนใจเฉพาะการตีความที่ได้ค่าจริงเท่านั้น ซึ่งเราเรียกการตีความที่ได้ค่าจริงแต่ละครั้งว่า *โมเดล* (model) ของประโยค และต่อจากนี้ไปจะพบการใช้คำว่าโมเดลในเนื้อหาบ่อยมากขึ้น

ตัวอย่างที่ 2-2: การตีความประพจน์ $((p \wedge q) \Rightarrow r)$

กำหนดให้ $PROP = \{p, q, r\}$ และประพจน์ $((p \wedge q) \Rightarrow r)$ เป็น $WFFp$ ซึ่งได้รับการตีความครั้งที่ 1 (แทนด้วย $\mathfrak{T}_1$) ว่าเป็นจริงโดยให้ $p=T,\ q=T,\ r=T$ หรืออาจจะมีการตีความครั้งที่ 2 (แทนด้วย $\mathfrak{T}_2$) ว่าเป็นจริงโดยให้ $p=F,\ q=T,\ r=F$ ก็ได้ หรืออาจจะมีการตีความครั้งอื่น ๆ ว่าเป็นจริงอีกหลายครั้ง จะเห็นได้ว่าเราจะสนใจเฉพาะการตีความที่ให้ค่าเป็นจริงเท่านั้น ส่วนการตีความที่ให้ค่าเป็นเท็จเรามักจะละไว้ไม่สนใจ

เราสามารถแจกแจงการตีความประพจน์ที่ให้ค่าจริงได้ทั้งหมดดังนี้

$\mathfrak{T}_1 \models ((p \wedge q) \Rightarrow r)$ โดยให้ $p=T, q=T, r=T$

$\mathfrak{T}_2 \models ((p \wedge q) \Rightarrow r)$ โดยให้ $p=F, q=T, r=F$

$\mathfrak{T}_3 \models ((p \wedge q) \Rightarrow r)$ โดยให้ $p=T, q=F, r=F$

$\mathfrak{T}_4 \models ((p \wedge q) \Rightarrow r)$ โดยให้ $p=F, q=F, r=T$

เราพบว่า $\mathfrak{T}_1$, $\mathfrak{T}_2$, $\mathfrak{T}_3$, $\mathfrak{T}_4$ เป็นการตีความที่ให้ค่าประพจน์นี้เป็นจริง และเราเรียกการตีความที่เป็นจริงเหล่านี้ว่า โมเดลแทน และถ้าสมาชิกของเซต $PROP$ เป็นค่าที่บ่งบอกถึงพฤติกรรมของระบบแล้ว จะทำให้เราจะทราบได้ทันทีว่า ระบบของเราทำงานได้ถูกต้องเมื่อมีพฤติกรรมอย่างไร

การพิสูจน์ที่ใช้ในตรรกศาสตร์เชิงประพจน์จะเน้นเฉพาะการพิสูจน์ที่ใช้การชักเหตุผลแบบนิรนัยเท่านั้น สิ่งที่เรามองหาในอันดับแรก คือ ข้อเท็จจริงที่มีอยู่ทั้งหมดว่ามีอะไรบ้าง ซึ่งจะเขียนอยู่ในรูปของประโยคหรือประพจน์ข้อเท็จจริง จากนั้นเราจะต้องทำการชักเหตุผลแบบนิรนัย โดยใช้กฎการนิรนัยที่รู้มา [15] เช่น กฎการเพิ่มประโยค กฎการลดประโยค กฎ *modus ponens* เป็นต้น ในการหาคำอธิบายที่ทำให้เกิดเป็นประโยคใหม่เพื่อนำไปสู่การสรุปประโยคสุดท้ายให้ได้

เราเรียกสิ่งที่เราจะพิสูจน์ว่าเป็นข้ออ้างเหตุผล (argument) โดยข้ออ้างเหตุผลจะประกอบด้วย ลำดับของประโยคที่เรียงกันอย่างต่อเนื่องเพื่อนำไปสู่ข้อยุติในประโยคสุดท้าย การพิสูจน์จึงเป็นการยืนยันว่าข้ออ้างเหตุผลที่มีอยู่นั้นเป็นข้ออ้างเหตุผลแบบสมเหตุสมผล (valid argument) นั่นเอง

นิยามที่ 2-4: การพิสูจน์ข้ออ้างเหตุผล [15]

กำหนดให้ข้ออ้างเหตุผลที่ต้องการพิสูจน์ประกอบด้วยประโยค $p_1, p_2, p_3, \ldots, p_n$ และ ข้อยุติคือประโยค q เราจะพิสูจน์ว่าข้ออ้างเหตุผลเป็นแบบสมเหตุสมผลได้โดยการพิสูจน์ว่า $(p_1 \wedge p_2 \wedge p_3 \wedge \ldots \wedge p_n) \Longrightarrow q$ จะต้องเป็นสัจนิรันดร์ (tautology) เท่านั้น กระบวนการพิสูจน์ก็ใช้การชักเหตุผลแบบนิรนัยหรือจะใช้แบบแจกแจงตารางค่าความจริงอย่างใดอย่างหนึ่งก็ได้

ตัวอย่างที่ 2-3: การพิสูจน์ $(p \wedge (pVq) \Rightarrow r) \Rightarrow r$ *ด้วยตารางค่าความจริง*

กำหนดให้ข้อเท็จจริงที่เป็นข้อตั้งจำนวนสองประโยค คือ p, $(pVq) \Rightarrow r$ และต้องการข้อยุติคือ r ดังนั้นเขียนเป็นข้ออ้างเหตุผลว่า $(p \wedge (pVq) \Rightarrow r) \Rightarrow r$ เราต้องการพิสูจน์ว่าข้ออ้างเหตุผลนี้มีความสมเหตุสมผลหรือไม่ โดยเราต้องพิสูจน์ว่าข้ออ้างเหตุผลนี้เป็นสัจนิรันดร์หรือไม่นั่นเอง ในตัวอย่างนี้ขอใช้วิธีการแจกแจงตารางค่าความจริง ซึ่งทำได้โดยการเขียนตารางค่าความจริงในตารางที่ 2.1 ซึ่งใช้การตีความส่วนประกอบของแต่ละประโยคทีละส่วนและนำมาพิจารณารวมกันในที่สุด ทั้งนี้การรวมแต่ละครั้งให้คำนึงถึงการมีลำดับเหนือกว่า (precedence) ของแต่ละสัญลักษณ์ด้วย และถ้าค่าความจริงในคอลัมน์สุดท้ายมีค่าเป็นจริงทุกกรณี แสดงว่าเป็นสัจนิรันดร์

ตารางที่ 2.1 ตารางแจกแจงค่าความจริงเพื่อการหาสัจนิรันดร์

P	**q**	**(pVq)**	**r**	**(pVq)⇒ r**	**(p ∧ (pVq)⇒ r)**	**(p ∧ (pVq)⇒ r) ⇒ r**
F	F	F	F	T	F	T
F	T	T	F	F	F	T
T	F	T	F	F	F	T
T	T	T	F	F	F	T
F	F	F	T	T	F	T
F	T	T	T	T	F	T
T	F	T	T	T	T	T
T	T	T	T	T	T	T

กฎการอนุมานแบบนิรนัย

กฎการอนุมานแบบนิรนัยเป็นกฎที่พิสูจน์มาแล้วว่าเป็นสัจนิรันดร์ และเรานำมาใช้ในการพิสูจน์ได้ ตัวอย่างของกฎที่ใช้บ่อยมีดังนี้ [15]

- กฎ *Modus Ponens*

$p, (p \Rightarrow q) \vdash q$

หมายถึงกำหนดข้อตั้งให้เป็น p และ $(p \Rightarrow q)$ แล้ว

เราสรุปได้ว่า q เป็นจริง

ข้อสังเกต คือ กรณีที่ $(p \Rightarrow q)$ เป็นจริงได้มีแค่สองกรณี คือ กรณีแรก ถ้า p เป็นเท็จแล้ว q จะเป็นจริงหรือเป็นเท็จก็ได้ กรณีที่สอง ถ้า p เป็นจริงแล้ว q จะต้องเป็นจริงเท่านั้นจะเป็นเท็จไม่ได้ เนื่องจากข้อตั้งที่ว่า p เป็นจริงแน่นอน ดังนั้น q ก็ต้องเป็นจริงแน่นอน

- กฎ *Modus Tollen*

  $\neg q, (p \Rightarrow q) \vdash \neg p$

  หมายถึงกำหนดข้อตั้งให้เป็น $\neg q$ และ $(p \Rightarrow q)$ แล้ว

  เราสรุปได้ว่า $\neg p$ เป็นจริง

  ข้อสังเกต คือ เนื่องจากข้อตั้งบอกว่า $\neg q$ เป็นจริง ดังนั้น q เป็นเท็จแน่นอน ทำให้ต้องมาดูกรณีที่ $(p \Rightarrow q)$ เป็นจริงนั้น ค่า q จะเป็นเท็จได้ไหม คำตอบคือ ค่า q เป็นเท็จได้ และค่า p ต้องเป็นเท็จเสมอเท่านั้น ทำให้เราสรุปได้ทันทีว่า $\neg p$ เป็นจริงเสมอเช่นกัน

- กฎ *Hypothetical Syllogism*

  $(p \Rightarrow q), (q \Rightarrow r) \vdash (p \Rightarrow r)$

  หมายถึงกำหนดข้อตั้งให้เป็น $(p \Rightarrow q)$ และ $(q \Rightarrow r)$ แล้ว

  เราสรุปได้ว่า $(p \Rightarrow r)$ เป็นจริง

  ข้อสังเกต คือ การพิสูจน์กฎข้อนี้ทำได้โดยการใช้ตารางค่าความจริงแจกแจงทุกกรณีของ p, q, r และจะพบข้อยุติตามกฎได้

- กฎ *Disjunctive Syllogism*

  $(p \vee q), \neg p \vdash q$

  หมายถึงกำหนดข้อตั้งให้เป็น $(p \vee q)$ และ $\neg p$ แล้ว

  เราสรุปได้ว่า q เป็นจริง

  ข้อสังเกต คือ กำหนดให้ $\neg p$ เป็นจริงเสมอ จึงแสดงว่า p เป็นเท็จเสมอเช่นกัน ดังนั้นกรณีเดียวที่ $(p \vee q)$ เป็นจริงเสมอ ก็คือ q ต้องเป็นจริงเสมอนั่นเอง

- กฎ $\vee$ *Addition*

  $p \vdash (p \vee q)$

  หมายถึงกำหนดข้อตั้งให้เป็น p แล้ว

  เราสรุปได้ว่า $(p \vee q)$ เป็นจริง

ข้อสังเกต คือ ถ้ากำหนดให้ p เป็นจริงเสมอแล้ว เมื่อนำ q ที่ค่าเป็นจริงหรือเป็นเท็จก็ตามมาใช้ $(p \vee q)$ ก็จะเป็นจริงเสมอ เนื่องจากมี p เป็นจริงตั้งต้นอยู่แล้ว

- กฎ *Simplification*

$$(p \wedge q) \vdash p$$

หมายถึงกำหนดข้อตั้งให้เป็น $(p \wedge q)$ แล้ว

เราสรุปได้ว่า p เป็นจริง

ข้อสังเกต คือ ถ้ากำหนดให้ $(p \wedge q)$ เป็นจริงเสมอแล้ว เป็นการบังคับเลยว่าทั้ง p, q จะต้องเป็นจริงเสมอตามมาด้วยนั่นเอง ดังนั้นทั้ง p และ q ก็ต้องเป็นจริงแน่นอนทั้งคู่

ตัวอย่างที่ 2-4: การพิสูจน์ $(p \wedge (p \vee q) \Rightarrow r) \Rightarrow r$ *ด้วยการอนุมานแบบนิรนัย*

จากตัวอย่างที่ 2-3 ที่ใช้การพิสูจน์สัจนิรันดร์ด้วยตารางค่าความจริง เรามีอีกทางเลือกหนึ่ง คือ การพิสูจน์โดยใช้การชักเหตุผลแบบนิรนัยไปข้างหน้า (forward deductive inference) จากตัวอย่างการพิสูจน์ว่า $(p \wedge (p \vee q) \Rightarrow r) \Rightarrow r$ ว่าเป็นสัจนิรันดร์หรือไม่นั้น เราพบว่ามีประโยคที่เป็นข้อตั้งอยู่สองประโยคคือ p และ $(p \vee q) \Rightarrow r$ และต้องการสรุปว่า r เป็นประโยคข้อยุติ เราสามารถเขียนการชักเหตุผลได้ดังนี้

(1)	p	*premise*$_1$
(2)	$(p \vee q) \Rightarrow r$	*premise*$_2$
(3)	$p \vee q$	(1), $\vee$ *addition*
(4)	r	(2), (3), *modus ponens*

อธิบายความการอนุมานข้างต้น ดังนี้

ประโยคที่ (1), (2) เป็นจริงเสมอ เนื่องจากกำหนดให้มาเป็นข้อตั้ง และเราต้องการสรุปประโยคที่ (4) ว่าจริงเสมอ ดังนั้นเราต้องหาประโยคมาอธิบายว่าจากประโยคที่ (1), (2) แล้ว จะอธิบายต่อเนื่องอย่างไรที่จะบอกได้ว่าประโยคที่ (4) เป็นจริง เราพบกฎนิรนัยข้อ $\vee$ *addition* ที่บอกว่าถ้าเรามี p เป็นจริงแล้วจากประโยคที่ (1) เราจะเพิ่ม q ที่จะมีค่าความจริงอย่างใดก็ได้สำหรับเครื่องหมาย $\vee$ เราจึงได้ว่าประโยคที่ (3) คือ $p \vee q$ ต้องจะเป็นจริงเสมอ และด้วยประโยคที่ (2), (3) พร้อมทั้งใช้กฎนิรนัยข้อ *modus ponens* เราจะสรุปที่ได้ว่าประโยคที่ (4) คือ r เป็นจริงเสมอ

เรามีวิธีการเขียนข้ออ้างเหตุผลที่พิสูจน์แล้วข้างต้น ได้อีกแบบหนึ่งในรูปแบบข้อตั้งและข้อยุติ ได้ดังนี้

p, $(p \vee q) \Rightarrow r \quad \vdash r$ โดยเครื่องหมาย ⊢ ซึ่งหมายถึงนิรนัยจากข้อตั้งทั้งหลายที่อยู่ทางด้านซ้ายของเครื่องหมายไปสู่ข้อยุติทางด้านขวา

2.6 ความพอใจและความมีเหตุผล

กำหนดให้สูตรเชิงประพจน์ A และการตีความ $\mathfrak{T}$ ถ้าเราสามารถหาการตีความ $\mathfrak{T}$ หนึ่งใดที่ทำให้การค่าความจริงของ A เป็นค่าจริงได้ เราจะเรียกได้ว่า "$\mathfrak{T}$ *satisfies* A" โดยที่เราเขียนได้ด้วยสัญลักษณ์นำมาซึ่ง (entail symbol) ว่า

$\mathfrak{T} \models A$

ตัวอย่างของการตีความที่ทำให้เกิดความพอใจ (satisfaction) คือ กำหนดให้ $A = (p \wedge q) \Rightarrow r$ และเราต้องการหาว่า $\mathfrak{T} \models (p \wedge q) \Rightarrow r$ เป็นจริงหรือไม่ ดังนั้นเราจะต้องหาการตีความอย่างน้อยหนึ่งกรณีที่ทำให้ค่าความจริงของ A เป็นค่าจริง และเราพบว่า $\mathfrak{T}_{\text{atomic}}(p) = F$, $\mathfrak{T}_{\text{atomic}}(q) = T$, and $\mathfrak{T}_{\text{atomic}}(r) = T$ เป็นการตีความที่ทำให้ A เป็นจริง ดังนั้นจึงสรุปได้ว่า $\mathfrak{T} \models (p \wedge q) \Rightarrow r$ เป็นจริงหรือเรียกอีกอย่างว่า A นั้นมีลักษณะเป็นที่พอใจได้ (satisfiable)

แต่ในทางกลับกันถ้าเราไม่สามารถหาการตีความแม้สักกรณีเดียวมายืนยันว่า A เป็นจริงได้ เราจะเรียกว่าไม่เป็นที่พอใจได้ (unsatisfiable) โดยเขียนได้ด้วยสมการ $\mathfrak{T} \not\models A$

ตัวอย่าง คือ

$(p \wedge q) \Rightarrow r$ เป็นสูตรที่พอใจได้

$(p \wedge F)$ เป็นสูตรที่ไม่เป็นที่พอใจได้

นอกจากนี้ ถ้ากรณีที่เราพบว่าทุกกรณีของการตีความ $\mathfrak{T}$ นั้นจะทำให้ค่า A เป็นจริงเสมอ เราเรียก A ว่ามีความมีเหตุผล (validity) หรือเขียนด้วย $\models A$ (โดยละสัญลักษณ์ด้านซ้ายของเครื่องหมาย ⊨ ไว้) ในบางครั้งเราสามารถเทียบเคียงความมีเหตุผลนี้ว่า เป็นสัจนิรันดร์นั่นเอง เช่น ประพจน์ "$(p \vee T)$" และ "$p \Rightarrow p$" มีความมีเหตุผลทั้งคู่ เป็นต้น

2.7 ตรรกศาสตร์ภาคแสดง

ตรรกศาสตร์ภาคแสดง (predicate logic) [14] คือ การนำตรรกศาสตร์มาใช้กับกรณีที่ประโยคไม่เป็นประโยคประพจน์ จากที่ทราบมาแล้วว่า ประพจน์ คือ ประโยคที่มีค่าความจริงเป็นจริงหรือเท็จอย่างใดอย่างหนึ่งนั้น ถ้าเราพบว่ามีประโยคที่มีตัวแปรเสรี (free variable) ปรากฏอยู่ในประโยคและสามารถรู้ค่าความจริงได้โดยการแทนค่าตัวแปรนั้น ๆ แล้ว เราเรียกประโยคที่มีตัวแปรนี้ว่าประโยคภาคแสดง (predicate statement)

โดยทั่วไปเมื่อเราพบประโยคภาคแสดง จะไม่สามารถประเมินว่า มีค่าความจริงเป็นค่าจริงหรือค่าเท็จได้ เช่น $x = 1$ เป็นต้น เราจะไม่รู้แน่จนกว่าจะมีการแทนค่าตัวแปรเสรีชื่อ x ก่อน ถ้าแทนค่า x ให้มีค่าเป็น 1 ประโยคนี้จะมีค่าความจริงเป็นจริง หรือถ้าแทนค่า x ให้มีค่าเป็น 2 ประโยคนี้จะมีค่าความจริงเป็นเท็จ

แคลคูลัสภาคแสดง

แคลคูลัสภาคแสดง [14] คือ การนำประโยคภาคแสดงมาประกอบกันเป็นประโยคใหม่โดยใช้ตัวเชื่อม เราอาจจะเรียกได้ว่า แคลคูลัสเชิงฟังก์ชัน (functional calculus) แต่เราสามารถปรับเปลี่ยนประโยคภาคแสดงที่มีตัวแปรเสรีที่ต้องแทนค่าก่อนนั้น ให้เป็นประโยคแบบใหม่ที่รู้ค่าความจริงได้ ทำให้ประโยคแบบใหม่นี้ไม่เป็นประโยคภาคแสดงอีกต่อไป เราทำได้ด้วยการเพิ่มส่วนขยายความของประโยคที่เรียกว่าตัวบ่งปริมาณแบบ *“for all”* แทนด้วยสัญลักษณ์ $\forall$ หรือแบบ *“there exists”* แทนด้วยสัญลักษณ์ $\exists$ ซึ่งจะทำให้ประโยคภาคแสดงเดิมไม่เป็นประโยคภาคแสดงอีกต่อไป

การกำหนดปริมาณ

การกำหนดปริมาณให้ตัวแปรมีสามประเภทดังต่อไปนี้

1. การกำหนดปริมาณสำหรับทุกตัว (universal quantification) เขียนโดยใช้สัญกรณ์ $\forall x \in X\ [P(x)]$ หมายถึง ทุกค่าของตัวแปร x ที่อยู่ในเซต X ทำให้ $P(x)$ เป็นจริง
2. การกำหนดปริมาณสำหรับตัวมีจริง (existential quantification) เขียนโดยใช้สัญกรณ์ $\exists x \in X\ [P(x)]$ หมายถึง บางค่าของตัวแปร x ที่อยู่ในเซต X ทำให้ $P(x)$ เป็นจริง
3. การกำหนดปริมาณสำหรับตัวมีจริงตัวเดียว (unique existential quantification) เขียนโดยใช้สัญกรณ์ $\exists ! x \in X\ [P(x)]$ หมายถึง ทุกค่าเดียวของตัวแปร x ที่อยู่ในเซต X ทำให้ $P(x)$ เป็นจริง

ตัวอย่างที่ 2-5: การกำหนดปริมาณให้ตัวแปรในประโยคภาคแสดง

กำหนดให้

$1+1=2$ เป็นประโยคประพจน์ที่มีค่าความเป็นจริงเป็นค่าจริง

$1+2=2$ เป็นประโยคประพจน์ที่มีค่าความเป็นจริงเป็นค่าเท็จ

$x+1=2$ ไม่เป็นประโยคประพจน์แต่เป็นประโยคภาคแสดง

เพราะมีตัวแปรเสรีชื่อ x ที่ต้องมีการแทนค่าเพื่อจะได้รู้ค่าความเป็นจริง

แต่เมื่อเรากำหนดปริมาณเพิ่มเข้าไปแล้วเป็น $\exists x \in \mathbb{N}\ [x+1=2]$ จะเป็นประโยคประพจน์และมีค่าความจริงเป็นค่าจริง เนื่องจากที่มีการขยายความว่า *"มีบางค่าของ x ที่เป็นตัวเลขธรรมชาติที่ทำให้ $x+1=2$ ได้"* นั่นก็คือค่า $x=1$ นั่นเอง โดยตัวแปร x จะไม่เป็นตัวแปรเสรีอีกต่อไป มันจะกลายเป็นตัวแปรแบบมีขอบเขตหลังจากการกำหนดปริมาณ

อีกตัวอย่างหนึ่ง คือ เมื่อเรากำหนดปริมาณเป็น $\forall x \in \mathbb{N}\ [x+1=2]$ จะเป็นประโยคประพจน์ได้เช่นกัน และมีค่าความจริงเป็นค่าเท็จ เนื่องจากมีการขยายความว่า *"มีทุกค่าของ x ที่เป็นตัวเลขธรรมชาติที่ทำให้ $x+1=2$ ได้"* ซึ่งไม่จริงเพราะมีบางค่าเท่านั้น

2.8 ทฤษฎีเซต

เซต [14] คือ กลุ่ม (collection) ของสิ่งของ (object) หรือสมาชิก (element) ที่รวบรวมไว้ด้วยกัน โดยสิ่งของหรือสมาชิกที่อยู่ในเซตนี้จะต้องเป็นชนิด (type) เดียวกัน

การเขียนอธิบายเซตสามารถทำได้หลายรูปแบบ แบบแรกที่ง่ายนั้น เราสามารถเขียนอธิบายเซตโดยการเขียนแบบแจกแจง ดังนี้ $\{1,\ 2,\ 3,\ 4,\ 5,\ \ldots\}$ กล่าวคือ ให้เขียนด้วยเครื่องหมายวงเล็บปีกกา {…} และให้เขียนสมาชิกทีละตัวภายในวงเล็บให้ครบ แต่กรณีที่จำนวนสมาชิกที่จำนวนมากๆ หรือมีจำนวนไม่สิ้นสุดและเราไม่สามารถเขียนแจกแจงได้ครบ ให้ใช้สัญลักษณ์จุดดังนี้ "…" เพื่อแทนส่วนที่เหลือ

เรานิยมเขียนอธิบายเซตอีกรูปแบบหนึ่ง คือ การเขียนบรรยายเซตแบบบอกเงื่อนไข โดยให้เขียนบรรยายเป็นสองส่วนหลัก คือ ส่วนลายเซ็น (signature part) และส่วนเพรดิเคต (predicate part) ส่วนลายเซ็นกำหนดตัวแปรและชนิดของตัวแปร และส่วนเพรดิเคตกำหนดเงื่อนไขบังคับ โดยคั่นกลางทั้งสองส่วนหลักด้วยเครื่องหมาย |

ตัวอย่างเช่น $\{\ x \in \mathbb{N} \mid x \geq 1\ \}$ ซึ่งส่วนประกาศ คือ $x \in \mathbb{N}$ และส่วนเพรดิเคตคือ $x \geq 1$ ซึ่งมีความหมายว่าเซตของค่า x ที่มีชนิดจำนวนธรรมชาติที่มีค่าตั้งแต่ 1 ขึ้นไป

ในที่นี้กำหนดให้มีสัญกรณ์ (notation) แทนเซตบางประเภทที่ใช้บ่อย เช่น ใช้สัญกรณ์ $\mathbb{N}$ แทนเซตจำนวนธรรมชาติ หรือใช้สัญกรณ์ $\mathbb{Z}$ แทนเซตจำนวนเต็ม เป็นต้น

ภาวะสมาชิกของเซต

เราระบุภาวะสมาชิกของเซตได้ โดยใช้สัญลักษณ์ $\in$ กล่าวคือ ถ้า x เป็นสมาชิกของเซต S เราสามารถเขียนระบุได้ว่า $x \in S$

ต่อไปนี้เป็นตัวอย่างการระบุภาวะสมาชิกของเซต

- $0 \in \{0,1,2\}$ คือ ตัวเลข 0 เป็นสมาชิกของเซต $\{0,1,2\}$
- $\varnothing \in \{\varnothing\}$ คือ $\varnothing$ เป็นสมาชิกของ $\{\varnothing\}$
- $0 \notin \varnothing$ คือ ตัวเลข 0 ไม่เป็นสมาชิกของ $\varnothing$ ทั้งนี้อย่าเข้าใจว่าเซตว่างจะต้องมีสมาชิกเป็นตัวเลข 0
- $Sombat \notin \{Somchai, Sirichai, Visoot\}$

การดำเนินการที่เกิดกับเซต

กำหนดให้ P และ Q เป็นเซตที่มีชนิดเดียวกัน และต่อไปนี้เป็นคำอธิบายของการดำเนินการที่เกิดกับเซต P และ Q

- $P \cup Q$ อ่านว่า "*P union Q*" คือ การหาส่วนรวมของเซต P และ Q โดยสมาชิกทุกตัวของเซต P จะถูกนำไปรวมกับสมาชิกทุกตัวของ Q กล่าวคือ เซตใหม่ที่เกิดขึ้นมาจะมีสมาชิกที่อยู่ในเซต P หรือ Q อย่างใดอย่างหนึ่งหรืออาจจะอยู่ในทั้งสองเซต
- $P \cap Q$ อ่านว่า "*P intersect Q*" คือ การหาส่วนร่วมของเซต P และ Q โดยสมาชิกของเซตใหม่ที่เกิดขึ้นของการดำเนินการนี้ จะมีสมาชิกที่ต้องอยู่ทั้งในเซต P และ Q ทั้งสองเซตเท่านั้น
- $P \backslash Q$ อ่านว่า "*P minus Q*" คือ การหาส่วนต่างของเซต P ที่ต่างจากเซต Q โดยเหมือนใช้เซต P เป็นตัวตั้งและลบออกด้วยเซต Q กล่าวคือ เซตใหม่ที่เกิดขึ้นของการดำเนินการนี้ จะมีสมาชิกที่อยู่ในเซต P แต่ไม่อยู่ใน Q เลย
- $P = Q$ อ่านว่า "*P equals Q*" คือ การระบุความสมมูลของเซต P และ Q โดยความสมมูลของเซตทั้งสองเซตไม่สนใจว่าสมาชิกจะมีการเรียงลำดับสมาชิกเหมือนกันและมีการซ้ำของสมาชิกหรือไม่ ตัวอย่างเช่น

  $\{Somchai\} = \{Somchai, Somchai\}$ ทั้งสองเซตยังคงสมมูลกันถึงแม้ว่ามีการซ้ำของสมาชิก

  $\{Sirichai, Somsri\} = \{Somsri, Sirichai\}$ ทั้งสองเซตยังคงสมมูลกันถึงแม้ว่าจะมีการเรียงลำดับสมาชิกไม่เหมือนกัน
- $(P \backslash Q) \cup (P \cap Q) \cup (Q \backslash P) = P \cup Q$ เซตที่เกิดจากการดำเนินการที่เกิดขึ้นกับเซตด้านซ้าย และเซตที่เกิดจากการดำเนินการด้านขวาจะ

สมมูลกัน ดังนั้นเราสามารถนำการสมมูลของเซตไปใช้ประโยชน์ในการลดรูป หรือการเขียนทดแทนกันได้

- $\neg P = Q$ ถ้านิเสธของเซต P สมมูลกับเซต Q แล้ว เราสามารถตีความได้ทันทีว่าเซต P และเซต Q ไม่เป็นเซตที่เหมือนกันแน่ ดังนั้นเราเขียนได้ว่า $P \neq Q$ โดยที่ $P \neq Q$ จะสมมูลกับ $\neg(P = Q)$
- $P \subseteq Q$ อ่านว่า "P *contained in* Q" คือ การระบุการเป็นเซตย่อย (subset) โดยระบุว่าเซต P เป็นเซตย่อยของเซต Q กล่าวคือสมาชิกของเซต P จะต้องปรากฏเป็นสมาชิกของเซต Q ทุกตัว สัญลักษณ์ $\subseteq$ นี้อาจจะรวมกรณี $P = Q$ ด้วยเช่นกัน
- $P \subset Q$ อ่านว่า "P *strictly contained in* Q" คือ การระบุการเป็นเซตย่อยแท้ (proper subset) โดยระบุว่าเซต P เป็นเซตย่อยแท้ของเซต Q กล่าวคือสมาชิกของเซต P จะต้องปรากฏเป็นสมาชิกของเซต Q ทุกตัวเช่นเดียวกัน แต่ทั้งนี้จะต้อง $P \neq Q$ ด้วย เราเขียนเป็น ประโยคได้ว่า ($P \subseteq Q \wedge P \neq Q$)
- P' อ่านว่า "*complement of* P" คือ การหาส่วนเติมเต็มที่เหลือ (complementation) ของเซต P ที่เป็นไปได้ทั้งหมดที่ไม่อยู่ในเซต P กล่าวคือ เซตที่ได้จะมีเป็นสมาชิกในชนิดนั้นทุกตัวที่เป็นไปได้ โดยมีข้อแม้ว่าจะไม่ปรากฏในเซต P หรือกล่าวสั้นๆว่า P' จะมีสมาชิกทุกตัวที่อยู่ในเซต P ยกตัวอย่างเช่น สำหรับชนิด T โดยที่ $P \subseteq T$ แล้ว $P' = T \setminus P$ เป็นต้น

การดำเนินการบนเซตแบบทั่วไป

เราเขียนการดำเนินการบนเซตแบบทั่วไป (generalized set operations) โดยการดำเนินการเหล่านั้น คือ การหาส่วนรวม (union), การหาส่วนร่วม (intersection) เป็นต้น

กำหนดให้เซต $X = \{A, B, C, \ldots\}$ โดยที่เซต X คือ เซตของเซตย่อยใด เช่น เซตย่อย A, เซตย่อย B เป็นต้น

เราสามารถเขียนการหาส่วนรวมของทุกเซตย่อยในเซต X ได้โดยเขียนสัญลักษณ์ big union นำหน้าเซต X

$\bigcup\{A, B, C, \ldots\} = A \cup B \cup C \cup \ldots$

เราสามารถเขียนการหาส่วนร่วม (Intersection) ของทุกเซตย่อยในเซต X ได้โดยเขียนสัญลักษณ์ big intersection นำหน้าเซต X

$\bigcap\{A, B, C, \ldots\} = A \cap B \cap C \cap \ldots$

การเขียนบรรยายเซตแบบบอกเงื่อนไข

เราเขียนเซต X ได้โดยการแจกแจงสมาชิกของเซตในเครื่องหมายวงเล็บ คือ $X = \{1,2,3,4,5, \ldots\}$ แต่การเขียนในลักษณะแจกแจงอาจจะทำให้เราเขียนอย่างยุ่งยาก

ดังนั้น เราสามารถเขียนเซต X ด้วยการเขียนบรรยายเซตแบบบอกเงื่อนไขแทนได้โดยทำการบรรยายส่วนของเซตเป็นสองส่วน คือ ส่วนลายเซ็นและส่วนเพรดิเคต ส่วนลายเซ็นเป็นส่วนที่ใช้ประกาศของตัวแปรและชนิดของตัวแปร และส่วนเพรดิเคตเป็นส่วนที่ใช้ระบุเงื่อนไขบังคับ (constraint) ซึ่งเขียนเป็นประโยคภาคแสดงได้

เราเขียนบรรยายเซตแบบบอกเงื่อนไขได้ดังนี้

$$X = \{x: type \mid predicate(x)\}$$

หรือ

$$X = \{x: type1;\ y: type2 \mid predicate(x,y)\}$$

ดูตัวอย่างดังต่อไปนี้

- $X = \{x: \mathbb{N} \mid PrimeNumber(x)\}$
  โดยที่ $PrimeNumber(x)$ เป็นประโยคเพรดิเคตที่ตรวจสอบจำนวนเฉพาะของค่าตัวแปร x ที่ส่งเข้าไปและให้ค่าความจริงเป็นจริงเท่านั้นจึงจะเป็นสมาชิกของเซตได้
- $X = \{x: Path \mid LeastCost(x)\}$
  โดยที่ $LeastCost(x)$ เป็นประโยคเพรดิเคตที่ตรวจสอบค่าใช้จ่ายที่น้อยที่สุดของเส้นทาง x ที่มีและให้ค่าความจริงเป็นจริงเท่านั้น

ความสัมพันธ์ของเซตและประโยคภาคแสดง

เราสามารถเทียบเคียงเขียนบรรยายการดำเนินการบนเซตได้ ด้วยการใช้ตัวเชื่อมในประโยคของส่วนเพรดิเคตได้ ดังนี้

การทำ set intersection ($\cap$) และการทำ logical conjunction ($\wedge$)

$$\{x: S \mid p\} \cap \{x: S \mid q\} = \{x: S \mid p \wedge q\}$$

การทำ set union ($\cup$) และการทำ logical disjunction ($\vee$)

$$\{x: S \mid p\} \cup \{x: S \mid q\} = \{x: S \mid p \vee q\}$$

การทำ set complementation และการทำ negation

$$\{x: S \mid p\}' = \{x: S \mid \neg p\}$$

การทำ subset ($\subseteq$) และการทำ implication ($\Longrightarrow$)

$$\{x: S \mid p\} \subseteq \{x: S \mid q\} \text{ iff } p \Longrightarrow q$$

การทำ set equality ($=$) และการทำ equivalence ($\Longleftrightarrow$)

$$\{x: S \mid p\} = \{x: S \mid q\} \text{ iff } p \Longleftrightarrow q$$

ผลคูณคาร์ทีเชียน

ผลคูณคาร์ทีเชียน $S \times U$ คือ เซตของคู่ลำดับ (s, u) โดยที่ $t \in T$ และ $u \in U$ ลองพิจารณาตัวอย่าง ดังนี้ กำหนดให้ $S=\{s_1, s_2, s_3\}$ และ $U=\{u_1,u_2\}$ ดังนั้น $S \times U = \{(s_1,u_1), (s_1,u_2), (s_2,u_1), (s_2,u_2), (s_3,u_1), (s_3,u_2)\}$ ซึ่งหมายถึงเป็นเซตของคู่ลำดับ (s, u) ที่เป็นไปได้ทุกกรณีซึ่งหมายถึงต้องมีการพบกันหมดทุกตัวจากทั้งสองเซต

จำนวนของสมาชิกของ $S \times U$ หาได้จากผลคูณของจำนวนสมาชิกเซต S และจำนวนสมาชิกเซต U นั่นคือ 3×2 เท่ากับ 6 และโดยทั่วไปเซต S และเซต U ไม่จำเป็นต้องเหมือนกันก็ได้

เซตของ n-สิ่งอันดับ

เซตของ n-สิ่งอันดับ (n-tuple) เขียนด้วยการคูณของเซตจำนวน n เซต ได้ดังนี้

$$E_1 \times E_2 \times \ldots \times E_n$$

โดยสมาชิกของเซตเรียกว่า n-สิ่งอันดับ และเขียนได้เป็น $(e_1, e_2, \ldots, e_n)$ ซึ่งเราสามารถยกตัวอย่างได้ดังนี้

กำหนดให้ $A = \{a_1, a_2\}$, $B = \{b_1, b_2, b_3\}$, $C = \{c_1, c_2\}$

$A \times B \times C$ ก็คือเซตของ *3-tuple* ที่มีค่าดังต่อไปนี้

$A \times B \times C = \{(a_1,b_1,c_1), (a_1,b_1,c_2), (a_1,b_2,c_1), (a_1,b_2,c_2), \ldots\ (a_2,b_3,c_2)\}$

โดยสมาชิกก็คือ (a,b,c) โดยที่ $a \in A$ และ $b \in B$ และ $c \in C$

ตัวอย่างที่เรามักพบในข้อกำหนดเชิงรูปนัยทั่วไปสำหรับระบบซอฟต์แวร์ เนื่องจากเรามักจะกำหนดเซตที่มีความหมายในการทำงานของระบบจริง เช่น เซตของชื่อคน เซตของชื่อวิธี เซตของชื่อสถานที่ เป็นต้น

กำหนดให้เซต $Methods = \{Z, CafeOBJ\}$ และเซต $People = \{Somchai, Sombat\}$ เราหาผลคูณคาร์ทีเชียนของ $Methods \times People$ ได้ดังนี้

$Methods \times People = \{(Z,Somchai), (Z,Sombat), (CafeOBJ, Somchai), (CafeOBJ,Sombat)\}$

เราสามารถเขียนระบุได้ว่า $(Z,Sombat) \in Methods \times People$

เซตกำลัง

กำหนดให้เซต S แล้วเซตกำลัง $\mathbb{P}S$ คือ เซตของเซตย่อยของเซต S ทุกเซตที่เป็นไปได้ นั่นคือ $X \in \mathbb{P}S \Leftrightarrow X \subseteq S$ กล่าวคือ เซตกำลังคือเซตของเซตอีกทีนั่นเอง และสมาชิกทุกตัวต้องเป็นเซตเสมอ

จะเห็นได้ว่า $\varnothing \in \mathbb{P}S$ ด้วยเพราะ $\varnothing \subseteq S$ เช่นกัน ดังนั้นเราจะเห็นได้เลยว่า ถ้ากำหนดให้เซต S แล้ว $\mathbb{P}S$ จะต้องมีสมาชิกเสมอ คือ จะต้องไม่เป็นเซตว่าง $\mathbb{P}S \neq \varnothing$ ดูตัวอย่างดังต่อไปนี้

- $\mathbb{P}\varnothing = \{\varnothing\}$ หมายถึงเซตว่างซึ่งไม่สมาชิกใด ก็สามารถหาเซตกำลังของเซตว่างใดๆได้ โดยผลลัพธ์ก็คือ เซตกำลังของเซตว่างจะมีสมาชิกอยู่หนึ่งเดียวคือเซตว่างนั่นเอง
- $\mathbb{P}\{a\} = \{\varnothing, \{a\}\}$ หมายถึงเซตกำลังของเซตใด จะประกอบด้วยสมาชิกที่เป็นเซตย่อยที่เป็นไปได้ของเซตที่เราสนใจ ทั้งนี้รวมถึงเซตว่างของเซตที่เราสนใจด้วย

2.9 ความสัมพันธ์

ความสัมพันธ์ R คือ เซตย่อยหนึ่งใดของผลคูณคาร์ทีเซียนของเซตสองเซต กล่าวคือ $R \subseteq P \times Q$ โดยที่ R คือ ความสัมพันธ์ P คือ เซตต้นทางที่เป็นโดเมน และ Q คือ เซตปลายทางที่เป็นเรนจ์

ตัวอย่างของความสัมพันธ์

$AdjacencyRel \subseteq Places \times Places$ หมายถึง ความสัมพันธ์ *AdjacencyRel* เป็นความสัมพันธ์ระหว่างสถานที่สองแห่งที่อยู่ติดกัน โดย *Places* เป็นเซตของสถานที่

ถ้า $Places = \{Bangkok, Nonthaburi, Thonburi\}$ แล้ว

$AdjacencyRel = \{(Bangkok, Thonburi), (Bangkok, Nonthaburi)\}$ ความสัมพันธ์ คือ เซตย่อยหนึ่งของ $Places \times Places$ และสมาชิก (*Bangkok, Thonburi*) หมายถึงสถานต้นทางที่ชื่อ *Bangkok* มีความสัมพันธ์ที่เรียกว่า *AdjacencyRel* กับสถานที่ปลายทางที่ชื่อ *Thonburi* เป็นต้น

$PriceListRel \subseteq Product \times Price$ หมายถึงความสัมพันธ์ *PriceListRel* เป็นความสัมพันธ์ระหว่างเซตต้นทางโดเมน *Product* ไปสู่เซตปลายทางเรนจ์ *Price*

ถ้า $Product = \{Ball, Chair, Pen\}$ และ $Price = \{100,200,300,400\}$ แล้ว $PriceListRel = \{(Ball,200), (Chair,400), (Pen,100)\}$ ความสัมพันธ์นี้เป็นการระบุว่าสินค้าใดมีราคาเท่าไร เช่น (*Ball,200*) หมายถึงสินค้าต้นทาง *Ball* จะมีความสัมพันธ์แบบ *PriceListRel* กับราคาปลายทางคือ *200* เป็นต้น

2.10 พีชคณิตบูลีน

หลังจากเกิดตรรกศาสตร์เชิงสัญลักษณ์หรือที่เรียกว่าคณิตตรรกศาสตร์แล้ว ในปี ค.ศ. 1847 *George Boole* ได้คิดค้นพีชคณิตบูลีนที่มีประโยชน์และข้อดีอย่างยิ่งในวงการคอมพิวเตอร์ *Claude Shannon* ได้นำพีชคณิตบูลีนมาประยุกต์ใช้ในทฤษฎีทางสวิตซ์ (switching theory) สำหรับการออกแบบวงจรตรรกศาสตร์เกต (logic gate) [12] โดยเราสามารถสร้างสมการพีชคณิตที่ประกอบด้วยตัวแปรบูลีนที่มีค่าจริงเขียนแทนด้วยสัญลักษณ์ T หรือค่าเท็จเขียนแทนด้วยสัญลักษณ์ F (บางครั้งพบว่าอาจจะใช้สัญลักษณ์ *0* และ *1* แทนตามลำดับ)

พีชคณิตบูลีน [12] คือ ระบบพีชคณิตทางคณิตศาสตร์ที่ดัดแปลงให้มีตัวแปรแบบใหม่คือ ตัวแปรบูลีน และมีตัวดำเนินการแบบใหม่ (แทนที่จะใช้การบวก ลบ คูณ หาร แบบระบบตัวเลขในคณิตศาสตร์) คือ ตัวดำเนินการบูลีน ซึ่งประกอบด้วยการดำเนินการ *“และ”* หรือเรียกว่าการเชื่อม (conjunction) ด้วยสัญลักษณ์ $\wedge$, การดำเนินการ *“หรือ”* หรือเรียกว่าการเลือก (disjunction) ด้วยสัญลักษณ์ $\vee$, การดำเนินการ *“ไม่”* หรือเรียกว่าการนิเสธ (negation) ด้วยสัญลักษณ์ $\neg$, ตัวดำเนินการ *“ถ้า..แล้ว”* หรือเรียกว่าการแจงเหตุสู่ผล (implication) ด้วยสัญลักษณ์ $\Longrightarrow$, ตัวดำเนินการ *“เท่ากับ”* หรือเรียกว่าการเทียบสมมูล (equivalence) ด้วยสัญลักษณ์ $\Leftrightarrow$ หรือ **≡**, ตัวดำเนินการ *“เลือก”* หรือเรียกว่าตัวเลือกอย่างใดอย่างหนึ่ง (exclusive or) ด้วยสัญลักษณ์ $\oplus$

ตารางที่ 2-2 ตารางค่าความจริงของตัวดำเนินการบูลีน [12]

x	$(\neg y)$
F	T
T	F

x	y	$(x \wedge y)$	$(x \vee y)$	$(x\ xor\ y)$	$(x \rightarrow y)$	$(x \leftrightarrow y)$
F	F	F	F	F	T	T
F	T	F	T	T	T	F
T	F	F	T	T	F	F
T	T	T	T	F	T	T

เราสามารถนำพีชคณิตบูลีนไปใช้กำหนดเงื่อนไขในประโยคในระบบตรรกศาสตร์เชิงประพจน์ หรือระบบตรรกศาสตร์อันดับที่สูงกว่าได้อย่างมี

กลมกลืนและมีประสิทธิภาพ การตีความค่าความจริงของผลการดำเนินการในพีชคณิตบูลีน พิจารณาตามกฎการตีความค่าความเป็นจริงแสดงในตารางที่ 2-2

2.11 แบบฝึกหัด

1. คณิตตรรกศาสตร์คืออะไร เกิดขึ้นมาได้อย่างไร?
2. ตรรกศาสตร์เชิงสัญลักษณ์คืออะไร?
3. การพิสูจน์ทางตรรกศาสตร์มีกี่แบบ แตกต่างกันอย่างไร?
4. เราเลือกวิธีการพิสูจน์ทางตรรกศาสตร์แบบใดจึงจะมีความแม่นยำและมั่นใจสูงสุด เพราะเหตุใด?
5. ประพจน์คืออะไร และใช้ประโยชน์ได้อย่างไร?
6. ตัวเชื่อมทางตรรกะที่ใช้ในการทำแคลคูลัสเชิงประพจน์มีอะไรบ้าง?
7. ประโยคภาคแสดงโดยทั่วไป (predicate) ไม่สามารถประเมินค่าความเป็นจริงได้แน่นอน จะต้องทำอย่างไรจึงจะทำให้ประโยคภาคแสดงกลายมาเป็นประพจน์ที่มีค่าความเป็นจริงแน่นอน?
8. การกำหนดปริมาณด้วยตัวบ่งปริมาณมีรูปแบบใดบ้าง?
9. เซตคืออะไร ใช้ทำประโยชน์อย่างไร?
10. เซตกำลังคืออะไร ใช้ประโยชน์อย่างไรในการออกแบบระบบ?
11. การดำเนินการหลักที่กระทำกับเซตทำได้อย่างไรบ้าง?
12. ผลคูณคาร์ทีเซียนอธิบายด้วยเซตได้อย่างไร?
13. A X B X C จะมีค่าสมมูล (equivalence) กับ A X (B X C) หรือกับ (A X B) X C หรือไม่ เพราะอะไร?
14. รูปแบบทั่วไปของการเขียนบรรยายเซตเป็นอย่างไร?
15. เราเขียนอธิบายความสัมพันธ์ (relation) ด้วยเซตได้อย่างไร?

3

ตรรกศาสตร์เชิงเวลา

3.1 ความสำคัญของบทนี้

เนื้อหาในบทนี้แนะนำเกี่ยวกับตรรกศาสตร์เชิงเวลา ซึ่งเป็นการต่อยอดตรรกศาสตร์เชิงประพจน์ โดยเพิ่มการใช้ตัวดำเนินการเชิงเวลาในอนาคต เราใช้ความรู้เดิมเรื่องตรรกศาสตร์เชิงประพจน์มาทำการต่อยอดนิยามวากยสัมพันธ์และความหมายให้ครอบคลุมตัวดำเนินการเชิงเวลา ในบทนี้จะกล่าวเน้นถึงตรรกศาสตร์เชิงเวลาแบบลำดับ (linear temporal logic) เท่านั้น พร้อมยกตัวอย่างให้เข้าใจวิธีการตีความสูตรเชิงเวลาที่ประกอบด้วยประพจน์ และตัวดำเนินการเชิงเวลาด้วยการใช้ตารางค่าความจริง หรือจะการตีความจากรูปลำดับโมเดลแบบเส้นตรงก็ได้ ในบทนี้ ได้เพิ่มส่วนงานวิจัยที่ผู้เขียนมีส่วนร่วมในการแปลงสูตรเชิงเวลาของตรรกศาสตร์เชิงเวลาแบบเมตริกให้ไปเป็นสูตรเชิงเวลาตรรกศาสตร์เชิงเวลาแบบลำดับที่จำลองการทำงานแบบดั้งเดิมได้

3.2 วัตถุประสงค์

- เพื่อให้เข้าใจที่มาของตรรกศาสตร์เชิงเวลา
- เพื่อให้เข้าใจการใช้งานตัวดำเนินการเชิงเวลา
- เพื่อให้ตีความสูตรเชิงเวลาได้

3.3 ตรรกศาสตร์เชิงเวลา

ตรรกศาสตร์เชิงเวลา [6] เป็นระบบตรรกศาสตร์ที่เกี่ยวข้องกับการจำลองพฤติกรรมของระบบและการใช้เหตุผลที่มีเงื่อนไขของเวลามาเกี่ยวข้อง ตัวอย่างของเงื่อนไขเวลาที่มักกล่าวถึงในระบบตรรกศาสตร์นี้ คือ *“เสมอ”* (always), *“ในที่สุด”* (eventually), *“ถัดไป”* (next), *“จนกระทั่ง”* (until) และอื่น ๆ นั่นหมายถึงต่อไปนี้ด้วยระบบตรรกศาสตร์แบบนี้ ประโยคหรือประพจน์ในตรรกศาสตร์เชิงประพจน์เดิมสามารถเขียนให้มีคำที่เป็นเงื่อนไขของเวลารวมอยู่ด้วย โดยใช้ตัวดำเนินการเชิงเวลามาช่วย เช่น ประพจน์ที่มีตัวดำเนินการ *“เสมอ”* คือ *“Somchai always wakes up before 6.00 am.”* หมายถึง คุณสมชายตื่นนอนก่อนหกโมงเช้าเสมอ ไม่มีวันไหนเลยที่ตื่นหกโมงหรือหลังหกโมง จากเดิมที่ไม่สามารถกำหนด

คำว่า *"เสมอ"* ได้ และเราจะต้องมีข้อมูลการตื่นนอนของคุณสมชายทุกวันด้วยเช่นกัน เพื่อจะได้สามารถทวนสอบว่าพฤติกรรมการตื่นนอนเป็นไปตามที่กำหนดหรือไม่ ตรรกศาสตร์เชิงเวลาจึงเป็นประโยชน์มากในกรณีที่ใช้ทวนสอบระบบที่มีสถานะที่เปลี่ยนแปลงไปอย่างต่อเนื่อง

ทำไมจึงต้องมีตรรกศาสตร์เชิงเวลา ถ้าเราพิจารณาระบบตรรกศาสตร์แบบเดิม เช่น ตรรกศาสตร์เชิงประพจน์ หรือตรรกศาสตร์อันดับที่หนึ่ง เป็นต้น เราจะพบว่า การตีค่าความจริงของประโยคที่เป็นประพจน์เดี่ยวหรือประพจน์ประกอบนั้น จะมีผลลัพธ์เป็นค่าความจริงค่าเดียวที่ตำแหน่งของเวลาเดียว t เท่านั้น และเรียกการตีค่าความจริง ณ เวลา t ว่าสถานะ (state) ของระบบที่เราสนใจ ณ เวลา t (ตำราบางเล่มเรียกตรรกศาสตร์เชิงประพจน์ที่พิจารณาได้สถานะหนึ่งเดียวว่า "single fixed state" หรือ "single fixed world") ดังนั้น เมื่อเรานำระบบตรรกศาสตร์ดังกล่าวไปกำหนดคุณลักษณะของระบบ เราทำได้แต่เพียงระบุการทำงานของแบบจำลองระบบที่เวลาใดเวลาหนึ่งเท่านั้น และการตีความค่าความจริงก็ทำได้ตอนหยุดเวลาแต่ละครั้งเท่านั้น เช่น ถ้าระบบมีประพจน์ p, q ที่เราสังเกตได้และเราต้องการหาค่าความจริงของคุณลักษณะ "$p \Longrightarrow q$" ณ เวลา $t=5$ ดั้งนั้น เราต้องหยุดระบบไว้ ณ เวลา $t=5$ และอ่านค่าความจริงของ p, q ออกมาเพื่อตีความคุณลักษณะ "$p \Longrightarrow q$" ทันที ไม่สามารถตอบคำถามว่า ถ้า p เป็นจริงแล้วเวลาผ่านไปสักพัก q จะเป็นจริงตามมาในที่สุด เป็นต้น

การกำหนดคุณลักษณะของระบบที่สามารถระบุเงื่อนไขเวลาในอนาคตได้เป็นสิ่งจำเป็นต่อการทวนสอบในระบบที่สมจริง เช่น เราไม่สามารถตอบคำถามการทวนสอบแบบจำลองสัญญาณไฟจราจรว่า ถ้ามีสัญญาณไฟสีแดงในระบบไฟจราจรแล้ว เมื่อเวลาผ่านสักระยะหนึ่งสัญญาณไฟจะต้องเปลี่ยนไปเป็นสีเขียวได้ในที่สุดและต้องเป็นอย่างนั้นเสมอได้หรือไม่ เป็นต้น

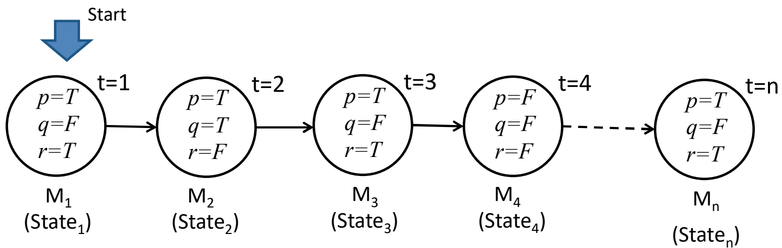
ตรรกศาสตร์เชิงเวลาที่มีการกล่าวถึงมีอยู่หลายประเภท ทั้งนี้ขึ้นอยู่กับการเลือกมาใช้งานให้เหมาะสม ตัวดำเนินการเวลาจะมีให้เลือกใช้แบบเฉพาะเวลาในอดีตเท่านั้นหรือแบบเฉพาะเวลาในอนาคตเท่านั้น หรือแบบรวมทั้งอดีตและอนาคตก็ได้ ตำราเล่มนี้ขอเลือกกล่าวตัวดำเนินการเวลาแบบเฉพาะเวลาในอนาคตเท่านั้น

เพื่อให้เป็นการง่ายในการเข้าใจ จึงขอนำเสนอระบบตรรกศาสตร์เชิงเวลาที่เกิดจากการขยายตรรกศาสตร์เชิงประพจน์เพิ่มเติม โดยการเพิ่มสัญลักษณ์เชิงเวลาในอนาคตเข้ามาเพื่อเป็นตัวดำเนินการเวลา และทำให้เราสามารถสร้างประพจน์ใหม่ที่มีข้อมูลด้านเวลามาเกี่ยวข้องได้ เราเรียกว่าตรรกศาสตร์เชิงเวลาดังกล่าวนี้ว่าตรรกศาสตร์เชิงเวลาประพจน์ (propositional temporal logic : PTL) ตัวอย่างตัวดำเนินการเวลาที่เราสนใจ เช่น ตัวดำเนินการ *"เสมอ"* ใช้

สัญลักษณ์ [], ตัวดำเนินการ *"ในที่สุด"* ใช้สัญลักษณ์ <> เป็นต้น ซึ่งจะกล่าวถึงตัวดำเนินการที่เหลือในภายหลัง

เมื่อเราใช้ตรรกศาสตร์เชิงเวลาประพจน์ในการกำหนดคุณลักษณ์ระบบแล้ว เราจำเป็นต้องกำหนดโมเดลของแบบจำลองของระบบเสียใหม่จากเดิมเรามีโมเดลได้แค่หนึ่งโมเดลเท่านั้น โดยต่อจากนี้ไปเราจะมีชุดของโมเดลที่วางเรียงกันเป็นลำดับ (sequence of models) ได้ในลักษณะที่เป็นเส้นตรงเรียงลำดับจากโมเดลแรกเปลี่ยนผ่านไปยังโมเดลที่สอง และเปลี่ยนผ่านต่อไปอย่างต่อเนื่องกัน ยกตัวอย่างคือ ถ้าเราย้อนกลับไปในการใช้ตรรกศาสตร์เชิงประพจน์กำหนดคุณลักษณะของระบบ กำหนดให้สูตรประพจน์ $A= p\wedge(q\vee r)$ เดิมทีเดียวถ้าระบบของเราชื่อ P สามารถหาโมเดลเดี่ยว M_1 ที่ระบุการตีความค่า $p=T, q=F, r=T$ ที่ทำให้สูตร A มีค่าจริง กล่าวได้ว่า "M_1 *satisfies* A" หรือเขียนด้วย "$M_1 \vDash A$" หรือ "$M_1 \vDash p\wedge(q\vee r)$" นั่นเอง (ในที่นี้ เราเรียกการตีความ $\mathfrak{T}$ ของประพจน์ p ที่ได้ค่าจริงว่าโมเดล M แทน) ในทำนองเดียวกันเราอาจจะหาโมเดลเดี่ยว M_2 จากระบบเดิม P ที่ระบุการตีความค่า $p=T, q=T, r=F$ ทำให้ "$M_2 \vDash p\wedge(q\vee r)$" ได้อีกเช่นกัน

แต่ต่อจากนี้การมองภาพระบบของเราจะเปลี่ยนไป โดยมองระบบที่สนใจ P จะประกอบด้วยเซตของการตีความ M_i หรือเขียนด้วยสัญลักษณ์เต็มว่า $<M, i>$ ที่เรียงลำดับ $M_1, M_2, M_3, \ldots, M_n$ โดย i คือตำแหน่งเวลา และ M_1 คือการตีความ ณ เวลา $t=1$ ขณะที่เวลาถัดไป คือ M_2 ที่มีการตีความ ณ เวลาที่ $t=2$ ไปเรื่อย ๆ จนถึง M_n คือ การตีความ ณ เวลา $t=n$ โดยวาดเป็นรูปลำดับแบบเส้นตรงของสถานะหรือโมเดลได้ แสดงในรูปที่ 3.1



รูปที่ 3.1 โมเดล M_i ที่เรียงลำดับเป็นเส้นตรงของระบบ P

รูปที่ 3.1 เป็นการอธิบายระบบ P ได้จากรูปลำดับแบบเส้นตรงของโมเดลที่แสดงถึงสถานะของระบบและการเปลี่ยนแปลงของสถานะเป็นลำดับ มักจะต้องเริ่มจากสถานะเริ่มต้นไปสู่สถานะถัดไปเรื่อย ๆ เราเทียบแต่ละสถานะในแผนภาพ

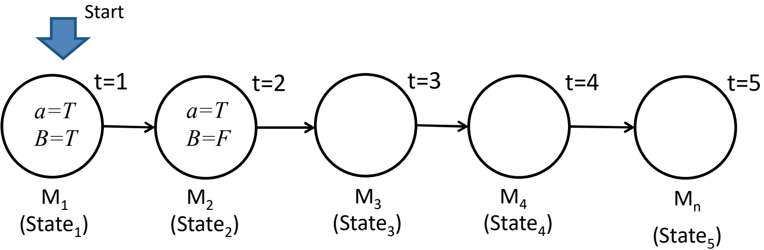
สถานะได้เท่ากับ M_i และการเปลี่ยนสถานะจาก M_i ถัดไปสู่สถาน M_{i+1} นั่นเอง เกิดลำดับของโมเดล M_i อย่างต่อเนื่อง จากนั้นเราสามารถใช้ตรรกศาสตร์เชิงเวลาเขียนอธิบายคุณลักษณะเชิงเวลาที่ต้องการทวนสอบของระบบ P ได้

ต่อไปนี้แสดงตัวอย่างการนำตรรกศาสตร์เชิงเวลาประพจน์ไปประยุกต์ใช้ โดยกำหนดให้มีตัวดำเนินการเวลา *"ถัดไป"* โดยใช้สัญลักษณ์ O มาร่วมในสูตรประพจน์ที่จัดดีแล้ว ซึ่งเราเรียกใหม่ว่า สูตรเชิงเวลา (temporal formula)

ตัวอย่างที่ 3-1: การตีความสูตรเชิงเวลา $(a\wedge b)\wedge O(a\vee b)$ สำหรับระบบ P

กำหนดให้ $X = (a\wedge b)\wedge O(a\vee b)$ โดยเมื่อพิจารณาแล้วจะพบว่าสูตรนี้ประกอบด้วยประพจน์ $(a\wedge b)$ และประพจน์ที่มีตัวดำเนินการ *"ถัดไป"* $O(a\vee b)$ ที่เชื่อมต่อกันด้วยตัวเชื่อม *"และ"*

ให้ระบบ P มีการเปลี่ยนแปลงค่าของ a และ b ตามที่แสดงในรูปที่ 3.2 คือ โมเดล M_1 มีการตีความ $a=T, b=T$ ทำให้ $(a\wedge b)$ เป็นจริง พอถึงโมเดล M_2 มีการตีความ $a=T, b=F$ ทำให้ $(a\vee b)$ เป็นจริง สรุปได้ว่า $(a\wedge b)\wedge O(a\vee b)$ เป็นจริง เพราะ $(a\wedge b)$ เป็นจริง ณ เวลา $t=1$ และต่อมา $(a\vee b)$ เป็นจริง ณ เวลาถัดไป $t=2$ สูตรเชิงเวลา $(a\wedge b)\wedge O(a\vee b)$ จึงเป็นจริงเมื่อใช้ M_1 และ M_2 ตามลำดับ



รูปที่ 3.2 ลำดับของโมเดลของระบบ P ในช่วงค่า t ระหว่าง 1 ถึง 5

อย่างไรก็ตาม ถ้าระบบ P มีพฤติกรรมเป็นเซตของเส้นทาง ρ_k เราจะต้องทำการพิจารณาการตีความของทุกเส้นทางให้ครบ โดยการหาว่า $\rho_k \models X$ โดยที่ $k=1, 2, 3, \ldots, m$ และ X คือ คุณลักษณะเชิงเวลาที่ต้องการทวนสอบ

การตีความตัวดำเนินการเชิงเวลา

การตีความตัวดำเนินการเชิงเวลาที่เราใช้ในบทนี้จะเน้นเฉพาะการตีความของเวลาจากจุดเริ่มต้นและบวกเวลาต่อไปในอนาคตโดยไม่สนใจกรณี

ย้อนเวลากลับจากจุดปัจจุบัน ดังนั้นเวลาทำการตีความในเงื่อนไขของเวลา เราจะเริ่มที่ เวลา t=1 และมองไปข้างหน้าเสมอ แต่ละสัญลักษณ์ของตัวดำเนินการเวลาจะมีความหมายดังต่อไปนี้

กำหนดให้สูตรเชิงเวลา Φ, Ψ ใด ๆ

- สูตรใหม่ OΦ หมายถึงการตีความที่พบว่า Φ จะมีค่าจริงในสถานะที่ตำแหน่งเวลาถัดไปเสมอ สถานะปัจจุบันเป็นอย่างไรไม่สนใจ
- สูตรใหม่ []Φ หมายถึงการตีความที่พบว่า Φ จะมีค่าจริงเสมอนับตั้งแต่สถานะตำแหน่งเวลาปัจจุบันไปเรื่อยๆ จนถึงสถานะสุดท้าย ห้ามมีค่าเท็จปรากฏเลย
- สูตรใหม่ <>Φ หมายถึงการตีความที่พบว่า Φ จะมีค่าจริงครั้งเดียวในที่สุด โดยอาจจะที่สถานะตำแหน่งเวลาปัจจุบันหรือสถานะใดก็ได้ หรือสถานะสุดท้าย
- สูตรใหม่ ΦUΨ หมายถึงการตีความที่พบว่า Φ จะมีค่าจริงเสมอนับตั้งแต่สถานะตำแหน่งเวลาปัจจุบันไปเรื่อยๆ จนถึงสถานะใดๆ ที่ Ψ มีค่าจริงถัดมาในที่สุด
- สูตรใหม่ ΦWΨ หมายถึงการตีความที่พบว่า Φ จะมีค่าจริงเสมอนับตั้งแต่สถานะตำแหน่งเวลาปัจจุบันไปเรื่อยๆ จนถึงสถานะใด ๆ ที่ Ψ มีค่าจริงถัดมาในที่สุด หรือสถานะสุดท้ายมาถึงก่อนโดยที่ Ψ ยังไม่มีค่าจริงก็ได้

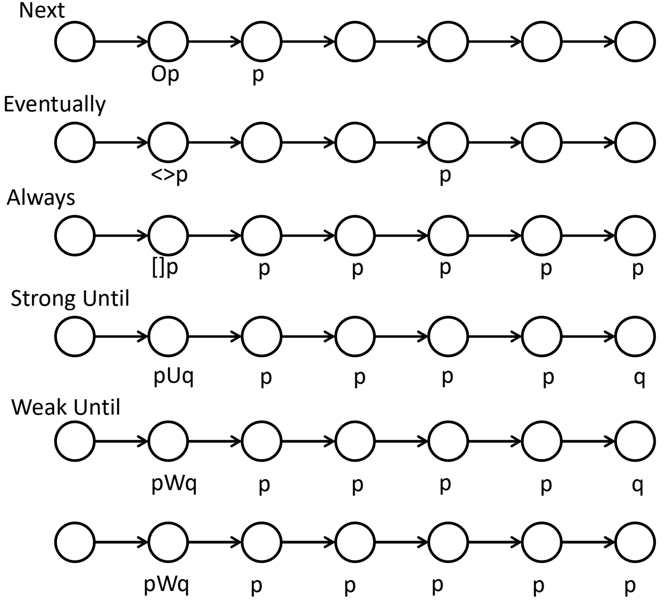
(เราอ่านอักขระ Φ ว่า Phi ฟี หรือ ไฟ และ Ψ อ่านว่า Psi ไซ)

จากคำอธิบายการตีความข้างต้น เราสามารถอธิบายความหมายต่างๆ ด้วยรูปภาพลำดับโมเดลแบบเส้นตรงของแต่ละตัวดำเนินการได้ดังแสดงในรูปที่ 3.3

รูปลำดับโมเดลแบบเส้นตรงของการทำงานระบบ P ใด ๆ ที่ใช้อธิบายการตีความของตัวดำเนินการเวลามีลักษณะเป็นเส้นทางของโมเดลหรือสถานะของระบบ P ณ เวลาใด ๆ เริ่มจากเครื่องหมายวงกลมซึ่งหมายถึงโมเดล ณ เวลา $t=1$ และมีเส้นลูกศรชี้ต่อไปยังโมเดล ณ เวลา $t=2$ และถัดไปเรื่อย ๆ เส้นลูกศรแสดงถึงลำดับของการเปลี่ยนสถานะไปยังเวลาถัดไป แต่ละโมเดลจะกำกับด้วยอักษร p, q ที่เป็นสูตรเชิงเวลาที่มีค่าจริงในโมเดลนั้น ถ้าสูตรเชิงเวลาใดมีค่าเท็จก็จะไม่เขียนกำกับโมเดล ณ เวลานั้น ๆ

จากรูปที่ 3.3 สำหรับการตีความสูตรเชิงเวลา Op คือ ตัวดำเนินการแบบ *"ถัดไป"* มีความหมายว่าโมเดล ณ เวลาถัดไป p จะมีค่าจริง ซึ่งจะทำให้การตีความ Op ณ เวลาปัจจุบันจะมีค่าจริงได้ กรณีการตีความสูตรเชิงเวลา <>p คือ

ตัวดำเนินการแบบ *"ในที่สุด"* มีความหมายว่าโมเดล ณ เวลาใดๆ นับตั้งแต่ปัจจุบันถึงในอนาคต p จะมีค่าจริงในที่สุด ทำให้โมเดล ณ เวลาปัจจุบัน <>p จะมีค่าจริงได้เช่นกัน สำหรับกรณีการตีความสูตรเชิงเวลา []p คือตัวดำเนินการแบบ *"เสมอ"* มีความหมายว่าโมเดลทุกเวลานับตั้งแต่ปัจจุบันถึงในอนาคต p จะมีค่าจริงเสมอ ทำให้โมเดล ณ เวลาปัจจุบัน []p จะมีค่าจริงได้ กรณีการตีความสูตรเชิงเวลา pUq คือตัวดำเนินการแบบ "จนกระทั่ง" มีความหมายว่าโมเดล ณ เวลาใดๆ นับตั้งแต่ปัจจุบันถึงในอนาคต p จะมีค่าจริงเสมอไปจนกระทั่ง q มีค่าจริงในที่สุด ทำให้โมเดล ณ เวลาปัจจุบัน pUq จะมีค่าจริงได้



รูปที่ 3.3 แบบจำลองเวลาแบบเส้นตรงของการทำงานระบบ P ใดๆ

ส่วนกรณีการตีความสูตรเชิงเวลา pWq คือตัวดำเนินการแบบ *"จนกระทั่งแบบอ่อน"* มีความหมายแบบเดียวกันกับ pUq และเพิ่มเติมว่าโมเดล ณ เวลาใดๆ นับตั้งแต่ปัจจุบันถึงในอนาคต p จะมีค่าจริงเสมอไปจนกระทั่งจบการทำงานในที่สุดโดย q ยังไม่มีค่าจริงก็ได้ตราบเท่าที่ p ยังเป็นค่าจริงอยู่ ทำให้โมเดล ณ เวลาปัจจุบัน pWq จะมีค่าจริงได้

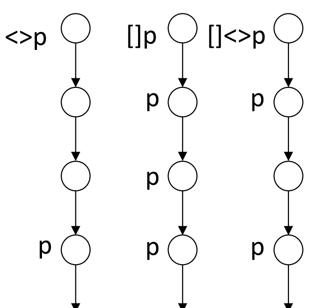
ประเภทของตรรกศาสตร์เชิงเวลา

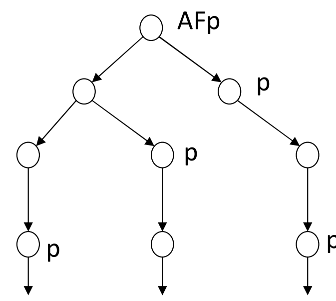
อันที่จริงตรรกศาสตร์เชิงเวลาแบ่งเป็นสองประเภทตามโครงสร้างของเส้นทางของเวลา คือ ตรรกศาสตร์เชิงเวลาแบบเส้นตรงและตรรกศาสตร์ต้นไม้

การคำนวณ (computation tree logic) ซึ่งมีความซับซ้อนมากกว่า เราสามารถสังเกตได้ชัดเจนว่าตรรกศาสตร์เชิงเวลาแบบเส้นตรงจะมีโครงสร้างเป็นเส้นทางลำดับสถานะหรือโมเดลแบบเส้นตรง โดยเริ่มจากสถานะหรือโมเดลแรก ณ เวลา $t=1$ จากนั้นเชื่อมต่อถึงสถานะถัดไป ณ เวลา $t=2$ และเชื่อมต่อไปเรื่อย ๆ จนจบการทำงานถ้าระบบนั้นมีจุดสิ้นสุดของเส้นทางการทำงาน รูปเส้นทางลำดับสถานะแบบเส้นตรงแสดงเป็นแต่ละเส้นในรูปด้านซ้ายมือของรูปที่ 3.4

สำหรับตรรกศาสตร์ต้นไม้การคำนวณจะมีโครงสร้างเป็นต้นไม้ โดยเริ่มจากสถานะหรือโมเดลแรก ณ เวลา $t=1$ เช่นกันที่ตำแหน่งรากของต้นไม้เสมอ จากนั้น ณ เวลา $t=2$ ถ้ามีทางเลือกหรือทางแยกมากกว่าหนึ่งสถานะก็จะเป็นการเชื่อมต่อจากสถานะแรกที่ตำแหน่งรากมาสู่สถานะที่ความลึกระดับที่หนึ่ง จะต้องมีเส้นเชื่อมไปถึงทุกสถานะที่ความลึกระดับที่หนึ่งให้ครบกรณีที่มีหลายสถานะในความลึกระดับเดียวกันนี้ จากรูปที่ 3.4 รูปต้นไม้ด้านขวาพบว่าในความลึกต้นไม้ระดับที่หนึ่งมีสองสถานะแยกมาทางซ้ายและทางขวา จากนั้นให้พิจารณาต่อจากระดับที่หนึ่งลึกลงไปถึงระดับที่สอง จะพบว่าสถานะด้านซ้ายก็จะแยกได้เป็นสองสถานะในระดับที่สอง ในขณะที่สถานะด้านขวาจะไปสู่สถานะเดียวในระดับลึกที่สอง เป็นต้น ต้นไม้นี้แสดงถึงทางแยกของสถานะที่เป็นไปได้ทั้งหมด

เราควรเลือกใช้ตรรกศาสตร์เชิงเวลาแบบเส้นตรงและตรรกศาสตร์ต้นไม้การคำนวณให้เหมาะสมกับการกำหนดคุณลักษณะที่ต้องการทวนสอบ เนื่องจากมีบางคุณลักษณะที่ตรรกศาสตร์ต้นไม้การคำนวณอธิบายได้แต่ตรรกศาสตร์เชิงเวลาแบบเส้นตรงอธิบายไม่ได้





a) Linear Temporal Logic b) Computation Tree Logic

รูปที่ 3.4 รูปเส้นทางของสถานะของตรรกศาสตร์เชิงเวลาแบบลำดับและแบบต้นไม้การคำนวณ

วากยสัมพันธ์และความหมาย

นิยามที่ 3-1: สัญลักษณ์ที่ใช้ในวากยสัมพันธ์ของตรรกศาสตร์เชิงเวลาประพจน์ [6]

ภาษาที่ใช้เขียนประโยคในระบบตรรกศาสตร์รวมถึงการเชื่อมประโยค และเพิ่มตัวดำเนินการเวลาเข้าร่วมด้วยจะประกอบด้วยสัญลักษณ์ที่กำหนดไว้ต่อไปนี้

- สัญลักษณ์ที่เป็นสมาชิกของเซต *PROP* ซึ่งเป็นกลุ่มของตัวแปรประพจน์ที่แต่ละตัวแสดงถึงประพจน์เดี่ยวที่เล็กที่สุดไม่สามารถแบ่งเป็นประโยคย่อยอีกได้ เช่น p, q, r เป็นประพจน์ที่เล็กที่สุดเป็นต้น
- สัญลักษณ์ที่เป็นสมาชิกของเซตของตัวเชื่อมประพจน์ที่แต่ละตัวใช้เชื่อมหรือตกแต่งประโยคให้เป็นประพจน์ประกอบ ตัวเชื่อมแบบนาวลารี คือ $\{T, F\}$ โดยสัญลักษณ์ T ใช้แทนค่าจริง และสัญลักษณ์ F ใช้แทนค่าเท็จ ตัวเชื่อมแบบเอกภาค คือ $\{\neg\}$ โดยสัญลักษณ์ $\neg$ ใช้เป็นการทำนิเสธหรือการหาค่าตรงข้าม ตัวเชื่อมแบบทวิภาค คือ $\{\wedge, \vee, \Rightarrow, \Leftrightarrow\}$ โดยสัญลักษณ์ $\wedge$ ใช้แทน *"และ"*, สัญลักษณ์ $\vee$ ใช้แทน *"หรือ"*, สัญลักษณ์ $\Rightarrow$ ใช้แทน *"ถ้า..แล้ว"*, และ สัญลักษณ์ $\Leftrightarrow$ ใช้แทน *"ก็ต่อเมื่อ"*
- สัญลักษณ์ ( ) ใช้แทนวงเล็บเพื่อใช้ในการจัดกลุ่มและลดความกำกวมของประพจน์
- ตัวดำเนินการเวลา คือ ตัวดำเนินการ *"ถัดไป"* โดยใช้สัญลักษณ์ O, ตัวดำเนินการ *"ในที่สุด"* โดยใช้สัญลักษณ์ <>, ตัวดำเนินการ *"เสมอ"* โดยใช้สัญลักษณ์ [], ตัวดำเนินการ *"จนกระทั่ง"* โดยใช้สัญลักษณ์ U, ตัวดำเนินการ *"จนกระทั่งแบบอ่อน"* โดยใช้สัญลักษณ์ W, ตัวดำเนินการ *"จุดเริ่ม"* โดยใช้สัญลักษณ์ *start*

นิยามที่ 3-2: สูตรเชิงเวลาที่จัดดีแล้วในวากยสัมพันธ์ของตรรกศาสตร์เชิงเวลาประพจน์ [6]

วิธีการเขียนประพจน์ที่มีตัวดำเนินการเวลาร่วมอยู่ด้วยอย่างถูกต้องจะต้องใช้สัญลักษณ์ที่กำหนดไว้ดังกล่าวข้างต้นเท่านั้น และเมื่อมีการเชื่อมประพจน์เดิมเข้าด้วยกันหรือมีตัวดำเนินการเวลาร่วมด้วยจะต้องอยู่ในรูปแบบที่ถูกต้องเหมาะสมเรียกว่า สูตรเชิงเวลาที่จัดดีแล้ว (well-formed temporal formula: WFF_t) ด้วยกฎต่อไปนี้

- ตัวแปรประพจน์ถือว่าเป็น WFF_t
- ตัวเชื่อมแบบนาวลารีที่เป็นค่า T หรือ F ถือว่าเป็น WFF_t
- ถ้า Φ และ Ψ เป็น WFF_t แล้ว $\neg\Phi$, $(\Phi \wedge \Psi)$, $(\Phi \vee \Psi)$, $(\Phi \Rightarrow \Psi)$, $(\Phi \Leftrightarrow \Psi)$, (Φ), $<>\Phi$, $[]\Phi$, $\mathrm{O}\Phi$, $(\Phi \cup \Psi)$, $(\Phi \,\mathrm{W}\, \Psi)$ ต่างก็ถือว่าเป็น WFF_t เช่นกัน

ตัวอย่างที่ 3-2: การเขียนสูตรเชิงเวลาแบบ WFF_t

กำหนดให้ $PROP = \{p, q, r\}$ ประพจน์ต่อไปนี้ถือว่าเป็น WFF_t คือ T, $T \wedge <>p$, $(<>p \vee []q)$, $([](p \wedge q) \Rightarrow <>r)$ เป็นต้น

นิยามที่ 3-3: ส่วนความหมายของตรรกศาสตร์เชิงเวลาประพจน์ [6]

ส่วนความหมายของตรรกศาสตร์เชิงเวลาประพจน์ ซึ่งจะนิยามวิธีการตีความสูตรเชิงเวลาที่เขียนขึ้นมาว่ามีค่าความจริงเป็นค่าจริงหรือค่าเท็จ สูตรเชิงเวลาก็คือ ประพจน์ที่มีตัวดำเนินการเวลาร่วมอยู่ด้วยนั่นเอง โดยการตีความจะเป็นไปตามนิยามที่กำหนดไว้ข้างล่างนี้

$<M, i> \models p$	iff $p \in PROP$ and $\mathfrak{T}_{atomic}(p) = T$
$<M, i> \models \neg\Phi$	iff not $<M, i> \models \Phi$
$<M, i> \models \Phi \wedge \Psi$	iff $<M, i> \models \Phi$ and $<M, i> \models \Psi$
$<M, i> \models \Phi \vee \Psi$	iff $<M, i> \models \Phi$ or $<M, i> \models \Psi$
$<M, i> \models \Phi \Rightarrow \Psi$	iff if $<M, i> \models \Phi$ then $<M, i> \models \Psi$
$<M, i> \models start$	iff $(i=1)$
$<M, i> \models \mathrm{O}\Phi$	iff $<M, i+1> \models \Phi$
$<M, i> \models <>\Phi$	iff $\exists j \geq i : <M, j> \models \Phi$
$<M, i> \models []\Phi$	iff $\forall j \geq i : <M, j> \models \Phi$
$<M, i> \models \Phi \cup \Psi$	iff $\exists j \geq i : <M, j> \models \Psi$ and $\forall\, i \leq k < j : <M, k> \models \Phi$

สมการ $<M, i> \models \Phi$ มีความหมายว่า "$<M, i>$ *satisfies* Φ" โดย $<M, i>$ หมายถึงโมเดล Mi หรือสถานะ ณ เวลา $t=i$ และ Φ คือสูตรเชิงเวลาที่เราใช้ตรวจสอบ ผลลัพธ์ที่เราสนใจคือการที่ $<M, i>$ นั้น satisfies Φ ก็ต่อเมื่อ Mi หรือสถานะ ณ เวลา $t=i$ มีการกำหนดให้ค่าตัวแปรประพจน์และพิจารณาตามเงื่อนไขตัวดำเนินการเวลาแล้วทำให้สูตร Φ มีค่าจริง

การที่เราจะรู้ว่า $<M, i> \models \Phi$ โดยที่ Φ เป็น WFF_t ใดๆ นั้นเราต้องใช้นิยามความหมายที่กล่าวมาแล้วข้างต้นพิจารณาประกอบ ดังนี้คือ $<M, i> \models p$ ก็ต่อเมื่อ p เป็นสมาชิกของ *PROP* ที่มีค่าจริง (เราสนใจเฉพาะค่าจริง ถ้าค่าเท็จเราจะถือว่าไม่ satisfy) และการตีความหมายส่วนที่เหลือก็ดูจากเงื่อนไขหลัง *iff* ซึ่งก็จะเป็นตัวกำหนดให้สรุปได้ว่า $<M, i>$ เป็นการทำงานของระบบที่ผ่านการทวนสอบได้

ตัวอย่างที่ 3-3 การเขียนสูตรเชิงเวลา

กำหนดให้สูตรเชิงเวลา

$send_msg \Rightarrow \Diamond receive_msg$

จะหมายถึง ถ้าโมเดล ณ เวลา $t=n$ สัญญาณ *send_msg* มีค่าจริงแล้ว โมเดลในอนาคตที่เวลา $t=n+k$ โดยที่ $k>=0$ สัญญาณ *receive_msg* มีค่าจริงในที่สุด จะทำให้การตีความสูตรเชิงเวลานี้ที่ t อยู่ในช่วงเวลา n ถึง $n+k$ มีค่าจริงด้วยเช่นกัน ถ้าอธิบายด้วยประโยคทั่วไป คือ ถ้ามีสัญญาณ *send_msg* เมื่อไรแล้วจะต้องมีสัญญาณ *receive_msg* ตามมาในที่สุด

กำหนดให้สูตรเชิงเวลา

$\Box((\neg have_passport \vee \neg have_ticket) \Rightarrow \bigcirc \neg board_flight)$

จะหมายถึง ถ้าโมเดล ณ เวลา $t=n$ *have_passport* หรือ *have_ticket* มีค่าเท็จแล้ว โมเดลในอนาคตที่เวลา $t=n+1$ *board_flight* จะมีค่าเท็จเสมอ จะทำให้การตีความสูตรเชิงเวลานี้ที่ t อยู่ในช่วงเวลา n ถึง $n+1$ มีค่าจริงด้วยเช่นกัน ถ้าอธิบายด้วยประโยคทั่วไปคือ ถ้าไม่มีหนังสือเดินทางหรือไม่มีตั๋วเครื่องบินอย่างใดอย่างหนึ่งแล้ว จะต้องไม่สามารถขึ้นเครื่องบินได้ และจะเป็นอย่างนี้เสมอ

กำหนดให้สูตรเชิงเวลา

$born \Rightarrow living \cup dead$

จะหมายถึง ถ้าโมเดล ณ เวลา $t=n$ *born* มีค่าจริงแล้ว โมเดลในอนาคตที่เวลา $t=n+k$ โดยที่ $k >= 0$ *living* มีค่าจริงเสมอจนกระทั่งที่เวลา $t=n+k+1$ *dead* มีค่าจริงในที่สุด จะทำให้การตีความสูตรเชิงเวลานี้ที่ t อยู่ในช่วงเวลา n ถึง $n+k+1$ มีค่าจริงด้วยเช่นกัน ถ้าอธิบายด้วยประโยคทั่วไป คือ ถ้าคนหนึ่งเกิดแล้ว เขาจะมีชีวิตคงอยู่ไปเรื่อยเสมอจนกระทั่งเขาตายในที่สุด

การตีความโดยใช้ตารางค่าความจริง

เพื่อให้เราสามารถเห็นวิธีการตีความสูตรเชิงเวลาด้วยวิธีการใช้ตารางค่าความจริง โดยมีการแจกแจงค่าความจริงของประพจน์เดี่ยวและสูตรต่างๆ ที่นำประพจน์เดี่ยวนั้นไปประกอบกันอย่างเป็นขั้นตอน

M_1 (send_msg, receive_msg)

t	1	2	3	4	5	6	7	8
send_msg	F	T	F	F	F	T	F	F
receive_msg	F	F	F	T	F	F	F	T
<>receive_msg	T	T	T	T	T	T	T	T
send_msg ⇒ <>receive_msg	T	T	T	T	T	T	T	T
[](send_msg ⇒ <>receive_msg)	T	T	T	T	T	T	T	T

$M_1 \models <>receive_msg$

$M_1 \models [](send_msg \Rightarrow <>receive_msg)$

รูปที่ 3.5 ตารางค่าความจริงแสดงการตีความกรณีจริงของสูตรเชิงเวลา

กำหนดให้สูตรเชิงเวลา Φ=[](*send_msg* $\Rightarrow$ <>*receive_msg*) และเส้นทางของโมเดล M_1 ที่มีค่าความจริงของประพจน์เดี่ยว *send_msg*, *receive_msg* และสูตร ณ เวลา t ระหว่าง *1* ถึง *8* แสดงในรูปที่ 3.5 เราพบว่า "M_1 |= Φ" เพราะบรรทัดสุดท้ายในการตีความแสดงว่าสูตร Φ ให้ค่าจริงทุกกรณีในช่วงเวลาที่เราสนใจหรือเป็นสัจนิรันดร์ในช่วงเวลานี้

M_2 (send_msg, receive_msg)

t	1	2	3	4	5	6	7	8
send_msg	F	T	F	F	F	T	F	F
receive_msg	F	F	F	T	F	F	F	F
<>receive_msg	T	T	T	T	F	F	F	F
send_msg ⇒ <>receive_msg	T	T	T	T	T	F	T	T
[](send_msg ⇒ <>receive_msg)	F	F	F	F	F	F	F	F

$M_2 \models <>receive_msg$

$M_2 \not\models [](send_msg \Rightarrow <>receive_msg)$

รูปที่ 3.6 ตารางค่าความจริงแสดงการตีความกรณีเท็จของสูตรเชิงเวลา

กรณีตรงกันข้าม กำหนดให้สูตรเชิงเวลา Φ = [](*send_msg* $\Rightarrow$ <>*receive_msg*) และเส้นทางของโมเดล M_2 ที่มีค่าความจริงของประพจน์เดี่ยวและสูตร ณ เวลา t ระหว่าง *1* ถึง *8* แสดงในรูปที่ 3.6 เราพบว่า "$M_2 \not\models \Phi$" เพราะบรรทัดสุดท้ายในการตีความแสดงว่าสูตร Φ ให้ค่าเท็จทุกกรณีในช่วงเวลา

ที่เราสนใจนี้ สาเหตุจากสูตรย่อย (*send_msg* ⟹ <>*receive_msg*) ไม่เป็นค่าจริงเสมอนั่นเอง และรูปที่ 3.7 แสดงการตีความช่วงค่าของตัวแปรสำหรับตัวดำเนินการ *"จนกระทั่ง"*

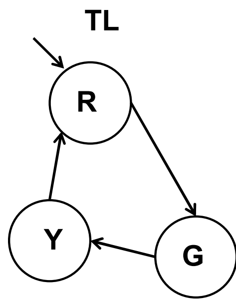
t	1	2	3	4	5	6	7	…
x	1	2	3	4	5	6	7	…
$3 \leq x \leq 5$	F	F	T	T	T	F	F	…
$x = 6$	F	F	F	F	F	T	F	…
$(3 \leq x \leq 5)\ U\ (x = 6)$	F	F	T	T	T	T	F	…

รูปที่ 3.7 ตารางค่าความจริงแสดงการตีความตัวดำเนินการเชิงเวลา U [16]

ตัวอย่างที่ 3-4 การเขียนคุณลักษณะของระบบด้วยสูตรเชิงเวลา

กำหนดให้ระบบสัญญาณไฟจราจรชื่อ *TL* ที่มีพฤติกรรมการทำงานตามแผนภาพสถานะในรูปที่ 3.8 โดยเสาไฟสัญญาณมีไฟสามดวงและจะเริ่มต้นเปิดไฟแดง (สถานะ *R*) ไฟเขียว (สถานะ *G*) ไฟเหลือง (สถานะ *Y*) และกลับมาเปิดไฟวนซ้ำไปไม่สิ้นสุด

เรากำหนดให้เซตประพจน์เดี่ยว *AP* = {*red, green, yellow*} โดยที่ *red* เป็นประพจน์เดี่ยวที่มีค่าจริงถ้ามีการเปิดสัญญาณไฟแดงสว่างขึ้น, *green* เป็นประพจน์เดี่ยวที่มีค่าจริงถ้ามีการเปิดสัญญาณไฟเขียวสว่างขึ้น, และ *yellow* เป็นประพจน์เดี่ยวที่มีค่าจริงถ้ามีการเปิดสัญญาณไฟเหลืองสว่างขึ้น เนื่องจากเราสนใจเฉพาะการสังเกตสีของไฟสัญญาณเท่านั้น ประพจน์เดี่ยวที่เราออกแบบไว้จึงเกี่ยวกับสีของไฟสัญญาณที่ติดสว่างเท่านั้น



รูปที่ 3.8 แผนภาพสถานะของพฤติกรรมการทำงานของเสาไฟสัญญาณจราจร

ตัวอย่างคุณลักษณะที่ดีของระบบ *TL* คือ *“สัญญาณไฟเขียวควรได้รับการเปิดสว่างได้เสมอ” “เมื่อมีสัญญาณไฟทั้งสามไม่ควรดับพร้อมกันเสมอ” “เมื่อมีสัญญาณไฟเขียวแล้ว ถัดไปต้องเป็นสัญญาณไฟเหลืองจะตามมา ก่อนเป็นสัญญาณไฟแดงในที่สุด” “สัญญาณไฟแดงสว่างพร้อมสัญญาณไฟเขียวไม่ได้เด็ดขาด”* เป็นต้น

เราเขียนสูตรเชิงเวลาของลักษณะของตัวอย่างระบบข้างต้นได้ดังนี้

- “สัญญาณไฟเขียวควรได้รับการเปิดสว่างได้เสมอ”

  $[]<>green$

- “เมื่อมีสัญญาณไฟทั้งสามไม่ควรดับพร้อมกันเสมอ”

  $[]\ \neg(\neg red \wedge \neg green \wedge \neg yellow)$

- “เมื่อมีสัญญาณไฟเขียวแล้ว ถัดไปต้องเป็นสัญญาณไฟเหลืองจะตามมา ก่อนเป็นสัญญาณไฟแดงในที่สุด”

  $[](green \Rightarrow \mathrm{O}(yellow\ \mathrm{U}\ red))$

- “สัญญาณไฟแดงสว่างพร้อมสัญญาณไฟเขียวไม่ได้เด็ดขาด”

  $[]\ \neg(red \wedge green)$

ความสมมูลของสูตรเชิงเวลาแบบลำดับ

กำหนดให้สูตรเชิงเวลาแบบลำดับ Φ เราสามารถหาสูตรที่สมมูลกัน Ψ โดยที่ $\Phi \equiv \Psi$ ได้ด้วยความสมมูลของสูตรเชิงเวลาแบบลำดับ (equivalence of LTL formula) [6] ทั้งนี้เพื่อนำไปใช้ในการแปลงสูตรเชิงเวลาแบบลำดับให้อยู่ในรูปแบบต่างๆ เช่น การหานิเสธของสูตรเชิงเวลา ซึ่งจำเป็นต้องนำมาใช้ในการทวนสอบ เป็นต้น

กฎ *duality*

$$\neg \mathrm{O}\Phi \equiv \mathrm{O}\neg\Phi$$

$$\neg<>\Phi \equiv []\neg\Phi$$

$$\neg[]\Phi \equiv <>\neg\Phi$$

กฎ *idempotency*

$$<><>\Phi \equiv <>\Phi$$

$$[][]\Phi \equiv []\Phi$$

$$\Phi \mathrm{U}(\Phi \mathrm{U}\Psi) \equiv \Phi \mathrm{U}\Psi$$

$$(\Phi \mathrm{U}\Psi)\mathrm{U}\Psi \equiv \Phi \mathrm{U}\Psi$$

กฎ *absorption*

$$<>[]<>[]\Phi \equiv []<>\Phi$$

$$[]<>[]<>\Phi \equiv <>[]\Phi$$

กฎ *expansion*

$$\Phi U \Psi \equiv \Psi \vee (\Phi \wedge O(\Phi U \Psi))$$

$$<>\Psi \equiv \Psi \vee O<>\Psi$$

$$[]\Psi \equiv \Psi \wedge O[]\Psi$$

กฎ *distribution*

$$O(\Phi U \Psi) \equiv (O\Phi)U(O\Psi)$$

$$<>(\Phi \vee \Psi) \equiv <>\Phi \vee <>\Psi$$

$$[](\Phi \wedge \Psi) \equiv []\Phi \wedge []\Psi$$

กฎ duality น่าจะนำไปใช้งานได้กับการหานิเสธของสูตรเชิงเวลา ในการทวนสอบที่จะกล่าวต่อไป มีความจำเป็นที่ต้องหานิเสธของสูตรเชิงเวลาด้วยเช่นกัน และกฎในกลุ่มอื่นน่าจะมีส่วนช่วยในการแปลงสูตรเชิงเวลาให้ซับซ้อนน้อยลง เช่น กฎ absorption อาจจะช่วยสัญลักษณ์เชิงเวลาลงได้ถ้ามีการกำหนดซ้ำกันมากจนเกินไป เป็นต้น

3.4 การแปลงเอ็มทีแอลไปเป็นแอลทีแอล

ระบบตรรกศาสตร์เชิงเวลาแบบเส้นตรง (linear temporal logic: LTL) หรือเรียกสั้นๆ ว่าแอลทีแอล แม้ว่าจะมีการกำหนดตัวดำเนินการเวลาในอนาคตไว้แล้ว แต่ในบางครั้งเราต้องการระบุช่วงเวลาที่เจาะจงคือช่วงเวลาในอนาคตตำแหน่งเวลาที่ *5* ถึง *10* นับจากปัจจุบัน จึงมีตรรกศาสตร์ระบบใหม่ที่ให้เราเพิ่มช่วงเวลา (time interval) โดยใช้สัญลักษณ์ *(5,10)* กำกับตัวดำเนินการเวลาของแอลทีแอลเดิมได้ และเราเรียกระบบตรรกศาสตร์ระบบใหม่นี้ว่า ตรรกศาสตร์เชิงเวลาแบบเมตริก (metric temporal logic: MTL) หรือเรียกสั้นๆ ว่าเอ็มทีแอลนั่นเอง

ปกติเรามักจะหาเครื่องมือทวนสอบที่สนับสนุนแอลทีแอลได้ไม่ยากนัก เช่น เครื่องมือสปิน เป็นต้น แต่เครื่องมือที่สนับสนุนแอลทีแอลอาจจะไม่สนับสนุนเอ็มทีแอลด้วย ดังนั้นถ้ากำหนดให้และจำเป็นต้องใช้สูตรเชิงเวลาแบบเอ็มทีแอลแล้ว เรามีความจำเป็นต้องนำเอ็มทีแอลมาแปลงให้อยู่ในรูปแบบแอลทีแอลก่อนนำมาใช้งานกับเครื่องมือที่ไม่สนับสนุนการกำหนดช่วงเวลา

การแปลงเอ็มทีแอลไปเป็นแอลทีแอลทำได้อย่างไร ในที่นี้ขอกล่าวถึงงานวิจัย [17] ที่นำเสนอกรรมวิธีในการแปลงเอ็มทีแอลไปเป็นแอลทีแอล โดยทำ

การนิยามสัญญาณนาฬิกาจริงและวิธีการนับจังหวะเวลาตลอดจนการตรวจสอบช่วงเวลาขึ้นมา และนำมาติดตั้งเพิ่มในระบบแบบแอลทีแอลเดิมเพื่อจำลองการนับช่วงเวลาอย่างได้ผล โดยงานวิจัยนี้ได้นำเสนอชุดคำสั่งภาษาโพรเมลาที่เป็นแผ่นแบบและวิธีการแปลง ไว้เป็นหกส่วนดังต่อไปนี้

- ส่วนนาฬิกานับเวลาจริง (real-value clock module)
- ส่วนควบคุมกำหนดการ (scheduling controller module)
- ส่วนประกาศตัวแปรเหตุการณ์เอ็มทีแอล (MTL event variables module)
- ส่วนกำหนดค่าตั้งต้นหลักของเครื่องประมวลเอ็มทีแอล (MTL engine initialization)
- ส่วนนับช่วงเวลาที่ระบุในตัวดำเนินการเอ็มทีแอล
- แผ่นแบบที่ใช้ในการแปลงตัวดำเนินการเอ็มทีแอล

ส่วนนาฬิกานับเวลาจริง

ส่วนนี้เป็นโพรเมลาที่ทำหน้าที่แทนนาฬิกานับเวลาจริง โดยตัวแปรนาฬิกานับเวลาชื่อ *GlobalClock* ให้นับเวลาเป็นหน่วย *Tick* และนับเวลาแบบมอดูโล *50* (กล่าวคือ ให้นับจาก *Tick* ที่ *0* ไปถึง *Tick* ที่ *49* แล้ววนกลับมาเริ่มนับใหม่ นับวนซ้ำไปเรื่อยๆ) แต่ละหน่วย *Tick* แทนเวลาจริงคือ *1* มิลลิวินาที นั่นหมายถึงรอบการนับเวลาจะนานสุดถึง *50* มิลลิวินาทีเพื่อนำมาใช้จำลองเวลาที่ชิ้นส่วนอุปกรณ์ใดในวงจรที่ต้องการระยะเวลาในการทำงานได้สูงสุดไม่เกิน *50* มิลลิวินาที

ตัวแปรอื่นคือ ตัวแปรชื่อ *ClockState* ซึ่งทำหน้าที่ระบุสถานะของช่วงสัญญาณนาฬิกา ที่จะเปิดโอกาสให้อุปกรณ์เข้าสู่โหมดทำงานหรือโหมดรอ โดยถ้าตัวแปร *ClockState* มีค่าเป็น *Executing* อุปกรณ์ที่มีความพร้อมที่มีทรัพยากรจัดสรรมาได้แล้วจะเริ่มลงมือทำงานได้ หรือถ้าตัวแปร *ClockState* มีค่าเป็น *Scheduling* และตัวแปร *Resource* ที่แสดงถึงทรัพยากรตัวอย่างมีค่าเป็น *_IDLE_* ก็จะเป็นโอกาสที่อุปกรณ์จะไปขอรับทรัพยากรนั้นๆ ได้ เพื่อเตรียมความพร้อมในการทำงานก่อนถึงจังหวะ *ClockState* กลับมาเป็น *Executing* อีกครั้ง โดยตัวอย่างโพรเมลาที่ทำหน้าที่นี้แสดงในรูปที่ 3.9

```
1 mtype = {C_Execute,C_Schedule,_IDLE_}
2 byte ClockState = C_Execute;
3 byte Resource = _IDLE_;
4 int GlobalClock = 1;
5 #define TotalTickPerLoop 50
6
7 active proctype Ticking(){
8 do ::((ClockState == C_Schedule)&& timeout)->
9        ClockState = C_Execute;
10     ::((ClockState == C_Execute) && timeout)->MTLEngine();
11 printf ("== TICK :%d Resource[%e]\n",GlobalClock,Resource);
12 GlobalClock++;
13 If  :: (GlobalClock > (TotalTickPerLoop)) -> GlobalClock = 1;
14     :: else->skip;
15 fi;
16 Resource = _IDLE_;
17 ClockState = C_Schedule;
18 od;}
```

รูปที่ 3.9 ตัวอย่างโพรเมลาส่วนนาฬิกานับเวลาจริงแบบมอดูโล 50 [17]

ส่วนควบคุมกำหนดการ

เป็นส่วนโพรเมลาที่ทำหน้าที่ควบคุมกำหนดการทำงานของอุปกรณ์ รูปที่ 3.10 เป็นตัวอย่างอุปกรณ์ที่มีงานอยู่สามงาน (task) ที่ต้องทำคือ งานชื่อ *"alarm"* งานชื่อ *"allclear"* และงานชื่อ *"shutdown"* โดยทุกงานจะต้องรอจังหวะที่ *ClockState* มีค่าเป็น *Scheduling* เพื่อขออนุญาตทำการจองทรัพยากรที่มีสถานะเป็น *_IDLE_* ก่อนทำงานจริงเสมอ เมื่อเป็นไปตามเงื่อนไขดังกล่าว งานแต่ละงานจะได้รับทรัพยากรจัดสรรให้ได้ตามต้องการ

```
1    inline Scheduler_EDF(TaskType){
2    if::(!CheckEDF(TaskType))-> Resource = TaskType;
3      :: else->skip;
4    fi;}
5
6    active proctype alarm(){
7    do   :: ((GlobalClockState == C_Schedule)&&(Resource == _IDLE_))->
8         Scheduler_EDF(event_alarm);
9    od;}
10   active proctype allclear(){
11   do   :: ((GlobalClockState == C_Schedule)&&(Resource == _IDLE_))->
12        Scheduler_EDF(event_allclear);
13   od;}
14   active proctype shutdown(){
15   do   :: ((GlobalClockState == C_Schedule)&&(Resource == _IDLE_))->
16        Scheduler_EDF(event_shutdown);
17   od;}
```

รูปที่ 3.10 ตัวอย่างโพรเมลาส่วนควบคุมกำหนดการ

ส่วนประกาศตัวแปรเหตุการณ์เอ็มทีแอลและส่วนกำหนดค่าตั้งต้นหลักของเครื่องประมวลเอ็มทีแอล

ส่วนนี้เป็นโพรเมลาที่ประกาศตัวแปรเหตุการณ์ที่ต้องใช้ในการจำลองเหตุการณ์แบบเอ็มทีแอล โดยชุดตัวแปรที่มีแสดงในรูปที่ 3.11 สังเกตสองตัวแปรสุดท้ายคือตัวแปรที่ใช้ระบุค่าช่วงเวลาคือ ตัวแปร *pTime1* ใช้กำหนดขอบซ้ายของช่วงเวลาและตัวแปร *pTime2* ใช้กำหนดขอบด้านขวาของช่วงเวลา โดยมีเงื่อนไขว่า *pTime1* <= *pTime2* เสมอ และค่าตั้งต้นหลัก (initial values) ของเครื่องประมวลผลเอ็มทีแอลแสดงไว้ในรูปที่ 3.12 โดยทำการกำหนดค่าตั้งต้นก่อนทำงานของระบบในครั้งแรก

```
1 typedef MTLevent{
2    bool    Event;            /*Event*/
3    bool    Reaction;         /*Corresponding Event*/
4    bool    EventFlag;        /*Event flag */
5    int     EventCounter;     /*Event Counter */
6    int     pTime1;           /*Left Boundary*/
7    int     pTime2;};         /*Right Boundary*/
```

รูปที่ 3.11 โพรเมลาที่ประกาศตัวแปรเหตุการณ์ [17]

```
1  inline MTLEngine(){
2    Event0.Event = alarm;
3    Event0.Reaction = allclear;
4    Event0.pTime1 = 0;
5    Event0.pTime2 = 10;
6    MTLCounter(Event0);
7    Event1.Event = alarm;
8    Event1.Reaction = shutdown;
9    Event1.pTime1 = 10;
10   Event1.pTime2 = 10;
11   MTLCounter(Event1);}
```

รูปที่ 3.12 โพรเมลาประกาศค่าตั้งต้นสำหรับการจำลอง MTLEvent [17]

ส่วนนับช่วงเวลาที่ระบุในตัวดำเนินการเอ็มทีแอล

เมื่องานของอุปกรณ์ที่ได้รับการจัดสรรพร้อมด้วยทรัพยากรที่ต้องการแล้วได้จังหวะเริ่มทำงาน ระหว่างที่มีดำเนินการไปตามเวลานาฬิกาจริงหลักของระบบเอ็มทีแอลจะต้องมีการดักจับช่วงเวลาที่กำหนดไว้ในตัวดำเนินการเอ็มทีแอลโดย

การคอยสังเกตจากค่าในตัวแปร *pTime1* และ *pTime2* ตัวอย่างของส่วนโพรเมลาที่ใช้นับช่วงเวลาดังกล่าวจะแสดงในรูปที่ 3.13

```
1 inline MTLCounter(CauseEvent){
2    if:: (CauseEvent.Event) ->
3       if::!CauseEvent.EventFlag->CauseEvent.EventCounter = 0;
4          CauseEvent.EventFlag = true;
5          ::else->skip;
6       fi;
7       ::(!CauseEvent.Event&&CauseEvent.EventFlag)->
8          if::((CauseEvent.EventCounter<=CauseEvent.pTime2)
9             &&(CauseEvent.EventCounter!=expired))->
10            CauseEvent.EventCounter++;
11            ::else->CauseEvent.EventCounter=expired;
12            CauseEvent.EventFlag = false;
13         fi;
14   ::else->skip;
15 fi;}
```

รูปที่ 3.13 ตัวอย่างของส่วนโพรเมลาที่ใช้นับช่วงเวลา [17]

แผ่นแบบที่ใช้ในการแปลงตัวดำเนินการเอ็มทีแอล

สำหรับตัวดำเนินการเอ็มทีแอลที่มีการกำกับช่วงเวลา เราจะใช้แผ่นแบบที่แสดงในรูปที่ 3.14 ในการแปลงให้เป็นส่วนโพรเมลาที่ทำงานให้ผลเหมือนกัน โดยแทรกการตรวจช่วงเวลาจากค่าตัวแปร *pTime1* และ *pTime2* ไว้ด้วย ผลการแปลงสามารถนำมาใช้กับเครื่องมือสปินได้ โดยสามารถจำลองการทำงานแบบช่วงเวลาเพิ่มเติมได้

ในการทวนสอบระบบจริงที่ต้องการคุณลักษณะเชิงเวลาที่มีช่วงเวลามาเกี่ยวข้องแต่ไม่สามารถกำหนดได้ด้วยแอลทีแอลได้ เราสามารถใช้เอ็มทีแอลกำหนดไปก่อน จากนั้นในขั้นตอนลงทวนสอบจริงสามารถนำวิธีการแปลงนี้มาประยุกต์ใช้เพื่อให้ได้แอลทีแอลที่มีพฤติกรรมเป็นแบบเอ็มทีแอลได้ โดยการเพิ่มส่วนต่างๆ ที่กล่าวมาแล้ว และนำไปใช้ได้จริงในเครื่องมือสปิน

ตัวอย่างเอ็มทีแอลข้างล่างนี้

$$[]\ (alarm \Longrightarrow (<>_{(0,10)} allclear \vee <>_{\{10\}} shutdown))$$

สูตรเชิงเวลาแบบเอ็มทีแอลที่มีการกำกับช่วงเวลา *(i, j)* โดยที่ i เป็นตำแหน่งเวลาแรก และ j เป็นตำแหน่งเวลาถัดไป ที่กำกับไว้ที่ตัวดำเนินการเชิงเวลา เช่นในตัวอย่างข้างล่าง ตัวดำเนินการเชิงเวลา <> ตัวที่สองจากซ้ายถูก

กำกับด้วยช่วงเวลา *(0,10)* และ <> ตัวที่สามก็ถูกกำกับด้วย *{10}* ซึ่งจะหมายถึง *(10,10)* คือ ณ เวลาที่ *10*

ขอยกตัวอย่างการตีความเอ็มทีแอลข้างต้นไว้ดังนี้

"ถ้ามี *alarm* เกิดขึ้นที่เวลา *t=i* แล้ว ในเวลาต่อมาจะต้องมี *allclear* เกิดขึ้น ณ เวลาถัดที่ *t=i+0* ถึง *t=i+10* หรือต้องมี *shutdown* เกิดขึ้น ณ เวลาที่ *t=i+10* และเป็นแบบนี้ตลอดไปเสมอ"

$\Diamond_{(t1,t2)}p$

```
#define MTL_Fpp(effect) <>(effect.Reaction&&(effect.EventCounter>
 effect.pTime1)&&(effect.EventCounter<effect.pTime2))
```

$\Diamond_{[t1,t2]}p$

```
#define MTL_Fss(effect) <>(effect.Reaction&&(effect.EventCounter>=
 effect.pTime1)&&(effect.EventCounter<=effect.pTime2))
```

$\Box_{(t1,t2)}p$

```
#define MTL_Gpp(effect) [](!((effect.EventCounter>effect.pTime1)&&
 (effect.EventCounter<effect.pTime2))||((effect.Reaction
 &&(effect.EventCounter>effect.pTime1))&&(effect.EventCounter
 <effect.pTime2)))
```

$\Box_{[t1,t2]}p$

```
#define MTL_Gss(effect) [](!((effect.EventCounter>=effect.pTime1)&&
 (effect.EventCounter<=effect.pTime2))||((effect.Reaction&&
 (effect.EventCounter>=effect.pTime1))&&(effect.EventCounter
 <=effect.pTime2)))
```

รูปที่ 3.14 แผ่นแบบที่ใช้ในการแปลงตัวดำเนินการเอ็มทีแอลเป็นแอลทีแอลในโพรเมลา [17]

3.5 แบบฝึกหัด

1. ตรรกศาสตร์เชิงเวลาคืออะไร มีประโยชน์อย่างไร?
2. ตรรกศาสตร์เชิงเวลามีกี่ประเภท อย่างไร?
3. ตัวดำเนินการเชิงเวลาแบบ *"ถัดไป"* หมายถึงอะไร?
4. ตัวดำเนินการแบบ *"จนกระทั่ง"* และแบบ *"จนกระทั่งแบบอ่อน"* ต่างกันอย่างไร?

5. จงอธิบายการตีความของสูตรเชิงเวลาเหล่านี้ ด้วยตารางค่าความจริง

   สูตรที่ 1: $(\neg resigned \wedge sad) \Rightarrow$ <>*famous*

   สูตรที่ 2: $sad \Rightarrow$ <>*happy*

   สูตรที่ 3: $is_Monday \Rightarrow$ <>*is_Friday*

   สูตรที่ 4: []*wet* $\Leftrightarrow \neg$<>*dry*

6. โมเดลในตรรกศาสตร์เชิงเวลาคืออะไร ใช้ประโยชน์อย่างไร?
7. ตัวดำเนินการเชิงเวลาในอนาคตต่างกับตัวดำเนินการเชิงเวลาในอดีตอย่างไร?

4

สร้างแบบจำลองเชิงรูปนัยด้วยภาษาเชิงรูปนัย

4.1 ความสำคัญของบทนี้

บทนี้จะกล่าวถึงการสร้างแบบจำลองเชิงรูปนัยของระบบ ด้วยการเขียนอธิบายพฤติกรรมของระบบด้วยภาษาเชิงรูปนัยที่ออกแบบมาอย่างเหมาะสม โดยมีชุดคำสั่ง หรือสัญลักษณ์คณิตตรรกศาสตร์ที่เอื้อให้ผู้สร้างแบบจำลองใช้งานสะดวก และเหมาะสมกับประเภทของระบบที่จำลอง ทั้งนี้เนื้อหาจะแนะนำภาษาเชิงรูปนัยที่ง่ายต่อการเข้าใจสำหรับผู้เริ่มต้น เช่น ภาษาเซด ภาษาบี และภาษาคาเฟโอบีเจ เป็นต้น พร้อมยกตัวอย่างประกอบโดยสังเขป การสร้างแบบจำลองมีเครื่องมือสนับสนุน เช่น เครื่องมือเซดอีฟ สำหรับสร้างและทวนสอบแบบจำลองที่เขียนด้วยภาษาเซด เครื่องมือคาเฟ (CAFÉ) สำหรับสร้างและทวนสอบแบบจำลองที่เขียนด้วยภาษาคาเฟโอบีเจ เป็นต้น

4.2 วัตถุประสงค์

- เพื่อให้รู้จักและเข้าใจภาษาเซด ภาษาบี และภาษาคาเฟโอบีเจ
- เพื่อให้รู้จักและเข้าใจสัญกรณ์ที่สำคัญที่ใช้ในภาษาข้อกำหนดเชิงรูปนัย
- เพื่อแสดงกรณีตัวอย่างการเขียนข้อกำหนดเชิงรูปนัยในแบบจำลอง

4.3 กลุ่มภาษาเชิงรูปนัย

ภาษาเชิงรูปนัย หรือภาษาข้อกำหนดเชิงรูปนัยที่มีอยู่นั้นจัดออกได้เป็นสองกลุ่มหลัก คือ กลุ่มภาษาแบบโมเดลเบส (model based) และกลุ่มภาษาแบบพรอพเพอร์ตีเบส (property based) ซึ่งถึงแม้ว่าทั้งสองแบบจะมีพื้นฐานมาจากภินทนคณิตศาสตร์ (discrete mathematics) เหมือนกัน แต่จุดที่แตกต่างของทั้งสองกลุ่ม คือ ส่วนวากยสัมพันธ์และเทคนิคการในการเขียนข้อกำหนด โดยแต่ละกลุ่มจะมีความเหมาะสมในการที่จะนำไปใช้อธิบายข้อกำหนดระบบซอฟต์แวร์ที่มีพฤติกรรมที่แตกต่างกัน

ภาษาเชิงรูปนัยช่วยในการกำหนดความต้องการโดยการอธิบายประมวลศัพท์ วากยสัมพันธ์และความหมายไว้อย่างเป็นระเบียบ เช่น ภาษาเซดประกอบด้วย กลุ่มของตัวอักษรที่แสดงสัญกรณ์ทางคณิตศาสตร์และตรรกศาสตร์

ภาคแสดงอันดับที่หนึ่งและมีวากยสัมพันธ์ระบุไว้อย่างชัดเจน ไม่คลุมเครือโดยใช้แผนภาพต้นไม้ไวยากรณ์ (grammar tree) จึงทำให้ความหมายที่ได้นั้นชัดเจนและแน่นอน เป็นต้น

ข้อดีและข้อเสียของการใช้ภาษาเชิงรูปนัย

ข้อดีของการใช้ภาษารูปนัย [18] มีดังนี้ คือ

- ประโยคในข้อกำหนดไม่กำกวม มีความหมายอย่างใดอย่างหนึ่งชัดเจน ถ้ามีการระบุในประโยคจะมีความหมายชัดเจน ถ้าไม่ระบุจะหมายถึงสรุปไม่ได้ หรือยังไม่สรุป
- ผู้เขียนข้อกำหนดสามารถพิสูจน์ความสอดคล้องของประโยคทุกประโยคในข้อกำหนดได้หลังจากเขียนข้อกำหนดแล้วก่อนเริ่มลงมือทำสิ่งต่อไป
- ข้อกำหนดช่วยให้เข้าใจการทำงานของระบบได้ดียิ่งขึ้น ตั้งแต่ในช่วงเริ่มต้นของการพัฒนาระบบก่อนการออกแบบ
- ข้อกำหนดที่ตรวจสอบแล้วช่วยลดความเป็นไปได้ที่จะเกิดความผิดพลาดของการออกแบบระบบได้
- ข้อกำหนดช่วยลดค่าใช้จ่ายในการพัฒนาระบบ เนื่องจากความผิดพลาดต่าง ๆ สามารถตรวจพบได้ในขั้นตอนของการวิเคราะห์และออกแบบ ซึ่งทำให้สามารถแก้ไขได้โดยเสียค่าใช้จ่ายไม่มากถ้าเทียบกับการตรวจพบความผิดพลาดในกิจกรรมท้าย ๆ ของการพัฒนาระบบ เช่น หลังจากเขียนโปรแกรมแล้ว เป็นต้น

ข้อเสียของการใช้ภาษาเชิงรูปนัย [18] มีดังนี้ คือ

- ภาษาและข้อกำหนดเชิงรูปนัยเข้าใจยาก เนื่องจากใช้รูปแบบทางคณิตศาสตร์และตรรกศาสตร์ในการเขียนข้อกำหนด
- เครื่องมือที่สนับสนุนในการใช้งานด้านอุตสาหกรรมยังใช้ไม่สะดวกและไม่แพร่หลาย

ข้อกำหนดแบบโมเดลเบส

ข้อกำหนดแบบโมเดลเบส เป็นการเขียนอธิบายระบบโดยใช้แบบจำลองสถานะของระบบ เสมือนระบบมีสถานะหนึ่งใดและสถานะของระบบสามารถเปลี่ยนแปลงไปเมื่อเวลาผ่านไป [18, 19] การอธิบายแบบจำลองสถานะของระบบทำได้โดยระบุเอนทิตีทางคณิตศาสตร์ (mathematical entity) และการดำเนินการที่เป็นไปได้ที่จะเปลี่ยนสถานะของระบบจากสถานะหนึ่งไปอีกสถานะหนึ่ง จากการเปลี่ยนค่าของเอนทิตีที่มีอยู่ในระบบนั่นเอง

เอนทิตีทางคณิตศาสตร์ที่นิยมเลือกใช้ในการเขียนข้อกำหนดแบบโมเดลเบส ได้แก่ เซต โดยแต่ละเซตก็คือ สิ่งหรือเอนทิตีที่เราสนใจอยู่ และฟังก์ชันที่กระทำกับเซตเหล่านั้น เป็นต้น หรือที่พบในภาษาเซดที่ใช้เขียนข้อกำหนดจะมีการใช้เซตระบุแบบชนิด (typed set) เป็นเอนทิตี เป็นต้น ตัวอย่างภาษาที่เป็นภาษาข้อกำหนดแบบโมเดลเบส คือ ภาษาเซด ภาษาวีดีเอ็ม-เอลแอล หรือภาษา *AMN* ของวิธีเชิงรูปนัยบี เป็นต้น

ข้อกำหนดแบบพรอพเพอร์ตีเบส

ข้อกำหนดแบบพรอพเพอร์ตีเบส [18, 19] เริ่มจากการกำหนดการดำเนินการ ที่เกิดขึ้นในระบบ และระบุความสัมพันธ์ระหว่างการดำเนินการที่มีอยู่ในระบบ ข้อกำหนดแบบพรอพเพอร์ตีเบสประกอบด้วยสองส่วนหลัก คือ ส่วนประกาศลายเซ็นของการดำเนินการ (operation signature part) และส่วนที่ระบุความสัมพันธ์ระหว่างการดำเนินการที่เขียนในรูปแบบของสมการ (equation) หรือเรียกว่าส่วนสัจพจน์ (axiom part) ตัวอย่างของภาษาที่เป็นภาษาข้อกำหนดแบบพรอพเพอร์ตีเบส คือ ภาษาโอบีเจ (OBJ) และภาษาคาเฟโอบีเจ เป็นต้น

4.4 ภาษาเซด

ภาษาเซดเป็นภาษาข้อกำหนดเชิงรูปนัยภาษาหนึ่ง [4, 18, 19] ที่จัดอยู่ในกลุ่มของภาษาแบบโมเดลเบส โดยภาษาเซดมีลักษณะโครงสร้างแบบมอดูล ใช้เขียนอธิบายระบบเป็นส่วน ๆได้ ทำให้ภาษาเซดเขียนได้ไม่ยาก และสามารถนำส่วนที่เขียนไว้แล้วมาใช้ใหม่ได้ โดยการนำมารวมกับส่วนประโยคที่เขียนใหม่ เรียกว่า การนำมารวมกัน (inclusion)

แต่ละมอดูลของภาษาเซดเรียกว่า *schema* โดยแต่ละมอดูลประกอบด้วยสองส่วนคือ ส่วนการประกาศ (declaration part) และส่วนเงื่อนไขบังคับ (constraints part) ภาษาเซตเป็นที่นิยมกันมาระยะหนึ่งและมีผู้พัฒนาเครื่องมือซอฟต์แวร์สนับสนุนการใช้งานด้วย เช่น โปรแกรมเซดอีฟ [20] เป็นต้น

4.5 ภาษาวีดีเอ็ม-เอสแอล

วิธีพัฒนาโปรแกรมแบบวีดีเอ็ม (VDM ย่อมาจาก Vienna development method) เป็นวิธีที่ใช้เทคนิคการเขียนข้อกำหนดเชิงรูปนัยโดยใช้ภาษาที่เรียกว่า ภาษาวีดีเอ็ม-เอสแอล (VDM specification language) [21] และมักจะมีเครื่องมือซอฟต์แวร์ที่สนับสนุนการพัฒนาในรูปแบบนี้

วิธีการทำงานในวีดีเอ็ม เริ่มจากการเขียนข้อกำหนดที่เป็นนามธรรมมากก่อน จากนั้นก็มีการลงรายละเอียดมากขึ้นวนซ้ำเรื่อย ๆ โดยการทำให้ข้อมูลเป็นรูปธรรมมากขึ้น และลงรายละเอียดของการดำเนินการ

การทำให้ข้อมูลเป็นรูปธรรม เป็นการปรับปรุงรายละเอียดแบบชนิดข้อมูลนามธรรม (abstract data type) ให้เป็นโครงสร้างข้อมูล (data structure) ในที่สุด ส่วนการทำการลงรายละเอียดการดำเนินการเป็นการปรับปรุงรายละเอียดให้กับการดำเนินการและฟังก์ชัน จนถึงเป็นอัลกอริทึมในที่สุดเช่นเดียวกัน ซึ่งทั้งโครงสร้างข้อมูลและอัลกอริทึมสามารถนำมาปรับให้เป็นโปรแกรมที่ทำงานได้ในระบบคอมพิวเตอร์ได้จริง

4.6 วิธีเชิงรูปนัยบี

วิธีเชิงรูปนัยบี (formal method B) [22] เป็นวิธีเชิงรูปนัยทางเลือกหนึ่งที่เป็นที่กล่าวถึงและนิยมกัน โดยใช้สัญกรณ์ที่เรียกว่าเอเอ็มเอ็น (AMN: abstract machine notations) โดยวิธีเชิงรูปนัยบี ได้รับการออกแบบเพื่อสนับสนุนการพัฒนาซอฟต์แวร์

วิธีเชิงรูปนัยบี ได้รับการริเริ่มพัฒนาโดย *Jean-Raymond Abrial* ซึ่งเป็นผู้ริเริ่มการพัฒนาภาษาเซดด้วยเช่นกัน ข้อดีของวิธีเชิงรูปนัยบี คือ การที่มีเครื่องมือสนับสนุนในการใช้งานด้านอุตสาหกรรมโดยเครื่องมือจะช่วยในการเขียนข้อกำหนดเชิงรูปนัย การออกแบบระบบต่อจากนั้น การพิสูจน์ ตลอดจนการเขียนโปรแกรมจากข้อกำหนดและการออกแบบ เราเรียกกระบวนวิธีในการพัฒนาซอฟต์แวร์โดยใช้วิธีเชิงรูปนัยบีว่า บีเมทรอด (B-Method)

วิธีเชิงรูปนัยบีเป็นที่กล่าวถึงกันว่าดีกว่าภาษาเซด เนื่องจากว่ามีการลงรายละเอียดในการกำหนดโครงสร้างและคุณลักษณะของซอฟต์แวร์ในระดับล่างหรือในระดับรูปธรรมได้ดีมากกว่าข้อกำหนดที่เขียนด้วยภาษาเซด จึงทำให้วิธีเชิงรูปนัยบี ได้รับการออกแบบการแปลงให้เป็นภาษาที่ใช้พัฒนาโปรแกรมได้ง่ายกว่าและเป็นรูปธรรมมากกว่าได้ เช่น การพัฒนาวิธีการแปลงข้อกำหนดที่เขียนด้วยวิธีเชิงรูปนัยบี ไปเป็นโปรแกรมภาษาจาวา เป็นต้น

4.7 ภาษาคาเฟโอบีเจ

ภาษาคาเฟโอบีเจ เป็นภาษารุ่นใหม่ที่อยู่ในกลุ่มภาษาพรอพเพอร์ตีเบส โดยเป็นภาษาข้อกำหนดเชิงพีชคณิต (algebraic specification language) กล่าวคือเป็นภาษาที่เขียนด้วยสมการพีชคณิต พัฒนาและสนับสนุนโดย *K. Futatsugi*

ภาษาคาเฟโอบีเจ [23] ได้รับการถ่ายทอดแนวคิดและคุณลักษณะภาษามาจากต้นแบบภาษาโอบีเจ โดยได้มีการเพิ่มทฤษฎีต่าง ๆ เข้าไป เช่น ทฤษฎี *rewriting logic* และ *hidden algebra* เป็นต้น ปัจจุบันผู้พัฒนาเครื่องมือซอฟต์แวร์สนับสนุนการใช้งานภาษาคาเฟโอบีเจ เรียกว่า *CAFE* เป็นต้น

4.8 การเขียนข้อกำหนดเชิงรูปนัยเบื้องต้น

โดยทั่วไปการเขียนข้อกำหนดโดยใช้ภาษาธรรมชาติ (natural language) ที่เป็นภาษาไทยหรือภาษาอังกฤษ เขียนเป็นรายการของประโยคที่อธิบายคุณลักษณะเฉพาะของซอฟต์แวร์ทีละข้ออย่างครบถ้วน แต่การเขียนข้อกำหนดด้วยภาษาธรรมชาติดังกล่าวมักจะมีปัญหา ดังนี้

- เนื้อหาบางส่วนมีข้อขัดแย้ง (contradictions)
  ข้อกำหนดที่เขียนด้วยประโยคจำนวนมาก อาจจะมีบางประโยคที่มีความหมายเป็นไปในทางตรงกันข้ามกัน ทำให้เมื่อรวมประโยคทุกประโยคเป็นหน้าที่ และขีดความสามารถของระบบแล้ว จะทำงานตามที่กำหนดไม่ได้ เพราะระบบจะต้องทำงานตามข้อกำหนดได้ทุกข้อ
- เนื้อหาบางส่วนกำกวม (ambiguities)
  ข้อกำหนดที่เขียนด้วยประโยคจำนวนมาก อาจจะมีบางส่วนมีความหมายกำกวม เมื่อผู้อ่านอ่านแล้วตีความได้สองแง่สองมุมหรือตีความได้หลายแง่มุมหลายความหมาย ทำให้ไม่เหมาะสมจะใช้เป็นข้อกำหนด
- เนื้อหาเลื่อนลอย (vagueness)
  ข้อกำหนดที่เขียนมีประโยคที่เขียนไม่เด่นชัด เนื้อหาเลื่อนลอยไม่มีการบ่งชี้ว่าต้องทำอะไรกันแน่
- เนื้อหาไม่ครบถ้วน (incompleteness)
  ข้อกำหนดมีเนื้อหาไม่ครบถ้วน โดยเมื่อต้องเขียนประโยคข้อกำหนดจำนวนมากๆ แล้ว การตรวจสอบว่ามีความครบถ้วนหรือไม่ อาจจะทำได้ยาก
- เนื้อหามีประโยคที่มีระดับนามธรรมต่างกันรวมปะปนกัน (mixed levels of abstraction)
  เนื้อหาข้อกำหนดควรมีประโยคที่บรรยายสิ่งที่ระบบต้องทำหน้าที่ได้ โดยแต่ละประโยคจะต้องมีระดับความเป็นนามธรรมในระดับเดียวกัน กล่าวคือ ไม่ควรมีบางประโยคระบุคร่าวๆ แต่

บางประโยคแจกแจงโดยละเอียด เนื้อหามีความละเอียดไม่เท่ากัน

ทางเลือกหนึ่งที่ช่วยแก้ปัญหาได้ คือ การหันมาเขียนข้อกำหนดดังกล่าวในรูปแบบเชิงรูปนัย กล่าวคือ การนำสัญกรณ์คณิตตรรกศาสตร์มาใช้บรรยายรูปประโยคแทนภาษาธรรมชาติเดิม เราจึงต้องมาเข้าใจวิธีการเขียนข้อกำหนดเชิงรูปนัยเบื้องต้นก่อน

ผู้เขียนข้อกำหนดจะต้องเข้าใจ และสามารถพิจารณาเนื้อหาของระบบ (system context) โดยทำการค้นหาส่วนประกอบที่สำคัญ คือ สถานะ ตัวยืนยงข้อมูล และการดำเนินการของระบบเป็นอันดับแรก ๆ โดยแต่ละการดำเนินการจะต้องมีค่าเงื่อนไขก่อนและเงื่อนไขหลังให้ครบถ้วน

สถานะ

โดยทั่วไปเรามักจะแยกความแตกต่างของสถานะของระบบได้ด้วยค่าข้อมูลในตัวแปรที่เราสังเกตได้ว่ามีค่าเปลี่ยนไป

สถานะแสดงถึงข้อมูลที่จัดเก็บในระบบ ข้อมูลดังกล่าวอาจจะได้รับการเปลี่ยนค่าใหม่ได้ระหว่างที่ระบบดำเนิน หรือทำงานอยู่ ทั้งนี้เพื่อให้ง่ายในการเข้าใจเรื่องของสถานะ เราอาจจะเปรียบเทียบสถานะกับตัวแปรที่ระบบมีอยู่เพื่อใช้เก็บข้อมูล

ตัวยืนยงข้อมูล

ตัวยืนยงข้อมูล แสดงถึงเงื่อนไขใด ๆ ของระบบที่เป็นจริงตลอดเวลาที่ระบบดำเนินไปหรือทำงานอยู่ ถ้าเงื่อนไขดังกล่าวเป็นเท็จแล้ว ระบบจะทำงานต่อไปไม่ได้หรือหยุดทำงานทันที ถ้าพิจารณาว่าในระบบที่เราสนใจมีการเก็บข้อมูลแล้ว เราสามารถกำหนดค่าของตัวยืนยงข้อมูล โดยการระบุเงื่อนไขของค่าของข้อมูลที่ควรเป็นจริงเสมอเท่านั้น ถ้าไม่เป็นไปตามนั้น ระบบจะทำงานต่อไปไม่ได้ เช่น ระบบห้องสมุด จะมีการจัดเก็บข้อมูลของจำนวนหนังสือในตัวแปรชื่อ *NumberOfBook* เราสามารถกำหนดตัวยืนยงข้อมูลว่า "*NumberOfBook >= 0*" กล่าวคือ ตัวแปรข้อมูล *NumberOfBook* จะเก็บค่าใด ๆ ที่มากกว่าหรือเท่ากับศูนย์เท่านั้น ถ้ามีค่าติดลบ ระบบห้องสมุดจะไม่สามารถทำงานต่อได้หรือต้องหยุดทำงานทันที เป็นต้น

ในระบบทั่วไปการหาค่าตัวยืนยงข้อมูลทำได้ไม่ง่ายนัก และเป็นการยากที่จะรู้ว่า เราหาค่ายืนยงข้อมูลดังกล่าวนี้ครบถ้วนทุกเงื่อนไขแล้วหรือยัง เรามักจะค้นพบค่าตัวยืนยงข้อมูลเพิ่มขึ้นเสมอ ระหว่างพิจารณาตรวจสอบหรือพิสูจน์ระบบ อย่างไรก็ตาม การกำหนดค่ายืนยงข้อมูลไว้น้อยจะทำให้ข้อกำหนดที่ได้มาเป็น

ข้อกำหนดหย่อน (loose specification) ในทางตรงกันข้าม ถ้ากำหนดไว้มากข้อ ข้อกำหนดที่ได้ก็จะเป็นข้อกำหนดเคร่งครัด (strict specification)

การดำเนินการ

การดำเนินการ เป็นขีดความสามารถการทำงานของระบบหรือเป็นหน้าที่ของระบบที่ต้องมีอยู่เพื่อให้ระบบทำงานได้อย่างครบถ้วน เรามักจะแจกแจงการดำเนินการเป็นข้อ ๆ โดยแต่ละข้อควรระบุเป็นหน้าที่หรือขีดความสามารถเดี่ยว กล่าวคือ การดำเนินการหนึ่ง ๆ ทำหน้าที่เดียวเท่านั้น เช่น ระบบห้องสมุด มีการดำเนินการหนึ่งที่ระบุได้คือ *"หน้าที่รับหนังสือเข้าใหม่"* โดยหน้าที่รับหนังสือเข้าใหม่ ก็ทำเฉพาะการรับหนังสือใหม่เท่านั้น ไม่ไปทำการให้ยืมหรือจองหนังสือ เป็นต้น

การกำหนดการดำเนินการให้กับระบบใด ๆ นั้น จะต้องมีการกำหนดเงื่อนไขก่อนและเงื่อนไขหลังที่เหมาะสมกับแต่ละการดำเนินการด้วย โดยจะได้อธิบายในหัวข้อต่อไป

เงื่อนไขก่อน

เงื่อนไขก่อน จะใช้ขยายความให้กับการดำเนินการใด ๆ เสมอ โดยเงื่อนไขก่อนเป็นเงื่อนไขที่ใช้ทดสอบก่อนจะเริ่มทำตามการดำเนินการนั้น ๆ ถ้าเงื่อนไขก่อนเป็นจริง ก็เริ่มทำงานตามหน้าที่ที่กำหนดไว้ในการดำเนินการนั้น ๆ ได้ แต่ถ้าเงื่อนไขก่อนมีผลเป็นเท็จ การดำเนินการนั้น ๆ จะไม่ถูกทำงานหรือถูกละเลยไปก่อน

ยกตัวอย่างของระบบห้องสมุด ที่มีการดำเนินการข้อหนึ่งที่ว่า *"การให้ยืมหนังสือ"* เราสามารถกำหนดเงื่อนไขก่อนได้คือ *"ผู้ยืมต้องเป็นสมาชิกห้องสมุดและหนังสือที่ยืมนั้นต้องมีอยู่ในห้องสมุด"* ซึ่งถ้าเงื่อนไขก่อนสำหรับการยืมหนังสือดังกล่าวเป็นจริงแล้ว การยืมหนังสือจะเกิดขึ้นได้ เมื่อหลังจากการดำเนินการได้ทำจบแล้ว เงื่อนไขก่อนอาจจะไม่เป็นจริงต่อไปอีกก็ได้ โดยหนังสือที่ยืมเล่มนั้นไม่มีเหลืออยู่ในห้องสมุดอีกต่อไป จนกว่าจะมีการนำมาคืน

เงื่อนไขก่อนไม่เหมือนกับเงื่อนไขที่กำหนดในค่าตัวยืนยงข้อมูล ข้อแตกต่างก็ คือ เงื่อนไขก่อนจะต้องเป็นจริงก่อนการทำงานของการดำเนินการและหลังจากการทำงานของการดำเนินการแล้วเงื่อนไขก่อนที่เคยเป็นจริง อาจจะไม่เป็นจริงอีกต่อไปได้ ต่างกับเงื่อนไขที่กำหนดในค่าตัวยืนยงข้อมูลจะต้องเป็นจริงเสมอทุกกรณี และตลอดเวลาที่ระบบทำงานอยู่

เงื่อนไขหลัง

เงื่อนไขหลังจะใช้ขยายความให้การดำเนินการใด ๆ เท่านั้นเช่นกัน โดยเงื่อนไขหลังเป็นเงื่อนไขที่จะต้องเป็นจริงหลังจากการทำงานของการดำเนินการนั้น ๆ สำเร็จตามคาดหมาย

ยกตัวอย่างของระบบห้องสมุด ที่มีการดำเนินการในการให้ยืมหนังสือ ที่กล่าวมาก่อนหน้าแล้ว เราสามารถกำหนดเงื่อนไขหลังสำหรับการยืมหนังสือที่ทำได้สำเร็จไว้ คือ *"จำนวนหนังสือที่ได้รับการยืมได้นั้นจะมีค่าลดลงไปเท่ากับจำนวนที่ยืมเสมอ"* เงื่อนไขดังกล่าวนี้ควรเป็นจริงทันทีหลังการให้ยืมหนังสือเสร็จสิ้นแล้ว แต่ถ้าเป็นเท็จจะหมายถึงการดำเนินการนั้น ๆ ทำได้ไม่สำเร็จตามที่กำหนด มีเหตุการณ์ผิดปกติ หรือเกิดความผิดพลาด

โดยทั่วไปเราจะใช้เงื่อนไขหลังไปช่วยในการออกแบบกรณีทดสอบที่ใช้ทดสอบซอฟต์แวร์ได้หลังพัฒนาโปรแกรมแล้วเสร็จ ดังนั้น เงื่อนไขหลังจึงมีความสำคัญ และควรเขียนระบุไว้เสมอ

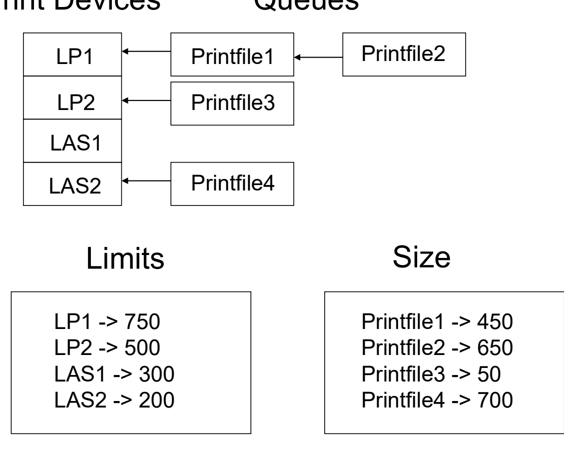
4.9 ตัวอย่างระบบ Print Spooler

คำอธิบาย *"ระบบ Print Spooler"*

ระบบ *Print Spooler* [1] เป็นระบบที่รองรับการส่งงานเอกสารเพื่อมาพิมพ์ที่เครื่องพิมพ์ โดยตามหลักการทั่วไปเอกสารที่ต้องการพิมพ์จะต้องถูกนำมาเข้าแถวคอยที่เรียกว่า *Queue* เพื่อรอให้ถึงเวลารับบริการพิมพ์ โดยแต่ละแถวคอยจะได้รับการเชื่อมต่อเข้ากับเครื่องพิมพ์แต่ละเครื่องที่ติดตั้งในระบบ ที่เรียกว่า *Print Device* จำนวนเครื่องพิมพ์จะมีได้สูงสุดไม่เกินค่า *MaxDevs* ซึ่งเป็นค่าคงที่ค่าหนึ่งใด ๆ

ทั้งนี้ ได้มีการกำหนดข้อจำกัดจำนวนบรรทัดสูงสุดในการบริการพิมพ์ของเครื่องพิมพ์แต่ละเครื่องไว้ด้วย เช่น กำหนดไว้ว่า *"LP1->750"* หมายถึงเอกสารงานที่ต้องการพิมพ์ที่ส่งมาพิมพ์ที่เครื่องพิมพ์ชื่อ *LP1* จะต้องมีจำนวนบรรทัดทั้งหมดไม่เกิน *750* บรรทัด เป็นต้น (ดังแสดงในรูปที่ 4.1)

ในระบบนี้จะมีการจัดเก็บชื่อของงานเอกสารแต่ละงาน พร้อมระบุจำนวนบรรทัดของเอกสาร เช่น มีการกำหนดไว้ว่า "*Printfile1-> 450*" หมายถึง งานเอกสารชื่อ *Printfile1* ซึ่งมีจำนวนบรรทัด *450* บรรทัดถูกส่งมาอยู่ในแถวคอยเพื่อการพิมพ์อยู่ขณะนี้ เป็นต้น



รูปที่ 4.1 แผนภาพส่วนประกอบระบบ Print Spooler [1]

การวิเคราะห์หาสถานะ

จากโจทย์ตัวอย่างที่กำหนดให้ดังกล่าว เรามีแนวทางการเขียนข้อกำหนดเชิงรูปนัยเบื้องต้น โดยพิจารณาว่าควรมีการจัดเก็บข้อมูลส่วนใดบ้าง เพื่อเป็นประโยชน์ในการทำงานของระบบ ทั้งนี้ระบบจะต้องสามารถประมวลผล คำนวณ และตรวจสอบสถานะ โดยตอบคำถามต่าง ๆ ที่ผู้ใช้ต้องการได้เป็นอย่างดี

ข้อมูลที่บ่งชี้ถึงสถานะที่กำหนดได้จากโจทย์คือ

- แถวคอยที่เรียกว่า *Queue*
- เครื่องพิมพ์ที่เรียกว่า *Print Device*
- จำนวนบรรทัดสูงสุดที่เครื่องพิมพ์ใด ๆรับพิมพ์ได้เรียกว่า *Printer Limit*
- ขนาดหรือจำนวนบรรทัดของแฟ้มเอกสารงานที่พิมพ์เรียกว่า *File Size*

การวิเคราะห์หาสถานะไม่ถูกกำหนดเป็นกฎตายตัวเสมอไป ทั้งนี้ต้องอาศัยประสบการณ์ของผู้เขียนข้อกำหนดเชิงรูปนัยเอง อย่างไรก็ตาม ลองมาดูวิธีที่เสนอต่อไปนี้ คือ การวิเคราะห์เนื้อหาของโจทย์ที่เรียกว่า การวิเคราะห์บริบท (context analysis) โดยเป็นการพิจารณาหาคำนามที่มีอยู่ในเนื้อหาบริบทที่เข้าใจได้ แล้วนำมาคัดเลือกเป็นข้อมูลบ่งชี้ถึงสถานะอีกทีหนึ่งได้ คำนามทุกคำควรจะได้รับการพิจารณาว่าคุ้มค่าที่จะได้รับการจัดเก็บไว้และเพื่อให้ระบบประมวลผลได้จริง

การวิเคราะห์หาตัวยืนยงข้อมูล

การวิเคราะห์หาเงื่อนไขของค่าตัวยืนยงข้อมูลสามารถทำได้โดยพิจารณาจากสถานะต่าง ๆ ที่กำหนดไว้ก่อนหน้านี้แล้ว โดยการวิเคราะห์ว่า แต่ละสถานะมีข้อจำกัดอะไรบ้างในระหว่างการทำงานของระบบ หรือสถานะคู่ใดจะต้องมีความสัมพันธ์กันอย่างไรในระหว่างการทำงานของระบบ เช่น *Print Device* จะต้องมีจำนวนได้ไม่เกิน *MaxDevs* โดยที่ค่า *MaxDevs* คือ จำนวนสูงสุดของเครื่องพิมพ์ที่ระบบจะติดตั้งได้ เป็นต้น

ต่อไปนี้เป็นรายการเงื่อนไขค่ายืนยงข้อมูลที่กำหนดได้สำหรับโจทย์

- เงื่อนไขที่ 1: เครื่องพิมพ์ *Print Device* แต่ละเครื่องจะต้องได้รับการกำหนดค่าจำนวนบรรทัดสูงสุดที่พิมพ์ได้เสมอ
  ตัวอย่างคือ ระบบจะต้องได้รับการกำหนดว่า *"LP1->750" "LP2->500" "LAS1->300"* และ *"LAS2->200"* โดยที่เครื่องพิมพ์ *Print Device* ขณะนี้มีทั้งสิ้นสี่เครื่องโดยมีชื่อว่า *LP1, LP2, LAS1* และ *LAS2* ตามลำดับ และได้รับการกำหนดค่าจำนวนบรรทัดสูงสุดที่พิมพ์ได้ไว้ว่า *750, 500, 300* และ *200* บรรทัดตามลำดับ
- เงื่อนไขที่ 2: เครื่องพิมพ์ *Print Device* แต่ละเครื่องจะต้องได้รับการเชื่อมต่อกับแถวคอย *Queue* เสมอ
  ตัวอย่างคือ ระบบจะต้องมีแถวคอย *Queue* จำนวนเท่ากับเครื่องพิมพ์ *Print Device* เสมอ โดยแต่ละแถวคอยจะได้รับการเชื่อมต่อกับเครื่องพิมพ์ที่หนึ่งๆ เสมอด้วย เมื่อแถวคอย *Queue* ใดๆมีแฟ้มงานเอกสารที่ส่งเข้ามาก็จะส่งงานต่อให้กับเครื่องพิมพ์ที่เชื่อมต่ออยู่แล้วอย่างต่อเนื่องตามลำดับ โดยทั่วไปการทำงานของแถวคอย *Queue* จะส่งงานไปให้เครื่องพิมพ์แบบ *First-In First-Out (FIFO)*
- เงื่อนไขที่ 3: แถวคอย *Queue* ที่เชื่อมต่อกับเครื่องพิมพ์ *Print Device* จะต้องบรรจุแฟ้มเอกสารพิมพ์ที่มีขนาดจำนวนบรรทัดไม่เกินจำนวนสูงสุดที่เครื่องพิมพ์นั้นๆพิมพ์ได้
  ตัวอย่างคือ เมื่อเราตรวจสอบจำนวนบรรทัดของชื่อแฟ้มงานเอกสารที่รอต่อแถวกันในแถวคอย *Queue* ใด ๆ โดยตรวจดูที่สถานะที่เรียกว่า *File Size* จะพบว่าทุกๆ แฟ้มงานเอกสารจะต้องมีจำนวนบรรทัดไม่เกินค่าจำนวนบรรทัดสูงสุดที่ระบุในสถานะที่เรียกว่า *Printer Limit*

- เงื่อนไขที่ 4: แฟ้มงานเอกสารที่พิมพ์จะต้องได้รับการกำหนดขนาดของแฟ้มเสมอ

  ตัวอย่างคือ แฟ้มงานเอกสารที่ถูกส่งเข้ามาในระบบนี้จะต้องได้รับการกำหนดขนาดโดยระบุใน *State* ที่เรียกว่า *File Size* เสมอ เช่น *"PrintFile1->350"* แสดงว่าแฟ้มงานเอกสารชื่อ *PrintFile1* จะขนาดแฟ้มหรือจำนวนบรรทัดที่พิมพ์ *350* บรรทัด เป็นต้น
- เงื่อนไขที่ 5: ระบบ *Print Spooler* นี้จะต้องมีจำนวนของเครื่องพิมพ์ *Print Device* ที่ติดตั้งไม่เกินค่าของ *MaxDevs* ซึ่งเป็นค่าคงที่ใด ๆ ที่กำหนดไว้แล้ว

  ตัวอย่างคือ เครื่องพิมพ์ *Print Device* ใหม่อาจจะได้รับการติดตั้งเข้ามาเพิ่มในระบบได้เสมอ โดยการติดตั้งจะมีการกำหนดค่าในสถานะที่เรียกว่า *Print Device* และมีการระบุ *Printer Limit* ด้วย อย่างไรก็ตาม ระบบจะสามารถรองรับเครื่องพิมพ์ทั้งหมดไม่เกินค่า *MaxDevs* ซึ่งเป็นค่าคงที่ใด ๆ เท่านั้น ดังนั้นการเพิ่มเครื่องพิมพ์ *Print Device* ใหม่จะต้องไม่ทำให้จำนวนเครื่องพิมพ์เกินไปจากค่า *MaxDevs* เสมอ

การวิเคราะห์หาการดำเนินการ

การวิเคราะห์หาการดำเนินการเพื่อให้รู้ว่าระบบควรจะทำหน้าที่อะไรบ้าง และควรจะหาการดำเนินการให้ครบถ้วนทุกกรณี เทคนิคหนึ่งในการหาว่าระบบจะต้องทำหน้าที่อะไรบ้างนั้น ทำได้โดยให้พิจารณาเหตุการณ์เป็นหลัก (scenario approach)

ตัวอย่างของส่วนประกอบการดำเนินการที่พิจารณาได้จากโจทย์ที่กำหนด แต่เราสามารถเพิ่มการดำเนินการได้จนสามารถอธิบายหน้าที่ของระบบให้ครบถ้วนได้ภายหลัง ให้มีดังต่อไปนี้

- การดำเนินการชื่อ *"AddPrintDevice"* ที่ทำหน้าที่เพิ่มเครื่องพิมพ์ *Print Device* ใหม่เข้าติดตั้งกับระบบนี้ โดยจะมีการกำหนดค่าจำนวนบรรทัดสูงสุดที่เครื่องพิมพ์ *Print Device*
- การดำเนินการชื่อ *"DeleteQueue"* ที่ทำหน้าที่ในการลบแฟ้มงานเอกสารที่อยู่ในแถวคอย *Queue* ใด ๆ ที่เชื่อมต่อกับเครื่องพิมพ์ *Print Device* ใด ๆ ได้

- การดำเนินการชื่อ *"AddQueue"* ที่ทำหน้าที่ในการเพิ่มแฟ้มงานเอกสารที่อยู่ในแถวคอย *Queue* ใด ๆ ที่เชื่อมต่อกับเครื่องพิมพ์ *Print Device* ใด ๆ ได้
- การดำเนินการชื่อ *"MoveQueueToPrintDevice"* ที่ทำหน้าที่ในการเคลื่อนย้ายแฟ้มงานเอกสารที่อยู่ในแถวคอย *Queue* ใดๆที่เชื่อมต่อกับเครื่องพิมพ์ *Print Device* ใด ๆ ไปยังแถวคอยอื่นๆที่เชื่อมต่อกับเครื่องพิมพ์อื่น ๆ ได้

การวิเคราะห์หาเงื่อนไขก่อนของแต่ละการดำเนินการ

เราจะต้องวิเคราะห์และกำหนดเงื่อนไขก่อนของทุกการดำเนินการที่กำหนดไว้แล้วเสมอ โดยวิเคราะห์เงื่อนไขที่ต้องมีค่าเป็นจริงก่อนที่การดำเนินการนั้นๆ จะเริ่มทำงานได้

ต่อไปนี้เป็นตัวอย่างของเงื่อนไขก่อนที่สามารถกำหนดให้กับ การดำเนินการใด ๆ ที่มีอยู่ในระบบนี้

- สำหรับการดำเนินการชื่อ *"AddPrintDevice"*
  เงื่อนไข *Precondition* ก่อนการทำงานของ *Operation* คือ *"จำนวนเครื่องพิมพ์ Print Device ขณะปัจจุบันมีค่าน้อยกว่าค่า MaxDevs"*
- สำหรับการดำเนินการชื่อ "*DeleteQueue*"
  เงื่อนไข *Precondition* ก่อนการทำงานของ *Operation* คือ *"จำนวนแฟ้มงานเอกสารในแถวคอยที่ต้องการลบแฟ้มงานเอกสารต้องมีค่ามากกว่าศูนย์"* กล่าวคือ จะต้องมีแฟ้มงานเอกสารอย่างน้อยหนึ่งแฟ้มงานให้ลบออกจากแถวคอย
- สำหรับการดำเนินการชื่อ "*AddQueue*"
  เงื่อนไข *Precondition* ก่อนการทำงานของ *Operation* คือ *"แฟ้มงานเอกสารที่ได้รับการเพิ่มจะต้องมีขนาดไม่เกินจำนวนบรรทัดสูงสุดที่ PrintDevice พิมพ์ได้"*
- สำหรับการดำเนินการชื่อ *"MoveQueueToPrintDevice"*
  เงื่อนไข *Precondition* ก่อนการทำงานของ *Operation* คือ *"จำนวนแฟ้มงานเอกสารในแถวคอยต้นทางต้องมีจำนวนมากกว่าศูนย์ และจำนวนบรรทัดของแฟ้มนั้นจะต้องไม่มากเกินกว่าจำนวนบรรทัดสูงสุดที่ PrintDevice พิมพ์ได้"*

การวิเคราะห์หาเงื่อนไขหลังของแต่ละการดำเนินการ

เช่นเดียวกับเงื่อนไขก่อน เราจะต้องวิเคราะห์และกำหนดเงื่อนไขหลังของทุกการดำเนินการเสมอ โดยเงื่อนไขหลังจะเป็นเงื่อนไขที่เป็นจริงเสมอ หลังจากที่ระบบได้ทำหน้าที่ตาม การดำเนินการได้สำเร็จแล้วเสร็จจริง

ต่อไปนี้เป็นตัวอย่างของเงื่อนไขหลังที่สามารถกำหนดให้กับการดำเนินการใด ๆ ที่มีอยู่ในระบบนี้

- สำหรับการดำเนินการชื่อ *"AddPrintDevice"*
  เงื่อนไข *Postcondition* ก่อนการทำงานของ *Operation* คือ *"จำนวนเครื่องพิมพ์ Print Device มีค่าเพิ่มขึ้นจากเดิม"*
- สำหรับการดำเนินการชื่อ *"DeleteQueue"*
  เงื่อนไข *Postcondition* ก่อนการทำงานของ *Operation* คือ *"จำนวนแฟ้มงานเอกสารในแถวคอยที่ต้องการลบแฟ้มงานเอกสารต้องมีค่าน้อยลงหนึ่งแฟ้มและแฟ้มงานเอกสารแฟ้มแรกสุดจะได้รับการลบออกจากแถวคอย"*
- สำหรับการดำเนินการชื่อ *"AddQueue"*
  เงื่อนไข *Postcondition* ก่อนการทำงานของ *Operation* คือ *"จำนวนแฟ้มงานเอกสารของแถวคอยจะเพิ่มขึ้นอีกหนึ่งแฟ้มและแฟ้มงานเอกสารจะได้รับการเพิ่มต่อท้ายแถวคอย"*
- สำหรับการดำเนินการชื่อ *"MoveQueueToPrintDevice"*
  เงื่อนไข *Postcondition* ก่อนการทำงานของ *Operation* คือ *"จำนวนแฟ้มงานเอกสารในแถวคอยต้นทางต้องมีจำนวนน้อยลงหนึ่งแฟ้ม"*

4.10 เขียนข้อกำหนดเชิงรูปนัยด้วยภาษาเซด

ภาษาเซด [4, 18, 19] เป็นภาษาข้อกำหนดเชิงรูปนัยที่ใช้ทฤษฎีของเซตและทฤษฎีของตรรกศาสตร์เชิงประพจน์ และตรรกศาสตร์ภาคแสดงชั้นที่หนึ่ง โดยได้มีการเพิ่มสัญกรณ์พิเศษบางส่วนเพื่อสนับสนุนการเขียนข้อกำหนด มีเป้าหมายให้การเขียนข้อกำหนดเป็นไปตามแบบข้อกำหนดเชิงโครงสร้าง (structural specification)

ผู้คิดค้นและสร้างภาษาเซด คือ *J. Abrial* ขณะที่เป็นศาสตราจารย์อยู่ที่มหาวิทยาลัย *Oxford* ประเทศอังกฤษ

ปัจจุบันมีกลุ่มผู้ใช้ภาษาเซดที่เรียกว่า *Z User Group* และมีการประชุมพบปะแลกเปลี่ยนความรู้กันในการประชุม *ZUM (Z User Meeting)* อยู่เป็นประจำ ดูรายละเอียดได้ที่ *http://www.zuser.org*

เนื่องจากความนิยมในหมู่ผู้ใช้ภาษาเซดมีมากขึ้น และติดต่อกันมาเป็นเวลานาน ภาษาเซดได้รับการกำหนดเป็นมาตรฐานโดยองค์การมาตรฐานสากล (International Standard Organization: ISO) โดยมีคณะกรรมการร่วมของ *ISO/IEC* ซึ่งมีรหัสว่า *JTC1/SC22/WG19* ซึ่งมีความหมายว่าเป็น *Joint Technical Committee* ที่ *JTC1* และมีคณะกรรมการย่อยที่ *SC22* ชุดที่ *WC19* ดูรายละเอียดได้ที่ *http://www.open-std.org/JTC1/SC22/WG19/*

โดยทั่วไปการเขียนข้อกำหนดความต้องการซอฟต์แวร์เชิงรูปนัย มักจะต้องระบุ ส่วนสถานะที่เก็บข้อมูล ส่วนตัวยืนยงข้อมูล ส่วนการดำเนินการ และจะต้องระบุเงื่อนไขก่อนพร้อมทั้งเงื่อนไขหลังสำหรับแต่ละการดำเนินการที่กำหนดไว้ ภาษาเซดเป็นภาษาหนึ่งที่ใช้เขียนข้อกำหนดเชิงรูปนัยดังกล่าวได้ อีกทั้งการเขียนข้อกำหนดด้วยภาษาเซดยังสามารถเขียนเป็นลักษณะโครงสร้างได้ กล่าวคือ เขียนข้อกำหนดเป็นส่วนย่อย ๆ และนำมาเชื่อมต่อรวมเพื่ออธิบายภาพรวมได้ในที่สุด

ในภาษาเซดมีการเขียนข้อกำหนดโดยแบ่งเป็นส่วนย่อย ๆ แยกจากกันได้ โดยเรียกแต่ละส่วนย่อยว่า *Schema* (มีความหมายเหมือนกับมอดูลได้) แต่ละ *Z Schema* จะใช้อธิบายสถานะและการดำเนินการได้

State Schema เป็น *Z Schema* ที่เขียนขึ้นเพื่ออธิบายส่วนประกอบของระบบ (system component) ที่สนใจ เช่น ในระบบห้องสมุดจะมีส่วนประกอบของระบบคือ หนังสือ บรรณรักษ์ ผู้ยืม เป็นต้น เราก็มักจะมี *State Schema* ที่ชื่อ *Book* สำหรับหนังสือ หรือ *State Schema* ที่ชื่อ *Librarian* สำหรับบรรณรักษ์

ภายใน *State Schema* เราจะเห็นตัวแปร *state variable, invariant* ที่เป็นข้อจำกัดของสถานะ

Operation Schema ก็จะเป็น *Z Schema* ที่เขียนขึ้นเพื่ออธิบาย การดำเนินการที่กระทำต่อ *State Schema* ที่มีอยู่ หรือกล่าวอีกนัยหนึ่งก็คือ อธิบายหน้าที่การทำงานของส่วนประกอบของระบบ จากตัวอย่างในระบบห้องสมุด เรามี *State Schema* ชื่อ *Book* ที่เป็นหนังสือไว้แล้ว เราจะพิจารณากำหนด *Operation Schema* เพิ่มสำหรับ *Book* ได้คือ *Operation Schema* ที่ชื่อ *BorrowBook* ได้

ภายใน *Operation Schema* มักจะต้องอ้างถึง *State Schema* ที่เกี่ยวข้องด้วย ระบุตัวแปรที่เป็นข้อมูลนำเข้า ข้อมูลส่งออกและระบุเงื่อนไข เงื่อนไขก่อนและเงื่อนไขหลังเป็นสำคัญ

การเขียน Z Schema แบบกล่อง

เราสามารถเขียน *Z Schema* ได้สองแบบ คือ การเขียน *Schema* แบบกล่อง (boxed form) และการเขียน *Schema* แบบแนวระดับ (horizontal form) เราสามารถเลือกเขียนได้ตามต้องการ โดยทั่วไปมักจะเขียนแบบกล่องเพื่อให้ง่ายในการทำความเข้าใจมากกว่า ดังตัวอย่างต่อไปนี้ ซึ่งเป็น *State Schema* ชื่อ *Book* [20]

```
┌─ Book ──────────────────────────
│ author: People
│ title: seq CHAR
│ readership: ℙPeople
│ rating: People ⇸ 0..10
├──────────────────────────────────
│ readership = dom rating
└──────────────────────────────────
```

การเขียน *Schema* แบบกล่องมักจะเขียนเป็นเส้นล้อมสามด้าน โดยแบ่งเป็นสองส่วนคือ ส่วนบนเป็นการระบุชื่อตัวแปรสถานะที่มีและเก็บข้อมูลพร้อมกันนี้ก็จะระบุชนิดของตัวแปรด้วยซึ่งเป็นเงื่อนไขบังคับโดยใช้ชนิดข้อมูล การระบุเงื่อนไขบังคับโดยใช้ชนิดข้อมูลนี้ ทำให้เราทราบได้ว่าตัวแปรชื่อนั้นจะมีค่าจริงอยู่ที่เป็นไปได้อย่างไร เช่น ตัวแปร x มีชนิดข้อมูลเป็นตัวเลข ดังนั้นค่าที่ x จะเก็บต้องเป็นตัวเลขเท่านั้น เป็นต้น

ส่วนล่างของ *Schema* แบบกล่องเป็นการระบุเงื่อนไขบังคับ โดยใช้เพรดิเคต สำหรับ *State Schema* การระบุเงื่อนไขบังคับโดยใช้เพรดิเคตนี้ ทำให้เราทราบถึงเงื่อนไขของค่ายืนยงของตัวแปรที่ต้องเป็นจริงตลอดเวลาที่ระบบทำงานอยู่ เช่น $x > 5$ ดังนั้น ค่าที่ x จะเก็บต้องเป็นตัวเลขที่มีค่ามากกว่า 5 ขึ้นไปเท่านั้น จะมีค่าน้อยกว่าหรือเท่ากับ 5 ไม่ได้เลย ในขณะที่ *State Schema* นี้นำไปใช้งาน

การเขียน Schema แบบบรรทัดฐาน

ภาษาเซด มีข้อดี คือ เราเขียนข้อกำหนดเชิงรูปนัยได้ไม่ยากและสามารถเขียนส่วนย่อย ๆ ของส่วนประกอบระบบทีละส่วนย่อย จากนั้นจึงนำมารวมเป็นข้อกำหนดเชิงรูปนัยรวมสุดท้ายได้อย่างเชิงโครงสร้าง โดยคำว่าเชิงโครงสร้างหมายถึง เราสามารถกำหนดโครงสร้างของข้อกำหนดภาพรวมสุดท้ายว่าจะประกอบด้วยข้อกำหนดย่อย ๆ ที่มีอยู่ได้ ถือว่าไม่ต้องมีการเขียนกันใหม่แต่เป็น

การนำมารวม อย่างไรก็ตาม ข้อเสียของภาษาเซดข้อหนึ่ง คือ ไม่มีวิธีการกำหนดรูปแบบการเขียนอย่างชัดเจน ผู้เขียนสามารถเขียนข้อกำหนดตามต้องการได้เท่าที่ประโยคไม่ขัดแย้งกัน เช่น สามารถกำหนดตัวแปรสถานะ ชนิดของตัวแปร เงื่อนไขที่เป็นขอบเขตและข้อจำกัดได้อย่างไม่ถูกบังคับ เป็นต้น

ในที่นี้จะกล่าวถึงทางเลือกหนึ่งในการเขียน *Z Schema* แบบบรรทัดฐาน (normalized form) [18] ที่จะใช้ให้ข้อกำหนดที่เขียนขึ้นเข้าใจและติดตามตรวจสอบได้ง่าย การเขียนแบบบรรทัดฐานนี้จะมีแนวคิดในการเขียนดังต่อไปนี้

- การระบุชนิด (type) ของตัวแปรโดยใช้
  - กีเวนเซต (given set)
  - ความสัมพันธ์ระหว่างเซต (set relation)
  - ไม่ใช้การดำเนินการหรือฟังก์ชันของ Z ในการระบุชนิดตัวแปร
- ย้ายการระบุการดำเนินการหรือฟังก์ชันของ Z มาอยู่ในส่วนล่างของ Schema คือให้ระบุเป็นส่วนเพรดิเคตแทน
- เขียนเพรดิเคตแยกเป็นทีละบรรทัดให้ชัดเจน โดยทุกบรรทัดจะได้รับการรวมความ (conjoin) กันอยู่แล้ว หรือที่เราเรียกว่า ทุกประโยคเพรดิเคตจะได้รับการนำมาทำการ *AND* กัน

พิจารณาตัวอย่างที่ *State Schema* ชื่อว่า *Book* สามารถนำมาเขียนใหม่ได้ดังนี้

```
┌─ Book ──────────────────
│ author: People
│ title: ℙ(ℤ X CHAR)
│ readership: ℙPeople
│ rating: ℙ(People X ℤ)
├─────────────────────────
│ title ∈ seq CHAR
│ rating ∈ People ⇸ 0..10
│ readership = dom rating
└─────────────────────────
```

จากตัวอย่าง *State Schema* เดิมจะเห็นได้ว่า ตัวแปรที่กำหนดสำหรับ *Schema Book* นั้นมีการระบุชนิดตัวแปรโดยใช้การดำเนินการ ของ Z ดังนี้

การระบุว่า *title: seq CHAR*

ตัวแปรชื่อ *title* จะเป็นตัวแปรชนิดที่ระบุด้วย *seq CHAR* โดย *seq* เป็นการดำเนินการของสัญกรณ์เซด ที่ใช้อธิบายการนำสมาชิกของเซตมาเรียงเป็นแถวตามลำดับก่อนหลัง และ *CHAR* จะเป็นกีเวนเซตที่กำหนดมาให้แล้ว กล่าวคือ เป็นเซตของตัวอักษร

ดังนั้น *seq CHAR* ก็คือเซตของชุดตัวอักษรใด ๆ จำนวนหนึ่งที่เรียงลำดับก่อนหลัง เช่น กำหนดให้เซต *CHAR = {A,B,C, ...}* แล้ว ดังนั้น *seq CHAR* ก็คือ เซตของชุดลำดับ (Sequence) ซึ่งอาจจะเป็นเซตต่อไปนี้คือ *{<C,O,M,P,U,T,E,R>, <S,O,F,T,W,A,R,E>}* ก็ได้ และตัวแปร title อาจจะมีค่าเป็น *<C,O,M,P,U,T,E,R>* เป็นต้น (สัญกรณ์ <...> คือการระบุชุดลำดับซึ่งจะกว่าถึงต่อไป)

การระบุว่า $rating: People \rightarrow 0..10$

ตัวแปรชื่อ rating จะเป็นตัวแปรชนิดที่ระบุด้วยฟังก์ชันบางส่วน (Partial Function) จากกีเวนเซตชื่อ *People* ที่กำหนดไว้แล้ว ไปยังเซต *0..10* ซึ่งมีค่าเป็น *{0,1,2,3,4,5,6,7,8,9,10}*

ถ้ากำหนดให้กีเวนเซต *People = {Sombat, Somsri, Visoot...}* แล้ว $People \rightarrow 0..10$ จะให้ผลเป็นเซตเหมือนกันโดยเป็นเซตของฟังก์ชันดังกล่าว เช่น *{ {(Sombat,1),(Somsri,5)}, {(Visoot,7), (Somsri,9), (Sombat,4)}, ...}* เป็นต้น

ดังนั้นตัวแปร *rating* จะมีค่าเป็นเซตของคู่ลำดับ *{(Sombat,1), (Somsri,5)}* ได้ตามตัวอย่าง โดยความหมายของ *rating* ก็คือ การเก็บค่าความนิยมที่ผู้อ่านมีต่อหนังสือเล่มนี้ โดยการให้ค่าความนิยมเป็นตัวเลขจาก *0* ถึง *10* และมีการเก็บไว้ด้วยว่าผู้อ่านชื่ออะไรให้ค่าความนิยมเท่าไร เช่น คู่ลำดับ *(Sombat,1)* จะหมายถึงผู้อ่านชื่อ *Sombat* ที่ให้ค่าความนิยมในหนังสือเล่มที่มีชื่อตามที่ระบุในตัวแปร *title* ไว้ด้วยค่า *1* เป็นต้น

เมื่อใช้วิธีการเขียนแบบบรรทัดฐานแล้ว เราจะเห็นได้ว่าชนิดของตัวแปรทั้งสองจะเปลี่ยนไป โดยการระบุชนิดตัวแปรไว้ไม่เคร่งครัดมากเกินไปดังนี้

จากเดิมระบุว่า *title: seq CHAR*

เปลี่ยนการระบุเป็น $title: \mathbb{P}(\mathbb{Z} \times CHAR)$

ตัวแปรชื่อ *title* จะมีชนิดของตัวแปรเปลี่ยนจากเดิมที่ใช้การดำเนินการของสัญกรณ์เซด *seq CHAR* จะได้รับการเปลี่ยนไปเป็น $\mathbb{P}(\mathbb{Z} \times CHAR)$ เพื่อให้เป็นเซตที่ไม่เจาะจงมากเกินไป

จากตัวอย่างของตัวแปรชื่อ title สามารถแจกแจงที่มาได้ ดังนี้

ถ้ากำหนดให้ $(\mathbb{Z} \times CHAR) = \{(1,A),(2,B), (3,C)\}$ ดังนั้น $\mathbb{P}(\mathbb{Z} \times CHAR) = \{\varnothing, \{(1,A)\}, \{(2,B)\}, \{(3,B)\}, \{(1,A), (2,B)\}, ...\}$

กรณีที่ $title: \mathbb{P}(\mathbb{Z} \times CHAR)$ จะมีความหมายว่า $title \in \mathbb{P}(\mathbb{Z} \times CHAR)$ นั่นเอง ดังนั้น *title* จะมีค่าเป็นสมาชิกตัวหนึ่งตัวใดของ $\mathbb{P}(\mathbb{Z} \times CHAR)$

กรณีของตัวแปรชื่อ rating ก็เช่นเดียวกันมีการปรับเปลี่ยนการระบุชนิดของตัวแปร

จากเดิมระบุว่า $rating: People \rightarrow 0..10$

เปลี่ยนการระบุเป็น $rating: \mathbb{P}(People \times \mathbb{Z})$

จากตัวอย่างของตัวแปรชื่อ *rating* สามารถแจกแจงได้ดังนี้

ถ้ากำหนดให้ $(People \times \mathbb{Z})=\{(Sombat,1),(Somsri,5), \ldots\}$ ดังนั้นแล้ว $\mathbb{P}(People \times \mathbb{Z})=\{\varnothing,\{(Sombat,1)\},\{(Somsri,5)\},\{(Sombat,1), (Somsri,5)\} \}$

กรณี $rating: \mathbb{P}(People \times \mathbb{Z})$ จะมีความหมายว่า $rating\ \mathbb{P}(People \times \mathbb{Z})$ นั่นเอง

ดังนั้น *rating* อาจจะเป็น $\{(Sombat,1),(Somsri,5)\}$ ได้

โดยสรุปการเขียนข้อกำหนดเชิงรูปนัยควรเขียนประโยคการประกาศตัวแปรในส่วนประกาศ *Declaration* อย่างไม่เจาะจง และให้ไประบุเพิ่มในส่วน *Constraint Predicate* จะดีกว่า ทำให้อ่านเข้าใจได้ง่าย

รูปแบบของ State Schema

Z Schema ที่เขียนขึ้นนั้นมักเขียนในรูปแบบของ *State Schema* หรือ *Operation Schema* ทั้งนี้เพื่อใช้อธิบายส่วนประกอบของแบบจำลองซอฟต์แวร์และการทำงานของส่วนประกอบต่าง ๆเหล่านั้นให้ครบถ้วน

ต่อไปนี้เป็นรูปแบบทั่วไป (generic pattern) ของ *State Schema*

StateSpace

$x_1: S_1$

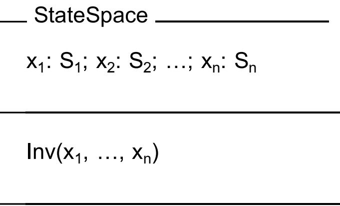
$x_2: S_2$

…

$x_n: S_n$

$Inv(x_1, \ldots, x_n)$

หรือเขียนแบบประหยัดพื้นที่ได้ดังนี้



หรือสามารถเขียน *Schema* แบบแนวระดับได้ดังนี้

StateSpace $\widehat{=}$ [x_1: S_1; x_2: S_2; …; x_n: S_n | Inv(x_1, …, x_n)]

รูปแบบของ Operation Schema

Operation

x_1: S_1; x_2: S_2; …; x_n: S_n

x'_1: S_1; x'_2: S_2; …; x'_n: S_n

i_1?: T_1; i_2?: T_2; …; i_m?: T_m

o_1!: U_1; o_2!: U_2; …; o_p!: U_p

Pre(i_1?, …, i_m?, x_1, …, x_n)

Inv(x_1, …, x_n)

Inv(x'_1, …, x'_n)

Op(i_1?, …, i_m?, x_1, …, x_n, x'_1, …, x'_n, o_p!, …, o_p!)

การประกาศตัวแปร X ใด ๆ ในส่วน Declaration

ใน *Z Schema* โดยปกติจะมีการเขียนส่วนประโยคประกาศในส่วนแรกหรือส่วนบนของ *Schema* แบบกล่อง และมีการเขียนประโยคเพรดิเคตต่าง ๆ ในส่วนที่สองหรือส่วนล่างของ *Schema* แบบกล่องคู่กันไป

การเขียนประโยคประกาศในส่วนแรกนั้น เราจะเขียนชื่อตัวแปร ตามด้วยเครื่องหมาย : และระบุชนิดของตัวแปรตามมาด้วย

กำหนดให้ประโยคดังนี้

x : SET1
y : SET2

เป็นการประกาศตัวแปรชื่อ *x* และ *y* โดยมีการระบุชนิดของตัวแปรชื่อ *SET1* และ *SET2* ตามลำดับ

ชนิดของตัวแปรที่ระบุ *SET1* นั้น เราจะพิจารณาเป็นเซตของสมาชิกที่ตัวแปร x จะมีค่าเป็นไปได้ ดังนั้นเรามักจะต้องเข้าใจด้วยความหมายของการประกาศข้างต้นไว้ดังนี้

$x : SET1$ จะหมายถึง $x \in SET1$

$y : SET2$ จะหมายถึง $y \in SET2$

ถ้า $SET1=\{ x1, x2, x3 \}$ แล้วนั้น x จะมีค่าเป็น $x1$ หรือ $x2$ หรือ $x3$ ได้ เราสามารถเขียนการประกาศในลักษณะเดียวกัน โดยระบุเซตไว้แตกต่างกัน เช่น

$x : \mathbb{N}$ จะหมายถึง $x \in \mathbb{N}$

จำนวนธรรมชาติจะเป็นตัวเลขที่มีค่าบวก (positive number) คือ $\{1, 2, 3, \ldots\}$ เพื่อใช้ในการนับ บางครั้งจำนวนธรรมชาติถูกกำหนดให้เป็นตัวเลขที่ไม่ติดลบ คือ $\{0, 1, 2, 3, \ldots\}$ ซึ่งรวมเลข 0 เข้าด้วย

เพื่อความเข้าใจร่วมกันในภาษาเซตใช้สัญกรณ์ที่แตกต่างกัน คือ

$\mathbb{N} = \{0, 1, 2, 3, \ldots\}$

$\mathbb{N}_1 = \{1, 2, 3, \ldots\}$

ชนิดตัวแปรเป็นกีเวนเซต

กำหนดให้ประโยคการประกาศต่อไปนี้

[People]

$friend : People$ จะหมายถึง $friend \in People$

ในภาษาเซต เราสามารถกำหนดเซตไว้ใช้เองได้เท่าที่ต้องการ โดยใช้เครื่องหมายวงเล็บ [...] เช่น [People] แสดงว่า *People* จะเป็นเซตที่กำหนดให้ไว้แล้ว เราเรียกว่ากีเวนเซต หรือเรียกว่า *Basic Type*

เมื่อกล่าวถึงกีเวนเซตแล้ว เราจะไม่สนใจต่อว่าเซตนั้นคืออะไร เราจะสนใจแค่ว่าตัวแปรที่ประกาศนั้นมีชนิดตัวแปรเป็นเซตที่กำหนดเท่านั้น โดยที่ไม่ทราบว่าสมาชิกมีค่าใดบ้าง ทำให้เราสามารถซ่อนรายละเอียดได้ก่อน เมื่อจำเป็นและถ้ามีเวลามากขึ้นเราจะมาแจกแจง ปรับเปลี่ยนส่วนที่ซ่อนอีกทีหนึ่ง คือการทำแบ่งรายละเอียด (refinement) ของข้อกำหนดนั่นเอง

ชนิดตัวแปรเป็นเซตที่เกิดจากความสัมพันธ์

ในภาษาเซดเราสามารถกำหนดชนิดตัวแปรเป็นเซตที่เกิดจากความสัมพันธ์ โดยเขียนประกาศไว้ดังนี้

$y: A \times B$

โดยกำหนดให้เซต A และเซต B และเซต $A \times B$ เป็นผลคูณคาร์ทีเชียนหรือเป็นความสัมพันธ์ที่เป็นไปได้ระหว่างเซตโดเมน A และเซตเรนจ์ B และเป็นชนิดของตัวแปร y และมีความหมายเทียบเท่ากับ

$y \in A \times B$

กล่าวคือ y เป็นสมาชิกของเซตของคู่ลำดับที่เป็นความสัมพันธ์ $A \times B$ นั่นเอง

ตัวอย่างคือ

$A = \{1,2,3\}$

$B = \{a,b\}$

$A \times B = \{(1,a), (1,b), (2,a), (2,b), (3,a), (3,b)\}$

และเพราะว่า $y: A \times B$ ดังนั้น y อาจจะมีค่าเป็น $(1,a)$ หรือ $(2,b)$

การใช้ Decoration

การประกาศตัวแปรในข้อกำหนดรูปนัยโดยทั่วไป ทำได้โดยการระบุชื่อตัวแปรตามด้วยชนิดตัวแปร ทั้งนี้การใช้งานตัวแปรในกรณีพิเศษสามารถทำได้โดยการทำ *Decoration* ตัวแปรด้วยสัญกรณ์พิเศษ

ในภาษาเซดผู้เขียนข้อกำหนดสามารถทำการ *Decoration* [18] เพิ่มเติมให้ตัวแปรมีความหมายพิเศษได้ดังต่อไปนี้

- การกำหนดตัวแปรแบบสถานะถัดไป (next state)
- การกำหนดตัวแปรนำเข้า
- การกำหนดตัวแปรแสดงผล

การกำหนดตัวแปรแบบ Next State

ให้ใช้เครื่องหมาย ‘ ต่อท้ายชื่อตัวแปรเพื่อกำหนดให้สามารถระบุตัวแปรแบบสถานะถัดไปที่มีความหมายว่าจะเก็บค่าหลังจากจบการทำงานใน *Schema* นั้นๆ ของตัวแปรสถานะปัจจุบัน

ตัวอย่างคือ

date: DATE

date’: DATE

โดยที่ตัวแปรชื่อ date เป็นตัวแปรสถานะปัจจุบัน เมื่อเริ่มต้นก่อนการทำงานใน *Schema* และเมื่อจบการทำงานใน *Schema* แล้วตัวแปรชื่อ *date’* เป็นตัวแปรแบบสถานะถัดไปที่เก็บค่าสุดท้าย

การกำหนดตัวแปรนำเข้า

ให้ใช้เครื่องหมาย ? ต่อท้ายชื่อตัวแปรเพื่อกำหนดให้สามารถระบุตัวแปรนำเข้า โดยตัวแปรนี้จะใช้เก็บค่าที่ได้รับป้อนเข้าสู่ระบบ ถือเป็นการทำการนำเข้าข้อมูลจากภายนอกระบบได้

ตัวอย่างคือ

MyFriend: ℙ*NAME*

name?: *NAME*

โดยที่ตัวแปรชื่อ *name?* เป็นตัวแปรนำเข้า โดยเมื่อเริ่มทำงานใน *Schema* จะมีการรอรับข้อมูลที่ต้องป้อนจากภายนอกระบบเพื่อนำไปใช้ต่อในประโยคข้อกำหนด เช่น การเปรียบเทียบข้อมูลนำเข้ากับเซตเป้าหมายที่สนใจ

name? ∈ *MyFriend* เป็นต้น

การกำหนดตัวแปรแสดงผล

ให้ใช้เครื่องหมาย ! ต่อท้ายชื่อตัวแปรเพื่อกำหนดให้สามารถระบุตัวแปรแสดงผล โดยตัวแปรนี้จะใช้เก็บค่าที่นำไปแสดงผลสู่ภายนอกระบบได้ ถือเป็นการนำผลลัพธ์ไปสู่โลกภายนอก

ตัวอย่างคือ

name!: NAME

โดยที่ตัวแปรชื่อ *name!* เป็นตัวแปรแสดงผล ประโยคข้อกำหนดใน *Schema* จะระบุให้มีการผลลัพธ์ของการทำงานได้

ตัวอย่างคือ

name!: NAME

Somchai ∈ *NAME*

name! = *Somchai*

โดยที่ระบบจะแสดงค่า *name!* ออกเป็นค่า *Somchai* หลังการทำงานของ *Schema* ในที่สุด

กีเวนเซต

กีเวนเซตเป็นการนิยามชนิดข้อมูลนามธรรมที่ใช้ในข้อกำหนด บางครั้งผู้เขียนข้อกำหนดยังไม่ต้องการลงรายละเอียดของเซตที่ใช้เป็นชนิดของตัวแปรใดที่อ้างถึง เช่น เซตของชื่อ เซตของวันที่ เป็นต้น ผู้เขียนไม่จำเป็นต้องกำหนดหรือนิยามในขณะนั้นว่า เซตของชื่อ หรือเซตของวันที่ มีสมาชิกเป็นอย่างไร ผู้เขียนสามารถระบุให้มีกีเวนเซตที่ใช้ชื่อตรงตามที่ต้องการ เช่น กีเวนเซตชื่อเรียกว่า "*NAME*" ที่เขียนด้วยสัญกรณ์ [NAME] และกีเวนเซตวันที่เรียกว่า "*DATE*" ที่

เขียนด้วยสัญกรณ์ [DATE] ได้และนำกีเวนเซตทั้งสองไปใช้อ้างอิงได้ทันทีโดยไม่ต้องอธิบายเพิ่มเติม คือ

[NAME]

[DATE]

n1: NAME

d1: DATE

ซึ่งหมายถึงมีตัวแปร *n1* และตัวแปร *d1* มีชนิดเป็นกีเวนเซต *NAME* และ *DATE*

การกำหนดชนิดตัวแปรแบบ Free type

ผู้เขียนข้อกำหนดสามารถกำหนดชนิดตัวแปรแบบ *Free type* ได้ โดยใช้สัญกรณ์ดังต่อไปนี้

FreeTypeName ::= Value_1 | Value_2 | … | Value_n

ยกตัวอย่างคือ

DayOfWeek ::= *Monday*|*Tuesday*|*Wednesday*|*Thursday*|*Friday*|*Saturday*|*Sunday*

d1: DayOfWeek

d2: DayOfWeek

โดย *DayOfWeek* เป็นชนิดตัวแปรแบบ *Free type* และมีตัวแปรชื่อ *d1* และ *d2* มีชนิดตัวแปรเป็น DayOfWeek เราจะทราบได้เลยว่า ค่าที่เป็นไปได้ของตัวแปร *d1* และ *d2* จะมีค่าเป็นหนึ่งใน 7 ค่าที่กำหนดไว้เท่านั้น คือ

$d1 \in DayOfWeek$

$d1 = Monday$

$d2 \in DayOfWeek$

$d2 = Thursday$

ภาษาที่ใช้ในการพัฒนาโปรแกรมมีขีดความสามารถในลักษณะเดียวกัน คือ การกำหนดชนิดตัวแปรแบบ *Enumeration* ซึ่งผู้เขียนข้อกำหนดสามารถเลือกกำหนดในข้อกำหนดเชิงรูปนัยได้

การทำการนำเข้า Schema

ตามที่กล่าวมาแล้วว่าข้อดีของภาษาเซด คือ การสนับสนุนการเขียนข้อกำหนดแบบโครงสร้าง ผู้เขียนข้อกำหนดสามารถเขียน *Schema* ย่อย ๆ ได้ก่อน และนำ *Schema* ที่เขียนไว้แล้วไปใช้ใหม่ได้

การทำการนำเข้า *Schema* (schema inclusion) [18] คือการนำข้อกำหนดใน *Schema* ที่กำหนดไว้แล้วมารวมเข้ากัน เพื่อลดภาระในการเขียน

ซ้ำซ้อน โดยจะมีการรวมประโยคในส่วนประกาศเข้าด้วยกัน และรวมประโยคในส่วนเพรดิเคต เข้าด้วยกัน

- การทำการนำเข้า *Schema* ทำได้สองแบบคือ
- การทำการนำเข้า *Schema* แบบ *Delta*
- การทำการนำเข้า *Schema* แบบ *Xi*

การทำการนำเข้า Schema แบบ Delta

การทำการนำเข้า *Schema* แบบ *Delta* คือการนำ *Schema* เดิมที่กำหนดไว้แล้วมาใช้สร้าง *Schema* ใหม่พร้อมให้มีการเปลี่ยนแปลงค่าตัวแปรได้ ใช้สัญกรณ์ประกาศ ดังนี้

ΔSchemaA โดย *SchemaA* เป็น *Schema* ที่กำหนดไว้แล้วก่อนหน้านี้ และสัญกรณ์นี้จะหมายถึง

```
┌─ ΔSchemaA ────────────
│ SchemaA
│ SchemaA'
└───────────────────────
```

SchemaA เดิมที่กำหนดไว้ถูกรวมเข้ากับ *SchemaA'* ดังนี้

```
┌─ SchemaA ─────────────
│ a: A
│ b: B
├───────────────────────
│ Inv(a,b)
└───────────────────────
```

```
┌─ SchemaA' ────────────
│ a': A
│ b': B
├───────────────────────
│ Inv(a', b')
└───────────────────────
```

ดังนั้นผลลัพธ์ก็คือ

ΔSchemaA

a: A

b: B

a’: A

b’: B

Inv(a, b)

Inv(a’, b’)

วิธีการใช้งาน ΔSchemaA ก็คือการบรรจุประโยคนี้เข้าไปในส่วนประกาศของ Schema ใหม่ใดๆ ที่ต้องการ ดังนี้

SchemaB

ΔSchemaA

c: C

Inv(c)

Op(a, b, a’, b’, c)

ผลลัพธ์จะเป็นดังนี้

SchemaB

a: A

b: B

a’: A

b’: B

c: C

Inv(a, b)

Inv(a’, b’)

Inv(c)

Op(a, b, a’, b’, c)

ดังนั้น *SchemaB* จะเป็นการรวม *SchemaA* และ *SchemaA'* เข้ามาด้วยกัน

การทำการนำเข้า Schema แบบ Xi

การทำการนำเข้า *Schema* แบบไม่มีการเปลี่ยนแปลงค่าตัวแปร ทำได้โดยระบุสัญกรณ์ที่มีเครื่องหมาย Ξ ปรากฏดังนี้

ΞSchemaA โดย *SchemaA* เป็น *Schema* ที่กำหนดไว้ก่อนหน้าและได้รับการรวมเข้ามาแบบไม่มีการเปลี่ยนแปลงค่าตัวแปร

```
┌── ΞSchemaA ──────────
│ ΔSchemaA
├──────────────────────
│ θSchemaA = θSchemaA'
└──────────────────────
```

ยกตัวอย่างดังนี้

```
┌── SchemaA ───────────
│ a: A
│ b: B
├──────────────────────
│ Inv(a,b)
└──────────────────────
```

```
┌── ΔSchemaA ──────────
│ a: A
│ b: B
│ a': A
│ b': B
├──────────────────────
│ Inv(a, b)
│ Inv(a', b')
└──────────────────────
```

ดังนั้นผลลัพธ์ของ ΞSchemaA คือ

```
┌─ ΞSchemaA ──────────────
│ a: A
│ b: B
│ a': A
│ b': B
├─────────────────────────
│ Inv(a, b)
│ Inv(a', b')
│ a = a'
│ b = b'
└─────────────────────────
```

โดยที่ θSchemaA หมายถึงตัวแปร *a* และตัวแปร *b* ที่ประกาศไว้และ θSchemaA' หมายถึงตัวแปร *a'* และตัวแปร *b'* ที่ประกาศไว้

ดังนั้น θSchemaA = θSchemaA' ก็จะหมายถึงประโยคต่อไปนี้

$a = a'$

$b = b'$

ซึ่งแสดงว่าตัวแปรที่ประกาศไว้ทั้งหมดไม่มีการเปลี่ยนแปลงหลังจบการทำงานใน *Schema*

วิธีการใช้งานคือการประกาศ ΞSchemaA ไว้ในส่วนประกาศดังนี้

```
┌─ SchemaC ───────────────
│ ΞSchemaA
│ c: C
├─────────────────────────
│ Inv(c)
│ Op(a, b, a', b', c)
└─────────────────────────
```

ผลลัพธ์การรวม *SchemaA* เข้าแบบไม่มีการเปลี่ยนแปลงได้ดังนี้

```
┌─ SchemaC ──────────────────
│ a: A
│ b: B
│ a': A
│ b': B
│ c: C
├────────────────────────────
│ Inv(a, b)
│ Inv(a', b')
│ a = a'
│ b = b'
│ Inv(c)
│ Op(a, b, a', b', c)
└────────────────────────────
```

ฟังก์ชัน Projection

กำหนดให้มี *Object* ใด ๆ ที่ประกอบด้วยส่วนประกอบหลายส่วน เช่น คู่ลำดับ *(x, y)* ซึ่งเป็น *Object* ที่มีส่วนประกอบสองส่วน คือ ส่วนแรกของคู่ลำดับ x และส่วนที่สองของคู่ลำดับ y เป็นต้น

ฟังก์ชัน *Projection* [18,19] เป็นฟังก์ชันที่ทำการฉายภาพเฉพาะส่วนประกอบใดๆ ส่วนหนึ่งที่สนใจเท่านั้นได้ เช่น การฉายภาพเฉพาะส่วนแรกของคู่ลำดับ หรือการฉายภาพเฉพาะส่วนที่สองของคู่ลำดับเท่านั้น เป็นต้น

ในภาษาเซดมีฟังก์ชันแบบ *Projection* ที่นิยมใช้สำหรับคู่ลำดับดังนี้

- ฟังก์ชัน *first((x, y))* ซึ่งจะให้ค่า *x* ออกมา
- ฟังก์ชัน *second((x, y))* ซึ่งจะให้ค่า *y* ออกมา

ยกตัวอย่างคือ

กำหนดให้ *(1, a)* เป็นคู่ลำดับใดๆ

- *first((1, a)) = 1*
- *second((1, a)) = a*

การทำ Abbreviation

เพื่อให้การเขียนข้อกำหนดแต่ละประโยคให้ง่ายและสะดวก ในภาษาเซดเราสามารถกำหนดตัวย่อสำหรับแทนค่าข้อมูลหรือนิยามต่างๆที่ต้องการใช้งานบ่อยครั้งได้ โดยใช้เครื่องหมาย == ดังนี้

ตัวอย่างคือ

$ASCII == 0..255$

$RelationAB == A \longleftrightarrow B$

โดยที่ *ASCII* จะเป็นตัวแทนของค่าข้อมูล *0..255* หรือ *RelationAB* จะเป็นตัวแทนของความสัมพันธ์ $A \longleftrightarrow B$

วิธีการใช้งานตัวย่อ *ASCII* และ *RelationAB* เป็นไปได้ดังนี้

code: *ASCII*

หรือ

AtoB: *RelationAB*

เครื่องมือที่มีใช้ในภาษาเซด

นอกจากสัญกรณ์มาตรฐานทั่วไปที่ใช้ในการประกาศเซต ความสัมพันธ์ ประพจน์และเพรดิเคตแล้ว ภาษาเซดยังมีเครื่องมือที่เป็นสัญกรณ์พิเศษเพิ่มเติม เพื่ออำนวยความสะดวกให้ผู้เขียนข้อกำหนดใช้ [18]

Identity Relation

เป็นความสัมพันธ์จากเซตโดเมนหนึ่งใดไปยังเซตเดิม กล่าวคือเซตของโดเมนและเซตของเรนจ์เป็นเซตเดียวกัน ใช้สัญกรณ์

$id\ A$

โดย A เป็นเซตโดเมนและเรนจ์ในขณะเดียวกัน

Forward Relational Composition

เป็นความสัมพันธ์แบบประกอบไปข้างหน้า กล่าวคือ มีความสัมพันธ์ Q เกิดขึ้นและมีความสัมพันธ์ R เกิดขึ้นตามมาประกอบรวมกันใช้สัญกรณ์

$Q \fatsemi R$

เซตโดเมนของความสัมพันธ์นี้จะเป็นเซตย่อยของเซตโดเมนของความสัมพันธ์แรก Q และเซตเรนจ์ของความสัมพันธ์นี้จะเป็นเซตย่อยของความสัมพันธ์ที่ตามมาคือ R ดังข้อกำหนดดังนี้

$dom\ (Q \fatsemi R) \subseteq dom\ Q$

$ran\ (Q \fatsemi R) \subseteq ran\ R$

เราเรียกความสัมพันธ์นี้อีกชื่อว่า ความสัมพันธ์แบบประกอบ (relational composition)

Backward Relational Composition

เป็นความสัมพันธ์แบบประกอบย้อนกลับ กล่าวคือ มีความสัมพันธ์ Q เกิดขึ้นก่อนและมีความสัมพันธ์ R เกิดขึ้นตามมาประกอบรวมกันใช้สัญกรณ์ดังนี้

$R \circ Q$

เซตโดเมนของความสัมพันธ์นี้จะเป็นเซตย่อยของเซตโดเมนของความสัมพันธ์ที่เกิดก่อนแต่เขียนตามหลังคือ ความสัมพันธ์ชื่อ Q และเซตเรนจ์ของความสัมพันธ์นี้จะเป็นเซตย่อยของความสัมพันธ์ที่เขียนด้านหน้า ซึ่งก็คือความสัมพันธ์ชื่อ R ดังข้อกำหนดดังนี้

$dom\ (R \circ Q) \subseteq dom\ Q$

$ran\ (R \circ Q) \subseteq ran\ R$

ความสัมพันธ์แบบประกอบย้อนกลับจะมีความหมายเหมือนกับการเขียนความสัมพันธ์แบบประกอบไปข้างหน้าคือ

$(R \circ Q)$ จะมีผลเท่ากับ $Q \fatsemi R$

เราจะเห็นการเขียนข้อกำหนดแบบไปข้างหน้ามากกว่าในภาษาเซต

Domain Restriction

เป็นความสัมพันธ์ที่ถูกจำกัดลักษณะของสมาชิก โดยเลือกเฉพาะคู่ลำดับ $(x,\ y)$ ใด ๆ ของความสัมพันธ์ R ที่มีค่า x เป็นสมาชิกของเซต A เราเขียนประโยคได้ดังนี้

$A \lhd R$

โดยที่เซต A เป็นเซตที่มีสมาชิกที่ใช้เป็นเงื่อนไขในการคัดเลือกสมาชิกของความสัมพันธ์ตั้งต้น R ผลลัพธ์ที่ได้จะเป็นความสัมพันธ์ใหม่ที่เริ่มจากความสัมพันธ์ R แต่มีการคัดเลือกเฉพาะคู่ลำดับที่มีตัวแรกเป็นสมาชิกอยู่ในเซต A เท่านั้น คู่ลำดับที่เหลือที่ไม่เข้าข่ายให้ตัดออกไป

ตัวอย่างคือ กำหนดให้

$R = \{(a, b), (c, d), (e, f)\}$

$A = \{c, e\}$

$A \lhd R = \{(c, d), (e, f)\}$

โดยที่ (a, b) ถูกตัดออกไปเพราะ $a \notin A$ นั่นเอง

การเขียนข้อกำหนดจะใช้เพื่อการทำ *Filter* สมาชิกที่ต้องการเท่านั้น

ตัวอย่างคือ กำหนดให้

$Friend = \{(Somchai, 100), (Sombat, 200), (Sirichai, 300)\}$

$Target = \{Sirichai, Somchai\}$

$Target \lhd Friend = \{(Somchai, 100), (Sirichai, 300)\}$

โดยจะได้เซตคู่ลำดับที่เลือกเฉพาะสมาชิกของโดเมนที่มีค่าเท่ากับสมาชิกของ Target เท่านั้น

เราเขียนนิยามในรูปแบบของเซตได้ดังนี้

$$A \lhd R = \{ x: X; y: Y \mid x \mapsto y \in R \land x \in A \bullet x \mapsto y \}$$

Domain Anti-Restriction

เป็นความสัมพันธ์ที่ถูกจำกัดลักษณะของสมาชิก โดยเลือกเฉพาะคู่ลำดับ (x, y) ใด ๆของความสัมพันธ์ตั้งต้น R ที่มีค่า x ไม่เป็นสมาชิกของเซต A โดยเราเขียนประโยคดังนี้

$$A \ntriangleleft R$$

ความหมายจะตรงกันข้ามกับการทำ *Domain Restriction* ผลลัพธ์ใหม่จะได้จากการเลือกคู่ลำดับจาก R ที่มีค่าตัวแรกไม่เป็นสมาชิกของเซต A มาเท่านั้น

ตัวอย่างคือ กำหนดให้

$$R = \{(a, b), (c, d), (e, f)\}$$

$$A = \{c, e\}$$

$$A \ntriangleleft R = \{(a, b)\}$$

โดยที่ (c, d) และ (e, f) ถูกตัดออกไปเพราะ $c, e \in A$ นั่นเอง

ในทางปฏิบัติ เรามักจะใช้ประโยคนี้เพื่อทำการลบคู่ลำดับที่เป็นสมาชิกของความสัมพันธ์ตั้งต้นเดิมที่ไม่ต้องการออกไป โดยให้กำหนดเซตเงื่อนไขที่บรรจุสมาชิกของตัวแรกของคู่ลำดับที่ไม่ต้องการ โดยทั่วไปจะเป็นคีย์ของข้อมูลและทำการลบออก ดังตัวอย่างต่อไปนี้

กำหนดให้

$$File = \{(key1, data1), (key2, data2), (key3, data3)\}$$

$$DelKey = \{key2, key1\}$$

$$DelKey \ntriangleleft File = \{(key3, data3)\}$$

โดยที่ $(key1, data1)$ และ $(key2, data2)$ ถูกตัดออกไปเพราะ $key1, key2 \in DelKey$ นั่นเอง

เราเขียนนิยามในรูปแบบของเซตได้ดังนี้

$$A \ntriangleleft R = \{ x: X; y: Y \mid x \mapsto y \in R \land x \notin A \bullet x \mapsto y \}$$

Range Restriction

เป็นความสัมพันธ์ที่ถูกจำกัดลักษณะของสมาชิก โดยเลือกเฉพาะคู่ลำดับ *(x, y)* ใดๆของความสัมพันธ์ R ที่มีค่า y เป็นสมาชิกของเซต A เราเขียนประโยคได้ดังนี้

$R \rhd A$

โดยที่เซต A เป็นเซตที่มีสมาชิกที่ใช้เป็นเงื่อนไขในการคัดเลือกสมาชิกของความสัมพันธ์ตั้งต้น R ผลลัพธ์ที่ได้จะเป็นความสัมพันธ์ใหม่ที่เริ่มจากความสัมพันธ์ R แต่มีการคัดเลือกเฉพาะคู่ลำดับที่มีตัวหลังเป็นสมาชิกอยู่ในเซต A เท่านั้น คู่ลำดับที่เหลือที่ไม่เข้าข่ายให้ตัดออกไป

ตัวอย่างคือ กำหนดให้

$R = \{(a, b), (c, d), (e, f)\}$

$A = \{b, f\}$

$R \rhd A = \{(a, b), (e, f)\}$

โดยที่ *(c, d)* ถูกตัดออกไปเพราะ $d \notin A$ นั่นเอง

ในทางปฏิบัติ เรามักจะเขียนข้อกำหนดใช้เพื่อการทำ *Filter* สมาชิกที่ต้องการเท่านั้น

ตัวอย่างคือ กำหนดให้

$$Friend = \{(Somchai, Bangkok), (Sombat, Thonburi), (Sirichai, Bangkok)\}$$

$$TargetProvince = \{Bangkok\}$$

$$Friend \rhd TargetProvince = \{(Somchai, Bangkok), (Sirichai, Bangkok)\}$$

โดยจะได้เซตคู่ลำดับที่เลือกเฉพาะเพื่อนที่อาศัยอยู่ที่ *Bangkok* เท่านั้น หรือกล่าวได้ว่าจะได้เซตคู่ลำดับที่มีส่วนที่สองมีค่าเป็นสมาชิกของ *TargetProvince* เท่านั้น

เราเขียนนิยามในรูปแบบของเซตได้ดังนี้

$$R \rhd B = \{ x: X; y: Y \mid x \mapsto y \in R \land y \in B \bullet x \mapsto y \}$$

Range Anti-Restriction

เป็นความสัมพันธ์ที่ถูกจำกัดลักษณะของสมาชิก โดยเลือกเฉพาะคู่ลำดับ *(x, y)* ใดๆของความสัมพันธ์ตั้งต้น R ที่มีค่า y ไม่เป็นสมาชิกของเซต A โดยเราเขียนประโยคดังนี้

$R \ntriangleright A$

ความหมายจะตรงกันข้ามกับการทำ *Range Restriction* ผลลัพธ์ใหม่จะได้จากการเลือกคู่ลำดับจาก R ที่มีค่าตัวหลังไม่เป็นสมาชิกของเซต A มาเท่านั้น

ตัวอย่างคือ กำหนดให้

$R = \{(a, b), (c, d), (e, f)\}$

$A = \{d\}$

$R \rhd A = \{(a, b), (e, f)\}$

โดยที่ (c, d) ถูกตัดออกไปเพราะ $d \in A$ นั่นเอง

ในทางปฏิบัติเรามักจะใช้ประโยคนี้เพื่อทำการลบคู่ลำดับที่เป็นสมาชิกของความสัมพันธ์ตั้งต้นเดิมที่ไม่ต้องการออกไป โดยให้กำหนดเซตเงื่อนไขที่บรรจุสมาชิกของตัวหลังของคู่ลำดับที่ไม่ต้องการดังตัวอย่างต่อไปนี้

กำหนดให้

$Friend = \{(Somchai, Bangkok), (Sombat, Thonburi), (Sirichai, Bangkok)\}$

$DelProvince = \{Bangkok\}$

$Friend \rhd DelProvince = \{(Sombat, Thonburi)\}$

โดยที่ *(Somchai, Bangkok)* และ *(Sirichai, Bangkok)* ถูกตัดออกไปเพราะมี $Bangkok \in DelProvince$ นั่นเอง

เราเขียนนิยามในรูปแบบของเซตได้ดังนี้

$R \rhd B = \{ x: X; y: Y \mid x \mapsto y \in R \land y \notin B \bullet x \mapsto y \}$

Relational Image

ในภาษาเซต ผู้เขียนสามารถใช้สัญกรณ์พิเศษ เพื่อหาค่า *Relational Image* ของความสัมพันธ์ R ใดๆ เมื่อมีสมาชิกในเซต A เป็นเป้าหมายในการหาค่าดังนี้

$R (\!| A |\!) = ran\ (A \lhd R)$

โดยนิยามของ $R (\!| A |\!)$ จะหมายถึงเซตของเรนจ์ของคู่ลำดับที่ได้จากการทำ *Domain Restriction* ของความสัมพันธ์ R ด้วยเซต A

ตัวอย่างคือ กำหนดให้

$R = \{(a, b), (c, d), (e, f)\}$

$A = \{a, e\}$

$R (\!| A |\!) = \{b, f\}$

Relational Iteration

การวนซ้ำของความสัมพันธ์ R ที่เกิดขึ้นอย่างต่อเนื่องสามารถเขียนได้ด้วยสัญกรณ์ดังนี้

$iter\ n\ R$

โดยใช้คำว่า *iter* ตามด้วยจำนวนการวนซ้ำ n ครั้ง และตามด้วยชื่อของความสัมพันธ์ที่ต้องการทำวนซ้ำ

ยกตัวอย่างคือ

$iter\ 3\ R$

หมายถึงการทำความสัมพันธ์ R วนซ้ำ 3 รอบต่อเนื่องกัน โดยมีความหมายเหมือนกับ $R3$

บางครั้งเราอาจจะเขียนด้วยสัญกรณ์พิเศษได้ดังนี้

$R ⨾ R ⨾ R$

Lambda Notation

กำหนดให้ f เป็นฟังก์ชันดังนี้

$f = \{x: X \mid p \bullet x \mapsto e\}$

นั่นคือ f เป็นฟังก์ชันที่ map ค่า x ที่เป็นสมาชิกของเซต X โดยเลือกเฉพาะค่า x ที่ทำให้เงื่อนไข p เป็นจริงเท่านั้น ไปยังปลายทางที่เป็นรูปแบบตามนิพจน์ e

เราเขียนฟังก์ชัน f ได้อีกวิธีหนึ่งโดยใช้สัญกรณ์ *Lambda* โดยการใช้ฟังก์ชัน Lambda ได้รับการกำหนดไว้ดังนี้

$\lambda\ declaration \mid constraint \bullet result$

λ เป็นฟังก์ชันที่ *map* ส่วนประกอบที่ได้รับการประกาศไว้ใน *Declaration* ทุกตัวที่ทำให้เงื่อนไขบังคับ *constraint* เป็นจริงเสมอ ไปยังนิพจน์ปลายทางที่ระบุในนิพจน์ *result*

เราสามารถนิยามฟังก์ชัน f ข้างต้นได้ใหม่โดยใช้ *Lambda* ดังนี้

$f = \lambda x: X \mid p \bullet e$

ตัวอย่างคือ

เราสามารถนิยาม Schema สำหรับการทำฟังก์ชัน double ได้ดังนี้

double: $\mathbb{N} \leftrightarrow \mathbb{N}$
double $= \lambda$ m: $\mathbb{N} \bullet$ m+m

โดยเราใช้งานคือ

double 5 จะมีค่าเป็น *10*

double 10 จะมีค่าเป็น *20*

ถ้าใช้การนิยามแบบเซต จะเขียนได้ดังนี้

$double = \{m: \mathbb{N} \mid true \bullet m \mapsto m+m\}$

การอธิบาย Axiomatic ของระบบ

การอธิบาย *Axiom* ของระบบ [19] ทำได้โดย

Declaration
$Predicate_1$; …; $Predicate_n$

หรือ

Declaration

เราเขียน *Axiomatic Schema* เพื่อใช้อธิบาย *Global Variables*

square: $\mathbb{N} \rightarrow \mathbb{N}$
$\forall n: \mathbb{N} \bullet square(n) = n*n$

การอธิบาย Schema ของระบบ

การอธิบาย *Schema* ปกติของระบบ [19] ทำได้โดย

SchemaName
Declaration
Predicates

SchemaName $\widehat{=}$ [Declaration | Predicates]

การอธิบาย Generic Schema ของระบบ

การอธิบาย *Schema* ทั่วไปของระบบ [19] ทำได้โดยมีการระบุ *Parameters Ident…*

```
┌── SchemaName [Ident₁, …, Identₙ ] ─────
│ Declaration
├────────────────────────────
│ Predicates
└────────────────────────────
```

การอธิบาย Generic Constants ของระบบ

การอธิบาย *Constants* เหมือน *Axiomatic with Parameters* [19]

```
╒══ [Ident₁, …, Identₙ ] ═════
│ Declaration
├─────────────────────
│ Predicates
└─────────────────────
```

4.11 เอเอ็มเอ็น

เอเอ็มเอ็น [22] คือ สัญกรณ์ที่ใช้ในการเขียนข้อกำหนดเชิงรูปนัยที่ใช้ในวิธีเชิงรูปนัยแบบบี จะประกอบด้วยสถานะและการดำเนินการที่ใช้ในการเปลี่ยนค่าสถานะของระบบ การศึกษาระบบจะมักจะเกิดจากการระบุคุณสมบัติสถิต (static property) และคุณสมบัติพลวัต (dynamic property) นั่นเอง

คุณสมบัติสถิตของระบบ คือ สถานะทั้งหลายที่ระบบพึงจะมี ส่วนคุณสมบัติพลวัต คือ การดำเนินการที่มีใช้ในระบบเพื่อเปลี่ยนแปลงค่าของสถานะของระบบนั่นเอง

โครงสร้างของเครื่องเชิงนามธรรม

เครื่องเชิงนามธรรม (abstract machine) คือ ตัวแทนที่ใช้แทนระบบที่เราสนใจอธิบายนั่นเอง ในวิธีเชิงรูปนัยแบบบีจะใช้ภาษาที่มีสัญกรณ์พิเศษเขียนเครื่องเชิงนามธรรม ทั้งนี้สัญกรณ์ที่ใช้มีลักษณะคล้ายกับสัญกรณ์ในภาษาเซด

[19] เนื่องจากผู้ออกแบบวิธีเชิงรูปนัยแบบบี เป็นคนเดียวกันกับผู้ที่ออกแบบภาษาเซดนั่นเอง คือ *J.R. Abrial*

เช่นเดียวกันกับภาษาเซด สัญกรณ์เครื่องเชิงนามธรรมจะอาศัยพื้นฐานความรู้ทางคณิตศาสตร์และตรรกศาสตร์ และใช้ทฤษฎีของเซต ตรรกศาสตร์เชิงประพจน์ แคลคูลัสภาคแสดง ฟังก์ชัน ความสัมพันธ์ เป็นต้น

เครื่องเชิงนามธรรมจะเป็นตัวแทนของระบบหนึ่ง หรือระบบย่อยแต่ละระบบย่อย โดยเขียนอนุประโยค (clause) ที่สำคัญเพื่ออธิบายคุณสมบัติทั้งสถิตและพลวัตของระบบได้ เช่น อนุประโยคชื่อ *MACHINE* หรืออนุประโยคชื่อ *VARIABLE* เป็นต้น

โครงสร้างของข้อกำหนดจะประกอบด้วยมอดูล ที่เขียนขึ้นต้นด้วยอนุประโยค *MACHINE* เสมอ แต่ละมอดูลจะอ้างถึงระบบย่อยแต่ละส่วน โดยภายใน *MACHINE* จะมีการประกาศตัวแปร สถานะ และการดำเนินการที่เกี่ยวข้อง

ปัจจุบันแนวคิดเรื่องเทคโนโลยีเชิงวัตถุเป็นที่กล่าวถึงกัน เราสามารถเขียนเครื่องเชิงนามธรรมให้เป็นมอดูลและเป็นตัวแทนของแต่ละวัตถุได้เป็นอย่างดี โดยแต่ละวัตถุสามารถอ้างถึงหรือร้องขอให้วัตถุที่เกี่ยวข้องช่วยทำงานให้ได้

อนุประโยคอธิบายคุณสมบัติสถิต

อนุประโยคที่ใช้อธิบายคุณสมบัติสถิตที่สำคัญ มีดังนี้

- อนุประโยคชื่อ *MACHINE*
- อนุประโยคชื่อ *VARIABLES*
- อนุประโยคชื่อ *INVARIANT*

โดยมีตัวอย่างการเขียนอนุประโยคดังต่อไปนี้

```
MACHINE
        booking
VARIABLES
        seat
INVARIANT
        seat ∈ ℕ
END
```

โดยที่อนุประโยคชื่อ *MACHINE* มักจะใช้เป็นจุดเริ่มมอดูลเสมอ และให้มีการระบุชื่อของเครื่องเชิงนามธรรมได้ อนุประโยคชื่อ *VARIABLES* เป็นอนุประโยคที่ใช้ระบุชื่อตัวแปรของสถานะของระบบที่เราสนใจ

ส่วนอนุประโยค *INVARIANT* เป็นการระบุตัวยืนยงข้อมูล ที่มีสำหรับตัวแปรสถานะทุกตัวของระบบ โดยการระบุตัวยืนยงทำได้โดยการเขียนเพรดิเคตที่มีตัวแปรที่สนใจปรากฏอยู่ด้วย เช่น $seat \in \mathbb{N}$ เป็นประโยคภาคแสดงสำหรับตัวแปรสถานะชื่อ *seat* เป็นต้น

อนุประโยคอธิบายคุณสมบัติพลวัต

อนุประโยคที่ใช้อธิบายคุณสมบัติพลวัตที่สำคัญ มีดังนี้

- อนุประโยคชื่อ *OPERATIONS*

โดยมีตัวอย่างการเขียนอนุประโยคดังต่อไปนี้

```
MACHINE
        booking
VARIABLES
        seat
INVARIANT
        seat ∈ ℕ
OPERATIONS
        book ≙ ...;
        cancel ≙ ...
END
```

จากตัวอย่างที่กล่าวมาเห็นได้ว่ามีการเพิ่มอนุประโยคชื่อ *OPERATIONS* เข้าไปเพื่ออธิบายการดำเนินการที่มีในมอดูลหรือเครื่องเชิงนามธรรมได้ จะเห็นได้ว่ามีการดำเนินการชื่อ *book* และการดำเนินการชื่อ *cancel* ที่เขียนไว้

การดำเนินการชื่อ *book* เป็นการจองที่นั่งและการดำเนินการชื่อ *cancel* เป็นการยกเลิกการจองที่นั่ง ซึ่งการดำเนินการทั้งสองจะทำการปรับเปลี่ยน และจัดการตัวแปรสถานะ *seat* ที่มีอยู่ (เนื้อหาของการดำเนินการทั้งสองถูกละไว้เพื่อง่ายในการเข้าใจ และจะกล่าวถึงในโอกาสถัดไป)

การเขียนการดำเนินการ

จากตัวอย่างมอดูลชื่อ *booking* มีการดำเนินการชื่อ *book* และ *cancel* เรามักจะใช้การดำเนินการเพื่อการปรับเปลี่ยนค่าตัวแปรสถานะ *seat* ได้ ดังนี้

```
book ≙
  BEGIN
      seat := seat-1
  END;
cancel ≙
  BEGIN
      seat := seat+1
  END
```

การดำเนินการชื่อ *book* เป็นการจองที่นั่งซึ่งผลของการจองที่นั่งจะทำให้จำนวนที่นั่งที่เป็นค่าในตัวแปรสถานะชื่อ *seat* มีค่าลดลงหนึ่งเสมอ คือ *seat := seat-1* เมื่อการจองที่นั่งทำงานเสร็จ

ในขณะที่การดำเนินการชื่อ *cancel* เป็นการยกเลิกการจองที่นั่งซึ่งจะทำให้ค่าในตัวแปรสถานะชื่อ *seat* มีค่าเพิ่มขึ้นหนึ่งเสมอ คือ *seat := seat+1* เพราะการยกเลิกทำให้มีที่นั่งเพิ่มเสมอ

การระบุเงื่อนไขก่อน

การเขียนข้อกำหนดให้กับการดำเนินการทั้งสองดังที่กล่าวมาแล้วนั้น การดำเนินการอาจจะมีความผิดพลาดได้ เนื่องจากการทำ *book* อาจจะทำการจองที่นั่งเกินจำนวนที่นั่งที่มีอยู่ก็ได้ เช่นเดียวกับการดำเนินการชื่อ *cancel* อาจจะทำการยกเลิกที่นั่งผิดพลาด โดยมีการยกเลิกเกินกว่าที่จะทำได้ ดังนั้นจึงจำเป็นต้องมีการระบุเงื่อนไขก่อนเสมอ เพื่อให้มีการทดสอบเงื่อนไขก่อนดำเนินการจริง เช่น ก่อนการจองที่นั่งให้ทดสอบก่อนว่า จำนวนที่นั่ง *seat* ต้องมีค่ามากกว่าศูนย์เสมอ คือ $0<seat$ หรือก่อนการยกเลิกที่นั่งเพื่อคืนจำนวนที่นั่งต้องมีเงื่อนไขก่อนว่าจำนวนที่นั่ง *seat* ต้องไม่เกินจำนวนสูงสุดที่ควรมี คือ $seat<max_seat$ เป็นต้น

เราทำการเพิ่มเงื่อนไขก่อนในการกำหนดการดำเนินการได้ ดังตัวอย่างต่อไปนี้ โดยถ้ากำหนดให้ *max_seat* เป็นค่าคงที่ที่เป็นจำนวนที่นั่งที่มีอยู่ทั้งหมดแล้ว

```
book ≙
  PRE
      0<seat
  THEN
      seat := seat-1
  END;
```

```
cancel ≙
  PRE
        seat<max_seat
  THEN
        seat := seat+1
  END
```

จะเห็นได้ว่าการดำเนินการชื่อ *book* และชื่อ *cancel* มีการทดสอบเงื่อนไขก่อนเสมอ ทำให้ข้อกำหนดที่เขียนขึ้นมีความสมบูรณ์มากยิ่งขึ้น

การกำหนดให้มอดูลรับพารามิเตอร์

กรณีข้อกำหนดมีการรับพารามิเตอร์เข้าก่อนเริ่มการทำงาน เราสามารถระบุชื่อตัวแปรเสริมที่เป็นพารามิเตอร์เพิ่มในวงเล็บต่อท้ายชื่อ *MACHINE* ดังนี้

```
MACHINE
          booking(max_seat)
     CONSTRAINTS
          max_seat ∈ NAT
     VARIABLES
          seat
     INVARIANT
          seat ∈ 0..max_seat
     INITIALIZATION
          seat := max_seat
     OPERATIONS
          book ≙
            PRE
                 0<seat
            THEN
                 seat := seat-1
            END;
          cancel ≙
            PRE
                 seat<max_seat
            THEN
                      seat := seat+1
            END
     END
```

กล่าวคือ *booking(max_seat)* จะเป็นชื่อของ *MACHINE* ที่มีตัวแปรเสริมชื่อ *max_seat* รับเข้าไปก่อนทำงาน และตัวแปรเสริมนี้เองจะเป็นจำนวนที่นั่งสูงสุดที่ระบบนี้จะให้จองได้ ตัวแปรเสริมที่ส่งมาเป็นพารามิเตอร์นี้แล้วจะต้องได้รับการระบุชนิดของตัวแปรด้วย โดยให้ระบุประโยคประกาศภาคแสดงในอนุประโยคชื่อ *CONSTRAINTS* คือ $max_seat \in NAT$ โดยที่ *NAT* คือแบบชนิดที่เป็นจำนวนธรรมชาติ

ในขณะเดียวกัน เราสามารถกำหนดค่าเริ่มต้นของตัวแปรสถานะ *seat* ก่อนเริ่มการทำงานจริงได้ให้มีค่าเท่าดับจำนวนสูงสุดที่จองที่นั่งได้ โดยใช้อนุประโยคชื่อ *INITIALIZATION* และมีประโยคประกาศไว้คือ *seat := max_seat*

การดำเนินการที่มีพารามิเตอร์เป็นข้อมูล

ในการเขียนข้อกำหนดเพื่อให้มีความสมบูรณ์มากขึ้น เราสามารถเขียนข้อกำหนดเพื่ออธิบายการดำเนินการให้มีพารามิเตอร์เป็นข้อมูลนำเข้าได้เช่นกัน โดยการเพิ่มรายการตัวแปรนำเข้า เป็นพารามิเตอร์ในเครื่องหมายวงเล็บต่อท้ายชื่อการดำเนินการที่ต้องการ เช่น การดำเนินการ *book(nbr)* จะมีพารามิเตอร์ชื่อ *nbr* ต่อท้ายในวงเล็บ เป็นต้น

```
book(nbr) ≙
  PRE
      nbr ∈ NAT ∧
      nbr ⩽ seat
  THEN
      seat := seat-nbr
  END;

cancel(nbr) ≙
  PRE
      nbr ∈ NAT ∧
      seat + nbr ⩽ max_seat
  THEN
      seat := seat+nbr
  END
```

หลังจากที่การดำเนินการชื่อ *book(nbr)* มีการรับตัวแปรนำเข้าได้แล้ว เราจะต้องเขียนประโยคระบุแบบชนิดของตัวแปรชื่อ *nbr* ด้วย คือ

$nbr \in NAT$ และตัวแปร nbr จะสามารถถูกนำมาใช้ต่อไปในการทดสอบเงื่อนไขก่อนได้คือ $nbr \leqslant seat$

ในขณะเดียวกันการดำเนินการชื่อ $cancel(nbr)$ ก็มีการรับตัวแปรนำเข้าได้แล้ว เราต้องเขียนประโยคระบุแบบชนิดของตัวแปร nbr ด้วยเช่นกัน คือ

$nbr \in NAT$ และตัวแปร nbr จะสามารถนำมาใช้ในการทดสอบเงื่อนไขก่อนของการดำเนินการ $cancel$ ได้คือ $seat + nbr \leqslant max_seat$

กรณีให้จองที่นั่งไม่เกิน 5 ที่นั่งต่อครั้ง สามารถทำได้โดยการเพิ่มประโยคภาคแสดงในส่วนเงื่อนไขก่อนคือ $nbr \leqslant 5$ ได้ดังตัวอย่าง

```
book(nbr) ≙
  PRE
      nbr ∈ NAT ∧
      nbr ⩽ seat ∧
      nbr ⩽ 5
  THEN
      seat := seat-nbr
  END
```

การดำเนินการที่มีการระบุพารามิเตอร์ขาออก

ในกรณีการดำเนินการที่ต้องการส่งข้อมูลเป็นพารามิเตอร์ขาออก เราสามารถระบุตัวแปรที่ต้องการส่งข้อมูลออกได้โดยใช้สัญกรณ์ ⟵ ดังตัวอย่างต่อไปนี้

```
value ⟵ val_seat ≙ BEGIN  value := seat END
```

จากตัวอย่างเป็นการดำเนินการชื่อ val_seat ซึ่งมีการส่งข้อมูลออกเป็นตัวแปรชื่อ value และในข้อกำหนดจะระบุค่าที่ส่งออกโดยใช้ประโยค value := seat ค่าที่ส่งออกเป็นจำนวนที่นั่งที่เหลืออยู่ในปัจจุบัน

การแทนค่าแบบพหุคูณ

กรณีการทำการแทนค่า (Substitution) คือการแทนค่าพจน์ด้านซ้ายในสมการด้วยค่าด้ายขวาของสมการ คือ

$seat := seat -1$

แสดงให้เราทราบว่าในข้อกำหนดระบุไว้ให้ตัวแปร $seat$ สามารถแทนได้ด้วย $seat\text{-}1$ ด้วยหลังจากผ่านการกำหนดประโยคนี้แล้ว

กรณีการทำการแทนค่าเกิดขึ้นหลายๆกรณี เราสามารถเขียนประโยคการแทนค่าแบบพหุคูณได้ คือ

report, *seat* := *good*, *seat-nbr*

ซึ่งจะมีความหมายเทียบเท่ากับการแทนค่าต่อไปนี้ คือ

report := *good*

seat := *seat-nbr*

การแทนค่าอย่างมีเงื่อนไข

บางครั้งเราต้องการให้เกิดการแทนค่าก็ต่อเมื่อเงื่อนไขหนึ่งๆ เป็นจริงก่อน เทียบได้กับการทำ *IF…THEN…ELSE…END* ดังตัวอย่างต่อไปนี้คือ

```
IF nbr ⩽ seat THEN
        report, seat := good, seat-nbr
ELSE
        report := bad
END
```

จากตัวอย่างข้างต้นจะเห็นได้ว่าการแทนค่าต่อไปนี้

report, seat := *good, seat-nbr*

จะเกิดขึ้นก็ต่อเมื่อเงื่อนไข $nbr \leqslant seat$ มีค่าความจริงเป็นจริงเท่านั้น มิฉะนั้นแล้วจะมีการแทนค่าด้วยค่าต่อไปนี้แทน

report := *bad*

การแทนค่าโดยการเลือกแบบมีขอบเขต (Bounded Choice Substitution)

บางครั้งในการดำเนินการอาจจำเป็นต้องมีการเลือกทำหรือไม่ทำอะไร เราสามารถใช้การเขียนการแทนค่าโดยการเลือกแบบมีขอบเขตได้ โดยใช้คำสั่ง *CHOICE… OR… END*

การเลือกในลักษณะจะเป็นการเลือกแบบไม่สามารถคาดการณ์ได้อย่างมีขอบเขต (Bounded Non-determinism) กล่าวคือ ในแต่ละครั้งที่เข้าสู่การตัดสินใจว่าจะเลือกทำอย่างใดอย่างหนึ่งนั้น จะไม่มีเกณฑ์แน่นอน ระบบจะไม่สามารถคาดเดาได้ว่าจะเลือกทางใด เช่น

CHOICE S OR T OR U OR V END

โดยที่ *S, T, U, V* เป็น ประโยคภาคแสดง (Predicate) ใดๆ

ระบบจะเลือกทำ *S* หรือ *T* หรือ *U* หรือ *V* ก็ได้อย่างไม่แน่นอนเสมอ

การแทนค่าโดยการเลือกแบบไม่มีขอบเขต

การแทนค่าโดยการเลือกแบบไม่มีขอบเขต ได้รับการออกแบบเพื่อให้การแทนค่า *S* ใดๆที่เกิดขึ้นขึ้นอยู่กับตัวแปรแบบท้องถิ่นใดๆ ชื่อ *z* ได้โดยสำหรับตัว

แปร z ตัวใดๆที่ทำให้ P เป็นจริงแล้วระบบจะทำการเลือกการแทนค่า S เสมอ โดยแสดงเป็นรูปแบบการใช้งานดังต่อไปนี้

```
Any z WHERE
  P
THEN
  S
END
```

โดยการทำ S จะขึ้นอยู่ตัวแปร z โดยที่ P เป็นจริงเสมอ กล่าวคือ สำหรับค่า z ใดๆ ที่ทำให้ P เป็นจริงได้แล้วระบบจะทำ S ให้

ตัวอย่างคือ

```
ANY angel WHERE
        angel ∈ PERSON – person
THEN
        person := person ∪ {angel} ||
        sex(angel) := sx ||
        status(angel) := living ||
        baby := angel
END
```

โดยหมายถึงค่าของตัวแปร *angel* ใดๆที่เป็นสมาชิกของเซต *PERSON – person* ถ้าเป็นจริงแล้ว จะให้ทำการแทนค่าดังต่อไปนี้เสมอ

```
person := person ∪ {angel} ||
sex(angel) := sx ||
status(angel) := living ||
baby := angel
```

4.12 ภาษาคาเฟโอบีเจ

ภาษาคาเฟโอบีเจ [23] เป็นภาษาข้อกำหนดเชิงรูปนัยเชิงพีชคณิตที่ได้รับออกแบบสืบทอดคุณสมบัติและความสามารถมาจากภาษาโอบีเจ เช่น ลักษณะโครงสร้างแบบมอดูล การกำหนดแบบชนิด (type) และแบบชนิดย่อย (sub type) การทำคำสั่งการนำเข้า หรือลักษณะมอดูลที่รับพารามิเตอร์ (parameterized module) เป็นต้น

ทั้งนี้ ภาษาคาเฟโอบีเจได้เพิ่มชุดหน่วยภาษา (paradigm) ใหม่เข้าไปด้วย เช่น ตรรกศาสตร์แบบ *rewriting logic และ hidden algebra* เป็นต้น

การเขียนข้อกำหนดด้วยภาษาคาเฟโอบีเจมีลักษณะที่เขียนคล้ายกับการเขียนด้วยภาษาโปรแกรมแบบฟังก์ชัน (functional programming language) โดยมีการใช้สมการพีชคณิตเพื่ออธิบายคุณสมบัติทางพลวัตของระบบซอฟต์แวร์

โครงสร้างข้อหนดที่เขียนด้วยคาเฟโอบีเจ

การเขียนข้อกำหนดด้วยภาษาคาเฟโอบีเจที่จะกล่าวต่อไป เป็นส่วนที่เรียกว่าการกำหนดแบบพื้นฐาน (basic specification) ซึ่งจะเขียนเป็นส่วน ๆ เรียกว่า “มอดูล” โดยแต่ละมอดูลสามารถแสดงถึง วัตถุเชิงคณิตศาสตร์ (mathematical object) ที่เราสนใจ

การเขียนประกาศมอดูล

มอดูลในภาษาคาเฟโอบีเจเป็นเหมือนตัวต่อ (building block) ที่สามารถนำมาประกอบกันเพื่อใช้งานร่วมกันได้ มอดูลในคาเฟโอบีเจแบ่งเป็น 2 ประเภท คือ

1. มอดูลแบบตึง (tight module)
2. มอดูลแบบหย่อน (loose module)

เราใช้มักใช้มอดูลแบบตึงเพื่อแสดงถึงสิ่งหนึ่งสิ่งใดที่มีอยู่หนึ่งเดียวในระบบ (unique object/model) ในขณะที่มอดูลแบบหย่อนใช้แทนกลุ่มของสิ่งที่มีอยู่ (class of objects/model) เช่น สิ่งที่เป็น ค่าคงที่จะแสดงได้ด้วยมอดูลแบบตึงได้ ส่วนสิ่งที่เป็นสิ่งของทั่วไปคือ หนังสือหรือนาฬิกา จะแสดงได้ด้วยมอดูลแบบหย่อนได้ เป็นต้น

มอดูลใด ๆ จะประกอบด้วยส่วนที่เรียกว่าลายเซ็น (signature part) และส่วนที่เรียกว่าสัจพจน์ (axiom part)

ส่วนลายเซ็นประกอบด้วยเซตของ *Sort* และการดำเนินการ (Operation) และประโยคภาคแสดง (Predicate) ที่ระบุความสัมพันธ์ของเซตของ Sort ที่ประกาศไว้

ส่วนสัจพจน์ประกอบด้วยกลุ่มของตัวแปร และประโยคสมการ (equation) ที่เขียนไว้ในลักษณะสมการพีชคณิต (algebraic equation) โดยสมการที่มีเป็นสมการที่ระบุความสัมพันธ์ระหว่างการดำเนินการต่าง ๆที่มีอยู่จนครบ

การเขียนมอดูลจะเขียนด้วยส่วนหลัก ๆที่สำคัญ [23] ต่อไปนี้

1. การประกาศมอดูลและชื่อมอดูล
2. วงเล็บปีกกา {...} เพื่อกำหนดขอบเขตของเนื้อหาของมอดูล
3. การประกาศการนำเข้า
4. การประกาศ *Sort*

5. การประกาศการดำเนินการ
6. การประกาศตัวแปร
7. การประกาศประโยคสมการ

กล่าวคือใช้คำว่า *module* ตามด้วยชื่อมอดูล ตามด้วยเนื้อหามอดูลภายในเครื่องหมายวงเล็บปีกกา {..} โดยเนื้อหาจะประกอบด้วยการระบุแบบชนิด ในคาเฟโอบีเจเราเรียกแบบชนิดว่าเป็น *Sort* แทน *Type* จากนั้นให้ประกาศการดำเนินการและประกาศประโยคสมการ

ตัวอย่างคือ

```
module! BARE-NAT {
        [ NzNAT Zero < Nat ]
        op 0 : -> Zero
        op s_ : Nat -> NzNat
}
```

โดยในตัวอย่างข้างต้นเป็นมอดูลแบบตึงชื่อว่า *BARE-NAT* มีการใช้ Sort อยู่ 3 แบบชนิดคือ *NzNat* *Zero* และ *Nat* โดยหมายความว่า

Nat คือ *Natural Number*

Zero คือ *Zero*

NzNat คือ *Non Zero Natural Number* นั่นเอง

โดยที่ *NzNat* และ *Zero* เป็นแบบชนิดย่อย (sub sort) ของ *Nat*

การประกาศแบบชนิดเราจะเขียนภายในวงเล็บ [..] เสมอ ส่วนที่ประกาศลายเซ็น ก็คือส่วนถัดไปคือ

```
op 0 : -> Zero
op s_ : Nat -> NzNat
```

โดย *op* หมายถึงการดำเนินการ ดังนั้นในมอดูลจะมีการดำเนินการชื่อ *0* และชื่อ *s_* แต่ไม่มีส่วนประกาศสัจพจน์แสดงไว้

มอดูลแบบตึง

มอดูลแบบตึงเป็นการเขียนข้อกำหนดอธิบายสิ่งที่มีอยู่หนึ่งเดียวในระบบ จะไม่มีซ้ำอีก ถ้าต้องการอ้างถึงให้อ้างมาที่มอดูลที่เดียวกัน คาเฟโอบีเจได้กำหนดสัญกรณ์ในการกำหนดมอดูลแบบตึงไว้ดังนี้

module! <ชื่อมอดูล> { … }

หรือเขียนย่อได้ว่า

mod! <ชื่อมอดูล> {…}

ตัวอย่างคือ

module! ON-OFF {…}

module! BARE-NAT {…}

มอดูลแบบหย่อน

มอดูลแบบหย่อนเป็นการเขียนข้อกำหนดอธิบายสิ่งที่มีอยู่ได้มากกว่าหนึ่งหน่วยขึ้นไป เช่น หนังสือหลายเล่ม ปากกาหลายด้าม สวิตช์หลายตัว เป็นต้น การเขียนบรรยายหนังสือ ปากกา หรือสวิตช์ดังกล่าวจะเขียนด้วยมอดูลที่มีรูปแบบดังนี้

*module** <ชื่อมอดูล> {…}

หรือ

*mod** <ชื่อมอดูล> {…}

ตัวอย่างมีดังนี้

module* Switch {…}

mod* Book {…}

mod* Pen {…}

การประกาศการนำเข้า (Import)

การเขียนข้อกำหนดคาเฟโอบีเจสามารถนำเข้ามอดูลเดิมที่มีอยู่แล้วมาอ้างถึงได้ โดยการนำเข้าทำให้มอดูลใหม่มี *Sort* ลายเซ็น ตัวแปร และส่วนประกาศอื่นๆ ที่มีอยู่ในมอดูลที่นำเข้ามาอ้างใช้งานได้ทันทีโดยไม่ต้องเขียนใหม่ ทั้งนี้โดยต้องระมัดระวังการตั้งชื่อไม่ให้ซ้ำกันจะเป็นเรื่องที่ดี

การนำเข้าทำได้ 3 แบบคือ

1. การนำเข้าแบบ *Protecting*
2. การนำเข้าแบบ *Extending*
3. การนำเข้าแบบ *Using*

การนำเข้าแบบ *Protecting* เป็นการนำเข้ามอดูลเพื่ออ้างอิงโดยไม่สามารถแก้ไขเนื้อหาในมอดูลต้นฉบับ เราถือได้ว่าเป็นการอ้างอิงแบบอ่านอย่างเดียว (read only)

การนำเข้าแบบ *Extending* เป็นการนำเข้ามอดูลเพื่ออ้างอิงและสามารถเพิ่มรายละเอียดในเนื้อหามอดูลต้นฉบับ

การนำเข้าแบบ *Using* เป็นการนำเข้ามอดูลเพื่ออ้างอิงและสามารถเพิ่มรายละเอียดในเนื้อหามอดูลต้นฉบับ

ตัวอย่างคือ

```
module! BARE-NAT {
  [ Nat ]
  op 0 : -> Nat
  op s : Nat -> Nat
}
module! BARE-NAT-AGAIN {
  protecting (BARE-NAT)
}
```

จากตัวอย่างว่า มีการเขียนข้อกำหนดไว้แล้วในมอดูลชื่อ *BARE-NAT* ซึ่งมีการประกาศ *Sort* และการดำเนินการไว้ก่อน มอดูลใหม่ *BARE-NAT-AGAIN* เป็นมอดูลที่นำเข้า *BARE-NAT* ที่มีอยู่มาแบบ Protecting โดยจะมีการอ้างถึงประโยคที่ประกาศไว้แล้วได้โดยไม่ต้องเขียนใหม่

ผลลัพธ์ของมอดูล *BARE-NAT-AGAIN* จะมีค่าเท่ากับข้อกำหนดต่อไปนี้

```
module! BARE-NAT-AGAIN {
  [ Nat ]
   op 0 : -> Nat
   op s : Nat -> Nat
}
```

การประกาศ Sort

Sort คือ ชื่อที่ใช้อ้างถึงสิ่งที่เหมือนกัน สิ่งที่แบบชนิดเดียวกัน ในทางปฏิบัติความหมาย *Sort* จะเหมือนกับ *Type* ในระบบซอฟต์แวร์นั่นเอง เพียงแต่เรียกให้แตกต่างกันในที่นี้เท่านั้น

ในคาเฟโอบีเจมีการแบ่ง *Sort* ออกเป็นสองประเภทคือ

1. *Sort* แบบสามัญ (Ordinary Sort)
2. *Sort* แบบซ่อน (Hidden Sort)

"Sort แบบสามัญ" หรือเรียกอีกชื่อว่า *"Sort แบบเห็นได้"* (visible sort) เราเขียนชื่อของ *Sort* ไว้ในเครื่องหมายวงเล็บ […] ทั้งนี้ชื่อที่ระบุในวงเล็บก็คือแบบชนิดของตัวแปรที่เป็นไปได้ในระบบนั่นเอง เช่น [*Nat*] เป็นต้น

"Sort แบบซ่อน" คือ *Sort* ที่ใช้อ้างถึงสถานะของสิ่งที่เราสนใจ (object state) หรือสถานะของเครื่องเชิงนามธรรมที่เราสนใจ เรามักจะเขียนประกาศ

Sort แบบซ่อนไว้เพื่ออธิบายข้อกำหนดเชิงพฤติกรรมโดยใช้เครื่องหมายวงเล็บพิเศษคือ *[…]* เช่น *[*Book*]* เป็นต้น

Sort ชื่อใดๆ ที่ประกาศไว้เป็น *Sort* แบบสามัญ ไม่สามารถนำชื่อเดียวกันนี้ไปใช้เป็นชื่อ Sort อื่นๆได้อีก กล่าวคือ ทุกชื่อของ *Sort* จะไม่มีส่วนร่วมกัน (Disjoint) เสมอ

ตัวอย่าง

[S1 S2]

[S3 S4]

จากการประกาศข้างต้นจะมี *Sort* ที่มีชื่อ *S1 S2 S3* และ S4 และทั้ง 4 *Sort* จะไม่มีส่วนร่วมกันเลย (disjoint sorts) เสมอ

เราสามารถกำหนด *Sort* ย่อย (subsort) ได้โดยใช้เครื่องหมาย < หรือ ⩽ ได้ดังต่อไปนี้

ตัวอย่างคือ

[S1 < S2 ⩽ S3]

[Zero NzNat < Nat]

ซึ่งจะหมายความว่า *S1* เป็น *Sort* ย่อยที่สุดโดยที่มี *S2* เป็นเซตที่ใหญ่กว่า *S1* กล่าวคือ *S2* จะครอบคลุม *S1* และไม่มีค่าเท่ากับ *S1*

ในขณะที่ *S2* เป็น *Sort* ย่อยกว่าหรือเท่ากับ *S3* กล่าวคือ *S3* จะครอบคลุม *S2* หรือมีค่าเท่ากับ *S2* ได้

อีกตัวอย่างหนึ่งแสดงให้เห็นว่า *Nat* เป็น *Sort* ที่ครอบคลุม *Zero* และ *NzNat* เสมอ เรามักใช้ชื่อย่อ *Nat* แทน *Natural Number* ชื่อย่อ *Zero* แทน *Zero* และ *NzNat* แทน *Non Zero Natural Number* ดังนั้นความหมายคือ จำนวนธรรมชาติ จะประกอบด้วย จำนวนศูนย์และจำนวนธรรมชาติที่ไม่เป็นศูนย์

Sort ย่อย

การที่มี *Sort* ย่อยทำให้มีลักษณะ *Partial Order* ในระบบ *Sort* ได้โดยนิยามของ *Partial Order* คือความสัมพันธ์ทวิภาค (binary relation) *R* ที่กระทำบนเซต *P* ซึ่งการทำความสัมพันธ์สะท้อน (reflexive relation) การทำความสัมพันธ์ปฏิสมมาตร (antisymmetric relation) และการทำความสัมพันธ์ถ่ายทอด (transitive relation) เป็นจริงได้สำหรับทุกๆ a b และ c ที่อยู่ในเซต P ดังนี้

1. aRa (reflexivity) เป็นจริง
2. ถ้า aRb และ bRa แล้ว $a = b$ (antisymmetry) เป็นจริง
3. ถ้า aRb และ bRc แล้ว aRc (transitivity) เป็นจริง

สำหรับ *Sub* ย่อยใด ๆ ที่ประกาศไว้เราสามารถกล่าวได้ว่าเป็น *Partial Order* ได้ดังนี้

1. $s \leqslant s$ (reflexivity) เป็นจริง
2. ถ้า $s1 \leqslant s2$ และ $s2 \leqslant s1$ แล้ว $s1=s2$ (antisymmetry) เป็นจริง
3. ถ้า $s1 \leqslant s2$ และ $s2 \leqslant s3$ แล้ว $s1 \leqslant s3$ (transitivity) เป็นจริง

การประกาศการดำเนินการ (Operation)

ผู้เขียนข้อกำหนดคาเฟโอบีเจจะต้องกำหนดการดำเนินการใด ๆ ที่มีอยู่ในระบบไว้ก่อนเขียนประโยคสมการเสมอ โดยหลังจากที่มีการประกาศ *Sort* ที่ใช้ได้แล้ว การดำเนินการที่กำหนดจะอ้างถึง Sort ที่มีอยู่เท่านั้น

รูปแบบการเขียนการดำเนินการมีดังนี้

op <ชื่อการดำเนินการ> : <*Si*> -> <*So*>

โดยที่

op คือคำสงวน (reserved word) ที่ใช้นำหน้าชื่อการดำเนินการ

Si คือรายการ *Sort* ที่ระบุแบบชนิดของตัวแปรนำเข้าของการดำเนินการ (เรียกว่า arity)

So คือรายการ *Sort* ที่ระบุแบบชนิดของตัวแปรผลลัพธ์ของกาดำเนินการ (เรียกว่า co-arity)

ตัวอย่างคือ

op add : Nat Nat -> Nat

หมายถึง *add* เป็นชื่อการดำเนินการ และการดำเนินการนี้จะมีตัวแปรนำเข้าการดำเนินการนี้ *2* ตัวแปรที่มีแบบชนิดเป็น *Nat* ทั้งคู่ และมีตัวแปรผลลัพธ์เป็น *Nat*

การทำ Overload ของการดำเนินการ

มอดูลใด ๆ ที่เขียนขึ้นมาโดยภาษาคาเฟโอบีเจมีการระบุการดำเนินการ และต้องตั้งชื่อไม่ซ้ำกัน โดยชื่อการดำเนินการมักจะสื่อความหมายถึงการทำงานของระบบด้วย ทำให้บางครั้งผู้เขียนต้องการตั้งชื่อการดำเนินการซ้ำกัน โดยการส่งตัวแปรที่แตกต่างกัน เราเรียกการกำหนดการดำเนินการที่มีชื่อซ้ำกันนี้ว่า การทำ *Overload* ของการดำเนินการ ตัวแปรที่แตกต่างกันนี้อาจจะแตกต่างกันที่จำนวนตัวแปรหรือแบบชนิดที่เรียงลำดับที่แตกต่างกันได้

ตัวอย่างการทำ *Overload* ของการดำเนินการคือ

```
op add : Nat Nat -> Nat
op add : Int Int -> Int
op add : Nat Int -> Nat
op add : Int Nat -> Nat
```

จากตัวอย่างข้างต้นจะเห็นว่ามีการดำเนินการชื่อ *add* ที่มีความแตกต่างกันรวม 4 การดำเนินการ กล่าวคือ ถ้าเริ่มต้นจาก *add* บรรทัดแรกเป็นหลัก จะมีการทำ *Overload* ครั้งแรกโดยได้การดำเนินการชื่อ *add* แบบเดิมแต่มี *Arity* และ *Co-arity* ต่างกัน ต่อมามีการทำ *Overload* ครั้งที่สองโดยได้การดำเนินการชื่อ *add* เหมือนกันแต่มี *Arity* และ *Co-arity* ต่างกัน และครั้งที่สามได้การดำเนินการชื่อ *add* โดยการสลับลำดับของ *Arity* จาก *add* ก่อนหน้าเท่านั้น

การกำหนดค่าคงที่

ผู้เขียนสามารถกำหนดค่าคงที่โดยให้ระบุการดำเนินการพิเศษโดยระบุให้ไม่มี *Arity* ดังตัวอย่างต่อไปนี้คือ

```
[Value]
op on : -> Value
op off : -> Value
```

จากตัวอย่างมีการประกาศ *Sort* แบบเห็นได้ที่มีชื่อว่า *Value* และมีการดำเนินการพิเศษชื่อ *on* และ *off* โดยทั้งสองการดำเนินการไม่มี *Arity* แต่ให้ผลลัพธ์ออกมาเป็นแบบชนิด *Value* ใด ๆ เสมอ เราสามารถใช้การดำเนินการพิเศษนี้ว่า ค่าคงที่

การใช้งานค่าคงที่ทำได้โดยการเรียกการดำเนินการพิเศษนี้ดังต่อไปนี้ ต่อไปนี้เป็นส่วนหนึ่งของข้อกำหนดของอุปกรณ์ *Switch*

```
mod! ON-OFF {
[Value]
op on : -> Value
op off : -> Value
}
mod* SWITCH {
protecting(ON-OFF)

*[ Switch ]*
op init-sw : -> Switch
op status : Switch -> Value
eq status(init-sw) = off.
}
```

จากตัวอย่างเห็นได้ว่ามีมอดูลชื่อ *ON-OFF* เป็นมอดูลแบบตึง ซึ่งแสดงถึงค่าคงที่สองค่า คือ *on* และ *off* การกำหนดค่าคงที่แบบนี้ใช้การดำเนินการพิเศษที่ไม่มี *Arity* นั่นเอง

มอดูลชื่อ *SWITCH* เป็นมอดูลใหม่ที่มีการนำเข้ามอดูลชื่อ *ON-OFF* เข้ามาเพื่ออ้างอิงถึงค่าคงที่ *on* และ *off* ที่มีอยู่แล้ว โดยการเขียนประโยคสมการดังนี้

```
eq status(init-sw) = off.
```

หมายถึงการดำเนินการชื่อ *status* ที่มีตัวแปรนำเข้าเป็น *Switch* ที่ได้จากผลลัพธ์ของการดำเนินการชื่อ *init-sw* มาเป็นค่านำเข้าซึ่งจะมีค่าเท่ากับ *off* เสมอ

การกำหนดการดำเนินการเชิงพฤติกรรม

การดำเนินการบางอย่างเป็นการดำเนินการเชิงพฤติกรรม โดยการดำเนินการที่มี *Sort* แบบซ่อนหนึ่งเดียวเท่านั้นอยู่ใน *Arity* และมีการใช้คำสงวนนำหน้าการดำเนินการเป็น *bop* แทน *op* ดังตัวอย่างส่วนข้อกำหนดต่อไปนี้

```
module* SWITCH {
protecting(ON-OFF)
*[Switch]*
op init-sw : -> Switch
bop on_ : Switch -> Switch
bop off_ : Switch -> Switch
…
}
```

จะเห็นได้ว่าการดำเนินการที่กำหนดไว้มี *init-sw* *on_* และ *off_* โดยการดำเนินการ *on_* และ *off_* เป็นการดำเนินการเชิงพฤติกรรมที่ต่างจากการดำเนินการปกติชื่อ *init-sw* ดังนั้นเราจะใช้คำสงวนว่า *bop* นำหน้าเสมอและเพราะมี *Switch* ซึ่งเป็น *Sort* แบบซ่อนหนึ่งเดียวอยู่เป็น *Arity*

สำหรับการดำเนินการชื่อ *init-sw* จะเห็นได้ว่าไม่มีการนำตัวแปรเข้าเลย แต่ให้ค่าออกมาเป็น *Switch* เราเรียกการดำเนินการนี้ว่าเป็นการดำเนินการแบบตัวสร้าง (constructor operation)

ถ้าเราใช้คาเฟโอบีเจในการเขียนข้อกำหนดระบบด้วยแนวคิดเชิงวัตถุ เราสามารถเขียนข้อกำหนดของการดำเนินการเชิงพฤติกรรมที่เรียกว่า *Attribute* ได้ถ้ามี *Co-arity* เป็น *Sort* แบบเห็นได้เสมอ และการดำเนินการเชิงพฤติกรรมที่เรียกว่า *Method* ได้ถ้ามี *Co-arity* เป็น *Sort* แบบซ่อน

การประกาศตัวแปร

เมื่อเริ่มการประกาศส่วนสัจพจน์ซึ่งประกอบด้วยการกำหนดตัวแปรที่ใช้ในสมการ และประกาศประโยคสมการ

ผู้เขียนข้อกำหนดประกาศตัวแปรได้ดังนี้

```
var  S : Switch
```

โดยกำหนดชื่อตัวแปร ซึ่งนำหน้าด้วยคำสงวนว่า *var* และชื่อตัวแปรจะต่อท้ายด้วยเครื่องหมาย : และตามด้วยแบบชนิดที่ระบุในส่วนประกาศ *Sort* ใด ๆ ในที่นี้กำหนดตัวแปรชื่อ *S* ที่มี *Sort* เป็น *Switch*

การประกาศสมการ

การประกาศสมการเป็นการแจกแจงสัจพจน์ โดยเขียนเป็นสมการพีชคณิตที่ระบุความสัมพันธ์ระหว่างการดำเนินการที่ประกาศไว้ด้วย การเขียนสมการเขียนได้ตามรูปแบบดังนี้

```
t = t' if C
```

โดย *t = t'* เป็นส่วนสมการที่ประกาศแบบพีชคณิต และเขียนในรูปแบบ *functional programming* และ *C* เป็นเงื่อนไขที่ต้องทดสอบว่าให้เป็นจริงก่อนทำ *t = t'* เราเรียกรูปแบบนี้ว่าสมการมีเงื่อนไข (conditional equation)

แต่ถ้าเงื่อนไขไม่ได้กำหนดไว้หรือละไว้ จะหมายถึงว่าไม่มีเงื่อนไขหรือเงื่อนไขเป็นจริงตลอดเวลา เราจึงสามารถทำ *t = t'* ได้โดยเรียกว่าสมการไม่มีเงื่อนไข (unconditional equation)

ในคาเฟโอบีเจเราใช้คำสงวนว่า *eq* สำหรับนำหน้าสมการไม่มีเงื่อนไข และใช้คำว่า *ceq* นำหน้าสมการมีเงื่อนไข เสมอ

ตัวอย่างคือ

```
*[ Switch ]*
…
var S : Switch
var A : Nat
eq state(init-sw) = off .
ceq state(S) = on if A==1.
```

จากตัวอย่างข้างต้นได้มีการประกาศตัวแปรชื่อ *S* ซึ่งมีแบบชนิดเป็น *Sort* แบบซ่อนชื่อว่า *Switch* ตัวแปรชื่อ *S* เมื่อถูกอ้างที่ใดในสมการจะมีความหมายว่า *"Switch S ใด ๆ ตัวหนึ่ง"*

สมการแรกเป็นสมการแบบไม่มีเงื่อนไขที่ต้องเป็นไปตามนี้เสมอคือ ถ้ามีการป้อน *switch* ที่เกิดจากการดำเนินการชื่อ *init-sw* เข้าไปในการดำเนินการชื่อ *state* แล้ว ค่าผลลัพธ์ที่ได้ออกมาจะมีค่าเท่ากับค่าคงที่ชื่อ *off* เสมอ กล่าวอย่างง่ายได้ว่า *switch* ที่เกิดใหม่จากการดำเนินการ *init-sw* จะเริ่มโดยมีสถานะเป็นค่าคงที่ *off* เสมอ

สมการที่สองเป็นสมการแบบมีเงื่อนไขซึ่งจะเกิดกรณีตามสมการที่ว่า *state(S) = on* ได้ก็ต่อเมื่อเงื่อนไข *A == 1* เป็นจริงเท่านั้น เงื่อนไข *A==1* หมายถึงค่าตัวแปร *A* ที่มีแบบชนิดเป็นจำนวนธรรมชาติจะมีค่าเท่ากับ *1* เครื่องหมาย == ใช้แทนการเปรียบเทียบทางตรรกะ

สมการที่ว่า *state(S) = on* หมายถึง ถ้ามี *switch S* ใด ๆแล้วสถานะของ *switch S* จะเท่ากับค่าคงที่ *on* เสมอ

กรณีศึกษาเรื่อง Switch

ข้อกำหนดตัวอย่างในกรณีศึกษาต่อไปนี้จะแสดงการกำหนดสิ่งที่สนใจคือ *Switch* ซึ่งตามที่กล่าวมาแล้วว่า เรามักจะเขียนมอดูลสำหรับอธิบายสิ่งที่เราสนใจเสมอ ดังนั้นจะมีมอดูลชื่อ *SWITCH* เกิดขึ้นในระบบ

เนื่องจากในระบบจะมีการอ้างถึงค่าคงที่ของ *SWITCH* ที่มีค่าได้เป็นค่าคงที่ *on* และ *off* เท่านั้น เราจะมีการเขียนมอดูลสำหรับค่าคงที่ทั้งสองนี้ก่อนเพื่อให้อ้างอิงต่อไป ดังนั้นต่อไปนี้จะเป็นการเขียนข้อกำหนดมอดูลชื่อ *ON-OFF* และมอดูลชื่อ *SWITCH*

ข้อกำหนดคาเฟโอบีเจสำหรับ SWITCH

```
-- -----------------------------------------------------------
-- ON-OFF
-- -----------------------------------------------------------
mod! ON-OFF {
  [ Value ]

  ops on off : -> Value
}

-- -----------------------------------------------------------
-- SWITCH
-- -----------------------------------------------------------
```

```
mod* SWITCH {
  protecting(ON-OFF)

  *[ Switch ]*

  op init : -> Switch
  bop on_ : Switch -> Switch      -- method
  bop off_ : Switch -> Switch    -- method
  bop state_ : Switch -> Value   -- attribute

  var S : Switch

  eq state init = off .
  eq state(on S) = on .
  eq state(off S) = off .
}
```

คำอธิบายข้อกำหนดคาเฟโอบีเจสำหรับ SWITCH

จากข้อกำหนดที่เขียนขึ้นเราเห็นว่า มีการเขียนหมายเหตุ (comment) เพื่อเป็นข้อความเตือนความจำและไม่มีผลกับข้อกำหนดหลักโดยใช้เครื่องหมาย "- -" และข้อความหมายเหตุจะปรากฏอยู่หลังเครื่องหมาย

มอดูลแบบตึงชื่อ *ON-OFF* โดยใช้ประโยคว่า

mod! ON-OFF

และในส่วนการประกาศลายเซ็น (Signature Part) โดยจะมีการประกาศแบบชนิดโดยมีการตั้งชื่อ Sort และประกาศการดำเนินการพร้อม *Arity* และ *Co-arity* ของการดำเนินการดังนี้

[Value]

เป็นการประกาศ *Sort* แบบเห็นได้ชื่อ *Value* โดยใช้เครื่องหมาย [ ]

ops on off : -> Value

เป็นการประกาศการดำเนินการชื่อ *on* และ *off* โดยใช้เครื่องหมาย *ops* และมีเครื่องหมาย : ตามหลังชื่อการดำเนินการ เนื่องจาก *on* และ *off* เป็นการดำเนินการที่ไม่มี *Arity* แต่มี *Co-arity* นั่นก็คือ ค่าคงที่นั่นเอง กล่าวคือมีค่าคงที่ชื่อ *on* และ *off* โดยมีค่าผลลัพธ์ที่มีแบบชนิด *Value*

เนื้อหาต่างๆในมอดูลจะอยู่ภายในวงเล็บ {...} เสมอ

มอดูลที่สองเป็นมอดูลแบบหย่อนชื่อ *SWITCH* โดยใช้ประโยคว่า

mod* SWITCH

มอดูลชื่อ *SWITCH* นี้ทำการนำเข้ามอดูลเดิมชื่อ *ON-OFF* โดยการนำเข้าจะทำให้มอดูล *SWITCH* สามารถอ้างถึง *Sort* และการดำเนินการพิเศษที่เป็นค่าคงที่ *on* และ *off* มาใช้ การนำเข้าทำได้ด้วยประโยคดังนี้

protecting(ON-OFF)

การนำเข้าแบบ *Protecting* หมายถึงการนำเข้าโดยไม่สามารถแก้ไขเนื้อหาต้นฉบับได้เลย เป็นการอ้างแบบการอ่านเท่านั้น

มอดูลชื่อ *SWITCH* ใหม่นี้จะมีการประกาศ *Sort* เพิ่มใหม่ที่เป็น *Sort* แบบซ่อนชื่อ *Switch* โดยใช้เครื่องหมาย *[ ]* ดังนี้

[Switch]

เรามักจะเห็น *Sort* แบบซ่อนที่ใช้ชื่อเหมือนกับชื่อมอดูล ซึ่งมักจะเลือกชื่อที่สื่อถึงสิ่งที่เราสนใจอยู่ กล่าวคือเราสนใจอุปกรณ์สวิตช์ดังนั้นเราจึงตั้งชื่อมอดูลว่า *SWITCH* และตั้งชื่อ Sort แบบซ่อนว่า *Switch* ในภาษาคาเฟโอบีเจการเขียนตัวอักษรจะเป็นแบบไวต่ออักษรใหญ่เล็ก (case sensitive)

ส่วนการประกาศการดำเนินการจะเป็นสิ่งต่อไปที่ผู้เขียนข้อกำหนดต้องระบุ โดยมีการเขียนกำหนดการดำเนินการแบบมีเงื่อนไขและไม่มีเงื่อนไข ทั้งนี้เพื่อให้ง่ายในการเข้าใจเมื่อเปรียบเทียบกันแนวคิดของการทำงานซอฟต์แวร์ การดำเนินการจะมองได้ว่าเป็น การดำเนินการที่ทำหน้าที่เป็นการดำเนินการแบบตัวสร้าง (constructor operation) การดำเนินการแบบค่าคงที่ การดำเนินการแบบ *Method* และแบบ *Attribute*

ในตัวอย่างจะมีการประกาศการดำเนินการดังนี้

```
op init : -> Switch
bop on_ : Switch -> Switch      -- method
bop off_ : Switch -> Switch     -- method
bop state_ : Switch -> Value   -- attribute
```

การดำเนินการชื่อ *init* เป็นการสร้าง *Switch* ขึ้นเมื่อมีการเรียกใช้การดำเนินการนี้จะได้ผลลัพธ์เป็น *Switch* ออกมา ในขณะที่การดำเนินการชื่อ *on_* หรือ *off_* หรือ *state_* เป็นการดำเนินการเชิงพฤติกรรม กล่าวคือมี *Arity* ที่เป็น *Sort* แบบซ่อนเท่ากับ 1 *Sort* เท่านั้น คือ *Switch* นั่นเอง

การดำเนินการชื่อ *on_* และการดำเนินการชื่อ *off_* จะเป็นการดำเนินการแบบ *Method* กล่าวคือ เป็นการดำเนินการที่กระทำกับ *Switch* แล้วได้ผลลัพธ์เป็น *Switch* เหมือนเดิม คำว่า *Method* คือการกระทำที่เกิดกับ *Sort* แบบซ่อน การกระทำได้กระทำกับ *Sort* และไม่สามารถสังเกตได้เพราะเกิดภายใน *Sort* แต่

จะสามารถสังเกตได้โดยใช้การดำเนินการแบบ *Attribute* คือเป็นการกระทำกับ *Sort* แบบซ่อนและได้ผลที่สังเกตได้ ในตัวอย่าง *Switch* ถูกส่งเข้าไปเพื่อถูกกระทำจาก *status* และให้ผลออกมาเป็น *Value* ที่สังเกตได้ เนื่องจาก *Value* เป็น *Sort* แบบเห็นได้

ถัดจากส่วนประกาศลายเซ็นแล้วจะเป็นส่วนการประกาศสัจพจน์ ซึ่งมีการกำหนดตัวแปรและกำหนดประโยคสมการทั้งแบบมีเงื่อนไขและไม่มีเงื่อนไข ดังนี้

```
var S : Switch


eq state init = off .
eq state(on S) = on .
eq state(off S) = off .
```

จากตัวอย่างจะเห็นได้ว่า มีการกำหนดตัวแปรชื่อ *S* ให้มีแบบชนิดเป็น *Sort* แบบซ่อน *Switch* ดังนั้นตัวแปร *S* จะได้รับการนำไปใช้ในสมการต่าง ๆ ที่ต้องการอ้างถึง *Sort* แบบซ่อน *Switch* ใด ๆ ได้

สมการ eq state init = off .

มีความหมายว่า ถ้านำ *Switch* ที่ได้จากการดำเนินการ *init* ไปตรวจดูสถานะที่สังเกตได้โดยใช้ *state* ผลที่ได้จะเป็น *Value* ใด ๆที่สังเกตเห็นได้ ซึ่งจะมีค่าเท่ากับ *Value* ที่ได้จากการดำเนินการค่าคงที่ชื่อ *off*

สมการ eq state(on S) = on .

มีความหมายว่า ถ้านำ *S* ซึ่งเป็น *Switch* ใด ๆที่มีอยู่แล้วไปกระทำด้วยการดำเนินการชื่อ *on_* หรือ on *S* แล้ว *Switch S* จะเปลี่ยนไปโดยเมื่อสังเกตสถานะด้วย *state* แล้วจะมีค่าเท่ากับค่าคงที่ *Value* ที่ได้จากการดำเนินการค่าคงที่ชื่อ *on* เสมอ

สมการ eq state(off S) = off .

หมายถึงถ้านำ *S* ซึ่งเป็น *Switch* ใด ๆที่มีอยู่แล้วไปกระทำด้วยการดำเนินการชื่อ *off_* หรือ *off S* แล้วผลที่ได้คือ *Switch S* ที่มีการเปลี่ยนแปลงโดยจะสังเกตสถานะได้ด้วย *state* และให้ผลเป็น *Value* ที่มีค่าเท่ากับผลของการดำเนินการค่าคงที่ชื่อ *off* เสมอ

กรณีศึกษาเรื่อง Counter

เมื่อได้ข้อกำหนดของ *Switch* ที่เราสนใจแล้ว โดยที่ *Switch* ที่กำหนดมาแล้วนั้นจะมีการดำเนินการที่สามารถเปิดหรือปิด *Switch* และสามารถสังเกตสถานะได้ว่ามีค่าคงที่เป็น *on* หรือ *off* ได้

เราจะกำหนดข้อกำหนดของ *Counter* ที่สนใจเพิ่มอีกสิ่งหนึ่ง จากนั้นจะนำ *Counter* และ *Switch* มาประกอบกันเป็น *Counter with Switch* ในภายหลัง

ข้อกำหนดคาเฟโอบีเจสำหรับ COUNTER

```
-- ----------------------------------------------------------
-- COUNTER
-- ----------------------------------------------------------
mod* COUNTER {
  protecting(INT)

  *[ Counter ]*

  op init : -> Counter
  bop add : Int Counter -> Counter     -- method
  bop read_ : Counter -> Int           -- attribute

  var I : Int
  var C : Counter

  eq read init = 0 .
  eq read add(I, C) = I + read C .
}
```

คำอธิบายข้อกำหนดคาเฟโอบีเจสำหรับ SWITCH

ข้อกำหนดสิ่งที่สนใจคือ *Counter* ซึ่งผู้เขียนมักจะใช้ชื่อ *COUNTER* และ *Counter* แทนชื่อมอดูลและ *Sort* แบบซ่อนตามลำดับ ทั้งนี้เพื่อให้ง่ายในการทำความเข้าใจในการอ่านข้อกำหนด

มอดูลชื่อ *COUNTER* เป็นมอดูลแบบหย่อน ซึ่งหมายถึงเราอาจจะมี *Counter* จำนวนหลายๆ *Counter* ได้ในระบบนี้ และเนื้อหาข้อกำหนดจะได้รับการเขียนภายในวงเล็บ { … } เสมอ

```
mod* COUNTER { … }
```

จากนั้นถ้ามีการนำเข้าก็ให้เขียนถัดต่อไปได้ ในตัวอย่างเป็นการนำเข้าแบบ *Protecting* และมอดูลชื่อ *INT* ได้รับการนำเข้า มอดูล *INT* เป็นมอดูลที่กำหนดจำนวน *Integer* ดังนี้

```
protecting(INT)
```

จากตัวอย่างมีการกำหนด *Sort* แบบซ่อนชื่อ *Counter* หมายถึงตัวนับ ดังนี้

```
*[ Counter ]*
```

ส่วนประกาศการดำเนินการต่อไปนี้ระบุการสร้าง *Counter* การบวกตัวนับ *Counter* และการอ่านตัวนับ *Counter* ดังต่อไปนี้

```
op init : -> Counter
bop add : Int Counter -> Counter    -- method
bop read_ : Counter -> Int          -- attribute
```

การดำเนินการแบบตัวสร้างชื่อ *init* เป็นการสร้าง *Counter* ใหม่ใดๆ ขึ้นมา และมีการดำเนินการเชิงพฤติกรรมชื่อ *add* และชื่อ *read_* กำหนดไว้โดยนำเข้าตัวนับ *Counter* ใด ๆ ไปใช้ในการกระทำ

ส่วนต่อไปนี้เป็นการประกาศตัวแปรชื่อ *I* และชื่อ *C* ซึ่งทำหน้าที่เป็นจำนวน *Integer* และ *Counter* ตามลำดับ

```
var I : Int
var C : Counter
```

สมการต่อไปนี้เป็นการระบุความสัมพันธ์ระหว่างการดำเนินการที่มีอยู่

```
eq read init = 0 .
```

สมการมีความหมายถึงการนำตัวนับ *Counter C* ใดๆที่สร้างขึ้นโดยการดำเนินการแบบตัวสร้างชื่อ *init* มาทำการอ่านค่าด้วยการดำเนินการชื่อ *read* ผลลัพธ์จะมีค่าเท่ากับ *0* เสมอ

```
eq read add(I, C) = I + read C .
```

สมการมีความหมายถึงการที่นำตัวนับ *Counter C* ใดๆที่มีอยู่แล้วและค่าจำนวนเต็ม *I* ใดๆค่าหนึ่ง นำเข้าไปในการดำเนินการชื่อ *add* ที่รับ *Arity* อยู่สองตัวคือ *Int* และ *Counter* ผลของการดำเนินการ *add* นี้จะให้ผลการเปลี่ยนแปลงตัวนับ *C* เดิมให้เปลี่ยนไปและเมื่อนำไปผ่านการดำเนินการชื่อ *read* ต่อไปจะให้ผลลัพธ์ที่สังเกตได้เป็น *Int* ซึ่งจะมีค่าตรงกับผลของการดำเนินการชุดด้านซ้ายของสมการ นั่นคือการอ่านค่า *read C* และบวกเข้ากับค่าจำนวนเต็ม *I* ที่มีเช่นเดียวกัน

กรณีศึกษาเรื่อง Counter with Switch

ข้อกำหนดที่เขียนมาแล้วที่เป็นมอดูลชื่อ *ON-OFF* หรือมอดูลชื่อ *SWITCH* และมอดูลชื่อ *COUNTER* ที่ทำหน้าที่ของแต่ละสิ่งที่เราสนใจอยู่ โดยเขียนแยกเป็นส่วนๆ หรือมอดูลๆ เมื่อต้องการนำมาใช้ก็สามารถอ้างถึงโดยการนำเข้าเข้ามาใช้ได้ ดังในตัวอย่างต่อไปนี้ซึ่งเป็นการสนใจตัวนับ *Counter* ที่สามารถนับเพิ่มและนับถอยหลังได้โดยมี *Switch* ที่ประกอบเข้ามาด้วยเพื่อกำหนดวิธีการนับ

ข้อกำหนดคาเฟโอบีเจสำหรับ COUNTER WITH SWITCH

```
-- -------------------------------------------
-- concurrent connection of  SWITCH and COUNTER
-- -------------------------------------------
mod* COUNTER-WITH-SWITCH {
  protecting(SWITCH + COUNTER)

  *[ Cws ]*

  op init : -> Cws
  bop put : Int Cws -> Cws            -- method
  bop add_ : Cws -> Cws               -- method
  bop sub_ : Cws -> Cws               -- method
  bop read_ : Cws -> Int              -- attribute
  bop counter_ : Cws -> Counter       -- projection function
  bop switch_ : Cws -> Switch         -- projection function

  var N : Int
  var C : Cws

  eq read C = read(counter C) .  -- abbreviation equation for "read"

-- ------------------------------------
-- equations for switch
-- ------------------------------------
```

```
  eq switch(init) = init .
  eq switch put(N, C) = switch C .
  eq switch add(C) = on(switch C) .
  eq switch sub(C) = off(switch C) .

-- ------------------------------------
-- equations for counter
-- ------------------------------------
  eq counter(init) = init .
  ceq counter(put(N, C)) = add(N, counter(C))
      if state(switch(C)) == on .
  ceq counter(put(N, C)) = add(-(N), counter(C))
      if state(switch(C)) == off .
  eq counter add(C) = counter C .
  eq counter sub(C) = counter C .
}
```

คำอธิบายข้อกำหนดคาเฟโอบีเจสำหรับ COUNTER WITH SWITCH

ข้อกำหนดมอดูลชื่อ *COUNTER-WITH-SWITCH* อธิบายตัวนับที่สามารถนับเพิ่มหรือนับถอยหลังได้โดยมี *Switch* กำหนดทิศทางการนับ โดยมีตัวอย่างดังต่อไปนี้

```
mod* COUNTER-WITH-SWITCH { … }
```

มอดูลนี้มีการนำเข้ามอดูลอีกสองมอดูลชื่อ *SWITCH* และ *COUNTER* โดยนำเข้ามาแบบ Protecting โดยประกาศประโยคดังนี้

```
protecting(SWITCH + COUNTER)
```

จากนั้นก็มีการประกาศ *Sort* แบบซ่อนชื่อ *Cws* เพื่อให้หมายถึงสิ่งที่เราสนใจคือตัวนับที่มี *Switch* นั่นเองดังนี้

```
*[ Cws ]*
```

การดำเนินการของมอดูลนี้ประกาศไว้ดังนี้

```
op init : -> Cws
```

เป็นการดำเนินการแบบตัวสร้าง ตัวนับ *Cws* ใด ๆ ได้

```
bop put : Int Cws -> Cws          -- method
```

เป็นการดำเนินการเพื่อระบุเลขจำนวนเต็ม *Int* เพื่อส่งให้กับตัวนับ *Cws* เป็นการตั้งค่าเริ่มต้นก่อนการนับสำหรับตัวนับ *Cws* ใด ๆ

```
bop add_ : Cws -> Cws            -- method
bop sub_ : Cws -> Cws            -- method
```

เป็นการดำเนินการเชิงพฤติกรรมเพื่อการนับเพิ่มโดยใช้ *add_* และเพื่อทำการนับถอยหลังโดยใช้ *sub_* โดยระบุตัวนับ *Cws* ที่ถูกกระทำ ผลลัพธ์ที่ได้คือ ตัวนับ *Cws* ตัวนั้นๆที่ผ่านการกระทำแล้ว

```
bop read_ : Cws -> Int           -- attribute
```

เป็นการดำเนินการเชิงพฤติกรรมเพื่อใช้ในการอ่านค่าตัวนับ *Cws* ที่ต้องการและให้ผลลัพธ์เป็นค่า *Int* ที่มีอยู่ใน *Cws* นั้นๆได้

```
bop counter_ : Cws -> Counter    -- projection function
```

เป็นการดำเนินการเชิงพฤติกรรมเพื่อใช้ในการเลือกส่วนประกอบของตัวนับ *Cws* โดยเลือกเฉพาะส่วนที่เป็น *Counter* เราเรียกการดำเนินการแบบนี้ว่าเป็นการทำ *Projection* เพื่อฉายภาพเฉพาะส่วนประกอบที่ต้องการเท่านั้น

```
bop switch_ : Cws -> Switch      -- projection function
```

เป็นการดำเนินการเชิงพฤติกรรมเพื่อใช้ในการเลือกส่วนประกอบของตัวนับ *Cws* โดยเลือกเฉพาะส่วนที่เป็น *Switch* เท่านั้น ซึ่งตัวนับ *Cws* นั้นอันที่จริงจะประกอบด้วยทั้งส่วน *Counter* และส่วนที่เป็น *Switch* การทำ *Projection* นี้จะฉายภาพเฉพาะส่วนประกอบที่ต้องการเท่านั้น

ต่อไปนี้เป็นส่วนสัจพจน์ โดยประกาศตัวแปรที่มีชื่อ *N* และ *C* ซึ่งมีแบบชนิดเป็น *Int* และ *Cws* ตามลำดับดังนี้

```
var N : Int
var C : Cws
```

ต่อจากนี้เป็นสมการที่แสดงความสัมพันธ์ระหว่างการดำเนินการ อย่างไรก็ตามเราสามารถกำหนดสมการเพื่อช่วยในการกำหนดการย่อคำสำหรับการดำเนินการหรือการเขียนแบบย่อ โดยเราสามารถเขียนการดำเนินการแบบย่อได้โดยเมื่อเขียนแบบด้านขวาของสมการ ก็จะเหมือนกับเราเขียนการดำเนินการแบบด้านซ้ายเสมอ

```
eq read C = read(counter C) .  -- abbreviation equation for "read"
```

สมการนี้กำหนดไว้เพื่อต้องการให้เขียน *read C* แทน *read(Counter C)* ได้เสมอ เพื่อให้เขียนแบบย่อได้ กล่าวคือการเขียน *read C* ก็เพื่อการอ่านค่าส่วน *Counter* ของตัวนับ *Cws C* ใดๆ ผลลัพธ์จะเป็นค่า *Int* เสมอ

เมื่อเราสังเกตดูจะเห็นได้ว่าจากการดำเนินการ *read C* จะเป็นชื่อ *read* ที่กำหนดไว้ในมอดูลนี้ *COUNTER-WITH-SWITCH* เพราะว่า

```
bop read_ : Cws -> Int                -- attribute
```

สมการนี้รับ *Cws* เป็นตัวแปรนำเข้า

ส่วน *read(Counter C)* จะเป็นชื่อการดำเนินการ *read* ที่อ้างมาจากมอดูล *Counter* เพราะว่า

```
bop read_ : Counter -> Int           -- attribute
```

สมการนี้มาจากมอดูล *COUNTER* โดยจะมีการส่ง *Counter* เข้าไปและได้ค่า *Int* เป็นผลลัพธ์ออกมาเสมอ

ต่อไปนี้เป็นการอธิบายสมการต่าง ๆที่เกี่ยวข้องกับการดำเนินการของ *Switch* ซึ่งเขียนรวมไว้ใกล้กันเพื่อง่ายต่อการติดตามและตรวจสอบเกี่ยวกับ *Switch*

```
-- ------------------------------------
-- equations for switch
-- ------------------------------------
eq switch(init) = init .
```

สมการนี้เป็นการระบุว่าเมื่อนำตัวนับ *Cws* ใด ๆที่เกิดจากการดำเนินการแบบตัวสร้าง *init* มาเลือกฉายภาพ (Projection) เฉพาะส่วนประกอบที่เป็น *Switch* ด้วยการดำเนินการ *switch* แล้ว ผลลัพธ์ที่ได้จะเป็น *Switch* ใด ๆ ที่มีค่าเท่ากับด้านขวาของสมการ และด้านขวาเป็น *Switch* ใด ๆที่ได้จากการสร้าง *Switch* ด้วยการดำเนินการชื่อ *init* ที่ปรากฏในมอดูล *SWITCH* นั่นเองดังนี้

```
op init : -> Switch
```

เราสังเกตได้ว่ามีชื่อการดำเนินการซ้ำกันในสมการนี้ คือ *init* ด้านซ้ายของสมการและ *init* ด้านขวาของสมการ ดังนั้นต้องระวังในการอ่านและตีความด้วย เราสามารถพิจารณาว่า *init* ด้านขวาเป็นการดำเนินการของมอดูลใดคือมอดูล *COUNTER-WITH-SWITCH* ได้เพราะเราจะสังเกต *Arity* ของ *init* ว่าเป็น *Sort* อะไรได้

เช่นเดียวกันการสังเกตว่าการดำเนินการของ *init* ด้านขวามาจากมอดูลชื่อ *SWITCH* ได้ก็เพราะการสังเกต *Arity* ด้วย ดังนั้นการกำหนด *Arity* และ *Co-arity* จึงมีความสำคัญมากในการประกาศการดำเนินการ และการนำเข้ามอดูลใดที่มีการดำเนินการชื่อซ้ำและมีการใช้ *Arity* และ *Co-arity* ซ้ำเหมือนกันทั้งหมดจะไม่เป็นที่ยอมรับได้ กรณีการทำ *Overloading* ของการดำเนินการก็ไม่มีการซ้ำเหมือนกันทั้งหมดเช่นกัน

```
eq switch put(N, C) = switch C .
```

สมการข้างต้นเป็นการกำหนดว่าถ้าทำการ *put* ตัวเลข *Int N* เข้าสู่ *Cws C* แล้ว ผลลัพธ์ที่ได้ยังคงเป็น *Cws C* เหมือนเดิม ซึ่งคาดว่าค่าภายในคือ *Counter* และ *Switch* ยังคงเหมือนเดิม

สังเกตว่าการดำเนินการ *put* นั้นเป็นการเตรียมพร้อมสำหรับการดำเนินการ *add* และ *sub* ซึ่งจะมีการกระทำตามมาอีกต่อหนึ่ง

```
eq switch add(C) = on(switch C) .
```

สมการนี้เป็นการกำหนดว่า *Cws C* ใด ๆ ถ้าถูกกระทำโดยการดำเนินการ *add* แล้ว ถ้าสนใจเฉพาะส่วนประกอบที่เป็น *Switch* โดยการดำเนินการแบบ *Projection* ชื่อ *switch* ของมอดูลนี้ ผลลัพธ์จะเป็น *Switch* ที่มีค่าเท่ากับ *Switch* ทีได้จากการนำ *Cws C* ใด ๆ มา และสนใจเฉพาะส่วนประกอบ *Switch* จากนั้นก็ถูกกระทำด้วยการดำเนินการ *on* จากมอดูล *SWITCH*

กล่าวโดยสรุปคือ เมื่อมีการดำเนินการ *add(C)* แล้วค่า *switch* ของมันจะมีค่าเท่ากับการทำการ *on* แก่ *Switch* ของ *C* นั้นเสมอ

```
eq switch sub(C) = off(switch C) .
```

เช่นเดียวกันสมการนี้เป็นการดำเนินการ *sub(C)* ซึ่งค่า *switch* ของมันจะมีค่าเท่ากับการทำการ *off* ของ *Switch C* นั้นเสมอ

ต่อไปนี้เป็นการรวบรวมประโยคสมการที่เกี่ยวข้องกับ *Counter*

```
-- ------------------------------------
-- equations for counter
-- ------------------------------------
```

ประโยคข้างต้นเป็นหมายเหตุ (Comment) เพื่อเตือนความจำ

```
eq counter(init) = init .
```

สมการนี้เป็นการกำหนดว่าการดำเนินการ *init* ของมอดูล *COUNTER-WITH-SWITCH* จะให้ *Cws* ใดๆ และส่วนประกอบที่เป็น *Counter* จะมีค่าเท่ากับ *Counter* ที่สร้างจากการดำเนินการ *init* ของมอดูล *COUNTER*

เช่นเดียวกันการดำเนินการ *init* ด้านซ้ายของสมการไม่ได้เป็นการดำเนินการเดียวกันกับการดำเนินการ *init* ด้านขวาของสมการ

```
ceq counter(put(N, C)) = add(N, counter(C))
    if state(switch(C)) == on .
```

ข้างต้นเป็นสมการแบบมีเงื่อนไข ซึ่งปรากฏหลังคำว่า *if* ดังนั้นกรณีเงื่อนไข *state(switch(C)) == on* เป็นจริงแล้ว สมการด้านหน้าจะได้รับการกระทำทันทีได้

เงื่อนไขที่จะต้องเป็นจริงคือ ถ้ากำหนดให้ *Cws C* ใด ๆ หนึ่งตัว สถานะ *state* ของ *Switch* ที่เป็นส่วนหนึ่งของ *C* นั้นจะต้องมีค่าเท่ากับค่าคงที่ on

ถ้าเงื่อนไขเป็นจริงแล้ว การทำการ *put* จำนวน *int N* ใด ๆ เข้ากับ *Cws C* แล้ว ส่วนประกอบที่เป็น *Counter* ของ *C* นั้นจะมีค่าเท่ากับ การดำเนินการ *add* แบบ *Counter* ที่กำหนดไว้ในมอดูล *COUNTER* การนำส่วนประกอบ *Counter* ของ *Cws C* นั้นๆมาเพิ่มด้วยจำนวน *Int N*

```
ceq counter(put(N, C)) = add(-(N), counter(C))
    if state(switch(C)) == off .
```

เป็นสมการแบบมีเงื่อนไขเช่นเดียวกัน โดยถ้าเงื่อนไข *state(switch(C)) == off* เป็นจริงแล้ว จะมีการกระทำตามสมการทันที

เงื่อนไขที่ต้องเป็นจริงคือ ถ้ากำหนดให้ *Cws C* ใด ๆ แล้ว สถานะของ *Switch* ของ *C* จะมีค่าเป็น *off*

ดังนั้น ถ้าเงื่อนไขเป็นจริงแล้ว การทำการ *put* จำนวน *int N* ใดๆเข้ากับ *Cws C* แล้ว ส่วนประกอบที่เป็น *Counter* ของ *C* นั้นจะมีค่าเท่ากับ การดำเนินการ *add* แบบ *Counter* ที่กำหนดไว้ในมอดูล *COUNTER* การนำส่วนประกอบ *Counter* ของ *Cws C* นั้นๆมาลบด้วยจำนวน *Int N*

```
eq counter add(C) = counter C .
eq counter sub(C) = counter C .
```

สมการข้างต้นนี้กำหนดว่า การดำเนินการ *add* หรือการดำเนินการ *sub* ที่กระทำกับ *Cws C* นั้นจะไม่มีผลกระทบกับส่วนที่เป็น *Counter* ของ *C* เลย เพราะไม่ว่าจะมีการดำเนินการ *add(C)* หรือ *sub(C)* แล้วก็ตามเมื่อดูเฉพาะส่วนประกอบ *Counter* ของ *C* นั้นๆแล้วจะมีค่าเท่ากับ *Counter* เดิมของ *C* เสมอ อย่างไม่มีเงื่อนไข

4.13 ภาษาโพรเมลา

ภาษาโพรเมลาไม่ถูกจัดว่าเป็นภาษาที่ใช้เขียนโปรแกรมสำเร็จรูปทั่วไปได้ (programming language) เนื่องจากโพรเมลาถูกออกแบบมาเพื่อเป็นภาษาข้อกำหนดเชิงรูปนัยเสียมากกว่า และชุดคำสั่งก็มีความพิเศษที่ทำงานในลักษณะเชิงไม่กำหนด (non-deterministic) สำหรับชื่อ *"Promela"* ได้มาจากตัวย่อของ *"**Pro**cess **me**ta-**la**nguage"* ซึ่งโพรเมลาเดิมทีเดียวถูกออกแบบมาเพื่อใช้เป็นภาษาที่สร้างแบบจำลองพฤติกรรมการทำงานของกระบวนการซอฟต์แวร์ที่ทำงานเป็นหลายสายโยงใย และมีการติดต่อประสานงานกัน เพื่อใช้ในการทวนสอบกระบวนการที่หน้าที่เป็นโพรโตคอลสื่อสาร ต่อไปนี้จะขอแนะนำโพรเมลาเบื้องต้นเพื่อให้ทราบโดยสังเขป และยกตัวอย่างการใช้ชุดคำสั่งโพรเมลาที่ทำงาน

ในลักษณะเชิงไม่กำหนดเท่านั้น ส่วนคำสั่งอื่นๆ ติดตามดูรายละเอียดเพิ่มเติมได้ที่คู่มือใช้งานได้ที่ *www.spinroot.com*

โพรเมลามีองค์ประกอบ คือ กระบวนการ การประกาศตัวแปรและชนิดตัวแปร ชุดคำสั่งควบคุม ชุดคำสั่งประมวลผล ชุดคำสั่งพิเศษอื่นที่ใช้ในการสื่อสารระหว่างสายโยงใย เช่น การสร้าง *channel* ที่เป็นท่อส่งข้อความผ่านไปมาระหว่างกระบวนการได้ มีการรอรับข้อความและรอส่งข้อความเมื่ออีกฝ่ายยังไม่พร้อมส่งหรือพร้อมรับเช่นกัน คำสั่งพิเศษที่ใช้ในการจัดกลุ่มคำสั่งให้ทำงานแบบต่อเนื่องกันโดยแบ่งแยกไม่ได้ คือ *atomic{..}* เป็นต้น

กระบวนการ

ส่วนกระบวนการเป็นกรอบโครงสร้างที่ใช้กำหนดพฤติกรรมของสายโยงใย โดยแบ่งเป็นกระบวนการหลัก *init* และกระบวนการ *proctype*

กระบวนการหลัก *init* เป็นโครงสร้างที่ใช้กำหนดพฤติกรรมสายโยงใยหลักแรกที่เริ่มทำงาน ทำหน้าที่เป็นจุดเริ่มต้นของระบบจุดเดียวเท่านั้น โดยทั่วไปกระบวนการหลัก *init* จะประกอบด้วยการประกาศตัวแปร มีชุดคำสั่งที่ทำงานตามต้องการได้ และใช้คำสั่งในการสั่งให้กระบวนการ *proctype* อื่นเริ่มทำงานได้ผ่านคำสั่ง *run*

กระบวนการ *proctype* เป็นโครงสร้างที่ใช้กำหนดพฤติกรรมสายโยงใยอื่นที่มีในระบบ กระบวนการนี้จะประกอบด้วยการประกาศตัวแปร และมีชุดคำสั่งที่ทำงานได้ตามต้องการเช่นเดียวกับกระบวนการอื่น กระบวนการ *proctype* ถูกเรียกให้เริ่มทำงานได้สองแบบ แบบแรกโดยใช้คำสั่ง *run* จากกระบวนการอื่น โดยทั่วไปมักจะถูกเรียกให้ทำงานโดยกระบวนการหลัก *init* แบบที่สองโดยระบุให้เป็นกระบวนการที่ทำงานอัตโนมัติแบบ *active*

ตัวอย่างโปรแกรมแรกที่ง่ายที่สุด คือ โปรแกรม *HelloWorld* ดังนี้

```
proctype helloWorld() {
   printf("Hello World");
}
init {
    run helloWorld();
}
```

จากนั้นลองดูตัวอย่างที่มีการใช้งานกระบวนการแบบ *active* ดังนี้

```
proctype myProcess() {
        bool  flag;
          flag = 1;
          printf("This is myProcess \n");
}
active proctype autoProcess() {
          int a = 5;
          printf("This is auto start process…a = %d\n", a);
}
init {
        printf("This is init process");
        run myProcess();
}
```

จากตัวอย่างข้างต้น กระบวนการหลัก *init* เริ่มทำงานก่อนเสมอ ในขณะเดียวกันกระบวนการแบบ *active* ก็เริ่มทำงานทำงานโดยอัตโนมัติด้วยเช่นกันคือ *proctype* ชื่อ *autoProcess()* ในจังหวะแรกนั้น *init* จะเริ่มพิมพ์ประโยคข้อความแรกก่อน และเรียกกระบวนการ *proctype* ชื่อ *myProcess()* เริ่มทำงานได้ การทำงานทั้งหมดเป็นการทำงานของสามสายโยงใย ที่ต่างคนต่างทำกันได้

การประกาศตัวแปรและชนิดตัวแปร

เราสามารถประกาศตัวแปรเพื่อใช้ในกระบวนการ *proctype* ได้ โดยประกาศเป็นตัวแปรส่วนกลางหรือตัวแปรท้องถิ่นได้ และมีชนิดตัวแปรจำนวนหนึ่งที่นำมาใช้ได้ คือ *bit, bool, byte, short, int, unsigned, mtype* เป็นต้น ซึ่งมีไม่มากนักเนื่องจากไม่ได้ถูกออกแบบมาใช้ในการประมวลผลข้อมูลในโปรแกรมประยุกต์

สำหรับชนิดตัวแปรแบบ *mtype* เราสามารถประกาศชื่อสัญลักษณ์ที่ใช้แทนค่า *0..255* (symbolic name) โดยใช้คำสั่ง *mtype* เช่น *mtype: dayofWeek = {Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday}* ซึ่งจะมีความหมายว่า *Sunday* มีค่าเท่ากับ *0* ส่วน *Monday* มีค่าเท่ากับ *1* และ *Tuesday* มีค่าเท่ากับ *2* ตามลำดับเรียงไป

เรามีตัวดำเนินการให้ใช้งานได้ คือ ตัวดำเนินการคณิตศาสตร์ *+, -, *, /, %* สำหรับการบวก ลบ คูณ หาร และมอดูโล ตามลำดับ และตัวดำเนินการตรรกะ *&&,* || สำหรับการทำ *"และ" "หรือ"* เป็นต้น

การใช้คำสั่งควบคุม

โพรเมลามีคำสั่งที่ใช้ควบคุมลำดับการทำงานของโปรแกรม คือ คำสั่ง *if..fi* และ คำสั่ง *do..od* เป็นต้น แต่ภายในคำสั่งทั้งสองนั้น จะเป็นการเลือกทำงานแบบเชิงไม่กำหนด ถ้าไม่มีการกำหนดเงื่อนไขกำกับ

ขอยกตัวอย่างต่อไปนี้

```
active proctype Count() {
int count =5;
if
      :: count = count +1;
      :: count = count +2;
      :: count = count +3;
fi;
printf (count);
}
```

โพรเมลาข้างต้นจะทำคำสั่ง *if..fi* โดยสามคำสั่งที่ไม่มีส่วนเงื่อนไขกำกับไว้ ทำให้การทำงานจะเป็นแบบเชิงไม่กำหนด กล่าวคือ โพรเมลาจะเลือกแบบสุ่มในการทำคำสั่งใดก็ได้ในสามคำสั่งของ *if..fi* เช่น ในการทำงานครั้งแรกอาจจะเลือกคำสั่ง *count=count+2* แล้วก็พิมพ์ค่า count และจบการทำงาน แต่ถ้าเราเรียก *proctype Count()* อีกครั้งก็มีการเลือกแบบสุ่มอีกเช่นกัน ทำให้เราไม่สามารถคาดเดาได้ว่าจะทำคำสั่งไหนใน *if..fi*

ดังนั้นเราต้องทำการกำกับเงื่อนไขไว้ก่อนประโยค ดังตัวอย่างข้างล่างนี้

```
active proctype P() {
int a= 6, b=5;
int max;
int branch;
if
      :: a >= b -> max = a; branch =1
      :: b >= a -> max = b; branch =2
      :: else -> branch = 3
fi;
printf ("The max of %d and %d = %d by branch %d\n",
                    a, b, max, branch)
}
```

จากตัวอย่างโพรเมลาข้างต้น อธิบายได้ว่ามี *proctype* ชื่อ *P()* กำหนดให้มีตัวแปร *a=6, b=5, max=0, branch=0* ในตอนเริ่มต้น จากนั้นให้ใช้คำสั่ง *if..fi* โดย จะมีการพิจารณาเงื่อนไข ที่เป็นส่วนแรกของประโยคคำสั่งใน *if..fi* ถ้าเงื่อนไขใดเป็นจริงให้ทำคำสั่งนั้น แต่ถ้าไม่มีเงื่อนไขใดจริงเลยให้ทำคำสั่ง

ประโยคที่กำกับด้วย *else* ในที่นี้เงื่อนไขในประโยคแรกจะเป็นจริงคือ $a >= b$ ทำให้มีการกำหนดค่า *max = a; branch =1* ตามมานั่นเอง

กรณีคำสั่ง *do..od* ก็เป็นไปในทำนองเดียวกัน

```
active proctype CountDo() {
int count =5;
do
      :: count = count +1;
      :: count = count +2;
      :: count = count +3;
     :: count>10 -> break
 od;
}
```

โพรเมลาจะทำคำสั่งใน *do..od* วนไปตลอดจนกว่าจะออกมาได้ โดยในที่นี้ โพรเมลาจะเลือกแบบสุ่มในการบวกค่าให้กับ *count* ในการวนแต่ละรอบก็ไม่รู้แน่ว่าจะทำคำสั่งใด คาดเดาไม่ได้เลย จนกระทั่งเงื่อนไข *count>10* เป็นจริงก็ให้ออกจาก *od..od* ได้ สำหรับคำสั่งนอกจากนี้ให้ติดตามรายละเอียดได้ที่ www.spinroot.com

4.14 แบบฝึกหัด

1) ภาษาในกลุ่มโมเดลเบสมักจะประกอบด้วยส่วนประกอบใดเป็นหลัก?
2) ภาษาในกลุ่มพร็อพเพอร์ตีประกอบด้วยส่วนประกอบใดเป็นหลัก?
3) จงยกตัวอย่างภาษาในกลุ่มโมเดลเบส?
4) จงยกตัวอย่างภาษาในกลุ่มพร็อพเพอร์ตีเบส?
5) การเขียนข้อกำหนดเชิงรูปนัยของระบบมักจะต้องเริ่มพิจารณาระบบอย่างไรบ้าง?
6) ตัวยืนยงข้อมูล (data invariant) ต่างกับเงื่อนไขก่อน (precondition) อย่างไร?
7) *Schema* ในภาษาเซดคืออะไร มีกี่ประเภท?
8) เราเขียน *Schema* ในภาษาเซดได้อย่างไร กี่แบบ?
9) เราเขียนอธิบายระบบด้วยข้อกำหนดแบบโมเดลเบสได้อย่างไรโดยใช้ภาษาเซด?
10) กีเวนเซตคืออะไร มีประโยชน์อย่างไร?
11) การทำการนำเข้า (inclusion) ของ *Schema* ในภาษาเซดทำได้กี่แบบ คืออะไรบ้าง?

12) กำหนดให้ *Schema* ชื่อ *Book* ต่อไปนี้

```
── Book ──────────────────────
│ author: People
│ title: ℙ(ℤ X CHAR)
│ readership: ℙPeople
│ rating: ℙ(People X ℤ)
├─────────────────────────────
│ title ∈ seq CHAR
│ rating ∈ People ⇸ 0..10
│ readership = dom rating
└─────────────────────────────
```

จงระบุค่าตัวยืนยงข้อมูล (data invariant) ของ *Schema* ชื่อ *Book*

13) โครงสร้างของ *MACHINE* หนึ่งๆในข้อกำหนดเชิงรูปนัย *AMN* ต่างกับโครงสร้างของ *Schema* หนึ่งๆในการเขียนข้อกำหนดเชิงรูปนัยด้วยภาษาเซดอย่างไร?
14) การกำหนดให้มีการเลือกเส้นทางการทำงานได้หลายอย่างในภาษา *AMN* ทำได้อย่างไร?
15) ข้อกำหนดข้างล่างนี้หมายถึงอะไร

    report, seat := *good, seat-nbr*

16) การกำหนดให้มีการส่งผลลัพธ์ออกมาจากการดำเนินการใดๆ ในภาษา *AMN* ทำได้อย่างไร?
17) มอดูล (module) ในภาษาคาเฟโอบีเจคืออะไร มีกี่ประเภท?
18) เราเขียนอธิบายระบบด้วยข้อกำหนดแบบพรอพเพอร์ตีเบสได้อย่างไรโดยใช้ภาษาคาเฟโอบีเจ?
19) *Sort* ต่างกับ *Type* อย่างไร?
20) *Hidden Sort* คืออะไร มีประโยชน์อย่างไร?
21) สมการที่เขียนในภาษาคาเฟโอบีเจมีกี่ประเภท?
22) ในภาษาคาเฟโอบีเจสามารถทำ *Projection* ได้อย่างไร?
23) กระบวนการ *active proctype* มีประโยชน์อย่างไร?
24) การทำงานแบบเชิงไม่กำหนดของโพรเมลาเป็นอย่างไร อธิบาย?
25) การสร้างเงื่อนไขเพื่อให้ออกจาก *do..od* ในโพรเมลาทำอย่างไร?

5

สร้างแบบจำลองเชิงรูปนัยด้วยแผนภาพ

5.1 ความสำคัญของบทนี้

บทนี้เป็นเนื้อหาเกี่ยวกับแผนภาพที่มีนิยามวากยสัมพันธ์และความหมายชัดเจนนำมาใช้อธิบายแบบจำลองเชิงรูปนัยของระบบได้ บางแผนภาพมีเครื่องมือสนับสนุนนำมาทวนสอบได้ด้วย เช่น ออโตมาตา โครงสร้างครีปกี บูชีออโตมาตา เพทริเน็ต เป็นต้น

ตามที่เราทราบมาแล้วว่า การทวนสอบจะต้องมีสองสิ่งที่จำเป็น คือ แบบจำลองเชิงรูปนัยของระบบ และคุณลักษณะเงื่อนไขที่ใช้ทวนสอบที่เขียนด้วยสูตรเชิงเวลา โดยที่นิยมกันก็คือ การอธิบายแบบจำลองเชิงรูปนัยโดยใช้แผนภาพที่สามารถแสดงสถานะและการเปลี่ยนแปลงไปมาของสถานะได้ และออโตมาตาเป็นหนึ่งในแผนภาพที่นิยมนำมาใช้อธิบายแบบจำลองเชิงรูปนัย

อย่างไรก็ตามการเขียนออโตมาตา หรือใช้แผนภาพสถานะอื่นของระบบขนาดใหญ่มักจะยุ่งยากมาก เมื่อมีจำนวนของสถานะและการเส้นเชื่อมต่อสถานะที่มากและซับซ้อน เราจึงเลี่ยงมาเขียนอธิบายระบบด้วยแผนภาพแบบอื่นแทน เช่น แผนภาพยูเอ็มแอล เพทริเน็ต เป็นต้น

5.2 วัตถุประสงค์

- เพื่อให้เข้าใจนิยามพื้นฐานของแผนภาพทางเลือก
- เพื่อให้เข้าใจการสร้างแบบจำลองเชิงรูปนัยด้วยแผนภาพ
- เพื่อให้สามารถเลือกแผนภาพเพื่อสร้างแบบจำลองได้อย่างเหมาะสม

5.3 ออโตมาตา

ความเป็นมาและความสำคัญของออโตมาตา (automata) ในวงการนักวิทยาศาสตร์คอมพิวเตอร์เริ่มขึ้นในยุคปี ค.ศ. 1943 ที่มีความพยายามในการอธิบายพฤติกรรมของระบบประสาทของสิ่งมีชีวิตที่เรียกว่า เครือข่ายประสาท (nerve nets) [24] โดยนักประสาทวิทยาสองคน คือ *Warren McCulloch* และ *Walter Pitts* พวกเขาได้ออกแบบการจำลองการทำงานหรือพฤติกรรมของสิ่งมีชีวิต รวมถึงจำลองการนึกคิดมนุษย์ด้วยเช่นกัน โดยใช้แผนภาพแบบออโต

มาตาที่มีสถานะจำกัด ซึ่งเป็นแผนภาพที่ประกอบด้วยกล่องดำที่มีการรับข้อมูลนำเข้า จากนั้นกล่องดำจะนำข้อมูลนำเข้าไปประมวลผลอย่างใดอย่างหนึ่งและให้ผลลัพธ์เป็นข้อมูลส่งออก กล่าวคือ ในขณะที่กำลังรอรับข้อมูลนำเข้านั้นกล่องดำจะอยู่ในสถานะหนึ่งชื่อ q_0 และเมื่อประมวลแล้วได้ผลลัพธ์ก็จะเปลี่ยนสถานะไปเป็นอีกสถานะอื่นถัดไปชื่อ q_1 ได้ หรืออาจจะยังคงอยู่ในสถานะเดิม q_0 ก็ได้ โดยการเปลี่ยนสถานะของกล่องดำนั้นจะขึ้นอยู่กับข้อมูลนำเข้าและสถานะในขณะนั้น

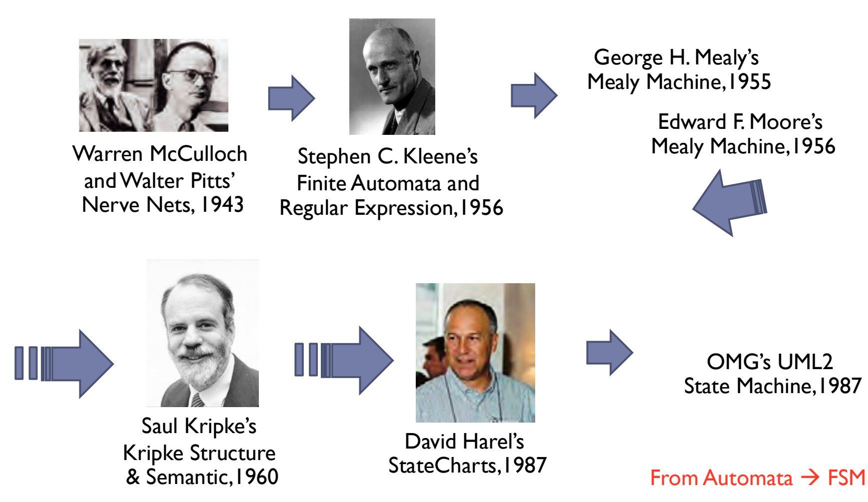
ต่อมาในช่วงปี ค.ศ. 1956 *S.C. Kleene* ได้อ้างถึงผลงานเดิมนี้ของ *Warren* และ *Pitts* และนำเสนอผลงานเรื่องออโตมาตาและนิพจน์ปกติ (regular expression) ซึ่งเป็นที่ยอมรับอย่างมากมายในเวลานั้น นิพจน์ปกติใช้ประโยชน์ในการกำหนดโครงสร้างสตริงซึ่งประกอบด้วยตัวอักขระที่กำหนดให้ นิพจน์ปกติทำงานได้เหมือนออโตมาตาในฐานะตัวยอมรับสตริง และในช่วงเวลาไล่เลี่ยกันราวปี ค.ศ. 1955-1956 นั้น *G.H. Mealy* และ *E.F.Moore* ได้นำเสนอแบบจำลองของการทำงานของออโตมาตา ในรูปแบบของเครื่องจักรมีลลี (Mealy machine) และเครื่องจักรมัวร์ (Moore machine) ซึ่งได้รับการยอมรับ และนำไปประยุกต์ใช้กับการออกแบบพฤติกรรมของระบบฮาร์ดแวร์ของเครื่องจักรคอมพิวเตอร์ในยุคนั้น ทั้งเครื่องจักรมีลลีและเครื่องจักรมัวร์ต่างเป็นทางเลือกในการอธิบายพฤติกรรมการทำงานของระบบฮาร์ดแวร์ โดยแต่ละแบบจะให้ผลลัพธ์ออโตมาตาที่มีความแตกต่างกันทั้งจำนวนสถานะที่ใช้ การแสดงข้อมูลนำเข้าและข้อมูลส่งออก

ต่อมาในปี ค.ศ. 1960 *Saul Kripke* ได้นำเสนอโครงสร้างแบบคริปกีและความหมาย (Kripke structure and semantic) ซึ่งเป็นการปรับปรุงออโตมาตาจากเดิมให้ใช้อธิบายระบบแบบปฏิสัมพันธ์ได้ดีกว่าเดิม จนมาถึงยุคที่นำออโตมาตามาดัดแปลงให้เข้าใจง่ายขึ้น มีคำอธิบายประกอบมากขึ้นเรียกว่าแผนภูมิสถานะ (state charts) ในปี ค.ศ. 1987 โดย *David Harel* และต่อมาแผนภูมิสถานะถูกนำมารวมเป็นกับแผนภาพหนึ่งในชุดแผนภาพยูเอ็มแอล ในปี ค.ศ. 1987 นั่นเอง แสดงลำดับความเป็นมาที่กล่าวมาในรูปที่ 5.1

จากเนื้อหาความเป็นมาที่กล่าวมาข้างต้นทำให้เราทราบว่า ออโตมาตาเป็นแผนภาพที่มีการคิดค้นริเริ่มมานาน และได้รับการพัฒนาต่อกันมาเป็นระยะปัจจุบันยังคงมีความสำคัญเพื่อใช้การอธิบายพฤติกรรมของระบบได้เป็นอย่างดี เราสามารถใช้ออโตมาตาในการออกแบบอัลกอริทึม เพื่อให้มีการคำนวณและสร้างเครื่องคอมพิวเตอร์สำหรับการทำงานนั้นได้ เช่นเดียวกับการใช้เครื่องจักร Turing ในการสร้างเครื่องถอดรหัสทางทหารสมัยสงครามโลกครั้งที่สอง

ในการทวนสอบเชิงรูปนัยแบบโมเดลเช็กกิงนั้น ก็มีความจำเป็นต้องสร้างแบบจำลองระบบที่สนใจด้วยปริภูมิสถานะในรูปแบบต้นไม้ของสถานะ (state

tree) หรือกราฟสถานะ (state graph) เพื่อที่เราสามารถนำมาแจกแจงพฤติกรรมของระบบ ด้วยการดูเส้นทางการเปลี่ยนแปลงของสถานะที่เป็นไปได้ทั้งหมด ในทางปฏิบัติการแจกแจงสถานะที่เป็นไปได้ทั้งหมดของระบบ จะทำได้โดยการสังเกตตัวแปรจำนวนหนึ่งที่เราสนใจเท่านั้น โดยคอยสังเกตว่าตัวแปรชุดนั้น ๆ มีค่าที่เปลี่ยนไปอย่างไร ข้อเสียของการใช้แผนภาพสถานะหรือออโตมาตา คือ การทำความเข้าใจระบบและการเริ่มสร้างออโตมาตาในการอธิบายพฤติกรรมระบบทั้งนี้ จำนวนสถานะของระบบขนาดใหญ่จะมีจำนวนมากและเกิดความซับซ้อนกลายเป็นอุปสรรคในการนำมาใช้ในการทวนสอบ



รูปที่ 5.1 การลำดับความเป็นมาของออโตมาตา

การวาดออโตมาตาแบบจำกัด

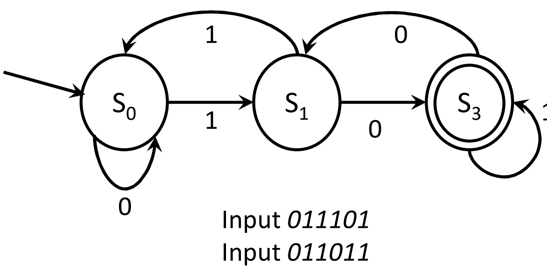
เราวาดออโตมาตาได้ด้วยแผนภาพสถานะ ซึ่งเป็นกราฟระบุทิศทาง (directed graph) ประกอบด้วยสัญลักษณ์โหนด (node) ที่ใช้แทนสถานะ และเส้นเชื่อม (edge) ที่เชื่อมระหว่างโหนดต้นทางไปสู่โหนดปลายทาง โดยเส้นเชื่อมเป็นลูกศรที่มีทิศทาง และกำกับบนเส้นเชื่อมด้วยป้าย (label) ที่เขียนด้วยอักขระที่กำหนดให้ไว้แล้วเท่านั้น มักจะมีโหนดเดียวที่กำหนดให้เป็นสถานะเริ่มต้น (initial state) และมีโหนดอย่างน้อยหนึ่งโหนดที่กำหนดให้เป็นสถานะสิ้นสุด (final state)

ตัวอย่างที่ 5-1: ออโตมาตาอย่างง่าย

กำหนดให้ ออโตมาตาที่ออกแบบเพื่อใช้ตรวจการยอมรับข้อมูลนำเข้าที่เป็นสตริงที่มีความยาวจำกัดของอักขระจากเซต *{0, 1}* โดยให้มีพฤติกรรมยอมรับสตริง *"011101"* และ *"011011"* หรือสตริงอื่นๆ ที่เมื่อปรากฏอักขระ *1* แล้ว

จะต้องมีอักขระ *0* ตามมาแบบไม่ทันทีแต่ให้มีได้ในที่สุด จากนั้นก็เป็นอักขระ 1 ได้ตลอดไป

เราสามารถวาดออโตมาตาที่ต้องการในรูปที่ 5.2 โดยมีเซตสถานะเป็น $\{S_0, S_1, S_2\}$ และมี S_0 เป็นสถานะเริ่มต้น มี S_2 เป็นสถานะสิ้นสุด ระหว่างสถานะจะมีเส้นลูกศรเชื่อมบอกการเปลี่ยนแปลงสถานะจากสถานะปัจจุบันไปสู่สถานะถัดไป และมีอักขระกำกับบนเส้นเชื่อมเป็นข้อมูลนำเข้าที่มีในขณะนั้น เช่น เริ่มต้นที่สถานะ S_0 เมื่อมีข้อมูลนำเข้าเป็น *0* สถานะก็ยังคงเป็นสถานะเดิม แต่ถ้ามีข้อมูลนำเข้าเป็น *1* สถานะจะเปลี่ยนไปเป็นสถานะ S_1 และเมื่ออยู่ในสถานะ S_1 แล้วได้รับข้อมูลนำเข้าเป็น *1* จะเปลี่ยนสถานะกลับไปเป็นสถานะ S_0 ในขณะที่ถ้าได้รับข้อมูลนำเข้าเป็น *0* ก็จะเปลี่ยนสถานะไปเป็นสถานะ S_3 ซึ่งเป็นสถานะสิ้นสุด ออโตมาตาแบบตัวยอมรับจะพิจารณายอมรับสตริงที่มีความยาวจำกัดตัวนั้นได้ก็ต่อเมื่อสามารถจบได้ที่สถานะสิ้นสุดพอดีเท่านั้น



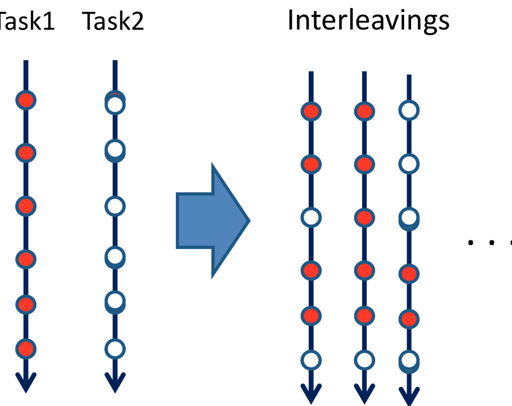
รูปที่ 5.2 การวาดออโตมาตาอย่างง่าย

โดยทั่วไปออโตมาตาถูกนำมาใช้ทำงานให้เป็นตัวยอมรับ (acceptor) หรือ ตัวเปลี่ยนแปร (transducer) โดยการทำงานของตัวยอมรับจะทำหน้าที่รับข้อมูลนำเข้าที่เป็นสตริงของอักขระที่กำหนดไว้ และตอบว่าสตริงนำเข้านั้นยอมรับได้ไหมว่าเป็นไปตามรูปแบบที่ออโตมาตาออกแบบมา เช่น ทำการตรวจว่าสตริงของเลขฐานสองที่นำเข้ามาเป็นเลขคู่เท่านั้นที่ยอมรับ เป็นต้น คำตอบที่ตัวยอมรับให้ได้ คือ ยอมรับหรือไม่ยอมรับเท่านั้น โดยไม่สามารถตอบเป็นอย่างอื่นได้ ถ้าออโตมาตาทำหน้าที่เป็นตัวเปลี่ยนแปรก็จะทำหน้าที่รับข้อมูลนำเข้าเป็นสตริงของอักขระที่กำหนดไว้ และสร้างคำตอบใหม่เป็นผลลัพธ์ออกมาได้ เช่น การออกแบบตัวเปลี่ยนแปรที่ทำหน้าที่เข้ารหัสเลขฐานสองที่นำเข้าให้เป็นรหัสลับ (cipher) เป็นต้น

ออโตมาตาเชิงกำหนดและไม่เชิงกำหนด

ออโตมาตาเชิงกำหนด (deterministic automata) คือ ออโตมาตาที่ทุกสถานะ p เมื่อมีข้อมูลนำเข้า a จะส่งผลให้เปลี่ยนเป็นสถานะ q ได้แบบเดียวเท่านั้น ทำให้ออโตมาตาชนิดนี้มีความชัดเจนและทำงานในรูปแบบที่คาดการณ์ได้เสมอ ส่วนออโตมาตาเชิงไม่กำหนด (nondeterministic automata) คือ ออโตมาตาที่มีสถานะ p เมื่อได้รับข้อนำเข้า a แล้วจะเปลี่ยนไปยังสถานะต่อไปได้มากกว่าหนึ่งสถานะโดยต้องเลือกทางใดทางหนึ่ง และการเลือกนั้นจะเป็นแบบสุ่ม คือไม่เหมือนกันทุกครั้งไป คาดการณ์ไม่ได้ ระบบซอฟต์แวร์หรือฮาร์ดแวร์ทั่วไปจะทำงานแบบออโตมาตาเชิงกำหนด แต่ในกรณีที่ระบบแบบพร้อมกันที่มีหลายสายโยงใย มักจะเกิดปัญหาการแทรกสลับของการทำงาน ทำให้แบบจำลองเชิงรูปนัยระบบต้องอธิบายด้วยออโตมาตาเชิงไม่กำหนด ระบบแบบนี้จะมีความไม่ชัดเจนและคาดการณ์ไม่ได้ การทดสอบระบบจึงทำได้ยากหรือทำไม่ได้ครบตามมาตรฐาน ต้องใช้เทคนิคการทวนสอบเชิงรูปนัย เพื่อให้เราเข้าใจออโตมาตาทั้งสองชนิดได้อย่างลึกซึ้งต้องพิจารณาจากนิยามที่กำหนดไว้ต่อไป

กำหนดให้สองสายโยงใยชื่อ *Task1* และ *Task2* ทำงานเป็นลำดับตามด้านซ้ายของรูปที่ 5.3 ในทางปฏิบัติหน่วยประมวลผลกลางหรือซีพียูจะทำงานตามคำสั่งได้ทีละคำสั่งเท่านั้น โดยซีพียูจะสุ่มเลือกทำคำสั่งของ *Task1* ก่อนหรือคำสั่งของ *Task2* ก่อนก็ได้ ระหว่างทำงานให้กับ *Task1* จะหยุดกลางทางและสลับมาทำงานให้กับ *Task2* บ้างก็ได้ จากนั้นก็สลับกลับไปทำงาน *Task1* ต่อ จะสลับไปมาอย่างไรนั้นเกิดจากการสุ่ม ทำให้เราคาดการณ์ไม่ได้เลย รูปที่ 5.3 ด้านขวาแสดงตัวอย่างการรูปแบบการแทรกสลับของ *Task1* และ *Task2* ที่เป็นไปได้บางส่วน



รูปที่ 5.3 ตัวอย่างการแทรกสลับของสายโยงใย Task1 และ Task2

นิยามออโตมาตาเชิงกำหนดแบบจำกัด

นิยามที่ 5-1: ออโตมาตาเชิงกำหนดแบบจำกัด

ออโตมาตาแบบจำกัด [25] คือ 5-tuple $M=(S,\Sigma, f, I, F)$ โดยที่

S คือ เซตจำกัดของสถานะ

Σ คือ เซตจำกัดของอักขระที่นำเข้า

$f: S \times \Sigma \longrightarrow S$ คือ ฟังก์ชันการส่งผ่าน (transition function) ที่รับคู่ลำดับ (s,i) โดยที่ $s \in S$ และ $i \in \Sigma$ และได้ผลลัพธ์เป็น $v \in S$ โดยที่ s เป็นสถานะปัจจุบัน i เป็นอักขระที่นำเข้า และ v เป็นสถานะถัดไป

I คือ สถานะเริ่มต้น

F คือ เซตของสถานะสิ้นสุด โดยที่ $F \subseteq S$

นิยามออโตมาตาเชิงไม่กำหนดแบบจำกัด

นิยามที่ 5-2: ออโตมาตาเชิงไม่กำหนดแบบจำกัด

ออโตมาตาแบบจำกัด [25] คือ 5-tuple $M=(S,\Sigma, f, I, F)$ โดยที่

S คือ เซตจำกัดของสถานะ

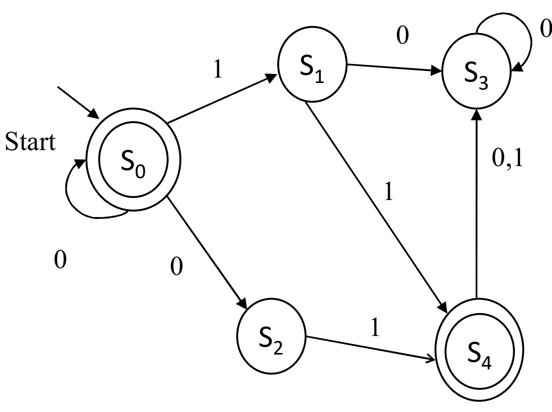
Σ คือ เซตจำกัดของอักขระที่นำเข้า

$f: S \times \Sigma \longrightarrow \mathbb{P}(S)$ คือ ฟังก์ชันการส่งผ่านที่รับคู่ลำดับ (s, i) โดยที่ $s \in S$ และ $i \in \Sigma$ และได้ผลลัพธ์เป็น $v \in \mathbb{P}(S)$ โดยที่ s เป็นสถานะปัจจุบัน i เป็นอักขระที่นำเข้า v เป็นเซตของสถานะถัดไป (โปรดสังเกตผลลัพธ์ของฟังก์ชันการส่งผ่าน f ที่ให้ค่าเป็นเซตของสถานะ)

I คือสถานะเริ่มต้น

F คือเซตของสถานะสิ้นสุด โดยที่ $F \subseteq S$

ตำรานี้จะสนใจเฉพาะออโตมาตาแบบจำกัดเท่านั้น นั่นคือ มีเซต S เป็นเซตจำกัดของสถานะ ส่วนข้อแตกต่างระหว่างออโตมาตาเชิงกำหนดแบบจำกัดและออโตมาตาเชิงไม่กำหนดแบบจำกัดอยู่ตรงที่ฟังก์ชันการส่งผ่าน f ที่รับพารามิเตอร์เป็นสถานะปัจจุบันและข้อมูลนำเข้า ให้ผลลัพธ์เป็นสถานะถัดไปที่เป็นสถานะเดียวหรือหลายสถานะ ถ้าให้ผลลัพธ์เป็นหลายสถานะจะเกิดปัญหาว่าจะเลือกเปลี่ยนไปสถานะใดนั่นเอง จากรูปที่ 5.5 ที่เป็นออโตมาตาเชิงไม่กำหนดโดยมีสถานะ S_0 ที่เมื่อได้รับข้อมูลนำเข้า 1 จะเปลี่ยนไปเป็นสถานะ S_1 แต่ถ้าได้รับข้อมูลนำเข้า 0 จะมีสองทางเลือกคือ อยู่ที่สถานะเดิม หรือ เปลี่ยนไปเป็นสถานะ S_2



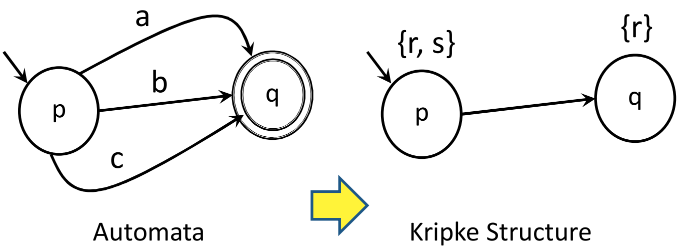
รูปที่ 5.5 ตัวอย่างออโตมาตาเชิงไม่กำหนด

5.4 โครงสร้างคริปกี

โดยทั่วไปเราใช้ออโตมาตาในการอธิบายพฤติกรรมของระบบได้ แต่ในระบบขนาดใหญ่และซับซ้อนอาจจะมีจำนวนสถานะและการเปลี่ยนสถานะที่มากมาย แต่ถ้าเราสนใจเฉพาะสถานะและลำดับการเปลี่ยนสถานะเท่านั้น โดยไม่สนใจข้อมูลนำเข้า เช่น ระบบเมื่อเริ่มทำงานแรกเริ่มแล้วจากนั้นมีโอกาสที่ระบบจะทำงานจบได้ไหม โดยที่เราไม่ต้องการทราบว่าระหว่างระบบทำงานนั้นมีการรับข้อมูลเข้าอย่างไร เราเพียงแต่สนใจเส้นทางลำดับการเปลี่ยนสถานะเท่านั้น เป็นต้น ออโตมาตาจะแสดงทุกเส้นเชื่อมระหว่างสถานะที่กำกับด้วยข้อมูลนำเข้าที่แตกต่างกันเสมอ ทำให้ต้องมีเส้นเชื่อมจำนวนมากเกินกว่าที่เราต้องการ ตัวอย่างเช่น สำหรับสถานะ p, q ใด ๆ ที่มีการเปลี่ยนสถานะจาก p ไปยัง q โดยการรับข้อมูลนำเข้า $a,\ b,\ c$ ออโตมาตาจะวาดแจกแจงเชื่อมที่กำกับด้วยข้อมูลนำเข้า a, b, c ระหว่างสถานะ p และ q จนครบทั้งสามเส้นเชื่อมเสมอ ทั้งที่เราอาจจะไม่สนใจข้อมูลนำเข้าก็ได้ เราแค่สนใจว่าจากสถานะ p จะไปสู่สถานะ q ได้หรือไม่เท่านั้น และต้องการรู้เพียงว่า ณ สถานะ p หรือ q นั้นค่าตัวแปรที่เราเฝ้าสังเกตอยู่มีค่าอะไรเท่านั้น ดังที่กล่าวมาแล้ว โครงสร้างคริปกีจะเป็นทางเลือกแทนออโตมาตาได้ในกรณีนี้

รูปที่ 5.6 แสดงการเขียนโครงสร้างคริปกีจากออโตมาตาเดิม โดยจะมีเส้นเชื่อมระหว่างสถานะ p และสถานะ q แค่เส้นเดียวเท่านั้น และต้องวาดการแจกแจงค่าตัวแปรที่สถานะ p คือ $\{r, s\}$ และค่าตัวแปรที่สถานะ q คือ $\{r\}$ เพิ่มเข้าไป

โดยในออโตมาตาเดิมไม่มีให้ (กรณีโครงสร้างคริปกีที่อธิบายการทำงานของระบบ ค่าตัวแปรที่แจกแจงมักจะเป็นค่าจริงของประพจน์เดี่ยวที่ใช้แทนคุณลักษณะที่สนใจ) เนื่องจากโครงสร้างคริปกีมักใช้อธิบายระบบเชิงปฏิสัมพันธ์ นั่นหมายถึงระบบจะทำงานไปอย่างไม่สิ้นสุด ดังนั้นโครงสร้างคริปกีไม่มีสถานะยอมรับเหมือนกับออโตมาตา



รูปที่ 5.6 การเขียนโครงสร้างคริปกีจากออโตมาตาต้นฉบับ

โครงสร้างคริปกีมักจะใช้ประโยชน์ในการอธิบายตรรกศาสตร์เชิงเวลาด้วยเช่นกัน โดยลักษณะของโครงสร้างคริปกีมองดูคล้ายกับออโตมาตาแต่ก็มีความแตกต่างกัน (ดูรูปที่ 5.6 ประกอบ) ดังนี้

- โครงสร้างคริปกีจะกำกับข้อความที่ตำแหน่งสถานะแทนที่จะเป็นการกำกับที่เส้นเชื่อมเหมือนออโตมาตา
- ข้อความที่กำกับที่ตำแหน่งสถานะของโครงสร้างคริปกีจะเป็นเซตย่อยของค่าตัวแปรของที่เราเฝ้าสังเกตอยู่ คือ เซตย่อยของประพจน์เดี่ยวที่เราต้องการรู้ว่ามีค่าจริง ณ สถานะนั้นหรือไม่ แทนที่จะเป็นข้อมูลนำเข้าที่ออโตมาตากำกับไว้
- โครงสร้างคริปกีไม่มีการกำกับข้อความบนเส้นเชื่อมระหว่างสถานะคู่ใด ทำให้ระหว่างสถานะคู่หนึ่งใด มีเส้นเชื่อมในแต่ละทิศทางนั้นเพียงเส้นเดียวเท่านั้น เป็นการลดจำนวนเส้นเชื่อม ลดความซับซ้อนของแผนภาพทำให้อ่านเข้าใจได้ง่ายขึ้น
- โครงสร้างคริปกีไม่มีการระบุสถานะสิ้นสุดหรือสถานะยอมรับ เหมือนกับออโตมาตา ซึ่งทำให้เข้าใจหรือตีความได้ว่า ทุกสถานะในโครงสร้างคริปกีจะเป็นสถานะสิ้นสุดได้ เนื่องจากระบบที่มักในโครงสร้างคริปกีไปใช้บรรยายก็คือ ระบบที่ทำงานแบบไม่สิ้นสุด

นิยามของโครงสร้างคริปกี

นิยามที่ 5-3: โครงสร้างคริปกี

โครงสร้างคริป [6] คือ *4-tuple* $M=(S, S_0, R, L)$ โดยที่

S คือ เซตจำกัดของสถานะ

S_0 คือ สถานะเริ่มต้น

R คือ ความสัมพันธ์จากเซตของสถานะต้นทางไปยังเซตของสถานถัดไป ซึ่งเขียนด้วย $R \subseteq S \times S$

$L:S \longrightarrow 2^{AP}$ คือ ฟังก์ชันการหาข้อความกำกับ (labelling function) ที่หาว่าสถานะใด $s \in S$ จะมีข้อความกำกับเป็นเซตของประพจน์เดี่ยวที่มีค่าจริง

AP คือเซตของประพจน์เดี่ยวที่เราสนใจทุกประพจน์ที่ไม่ใช่เซตว่าง เช่น $\{p, q, r\}$ เป็นต้น และ 2^{AP} คือเซตกำลังของเซต AP

ตัวอย่าง 5-2 โครงสร้างคริปกีอย่างง่าย

กำหนดให้ เซตประพจน์เดี่ยวที่เราสนใจ $AP=\{p, q\}$ และโครงสร้างคริปกี $M=(S, S_0, R, L)$ ดังนี้

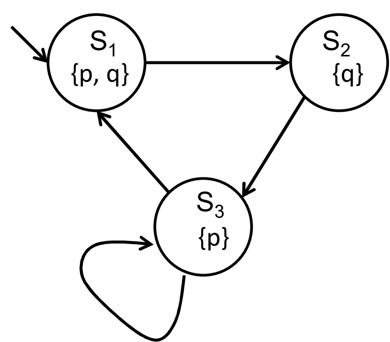
$S = \{s_1,, s_2, s_3\}$

$S_0, = s_1$

$R = \{(s_1, s_2), (s_2,s_3), (s_3,s_1), (s_3,s_3)\}$

$L = \{(s_1, \{p, q\}), (s_2, \{q\}), (s_3, \{p\})\}$

เราสามารถวาดรูปโครงสร้างคริปกีได้ตามรูปที่ 5.7



รูปที่ 5.7 โครงสร้างคริปกีอย่างง่าย

โครงสร้างคริปกีมีลักษณะเป็นกราฟของสถานะที่ไม่สนใจข้อมูลนำเข้า แต่เน้นให้เราสามารถสังเกตตัวแปรของแต่ละสถานะได้ ในการทำงานของระบบซอฟต์แวร์และฮาร์ดแวร์ที่เราอยากทราบว่ามีพฤติกรรมของการเปลี่ยนลำดับของสถานะอย่างไร โดยเราอาจไม่ทราบข้อมูลนำเข้าเนื่องจากเป็นแบบจำลองที่มีความเป็นนามธรรมมากในตอนที่ออกแบบนั้น เราก็สามารถใช้โครงสร้างคริปกีเพื่อมาสร้างเป็นแบบจำลองเชิงรูปนัยได้ และถ้าเป็นการอธิบายพฤติกรรมที่ต้องนำไปทวนสอบกับตรรกศาสตร์เชิงเวลาด้วยแล้วก็ยิ่งเหมาะสม

5.5 บูชีออโตมาตา

บูชีออโตมาตา (Büchi automata) คือ โอเมกาออโตมาตา (omega automata) ชนิดหนึ่ง โดยที่โอเมกาออโตมาตาเป็นออโตมาตาที่สามารถตรวจการยอมรับ ของสตริงที่มีความยาวไม่จำกัดได้ เราสามารถเลือกอธิบายตรรกศาสตร์เชิงเวลาที่มีลักษณะไม่สิ้นสุดแบบพิเศษด้วยบูชีออโตมาตาได้ด้วยเช่นกัน

บูชีออโตมาตากันซึ่งคิดค้นโดยนักตรรกศาสตร์ชาวสวิสเซอร์แลนด์ชื่อ *Julius Richard Büchi* โดยได้กำหนดไว้ว่าบูชีออโตมาตา คือ โอเมกาออโตมาตาที่จะต้องมีอย่างน้อยหนึ่งสถานะสิ้นสุดที่รองรับพฤติกรรมแบบเป็นประจำอย่างนับไม่ถ้วน

ถ้าเรากำลังสนใจระบบที่มีลักษณะพิเศษ คือ ระบบที่มีสถานะที่จำกัด แต่มีพฤติกรรมที่ทำงานไปเรื่อย ๆ แบบไม่สิ้นสุดแล้ว คุณลักษณะที่ต้องการทวนสอบมักจะเป็นคุณลักษณะเชิงเวลาด้วยเช่นกัน โดยเรามักต้องการรู้ว่าระบบจะทำงานในรูปแบบที่เรียกว่า “เป็นประจำอย่างนับไม่ถ้วน” (infinitely often) เขียนได้ด้วยสูตรเชิงเวลา []<>p ซึ่งหมายถึงการที่ประพจน์เดี่ยว p มีค่าจริงในที่สุดและเป็นเช่นนี้แบบเป็นประจำอย่างนับไม่ถ้วน กล่าวคือ ประพจน์ p ที่โมเดล ณ เวลา $t=1$ ยังมีค่าความจริงเป็นค่าเท็จ และเวลาต่อมาประพจน์ p ที่โมเดล ณ เวลา $t=n$ มีค่าความจริงเป็นค่าจริงในที่สุด (นั่นคือ <>p เป็นค่าจริงแล้ว) และต่อมาประพจน์ p ที่โมเดล ณ เวลา $t=n+1$ ประพจน์ p กลับมีค่าความจริงเป็นค่าเท็จอีกครั้ง จนเวลาผ่านไปสักระยะประพจน์ p ที่โมเดล ณ เวลา $t=n+9$ มีค่าความจริงเป็นค่าจริงอีกครั้งในที่สุด ค่าความจริงของประพจน์ p จะมีค่าความจริงเป็นค่าจริงสลับแบบนี้อย่างที่เรียกว่า *“เป็นประจำอย่างนับไม่ถ้วน”* ตลอดการทำงานของระบบที่ไม่สิ้นสุด

กรณีที่กล่าวมานี้ คือ การที่ระบบทำงานอย่างเป็นประจำอย่างนับไม่ถ้วนเราจะอธิบายด้วยโครงสร้างคริปกีได้ลำบาก ทางเลือกก็คือ ให้ใช้บูชีออโตมาตาแทนโครงสร้างคริปกี

นิยามบูชีออโตมาตา

นิยามที่ 5-4: บูชีออโตมาตา

บูชีออโตมาตา [6] คือ ออโตมาตาพิเศษแบบหนึ่ง $BA=(A, S, f, I, F)$ โดยที่

A คือ เซตของสัญลักษณ์ของค่าที่สังเกตได้ โดยแต่ละสัญลักษณ์แสดงถึงเซตย่อยของเซต AP

AP คือ เซตของประพจน์เดี่ยวที่เราสนใจ

S คือ เซตจำกัดของสถานะของระบบ

f คือ ความสัมพันธ์จากสถานะปัจจุบันและสัญลักษณ์ของค่าที่สังเกตได้เชื่อมไปสถานะถัดไปแสดงด้วย

$f \subseteq S \times A \times S$

I คือ สถานะเริ่มต้นมาจากเซตสถานะของระบบแสดงด้วย $I \subseteq S$

F คือ สถานะสิ้นสุดมาจากเซตสถานะของระบบแสดงโดย $F \subseteq S$

และการยอมรับสตริงที่ไม่สิ้นสุดของบูชีออโตมาตามีเงื่อนไข คือ เส้นทางการทำงานของระบบที่ไม่สิ้นสุด ρ ที่ทำให้เกิดสตริงที่จะได้รับการยอมรับ ก็ต่อเมื่อเส้นทาง ρ มีสถานะสิ้นสุดในเซต F อย่างน้อยหนึ่งสถานะที่ปรากฏในส่วนท้ายของเส้นทางที่มีลักษณะเป็น *“ประจำอย่างนับไม่ถ้วน”*

ตัวอย่างที่ 5-3 การเขียนบูชีออโตมาตา

กำหนดให้ สูตรเชิงเวลา $p \cup q$ และให้เขียนบูชีออโตมาตา $BA=(A, S, f, I, F)$ โดยให้มี

$AP = \{p, q, r\}$

$A = \{a, b\}$ โดย $a = \{p, r\}$ และ $b = \{q\}$ เราจะใช้สัญลักษณ์ a แทน $\{p, r\}$ และ b แทน {q}

$S = \{s_1, s_2, s_3\}$

$f = \{(s_1,a,s_2), (s_2,a,s_2), (s_2,b,s_3), (s_3,True,s_3)\}$

ในที่นี้ *True* หมายถึงไม่ว่ากำกับเส้นด้วยค่า a หรือ b ก็ตาม

$I = \{s_1\}$

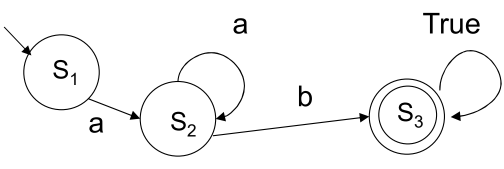
$F = \{s_3\}$

เราสามารถวาดรูปบูชีออโตมาตาประกอบนิยามนี้ได้ดังรูปที่ 5.9 ให้พิจารณาต่อเรื่องการยอมรับสตริงที่ไม่สิ้นสุด ดังนี้

กำหนดให้สตริงที่ไม่สิ้นสุด *strx* คือ $a, a, a, b, b, \ldots$ ซึ่งเกิดจากเส้นทาง $\rho_1 = s_1, s_2, s_2, s_2, s_3, s_3, s_3, s_3, \ldots$

จากนิยาม เราทราบว่าสตริงที่ไม่สิ้นสุด *strx* จะได้รับการยอมรับจากบูชีออโตมาตา *BA* ก็ต่อเมื่อเราพบหนึ่งสถานะสิ้นสุดใน *F* คือ s_3 ที่ปรากฏบน ρ_1 แบบ *"ประจำอย่างนับไม่ถ้วน"* ซึ่งเราพบว่าส่วนท้ายของเส้นทางดังกล่าวเป็น s_3 ไปตลอดนั่นเอง จึงสรุปได้ว่าสตริงไม่สิ้นสุด *strx* นั้นยอมรับโดย *BA*

ดังนั้น บูชีออโตมาตานี้จะแสดงพฤติกรรมของสูตรเชิงเวลา *p*U*q* ได้อธิบายได้ดังนี้ จากค่าของ *strx* = *a, a, a, b, b, …* เราพบว่าเป็นการเกิดค่าของลำดับต่อไปนี้คือ *{p}, {p}, {p}, {q}, {q}, …* และสูตรเชิงเวลา *p*U*q* หมายถึงการที่ค่าความจริงของประพจน์ *p* เป็นค่าจริงตั้งแต่เริ่มต้นไปจนกระทั่งค่าความจริงของประพจน์ *q* เป็นค่าจริงได้นั่นเอง



รูปที่ 5.9 บูชีออโตมาตาอย่างง่าย

ตัวอย่างที่ 5-4 การเขียนบูชิออโตมาตาสำหรับ Op

กำหนดให้สูตรเชิงเวลา O*p* และให้เขียนบูชิออโตมาตา *BA=(A,S, f, I, F)* โดยให้มี

$AP = \{p, q, r\}$

$A = \{a, b\}$ โดย $a = \{p\}$ และ $b = \{p, q, r\}$

$S = \{s_1, s_2, s_3\}$

$f = \{(s_1,True,s_2), (s_2,a,s_3), (s_3,True,s_3)\}$

$I = \{s_1\}$

$F = \{s_3\}$

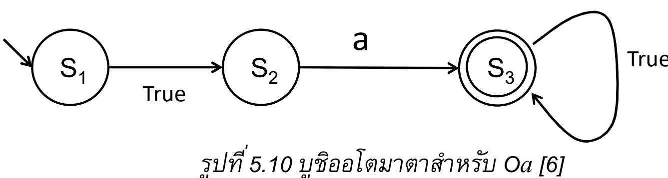
เราสามารถวาดรูปบูชีออโตมาตาประกอบนิยามนี้ได้ดังรูปที่ 5.10 และให้พิจารณาต่อเรื่องการยอมรับสตริงที่ไม่สิ้นสุด ดังนี้

กำหนดให้สตริงที่ไม่สิ้นสุด *strx* คือ *True, a, a, a, a, …* ซึ่งเกิดจากเส้นทาง $\rho_1 = s_1, s_2, s_3, s_3, s_3, s_3, \ldots$

จากนิยาม เราทราบว่าสตริงที่ไม่สิ้นสุด *strx* จะได้รับการยอมรับจากบูชีออโตมาตา *BA* ก็ต่อเมื่อเราพบหนึ่งสถานะสิ้นสุดใน *F* คือ s_3 ที่ปรากฏบน ρ_1 แบบ *"ประจำอย่างนับไม่ถ้วน"* ซึ่งเราพบว่าส่วนท้ายของเส้นทางดังกล่าวเป็น s_3 ไปตลอดนั่นเอง จึงสรุปได้ว่าสตริงไม่สิ้นสุด *strx* นั้นยอมรับโดย *BA*

ดังนั้น บูชีออโตมาตานี้จะแสดงพฤติกรรมของสูตรเชิงเวลา Op ได้อธิบายได้ดังนี้ จากค่าของ *strx = True, a, a, a, b, b, …* เราพบว่าเป็นการเกิดค่าของลำดับต่อไปนี้คือ *True, {p}, {p}, {p}, {p, q, r}, {p, q, r}, …* และสูตรเชิงเวลา O*p* หมายถึงค่าความจริงโมเดลเริ่มแรกเป็นอย่างไรไม่สนใจแต่โมเดลถัดไปประพจน์ *p* ต้องมีค่าจริงเสมอ และจากนั้นต่อไปไม่สนใจแล้ว

ดังนั้น บูชีออโตมาตานี้จะแสดงพฤติกรรมของสูตรเชิงเวลา O*p* ได้



รูปที่ 5.10 บูชิออโตมาตาสำหรับ Oa [6]

ตัวอย่างที่ 5-5 การเขียนบูชีออโตมาตาสำหรับ O(p∨Oq)

กำหนดให้สูตรเชิงเวลา O(p∨Oq) และให้เขียนบูชีออโตมาตา *BA=(A,S, f, I, F)* โดยให้มี

$AP = \{p, q, r\}$

$A = \{a, b\}$ โดย $a = \{p, r\}$ และ $b = \{q\}$

$S = \{s_1, s_2, s_3, s_4\}$

$f = \{(s_1,True,s_2),(s_2,a,s_4),(s_2,True,s_3),(s_3,b,s_4),(s_4,True,s_4)\}$

$I = \{s_1\}$

$F = \{s_4\}$

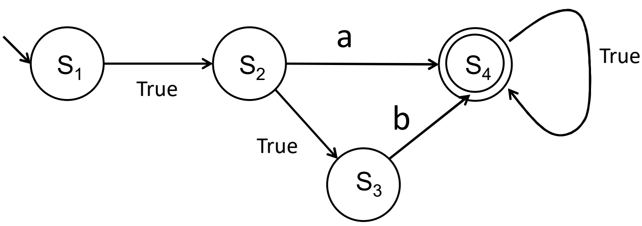
เราสามารถวาดรูปบูชีออโตมาตาประกอบนิยามนี้ได้ดังรูปที่ 5.11 และให้พิจารณาต่อเรื่องการยอมรับสตริงที่ไม่สิ้นสุด ดังนี้

กำหนดให้สตริงที่ไม่สิ้นสุด *strx* คือ *True, a, b, b, b, …* ซึ่งเกิดจากเส้นทาง $\rho_1 = s_1, s_2, s_4, s_4, s_4, s_4, \ldots$ และสตริงที่ไม่สิ้นสุด *stry* คือ *True, True, b, b, …* ซึ่งเกิดจากเส้นทาง $\rho_2 = s_1, s_2, s_3, s_4, s_4, \ldots$

จากนิยาม เราทราบว่าสตริงที่ไม่สิ้นสุด *strx* จะได้รับการยอมรับจากบูชีออโตมาตา *BA* ก็ต่อเมื่อเราพบหนึ่งสถานะสิ้นสุดใน *F* คือ s_4 ที่ปรากฏบน ρ_1 แบบ “ประจำอย่างนับไม่ถ้วน” ซึ่งเราพบว่าส่วนท้ายของเส้นทางดังกล่าวเป็น s_4 ไปตลอดนั่นเอง จึงสรุปได้ว่าสตริงไม่สิ้นสุด *strx* นั้นยอมรับโดย *BA* ในขณะเดียวกันด้วยเหตุผลเดียวกัน สตริงไม่สิ้นสุด *stry* ก็ได้รับการยอมรับด้วยเช่นกัน

ดังนั้น บูชีออโตมาตานี้จะแสดงพฤติกรรมของสูตรเชิงเวลา O(p∨Oq) ได้อธิบายได้ดังนี้ จากค่าของ *strx = True, a, b, b, b, …* เราพบว่าเป็นการเกิดค่า

ของลำดับต่อไปนี้คือ *True {p,r},{q},{q},{q},* ... และสูตรเชิงเวลา O($p \vee$Oq) หมายถึงค่าความจริงโมเดลเริ่มแรกเป็นอย่างไรไม่สนใจแต่โมเดลถัดไปประพจน์ p ต้องมีค่าจริง และจากนั้นต่อไปไม่สนใจแล้ว หรือกรณีที่ค่าของ *stry* คือ *True, True, b, b,* ... หมายถึงค่าความจริงสองโมเดลเริ่มแรกเป็นอย่างไรไม่สนใจแต่โมเดลที่สามประพจน์ q ต้องมีค่าจริง และจากนั้นต่อไปไม่สนใจแล้ว



รูปที่ 5.11 บูชีออโตมาตาสำหรับ O(a∨Ob) [6]

5.6 เพทริเน็ต

เพทริเน็ต [26] คือกราฟที่ใช้เป็นทางเลือกในการอธิบายการทำงานของระบบที่สนใจ โดยแสดงสถานะของระบบและการเปลี่ยนสถานะไปมาระหว่างการทำงาน ซึ่งก็คือพฤติกรรมการทำงานของระบบ สถานะของระบบจะวิเคราะห์หาได้จากรูปกราฟการเข้าถึงได้ของเพทริเน็ต (reachability graph) อีกต่อหนึ่ง เพทริเน็ตถูกคิดค้นขึ้นมาครั้งแรกเมื่อ ปี ค.ศ. 1939 โดย *Carl Adam Petri* เมื่อเขาอายุเพียง 13 ปีเท่านั้น และปัจจุบันเพทริเน็ตก็เป็นแผนภาพที่นิยมนำมาใช้อย่างแพร่หลาย เพื่ออธิบายพฤติกรรมของระบบที่ประกอบด้วยเหตุการณ์ที่แยกจากกัน (discrete events) และมีรูปแบบต่างๆ ที่ชัดเจนคือ เหตุการณ์แบบลำดับ (sequential events) เหตุการณ์แบบพร้อมกัน (concurrent events) เหตุการณ์แบบทางเลือก (choice events) หรือเรียกได้อีกชื่อหนึ่งว่า เหตุการณ์แบบขัดแย้ง (conflict events) เหตุการณ์แบบประสานกัน (synchronized events) และอื่นๆ ทั้งนี้การวาดรูปเพทริเน็ตยังเข้าใจและสื่อสารกันได้ง่าย ในขณะที่สามารถนำมาเขียนเป็นแบบจำลองคณิตตรรกศาสตร์ได้ดีด้วยเช่นกัน

เพทริเน็ตถูกกำหนดให้เป็นกราฟซึ่งมีโหนดสองชนิดที่วางสลับกันเสมอ และมีเส้นเชื่อมระหว่างโหนด โหนดชนิดแรกมีลักษณะเป็นเครื่องหมายวงกลมคือ เพลส (place) และโหนดอีกชนิดมีลักษณะเป็นเครื่องหมายสี่เหลี่ยมผืนผ้าคือ ทรานสิชัน (transition) เส้นเชื่อมจะเป็นเส้นที่เชื่อมระหว่างเพลสไปสู่ทรานสิชัน และเส้นที่เชื่อมระหว่างทรานสิชันไปสู่เพลส เพลสใช้แทนสถานะของระบบและทรานสิชันใช้แทนการเปลี่ยนสถานะ ภายในเพลสจะมีโทเคน (token) ซึ่งแสดงด้วยเครื่องหมายจุดวงกลมสีดำ (black dot) โทเคนจำนวนหนึ่งจะถูกกำหนดให้อยู่ใน

ตำแหน่งเพลสใดก็ได้ตอนเริ่มต้นการทำงานของระบบเราเรียก จำนวนโทเคนในแต่ละตำแหน่งเพลสตอนเริ่มต้นระบบว่ามาร์กกิงแรกเริ่ม หลังจากนั้นโทเคนที่ปรากฏในทุกเพลสก่อนหน้าทรานสิชันใดๆ จะกลายเป็นเงื่อนไขในการเปิดทาง (enable) ให้ทรานสิชันทำการยิง (firing) การยิงจะดำเนินการโดยลบโทเคนที่อยู่ในทุกเพลสก่อนหน้าออก และเพิ่มโทเคนให้กับทุกเพลสถัดไปจากทรานสิชันที่ทำการยิง จำนวนโทเคนที่ลบและเพิ่มในการยิงนั้นจะเท่ากับค่าถ่วงน้ำหนักที่กำกับไว้บนเส้นเชื่อมด้วย ในทางปฏิบัติการยิงของทรานสิชันแสดงถึงเหตุการณ์ได้เกิดขึ้นนั่นเอง และเงื่อนไขในการเปิดทางก็แสดงถึงเงื่อนไขก่อนที่จะเกิดเหตุการณ์หรือเรียกว่า เงื่อนไขก่อน (precondition) ระบบจะดำเนินต่อไปเรื่อยๆ นั่นหมายถึงเพทริเน็ตก็จะเกิดการเปิดทางและการยิงของแต่ละทรานสิชันไปเรื่อยๆ จนจบการทำงาน การจบการทำงานหมายถึงการที่โทเคนไม่สามารถถูกยิงไปต่อเนื่องจากเป็นเพลสสุดท้ายหรือทรานสิชันสุดท้าย ถ้ามีการหยุดนิ่งแต่มีเส้นทางต่อไปได้ เราจะเรียกได้ว่าเกิดการติดตายในระบบ (deadlock)

นิยามเพทริเน็ต

นิยามที่ 5-5: เพทริเน็ต

เพทริเน็ต [26] คือ *6-tuple* $PN = (P, T, F, W, M, M_0)$ โดยที่

P คือเซตจำกัดของเพลส

T คือเซตจำกัดของทรานสิชัน โดยที่ $P \cap T = \varnothing$

F คือเซตจำกัดของเส้นเชื่อมระหว่างเพลสและทรานสิชัน แทนด้วย $F \subseteq (P \times T) \cup (T \times P)$

W คือฟังก์ชันหาค่าถ่วงน้ำหนักของเส้นเชื่อมระหว่างเพลสไปสู่ทรานสิชันหรือเส้นเชื่อมระหว่างทรานสิชันไปสู่เพลส เขียนแทนด้วย $W: F \rightarrow \mathbb{N}$ โดยที่ค่าถ่วงน้ำหนักที่ได้เป็นเลขจำนวนเต็มมากกว่าศูนย์

M คือฟังก์ชันหามาร์กกิงแต่ละเพลสที่มี เขียนแทนด้วย $M: P \rightarrow \mathbb{N}$ ผลลัพธ์ของฟังก์ชัน M จะได้ *Multiset* หรือเวคเตอร์ของจำนวนโทเคน ณ ตำแหน่งของเพลสที่มีเรียงลำดับไว้

M_0 คือมาร์กกิงแรกเริ่มของเพลสที่มี แสดงด้วย $M(P)$ โดยที่ P ณ เวลาที่ระบบเริ่มต้น

มาร์กกิงแรกเริ่มมีความสำคัญอย่างไร เพทริเน็ตที่สมบูรณ์เหมาะสมจะต้องมีการกำหนดมาร์กกิงแรกเริ่ม M_0 มาให้เสมอ ค่ามาร์กกิงแรกเริ่มมี

ความสำคัญมาก โดยเพทริเน็ตที่มีโครงสร้างรูปแบบการเชื่อมต่อแบบเดียวกันแต่มาร์กกิงแรกเริ่มต่างกัน ก็ทำให้มีพฤติกรรมที่แตกต่างกันได้ เหมือนเรากำหนดค่าเริ่มต้นที่ผิดพลาดพฤติกรรมระบบอาจจะมีความผิดพลาดในที่สุดเช่นกัน

ตัวอย่างที่ 5-6 เพทริเน็ตแบบง่าย

กำหนดให้เพทริเน็ต [26] $PN = (P, T, F, W, M, M_0)$ โดยที่

$$P = \{p_1, p_2, p_3, p_4, p_5\}$$

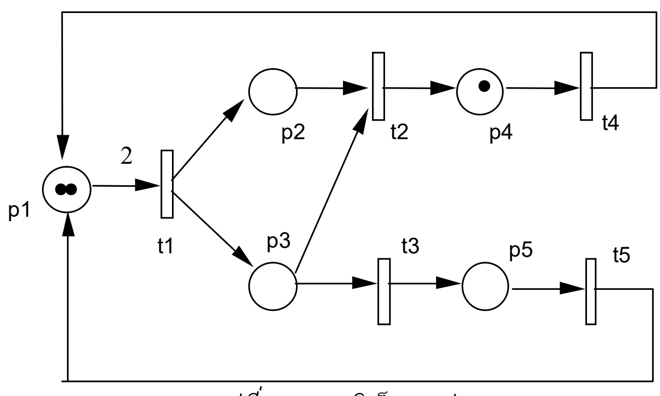
$$T = \{t_1, t_2, t_3, t_4, t_5\}$$

$$F = \{(p_1,t_1), (t_1,p_2), (t_1,p_3), (p_2,t_2), (p_3,t_3), (p_3,t_2), (t_2,p_4), (t_3,p_5), (p_4,t_2), (p_5,t_5), (t_4,p_1), (t_5,p_1)\}$$

$$W = \{((p_1,t_1),2), ((t_1,p_2),1), ((t_1,p_3),1), ((p_2,t_2),1), ((p_3,t_3),1), ((p_3,t_2),1), ((t_2,p_4),1), ((t_3,p_5),1), ((p_4,t_2),1), ((p_5,t_5),1), ((t_4,p_1),1), ((t_5,p_1),1)\}$$

$$M_0 = (2,0,0,1,0)$$

รูปเพทริเน็ตในตัวอย่างวาดได้ตามในรูปที่ 5.12 โดยจะพบว่ามาร์กกิงแรกเริ่มจะมีจำนวนโทเคนเป็น 2 ที่ตำแหน่งเพลส p_1 และมีจำนวนโทเคนเป็น 1 ที่ตำแหน่งเพลส p_4 ที่เหลือจะมีจำนวนโทเคนเป็น 0 ดังนั้น มาร์กกิงแรกเริ่ม $M_0 = (2,0,0,1,0)$ นั่นเอง

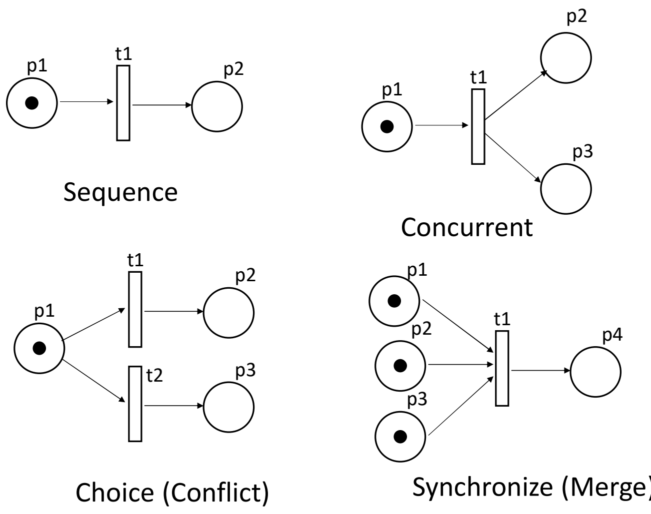


รูปที่ 5.12 เพทริเน็ตแบบง่าย [26]

การวาดรูปเพทริเน็ตเพื่อให้ง่ายและไม่ซับซ้อนเราสามารถละการกำกับตัวเลขค่าถ่วงน้ำหนักที่อยู่บนเส้นเชื่อมระหว่างเพลสและทรานสิชันได้ถ้าค่าถ่วงน้ำหนักมีค่าเป็น 1 กรณีที่ค่าถ่วงน้ำหนักมีค่ามากกว่า 1 คือเส้นเชื่อม (p_1,t_1) นั้นจะต้องกำกับตัวเลข 2 ไว้ซึ่งจะทำให้เมื่อ t_1 เปิดทางและเกิดการยิงแล้ว จะต้องทำ

การลบโทเคนที่ p_1 ออกเท่ากับค่าถ่วงน้ำหนักคือ 2 โทเคน ในที่นี้ทรานสิชัน t_1 จะมีเงื่อนไขการเปิดทางเป็นจริงพอดีเนื่องจากมีจำนวนโทเคนที่เพลสก่อนหน้า p_1 มากกว่าหรือเท่ากับค่าถ่วงน้ำหนัก

แบบรูปการทำงานหลักของระบบที่เพทริเน็ตแสดงได้คือ ทำงานแบบลำดับ ทำงานแบบพร้อมกัน ทำงานแบบทางเลือกซึ่งเกิดความขัดแย้ง ทำงานแบบวนซ้ำ ทำงานแบบประสาน ดังแสดงในรูปที่ 5.13



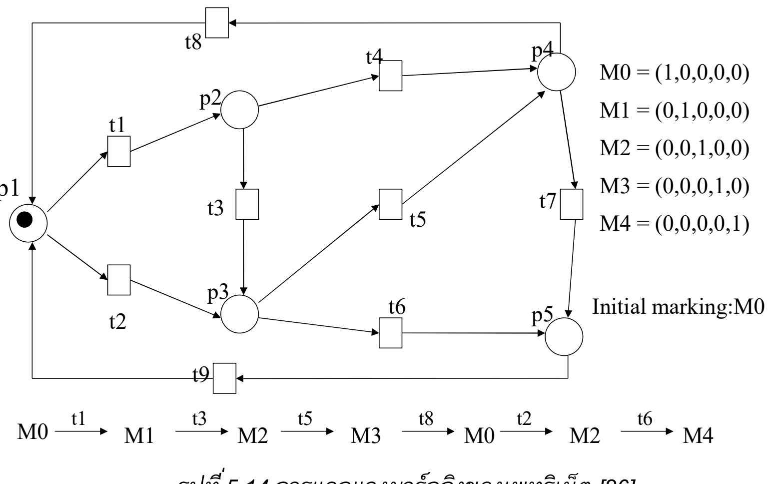
รูปที่ 5.13 แบบรูปการทำงานของเพทริเน็ต [26]

กราฟการเข้าถึงของเพทริเน็ต

รูปกราฟการเข้าถึงของเพทริเน็ตมีความสำคัญและมีประโยชน์อย่างไร เนื่องจากเพทริเน็ตแสดงพฤติกรรมหรือเหตุการณ์ที่จะเกิดในระบบที่เราสนใจ เราจึงต้องการทราบว่าที่เราออกแบบเพทริเน็ตมาแล้วนั้นมันจะมีพฤติกรรมตามที่เรากำหนดไว้หรือไม่ เราหาคำตอบแรกที่ง่ายที่สุดได้โดยการหาว่ามีเพลสใดบ้างที่ไม่สามารถเข้าถึงได้ โดยการหารูปกราฟการเข้าถึงของเพทริเน็ตนั่นเอง เราวิเคราะห์เบื้องต้นได้ทันทีว่าเพลสใดที่โทเคนไม่ผ่านไปเลยได้

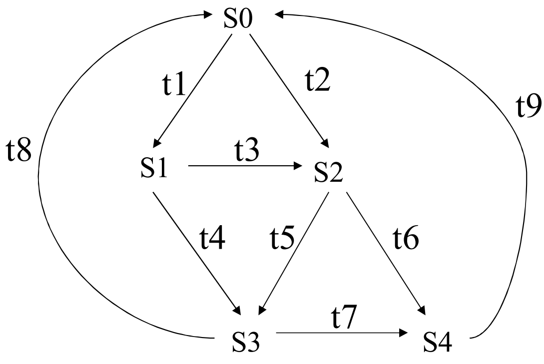
การหากราฟการเข้าถึงนั้น จะเริ่มด้วยการหามาร์กกิงที่เป็นไปได้ทั้งหมดเสียก่อน โดยความสำคัญอยู่ที่มาร์กกิงแรกเริ่มที่จะเป็นตัวกำหนดมาร์กกิงตัวแรก จากนั้นเราจะจำลองเหตุการณ์ว่ามีการเปิดทางและมีการยิงของทรานสิชันแต่ละตัวอย่างไร และติดตามการยิงของทรานสิชันที่เกิดขึ้นทุกกรณี การยิงของทรานสิชันนั่นเองที่ทำให้มาร์กกิงเปลี่ยนไป เราก็จดบันทึกมาร์กกิงใหม่ที่พบให้ครบทุกการเกิดของทรานสิชันที่เป็นไปได้ จากรูปที่ 5.14 แสดงถึงค่ามาร์กกิงที่หาได้ทั้งหมดจากเพทริเน็ตที่มีมาร์กกิงแรกเริ่ม M_0 ตามรูป โดยกำหนดให้เป็น M_0,

M_1, M_2, M_3, และ M_4 ค่ามาร์กกิง M_0 คือ $(1,0,0,0,0)$ เนื่องจากมีเพลส p_1, p_2, p_3, p_4, p_5 เรานับจำนวนโทเคนตามจังหวะการยิงของทรานสิชันแล้วแจกแจงตามลำดับนั่นเอง โดยค่ามาร์กกิง M_0 แสดงตัวเลขแบบเวกเตอร์ตัวแรกในชุดเป็นจำนวนโทเคนของเพลส p_1 ตัวเลขตัวที่สองในชุดเป็นจำนวนโทเคนของเพลส p_2 เป็นอย่างแบบนี้สำหรับเพลสที่เหลือคือ p_3, p_4, p_5 ตามลำดับนั่นเอง



รูปที่ 5.14 การแจกแจงมาร์กกิงของเพทริเน็ต [26]

จากชุดมาร์กกิงที่เป็นไปได้ทั้งหมดเราจะนำมาเขียนเป็นรูปกราฟการเข้าถึง (reachability graph) ของเพทริเน็ต ให้เปรียบเทียบแต่ละมาร์กกิงที่มีเป็นสถานะของระบบ ดังนั้นสถานะแรกเริ่ม S_0 ก็คือมาร์กกิงแรกเริ่มนั่นเอง M_0 เมื่อเราพิจารณาต่อว่าจาก M_0 จะมีทรานสิชันใดบ้างที่ถูกเปิดทางและพร้อมจะยิง ซึ่งจากในรูปตัวอย่างก็คือทรานสิชัน t_1 หรือ t_2 เข้าข่ายกรณีแบบทางเลือกต้องเลือกยิง t_1 หรือไม่ก็ t_2 ทรานสิชันเดียวเท่านั้น เราจะเลือกทรานสิชันที่พร้อมจะยิงมาพิจารณาทีละทรานสิชันเพื่อหามาร์กกิงถัดไป และลากเส้นเชื่อมแสดงการเปลี่ยนสถานะจากมาร์กกิง M_0 ไปสู่สถานะใหม่ ในที่นี้ถ้าเราเลือกทรานสิชัน t_1 มาร์กกิงถัดไปก็จะเป็นเป็น M_1 เรียกใหม่ว่าสถานะ S_1 แต่ถ้าเราเลือกยิงทรานสิชัน t_2 มาร์กกิงถัดไปก็จะเป็น M_2 เรียกใหม่ว่าสถานะ S_2 แทน กราฟการเข้าถึงที่วาดจะเริ่มด้วยสถานะ S_0 และมีเส้นเชื่อมจากสถานะถัดไปสองสถานะที่เป็นไปได้คือ S_1 และ S_2 ตามที่วิเคราะห์มานั่นเอง หลังจากนี้ให้ทยอยพิจารณากรณีถ้าสถานะ S_1 แล้วจะมีสถานะอะไรต่อไปแล้ววาดรูปต่อไปเรื่อยๆ ทำแบบนี้ไปจนครบจะได้รูปกราฟการเข้าถึงของเพทริเน็ตตัวอย่างตามรูปที่ 5.15



รูปที่ 5.15 กราฟการเข้าถึงของเพทริเน็ต [26]

เพทริเน็ตอาจจะได้รับการกำหนดเงื่อนไขเรื่องความมีขอบเขต (Boundedness) ไว้ กล่าวคือเราเรียกเพทริเน็ต *PN* ว่ามีลักษณะ *k-bounded* ก็ต่อเมื่อจำนวนโทเคนในแต่ละเพลสใดๆ ที่มีอยู่นั้นไม่เกินค่า k ตลอดการทำงานของเพทริเน็ต โดยทั่วไปเรามักจะพบเพทริเน็ตที่มีลักษณะ *1-bounded* หมายถึงเพลสใดๆ ของเพทริเน็ตจะมีโทเคนได้ไม่เกิน *1* ตัวเสมอ ทรานสิชันจะไม่ทำการยิงถ้าเงื่อนไขความมีขอบเขตได้กำหนดไว้

5.7 แบบฝึกหัด

1) การสร้างแบบจำลองเชิงรูปนัยด้วยแผนภาพมีข้อดีอย่างไร?
2) ออโตมาตาทั่วไปต่างกับบูชีออโตมาตาอย่างไร?
3) ออโตมาตาทั่วไปต่างกับโครงสร้างคริปกีอย่างไร?
4) ออโตมาตาเชิงไม่กำหนดเกิดขึ้นในการสร้างแบบจำลองเชิงรูปนัยได้อย่างไร อธิบายเหตุผล?
5) การแทรกสลับของการทำงานของระบบเกิดขึ้นได้อย่างไร?
6) จงวาดบูชีออโตมาตาของสูตรเชิงเวลา $[]p \wedge []q$
7) เราจะสร้างปริภูมิสถานะได้จากเพทริเน็ตอย่างไร?
8) มาร์กกิงเริ่มต้นสำคัญอย่างไรในเพทริเน็ต ไม่กำหนดไว้ได้หรือไม่?
9) เพทริเน็ตและออโตมาตา ใช้งานต่างกันอย่างไร?
10) เราใช้เพทริเน็ตอธิบายการทำงานของหลายสายโยงใยที่ทำงานประสานกันได้หรือไม่?
11) ลักษณะการทำงานแบบ $[]<>p$ เรียกว่าอะไร มีความสำคัญอย่างไรกับระบบที่ทำงานแบบไม่สิ้นสุด?

6

การทวนสอบด้วยการพิสูจน์ทฤษฎีบท

6.1 ความสำคัญของบทนี้

บทนี้จะกล่าวถึงการทวนสอบด้วยการพิสูจน์ทฤษฎีบท หมายถึงการใช้แบบจำลองเชิงรูปนัยของระบบเป็นข้อตั้งแรกเริ่ม และทำการพิสูจน์พันธกรณีการพิสูจน์เป็นข้อยุติด้วยการใช้วิธีพิสูจน์แบบนิรนัย ในบทนี้ขอใช้แบบจำลองเชิงรูปนัยที่เขียนขึ้นด้วยภาษาเซด ดังที่ได้กล่าวมาในบทก่อนหน้า และใช้เครื่องมือทวนสอบเซดอีฟพิสูจน์คุณลักษณะที่ต้องการ โดยทั่วไป คุณลักษณะที่เราต้องการทวนสอบหรือพิสูจน์ คือ ความต้องกันขององค์ประกอบของแบบจำลอง การเข้าถึงได้ ความคงอยู่ของเงื่อนไขค่ายืนยงของข้อมูลตลอดการทำงานของระบบ และกรณีอื่นที่เขียนขึ้นเป็นพันธกรณีการพิสูจน์เพิ่มเติมได้ เนื่องจากต้องใช้ภาษาเซด จึงจะมีการอ้างถึงองค์ประกอบและชุดคำสั่งในภาษาเซดที่มีชื่อเรียกเฉพาะที่ผู้เขียนตำราขอใช้เป็นภาษาอังกฤษ เพื่อให้สะดวกต่อการอ้างอิงคู่มือใช้งานภาษาเซด

6.2 วัตถุประสงค์

- เพื่อให้รู้จักและเข้าใจวิธีการทวนสอบด้วยการพิสูจน์ทฤษฎีบท
- เพื่อให้เข้าใจการวางแผนการทวนสอบอย่างเหมาะสม
- เพื่อนำเสนอตัวอย่างการทวนสอบด้วยเครื่องมือเซดอีฟ

6.3 สร้างแบบจำลองระบบด้วยภาษาเซด

ต่อไปนี้เป็นการสร้างแบบจำลองระบบของกรณีศึกษาเรื่องระบบจัดการข้อมูลวันเกิด [20] เป็นตัวอย่างข้อกำหนดเชิงรูปนัยของระบบ เนื่องจากเป็นระบบที่ทำความเข้าใจได้ง่าย การเขียนข้อกำหนดเริ่มจากการกำหนดกีเวนเซตที่เป็นเซตของชื่อคนที่เรารู้จักหรือสนใจ และเซตของวันที่ซึ่งจะใช้อ้างเป็นวันเกิด ระบบจะเริ่มจากการมี *State Schema* หลักที่ใช้ชื่อว่า *BirthdayBook* ซึ่งจะจัดเก็บวันเกิดที่เป็นเซตของคู่ลำดับหรือฟังก์ชันจากโดเมนของเซตชื่อคนที่รู้จักไปยังเรนจ์ของเซตวันที่ซึ่งเป็นวันเกิด ระบบจัดการข้อมูลวันเกิดจะมีขีดความสามารถในการเพิ่มชื่อและวันเกิดของคนที่เรารู้จักเข้าสู่ระบบ และสามารถค้นหาวันเกิดโดยใช้ชื่อเป็นคำสำคัญในการค้นหา ตลอดจนการลบข้อมูลวันเกิด เป็นต้น

ข้อกำหนดเชิงรูปนัยที่ได้

[NAME, DATE]

BirthdayBook

$known : \mathbb{P}\ NAME$

$birthday : NAME \rightarrowtail DATE$

$known = \mathrm{dom}\ birthday$

AddBirthday

$\Delta BirthdayBook$

$name? : NAME$

$date? : DATE$

$name? \notin known$

$birthday' = birthday \cup \{name? \mapsto date?\}$

FindBirthday

$\Xi BirthdayBook$

$name? : NAME$

$date! : DATE$

$name? \in known$

$date! = birthday(name?)$

Remind

$\Xi BirthdayBook$

$today? : DATE$

$card! : \mathbb{P}\ NAME$

$card! = \{n : known \mid birthday(n) = today?\}$

InitBirthday

BirthdayBook

$known = \varnothing$

REPORT ::= ok | already_known | not_known

Success

result!: REPORT

---

result! = ok

AlreadyKnown

ΞBirthdayBook

name?: NAME

result!: REPORT

$name? \in known$

result! = ok

$RAddBirthday \mathrel{\widehat{=}} (AddBirthday \wedge Success) \vee AlreadyKnown$

NotKnown

ΞBirthdayBook

name?: NAME

result!: REPORT

$name? \notin known$

result! = not_known

$RFindBirthday \mathrel{\widehat{=}} (FindBirthday \wedge Success) \vee NotKnown$

$RRemind \mathrel{\widehat{=}} Remind \wedge Success$

วิเคราะห์ข้อกำหนดที่ได้

ต่อไปนี้เป็นการวิเคราะห์ข้อกำหนดที่เขียนไว้ข้างต้น โดยจะอธิบายเพื่อทำความเข้าใจความหมายของแต่ละประโยค และยกตัวอย่างผลลัพธ์ที่เกิดขึ้นในแต่ละขั้นตอน

การกำหนดกีเวนเซต

ในส่วนต้นของข้อกำหนดจะต้องมีการกำหนดกีเวนเซตที่จำเป็น และคาดว่าจะถูกอ้างถึงไว้ก่อนเสมอ โดยในระบบการจัดการข้อมูลวันเกิดนี้จะมีกีเวนเซตอยู่สองเซต คือ กีเวนเซตชื่อ *NAME* และชื่อ *DATE* ดังนี้

[NAME, DATE]

กีเวนเซตชื่อ *NAME* เป็นเซตของทุกคนที่เป็นไปได้ในระบบนี้ และ *DATE* เป็นเซตของทุกวันที่ที่เป็นไปได้ในระบบนี้

การกำหนด State Schema หลัก

กำหนดให้มี *State Schema* หลักซึ่งใช้อธิบายสมุดที่เก็บวันเกิดของคนที่รู้จัก โดยตั้งชื่อ *Schema* ว่า *BirthdayBook* ภายในสมุดเล่มนี้จะเก็บเฉพาะวันเกิดของคนที่รู้จักเท่านั้น ไม่เก็บวันเกิดของคนทุกคน

```
── BirthdayBook ──────────────
│ known: ℙNAME
│ birthday: NAME ⇸ DATE
├──────────────────────────────
│ known = dom birthday
└──────────────────────────────
```

ข้อกำหนดใน *Schema* หลักดังกล่าวได้ประกาศตัวแปรชื่อ *known* ไว้เป็นเซตของชื่อคนที่รู้จักเท่านั้นด้วยภาษาเซต ดังนี้

$known: \mathbb{P}NAME$ มีความหมายว่า $known \in \mathbb{P}NAME$

หมายถึงตัวแปรชื่อ *known* เป็นเซตย่อยของเซตกำลัง $\mathbb{P}NAME$ ซึ่ง *known* จะมีสมาชิกเพิ่มขึ้นหรือลดลงก็ได้ แล้วแต่เราจะเพิ่มคนที่เรารู้จักเข้าไป หรืออาจจะเป็นกรณีที่ *known* มีค่าเท่ากับเซตว่างก็ได้

ในระบบนี้จะมีการประกาศตัวแปรชื่อ *birthday* ซึ่งใช้อ้างถึงกลุ่มของคู่ลำดับ (x, y) โดย x เป็นชื่อคนที่รู้จัก และ y เป็นชื่อของวันที่ของวันเกิดของ x เสมอ อย่างไรก็ตาม จากประโยคประกาศดังต่อไปนี้

$birthday: NAME \nrightarrow DATE$

สังเกตได้ว่า ไม่มีการระบุเจาะจงว่า *birthday* จะเก็บคู่ลำดับเฉพาะคนที่รู้จักเท่านั้นในประโยคดังกล่าว แต่จะกำหนดเป็นเงื่อนไขบังคับเจาะจงอีกครั้งด้วยประโยคต่อไปนี้

$known = dom\ birthday$

โดยประโยคนี้จะอยู่ในส่วนล่างของ *Schema* ประโยคดังกล่าวเป็นการบังคับให้เซตชื่อ *known* ที่เป็นกลุ่มของคนที่รู้จัก มีค่าเท่ากับเซตโดเมนของเซตของคู่ลำดับ *birthday* เสมอ เราเรียกประโยคนี้ว่าเป็นประโยคที่แสดงตัวยืนยงข้อมูล ดังที่เราทราบกันมาแล้วว่าประโยค *Invariant* จะกำหนดเงื่อนไขที่เป็นจริงตลอดเวลาที่ระบบนี้ทำงาน [18]

การกำหนด Operation Schema สำหรับการเพิ่มข้อมูลวันเกิด

เรามีการกำหนดให้ระบบมีขีดความสามารถในการเพิ่มข้อมูลวันเกิดได้ในสมุดวันเกิด โดยมีการเขียนไว้ดังนี้

```
┌─ AddBirthday ──────────────────
│ ΔBirthdayBook
│ name?: NAME
│ date?: DATE
├────────────────────────────────
│ name? ∉ known
│ birthday' = birthday ∪ {name? ↦ date?}
└────────────────────────────────
```

ในส่วนการประกาศจะเริ่มจากการทำ *Schema Inclusion* โดยทำการรวม *Schema* ชื่อ *BirthdayBook* เข้ามาแบบ Delta กล่าวคือตัวแปรใน BirthdayBook จะมีการเปลี่ยนแปลงค่าถัดไปได้ ทั้งนี้เราคาดการณ์ได้ว่า *known'* และ *birthday'* จะมีค่าเปลี่ยนไปเนื่องจากมีการเพิ่มสมาชิกใหม่เข้ามา

ประโยคแรกคือ

ΔBirthdayBook

จะมีความหมายของประโยคนี้ที่เราทราบมาก็คือ

ΔBirthdayBook

known: $\mathbb{P}$NAME

birthday: NAME $\rightarrow\!\!\!\!\!+$ DATE

known': $\mathbb{P}$NAME

birthday': NAME $\rightarrow\!\!\!\!\!+$ DATE

---

known $=$ dom birthday

known' $=$ dom birthday'

ซึ่งเมื่อนำไปทำ *Schema Inclusion* แล้วจะได้ดังนี้

AddBirthday

known: $\mathbb{P}$NAME

birthday: NAME $\rightarrow\!\!\!\!\!+$ DATE

known': $\mathbb{P}$NAME

birthday': NAME $\rightarrow\!\!\!\!\!+$ DATE

name?: NAME

date?: DATE

---

known $=$ dom birthday

known' $=$ dom birthday'

name? $\notin$ known

birthday' $=$ birthday $\cup$ {name? $\mapsto$ date?}

การเพิ่มข้อมูลวันเกิดจะต้องมีการนำข้อมูลเข้า ระบบนี้จะมีการป้อนข้อมูลผ่านเข้ามาทางตัวแปรนำเข้าชื่อ *name?* และ *date?* โดยมีชนิดตัวแปรเป็น *NAME* และ *DATE* ตามลำดับ ดังประกาศไว้ดังนี้

name?: *NAME*
date?: *DATE*

ในการป้อนข้อมูลที่เป็นชื่อผ่านมาทางตัวแปรนำเข้าชื่อ *name?* จะไม่ระบุเจาะจงว่ามีเงื่อนไขบังคับอะไร ทั้งนี้จะต้องทำการตรวจสอบเงื่อนไขก่อนอีกครั้งดังระบุในประโยคข้างล่างนี้ว่า

$name? \notin known$

ประโยคดังกล่าวเป็นการตรวจสอบว่า ค่าที่ป้อนเข้ามาในตัวแปร *name?* จะต้องไม่เป็นสมาชิกของเซต *known* มาก่อนเลย กล่าวคือ ต้องไม่ซ้ำกับที่เคยเก็บไว้แล้วเสมอ ดังนั้นระบบนี้จะไม่สามารถเก็บข้อมูลวันเกิดของคนที่รู้จักที่มีชื่อซ้ำกันได้ ประโยคตรวจสอบก่อนนี้เราเรียกว่าเป็นประโยคเงื่อนไขก่อน

เมื่อตัวแปรนำเข้า *name?* และ *date?* ได้รับการป้อนเข้ามาแล้วและมีการตรวจสอบเงื่อนไขก่อนครบถ้วนแล้ว ก็มาถึงการทำงานขั้นต่อไปคือ การทำการดำเนินการที่เป็นการเปลี่ยนค่าของเซต *birthday'* (next state ของ birthday)

$birthday' = birthday \cup \{name? \mapsto date?\}$

เซต *birthday'* จะได้รับการเปลี่ยนค่าจากเดิม *birthday* โดยทำการ union เซตของคู่ลำดับใหม่ คือ *(name?, date?)* เข้าไปด้วย โดยจะมีความหมายดังนี้

$birthday' = birthday \cup \{ (name?, date?) \}$

การกำหนด Operation Schema สำหรับการค้นหาข้อมูลวันเกิด

ข้อกำหนดสำหรับการค้นหาข้อมูลวันเกิดจะอ้างถึงสมุดวันเกิด และมีตัวแปรนำเข้าชื่อ *name?* เพื่อรับการป้อนข้อมูลชื่อคนที่รู้จักที่จะใช้ค้นหาในสมุดวันเกิด จากนั้นวันเกิดที่ค้นพบจะถูกนำมาแสดงผลทางตัวแปรแสดงผลชื่อ *date!* ข้อกำหนดแสดงดังต่อไปนี้

```
── FindBirthday ──────────────
│ ΞBirthdayBook
│ name?: NAME
│ date!: DATE
├──────────────────────────────
│ name? ∈ known
│ date! = birthday(name?)
└──────────────────────────────
```

ส่วนประกาศแรกของ *Operation Schema* ส่วนมากจะอ้างถึง *State Schema* ที่อ้างอิงเสมอ ในการนี้จะอ้างถึงการทำ *Schema Inclusion* แบบที่ไม่มีการเปลี่ยนแปลงค่าหลังทำงาน ดังนี้

ΞBirthdayBook

โดยทางปฏิบัติประโยคประกาศดังกล่าวจะมีค่าเป็น

```
┌── ΞBirthdayBook ──────────────
│ known: ℙNAME
│ birthday: NAME ⇸ DATE
│ known': ℙNAME
│ birthday': NAME ⇸ DATE
├──────────────────────
│ known = dom birthday
│ known' = dom birthday'
│ known' = known
│ birthday' = birthday
└──────────────────────
```

เมื่อทำ Schema Inclusion จะได้ผลลัพธ์ดังต่อไปนี้

```
┌── FindBirthday ──────────────
│ known: ℙNAME
│ birthday: NAME ⇸ DATE
│ known': ℙNAME
│ birthday': NAME ⇸ DATE
│ name?: NAME
│ date!: DATE
├──────────────────────────
│ known = dom birthday
│ known' = dom birthday'
│ known' = known
│ birthday' = birthday
│ name? ∈ known
│ date! = birthday(name?)
└──────────────────────────
```

กล่าวคือมีการแทนค่า ΞBirthdayBook ที่ประกาศในประโยคแรกนั้นจะได้ข้อกำหนดที่ครบถ้วน และจะสังเกตได้ว่าสถานะถัดไปของตัวแปรที่ประกาศไว้

ทุกตัวจะไม่เปลี่ยนแปลง ทำให้ทราบหลังจากการดำเนินการนี้ทำงานแล้วเสร็จ *BirthdayBook* จะไม่มีการเปลี่ยนแปลงค่าจะเดิมก่อนทำการดำเนินการ โดย *known* มีค่าสมาชิกชื่อคนที่รู้จักกี่คน *known'* จะมีค่าสมาชิกเท่าเดิม และ *birthday* มีค่าสมาชิกของคู่ลำดับวันเกิดเท่าไร *birthday'* ก็จะมีค่าสมาชิกเท่าเดิม

ในส่วนประกาศจะประกาศตัวแปรรับข้อมูลนำเข้าชื่อ *name?* ซึ่งคาดได้ว่าการค้นหาข้อมูลวันเกิดจะเริ่มจากการรับชื่อนำเข้าก่อน

name?: *NAME*

การแสดงผลจะแสดงผลผ่านตัวแปรแสดงผลที่ชื่อ *date!* ซึ่งเราทราบได้ว่าการแสดงผลนั้นจะแสดงเป็นวันที่ผ่านทางตัวแปรนี้เท่านั้น

date!: *DATE*

ในข้อกำหนดส่วนที่สอง เราจะต้องหาประโยคที่เป็นเงื่อนไขก่อน เราจะเห็นได้ว่าก่อนทำงานจะต้องให้แน่ใจว่า ชื่อที่ป้อนเข้ามาจะต้องมีค่าเหมือนกับสมาชิกหนึ่งใดในเซต *known* เสมอก่อน

$name? \in known$

ถ้าเงื่อนไขก่อนไม่เป็นจริงการดำเนินการนี้จะทำงานไม่ได้เสมอ ในข้อกำหนดนี้จะมีค่าตัวยืนยงเป็นประโยคดังต่อไปนี้

$known = dom\ birthday$

$known' = dom\ birthday'$

ซึ่งมีความหมายว่า *known* จะต้องมีค่าเท่ากับเซตโดเมนของเซตคู่ลำดับ *birthday* เสมอทั้งก่อน และหลังการทำการดำเนินการ

และประโยคต่อไปนี้เป็นประโยคที่เป็นเงื่อนไขหลัง

$known' = known$

$birthday' = birthday$

$date! = birthday(name?)$

เซต *known* และเซต *birthday* จะไม่มีการเปลี่ยนแปลงค่าใดๆ และ *date!* จะนำเสนอผลลัพธ์ของการค้นหาวันเกิด เราสามารถหาค่าวันเกิดของตัวแปร *name?* ได้โดยการใช้ฟังก์ชัน *birthday(name?)*

เมื่อกล่าวถึงฟังก์ชัน *birthday* ที่กำหนดไว้ใน *Schema* ดังนี้

$birthday: NAME \rightarrowtail DATE$

เราสามารถใช้งานฟังก์ชันใดๆได้โดยการเขียนตามรูปแบบดังนี้

กำหนดให้ $FunctionName: DomainSet \rightarrowtail RangeSet$

$x?: DomainSet$

$y!: RangeSet$

เราเขียนการใช้ฟังก์ชันได้เป็น

FunctionName(x?) จะให้ค่าผลลัพธ์เป็น *y!*

การกำหนด Operation Schema ของการเตือนเพื่อแจ้งข้อมูลวันเกิด

ข้อกำหนดของการเตือนเพื่อแจ้งข้อมูลวันเกิด โดยใช้วันที่ปัจจุบันป้อนเข้าไปเก็บในตัวแปรนำเข้าชื่อ today? ซึ่งเมื่อนำไปค้นหารายชื่อของคนที่รู้จักได้แล้วจะนำไปแสดงผลในตัวแปรแสดงผลชื่อ card! ได้

```
── Remind ──────────────────────────
│ ΞBirthdayBook
│ today?: DATE
│ card!: ℙNAME
├───────────────────────────────────
│ card! = {n: known | birthday(n) = today?}
└───────────────────────────────────
```

ข้อกำหนดเริ่มจากการทำ *Schema Inclusion* เช่นเดิมเพื่อให้อ้างถึงสมุดวันเกิดที่กำหนดไว้ก่อนหน้านี้แล้ว โดยการแทนค่า ΞBirthdayBook แล้วจะได้ผลลัพธ์ดังต่อไปนี้คือ

```
── Remind ──────────────────────────
│ known: ℙNAME
│ birthday: NAME ⇸ DATE
│ known': ℙNAME
│ birthday': NAME ⇸ DATE
│ today?: DATE
│ card!: ℙNAME
├───────────────────────────────────
│ known = dom birthday
│ known' = dom birthday'
│ known' = known
│ birthday' = birthday
│ card! = {n: known | birthday(n) = today?}
└───────────────────────────────────
```

จะเห็นได้ว่าเมื่อแทนค่าการทำ *Inclusion* ไปแล้วจะเห็นว่าเราจะพบการประกาศตัวแปรทั้งชื่อ *known* และ *birthday* ที่ทราบได้ว่าหลังการทำงานแล้ว ค่าสถานะถัดไปยังคงเหมือนเดิม เราควรจะรู้ว่าประโยคใดเป็นประโยคเงื่อนไขก่อน

ผลลัพธ์ที่ได้จะเป็นเซตของชื่อคนที่รู้จัก กล่าวคือ เป็นเซตย่อยของ *known* ที่มีเงื่อนไขว่าเมื่อใช้ฟังก์ชัน *birthday()* หาค่าวันที่ของสมาชิกเซต *known* แล้วจะมีค่าเท่ากับวันที่ปัจจุบันที่ป้อนเข้าไป เราเขียนประโยคเงื่อนไขดังนี้

$card! = \{n: known \mid birthday(n) = today?\}$

การกำหนด Operation Schema ของการตั้งต้นค่าข้อมูลวันเกิด

```
┌── InitBirthday ──────
│ BirthdayBook
├──────────────────────
│ known = ∅
└──────────────────────
```

ข้อกำหนดก่อนเริ่มใช้งานจริงจะมีการตั้งต้นค่าข้อมูลวันเกิดในสมุดวันเกิด *BirthdayBook* เราเริ่มจากการทำ *Schema Inclusion* และได้ผลลัพธ์ดังนี้

```
┌── InitBirthday ──────
│ known: ℙNAME
│ birthday: NAME ⇸ DATE
├──────────────────────
│ known = dom birthday
│ known = ∅
└──────────────────────
```

โดยการประกาศชื่อตัวแปร *known* และ *birthday* เราจะทำการตั้งต้นค่าของ *known* ให้เป็นเซตว่าง ดังนี้

$known = \varnothing$

และเมื่อ *known* เป็นเซตว่างแล้ว จะเห็นได้ว่าเนื่องจากตัวยืนยงข้อมูลของ *BirthdayBook* มีค่าเท่ากับ

$known = dom\ birthday$

เราจะทราบได้ต่อไปว่าเมื่อแทนค่าเซตว่างให้กับ *known* แล้ว

$\varnothing = dom\ birthday$

เมื่อ *dom birthday* เป็นเซตว่างแล้ว เซต *birthday* ก็จะเป็นเซตว่างด้วยเช่นกันในที่สุด ดังนั้นเราจึงไม่ต้องเขียนประโยคว่า

$Birthday = \emptyset$

ไว้ในการทำตั้งต้นค่าข้อมูลใน *Schema* ชื่อ *InitBirthday* อีก

การกำหนด Operation Schema และ Freetype

ในการเขียนข้อกำหนดให้สมบูรณ์นั้น เราต้องคำนึงถึงการเกิดปัญหาขึ้นกรณีที่ไม่สามารถทำงานได้ถูกต้อง โดยจะต้องมีการเตรียมข้อกำหนดเพื่อแจ้งข้อความเตือนหรือรายงานผลการทำงานให้ทราบ

ในที่นี้จะมี *Free Type* ที่ชื่อ *REPORT* ที่มีค่าที่เป็นไปได้แค่สามค่า คือค่า “ok” ค่า “already_known” และค่า “not_known” ดังแสดงไว้ ดังนี้

REPORT ::= *ok* | *already_known* | *not_known*

การแสดงผลหรือรายงานกรณีที่ทำงานได้สำเร็จตามต้องการก็จะแสดงผลออกทางตัวแปรแสดงผลชื่อ *result!* โดยให้แสดงค่า “ok” ดังแสดงไว้ดังนี้

```
┌── Success ────────────
│ result!: REPORT
├──────────────────────
│ result! = ok
└──────────────────────
```

กรณีที่การเพิ่มข้อมูลใหม่ของวันเกิดเข้าสู่ระบบ ถ้ามีการระบุป้อนข้อมูลชื่อคนที่รู้จักซ้ำซ้อนเดิมที่มีอยู่แล้ว เราจะไม่ให้ทำงานต่อโดยจะแสดงผลด้วยค่า “already_known”

```
┌── AlreadyKnown ────────────────────
│ ΞBirthdayBook
│ name?: NAME
│ result!: REPORT
├───────────────────────────────────
│ name? ∈ known
│ result! = already_known
└───────────────────────────────────
```

ในข้อกำหนดมีการตรวจสอบเงื่อนไขก่อนไว้ดังนี้ คือ

name? $\in$ *known*

เพื่อทำการตรวจสอบว่า *name?* ที่มีค่าป้อนเข้ามาใหม่นั้นเมื่อเคยมีอยู่ในเซต *known* แล้ว จะแสดงผลลัพธ์ของ *result!* ว่ามีข้อมูลอยู่แล้วและจะไม่ทำการเพิ่มข้อมูลใหม่ให้

กรณีที่มีการสืบค้นหาข้อมูลวันเกิด โดยการป้อนข้อมูลชื่อที่ไม่มีอยู่ในเซต *known* เข้าไปใช้สืบค้น ข้อกำหนดจะแสดงผลลัพธ์ของ *result!* ว่ามีค่าเป็น "not_known" ดังแสดงไว้ในข้อกำหนด ดังนี้

NotKnown

ΞBirthdayBook

name?: NAME

result!: REPORT

---

name? $\notin$ known

result! = not_known

ในข้อกำหนดข้างต้นมีการกำหนดเงื่อนไขก่อนว่า

name? $\notin$ *known*

ซึ่งทำการตรวจสอบว่าชื่อที่ป้อนเข้ามานั้นไม่มีค่าเท่ากับค่าใดที่ปรากฏเป็นสมาชิกของเซต *known* การค้นหาก็จะไม่สำเร็จ จึงต้องทำการตรวจสอบก่อนและแสดงผลลัพธ์ว่าไม่เป็นที่รู้จัก

ในทางปฏิบัติเราจะนำ *Schema* มารวมกันกำหนดให้เป็นการเพิ่มข้อมูลวันเกิดให้สมบูรณ์ยิ่งขึ้นโดยมีการแสดงผลการทำงานกรณีที่สำเร็จและกรณีที่ไม่สำเร็จด้วยดังนี้

RAddBirthday $\widehat{=}$ (AddBirthday $\wedge$ Success) $\vee$ AlreadyKnown

ในที่นี้จะเห็นได้ว่า *RAddBirthday* จะเป็น *Schema* ที่สมบูรณ์ที่แสดงผลกรณีทำการเพิ่มได้สำเร็จได้ด้วย *AddBirthday* และ *Success* กรณีที่ทำงานไม่สำเร็จจะทำได้ด้วย *AlreadyKnown*

ให้เราสังเกตว่า (*AddBirthday* $\wedge$ *Success*) $\vee$ AlreadyKnown จะให้ผลว่าด้านใดด้านหนึ่งของเครื่องหมาย $\vee$ จะเป็นจริงได้ด้านเดียวเท่านั้น กล่าวคือ ถ้า (*AddBirthday* $\wedge$ *Success*) เป็นจริง แล้ว *AlreadyKnown* จะเป็น

เท็จเสมอ และถ้า *AlreadyKnown* เป็นจริงแล้ว (*AddBirthday* ∧ *Success*) เป็นเท็จเสมอ เราเรียกการทำ *OR* แบบนี้ว่า *Exclusive OR*

ผู้เขียนข้อกำหนดจะต้องเขียนประโยคที่ทำการป้องกันไม่ให้เกิดความผิดพลาดโดยใช้ประโยคที่มีค่าความจริงในทางตรงกันข้ามมาใช้ ดังนี้

AddBirthday จะมีเงื่อนไขก่อนเป็น "*name?* ∉ *known*"

ในขณะที่ *AlreadyKnown* จะมีเงื่อนไขก่อนเป็น "*name?* ∈ *known*"

การกำหนดการสืบค้นข้อมูลวันเกิดที่สมบูรณ์ ดังแสดงได้ ดังนี้

RFindBirthday ≙ (FindBirthday ∧ Success) ∨ NotKnown

คำอธิบายได้เช่นเดียวกันกับการกำหนดการเพิ่มข้อมูลวันเกิด ซึ่งถ้า (*FindBirthday* ∧ *Success*) เป็นจริงแล้ว *NotKnown* จะต้องเป็นเท็จ และถ้า *NotKnown* เป็นจริงแล้ว (*FindBirthday* ∧ *Success*) จะต้องเป็นเท็จเสมอ

ผู้เขียนข้อกำหนดจะกำหนดประโยคป้องกันไว้ดังนี้

FindBirthday จะมีเงื่อนไขก่อนเป็น "*name?* ∈ *known*" จึงทำงานได้

ในขณะที่ *NotKnown* จะมีเงื่อนไขก่อนเป็น "*name?* ∉ *known*"

ส่วนการเตือนข้อมูลวันเกิดจะใช้กรณีเดียว เนื่องจากกำหนดไว้ว่า ไม่ว่าจะมีชื่อที่มีวันเกิดในวันปัจจุบันหรือไม่ก็ตาม ก็สามารถจะทำการ *Remind* ได้

RRemind ≙ Remind ∧ Success

กรณีพบชื่อคนที่เกิดในวันที่ปัจจุบัน ก็จะแสดงรายชื่อทุกคนที่เข้าข่าย แต่ถ้ากรณีไม่พบคนที่เกิดในวันที่ปัจจุบันก็จะแสดงเป็นเซตว่าง

6.4 การวางแผนทวนสอบ

การวางแผนทวนสอบมักจะเริ่มจากการที่เราเลือกเครื่องมือการทวนสอบให้เหมาะสมกับระบบที่เราต้องการทวนสอบ และความพร้อมของคณะทำงานทวนสอบที่มีความรู้พื้นฐานเพียงพอ และเข้าใจในโดเมนของระบบที่เราต้องการทวนสอบเป็นอย่างดี การทวนสอบด้วยการพิสูจน์ทฤษฎีบทยิ่งเป็นเรื่องยาก ถ้าผู้ทวนสอบไม่เข้าใจโดเมนและความเป็นไปของระบบเป็นอย่างดี เนื่องจากระหว่างการพิสูจน์จะต้องมีการเพิ่มเติม ข้อเท็จจริง สมมุติฐาน หรือข้อตั้งเพิ่มเติมจาก การกำหนดประโยคอธิบายเข้าในเครื่องมือ และนำทางเครื่องมือให้ทำการทวนสอบอย่างต่อเนื่องไปเรื่อยจนได้ข้อยุติ

เครื่องมือที่เราเลือกใช้และนำเสนอในบทนี้ คือ เครื่องมือเซดอีฟ ซึ่งทำการพิสูจน์ได้แบบกึ่งอัตโนมัติ [4] เป็นโปรแกรมที่ใช้เป็นเครื่องมือที่ได้รับการพัฒนาขึ้นโดย *ORA* ประเทศแคนาดา ซึ่งโปรแกรมสนับสนุนการนำเข้าข้อกำหนดเชิงรูปนัยภาษาเซด การนำข้อกำหนดเข้าทำได้โดยการใช้สัญลักษณ์

ลาเท็กซ์ที่มีความหมายเป็นไปตามสัญกรณ์เซต ทำให้การกำหนดรหัสสัญกรณ์ของเซตทำได้ยาก

การใช้งานโปรแกรมเซตอีฟทำได้ทั้งในลักษณะเชิงตอบโต้และใช้ในลักษณะแบบนำเข้าเป็นกลุ่ม โดยมีความสามารถในการตรวจสอบวากยสัมพันธ์ของข้อกำหนด และตรวจสอบความสอดคล้องของข้อกำหนดทั้งหมดที่บันทึกเข้ามาได้ นอกจากนี้ยังสามารถกำหนดทฤษฎีและทำการพิสูจน์ข้อกำหนดได้เช่นกัน

โดยทั่วไปการทวนสอบด้วยเครื่องมือเซตอีฟ มักจะครอบคลุมกิจกรรมต่อไปนี้ [27]

1. ตรวจวากยสัมพันธ์และชนิดตัวแปร (syntax & type checking)
2. ตรวจโดเมน (domain checking)
3. การแจกแจงสกีมาเพิ่มเติม (schema expansion)
4. การคำนวณหาเงื่อนไขก่อน (precondition calculation)
5. การพิสูจน์ทฤษฎีบท (theorem proving)

เครื่องมือเซตอีฟทำงานแบบกึ่งอัตโนมัติ โดยจะทำการตรวจกิจกรรมข้อที่ 1 ถึงข้อที่ 4 ให้แบบกึ่งอัตโนมัติจากแบบจำลองระบบที่นำเข้าได้ทันที โดยทำการแจกแจงสกีมาให้ครบถ้วน และเริ่มการตรวจวากยสัมพันธ์และชนิดตัวแปรสำหรับทุกประโยคและทุกตัวแปรที่เขียนไว้ ทำการตรวจโดเมน คำนวณหาเงื่อนไขก่อน ถ้าพบกรณีที่มีความขัดแย้งหรือมีตัวอย่างค้านก็แสดงผลลัพธ์ให้ผู้ทวนสอบทันที โดยผู้ทวนสอบต้องทำการแก้ไขต่อไป อย่างไรก็ตาม สำหรับการพิสูจน์ทฤษฎีบทในกิจกรรมสุดท้ายผู้ทวนสอบจะต้องระบุพันธกรณีการพิสูจน์ (proof obligation) ให้ก่อนเสมอ

ในที่นี้ขอยกตัวอย่างบางคำสั่งที่นิยมใช้ในเครื่องมือเซตอีฟสำหรับการพิสูจน์แบบจำลอง คือ คำสั่ง *"try"* คำสั่ง *"reduce"*

คำสั่ง "*reduce*" ทำงานในสองลักษณะ คือ การใช้กฎ *Rewriting rules* ซึ่งเครื่องมือเซตอีฟมีให้มาเบื้องต้น และผู้ทวนสอบกำหนดเพิ่มจาก *Theorem* ที่หาเพิ่มได้ และการทำ *Simplification* ของประโยค ในขณะที่คำสั่ง *"try"* ทำการเพิ่มประโยคระหว่างการพิสูจน์ได้ เช่น การกำหนดค่าให้กับตัวแปรใด ๆ และให้ทำการพิสูจน์ต่อ หรือการกำหนดพันธกรณีการพิสูจน์โดยใช้คำสั่ง *"try"* เป็นต้น

ตัวอย่างที่ 6-1: ผลของการใช้คำสั่ง try เบื้องต้น

ถ้าเราต้องการพิสูจน์ประโยคข้างล่างเป็นจริง

$$\forall a, b: \mathbb{Z} \mid a \in \mathbb{N} \bullet a..b \subseteq \mathbb{N}.$$

เราก็จะใช้คำสั่ง

=>*try \forall a,b: \num | a \in \nat @ a \upto b \subseteq \nat;*

เป็นคำสั่ง try ตามด้วย LaTeX ของประโยคที่ใส่ จากนั้นเครื่องมือเซดอีฟจะทำการ simplify ให้เป็น

$$a \in \mathbb{Z} \land b \in \mathbb{Z} \land a \in \mathbb{N} \Rightarrow a..b \subseteq \mathbb{N}.$$

โดยการแปลงประโยคเดิมที่มีการเขียนแบบใช้ตัวบ่งปริมาณที่ยากกว่า และเมื่อเราทำคำสั่ง *"rewrite"* จะได้ผลลัพธ์

$$a \in \mathbb{Z} \land b \in \mathbb{Z} \land a \geq 0 \Rightarrow a..b \in \mathbb{P}\mathbb{N}.$$

โดยทำการเขียนประโยคใหม่ตามข้อตั้งที่ว่า $\mathbb{N}$ ก็คือเซตของเป็นเลขจำนวนเต็มนับจากศูนย์โดยเปลี่ยน $a \in \mathbb{N}$ ให้เป็น $a \geq 0$ ให้ได้เอง เป็นต้น

6.5 แบบฝึกหัด

1) โครงสร้างข้อกำหนดเชิงรูปนัยเซดของระบบจัดการข้อมูลวันเกิด (birthday book) ประกอบด้วย *Schema* อะไรบ้าง จัดเป็นประเภทได้อย่างไร?
2) ค่าตัวยืนยงข้อมูล (data invariant) ของระบบจัดการข้อมูลวันเกิดคืออะไร?
3) ค่าเงื่อนไขก่อน (precondition) ของ *Schema* ชื่อ *AddBirthday* ของระบบจัดการข้อมูลวันเกิดคืออะไร?
4) ในระบบจัดการข้อมูลวันเกิด เราสามารถเก็บข้อมูลเพื่อนที่มีชื่อซ้ำกันได้หรือไม่?
5) ในระบบจัดการข้อมูลวันเกิด เรามีเพื่อนที่เกิดในวันเดือนปีเดียวกันได้หรือไม่?
6) เราค้นหาวันเกิดของเพื่อนที่เกิดในวันเดือนปีเดียวกันได้หรือไม่ ผลลัพธ์จะปรากฏเป็นชื่อเพื่อนหลาย ๆ คนใช่ไหม?
7) การวางแผนพิสูจน์ระบบจัดการข้อมูลวันเกิด ทำได้อย่างไร อธิบาย?

7
การทวนสอบด้วยโมเดลเช็กกิง

7.1 ความสำคัญของบทนี้

บทนี้กล่าวถึงความเป็นมาของวิธีทวนสอบแบบโมเดลเช็กกิง รวมถึงความแตกต่างจากวิธีทวนสอบด้วยการพิสูจน์ทฤษฎีบทและใช้กฎนิรนัยในการพิสูจน์สัจนิรันดร์ของพันธกรณีที่เป็นคุณลักษณะที่ต้องการทวนสอบ

ข้อดีโมเดลเช็กกิงใช้งานได้ไม่ยุ่งยากเพราะมีเครื่องอัตโนมัติช่วยเหลืออยู่ โดยให้เตรียมองค์ประกอบสองส่วนหลัก คือ แบบจำลองเชิงรูปนัยของระบบที่สนใจ และคุณลักษณะที่ต้องการทวนสอบ เครื่องมือโมเดลเช็กกิงที่มีอยู่จะทวนสอบอย่างอัตโนมัติว่า แบบจำลองเชิงรูปนัยของระบบนั้นทำงานเป็นไปตามคุณลักษณะที่ต้องการทวนสอบหรือไม่ อย่างไรก็ตาม ความสำคัญอยู่ที่การเขียนอธิบายแบบจำลองเชิงรูปนัยของระบบทำได้อย่างไรบ้าง

โดยทั่วไป หลักการทวนสอบด้วยโมเดลเช็กกิงจะเริ่มจากการแปลงแบบจำลองเชิงรูปนัยให้เป็นกราฟสถานะที่ระบุสถานะและการเปลี่ยนสถานะของระบบในขณะทำงาน เพื่อนำกราฟสถานะนี้ไปแจกแจงเป็นปริภูมิสถานะอีกต่อหนึ่ง และนำมาค้นหารูปแบบของการทำงานตามคุณลักษณะที่ต้องการทวนสอบ อย่างไรก็ดี การเขียนกราฟสถานะยังมีอุปสรรคและเขียนได้ยากสำหรับผู้ที่ไม่คุ้นเคย ทั้งยังมีความซับซ้อนสำหรับระบบที่ทำงานพร้อมกันของหลายสายโยงใยที่เกิดการแทรกสลับและเกิดการทำงานแบบเชิงไม่กำหนด ทำให้ตามรอยสถานะของระบบได้ยาก

ในบทนี้จะกล่าวถึงทางเลือกในการสร้างแบบจำลองเชิงรูปนัยจากแผนภาพอื่นที่คุ้นเคยและใช้งานในการออกแบบระบบซอฟต์แวร์และฮาร์ดแวร์เป็นประจำอยู่แล้ว เช่น แผนภาพยูเอ็มแอล แผนภาพบีเพล แผนภาพบีพีเอ็มเอ็น และเพทริเน็ต เป็นต้น ทั้งนี้มีงานวิจัยจำนวนไม่น้อยในปัจจุบันที่สนใจ และพัฒนากฎการแปลงแผนภาพมาเป็นแบบจำลองเพื่อใช้ในการทวนสอบได้

ตัวอย่างที่ใช้ในบทนี้จะเป็นเครื่องมือการทวนสอบสปินเป็นส่วนใหญ่ โดยใช้ภาษาโพรเมลาในการเขียนอธิบายแบบจำลองระบบ ดังนั้น บทนี้จะนำเสนองานวิจัยที่อธิบายถึงกฎในการแปลงแผนภาพการออกแบบที่คุ้นเคยมาเป็นโพรเมลาได้ทั้งอัตโนมัติและกึ่งอัตโนมัติ รวมถึงการแปลงโปรแกรมภาษาจาวาย้อนกลับมาเป็นโพรเมลาด้วย เช่นกัน นอกจากนี้ยังมีเครื่องมือการทวนสอบที่ใช้

กับเพทริเน็ตชนิดต่าง ๆ ที่มีขีดความสามารถในการทวนสอบสูตรเชิงเวลาได้ เช่น เครื่องมือซีพีเอ็น ที่รับการนำแบบจำลองที่เขียนด้วยคัลเลอร์เพทริเน็ตและเงื่อนไขที่เขียนด้วยภาษาเอ็มแอลเข้ามาทวนสอบได้ เป็นต้น

7.2 วัตถุประสงค์

- เพื่อให้รู้จักและเข้าใจวิธีการทวนสอบด้วยโมเดลเช็กกิง
- เพื่อให้เข้าลักษณะที่ใช้ในการทวนสอบ
- เพื่อนำเสนองานวิจัยที่สนับสนุนการสร้างแบบจำลองสำหรับโมเดลเช็กกิงจากแผนภาพที่ใช้ในการออกแบบระบบทั่วไป

7.3 โมเดลเช็กกิง

โดยทั่วไปก่อนลงมือทวนสอบจะต้องมีของสองสิ่ง คือ แบบจำลองเชิงรูปนัยของระบบที่สนใจ และข้อกำหนดคุณลักษณะของระบบที่ต้องการทวนสอบ ในกรณีที่เราใช้วิธีการทวนสอบเชิงรูปนัยแบบพิสูจน์ทฤษฎี แบบจำลองเชิงรูปนัยจะเป็นการเขียนอธิบายโครงสร้างและพฤติกรรมของระบบด้วยภาษาเชิงรูปนัยก่อน เช่น ภาษาเซด ภาษาวีดีเอ็ม-เอสแอล เป็นต้น จากนั้นก็ทำการพิสูจน์ โดยใช้วิธีนิรนัย (deductive method) กำหนดให้แบบจำลองเชิงรูปนัยของระบบเป็นข้อเท็จจริงหรือสมมุติฐานตั้งต้น และใช้ข้อกำหนดคุณลักษณะของระบบที่ต้องการทวนสอบเป็นเป้าหมายของการพิสูจน์เรียกว่าพันธกรณีการพิสูจน์ ถ้าเราเลือกใช้กฎนิรนัยแล้วได้ผลลัพธ์การพิสูจน์ว่าพันธกรณีดังกล่าวมีค่าจริงแบบสัจนิรันดร์ ก็จะสรุปได้ว่า ระบบทำงานได้ไม่ผิดพลาดจากข้อกำหนดคุณลักษณะที่ต้องการการพิสูจน์ วิธีนี้ค่อนข้างยากและต้องใช้ความรู้พื้นฐานคณิตตรรกศาสตร์เป็นอย่างดี ทำให้ไม่เป็นที่นิยมมากนัก

โมเดลเช็กกิงเป็นทางเลือกหนึ่งของการทวนสอบเชิงรูปนัยของพฤติกรรมการทำงานของระบบที่สนใจว่า ยอมรับคุณลักษณะเป้าหมายหรือไม่ การทวนสอบแบบโมเดลเช็กกิงไม่ใช้เทคนิคการพิสูจน์ทฤษฎีบทหรือใช้หลักการอนุมานทางคณิตตรรกศาสตร์แต่อย่างใด แต่พิสูจน์โดยใช้การแจกแจงปริภูมิสถานะ ของพฤติกรรมการทำงานของระบบที่เราสนใจ คำว่าปริภูมิสถานะหมายถึง ทุกสถานะที่ระบบจะได้รับการแจกแจงอย่างหมดจด พร้อมแสดงการเปลี่ยนแปลงจากสถานะหนึ่งไปสู่อีกสถานะหนึ่งของระบบ เมื่อได้ปริภูมิสถานะแล้ว ก็นำคุณลักษณะเป้าหมายที่ต้องการมาค้นหาว่าพบหรือไม่ เช่น คุณลักษณะเป้าหมายที่เขียนกำหนดไว้ว่า ค่า “$balance \geq 0$” เสมอ ดังนั้นถ้าเราค้นหาในปริภูมิสถานะแล้วพบว่า ค่าตัวแปรชื่อ *balance* มีค่ามากกว่าหรือเท่ากับศูนย์เสมอ โดยที่บางสถานะจะมีค่า *1000* บางครั้งอาจจะเท่ากับ *0* แต่ไม่มีสถานะของระบบใดเลยที่

balance มีค่าน้อยกว่า *0* ผลการทวนสอบ คือ ผ่าน เป็นต้น กรณีที่ค้นหาแล้วพบสถานะที่มีผลไม่เป็นไปตามคุณลักษณะเป้าหมาย การทวนสอบอาจจะแสดงตัวอย่างค้านที่พบได้เช่นกัน และด้วยการทำโมเดลเช็กกิงสามารถประยุกต์ใช้อัลกอริธึมการค้นปริภูมิสถานะได้ จึงนิยมสร้างเป็นเครื่องมือสนับสนุนการทวนสอบแบบโมเดลเช็กกิงแบบอัตโนมัติ และเป็นทางเลือกสำหรับผู้ทำการทวนสอบที่อาจจะไม่เชี่ยวชาญเรื่องพื้นฐานด้านคณิตตรรกศาสตร์นัก

อย่างไรก็ดี ในทางปฏิบัติถ้าต้องการลดเวลาในการค้นเพื่อทวนสอบก็ต้องใช้เทคนิคการค้นหานิเสธของคุณลักษณะดังที่กล่าวมาแล้วข้างต้นแทนได้ ทำให้บางครั้งเรานิยามโมเดลเช็กกิงว่า เป็นเทคนิคการทวนสอบที่ใช้อัลกอริทึมการค้นหาข้อผิดพลาดของระบบที่ไม่เป็นไปตามข้อกำหนดคุณลักษณะที่ต้องการทวนสอบได้อย่างอัตโนมัติ [6] ได้รับการนำเสนอครั้งแรกในปี ค.ศ. 1981 โดย *Clarke Allen Emerson* เพื่อทวนสอบระบบที่ทำงานพร้อมกันและระบบทำงานแบบกระจาย ในช่วงเวลาใกล้เคียงกันนี้ก็มีการเสนอวิธีการค้นในทำนองเดียวกันนี้โดย *Quielle* และ *Sifakis* โมเดลเช็กกิงเป็นวิธีที่มีข้อดีกว่าการทวนสอบเดิมที่ใช้การพิสูจน์แบบการพิสูจน์ทฤษฎีบท ผู้ทวนสอบไม่จำเป็นต้องมีความรู้พื้นฐานเรื่องการพิสูจน์ด้านคณิตตรรกศาสตร์มากนัก อย่างไรก็ตามโมเดลเช็กกิงในยุคแรกเริ่มนี้มีข้อจำกัดอยู่มาก เนื่องจากวิธีนี้จะต้องทำการหาปริภูมิสถานะของระบบที่ต้องการทวนสอบทุกสถานะที่เป็นไปได้ทั้งหมด สำหรับระบบที่มีอยู่จริงและทำงานซับซ้อนมักจะมีจำนวนสถานะ และการเปลี่ยนสถานะมากมายมหาศาล ทำให้เกิดการระเบิดของปริภูมิสถานะ (state space explosion) และไม่สามารถทวนสอบได้สำเร็จ

ตัวอย่างการใช้โมเดลเช็กกิงทวนสอบโปรแกรมที่เขียนโดยโปรแกรมเมอร์ โดยมีข้อสมมุติฐานว่าโปรแกรมจะเรียกใช้ตัวแปรต่าง ๆ และค่าในตัวแปรที่เราสนใจเลือกมาจะแสดงถึงสถานะของระบบที่เราสนใจ ชนิดตัวแปรในโปรแกรมที่เขียนด้วยภาษาใด ๆ นั้นมักจะมีข้อจำกัดตามขนาดของหน่วยความจำ เช่น ชนิดตัวแปรแบบ *int* จะเก็บด้วยหน่วยความจำ *32* bit ซึ่งจะหมายถึง ตัวแปรที่เป็น *int* จะเก็บค่าจำนวนเต็มระหว่าง (-2^{31}) ถึง $(2^{31}-1)$ รวมค่า 0 ด้วย ซึ่งถือว่ามีจำนวนจำกัด เป็นต้น ยกตัวอย่างโปรแกรมการเรียงตัวเลขแบบ *BubbleSort* ดังแสดงในรูปที่ 7.1

ปริภูมิสถานะจะมีจำนวนเท่าไรขึ้นอยู่กับตัวเลขที่นำเข้ามาเรียงลำดับในตัวแปรชื่อ *a* ซึ่งเป็นแถวลำดับ (array) ของค่าจำนวนเต็มชนิด *int* ถ้าตัวเลขที่นำเข้ามาเรียงลำดับมีสองค่า เราจะต้องทำการแจกแจงปริภูมิสถานะเพื่อให้ครอบคลุมค่าจำนวนเต็มในตัวแปรนำเข้า *a* ทุกกรณีคือ $2x2^{32}$ ซึ่งจะมีจำนวนราวแปดพันหน้าร้อยล้านกว่ากรณีที่เป็นไปได้ และในทางปฏิบัติตัวเลขที่นำมา

เรียงลำดับอาจจะมีจำนวนมากเป็นหลักร้อยตัวก็ได้ ซึ่งกรณีที่ค่าตัวเลขนำเข้ามาทวนสอบถูกแจกแจงอย่างหมดจด ต้องมีจำนวนมากจนไม่สามารถทวนสอบได้ เราเรียกกรณีที่ปริภูมิสถานะที่มีจำนวนมากจนทวนสอบไม่ได้ว่าเกิดการระเบิดของปริภูมิสถานะ ซึ่งมักจะเกิดกับระบบซอฟต์แวร์ เนื่องจากมีจำนวนตัวแปรมาก

```
BubbleSort(int[] a, int n)
{
   for (i=0; i<n-1; i++)  {
      for (j=0; j<n-1-i; j++)  {
         if (a[j+1] < a[j]) {
            tmp = a[j];
            a[j] = a[j+1];
            a[j+1] = tmp;
         }
      }
   }
}
```

รูปที่ 7.1 โปรแกรม BubbleSort ที่ใช้เรียงลำดับตัวเลข

ในการทวนสอบจริงเราจำเป็นจะต้องดัดแปลงวิธีการเดิม มาใช้ประพจน์แทนการสังเกตค่าทุกค่าที่เป็นไปได้ของตัวแปร เช่น ถ้าเรามีโปรแกรมที่กำหนดตัวแปร *int* ชื่อ x เราเพียงต้องการสังเกตพฤติกรรมของโปรแกรมว่ามีสถานะใดของโปรแกรมที่ทำให้ตัวแปร x มีค่าน้อยกว่า 0 หรือไม่เท่านั้น เราจึงกำหนดประพจน์เดี่ยว $p = x<0$ และเรามาสังเกตค่าความจริงของประพจน์ p แทน ซึ่งจะทำให้จำนวนปริภูมิสถานะของระบบที่สนใจมีจำนวนน้อยลงมาก

ในบทนี้จะกล่าวถึงเทคนิคการทวนสอบแบบจำลองเชิงรูปนัยของระบบที่สนใจ โดยแสดงวิธีการที่จะช่วยลดจำนวนปริภูมิสถานะได้ และแสดงตัวอย่างการใช้เครื่องมือสปิน และภาษาโพรเมลาในการทวนสอบ

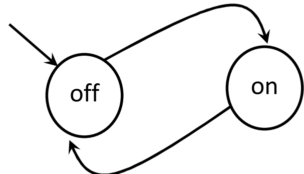
ความเป็นมาของการทวนสอบเชิงรูปนัย

นักบุกเบิกการทวนสอบเชิงรูปนัยชื่อ *C.A.R. Hoare* ได้นำเสนอวิธีนิรนัยในการพิสูจน์โปรแกรมที่เขียนขึ้นว่าถูกต้องหรือไม่ ต่อมาระบบการพิสูจน์ด้วยการนิรนัยได้รับการขยายผลไปครอบคลุมการพิสูจน์ความถูกต้องของระบบที่ทำงานพร้อมกัน โดย ในช่วงปี ค.ศ. 1977 *Amir Pnueli* นำเสนอว่าควรนำตัวดำเนินการเชิงเวลาของตรรกศาสตร์เชิงเวลาแบบลำดับมาทวนสอบระบบได้ และในปี ค.ศ. 1981 *Amir Pnueli* และ *Zohar Manna* โดยการนำเสนอขยายผลไปใช้ตรรกศาสตร์ต้นไม้การคำนวณมาใช้ทวนสอบระบบอย่างมีประสิทธิภาพมากขึ้น ทำให้ *Amir Pnueli* และคณะได้รับรางวัล *Turing Award* ในปี ค.ศ. 1996

สำหรับผลงานที่นำตรรกศาสตร์เชิงเวลามาปรับใช้ในการทวนสอบ แต่การใช้เทคนิคเหล่านี้ยังคงต้องมีความรู้ลึกด้านคณิตตรรกศาสตร์อยู่ดี (ขณะนั้นยังไม่มีการคิดค้นเทคนิคแบบโมเดลเช็กกิง แค่พบว่าใช้ตรรกศาสตร์เชิงเวลามาทวนสอบจะมีประสิทธิภาพมากมาย) ในขณะที่นักคณิตศาสตร์ส่วนใหญ่ยังแย้งว่า ควรพิสูจน์คุณสมบัติเหล่านี้โดยใช้เกณฑ์ว่าเป็นสัจนิรันดร์หรือไม่จะดีกว่า ต่อมาในปี ค.ศ. 2008 *Edmund Clarke, Allen Emerson,* และ *Joseph Sifakis* ได้รับรางวัล *Turing Award* สำหรับผลงานเรื่องการคิดค้นโมเดลเช็กกิงเพื่อนำมาใช้ในการทวนสอบแทนวิธีเดิม และโมเดลเช็กกิงที่ว่านี้กลายเป็นที่นิยม และใช้กันแพร่หลายในการทวนสอบระบบฮาร์ดแวร์และซอฟต์แวร์ แต่มีข้อจำกัดอยู่บ้างในหลักการโมเดลเช็กกิงในยุคแรกเริ่ม และต่อมาจึงมีการดัดแปลงเทคนิคและอัลกอริธึม ตลอดจนหาวิธีการลดขนาดปริภูมิสถานะ

หลักการที่ *Clarke* และคณะนำเสนอ คือ การทวนสอบทุกกรณีของการทำงานของระบบหรือโปรแกรมแบบทำงานพร้อมกันสามารถทำได้ โดยการนำโปรแกรมระบบทำงานพร้อมกันมาแปลงเป็นออโตมาตาแบบจำกัดเชิงไม่กำหนด (nondeterministic finite automata) $NDFA_1$ และนิเสธของคุณลักษณะที่ต้องการทวนสอบแปลงเป็นออโตมาตาแบบจำกัดเชิงไม่กำหนด $NDFA_2$ จากนั้นให้นำ $NDFA_1$ และ $NDFA_2$ มาจำลองทำงานพร้อมกัน เพื่อหาว่าพบกรณีที่มีสตริงนำเข้าที่ทั้งคู่ $NDFA_1$ และ $NDFA_2$ ยอมรับด้วยกันไหม (สตริงนำเข้าเป็นตัวแทนของเส้นทางการทำงานของระบบ) ถ้าพบการยอมรับสตริงจะแสดงว่าสตริงนำเข้านั้นเป็นกรณีที่ผิดพลาดที่เป็นตัวอย่างค้านที่ทำให้ระบบทำงานผิดพลาดจากคุณลักษณะที่ต้องการทวนสอบนั่นเอง ตั้งแต่นั้นมาโมเดลเช็กกิงก็มักจะได้รับการระบุให้ทำการหานิเสธของคุณลักษณะที่ต้องการแทน และคาดว่าจะไม่พบนิเสธของคุณลักษณะนั้น

ระบบที่สนใจในการทวนสอบต้องทำงานด้วยออโตมาตาแบบจำกัดที่มีสถานะจำกัดเท่านั้น แต่อาจจะมีการทำงานแบบไม่สิ้นสุดก็ได้ (infinite execution of finite automata) จากรูปที่ 7.2 แสดงระบบสัญญาณไฟกระพริบที่ทำงานแบบไม่สิ้นสุด โดยจะอยู่ในสถานะไฟดับ *"off"* และไฟสว่าง *"on"* สลับกันไปตลอดการทำงาน



รูปที่ 7.2 สถานะจำกัดของระบบสัญญาณไฟกระพริบที่ทำงานแบบไม่สิ้นสุด

ทำไมเราต้องใช้ออโตมาตาแบบจำกัดเชิงไม่กำหนดมาสร้างแบบจำลองระบบที่ทำงานแบบพร้อมกันด้วย คำตอบก็คือ ระบบที่ทำงานแบบพร้อมกันประกอบด้วยการทำงานของหลายเส้นโยงใยที่ทำงานพร้อมกัน และอาจจะมีการประสานกันในบางจังหวะการทำงานเพื่อแลกเปลี่ยนข้อมูล ถ้าเราคิดแบบง่าย ๆ โดยการใช้ออโตมาตาแบบจำกัดเชิงกำหนดมาจำลองเส้นโยงใยแต่ละส่วนและให้ทำงานพร้อม ๆ กัน จะพบว่าเกิดการแทรกสลับของการทำงานจากการที่แย่งกันใช้หน่วยประมวลผลกลาง ทำให้ทำความเข้าใจได้ยาก ดังนั้นเราจึงต้องรวมออโตมาตาแบบจำกัดเชิงกำหนดทุกตัวเข้าด้วยกัน กลายเป็นออโตมาตาแบบจำกัดเชิงไม่กำหนดตัวเดียวที่แจกแจงทุกกรณีของการแทรกสลับของระบบได้ครบถ้วน ด้วยหลักการของออโตมาตาที่ว่า เราสามารถเขียนออโตมาตาแบบจำกัดเชิงไม่กำหนดได้ โดยรวมจากหลายออโตมาตาแบบจำกัดเชิงกำหนด

สรุปอีกครั้งว่า โมเดลเช็กกิง เป็นวิธีการที่ใช้ในการทวนสอบระบบโดยวิเคราะห์ปริภูมิสถานะเป็นหลัก ไม่ได้ใช้หลักการพิสูจน์แบบนิรนัยที่ทำกันอยู่ ดังนั้นผู้ใช้งานโมเดลเช็กกิงไม่ต้องมีความรู้พื้นฐานมากและปัจจุบันมีเครื่องมือช่วยมากมาย เรามักจะพบว่าโมเดลเช็กกิงเหมาะกับการทวนสอบระบบที่ทำงานพร้อมกัน และระบบที่ทำงานแบบกระจาย เนื่องจากออกแบบและสร้างกรณีทดสอบ (test case)

ปริภูมิสถานะ

ปริภูมิสถานะ คือ สถานะที่มีทั้งหมดของระบบและความเชื่อมโยงของสถานะที่แสดงถึงการเปลี่ยนสถานะจากที่หนึ่งไปอีกที่หนึ่ง ปริภูมิสถานะมักเขียนด้วยโครงสร้างกราฟหรือโครงสร้างต้นไม้ และเราสามารถปริภูมิสถานะจากโครงสร้างกราฟไปเป็นโครงสร้างต้นไม้ และในทางกลับกันได้

การที่ปริภูมิสถานะมีโครงสร้างเป็นต้นไม้ จะทำให้เราเห็นสถานะเริ่มต้นและสถานะถัดไปได้สะดวก และการค้นหาปริภูมิสถานะสามารถทำได้โดยการเดินตัดผ่าน (traverse) ต้นไม้ในรูปแบบค้นหาแนวลึกก่อน (depth first search) หรือการค้นหาแนวกว้างก่อน (bread first search) เป็นต้น เราจะพบว่าระบบที่ทำงานสิ้นสุดได้ สถานะสุดท้ายจะอยู่ที่โหนดใบ (leaf node) ของต้นไม้

กรณีที่ระบบที่ทำงานแบบไม่สิ้นสุด มักจะเกิดปัญหาในการวาดปริภูมิสถานะเนื่องจากต้นไม้จะมีความลึกไม่สิ้นสุดเช่นเดียวกัน และเราสังเกตได้ว่าเมื่อแจกแจงต้นไม้ลงลึกไปเรื่อย ๆ จะพบว่าจะไม่พบสถานะใหม่อีกเลย ต้นไม้จะวนกลับมาใช้สถานะเดิมที่มีอยู่แล้วเท่านั้น เราเรียกระบบแบบนี้ว่า ระบบที่มีสถานะจำกัด แต่ทำงานแบบไม่สิ้นสุด วิธีการแก้ปัญหากรณีที่ต้นไม้มีความลึกไม่สิ้นสุดแบบนี้ก็คือ เราต้องกลับมาใช้โครงสร้างแบบกราฟแทน หรือ ต้องมีการจำกัด

ความลึกในการค้นต้นไม้ไว้ หรืออาจจะใช้วิธีการจำว่าสถานะที่เราค้นถึงเริ่มซ้ำกับสถานะที่เคยพบแล้วให้หยุดได้ เป็นต้น

ส่วนกรณีที่ระบบทำงานไม่สิ้นสุดและมีสถานะไม่จำกัดจะเป็นปัญหาอุปสรรคมาก และในทางปฏิบัติเรามักจะสร้างปริภูมิสถานะสำหรับระบบที่มีสถานะไม่จำกัดและทำงานไม่สิ้นสุดได้เลย

ขอยกตัวอย่างปริภูมิสถานะของระบบสัญญาณไฟจราจรที่วาดด้วยออโตมาตาในรูปแบบกราฟปริภูมิสถานะ และต้นไม้ปริภูมิสถานะที่แสดงในรูปที่ 7.3 และสถานะกำกับด้วย R, G, Y แสดงถึงการเปิดสัญญาณไฟแดง ไฟเขียว และไฟเหลืองตาลำดับ เราอธิบายได้ว่าระบบเริ่มทำงานที่สถานะเริ่มต้นที่สถานะ R คือ เริ่มเปิดไฟแดง จากนั้นทางเลือกที่ไปได้คือ การคงเปิดไฟแดงต่อไป หรือเปลี่ยนเป็นไฟเขียว

สำหรับกราฟปริภูมิสถานะจะมีเส้นออกจากสถานะ R วาดวนย้อนกลับสถานะเดิม และอีกเส้นออกจากสถานะ R ไปสู่สถานะ G จากนั้นเราก็พิจารณาต่อที่สถานะ G ว่าจะไปสถานะถัดไปใดได้บ้าง ซึ่งก็คือวนกลับสถานะเดิม หรือเปลี่ยนสถานะต่อไปที่สถานะ Y และจากสถานะ Y ก็เปลี่ยนสถานะกลับมาที่สถานะ R เราพบว่าถัดจากนี้ไประบบจะเริ่มใช้สถานะเดิมที่มีอยู่แต่เปลี่ยนไปมาอย่างไม่สิ้นสุด

ในขณะที่ต้นไม้สถานะเริ่มต้นจากโหนดรากที่เป็นสถานะเริ่มต้น R จะเปลี่ยนไปเป็นสถานะอื่นใดถัดไปได้บ้าง เราจะต้องให้แจกแจงสถานะให้ครบในความลึกของต้นไม้ชั้นถัดไป คือ จากสถานะ R ที่โหนดรากต้นไม้มีเส้นเชื่อมไปชั้นล่างคือสถานะ G และสถานะ R นั่นเอง สังเกตว่าเราจะวาดโหนดสถานะ R ซ้ำอีกครั้ง และพิจารณาต่อไปลงไปที่ความลึกระดับที่สองของต้นไม้ถัดไป เราสังเกตได้ว่า จะสถานะ G จะไปต่อที่สถานะ Y และพบว่าเริ่มมีสถานะที่ซ้ำเกิดขึ้นในทางปฏิบัติเราจะวาดสถานะที่ซ้ำไปได้เรื่อยๆ แต่เนื่องจากระบบนี้เป็นระบบทำงานไม่สิ้นสุด ดังนั้น เราเลือกวิธีให้ต้นไม้ปริภูมิสถานะหยุดที่ความลึกชั้นที่สามเนื่องจากเราพบว่ามีโหนดสถานะซ้ำหมดทุกโหนดใบแล้ว เราอาจจะเรียกต้นไม้ปริภูมิสถานะว่าต้นไม้การคำนวณ (computation tree) ก็ได้

จากรูปต้นไม้ปริภูมิสถานะ เราสามารถทำการค้นหาคุณลักษณะที่ต้องการทวนสอบได้ เช่น ถ้าเราต้องการรู้ว่าระบบสัญญาณไฟจราจรนั้นมีคุณลักษณะว่า “จะต้องเปิดไฟเขียวได้ในที่สุด” หรือใช้สูตรเชิงเวลาคือ $<>G$ แล้วดังนั้นเราจะทวนสอบโดยการหาเส้นทางการกระทำ ρ จากโหนดรากเดินตัดผ่านลงลึกไปทีละเส้นทางจะได้เส้นทางทั้งหมด คือ

ρ_1: $R, G, Y, R, \ldots$

ρ_2: $R, G, Y, Y, \ldots$

ρ_3: $R, G, G, Y, \ldots$

ρ_4: $R, G, G, G, \ldots$

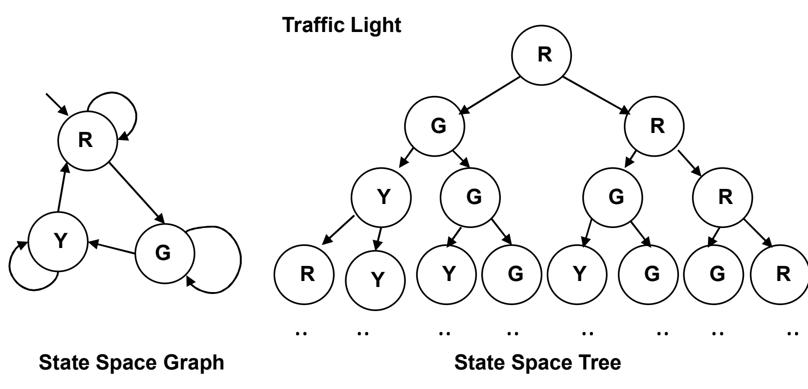
ρ_5: $R, R, G, Y, \ldots$

ρ_6: $R, R, G, G, \ldots$

ρ_7; $R, R, R, G, \ldots$

ρ_8: $R, R, R, R, \ldots$

เราค้นพบว่าเส้นทางการกระทำ ρ_1 ถึง ρ_7 เราหาสถานะ G ได้ในที่สุด ยกเว้นเส้นทาง ρ_8 ที่สถานะคงตัวอยู่ที่ R ตลอดเวลา ทำให้เกิดกรณีที่เรียกว่าติดตายของระบบ กล่าวคือ สัญญาณไฟแดงเปิดสว่างตลอดการทำงานในเส้นทางนี้

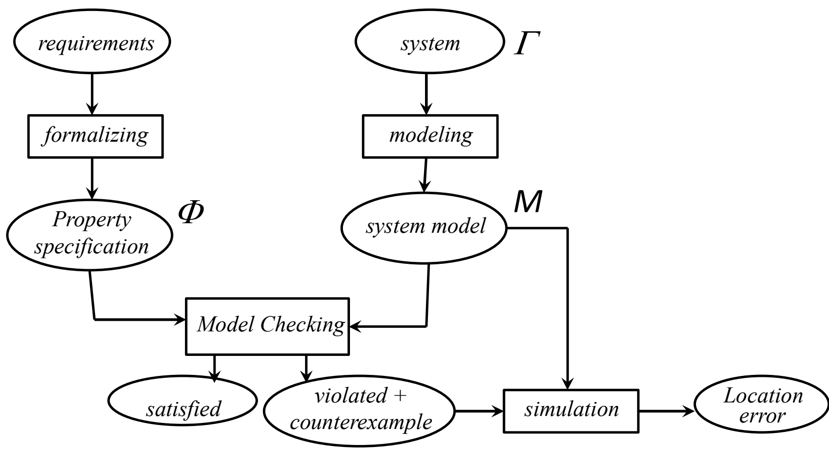


รูปที่ 7.3 กราฟปริภูมิสถานะและต้นไม้ปริภูมิสถานะของระบบ

ภาพรวมกระบวนการทวนสอบด้วยโมเดลเช็กกิง

กระบวนการทวนสอบเชิงรูปนัยโดยใช้โมเดลเช็กกิงได้รับการกำหนดไว้เป็นมาตรฐานทั่วไปว่า เริ่มต้นจะต้องมีองค์ประกอบสองส่วน คือ แบบจำลองเชิงรูปนัยของระบบที่สนใจ (แบบจำลองที่นำมาใช้นี้ เราอาจจะยกเพียงบางส่วนของระบบจริงมาก็ได้ เช่น กรณีใช้ระบบตรรกศาสตร์เชิงเวลาแบบลำดับ ถ้าระบบจริงมีเซตของเส้นทางลำดับการงาน Γ แล้ว เราเลือกเซตย่อย M โดยที่ $M \subseteq \Gamma$ มาใช้แบบจำลองเชิงรูปนัยที่ใช้ในการทวนสอบก็ได้ แต่ M ต้องครอบคลุมพฤติกรรมที่คาดว่าจะทวนสอบให้ครบ) และคุณลักษณะที่ต้องการทวนสอบ Φ นำมาทวนสอบโดยเครื่องมือการทวนสอบแบบโมเดลเช็กกิงที่พัฒนาขึ้นมา การสร้างแบบจำลองระบบ M และการกำหนดคุณลักษณะ Φ ที่ต้องการทวนสอบทำได้อย่างไรและใช้หลักการอย่างไร ให้ดูจากเครื่องมือทวนสอบเป็นสำคัญ เครื่องมือทวนสอบส่วนใหญ่จะทำงานอย่างอัตโนมัติเพื่อให้คำตอบเป็นถูกหรือผิดเท่านั้น กรณีที่คำตอบเป็นถูกหมายถึงระบบที่สนใจมีคุณลักษณะที่ต้องการทวนสอบ

ในทางตรงข้ามกรณีที่คำตอบเป็นผิด จะแสดงตัวอย่างค้านออกมาให้ กระบวนการนี้แสดงในรูปที่ 7.4



รูปที่ 7.4 กระบวนการทวนสอบด้วยโมเดลเช็กกิงแบบมาตรฐาน [5]

ตัวอย่างเช่น ถ้าเราเลือกใช้เครื่องมือทวนสอบชื่อสปิน ซึ่งกำหนดไว้ว่าให้สร้างแบบจำลองระบบที่สนใจด้วยภาษาโพรเมลา และให้กำหนดคุณลักษณะที่ต้องการทวนสอบไว้ด้วยสูตรตรรกศาสตร์เชิงเวลาแบบลำดับ จากนั้นเครื่องมือสปินจะทำการทวนสอบให้แบบอัตโนมัติและให้ผลลัพธ์เป็นคำตอบว่าถูกหรือผิดเท่านั้น คำตอบว่าถูกหมายถึง ระบบที่สนใจมีพฤติกรรมเป็นไปตามคุณลักษณะที่ต้องการทวนสอบได้ ส่วนคำตอบว่าผิดหมายถึง มีกรณีหนึ่งใดที่ระบบที่สนใจไม่เป็นไปตามคุณลักษณะที่ต้องการทวนสอบ และสปินจะแจ้งเส้นทางของการทำงานกรณีที่ผิดให้ด้วยเป็นตัวอย่างค้าน เพื่อผู้ทำการทวนสอบจะได้นำมาจำลองการทำงานเฉพาะกรณีนั้นเพื่อหาสาเหตุข้อผิดพลาดที่มีได้ เป็นต้น

คุณลักษณะทั่วไปที่ใช้ในการทวนสอบ

เราควรจะทวนสอบคุณลักษณะพื้นฐานของระบบ คือ ความปลอดภัย ความดำเนินชีวิต และความเป็นธรรมเป็นเบื้องต้นเสมอ จากนั้นเราอาจจะต้องการทราบเพิ่มเติมว่า ระบบที่สนใจทำงานผิดพลาดหรือไม่ เช่น ในระบบฮาร์ดแวร์ เราอาจจะต้องการทราบว่าระบบมีการตอบสนองต่อสัญญาณร้องได้ดีหรือไม่ หรือเมื่อมีการร้องขอแล้วมีการตอบสนองแบบเรียงลำดับความสำคัญก่อนหลังหรือไม่ หรือวงจรการทำงานมีความเสถียรภาพของสัญญาณหรือไม่ หรือวงจรมีสัญญาณคู่ใดที่มีสหสัมพันธ์หรือไม่ หรือความไม่เกิดร่วม หรือ เกิดคำร้องขอจนกว่าตอบรับ [5] เป็นต้น

การทวนสอบความปลอดภัย หมายถึง การทวนสอบให้แน่ใจว่าระบบจะมีพฤติกรรมที่ไม่ก่อให้เกิดข้อผิดพลาดเสมอ หรือที่ประโยคกล่าวไว้ว่า *"Something bad will not happen ever"* โดยการหาข้อผิดพลาดต่าง ๆ มาใส่เครื่องหมายนิเสธ ดังนั้นการกำหนดคุณลักษณะความปลอดภัยจะทำได้ด้วยแบบรูปของสูตรเชิงเวลาคือ []$\neg p$ โดยที่ประพจน์ p คือข้อผิดพลาดที่เราไม่ต้องการนั่นเอง

ตัวอย่างที่ 7-1: คุณลักษณะความปลอดภัยเบื้องต้น

กำหนดให้คุณลักษณะความปลอดภัยของการควบคุมอุณหภูมิแกนปฏิกรณ์ปรมาณู []$\neg$(*reactor_temperature>1000*) หมายถึงอุณหภูมิแกนปฏิกรณ์ปรมาณูจะเกิน *1000* องศาไม่ได้เสมอในการทำงาน

กำหนดให้ []$\neg$(*request* $\wedge$ O *ignore*) หมายถึงถ้ามีคำร้องเกิดขึ้นในเวลาถัดมาหนึ่งหน่วยเวลาจะได้รับการเพิกเฉยไม่ได้เสมอในการทำงาน เป็นต้น

การทวนสอบความดำเนินชีวิต หมายถึง การทวนสอบให้แน่ใจว่าระบบจะมีพฤติกรรมที่ต้องการในที่สุดจนได้ หรือประโยคที่กล่าวว่า *"Something good will eventually happen"* โดยการหาข้อดีที่ต้องการมากำหนดเป็นประพจน์ในแบบรูปของสูตรเชิงเวลาคือ <>q โดยที่ประพจน์ q คือข้อดีที่ต้องการในที่สุดแม้ว่าจะไม่เกิดทันทีทันใดก็ได้ หรือเกิดเมื่อไรก็ได้ขอให้เกิดอย่างน้อยครั้งเดียวในระบบ

ตัวอย่างที่ 7-2: คุณลักษณะความดำเนินชีวิตเบื้องต้น

กำหนดให้ <>*terminate* หมายถึงระบบจะสามารถจบการทำงานได้ในที่สุด หรือ <>($y>5$) หมายถึงตัวแปร y ในระบบจะมีค่ามากกว่า 5 ได้ ณ สถานะใดก็ได้ในอนาคต ไม่ต้องเป็นจริงทันที กล่าวคือ ประพจน์ ($y>5$) จะเป็นจริงในที่สุดนั่นเอง เป็นต้น เนื่องจากการออกแบบระบบให้มีพฤติกรรมการทำงานแบบใดได้บ้างก็เป็นความต้องการให้เกิดการทำงานเช่นนั้นครบถ้วน นั่นหมายถึงทุกสถานะของระบบควรจะต้องได้รับการเข้าถึงครบทุกจุดในที่สุด ดังนั้นเราถือว่าการเข้าถึงได้ครบ (reachability) ก็เป็นส่วนหนึ่งของความดำเนินชีวิตด้วยเช่นกัน รวมถึงการจบการทำงานได้จริง (termination) ก็เป็นความต้องการที่ควรเกิดขึ้นในที่สุดเช่นกัน

การทวนสอบความเป็นธรรม หมายถึง การทวนสอบให้แน่ใจว่าระบบจะมีพฤติกรรมที่ให้การตอบสนองอย่างสม่ำเสมอตลอดการทำงานของระบบ หรือประโยคที่กล่าวว่า *"If we attempt something infinitely often, then we will*

succeed infinitely often" โดยการพิจารณาว่าระบบจะมีการร้องขอและตอบสนองตามมาได้ในที่สุดอย่างสม่ำเสมอ

ตัวอย่างที่ 7-3: คุณลักษณะความเป็นธรรมเบื้องต้น

กำหนดให้ []<>*attempt* ⟹ []<>*succeed* หมายถึงถ้ามีการเกิด *attempt* แบบเป็นประจำอย่างไม่นับถ้วนแล้วจะมีการเกิดการตอบสนอง *succeed* แบบเป็นประจำอย่างไม่นับไม่ถ้วนตามมาเช่นกันเสมอ เป็นต้น

การทวนสอบการตอบสนอง หมายถึง การทวนสอบให้แน่ใจว่าระบบเมื่อได้รับคำร้องขอครั้งหนึ่งแล้ว จะต้องมีการตอบสนองในที่สุดจนได้

ตัวอย่างที่ 7-4: คุณลักษณะการตอบสนอง

กำหนดให้ *request* ⟹ <>*response* หมายถึงถ้ามีการเกิดสัญญาณ *request* แล้วครั้งใด จะมีการเกิดการตอบสนอง *response* ทันทีหรือตามมาในที่สุด เป็นต้น

การทวนสอบการเรียงลำดับก่อนหลัง หมายถึง การทวนสอบให้แน่ใจว่าระบบเมื่อได้รับคำร้องขอครั้งหนึ่งแล้ว จะต้องมีการตอบสนองทันทีโดยมีการทำงานแรกระยะหนึ่งแล้วตามมาด้วยการทำงานสุดท้ายเมื่อจบงานแรกทันที

ตัวอย่างที่ 7-5: คุณลักษณะการเรียงลำดับก่อนหลัง

กำหนดให้ *request* ⟹ *pre-task* U *post-task* หมายถึงถ้ามีการเกิดสัญญาณ *request* แล้วครั้งใด จะมีการเกิดการตอบสนอง *pre-task* ทันทีและดำเนินไปจนจบและตามด้วยการตอบสนอง *post-task* ตามมาทันที เป็นต้น

การทวนสอบความเป็นเสถียรภาพ หมายถึง การทวนสอบให้แน่ใจว่าระบบมีการทำงานอย่างใดอย่างหนึ่งที่อาจเกิดขึ้นและหยุดไปสลับไปมาช่วงแรกจนในที่สุดการทำงานนั้นเกิดขึ้นประจำตลอดเสมอมีความเสถียรภาพ

ตัวอย่างที่ 7-6: คุณลักษณะความเป็นเสถียรภาพ

กำหนดให้ <>[]*run* หมายถึงจะเกิดสัญญาณ *run* ที่เป็นจริงตลอดไปในที่สุด กล่าวคือสัญญาณ *run* อาจจะเกิดบ้างไม่เกิดบ้างในระยะแรก แต่ในที่สุดก็มีสัญญาณ *run* ไปตลอดไม่สิ้นสุดหรือจนระบบจบการทำงาน

การทวนสอบความมีสหสัมพันธ์ หมายถึง การทวนสอบให้แน่ใจว่าเมื่อระบบทำงานหนึ่งใดแล้ว ในที่สุดจะต้องทำงานอีกอย่างหนึ่งที่คู่กันเสมอ

ตัวอย่างที่ 7-7: คุณลักษณะความมีสหสัมพันธ์

กำหนดให้ $<>openfile \Rightarrow <>closefile$ หมายถึงเมื่อระบบมีการทำงานอย่างหนึ่งจะต้องทำงานอีกอย่างที่คู่กันตามมาเสมอ เช่น ถ้ามีการ *openfile* แล้วก็จะต้อง *closefile* ในที่สุดให้ได้

การทวนสอบความไม่เกิดร่วมกัน หมายถึง การทวนสอบให้แน่ใจว่าถ้าระบบมีการทำงานสองอย่างที่เกิดขึ้นพร้อมกันไม่ได้เด็ดขาด ดังนั้นระบบต้องเลือกทำงานทำทีละงานเท่านั้นหรือไม่ทำทั้งสองงานเลย

ตัวอย่างที่ 7-8: คุณลักษณะความไม่เกิดร่วม

กำหนดให้ $[]\neg(ack1 \wedge ack2)$ หมายถึงเมื่อระบบมีการทำงานสองอย่าง *ack1* และ *ack2* ที่เกิดขึ้นพร้อมกันไม่ได้เด็ดขาด ถ้ามีการทำงานต้องทำทีละงานเท่านั้นหรือไม่ทำเลย คือ เลือกทำ *ack1* หรือ *ack2* อย่างใดอย่างหนึ่งเท่านั้นหรือไม่ทำทั้ง *ack1* และ *ack2* เสมอ

การทวนสอบการเกิดคำร้องขอจนกว่าตอบรับ หมายถึง การทวนสอบให้แน่ใจว่า เมื่อมีสัญญาณคำร้องขอให้ทำงานบางอย่างเริ่มต้น ต้องให้สัญญาณนั้นมีอยู่ไปเรื่อย ๆ จนกว่าจะได้รับการตอบรับ สัญญาณคำร้องขอจึงจะหยุดได้

ตัวอย่างที่ 7-9: คุณลักษณะเกิดคำร้องขอจนกว่าตอบรับ

กำหนดให้ $[](request \Rightarrow (request \cup ack))$ หมายถึงเมื่อมีสัญญาณคำร้องขอให้ทำงาน *request* ต้องให้สัญญาณ *request* นั้นมีอยู่ไปเรื่อย ๆ จนกว่าจะได้รับการตอบรับ *ack* สัญญาณคำร้องขอ *request* จึงจะหยุดได้ ตัวอย่างเช่นกับประโยคนี้ $[](alert \Rightarrow (alarm \cup safe))$

การทวนสอบในทางปฏิบัติทำได้สองลักษณะ คือ การจำลองการทำงานทีละขั้นตอน (simulation) และการวิเคราะห์แบบหมดจด โดยการทวนสอบแบบแรกจะแสดงให้เห็นการทำงานทีละขั้นตอนไปเรื่อยๆ และอาจจะต้องทำการจำลองหลายครั้ง ทำให้ไม่เป็นลักษณะทวนสอบแบบหมดจด แต่มีข้อดีตรงที่ถ้าเราสามารถคาดเดาได้ว่าส่วนใดน่าจะทำงานผิดพลาดและระบุจุดที่จำลองการทำงานได้จะทำให้ได้ผลลัพธ์อย่างรวดเร็วและใช้จำนวนสถานะไม่มากนัก ในขณะที่การทวนสอบแบบวิเคราะห์แบบหมดจด จะเป็นการทำงานแบบอัตโนมัติและอาจจะใช้เนื้อที่มากและใช้เวลานานจนเกิดการระเบิดของปริภูมิสถานะได้

7.4 โมเดลเช็กกิงตามทฤษฎีออโตมาตา

ในบทนี้เราเน้นเฉพาะโมเดลเช็กกิงสำหรับระบบตรรกศาสตร์เชิงเวลาแบบลำดับเท่านั้น โดยโมเดลเช็กกิงจะต้องมีสองสิ่งที่ใช้ในการทวนสอบ คือ แบบจำลองเชิงรูปนัย M ที่มักจะเขียนหรืออธิบายด้วยออโตมาตา และคุณลักษณะที่เราต้องการทวนสอบ Φ ที่มักเขียนด้วยสูตรเชิงเวลา แล้วนำมาทวนสอบ คือ การหาว่า

$M \vDash \Phi$ เป็นจริงหรือไม่นั่นเอง

และถ้า M เป็นเซตของเส้นทางลำดับการทำงาน ρ แล้ว ดังนั้นการทวนสอบก็คือการหาว่า

$\forall \rho \in M, \rho \vDash \Phi$ เป็นจริงหรือไม่นั่นเอง

ขั้นตอนของโมเดลเช็กกิงตามทฤษฎีออโตมาตา

โมเดลเช็กกิงตามทฤษฎีออโตมาตา [6] คือ การหาว่า $M \vDash \Phi$ เป็นจริงหรือไม่ และถ้ากำหนดให้ว่า M คือ เซตของเส้นทางลำดับของโมเดลของระบบ และ $sequence_satisfying(\Phi)$ คือเซตของเส้นทางลำดับของโมเดลที่มีคุณลักษณะที่ต้องการ ความหมายจึงเปลี่ยนไปเป็นว่าเราต้องการหาว่า

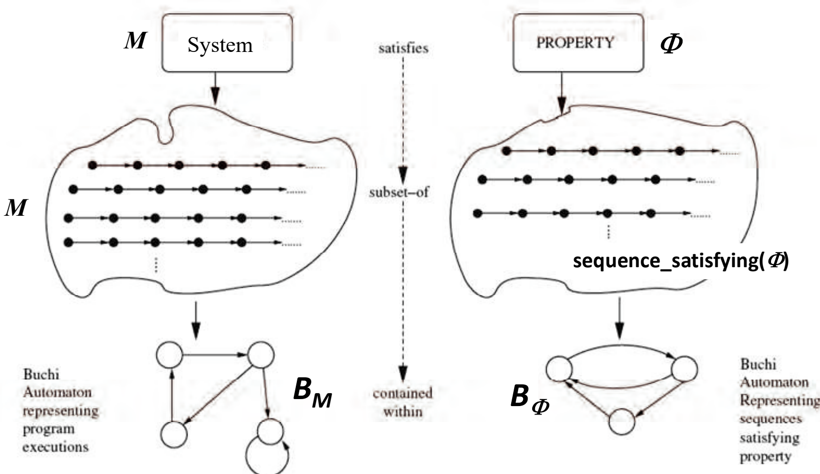
$M \subseteq sequence_satisfying(\Phi)$ เป็นจริงหรือไม่นั่นเอง

ถ้าระบบของเราเป็นระบบที่ทำงานแบบไม่สิ้นสุดแล้ว เราควรเลือกบูชีออโตมาตามาใช้ทำหน้าที่เป็นตัวยอมรับในการสร้างแบบจำลอง M ใช้ชื่อเรียกว่า $accepted_by\ (B_M)$ และในขณะเดียวกันเพื่อให้เป็นรูปแบบเดียวกัน เราควรใช้บูชีออโตมาตาเป็นตัวยอมรับในการสร้าง $sequence_satisfying(\Phi)$ ใช้ชื่อเรียกว่า $accepted_by\ (B_\Phi)$ และเปลี่ยนมาการพิจารณาแทนว่า (แสดงในรูปที่ 7.5)

$accepted_by\ (B_M) \subseteq accepted_by\ (B_\Phi)$ เป็นจริงหรือไม่ และเราจะเปลี่ยนไปเป็นสมการ $accepted_by\ (B_M) \cap \neg accepted_by\ (B_\Phi) = \emptyset$ แทนว่าเป็นจริงหรือไม่ แต่เราทราบว่า $\neg accepted_by\ (B_\Phi) = accepted_by\ (B_{\neg\Phi})$ เราจึงเขียนใหม่ว่า $accepted_by\ (B_M) \cap accepted_by\ (B_{\neg\Phi}) = \emptyset$ ว่าจะเป็นจริงหรือไม่ แต่เรารู้ว่า $accepted_by\ (B_M) \cap accepted_by\ (B_{\neg\Phi})$ หมายถึง $accepted_by\ (B_M \times B_{\neg\Phi})$ เราจึงเขียนใหม่ว่า

$accepted_by\ (B_M \times B_{\neg\Phi}) = \emptyset$ เป็นจริงหรือไม่ จึงเป็นที่มาของการทำโมเดลเช็กกิงตามทฤษฎีออโตมาตาว่า ถ้ากำหนดให้ระบบจำลอง M และคุณลักษณะที่ต้องการทวนสอบ Φ แล้ว เราต้องมีขั้นตอน ดังนี้

- หานิเสธของคุณลักษณะ Φ คือ ¬Φ
- หาบูชีออโตมาตาของ M คือ B_M
- หาบูชีออโตมาตาของ ¬Φ คือ $B_{\neg\Phi}$
- ให้ $C = accepted_by\ (B_M \times B_{\neg\Phi})$ โดยที่ C เป็นเซตตัวอย่างค้าน
- หาว่า $C = \varnothing$ จริงหรือไม่ ถ้าเป็นจริงแสดงว่าทวนสอบผ่าน หรือเรียกได้ว่าเซตของตัวอย่างค้านเป็นเซตว่าง แต่ถ้าเป็นเท็จแสดงว่าทวนสอบไม่ผ่าน เกิดข้อผิดพลาด โดยดูตัวอย่างค้านได้ที่ C



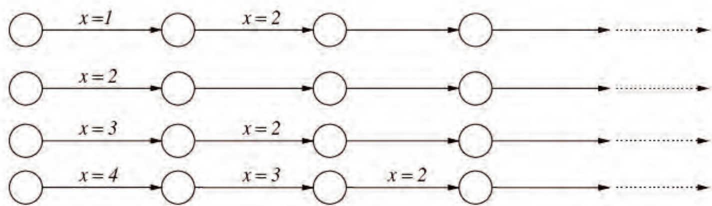
รูปที่ 7.5 ขั้นตอนของโมเดลเช็กกิงตามทฤษฎีออโตมาตา [6]

ตัวอย่างที่ 7-10: โมเดลเช็กกิงด้วยทฤษฎีออโตมาตาอย่างง่าย

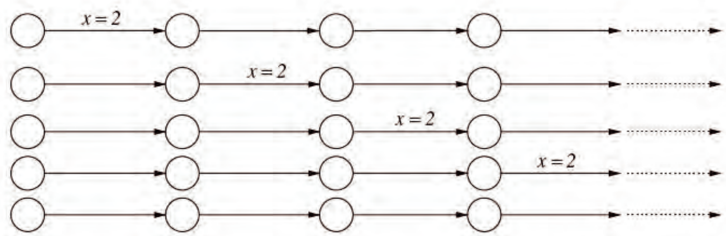
กำหนดให้ระบบ M ทำงานตามโปรแกรมดังต่อไปนี้ เพื่อทวนสอบคุณลักษณะ Φ = *(<>(x=2)*

```
int x = random(1,4);
while (x != 2)
do
  if
    :: (x < 2) -> x:=x+1;
    :: (x > 2) -> x:=x-1;
  fi
do
```

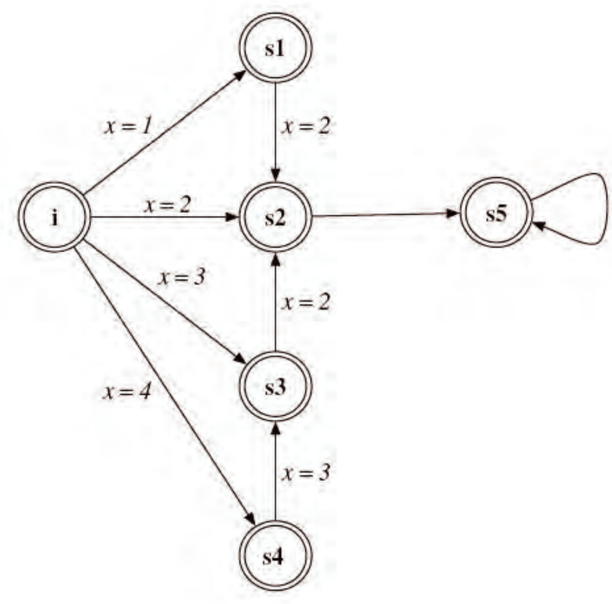
รูปที่ 7.6 แสดงเซตของเส้นทางแบบลำดับ *M* และรูปที่ 7.7 แสดงเซตของเส้นทางแบบลำดับของคุณสมบัติ *sequence_satisfying(<>(x=2))*



รูปที่ 7.6 เซตของเส้นทางแบบลำดับ *M [6]*



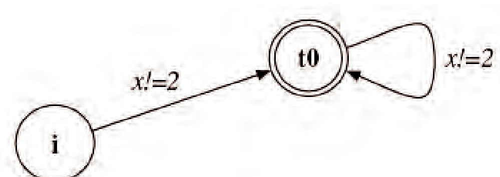
รูปที่ 7.7 เซตของเส้นทางแบบลำดับของคุณสมบัติ sequence_satisfying(<>(x=2)) [6]



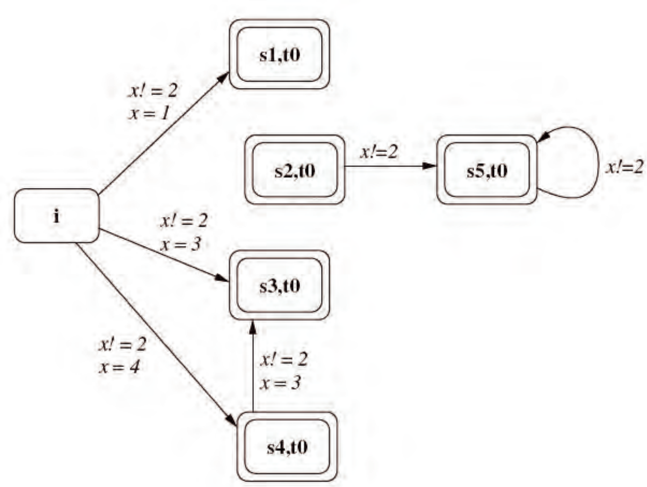
รูปที่ 7.8 บูชีออโตมาตา B_M *[6]*

การหาว่า $M \subseteq sequence_satisfying(<>(x=2))$ ทำได้ยาก ดังนั้นเราจึงเขียนบูชีออโตมาตา B_M แสดงในรูปที่ 7.8 และหานิเสธของ $<>(x=2)$ ซึ่งก็คือ $[](x \neq 2)$ เราจึงเขียนบูชีออโตมาตา $B_{[](x \neq 2)}$ แสดงในรูปที่ 7.9

เราหาผลคูณบูชีออโตมาตา $(B_M \times B_{[](x \neq 2)})$ โดยแสดงในรูปที่ 7.10 และพบว่า $accepted_by\ (B_M \times B_{[](x \neq 2)})$ เป็นเซตว่าง กล่าวคือไม่มีเส้นทางใดเป็นไปตามนั้นได้เลย ทำให้เราสามารถสรุปได้ว่า ทวนสอบผ่าน



รูปที่ 7.9 บูชีออโตมาตา $B_{[](x \neq 2)}$ [6]

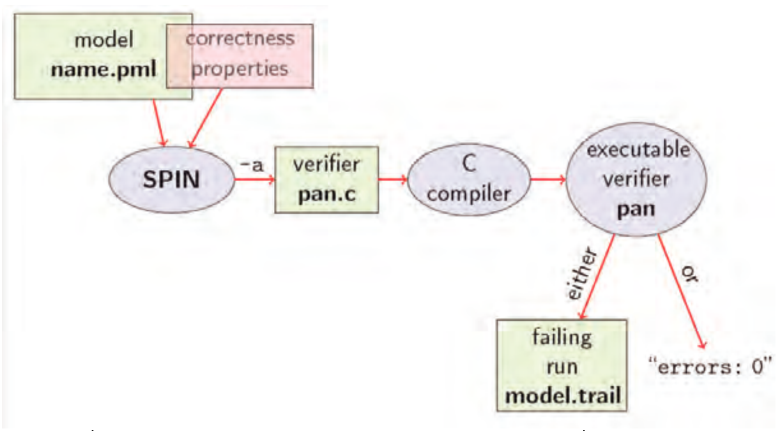


รูปที่ 7.10 บูชีออโตมาตา $(B_M \times B_{[](x \neq 2)})$ [6]

เครื่องมือสปินที่ใช้โมเดลเช็กกิงตามทฤษฎีออโตมาตา

ต่อไปนี้เป็นตัวอย่างการทวนสอบเชิงรูปนัยโดยอ้างอิงกระบวนการทวนสอบแบบมาตรฐานที่กล่าวมาแล้วนั้น โดยเลือกใช้เครื่องมือสปินและภาษาโพรเมลา กระบวนการเกิดขึ้นโดยการสร้างแบบจำลองระบบที่สนใจโดยใช้ภาษาโพรเมลาและตั้งชื่อระบบลงท้ายด้วย *.pml* และกำหนดคุณลักษณะที่ต้องการทวนสอบ

ไว้ด้วยการเขียนสูตรเชิงเวลา เลือกนำเข้าสปินได้สองลักษณะ คือ แยกแฟ้มคุณลักษณะหรือรวมคุณลักษณะในแฟ้มแบบจำลองระบบ โพรเมลาจะมีคำสั่ง *ltl* ในการกำหนดคุณลักษณะที่ต้องการทวนสอบให้ไว้โดยสามารถใช้ตัวดำเนินการเชิงเวลาได้เช่นกัน จากรูปที่ 7.11 มุมซ้ายบนแสดงถึงการนำแฟ้มแบบจำลองระบบและคุณลักษณะที่ต้องการทวนสอบส่งเข้าไปสู่เครื่องมือสปิน จากนั้นเครื่องมือสปินจะแปลงให้เป็นโปรแกรมภาษาซี และผ่านตัวแปลภาษาซีนำไปเป็นโปรแกรมที่ทำงานได้แบบ *.exe* และทำการสั่งให้ทำงานจนได้ผลลัพธ์ว่าไม่มีข้อผิดพลาดใด ๆ หรือมีข้อผิดพลาดและเก็บตัวอย่างค้านไว้ในแฟ้ม *.trail* ตามที่แสดงในรูปที่ 7.11



รูปที่ 7.11 กระบวนการทวนสอบด้วยโมเดลเช็กกิงด้วยเครื่องมือสปิน [7]

การกำหนดคุณลักษณะที่ต้องการทวนสอบของเครื่องมือสปิน กรณีที่ต้องการระบุไว้ในแฟ้มแบบจำลอง *.pml* นั้นทำได้โดยการกำหนดคำสั่ง *assert(expr)* โดยคำสั่งนี้จะทำหน้าที่ตรวจสอบว่านิพจน์ *expr* ในวงเล็บมีค่าจริงหรือไม่ ถ้ามีค่าจริงจะปล่อยให้ระบบทำงานต่อไป แต่ถ้ามีค่าเท็จจะหยุดการทำงานทันที ดังนั้นเรามักจะใช้คำสั่ง *assert(expr)* ไว้ในการทวนสอบเงื่อนไขค่ายืนยงของข้อมูลในระบบ โดยกำหนดให้ *expr* มีเงื่อนไขค่ายืนยงนั่นเอง

การทวนสอบกรณีที่ระบบทำงานแบบไม่สิ้นสุดก็สามารถจัดการได้ในเครื่องมือสปินด้วยเช่นกัน โดยใช้เทคนิคของการกำหนดลาเบลกำกับประโยคเรียกว่า เมตาลาเบล (meta label) ในภาษาโพรเมลามีลาเบลให้กำหนดได้สามแบบคือ *"End Label"*, *"Accept Label"*, และ *"Progress Label"* อย่างไรก็ตามเครื่องมือสปินก็มีความสามารถในการทวนสอบกรณีการเข้าถึงได้ทุกสถานะ

และการจบการทำงานได้จริงของระบบโดยอัตโนมัติเช่นกัน หรือถ้าต้องการใช้คำสั่ง *ltl* ก็สามารถสั่งต่อท้ายโพรเมลาได้ในแฟ้ม *.pml* นี้

สำหรับกรณีที่ต้องการเขียนคุณลักษณะที่ต้องการทวนสอบแยกแฟ้มไว้ก็สามารถสร้างแฟ้มที่เรียกว่า *"Never Claim"* แยกไว้เพื่อนำมาใช้ในการทวนสอบภายหลังได้ ข้อดีก็คือ เงื่อนไขในการทวนสอบจะไม่ปะปนกับโพรเมลาที่เป็นแบบจำลองระบบทำให้มีระเบียบเรียบร้อยและดูแลง่ายกว่า

ตัวอย่างที่ 7-11 การทวนสอบการจบการทำงานได้ของระบบด้วยเครื่องมือสปิน

กำหนดให้แบบจำลองระบบ *Counter* ที่เขียนด้วยภาษาโพรเมลา โดยระบบนี้จะมีตัวแปรชื่อ *count* ที่ทำหน้าที่เป็นตัวนับ โดยมีเส้นโยงใยการทำงานที่ชื่อ *Counter()* ซึ่งจะวนลูปนับค่าตัวแปร *count* โดยเริ่มที่ 0 ภายใน *do..od* ลูปจะทำงานแบบไม่เชิงกำหนดด้วยสามคำสั่ง กล่าวคือ ระบบจะสุ่มว่าทำคำสั่งใดก็ได้ ดังนั้นค่าตัวแปร *count* จะถูกบวกเพิ่ม หรือลบออก ได้เสมอ หรืออาจจะได้รับการตรวจเงื่อนไขว่ามีค่าเป็น *0* หรือไม่เพื่อจะได้หยุดทำงาน

```
int count;
proctype counter()
{       do
          :: count = count + 1
          :: count = count - 1
          :: (count == 0) -> break
        od
}
```

เราสามารถทวนสอบว่าการทำงานของ *Counter()* นั้นจะสิ้นสุดหรือจบการทำงานได้เสมอหรือไม่ ในเครื่องมือสปินสามารถทวนสอบเรื่องการทำงานจบได้หรือไม่อย่างอัตโนมัติ โดยจากการทวนสอบระบบ *Counter()* นี้จะพบว่ามีโอกาสที่ระบบจะทำงานแบบไม่สิ้นสุดโดยหมายถึงจะวนลูปไปไม่สิ้นสุดนั่นเอง เนื่องจากคำสั่งการตรวจค่า *(count==0) -> break* ไม่ได้รับการเลือกมาทำงานในจังหวะที่ค่าเป็น *0* ได้เลย

การแก้ไขปัญหาการจบการทำงานสามารถดัดแปลงโปรแกรมโพรเมลาให้มาเป็น *safercounter()* แทนซึ่งจะทำให้โอกาสที่ *count* เป็น *0* ได้รับการตรวจสอบได้แน่นอน

```
proctype safercounter()
{       do
          :: (count != 0) -> count = count+1
          :: (count != 0) -> count = count-1
          :: (count == 0) -> goto done
        od;
    done: skip
}
```

ตัวอย่างที่ 7-12: การทวนสอบการเข้าถึงได้ทุกสถานะของระบบด้วยเครื่องมือสปิน

กำหนดให้ระบบเป็นการทำงานของสายโยงใยชื่อ *MyProcess()* และ *YourProcess()* โดยที่มีตัวแปรส่วนกลาง (global variable) ชื่อ *state* สำหรับใช้ร่วมกันเป็นการประสานระหว่างสายโยงใย (inter-thread synchronization) โดยที่ทั้ง *MyProcess()* และ *YourProcess()* จะตรวจสอบค่าตัวแปร *state* ว่ามีค่าเท่ากับ *1* ก่อนลงมือเปลี่ยนค่าตัวแปร *state* ใหม่อีกครับ โดยทำการเพิ่ม หรือลดค่าในตัวแปร *state* นั่นเอง ในขั้นตอนการเปลี่ยนค่าตัวแปร *state* ใหม่จะใช้สามคำสั่งเพื่อจำลองว่าการเปลี่ยนค่าตัวแปร *state* มีกระบวนการย่อยอยู่ด้วยเพื่อให้เห็นเหตุการณ์ที่มีการแย่งทรัพยากรได้ชัดเจนขึ้นในการทำงานของระบบที่ทำงานพร้อมกัน และเกิดเหตุการณ์ที่เรียกว่าการแทรกสลับของคำสั่งในการทำงานแบบไม่เชิงกำหนดของโพรเมลาข้างล่างนี้

```
int state = 1;
proctype MyProcess()
{  int tmp;
   (state==1) -> tmp = state;
   tmp = tmp+1;
   state = tmp
}
proctype YourProcess()
{    int tmp;
    (state==1) -> tmp = state;
    tmp = tmp-1;
    state = tmp
}
init { run MyProcess(); run YourProcess() }
```

เราต้องการให้เครื่องมือสปินทวนสอบการเข้าถึงได้ทุกสถานะของระบบหรือไม่ ก็คือการที่สายโยงใยทำงานได้ครบทุกคำสั่งหรือไม่ มีการติดตายหรือไม่นั่นเอง และพบว่าเกิดกรณีการติดตายของ *MyProcess()* หรือ *YourProcess()*

ขึ้นได้ และเครื่องมือสปินได้แสดงตัวอย่างค้านออกมาให้ทราบว่าเส้นทางการทำงานใดที่เกิดการติดตายของแต่ละสายโยงใยทั้งสอง

ตัวอย่างการติดตายที่เกิดขึ้น คือ กรณีที่ *MyProcess()* ทำงานไปได้เร็วกว่าโดยมีการเพิ่มค่า *state* เป็น *2* ได้ก่อน ทำให้ *YourProcess()* ที่เพิ่งเริ่มทำงานเกิดการติดตาย หยุดรอค่าในตัวแปร *state* ต้องมีค่าเท่ากับ *1* ก่อนจึงจะทำงานต่อได้ ในทางกลับกันถ้า *YourProcess()* ทำงานได้เร็วกว่าก็จะทำให้ *MyProcess()* ติดตายได้ด้วยเหตุผลเดียวกัน

ตัวอย่างที่ 7-13 การทวนสอบการเข้าถึงได้ทุกสถานะของระบบด้วยเครื่องมือสปิน

กำหนดให้ระบบทำงานด้วยสายโยงใย *P()_id1* และ *P()_id2* และ *WaitForFinish()* โดยมีตัวแปรส่วนกลางคือ n และ finish เพื่อใช้ในการประสานระหว่างกัน *P()_id1* และ *P()_id2* จะทำงานเหมือนกันทุกอย่างเนื่องจากเกิดจาก *protype P()* เดียวกันแต่แยกกันออกเป็นสองสายโยงใยด้วยส่วนคำสั่ง *“active [2]”* ซึ่งจะวนนับค่าตัวแปรเฉพาะที่ *counter* ของตนจากค่า *0* ถึง *10* ระหว่างที่นับก็จะทำการบวกค่าตัวแปรส่วนกลาง *n* ไปเรื่อยๆ จนเมื่อทำงานเสร็จก็จะเพิ่มค่าตัวแปรส่วนกลาง *finish* อีกหนึ่ง ทั้งสองสายโยงใยทำงานแบบพร้อมกันทำให้เกิดการสอดแทรกของคำสั่งแบบเชิงไม่กำหนดด้วยทำให้มีความกังวลว่าอาจจะเกิดปัญหาติดตายของการทำงานได้

```
byte n = 0, finish = 0;
active [2] proctype P() {
        byte register1, counter = 0;
        do :: counter == 10 -> break
            :: else ->
                register1 = n;
                register1++;
                n = register1;
                counter++
        od;
        finish++
}
active proctype WaitForFinish() {
        finish == 2;
        printf(“n = %d\n”, n)
}
```

เครื่องมือสปินใช้ทวนสอบการติดตายหรือการทวนสอบการเข้าถึงได้ทุกสถานะโดยอัตโนมัติได้ พบว่าทวนสอบได้ไม่เกิดการติดตาย กล่าวคือมีการเข้าถึงได้ทุกสถานะ ทำให้ทวนสอบผ่าน และถ้าต้องการทราบว่าค่าตัวแปร *n* เมื่อจบการทำงานระบบจะมีค่าอยู่ช่วงใด และจากการคาดเดาค่า n น่าจะมีค่าระหว่าง *10* ถึง *20* เนื่องจากทั้ง *P()_id1* และ *P()_id2* ทำการวนซ้ำบวกค่า *n* ไว้สิบรอบ

ดังนั้น เราลองใช้การทวนสอบโดยเขียนสูตรเชิงเวลาในการทวนสอบด้วยคำสั่งโพรเมลาว่า

ltl p1 { [](finish==2 -> (10<=n && n<=20)) }

โดยหมายถึงถ้าระบบหยุดทำงานแล้วคือ *finish==2* แล้วค่าตัวแปร *n* จะอยู่ในช่วงนั้นเสมอหรือไม่ ผลปรากฏว่าทวนสอบไม่ผ่าน และตัวอย่างค้านที่ได้ก็ฟ้องว่าตัวแปร *n* มีค่าน้อยกว่า *10* ได้ ดังนั้นเราจึงลองเปลี่ยนการตรวจค่าตัวแปร *n* ให้ลดลงไปแล้วทำการทวนสอบต่อไปอีก ในที่สุดเราพบว่า *n* มีค่าอยู่ระหว่าง *2* ถึง *20* ดังนั้นสูตรเชิงเวลาที่เขียนด้วยโพรเมลาที่เราใช้แล้วทวนสอบผ่าน คือ

ltl p1 {[](finish==2 -> (2<=n && n<=20)) }

ตัวอย่างที่ 7-14 การทวนสอบระบบสัญญาณไฟจราจร

กำหนดให้ระบบสัญญาณไฟจราจรที่มีสัญญาณไฟสามดวงคือ ไฟแดง ไฟเหลือง ไฟเขียว ในการทำงานของระบบเริ่มต้นด้วยสัญญาณไฟแดง จากนั้นก็เปลี่ยนไปเป็นไฟเขียว และไฟเหลือง แล้ววนกลับมาไฟแดงอีกครั้งวนซ้ำไม่สิ้นสุด เราสร้างแบบจำลองด้วยภาษาโพรเมลาได้ดังต่อไปนี้

```
int redLight = 0;
int yellowLight = 0;
int greenLight = 0;

proctype simpleTrafficLight()
{
   //Start with redLight
   redLight = 1;
   do
     ::(redLight ==1) ->
               greenLight =1;
               yellowLight = 0;
               redLight = 0;
     ::(greenLight ==1) ->
               redLight = 0;
               yellowLight = 1;
               greenLight =0;
     ::(yellowLight == 1) ->
               redLight = 1;
               yellowLight = 0;
               greenLight = 0;
   od
}

init {
  run simpleTrafficLight()
}
```

โพรเมลานี้ทำงานด้วยสายโยงใยเดียวชื่อ *simpleTrafficLight()* ซึ่งมีตัวแปรชื่อ *redLight, greenLight, yellowLight* แทนสัญญาณไฟสีแดง สีเขียว และสีเหลืองตามลำดับ โดยถ้าค่าในตัวแปรมีค่าเป็น 1 แสดงว่าไฟสว่าง และถ้ามีค่าเป็น 0 แสดงว่าไฟดับ ระบบเริ่มต้นด้วยการเปิดให้สัญญาณไฟแดงสว่างก่อน จากนั้นก็เป็นการทำงานแบบวนซ้ำด้วยคำสั่ง *do..od* เพื่อทำงานเปิดปิดไฟสัญญาณแบบไม่สิ้นสุด ในการทำงานวนซ้ำนั้น จะเริ่มจากการตรวจสอบสามเงื่อนไข *(redLight==1), (greenlight==1), (yellowLight==1)* โดยที่เงื่อนไขแรกคือ *(redLight==1)* หมายถึงค่าตัวแปร *redLight* เท่ากับ 1 หรือไม่ ถ้ามีค่าเป็น 1 ให้ทำการเปลี่ยนค่าในตัวแปรไฟสีเขียวให้ติดสว่างแทนไฟสีแดง และเงื่อนไขอื่นอีกสองเงื่อนไขที่เหลือก็เป็นไปในทำนองเดียวกัน โดยให้เปลี่ยนค่าตัวแปรไฟสีถัดไปให้ติดสว่างตามลำดับ เมื่อพิจารณาแล้วไม่น่าจะมีปัญหาอะไร หลังจากเราทวนสอบด้วยคุณลักษณะแบบไม่เกิดร่วมกันของสัญญาณไฟแดงและไฟเขียวด้วยสูตรเชิงเวลา ดังนี้

ltl p1 { []!(redLight==1 && greenlight==1) }

จะพบว่าทวนสอบไม่ผ่าน เพราะเกิดกรณีสัญญาณไฟแดงและไฟเขียวสว่างพร้อมกัน ตัวอย่างค้านทำให้เราทราบว่าเกิดจากเส้นทางการทำงานใด และพบว่าข้อผิดพลาดเกิดขึ้นในตอนสัญญาณไฟแดงเปลี่ยนมาเป็นสัญญาณไฟเขียว เราสังเกตว่า ระบบทำงานโดยการเปิดสัญญาณไฟเขียวก่อนแล้วค่อยปิดสัญญาณไฟแดง ดังนั้นเราจึงสามารถทำการแก้ไขลำดับการทำงานเสียใหม่ ได้เป็นแบบจำลองที่ทำงานได้ถูกต้องโดยทวนสอบเรื่องการไม่เกิดร่วมกันของสัญญาณไฟแดงและไฟเขียวได้แสดงต่อไปนี้

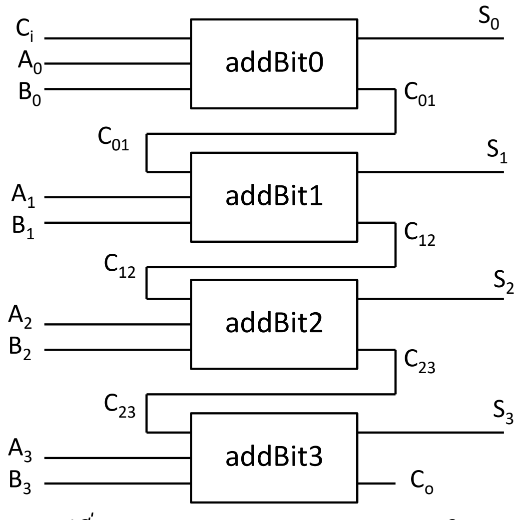
```
int redLight = 0;
int yellowLight = 0;
int greenLight = 0;

proctype simpleTrafficLight()
{
   //Start with redLight
   redLight = 1;
   do
     ::(redLight ==1) ->
                 redLight = 0;
                 yellowLight = 0;
                 greenLight =1;
     ::(greenLight ==1) ->
                 redLight = 0;
                 yellowLight = 1;
                 greenLight =0;
     ::(yellowLight == 1) ->
                 redLight = 1;
```

```
                        yellowLight = 0;
                        greenLight =0;
      od
    }

    init {
      run simpleTrafficLight()
    }
```

ทั้งนี้เราสามารถทวนสอบคุณลักษณะอื่นของระบบนี้ได้ เช่น สัญญาณไฟสีเขียวจะเปิดสว่างได้ไหมในที่สุด <>(*greenlight==1*) เป็นต้น

ตัวอย่างที่ 7-15 การทวนสอบระบบวงจรบวกแบบ 4 บิต

กำหนดให้ระบบวงจรบวกแบบ *4* บิต ที่เกิดจากวงจรบวกเลข *1* บิตมาประกอบกัน โดยรับค่าเลขฐานสองเข้ามาสองค่าคือ A และ B โดยการป้อนสัญญาณเลข *0* และ *1* เข้าที่ขานำเข้า $A_3A_2A_1A_0$ และ $B_3B_2B_1B_0$ ตามลำดับ โดยมีการนำเข้าบิตที่ทดมาด้วย คือ ขานำเข้า C_i และผลลัพธ์การบวกแสดงไปที่ S โดยการส่งออกสัญญาณเลข *0* และ *1* ที่ $S_3S_2S_1S_0$ พร้อมทั้งส่งออกค่าทดออกที่ C_o แสดงวงจรการเชื่อมต่อในรูปที่ 7.12



รูปที่ 7.12 แผนภาพระบบวงจรบวกแบบ 4 บิต

โพรเมลาที่อธิบายพฤติกรรมการทำงานของวงจรบวกเลข 4 บิตดังกล่าวแสดงไว้ข้างล่างนี้

```
int A0= 0, A1=0, A2=0, A3=0, Ci=0 , B0=0, B1=0, B2=0, B3= 0, Co =
0, S3=0, S2=0, S1=0, S0=0, C01=0, C12=0, C23=0;

active proctype add(){
```

```
int a, b, s;

   addBit0: if
               :: (A0 ==0 && B0 == 0 && Ci == 0) ->
                     atomic {C01 = 0; S0 = 0; goto addBit1;}
               :: (A0 ==0 && B0 == 0 && Ci == 1) ->
                     atomic {C01 = 0; S0 = 1; goto addBit1;}
               :: (A0 ==0 && B0 == 1 && Ci == 0) ->
                     atomic {C01 = 0; S0 = 1; goto addBit1;}
               :: (A0 ==0 && B0 == 1 && Ci == 1) ->
                     atomic { C01 = 1; S0 = 0; goto addBit1;}
               :: (A0 ==1 && B0 == 0 && Ci == 0) ->
                     atomic { C01 = 0; S0 = 1; goto addBit1;}
               :: (A0 ==1 && B0 == 0 && Ci == 1) ->
                     atomic { C01 = 1; S0 = 0; goto addBit1;}
               :: (A0 ==1 && B0 == 1 && Ci == 0) ->
                     atomic { C01= 1; S0 = 0; goto addBit1;}
               :: (A0 ==1 && B0 == 1 && Ci == 1) ->
                     atomic {C01 = 1; S0 = 1; goto addBit1;}
            fi ;

   addBit1: if
               :: (A1 ==0 && B1 == 0 && C01 == 0) ->
                     atomic {C12 = 0; S1 = 0; goto addBit2;}
               :: (A1 ==0 && B1 == 0 && C01 == 1) ->
                     atomic { C12 = 0; S1= 1; goto addBit2;}
               :: (A1 ==0 && B1 == 1 && C01 == 0) ->
                     atomic { C12 = 0; S1 = 1; goto addBit2;}
               :: (A1 ==0 && B1 == 1 && C01 == 1) ->
                     atomic { C12 = 1; S1 = 0; goto addBit2;}
               :: (A1 ==1 && B1 == 0 && C01 == 0) ->
                     atomic { C12= 0; S1 = 1; goto addBit2;}
               :: (A1 ==1 && B1 == 0 && C01 == 1) ->
                     atomic { C12 = 1; S1 = 0; goto addBit2;}
               :: (A1 ==1 && B1 == 1 && C01 == 0) ->
                     atomic { C12 = 1; S1 = 0; goto addBit2;}
               :: (A1 ==1 && B1 == 1 && C01 == 1) ->
                     atomic { C12 = 1; S1 = 1; goto addBit2;}
            fi ;

   addBit2: if
               :: (A2 ==0 && B2 == 0 && C12 == 0) ->
                     atomic { C23 = 0; S2 = 0; goto addBit3;}
               :: (A2 ==0 && B2 == 0 && C12 == 1) ->
                     atomic { C23 = 0; S2 = 1; goto addBit3;}
               :: (A2 ==0 && B2 == 1 && C12== 0) ->
                     atomic { C23 = 0; S2 = 1; goto addBit3;}
               :: (A2 ==0 && B2 == 1 && C12== 1) ->
                     atomic { C23 = 1; S2 = 0; goto addBit3;}
```

```
        :: (A2 ==1 && B2 == 0 && C12== 0) ->
                atomic { C23 = 0; S2 = 1; goto addBit3;}
        :: (A2 ==1 && B2 == 0 && C12== 1) ->
                atomic { C23= 1; S2 = 0; goto addBit3;}
        :: (A2 ==1 && B2 == 1 && C12 == 0) ->
                atomic { C23= 1; S2 = 0; goto addBit3;}
        :: (A2 ==1 && B2 == 1 && C12== 1) ->
                atomic {C23 = 1; S2 = 1; goto addBit3;}
      fi ;

addBit3: if
        :: (A3 ==0 && B3 == 0 && C23== 0) ->
                atomic {Co = 0; S3 = 0; goto wait;}
        :: (A3 ==0 && B3 == 0 && C23== 1) ->
                atomic {Co = 0; S3 = 1; goto wait;}
        :: (A3 ==0 && B3 == 1 && C23 == 0) ->
                atomic {Co = 0; S3 = 1; goto wait;}
        :: (A3 ==0 && B3 == 1 && C23== 1) ->
                atomic {Co = 1; S3 = 0; goto wait;}
        :: (A3 ==1 && B3 == 0 && C23 == 0) ->
                atomic {Co = 0; S3 = 1; goto wait;}
        :: (A3 ==1 && B3 == 0 && C23 == 1) ->
                atomic {Co = 1; S3 = 0; goto wait;}
        :: (A3 ==1 && B3 == 1 && C23 == 0) ->
                atomic {Co = 1; S3 = 0; goto wait;}
        :: (A3 ==1 && B3 == 1 && C23 == 1) ->
                atomic {Co = 1; S3 = 1; goto wait;}
      fi ;

wait: a = A3 * 8 + A2 * 4 + A1 * 2 + A0 * 1;
      b = B3 * 8 + B2 * 4 + B1 * 2 + B0 * 1;
      s = Co * 16 + S3 * 8 + S2 * 4 + S1 * 2 + S0 * 1;
      assert((a+b+Ci) == s);
      goto setInput;

setInput: if
        :: skip; atomic{ A3 = 0; }
        :: skip; atomic{ A3 = 1; }
      fi;
      if
        :: skip; atomic{ A2 = 0; }
        :: skip; atomic{ A2 = 1; }
      fi;
      if
        :: skip; atomic{ A1 = 0; }
        :: skip; atomic{ A1 = 1; }
      fi;
      if
        :: skip; atomic{ A0 = 0; }
        :: skip; atomic{ A0 = 1; }
```

```
            fi;
            if
                :: skip; atomic{ B3 = 0; }
                :: skip; atomic{ B3 = 1; }
            fi;


            if
                :: skip; atomic{ B2 = 0; }
                :: skip; atomic{ B2 = 1; }
            fi;
            if
                :: skip; atomic{ B1 = 0; }
                :: skip; atomic{ B1 = 1; }
            fi;
            if
                :: skip; atomic{ B0 = 0; }
                :: skip; atomic{ B0 = 1; }
            fi;
            if
                :: skip; atomic{ Ci = 0; goto addBit0;}
                :: skip; atomic{ Ci = 1; goto addBit0;}
            fi;
}
```

โพรเมลาของระบบวงจรบวกเลข *4* บิตนี้แบ่งการทำงานออกเป็นสองส่วน คือ ส่วนการทำการบวกตัวเลขฐานสองสองตัวเข้าด้วยกัน และส่วนที่สองคือ ส่วนที่ใช้ทวนสอบโดยการใช้คำสั่ง *assert((a+b+Ci) == s)* เพื่อใช้ทวนสอบผลการบวกว่ามีความถูกต้องหรือไม่ ถ้าไม่ถูกต้องโปรแกรมจะหยุดทำงานทันที และส่วนที่ช่วยในการสุ่มตัวเลขมาบวกก็คือส่วนท้ายสุด ที่มี *Label* กำกับว่า *setInput:* ที่ตามด้วยคำสั่ง *if..fi* จะทำหน้าที่ในการสุ่มค่าต่าง ๆ ให้กับแต่ละขาข้อมูลนำเข้าของส่วนวงจรบวก

ส่วนการทำการบวกตัวเลข จัดแบ่งเป็นสัดส่วนตามชื่อ *Label* ที่กำกับ คือ *addBit0:, addBit1:, addBit2:, addBit3:* สำหรับการบวกบิตที่ *0* บิตที่ *1* บิตที่ *2* และบิตที่ ตามลำดับ ทั้งนี้ในการบวกบิต *0* จะมีการนำบิตทดนำเข้ามารวมด้วย และการบวกบิต *3* จะมีค่าบิตทดส่งออกแสดงให้ด้วยเช่นกัน

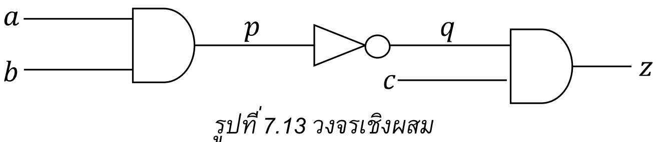
ตัวอย่างที่ 7-16 การทวนสอบแบบการตรวจสอบความสมมูลของวงจรเชิงผสม

กำหนดให้แผนภาพเค้าร่างวงจรเชิงผสมที่ออกแบบมาเชื่อมต่อกันตามรูปที่ 7.13 ซึ่งการออกแบบเขียนเป็นพีชคณิตบูลีนดังนี้

$p = (a \wedge b)$, $q = (\neg p)$, $z = (q \wedge c)$

และคุณสมบัติที่ต้องการทวนสอบความสมมูล คือ

$z = (\neg a \vee \neg b) \wedge c$



รูปที่ 7.13 วงจรเชิงผสม

เราสร้างโพรเมลาเพื่อตรวจความสมมูลของทั้งสองชุดพีชคณิตบูลีนได้ดังโปรแกรมที่แสดง ดังนี้

```
bool a,b,c,z;
inline InputGenerator()
{
      if
         ::a=0;
         ::a=1;
       fi;
      if
          ::b=0;
          ::b=1;
      fi;
      if
          ::c=0;
          ::c=1;
      fi;
}
proctype Imp()
{
   bool p,q;
  do
       ::  InputGenerator();
              printf("Input Now --- a=%d, b=%d, c=%d \n", a, b, c);
              atomic { p=a&&b; q=!p; z=q&&c; };
              assert((!(a&&b)&&c)==z);
   od;
}

init { run Imp() }
```

โพรเมลามีตัวแปรชื่อ *a, b, c* แทนค่าของขานำเข้าข้อมูลของวงจรที่ตำแหน่ง *a, b, c* ตามลำดับ โปรแกรมแบ่งเป็นสองส่วน คือ ส่วนการสุ่มค่านำเข้า *InputGenerator()* ซึ่งเขียนขึ้นเพื่อสุ่มค่าตัวแปรเพื่อป้อนให้กับวงจร ผ่านทางค่าในตัวแปร *a, b, c* และส่วนที่สอง คือ ส่วนการตรวจสอบความสมมูลโดยการใช้คำสั่ง *assert()* โดยข้อดีคือ โพรเมลามีชนิดตัวแปรแบบ *Boolean* และตัวดำเนินการตรรกะที่ใช้กับตัวแปรบูลีน ซึ่งนำมาใช้ได้ทันทีในการหาค่าพีชคณิตได้ทันที ผลการทวนสอบพบว่าชุดพีชคณิตทั้งสองมีความสมมูลกัน

7.5 งานวิจัยสนับสนุนการทวนสอบด้วยโมเดลเช็กกิง

การทวนสอบด้วยโมเดลเช็กกิงต้องมีสองสิ่งที่จำเป็นในเบื้องต้นก่อน คือ แบบจำลองเชิงรูปนัยระบบและคุณลักษณะเชิงเวลา โดยปกติแบบจำลองเชิงรูปนัยมักจะต้องอยู่ในรูปแบบแผนภาพสถานะ เช่น ออโตมาตา โครงสร้างคริปกี เป็นต้น ซึ่งการเขียนออโตมาตาหรือโครงสร้างคริปกีเพื่ออธิบายระบบซอฟต์แวร์หรือฮาร์ดแวร์ที่มีขนาดใหญ่และซับซ้อนย่อมทำได้ยาก การมองภาพรวมก็ทำได้ยาก ไม่สามารถแบ่งงานกันในทีมงานได้ชัดเจน แผนภาพดังกล่าวจัดการให้มีการแบ่งชั้นเชิงลำดับ (hierarchical layer) ได้ยาก ดังนั้น เพื่อให้ทีมออกแบบทำงานร่วมกันได้ จึงเลี่ยงมาใช้แผนภาพที่คุ้นเคยกว่า และอาจจะเป็นแผนภาพการออกแบบที่ใช้กันอยู่เป็นปกติปัจจุบัน เช่น แผนภาพยูเอ็มแอล เพทริเน็ต เป็นต้น

ต่อไปนี้เป็นส่วนหนึ่งของงานวิจัยที่ผู้เขียนและคณะได้นำเสนอ เพื่อสนับสนุนการนำแผนภาพเดิมที่ใช้ในการออกแบบระบบอยู่แล้วดังกล่าวข้างต้นมาแปลงให้เป็นแบบจำลองเชิงรูปนัยในรูปแบบโพรเมลาหรือแผนภาพสถานะ เพื่อให้เหมาะสมกับเครื่องมือทวนสอบที่ใช้งานอยู่ ในที่นี้จะเน้นเครื่องมือสปินเป็นหลัก อย่างไรก็ตาม ยังมีเครื่องมือทวนสอบที่สามารถทำการทวนสอบได้โดยตรงจากแผนภาพที่ไม่ใช่แผนภาพสถานะ โดยเครื่องมือจะทำการแปลงและแจกแจงปริภูมิสถานะให้เอง เช่น เครื่องมือซีพีเอ็น เป็นต้น ซึ่งในที่นี้มีการนำเสนองานวิจัยบางส่วนที่แปลงแผนภาพยูเอ็มแอลไปเป็นซีพีเอ็นด้วยเช่นกัน

การแปลงยูเอ็มแอลเป็นโพรเมลาเพื่อการทวนสอบ

การสร้างแบบจำลองด้วยการเขียนภาษาโพรเมลาตั้งแต่ต้น จากที่ไม่มีอะไรอยู่ในมือเลยนั้น ดูเหมือนจะยุ่งยากไม่ใช่น้อย และยังเป็นอุปสรรคสำหรับผู้ที่เริ่มต้นใหม่ จึงมีงานวิจัยจำนวนหนึ่งที่ช่วยให้แผนภาพที่ผู้ออกแบบระบบใช้อยู่เป็นประจำและคุ้นเคยมากกว่าถูกนำมาใช้ประโยชน์ในการอธิบายพฤติกรรมการทำงานของระบบได้ เช่น การใช้ชุดแผนภาพยูเอ็มแอลในการอธิบายโครงสร้างและพฤติกรรมของระบบ ซึ่งได้ทำกันอยู่แล้วในขั้นตอนการออกแบบระบบซอฟต์แวร์หรือระบบฮาร์ดแวร์ เป็นต้น นำแผนภาพยูเอ็มแอลมาแปลงให้เป็นภาษาโพรเมลาได้โดยอ้อมหรือโดยตรงเพื่อใช้เป็นแบบจำลองในเครื่องมือสปินได้ มีงานวิจัยที่แปลงแผนภาพซีเควนซ์และแผนภาพเครื่องสถานะ (state machine diagram) มาเป็นภาษาโพรเมลาแบบกึ่งอัตโนมัติอยู่จำนวนหนึ่ง แต่ในที่นี้ขอยกตัวอย่างเทคนิคการแปลงแผนภาพเครื่องสถานะมาเป็นโพรเมลา [28] ซึ่งนำเสนอกฎการแปลงแผนภาพไว้ 6 ข้อ โดยกฎที่ได้เป็นการแปลงสัญลักษณ์ที่พบในแผนภาพเครื่องสถานะไปเป็นโพรเมลาผ่านการใช้แผ่นแบบ (template) และการจัดการกรอกข้อมูลเพิ่มเล็กน้อยเกี่ยวกับชนิดตัวแปรให้ครบถ้วน เพื่อให้

พร้อมทำงานได้อัตโนมัติ ตามโครงสร้างการเปลี่ยนแปลงสถานะที่อธิบายในแผนภาพเครื่องสถานะได้อย่างครบถ้วน อย่างไรก็ตาม เนื่องจากแผนภาพเครื่องสถานะไม่มีการระบุการกระทำ (action) ภายในแต่ละสถานะ ผู้ใช้จึงต้องทำการเพิ่มส่วนการกระทำด้วยภาษาโพรเมลาเองในภายหลังได้

ข้อ	สัญลักษณ์แผนภาพ	โครงสร้างภาษาโพรเมลา

1	sti
	<PRECOND(sti)>
	<POSTCOND(sti)>
		1: mtype = {idle,running,done};
		2: mtype stateStatussti= idle;
		3: bool precFailsti= false, postFailsti= false;
		4: Proctype Statesti (){
		5: d_step {
		6: stateStatussti = running;
		7: precon:
		8: if PRECOND(sti) == true
		9: then goto stiOperation;
		10: else
		11: print "case else"
		12: if !PRECOND(sti) == true
		13: then precFailsti= true; Terminate =1;
		14: else
		15: print "case else"
		16: stiOperation:
		17: atomic { stateStatussti = running;
		18: /* ...Fill in details of operation.......... */
		19: goto poscon;
		20: }
		21: postcon:
		22: if POSTCOND(sti)== true
		23: then stateStatussti = done;
		24: else
		25: print "case else"
		26: if !POSTCOND(sti)== true
		27: then postFail = true; Terminate =1;
		28: else
		29: print "case else"
		30 } /* ...end d_step...*/
		31: }

รูปที่ 7.14 กฎการแปลงแผนภาพเครื่องสถานะข้อ 1 [28]

งานวิจัยนี้ครอบคลุมห้าสัญลักษณ์หลักของแผนภาพเครื่องสถานะ คือ สัญลักษณ์สถานแรกเริ่ม สถานะสิ้นสุด สถานะปกติ สัญลักษณ์การเลือกแยกทาง และสัญลักษณ์การรวม พร้อมเงื่อนไขที่กำกับบนเส้นเชื่อมสถานะด้วยที่ขึ้นด้วยภาษาโอซีแอล (object constraint language: OCL) [29]

ข้อ	สัญลักษณ์แผนภาพ	โครงสร้างภาษาโพรเมลา
2		1: init { 2: ...... 3: ...... 4: }

3		1: bit CompleteFinalState = 0;
		2: bit Terminate = 0;
		3: proctype *Finalst$_i$*( )
		4: { CompleteFinalState = 1; }
		5:active protype checkInvariantNormalTerminate ()
		6:{
		7: do
		8: ::assert(CompleteFinalState ==0)
		9: od;
		10:}
		11:active proctype checkInvariantAbNormalTerminate()
		12:{
		13: do
		14: ::assert(Terminate ==0)
		15:od;
		16:}

4	st_i [$Guard_{ij}$] st_j	: 1: st_i : 2: if stateStatusst_i== idle 3: then run statest_i( ); 4: else print "case else" 5: if stateStatusst_i== done && $Guard_{ij}$ ==true 6: then stateStatusst_i = idle; goto st_j; 7: else print "case else" 8: goto st_i; 9: st_j : :

รูปที่ 7.15 แสดงกฎการแปลงแผนภาพเครื่องสถานะข้อ 2-4 [28]

เราเริ่มการแปลงโดยนำเข้าแผนภาพเครื่องสถานะต้นฉบับและสกัดหาสัญลักษณ์ที่พบ และเมื่อพบแล้วให้ใช้กฎการแปลงที่นำเสนอแผ่นแบบไว้เพื่อให้ได้โครงร่างของโพรเมลาเป็นส่วน ๆ และนำจัดการมารวมกันในที่สุดอย่างอัตโนมัติ

กฎการแปลงข้อแรกเป็นการแปลงสัญลักษณ์สถานะ st ใดๆ ที่มีการกำหนดเงื่อนไขก่อนและเงื่อนไขหลังไว้ โดยแสดงแผ่นแบบที่ใช้สำหรับกฎการแปลงข้อที่ 1 ในรูปที่ 7.14 กฎข้อที่ 2 และข้อที่ 3 แปลงสัญลักษณ์สถานะแรกเริ่มและสถานะสิ้นสุดเพื่อนำมากำหนดโครงสร้างหลักของโพรเมลา กฎถัดไปข้อ 4 แปลงสัญลักษณ์เส้นเชื่อมต่อจากสถานะ st_i ไปยังสถานะถัดไป st_j ที่ระบุเงื่อนไขป้องกัน (guard condition) บนเส้นเชื่อม และสัญลักษณ์ทางเลือก ดูแผ่นแบบที่ใช้ในกฎข้อ 2-4 ได้ในรูปที่ 7.15 ส่วนกฎข้อ 5 เป็นการแปลงสถานะที่มีทางเลือกแยกเป็นสองทางแยก และกฎข้อ 6 เป็นการแปลงจุดรวมทางแยกเข้าด้วยกัน โดยแผ่นแบบสำหรับสองข้อสุดท้ายนี้แสดงในรูปที่ 7.16 และรูปที่ 7.17 ตามลำดับ

ข้อ	สัญลักษณ์แผนภาพ	โครงสร้างภาษาโพรเมลา
5	st_i [$Guard_{ij}$] $Choic_i$ [$Guard_{jm}$] [$Guard_{jn}$] st_m st_n	: 1: st_i : 2: if stateStatusst_i== idle 3: then run statest_i( ); 4: else print "case else" 5: if stateStatusst_i== done && $Guard_{ij}$ ==true 6: then if $Guard_{jm}$ ==true then stateStatusA=idle; goto st_m else print "case else" 7: if $Guard_{jn}$ ==true then stateStatusA=idle; goto st_n else print "case else" 8: else print "case else" 9: goto st_i; 10: st_n : :

รูปที่ 7.16 แสดงกฎการแปลงแผนภาพเครื่องสถานะข้อ 5 [28]

ดังที่กล่าวมาแล้ว กฎทั้ง 6 ข้อกำหนดลักษณะแผ่นแบบ ที่มีโครงร่างของภาษาโพรเมลา (Promela skeleton) บางส่วนไว้แล้ว และเมื่อพบสัญลักษณ์ต่างๆ ในรูปแผนภาพเครื่องสถานะต้นฉบับ เครื่องมือแปลงที่พัฒนาไว้แล้วก็จะทำการแปลงให้และทำการเพิ่มส่วนรายละเอียดในแผ่นแบบที่มีแล้วนั้นอย่างเป็นระบบจนกระทั่งได้โปรแกรมโพรเมลาที่ครบถ้วนพร้อมทำงานได้เบื้องต้น อย่างไรก็ตาม ผู้แปลงจะต้องจัดการส่วนสุดท้ายด้วยมือ เช่น การกำหนดค่าแรกเริ่มของตัวแปร การเพิ่มการกระทำในแต่ละ *proctype* สำหรับสถานะที่มีให้ครบ เป็นต้น

ข้อ	สัญลักษณ์แผนภาพ	โครงสร้างภาษาโพรมีลา
6	st_i st_j [$Guard_i$] [$Guard_j$] choice [$Guard_k$] st_k	: 1: st_i : 2: if stateStatusst_i== idle 3: then run statest_i(); 4: else print "case else" 5: if stateStatusst_i== done && $Guard_i$==true 6: then 7: if $Guard_k$ == true 8: then stateStatusst_i= idle; goto st_k 9: else print "case else" 10: else print "case else" 11: goto st_i; 12: st_j : 13: if stateStatusst_j ==idle 14: then run statest_j (); 15: else print "case else" 16: if stateStatusst_j==done && $Guard_j$==true 17: then 18: if $Guard_k$ == true 19: then stateStatusst_j = idle; goto st_k; 20: else print "case else" 21: else print "case else" 22: goto st_j; 23: st_k: 24: if stateStatusst_k== idle 25: then run statest_k(); :

รูปที่ 7.17 แสดงกฎการแปลงแผนภาพเครื่องสถานะข้อ 6 [28]

การแปลงระบบเชิงเวลาจริงเป็นโพรเมลาเพื่อการทวนสอบ

ระบบเชิงเวลาจริงแบบฝังตัว (embedded real-time system) เป็นระบบที่มีลักษณะการทำงานที่จำเป็น คือ การจัดการเงื่อนไขที่เป็นข้อจำกัดด้านทรัพยากร (resource-constrained handling) การจัดลำดับก่อนหลังของงานที่ต้องทำ (task prioritization) การจองแบบเวลาจริง (real-time preemption) ตลอดจนการมีทรัพยากรแบบขนาน (parallelism of resource) ทำให้กำหนดแบบจำลองเชิงรูปนัยยุ่งยากมาก และถ้าเราเริ่มต้นเขียนแบบจำลองของระบบเชิง

เวลาจริงด้วยภาษาโพรเมลาจากบรรทัดแรกจะมีความยุ่งยากและซับซ้อนแน่นอน มีงานวิจัยจำนวนหนึ่งที่แสดงวิธีการแปลงวงจรการทำงานของระบบเชิงเวลาจริงเป็นโพรเมลาด้วยเช่นกัน [30]

อย่างไรก็ตามในที่นี้ขอกล่าวถึงงานวิจัยในการสร้างแบบจำลองระบบเชิงเวลาจริงเป็นโพรเมลาจาก [30] โดยได้กำหนดส่วนโพรเมลาที่เป็นแผ่นแบบไว้ให้ใช้งาน โดยเฉพาะอย่างยิ่งการจัดการเรื่องสัญญาณนาฬิกา คือกระบวนการนับเวลา (tick process) ที่ทำหน้าที่นับเวลาจริงของระบบโดยกำหนดหน่วยเวลาของระบบเป็นหน่วย *Tick* ซึ่งแต่ละหน่วย *Tick* มีค่าประมาณ *1* มิลลิวินาที เพื่อให้สามารถจำลองการนับเวลาการทำงานของชิ้นส่วนจริงต่างๆ ของวงจรในระดับไมโครโปรเซสเซอร์ได้ และแผ่นแบบโพรเมลาจัดให้มีการนับเวลาแบบไม่สิ้นสุด โดยทำการนับวนซ้ำในช่วงเวลา *50 Tick* ตั้งแต่ *Tick* ที่ *0* ถึง *Tick* ที่ *49* และวนกลับไปนับ *0* ใหม่เป็นเช่นนี้ตลอดเวลา

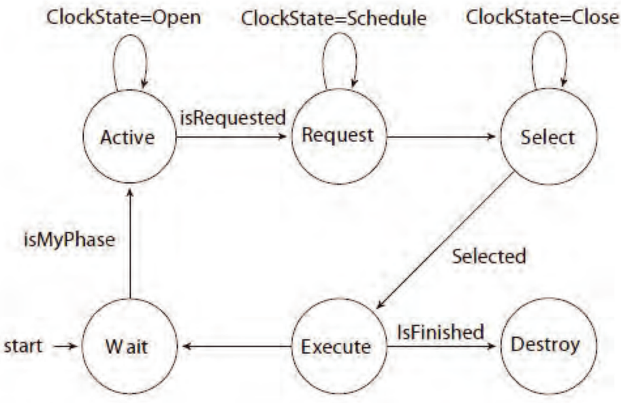
ทั้งนี้การนับเวลาแบบ *50 Tick* จะสามารถครอบคลุมงานที่ใช้เวลานานถึง *50* มิลลิวินาที สถานะของงานแบ่งเป็น *4* สถานะคือ สถานะ *Open*, สถานะ *Schedule*, สถานะ *Close*, และสถานะ *Execute* ตามลำดับโดยแสดงไว้ในส่วนโพรเมลาในรูปที่ 7.18

```
1    #define isMyPhase (((Tick - (myTask.phase)) % myTask.period == 0))
2    mtype = {Open, Close, Execute, Schedule}
3    proctype Ticking() {
4    do::((ClockState == Open)&& timeout) ->ClockState=Schedule;
5       ::((ClockState == Schedule)&& timeout) ->ClockState = Close;
6       ::((ClockState == Close)&& timeout) ->ClockState = Execute;
7       ::((ClockState == Execute) && timeout) ->ClockState = Open;
8              Tick++;          /* advance to next tick  */
9         if :: (Tick > (TotalTickPerLoop)) -> Tick = 0;
10           :: else;
11        fi;    /*  implement reset all resources here  */
12   od;}
```

รูปที่ 7.18 ส่วนโพรเมลาที่อธิบายกระบวนการนับเวลาและการเปลี่ยนลำดับสถานะของงาน [30]

ดังนั้นชิ้นส่วนต่างๆ ที่ต้องมีการงานจะถูกนิยามให้เป็น *Task* ที่จะมีการทำงานตามแผนภาพเครื่องสถานะในรูปที่ 7.19 เสมอ โดยแผนภาพเครื่องสถานะนี่ใช้งานได้กับชิ้นส่วนอุปกรณ์ในระบบเชิงเวลาแบบฝังตัวได้โดยทั่วไป

โดยผู้วิจัยได้กำหนดแผ่นแบบโพรเมลาที่ใช้นิยามการทำงานตามแผนภาพเครื่องสถานะที่กำหนดไว้ในรูปที่ 7.20



รูปที่ 7.19 แผนภาพเครื่องสถานะของงานที่เกิดขึ้นในระบบ (Task) [30]

ด้วยแผ่นแบบโพรเมลาดังที่กล่าวมาแล้วสามารถนำมาสร้างแบบจำลองโพรเมลาสำหรับระบบเชิงเวลาจริงเบื้องต้นได้โดยสั่งให้ชิ้นส่วนต่างๆ ที่มีอยู่ในระบบนี้ให้เริ่มทำงานของตนซึ่งแต่ละงานจะมีสถานะไปตามแผนภาพเครื่องสถานะที่กำหนดไว้ในแผ่นแบบ พร้อมกันนี้สัญญาณนาฬิกาก็เริ่มทำงานอย่างอัตโนมัติเช่นกัน ระบบดังกล่าวภายใต้การควบคุมจากสัญญาณนาฬิกาก็ทำงานไปเรื่อยๆ ระบบเชิงเวลาจริงส่วนใหญ่จะมีลักษณะการทำงานแบบไม่สิ้นสุดนั่นเอง อย่างไรก็ตามยังมีส่วนแผ่นแบบรายละเอียดอื่นๆ สำหรับการจัดการเงื่อนไขและข้อจำกัดของทรัพยากรอีกจำนวนหนึ่งที่แสดงไว้ในงานวิจัยที่ไม่ได้กล่าวในที่นี้สนใจเพิ่มเติมติดตามในต้นฉบับ [30]

ผลลัพธ์จากการแปลงระบบเชิงเวลาแบบฝังตัวให้เป็นโพรเมลาจะได้แบบจำลองครบถ้วนที่สามารถนำไปใช้ในเครื่องมือสปินได้ทันที ข้อดี (merit) ของงานวิจัยนี้คือ การที่ใช้โพรเมลาในการจัดการสัญญาณนาฬิกา และการนำเสนอแผนแบบสำหรับแผนภาพเครื่องสถานะสำหรับงานที่เกิดในระบบ ซึ่งทำให้เครื่องมือสปินซึ่งโดยทั่วไปจะใช้ทวนสอบเฉพาะระบบแบบอสมวารให้สามารถนำมาทวนสอบกับระบบสมวาร ที่มีสัญญาณนาฬิกาจริงควบคุมได้

```
1   proctype CreateTask(taskdef myTask){
2   int cycleCount = 0;          /* Cycle counter */
3   int durationCount = 0;       /* Duration counter for task */
4   bool isTaskActive = false;  /* Guarding for this task if not Active yet*/
5   mtype RES[MaxResource];
6   WAIT:   /* When finish every tick, task will wait here */
7   do
8   ::((!isTaskActive)&&(GlobalClockState==Open)
9     &&(isMyPhase))->ACTIVE:      isTaskActive=true;
10  ::((isTaskActive)&&(GlobalClockState==Open)
11    &&(!Tlist[myTask.my_type].is_Requested))->
12    TaskArray[myTask.my_type].is_Requested = true;
13  REQUEST:
14  ::((isTaskActive)&&(GlobalClockState == Schedule))->
15      /* Implement Scheduler here*/
16  ::((isTaskActive)&&(GlobalClockState==Close)
17     &&(!isLessPriority(myTask.my_type))
18     &&(Selected == none)
19     &&(TaskArray[myTask.my_type].is_Requested))->
20   SELECTING: Selected = myTask.my_type;
21  :: ((isTaskActive)
22      &&(GlobalClockState == Execute)
23      &&(TaskArray[myTask.my_type].is_Requested)
24      &&(Selected == myTask.my_type))->
25  EXCUTION:  TaskArray[myTask.my_type].is_Requested = false;
26      durationCount++;      /* Increase task execution duration */
27       if :: (durationCount >= myTask.duration)->durationCount=0->
28             goto TaskIsFinished;
29        :: else->skip;
30        fi;
31        HighestGetResource = true;
32   ::((GlobalClockState == Excecute)&&(isTaskActive)
33     &&(TaskArray[myTask.my_type].is_Requested)
34     &&(HighestGetResource)&&(CheckMyResource))->
35  Tlist[myTask.my_type].is_Requested = false;
36  Tlist[myTask.my_type].is_Scheduled = false;
37   If ::(Resource[0].owner==_IDLE_ && myTask.isreqRES0)
38        ->Resource[0].owner=myTask.my_type;
39      :: else;
40   fi;
41   if ::(Resource[1].owner==_IDLE_ && myTask.isreqRES1)
42      ->Resource[1].owner=myTask.my_type;
43     :: else;
44  fi;
```

รูปที่ 7.20 แผ่นแบบโพรเมลาสำหรับการนิยามส่วนงานที่ระบบทำ (Task) [30]

```
45    if ::(Resource[2].owner==_IDLE_ && myTask.isreqRES2)
46       ->Resource[2].owner=myTask.my_type;
47       :: else;
48    fi; durationCount++;
49    if :: (durationCount >= myTask.duration)->durationCount=0
50        ->goto TaskIsFinished;
51       :: else->skip;
52    fi;
53    GlobalClockState == C_Open;
54     od;
55    TaskIsFinished: isTaskFinished[myTask.my_type] = true;
56     durationCount = 0;
57     isTaskActive = false;
58     if ::(myTask.is_periodic)->cycleCount=0;
59       goto WAIT; /* Run infinitely eventually */
60       ::((cycleCount+1) < myTask.cycle)->
61       cycleCount++;printf("cycle count %d\n",cycleCount); goto WAIT;
62       ::else->cycleCount=0;
63     fi;
64    DESTROY:
65    }
```

รูปที่ 7.20 แผ่นแบบโพรเมลาสำหรับการนิยามส่วนงานที่ระบบทำ [30] (ต่อ)

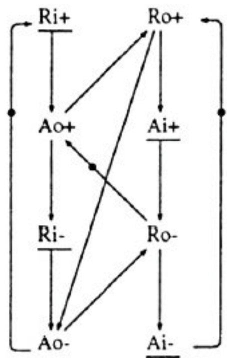
การแปลงแผนภาพเอสทีจีเป็นโพรเมลาเพื่อการทวนสอบ

สำหรับการทวนสอบระบบฮาร์ดแวร์ที่ต้องการใช้โพรเมลาสร้างแบบจำลอง ก็สามารถจัดทำได้เช่นกัน ในที่นี้ขอนำเสนองานวิจัยของ [31] ที่เริ่มนำการออกแบบวงจรอสมวารโดยใช้แผนภาพเอสทีจีหรือเรียกแบบเต็มว่า แผนภาพกราฟการเปลี่ยนระดับสัญญาณ (signal transition graph) มาแปลงเป็นแบบจำลองด้วยโพรเมลาและนำมาจำลองการทำงานด้วยเครื่องมือสปิน และนำผลลัพธ์การตามรอย (trail) จากการจำลองโดยเครื่องมือสปินมาวิเคราะห์ต่อเพื่อตรวจหาคุณสมบัติการคงอยู่ของสัญญาณ (signal persistence checking) ของวงจรอสมวาร

แผนภาพเอสทีจี [32] คือ แผนภาพในรูปแบบกราฟที่ถูกลดรูปลดจากเพทริเน็ตเพื่อให้สามารถใช้ในการจำลองพฤติกรรมของวงจรที่มีการเปลี่ยนแปลงขึ้นลงของสัญญาณ เราเรียกการเปลี่ยนแปลงขึ้นลงของสัญญาณว่า ทรานสิชันของสัญญาณ โดยถ้าเป็นการเปลี่ยนสัญญาณที่ตำแหน่ง A_i ของวงจรจากค่า 0 เป็นค่า 1 เราจะใช้สัญลักษณ์ A_i^+ แทนพฤติกรรมนี้ แต่ถ้าเป็นการเปลี่ยนสัญญาณที่ตำแหน่ง A_i จากค่า 1 ไปเป็นค่า 0 เราใช้สัญลักษณ์ A_i^- ดูรูปที่ 7.21 ประกอบด้วย ดังนั้น สัญลักษณ์ที่เห็น R_o^+ ก็หมายถึงจังหวะนั้นสัญญาณตำแหน่ง R_o เปลี่ยนค่าหรือมีทรานสิชันจาก 0 ไปเป็น 1 นั่นเอง

งานวิจัยของ [31] ได้เสนอวิธีการแปลงแผนภาพเอสทีจีที่กำหนดให้และวงจรอสมวารในระดับเกตให้เป็นโพรเมลาโดยใช้กฎการแปลงและแผ่นแบบเพื่อความสะดวกในการใช้งาน นั่นหมายถึง แผนภาพเอสทีจีอธิบายพฤติกรรมของวงจร และแผนภาพเค้าร่างวงจรระดับเกตแสดงด้วยการเชื่อมต่อของเกตแต่ละตัว

ยกตัวอย่างแผนภาพเอสทีจีต้นฉบับของพฤติกรรมของวงจร *Celement* แสดงในรูปที่ 7.21 แผนภาพนี้จะถูกแปลงเป็นแผ่นแบบโพรเมลาที่ใช้ในการขับการเปลี่ยนระดับสัญญาณ (signal transition driver) ในรูปที่ 7.22 ผลที่ได้จะเป็นโปรแกรมโพรเมลาพฤติกรรมวงจรที่สั่งการให้มีการเปลี่ยนระดับสัญญาณที่เกิดขึ้นจริงในแผนภาพเอสทีจี



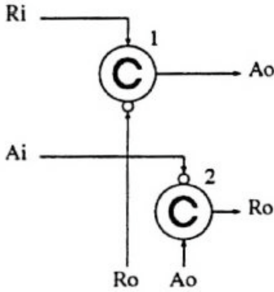
รูปที่ 7.21 แผนภาพเอสทีจีต้นฉบับที่ถูกแปลง [31]

ส่วนที่สอง คือ ส่วนโครงสร้างการต่ออุปกรณ์ในวงจร การแปลงโพรเมลาสำหรับวงจรอสมวารที่มีชิ้นส่วนในระดับเกต (gate) นำมาต่อกันโดยการเชื่อมสัญญาณขาเข้าและขาออกเป็นวงจรอย่างสมบูรณ์ในที่นี้ใช้ตัวอย่างวงจรอสมวารของ *Celement1* และ *Celement2* ในรูปที่ 7.23 โพรเมลาที่แปลงได้แสดงในรูปที่ 7.24

จากนั้นก็นำโพรเมลาสองส่วนที่ได้มารวมกันและสั่งให้มีการจำลองในเครื่องมือสปินแบบทีละขั้น ทำให้สามารถเห็นการตามรอยของค่าตัวแปรต่างๆ และนำผลมาวิเคราะห์ต่อด้วยโปรแกรมอื่นโดยตรวจหาคุณสมบัติการคงอยู่ของสัญญาณแบบสองระยะ (2-phase signal persistence) ที่เกิดขึ้นในวงจรอสมวารได้

```
active proctype MonitorA(){
do
:: (Ao == 1) -> atomic {
        if
        :: (Ri == 1) -> { Ri = 0 ; i = i + 1;
        printf("\n Output > |%d| Ri=|%d|,
Ro=|%d|, Ai=|%d|, Ao=|%d| \n", i, Ri,Ro,Ai,Ao); }
        :: skip
        fi;
        }
:: (Ao == 0) -> atomic {
        if
        :: (Ri == 0) -> { Ri = 1 ; i = i + 1;
        printf("\n Output > |%d| Ri=|%d|,
Ro=|%d|, Ai=|%d|, Ao=|%d| \n", i, Ri,Ro,Ai,Ao); }
        :: skip
        fi;
        }
od;
}
active proctype MonitorR(){
do
:: (Ro == 1) -> atomic {
        if
        :: (Ai == 0) -> { Ai = 1 ; i = i + 1;
        printf("\n Output > |%d| Ri=|%d|,
Ro=|%d|, Ai=|%d|, Ao=|%d| \n", i, Ri,Ro,Ai,Ao); }
        :: skip
        fi;
        }
:: (Ro == 0) -> atomic {
        if
        :: (Ai == 1) -> { Ai = 0 ; i = i + 1;
        printf("\n Output > |%d| Ri=|%d|,
Ro=|%d|, Ai=|%d|, Ao=|%d| \n", i, Ri,Ro,Ai,Ao); }
        :: skip
        fi;
        }
od;
}
```

รูปที่ 7.22 แผ่นแบบโพรเมลาสำหรับการขับการเปลี่ยนระดับสัญญาณในแผนภาพเอสทีจี [31]



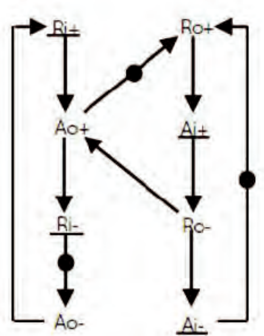
รูปที่ 7.23 วงจรอสมวารต้นฉบับที่ถูกแปลง [31]

```
bit Ri=0, Ro=0, Ai=1, Ao=1;
int i=0;

active proctype Celement1(){
do
:: (Ri == 0 && Ro == 1) -> atomic {
          if
          :: (Ao == 1) -> { Ao = 0;  i = i+1;
          printf("\n Output >  |%d| Ri=|%d|,
Ro=|%d|, Ai=|%d|, Ao=|%d| \n", i, Ri,Ro,Ai,Ao); }
          :: skip
          fi;
          }
:: (Ri == 1 && Ro == 0) -> atomic {
          if
          :: (Ao == 0) -> { Ao = 1;  i = i+1;
          printf("\n Output >  |%d| Ri=|%d|,
Ro=|%d|, Ai=|%d|, Ao=|%d| \n", i, Ri,Ro,Ai,Ao); }
          :: skip
          fi;
          }
od;
}

active proctype Celement2(){
do
:: (Ai == 1 && Ao == 0) -> atomic {
          if
          :: (Ro == 1) -> { Ro = 0;  i = i+1;
          printf("\n Output >  |%d| Ri=|%d|,
Ro=|%d|, Ai=|%d|, Ao=|%d| \n", i, Ri,Ro,Ai,Ao); }
          :: skip
          fi;
          }
:: (Ai == 0 && Ao == 1) -> atomic {
          if
          :: (Ro == 0) -> { Ro = 1;  i = i+1;
          printf("\n Output >  |%d| Ri=|%d|,
Ro=|%d|, Ai=|%d|, Ao=|%d| \n", i, Ri,Ro,Ai,Ao); }
          :: skip
          fi;
          }
od;
}
```

รูปที่ 7.24 แบบจำลองวงจร Celement เขียนด้วยโพรเมลา [31]

ต่อมางานวิจัยของ [33] ได้ทำการออกแบบอัลกอริธึมเพื่อแปลงแผนภาพเอสทีจีเป็นโพรเมลาอย่างอัตโนมัติ และออกแบบวิธีการทวนสอบเพื่อตรวจหาคุณสมบัติการคงอยู่ของสัญญาณ (signal persistency checking) อย่างอัตโนมัติ โดยใช้วิธีการทวนสอบแบบวิเคราะห์หมดจด (exhaustive analysis) ด้วยเครื่องมือสปิน โดยไม่ใช้เทคนิคการตามรอยจากการจำลองวงจรเหมือนแบบเดิมข้างต้น ตัวอย่างของแผนภาพเอสทีจีที่ได้รับการแปลงแสดงในรูปที่ 7.25 และโพร

เมลาผลลัพธ์ที่แปลงได้แสดงในรูปที่ 7.26 อย่างไรก็ตามในที่นี้ไม่ได้นำอัลกอริธึมการแปลงมาแสดง สนใจติดตามรายละเอียดใน [33]



รูปที่ 7.25 ต้นแบบ STG สำหรับการแปลง [33]

```
1   //define signal name
2   #define signal    byte
3   signal ripaop,aoprop,aoprim,rimaom,aomrip;
4   signal ropaip,aiprom,romaim,romaop,aimrop;
5
6   //define transition rule
7   #define inp1(x)      (x>0) -> x = x-1
8   #define inp2(x,y)    (x>0 && y>0) -> x = x-1; y=y-1
9   #define out1(x)      x = x+1
10  #define out2(x,y)    x = x+1; y = y+1
11
12
13  init {
14  //define initial marking
15  aomrip = 1; romaop = 1; aimrop = 1; /*initial marking*/
16
17  //define flow relation
18  do
19      :: atomic { inp1(aomrip)    -> out1(ripaop) }
20      :: atomic { inp2(ripaop,romaop) -> out2(aoprop,aoprim) }
21      :: atomic { inp1(aoprim)    -> out1(rimaom)}
22      :: atomic { inp1(rimaom)    -> out1(aomrip) }
23      :: atomic { inp2(aimrop,aoprop) -> out1(ropaip) }
24      :: atomic { inp1(ropaip)    -> out1(aiprom) }
25      :: atomic { inp1(aiprom)    -> out2(romaim,romaop)}
26      :: atomic { inp1(romaim)    -> out1(aimrop) }
27  od
28  }
```

รูปที่ 7.26 โพรเมลาที่แปลงจากต้นแบบ STG เพื่อใช้ในการทวนสอบ [33]

การแปลงแผนภาพยูเอ็มแอลเป็นซีพีเอ็นเพื่อการทวนสอบ

เพทริเน็ตเป็นวิธีหนึ่งที่เราวาดรูปแบบจำลองเชิงรูปนัยเพื่ออธิบายพฤติกรรมของระบบ และคัลเลอร์เพทริเน็ตเป็นเพทริเน็ตที่ขยายขีดความสามารถในการรับรู้ชนิดของโทเคน โดยให้โทเคนมีความแตกต่างกันมีการเก็บค่าในโทเคนและทำการยิงส่งไปมาระหว่างเพลสทำให้เราสามารถจำลองการส่งข้อมูลได้

และขณะเดียวกันคัลเลอร์เพทริเน็ตยังให้เราสามารถเขียนโปรแกรมสคริปท์ภาษาเอ็มแอล ที่ทำงานในส่วนต่างๆ กำกับไว้ในแต่ละจุดของโครงสร้างคัลเลอร์เพทริเน็ตได้ เพื่อทำการตรวจสอบเงื่อนไขการเปิดทางและเงื่อนไขการยิงได้มากขึ้น เครื่องมือซีพีเอ็นสามารถอ่านแบบจำลองระบบที่วาดขึ้นด้วยคัลเลอร์เพทริเน็ตและทำการจำลองการทำงานตลอดจนหาปริภูมิสถานะของระบบเพื่อใช้ค้นหาคุณลักษณะที่ต้องการทวนสอบได้เช่นเดียวกับเครื่องมือสปินที่กล่าวมาแล้ว

ในทำนองเดียวกันการวาดรูปคัลเลอร์เพทริเน็ตตั้งต้นเพื่อใช้อธิบายระบบที่เราสนใจ มักจะยุ่งยากสำหรับผู้ไม่คุ้นเคยสัญลักษณ์ของคัลเลอร์เพทริเน็ต แต่เราสามารถนำแผนภาพที่คุ้นเคยที่ทำไว้ในกระบวนการพัฒนาระบบซอฟต์แวร์หรือระบบฮาร์ดแวร์มาใช้ให้เป็นประโยชน์ได้ เช่น แผนภาพยูเอ็มแอล เป็นต้น และมีงานวิจัยจำนวนไม่น้อยที่ทำการแปลงแผนภาพยูเอ็มแอลมาเป็นคัลเลอร์เพทริเน็ตได้ แต่ทั้งนี้ก็ต้องมีการปรับแต่งเพิ่มเติมและไม่ครอบคลุมลาเบลที่มีอยู่ ในที่นี้ขอนำเสนองานวิจัยการแปลงแผนภาพกิจกรรมมาเป็นคัลเลอร์เพทริเน็ตพร้อมกำกับข้อความบนลาเบลให้ด้วยอย่างอัตโนมัติ [34] โดยนำเสนอกฎการแปลงสัญลักษณ์แผนภาพกิจกรรมหลัก 7 รายการไปเป็นสัญลักษณ์คัลเลอร์เพทริเน็ต ทั้งนี้มีการแปลงส่วนที่กำกับเครื่องหมายไปด้วยในเวลาเดียวกันแสดงเพิ่มเติมใน [34]

เริ่มต้นจากการนำแผนภาพกิจกรรมยูเอ็มแอล [29] เข้ามาเพื่อค้นหาสัญลักษณ์ที่สามารถแปลงไปเป็นส่วนของสัญลักษณ์ในคัลเลอร์เพทริเน็ต สัญลักษณ์ทั้งเจ็ดรายการแสดงในรูปที่ 7.27 คือ สัญลักษณ์โหนดเริ่มต้นที่ปรากฏทางด้านซ้ายของรูปที่ 7.27 จะถูกแปลงไปเป็นสัญลักษณ์เพลสเริ่มต้น ต่อมา ถ้าเราพบสัญลักษณ์โหนดสิ้นสุดในแผนภาพกิจกรรมก็ให้นำชุดทรานสิชันคู่กับเพลสของคัลเลอร์เพทริเน็ตมาแทน ทำเช่นนี้ไปจนครบทุกสัญลักษณ์ที่พบ ก็จะได้คัลเลอร์เพทริเน็ตที่เป็นแบบร่างเริ่มต้นได้ ในงานวิจัยนี้ได้ทำการแปลงส่วนที่เรียกว่าข้อความจารึก (inscription) ที่ปรากฏในแผนภาพกิจกรรมมาเติมในคัลเลอร์เพทริเน็ต เพื่อที่จะได้มีข้อมูลเพิ่มเติมในการอธิบายการทำงานได้ เช่น ข้อมูลตัวแปรเงื่อนไขต่างๆ ที่เป็นข้อความจารึก เป็นต้น ทำให้คัลเลอร์เพทริเน็ตมีรายละเอียดมากขึ้น และเมื่อผู้ทำการแปลงเพิ่มเติมรายละเอียดในส่วนของตัวแปรและชนิดตัวแปรแล้ว ก็จะพร้อมที่จะนำไปสั่งให้ทำงานได้ในเครื่องซีพีเอ็น เนื่องจากคัลเลอร์เพทริเน็ตมีความแตกต่างจากเพทริเน็ตทั่วไปที่มีความสามารถในการกำหนดให้โทเคนแต่ละโทเคนเป็นเสมือนตัวแปรที่เก็บค่าตามชนิดโทเคนที่ระบุ ส่วนนี้เป็นเรื่องยากที่ทำได้โดยอัตโนมัติ ผู้ใช้ต้องเพิ่มเติมชนิดโทเคนและค่าเริ่มต้นในมาร์กกิงเริ่มต้นให้กับแผนภาพคัลเลอร์เพทริเน็ตที่แปลงมาด้วย

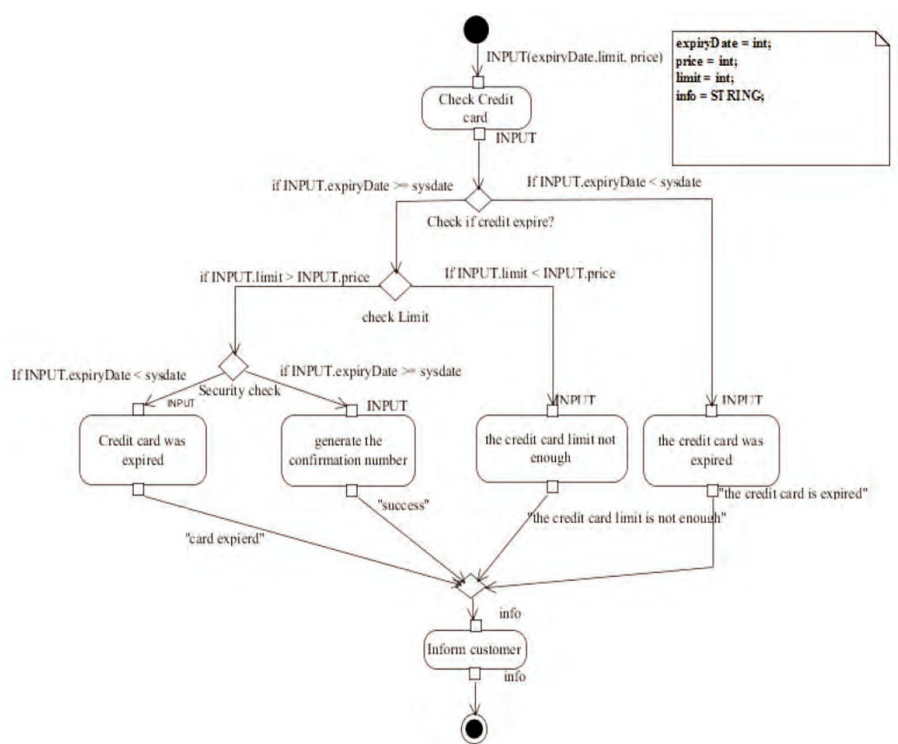
No	Activity Diagrams	CPN Models
1		colset Inscription
2		Inscription colset
3		Inscription Inscription colset
4		If If colset colset
5		Inscription Inscription colset
6		Inscription Inscription colset colset
7		Inscription Inscription Inscription colset

รูปที่ 7.27 สัญลักษณ์ที่แปลงจากแผนภาพกิจกรรมไปยังคัลเลอร์เพทริเน็ต [34]

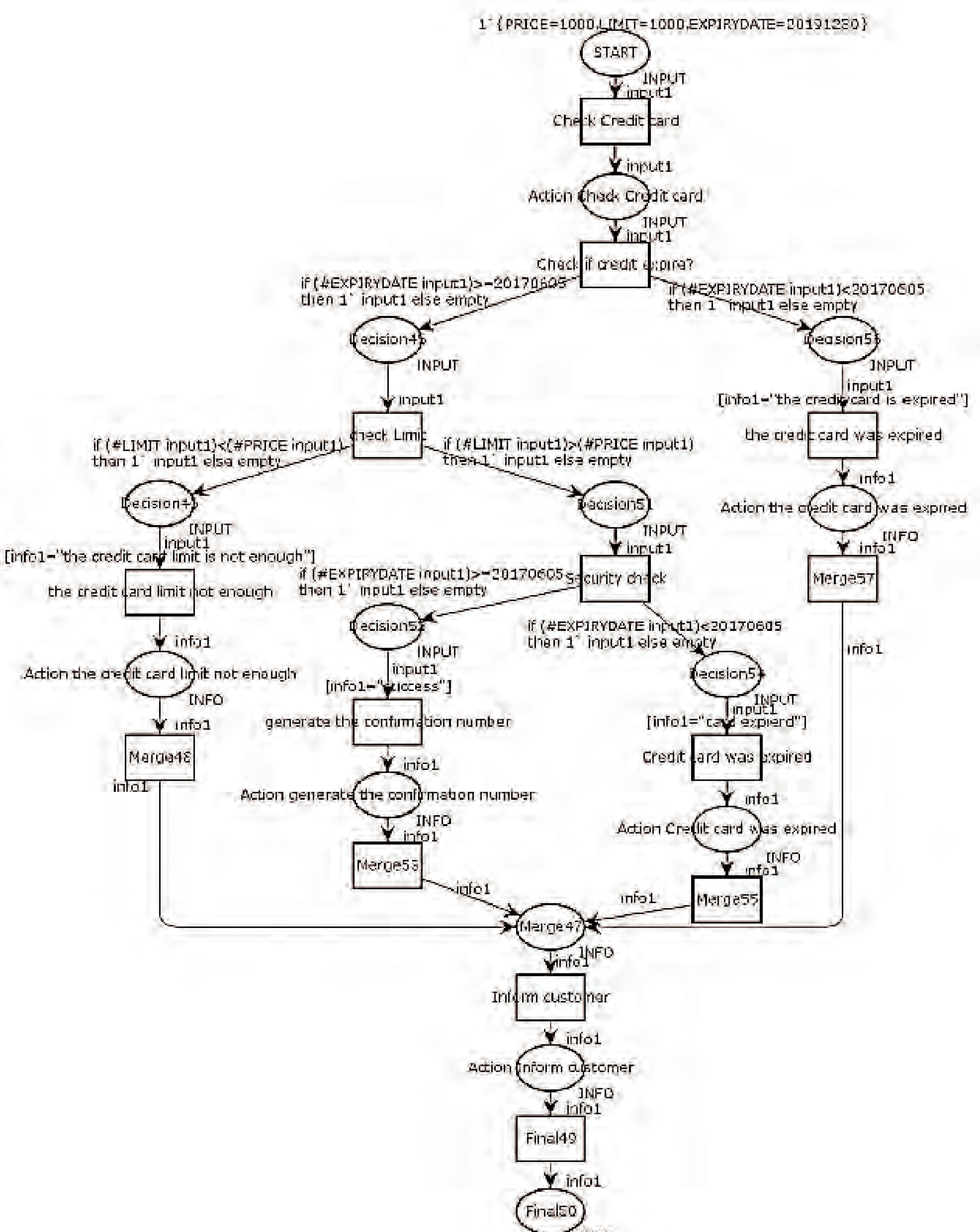
รูปที่ 7.28 แสดงตัวอย่างแผนภาพกิจกรรมในยูเอ็มแอลที่อธิบายพฤติกรรมการตรวจสอบการใช้บัตรเครดิตของลูกค้าที่สั่งซื้อสินค้าออนไลน์ว่าบัตรเครดิตหมดอายุหรือไม่ ถ้ายังไม่หมดอายุใช้งานก็ให้มาตรวจสอบว่าวงเงินบัตรเครดิตมีเพียงพอในการชำระค่าสินค้าที่ลูกค้าต้องการในการสั่งซื้อหรือไม่ ถ้าวงเงินเพียงพอก็รายงานผลว่า ใช้งานได้เรียบร้อยดี แต่ถ้ามีปัญหาก็ให้รายงานผลว่ามีปัญหาด้านใดกับลูกค้า ในการนี้แผนภาพกิจกรรมที่ให้มาตั้งต้นอาจจะต้องมีการกำหนดกล่อง annotation เพื่อระบุรายละเอียดเพิ่มเติมของชื่อตัวแปรและ

ชนิดตัวแปรที่ใช้จารึกในแผนภาพกิจกรรมด้วย ทั้งนี้ข้อความส่วนนี้จะเป็นตัวช่วยให้ผู้ใช้นำไปประบุเพิ่มในคัลเลอร์เพทริเน็ตในส่วนการกำหนดค่าและชนิดของโทเคนได้ถูกต้องและสะดวกมากขึ้น

รูปที่ 7.29 แสดงคัลเลอร์เพทริเน็ตที่แปลงมาจากแผนภาพกิจกรรมโดยมีการใส่รายละเอียดเรื่องตัวแปรและชนิดตัวแปรเพิ่ม ทำให้สามารถนำไปจำลองการทำงานของพฤติกรรมการสั่งสินค้าได้ในเครื่องมือซีพีเอ็น ได้เลยทีเดียว ข้อดีของงานวิจัยนี้คือคัลเลอร์เพทริเน็ตมีข้อมูลที่จำเป็นเกือบพร้อมใช้งาน ในขณะที่งานวิจัยอื่นมักจะไม่เน้นเรื่องการระบุแปลงข้อความจารึกเหล่านี้ แต่เน้นการแปลงลำดับการทำงานเสียมากกว่า ทำให้เมื่อแปลงได้แล้วยังไม่สามารถนำไปใช้งานได้ต้องนำมาออกแบบกันใหม่



รูปที่ 7.28 ตัวอย่างแผนภาพกิจกรรมที่มีลาเบลกำกับ [34]



รูปที่ 7.29 ตัวอย่างผลลัพธ์แผนภาพคัลเลอร์เพทริเน็ตที่มีลาเบลกำกับ [34]

สำหรับคัลเลอร์เพทริเน็ตที่มีการกำหนดว่าถ้าพบเงื่อนไขใด ๆ แล้วจะต้องมีการกระทำใด ๆ เกิดขึ้นตามมาเสมอ โดยเงื่อนไขอาจจะเป็นเงื่อนไขประกอบ (compound conditions) ที่มีความซับซ้อนและการกระทำเป็นชุดการกระทำ

จำนวนมากกว่าหนึ่งรายการ โดยทั่วไปแล้วผู้ออกแบบมักจะใช้ตารางการตัดสินใจ (decision table) มาช่วยในการกำหนดกฎทางธุรกิจแบบนี้ (business rules)

การนำกฎทางธุรกิจในตารางไปเพิ่มเติมในคัลเลอร์เพทริเน็ตมักจะต้องอาศัยผู้มีพื้นฐานด้านโปรแกรมภาษาเอ็มแอล ดังนั้นงานวิจัย [35] นำเสนอเครื่องมืออัตโนมัติที่รับตารางการตัดสินใจของกฎธุรกิจเหล่านี้ที่แสดงในรูปที่ 7.30 และสร้างฟังก์ชันภาษาเอ็มแอลเพื่อนำไปใช้ในคัลเลอร์เพทริเน็ตได้ทันที โดยเครื่องมือนี้จะทำการแทรกฟังก์ชันเข้าสู่คัลเลอร์เพทริเน็ตอย่างอัตโนมัติ และทำงานประสานกันกับเครื่องมือซีพีเอ็นด้วยเช่นกัน

	TABLE	RULE						
		1	2	3	4	5		N
CONDITION STUB	CONDITION 1							
	CONDITION 2							
			CONDITION ENTRIES					
	CONDITION N							
ACTION STUB	ACTION 1							
	ACTION 2							
			ACTION ENTRIES					
	ACTION N							

รูปที่ 7.30 ตัวอย่างตารางการตัดสินใจที่ใช้ในการกำหนดการกระทำภายใต้เงื่อนไข [35]

โดยการวิเคราะห์โครงสร้างฟังก์ชันในภาษาเอ็มแอลและแบ่งพื้นที่เป็นสามส่วนคือ 1) ส่วนชื่อฟังก์ชัน 2) ส่วนพารามิเตอร์พร้อมอธิบายชนิด 3) ส่วนโครงสร้างภาษาแบบ *If-then-else* และ *action* ที่ปรากฏเพื่อนำเงื่อนไขต่างๆ และการกระทำต่างๆ มาสร้างให้เป็นฟังก์ชันอย่างเหมาะสมโดยอัตโนมัติ แสดงในรูปที่ 7.31 และนำเข้าไปเก็บไว้ในคัลเลอร์เพทริเน็ตอย่างอัตโนมัติ

```
        1                          2
fun _functionName ( _parameterName:_parameterType ) =
    if ( _condition ) then _action
    else if ( _condition ) then _action;      3
    else _action ;
```

รูปที่ 7.31 โครงสร้างฟังก์ชันในภาษาเอ็มแอลที่ใช้ในการกำหนดกฎธุรกิจ [35]

การแปลงแผนภาพบีเพลเป็นโพรเมลาเพื่อการทวนสอบ

สำหรับระบบที่ใช้เทคโนโลยีเซอร์วิสในการพัฒนาระบบนั้น ผู้ออกแบบมักจะไม่เลือกใช้แผนภาพยูเอ็มแอล แต่หันมาใช้แผนภาพและมาตรฐานใหม่ที่ คือ แผนภาพบีเพล (business process execution language: BPEL) [36] แทน

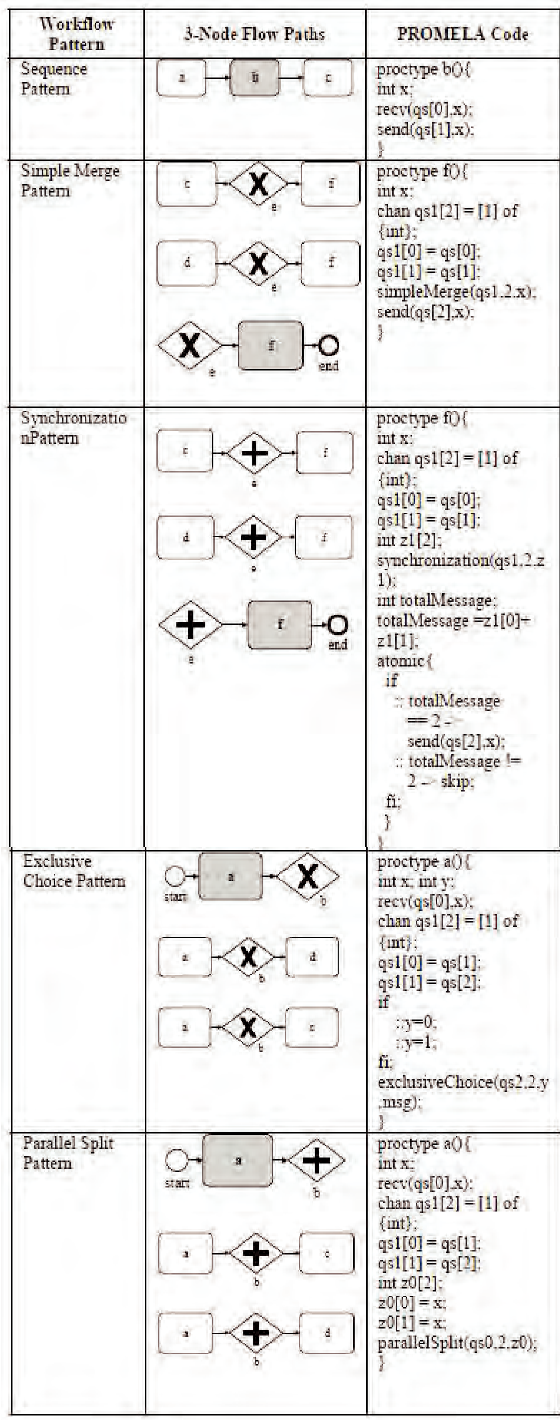
ระบบที่พัฒนาด้วยเทคโนโลยีเซอร์วิสนี้สามารถนำไปติดตั้งกระจายบนเครือข่ายอินเทอร์เน็ตได้อย่างกว้างขวาง และสามารถให้บริการทั่วไปได้ ผ่านการเรียกใช้โดยใช้โพรโตคอลของเว็บ บางครั้งเราก็เรียกว่า เว็บเซอร์วิส การที่มีเซอร์วิสมากมายทำให้การออกแบบมีความซับซ้อนมากขึ้น และจำเป็นต้องทำการทวนสอบด้วยเช่นกัน ครั้นจะสร้างแบบจำลองเชิงรูปนัยของระบบโดยรวมก็คงซับซ้อนมาก เราไม่สามารถเขียนโพรเมลาเองบนกระดาษเปล่าได้แน่นอน

และถ้าเรานำการออกแบบเซอร์วิสที่ทำอยู่แล้วด้วยแผนภาพบีเพลมาใช้ต่อในการแปลงให้เป็นโครงร่างโพรเมลาได้ แม้จะไม่สมบูรณ์ครบถ้วนแบบอัตโนมัติ ก็มีประโยชน์มากในการขึ้นโครงร่างแบบจำลองเชิงรูปนัย และทำงานต่อเติมให้สมบูรณ์ง่ายกว่าเดิม

ในที่นี้ขอนำเสนอการแปลงแผนภาพบีเพล โดยเริ่มจากการพบว่า ในระบบขนาดใหญ่และซับซ้อน ผู้ออกแบบระบบงานมักจะนิยมใช้แบบรูปกระแสงาน (workflow pattern) เพื่อให้ลดภาระในการตรวจสอบ เนื่องจากแบบรูปกระแสงานที่นิยามไว้นั้นจะมีพฤติกรรมที่วิเคราะห์มาเรียบร้อยแล้วว่ามีข้อดีและข้อเสียอย่างไร และเลือกนำมาประกอบใช้ได้ตามความเหมาะสมกับความต้องการได้

สำหรับงานวิจัย [37] ได้นำเสนอวิธีการแปลงแบบรูปกระแสงานพื้นฐานจำนวน 5 แบบรูปไปเป็นภาษาโพรเมลา โดยการจัดกลุ่มสัญลักษณ์ของบีเพลและเทียบเคียงว่าเข้าข่ายเป็นแบบรูปใด จากนั้นก็ทำการสร้างโพรเมลาที่มีความหมายหรือการทำงานในลักษณะเดียวกัน รูปที่ 7.32 แสดงลักษณะแบบรูปกระแสงานต่างๆ และโพรเมลาที่สอดคล้องกัน แบบรูปกระแสงานที่ครอบคลุม คือ

- แบบรูปแบบลำดับ (sequence pattern)
- แบบรูปผสานพื้นฐาน (simple merge pattern)
- แบบรูปการประสาน (synchronization pattern)
- แบบรูปทางเลือกอย่างใดอย่างหนึ่ง (exclusive choice pattern)
- แบบรูปการแยกขนาน (parallel split pattern)



Workflow Pattern	3-Node Flow Paths	PROMELA Code

Sequence Pattern	a b c	proctype b(){
		int x;
		recv(qs[0],x);
		send(qs[1],x);
		}

Simple Merge Pattern	c e f
	d e f
	e f end
		proctype f(){
		int x;
		chan qs1[2] = [1] of {int};
		qs1[0] = qs[0];
		qs1[1] = qs[1];
		simpleMerge(qs1,2,x);
		send(qs[2],x);
		}

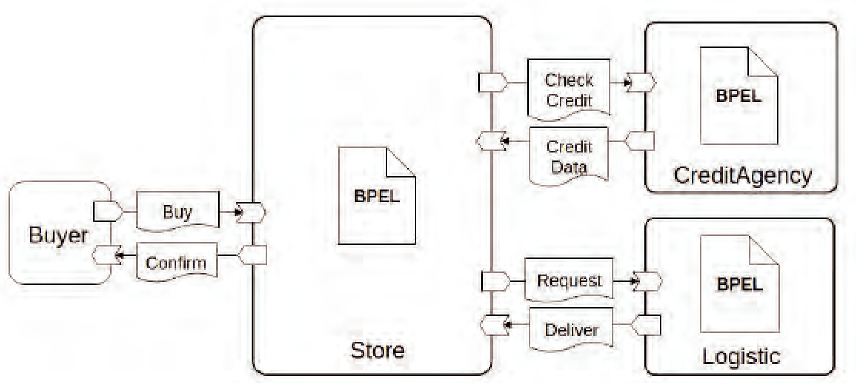
Synchronizatio nPattern	c e f
	d e f
	e f end
		proctype f(){
		int x;
		chan qs1[2] = [1] of {int};
		qs1[0] = qs[0];
		qs1[1] = qs[1];
		int z1[2];
		synchronization(qs1,2,z1);
		int totalMessage;
		totalMessage =z1[0]+z1[1];
		atomic{
		if
		:: totalMessage == 2 -> send(qs[2],x);
		:: totalMessage != 2 -> skip;
		fi;
		}
		}

Exclusive Choice Pattern	start a b
	a b d
	a b c
		proctype a(){
		int x; int y;
		recv(qs[0],x);
		chan qs1[2] = [1] of {int};
		qs1[0] = qs[1];
		qs1[1] = qs[2];
		if
		::y=0;
		::y=1;
		fi;
		exclusiveChoice(qs2,2,y,msg);
		}

Parallel Split Pattern	start a b
	a b c
	a b d
		proctype a(){
		int x;
		recv(qs[0],x);
		chan qs1[2] = [1] of {int};
		qs1[0] = qs[1];
		qs1[1] = qs[2];
		int z0[2];
		z0[0] = x;
		z0[1] = x;
		parallelSplit(qs0,2,z0);
		}

รูปที่ 7.32 แบบรูปกระแสงานที่นำมาแปลงเป็นโพรเมลาในคอลัมน์ขวาสุด [37]

นอกจากนี้ ในขณะที่การออกแบบระบบในระดับภาพรวมของการทำงานร่วมกันระหว่างเซอร์วิสมักจะใช้แผนภาพทอสกา (topology and orchestration specification for cloud application: TOSCA) ซึ่งเป็นแผนภาพแสดงถึงการทำงานร่วมกันของเซอร์วิส



รูปที่ 7.33 ตัวอย่างแผนภาพทอสการะบบการขายสินค้าโดยเซอร์วิสชื่อ "Store" ที่เขียนด้วยบีเพลติดต่อประสานกับเซอร์วิสรอบ ๆ [38]

สำหรับการเขียนอธิบายแบบจำลองเชิงรูปนัยนั้น งานวิจัย [38] ได้นำเสนอการแปลงแผนภาพทอสกาเป็นโพรเมลาด้วยเช่นกันโดยเน้นแปลงส่วนการเชื่อมต่อระหว่างเซอร์วิสที่ติดต่อกันโดยตรง และใช้ในการทวนสอบการประสานข้อมูลกันระหว่างเซอร์วิสที่มีในแผนภาพทอสกา เช่นการทวนสอบว่าแผนภาพทอสกามีการติดตายเกิดขึ้นหรือไม่ เป็นต้น หลักการที่ใช้ในการสร้างโพรเมลาคือการวิเคราะห์แผนภาพทอสกาและบีเพลที่กำหนดมาให้เบื้องต้นเพื่อหาช่องทางในการส่งข้อมูลติดต่อกัน เนื่องจากแผนภาพทอสกาและบีเพลมีลักษณะจัดเก็บเป็นแฟ้มเอ็กซ์เอ็มแอล และในแฟ้มบีเพลเราสามารถตรวจหา *PartnerLink* ที่ติดต่อกันและมีการกำหนดช่องทางเชื่อมต่อกัน เช่น ชุดคำสั่งการเรียกและให้บริการของบีเพลที่ใช้ *<bpel:invoke ...>* หรือ *<bpel:receive ...>* และตรวจประเภทของชนิดตัวแปรที่ส่งไปมาจากแฟ้มประกาศการเชื่อมต่อแบบ WSDL ที่เกี่ยวข้องจาก *<bpel:import ...>* เป็นต้น ในขณะที่ในแฟ้มทอสกาเราสามารถตรวจหา *Element* ที่ชื่อ *RequirementType*, *CapabilityType* และ *RelationshipType* ได้เช่นกัน ตัวอย่างการแปลงเซอร์วิสชื่อ "Store" จากแผนภาพทอสกาและบีเพลจากรูปที่ 7.33 มาเป็นโพรเมลาชื่อ *StoreService()* ได้ในรูปที่ 7.34

```
chan chan_requestbuy = [0] of {mtype};
chan chan_buyconfirm = [0] of {mtype};

vars_bts req_requestbuy;
vars_stb res_buyconfirm;

proctype StoreService() {

    chan_requestbuy ? MSG_TYPE_REQUESTBUY;

    run CreditAgencyService();

    req_checkcredit.c_req.id = req_bts.b_req.id;
    req_checkcredit.c_req.customer = req_bts.b_req.customer;

    chan_checkcredit ! MSG_TYPE_CHECKCREDIT;
    chan_creditdata ? MSG_TYPE_CREDITDATA;

    if
    :: (res_creditdata.c_res.rating > 5) -> ;
      run LogisticsService();

      req_requestdeliver.d_req.id = res_creditdata.c_res.id;

      chan_requestdeliver ! MSG_TYPE_REQUESTDELIVER;
      chan_deliverinfor ? MSG_TYPE_DELIVERINFOR;

      res_buyconfirm.b_res.id = res_deliverinfor.d_res.id;

      chan_buyconfirm ! MSG_TYPE_BUYCONFIRM;

    :: else -> ;

      res_buyconfirm.b_fault.id = res_creditdata.c_res.id;

      chan_buyconfirm ! MSG_TYPE_BUYCONFIRM;

    fi

}
```

รูปที่ 7.34 ตัวอย่างโพรเมลาสำหรับเซอร์วิสชื่อ “Store” ที่แปลงมาได้ [38]

การแปลงโปรแกรมจาวาเป็นโพรเมลาเพื่อการทวนสอบ

สำหรับระบบที่พัฒนาไปแล้วเป็นโปรแกรมที่เขียนใช้งานแล้ว และมีความต้องการนำกลับมาทวนสอบอีกรอบ ก็มีงานวิจัยจำนวนหนึ่งในนำเสนอวิธีการทำวิศวกรรมย้อนรอย (reverse engineering) โดยการแปลงโปรแกรมย้อนกลับมาเป็นแบบจำลองเชิงรูปนัยที่แสดงถึงการออกแบบระบบ ในที่นี้ขอนำเสนองานวิจัยที่ทำการแปลงโปรแกรมจาวาย้อนกลับมาเป็นโพรเมลาเพื่อทำการทวนสอบ [39] หลักการในการแปลงใช้แบบเดียวกับตัวแปลภาษา คือ การสร้างตัวแปลภาษาจากภาษาจาวามาเป็นภาษาโพรเมลานั่นเอง โดยการพิจารณาโครงสร้างของประโยคคำสั่งของทั้งสองภาษา รูปที่ 7.35 แสดงโครงสร้างของภาษาโพรเมลาเบื้องต้นที่ประกอบด้วยการประกาศ *proctype* ประกาศพารามิเตอร์และส่วนเนื้อโปรแกรม

```
                  Name        Parameter
proctype Example (chan input: chan output) {
   int count:
   int a:         } Local variable
   do
   :: count = count+1
   if
Body :: count == 10 -> skip
   :: else -> a = a+count
   Fi
   od
}
```

รูปที่ 7.35 ตัวอย่างโครงสร้างของ proctype ในโพรเมลา [39]

งานวิจัยทำการพิจารณาส่วนกระแสการควบคุม (control flow) พื้นฐานที่มีในทั้งสองภาษาและทำการเปรียบเทียบไว้ในรูปที่ 7.36 เพื่อการแปลงเมื่อพบโครงสร้างกระแสการควบคุมของภาษาจาวา ก็แปลงไปเป็นโพรเมลา

Basic Control Flow	Promela Statements

atomic Sequence	atomic{
	statement_1;
	statement_2;
	}

if statement	if :: (a != b) -> statement_1; :: (a == b) -> statement_2 fi
do-loop	do :: count = count +1 :: a = b+2 :: (count == 0) -> break od

for-loop	for(int i=0; i<5; i++){
	statements;
	}

รูปที่ 7.36 ตัวอย่างการแปลงกระแสการควบคุมจากภาษาจาวาเป็นโพรเมลา [39]

อย่างไรก็ตามโดยเทคนิคการเขียนโพรเมลาสามารถทำได้โดยตรงและโดยอ้อม การเขียนโพรเมลาสามารถเขียน *proctype* ที่ทำหน้าที่คล้ายฟังก์ชันการทำงานของเส้นโยงใย ทำได้โดยอ้อมด้วยการใช้คำสั่ง *#define* กล่าวคือการใช้ *#define* คือการกำหนดคำสั่งให้ทำงานแบบ *Macro* นั่นเอง โดยการแปลงให้คลาสแต่ละคลาสในโปรแกรมมาเป็นฟังก์ชันในโพรเมลา และการเรียกบริการของคลาสเป็นการเรียกใช้ฟังก์ชันนั่นเอง ในงานวิจัยนี้ได้ใช้แกรมมา (grammar) ของ

ภาษาจาวา เขียน *Parser* ภาษาจาวา และเขียนเครื่องมือในการแปลงไปเป็นโพรเมลาจากผลการทำ *Parser* แต่ละประโยคในโปรแกรมจาวา ตัวอย่างโปรแกรมภาษาจาวาต้นฉบับแสดงในรูปที่ 7.37 และโพรเมลาที่แปลงออกมาได้ในการเขียนแบบ *#define* แสดงในรูปที่ 7.38

```
class X {
   public int x;
   public X() { x=0; }
   public void add2x (int d) { x = x+d;
   System.out.println("x = "+x);}
}

class XY extends X {
   public int y;
   public XY() { y = 0; }
   public void add2y (int d) { y=y+d;
      System.out.println("y = "+y);}
   public void add(int dx, int dy) {
      int old_x = x;
      int old_y = y;
      add2x(dx);
      add2y(dy);
   }
}

class Adder extends Thread {
   private XY xy;
   public Adder(XY xy) { this.xy = xy; }
   public void run() { xy.add(4,4); }
}

class Main {
   public static void main(String[] args) {
      XY xy = new XY();
      Adder adder1 = new Adder(xy);
      Adder adder2 = new Adder(xy);
      adder1.start();
      adder2.start();
   }
}
```

รูปที่ 7.37 ตัวอย่างโปรแกรมภาษาจาวาต้นฉบับก่อนการแปลง [39]

```
#define ClassName mtype
#define Index byte
#define undefined 0
#define MAX 5
#define null -1
#define this _pid
ClassName = {X,XY,Adder,Main}
typedef ObjRef{ClassName class; Index index};
typedef X_Class{int x; };
X_Class X_Obj[MAX];
Index X_Next = 0;
#define X_get_x(obj)
   (obj.class == X -> X_Obj[obj.index].x :
   (obj.class == XY -> XY_Obj[obj.index].x : undefined))
#define X_set_x(obj,value)
   if :: obj.class == X -> X_Obj[obj.index].x = value
   :: obj.class == XY -> XY_Obj[obj.index].x = value fi
#define X_constr(obj)
   ObjRef obj; obj.class = X;
   atomic{obj.index = X_Next; X_Next++};
   X_set_x(obj,0);
#define X_add2x(obj,d)
   X_set_x(obj,X_get_x(obj)+d);
typedef XY_Class{int y; int x; };
XY_Class XY_Obj[MAX];
Index XY_Next = 0;
#define XY_get_y(obj)
   XY_Obj[obj.index].y
#define XY_set_y(obj,value)
   XY_Obj[obj.index].y = value
#define XY_constr(obj)
   ObjRef obj; obj.class = XY;
   atomic{obj.index = XY_Next; XY_Next++};
   X_set_x(obj,0); XY_set_y(obj,0);
#define XY_add2y(obj,d)
   XY_set_y(obj,XY_get_y(obj)+d);
#define XY_add(obj,dx,dy)
   old_x = X_get_x(obj); old_y = XY_get_y(obj);
   X_add2x(obj,dx); XY_add2y(obj,dy);
typedef Adder_Class{ObjRef xy};
Adder_Class Adder_Obj[MAX];
Index Adder_Next = 0;
#define Adder_get_xy(obj)
   Adder_Obj[obj.index].xy
#define Adder_set_xy(obj,value)
   Adder_Obj[obj.index].xy.class = value.class;
   Adder_Obj[obj.index].xy.index = value.index;
#define Adder_constr(obj,xy)
   ObjRef obj; obj.class = Adder;
   atomic{obj.index = Adder_Next; Adder_Next++};
   Adder_set_xy(obj,Adder_get_xy(obj));
proctype Adder_Thread(ObjRef obj){
   int old_x; int old_y; XY_add(Adder_get_xy(obj),4,4);
}
init{int old_x; int old_y; XY_constr(xy);
   Adder_constr(adder1,xy); Adder_constr(adder2,xy);
   run Adder_Thread(adder1); run Adder_Thread(adder2);
}
```

รูปที่ 7.38 ตัวอย่างโปรแกรมโพรเมลาหลังการแปลง [39]

7.6 แบบฝึกหัด

1) โมเดลเช็กกิงมีความแตกต่างกับการพิสูจน์แบบทฤษฎีบทอย่างไร?
2) เราเริ่มต้นสร้างแบบจำลองเชิงรูปนัยของระบบขึ้นมาได้อย่างไร อธิบายวิธี?
3) ปริภูมิสถานะของระบบที่ต้องการทวนสอบคืออะไร มีประโยชน์อย่างไร?
4) จงอธิบายวิธีการแจกแจงต้นไม้ปริภูมิสถานะ?
5) ปริภูมิสถานะแบบต้นไม้ต่างกับแบบกราฟอย่างไร ข้อดีข้อเสียเป็นอย่างไร?
6) ถ้าระบบที่ต้องการทวนสอบมีขนาดใหญ่และซับซ้อน การทวนสอบมักจะเกิดปัญหาใด และเราสามารถทวนสอบได้หรือไม่?
7) คุณลักษณะพื้นฐานทั่วไปที่ควรทวนสอบคืออะไร ยกตัวอย่าง?
8) โมเดลเช็กกิงตามทฤษฎีออโตมาตาคืออะไร อธิบาย?
9) ตัวอย่างค้านคืออะไร มีประโยชน์อย่างไร?
10) แนวคิดในการใช้แผนภาพบีเพลช่วยในการสร้างแบบจำลองเชิงรูปนัย ทำได้อย่างไร?
11) แนวคิดในการใช้แผนภาพกิจกรรมในยูเอ็มแอลมาช่วยสร้างแบบจำลองเชิงรูปนัยทำได้อย่างไร?
12) การทวนสอบโปรแกรมที่เขียนเสร็จแล้ว ทำได้อย่างไร?
13) การทดสอบระบบเชิงพร้อมกันที่มีหลายสายโยงใยทำงานประสานกันด้วยกรณีทดสอบทำได้หรือไม่ มีอุปสรรคอย่างไร?
14) โพรเมลาจำลองการทำงานของระบบที่มีลักษณะเชิงไม่กำหนด (non-deterministic) ได้หรือไม่ จงยกตัวอย่างอธิบาย?
15) เราทวนสอบวงจรฮาร์ดแวร์ในระดับเกตได้อย่างไรบ้าง?
16) จงเขียนโพรเมลาอธิบายระบบสัญญาณไฟจราจรที่สี่แยก ที่มีการใช้สัญญาณไฟจราจรสองชุดที่ทำงานประสานกันเพื่อให้รถยนต์สามารถผ่านไปมาได้โดยไม่เกิดอุบัติเหตุ พร้อมหาคุณลักษณะที่ใช้ทวนสอบเรื่องความปลอดภัยด้วย?

บรรณานุกรม

[1] Pressman, R. S. (2014). *Software Engineering: A Practitioner's Approach, 8th Edition*, McGraw-Hill Education.

[2] Shirey, R. (2000). *RFC2828 Internet Security Glossary*, Retrieved from http://rfc.net/rfc2828.html

[3] *15288-2015 - ISO/IEC/IEEE International Standard - Systems and software engineering -- System life cycle processes*, (2015). IEEE.

[4] Saaltink, M. (1997). The Z/EVES User’s Guide, TR-97-5493-06, ORA Canada.

[5] Baier, C., & Katoen, J. (2008). Principle of Model Checking, MIT Press.

[6] Fisher, M. (2011). *An Introduction to Practical Formal Methods Using Temporal Logic*, Wiley.

[7] Ben-Ari, M. (2008). *Principle of the SPIN Model Checker*, Springer.

[8] Meisels, I., & Salltink, M. (1997). *The Z/EVES Reference Manual, TR-97-5493-03d, Technical Report*, ORA Canada.

[9] Jastram M. (2014). *Rodin User’s Handbook*, CreateSpace Independent Publishing Platform.

[10] Bonet, P., Llado, M., Puigjaner R., & Knottenbelt W. (2007). *PIPE v2.5: a Petri Net Tool for Performance Modeling*, Retrieved from http://www.doc.ic.ac.uk/~wjk/publications/bonet-llado-knottenbelt-puijaner-clei-2007.pdf

[11] Jensen K., & Kristensen L. (2009). *Coloured Petri Nets: Modelling and Validation of Current Systems*, Springer.

[12] Rao, C. V. S. (2005). *Switching Theory and Logic Design*, Pearson.

[13] Knorr W., et.al. (2018). *Mathematics*, Encyclopedia Britannica, Retrieved from www.britannica.com/science/mathematics

[14] Ben-Ari, M. (2001). *Mathematical Logic for Computer Science*, Springer.

[15] Hurley, P.J. (2002). *A Concise Introduction to Logic*, Wadsworth Publishing.

[16] Manna, Z., & Pnueli A. (1992). *The Temporal Logic of Reactive and Concurrent Systems: Specification*, Springer.

[17] Sukvanich, P., Thongtak, & A. Vatanawood, W. (2016). *Translating Basic Metric Temporal Logic Formulas into Promela*, Presented at The 7th International Conference on Information Science and Applications 2016, ICISA 2016, Ho Chi Minh City, Vietnam.

[18] Bowen, J. (1996). *Formal Specification and Documentation Using Z – A Case Study Approach*, International Thomson Computer Press.

[19] Wookcock, J., & Davies, J. (1996). *Using Z Specification, Refinement, and Proof*, Prentice Hall.

[20] Spivey, J.M. (1989), *The Z Notation: A Reference Manual*, Prentice Hall.

[21] Harry, A. (1996). *Formal Methods Fact File: VDM and Z*, John Wiley & Sons.

[22] Schmeider, S. (2001). *the b-method- an introduction*, Palgrave.

[23] Diaconescu, R., Futatsugi, K. (1998). *CafeOBJ Report*, World Scientific.

[24] Aziz, A., et al. (2004). *Automata Theory*, Retrieved from https://cs.stanford.edu/people/eroberts/courses/soco/projects/2004-05/automata-theory/basics.html

[25] Lewis, H., & Papadimitriou C. (1997). *Elements of the Theory of Computation*, Prentice-Hall.

[26] Reisig, W. (2013). *Understanding Petri Nets: Modeling Techniques, Analysis Methods, Case Studies*, Springer.

[27] Saaltink, M. (1997). *Z/EVES System*, Presented at The 10th International Conference of Z Users Reading, UK.

[28] Damjan, P., & Vatanawood, W. (2017, March). *Translating UML State Machine Diagram into Promela*, Paper presented at The International MultiConference of Engineering and Computer Scientists 2017, Hong Kong.

[29] Fowler, M. (2003). *UML Distrilled: A Brief Guide to the Standard Object Modeling Language*, Addison-Wesley Professional.

[30] Sukvanich, P., Thongtak, A., & Vatanawood, W. (2015, November). *Formalizing Real-Time Embedded System into Promela*, Presented at The 4th International Conference on Mechanics and Control Engineering 2015, ICMCE 2015, Lisbon, Portugal.

[31] Lawsunnee, W., Thongtak, A., & and Vatanawood, W. (2015, March). *Signal Persistence Checking of Asynchronous System Implementation using SPIN*, Presented at The International MultiConference of Engineering and Computer Scientists 2015 Vol. II, IMECS 2015, Hong Kong.

[32] Park, S. (1996). *Synthesis of Asynchronous VLSI circuit from Signal Transition Graph Specifications*, Ph.D. Dissertation, Tokyo Institute of Technology.

[33] Boonroeangkaow, K., Thongtak, A., & Vatanawood, W. (2017, March). *Formal Modeling for Persistence Checking of Signal Transition Graph Specification with Promela*, Presented at The International MultiConference of Engineering and Computer Scientists 2017, IMECS 2017, Hong Kong.

[34] Maneerat, N., and Vatanawood, W. (2016, July). Translation UML Activity Diagram into Colored Petri Net with Inscription, Presented at The 13th International Joint Conference on Computer Science and Software Engineering (JCSSE 2016), Khon Kaen, Thailand.

[35] Deesukying, J., & Vatanawood, W. (2016, June). *Generating of Business Rules for Coloured Petri Nets*, Presented at The IEEE/ACIS 15th International Conference on Computer and Information Science, ICIS 2016, Okayama, Japan.

[36] Juric, M., & Pant, K. (2008). *Business Process Driven SOA using BPMN and BPEL*, Packt Publishing.

[37] Yamasathien, S., & Vatanawood, W. (2014, May). *An Approach to Construct Formal Model of Business Process Model from BPMN Workflow Patterns*, Presented at The Fourth International Conference on Digital Information and Communication Technology and its Applications (DICTAP2014), Bangkok, Thailand.

[38] Chareonsuk, W., & Vatanawood, W. (2016, June). *Formal Verification of Cloud Orchestration Design with TOSCA and BPEL*, Presented at The 13th International Conference on Electrical Engineering/Electronics, Computer, Telecommunications and Information Technology, ECTI-CON 2016, Chiang Mai, Thailand.

[39] Lueangviriyayarn, J., & Vatanawood, W. (2015, March). *Automated Translation of Java Control Flow into Promela*, Presented at The International MultiConference of Engineering and Computer Scientists 2015 Vol. I, IMECS 2015, Hong Kong.

[40] Grumberg, O., & Veith, H. (Eds.) (2008). *25 Years of Model Checking: History, Achievements, Perspectives*, Springer.

[41] Drechsler, R. (Ed.) (2004). *Advanced Formal Verification*, Kluwer Academic Publishers.

[40] Clarke, E., Grumberg, O., & Peled D. (2000). *Model Checking*, MIT Press.

[42] Ganai, M., & Gupta, A. (2007). *SAT-Based Scalable Formal Verification Solutions*, Springer.

[43] Ray, S. (2010). *Scalable Techniques for Formal Verification*, Springer.

[44] Jones, R. (2002). *Symbolic Simulation Methods for Industrial Formal Verification*, Springer.

ดัชนี

ก

กฎการแปลง, 210
กฎการอนุมาน, 31
กรอบการพัฒนาระบบ, 2
กระบวนการ, 138
กราฟการเข้าถึง, 160
กราฟปริภูมิสถานะ, 185
การแทรกสลับ, 198
การกำกับข้อความ, 151
การกำหนดคุณลักษณะ, 187
การกำหนดปริมาณ, 43
การดำเนินการ, 79
การตอบสนอง, 189
การตีความ, 31, 42
การตีความ, 55, 56
การตีความประพจน์, 38
การทวนสอบเชิงรูปนัย, 182
การพิสูจน์ทฤษฎีบท, 163
การพิสูจน์ทางตรรกศาสตร์, 31
กีเวนเซต, 163

ข

ข้อเท็จจริง, 31
ข้อกำหนดเชิงรูปนัย, 5, 7
ข้อกำหนดแบบโมเดลเบส, 74
ข้อกำหนดแบบพรอพเพอร์ตีเบส, 75
ข้อกำหนดหย่อน, 79
ข้อความจารึก, 220
ข้ออ้างเหตุผล, 38
ขั้นตอนอุปมาน, 32

ค

เครื่องเชิงนามธรรม, 108
เครื่องมือสปิน, 195
แคลคูลัสเชิงตรรกะ, 30
แคลคูลัสภาคแสดง, 43
โครงสร้างคริปกี, 149
คณิตตรรกศาสตร์, 27
ควบคุมคุณภาพ, 3
ความเป็นธรรม, 189
ความดำเนินชีวิต, 188
ความปลอดภัย, 188
ความพอใจ, 42
ความสมมูล, 65
ความสัมพันธ์, 49
ความหมาย, 35
คัลเลอร์เพทริเน็ต, 220
คุณลักษณะเชิงเวลา, 206
เงื่อนไขก่อน, 79, 84, 111
เงื่อนไขหลัง, 80, 85
เซต, 44
เซตกำลัง, 49

ต

ต้นไม้การคำนวณ, 185
ต้นไม้ปริภูมิสถานะ, 185
ตรรกบท, 29
ตรรกศาสตร์เชิงเวลา, 53
ตรรกศาสตร์เชิงเวลาแบบเส้นตรง, 59

ตรรกศาสตร์เชิงเวลาประพจน์, 54, 60
ตรรกศาสตร์เชิงประพจน์, 5,31,55,85
ตรรกศาสตร์เชิงสัญลักษณ์, 30
ตรรกศาสตร์ต้นไม้การคำนวณ, 59
ตรรกศาสตร์ภาคแสดง, 42
ตรรกศาสตร์อัญรูป, 31
ตรรกศาสตร์อันดับที่ศูนย์, 34
ตรรกศาสตร์อันดับที่หนึ่ง, 34
ตัวแทนเชิงนามธรรม, 7
ตัวแปรเสรี, 42
ตัวแปรประพจน์, 34
ตัวดำเนินการเชิงเวลา, 56
ตัวยอมรับ, 146
ตัวยืนยง, 78,173
ตารางการตัดสินใจ, 224
ตารางค่าความจริง, 39

ท

ทฤษฎีการจำลองแบบ, 31
ทฤษฎีการพิสูจน์, 31
ทฤษฎีบท, 31
ทวนสอบเชิงรูปนัย, 4
ทวนสอบเชิงอรูปนัย, 4
ทันกาล, 5

น

นำมาซึ่ง, 37
นิเสธของสูตรเชิงเวลา, 66

บ

แบบจำลองเชิงรูปนัย, 5, 7
แบบรูปกระแสงาน, 225
แบบสมเหตุสมผล, 38
บูชีออโตมาตา, 152,194

ป

เป็นที่พอใจได้, 42
เป็นประจำอย่างนับไม่ถ้วน, 153
ประเภทของตรรกศาสตร์เชิงเวลา, 58
ประโยคภาคแสดง, 42
ประชุมแบบตรวจตลอด, 4
ประพจน์, 42
ประพจน์เดี่ยว, 33
ประพจน์ประกอบ, 33
ปริภูมิสถานะ, 184

ผ

แผนภาพเครื่องสถานะ, 208
แผนภาพเอสทีจี, 215
แผนภาพกิจกรรม, 221
ผแผนภาพทอสกา, 227
แผนภาพบีเพล, 225
แผนภาพยูเอ็มแอลมา, 207
ผลคูณคาร์ทีเชียน, 48
ผลคูณบูชีออโตมาตา, 194

พ

เพทริเน็ต, 156
เพรดิเคต, 35, 44
พรอพเพอร์ตีเบส, 73
พันธกรณีการพิสูจน์, 177
พิสูจน์ความถูกต้อง, 6
พิสูจน์ทางคณิตตรรกศาสตร์, 4
พีชคณิตบูลีน, 50

ภ

ภาษาเชิงรูปนัย, 73
ภาษาเซด, 75, 85
ภาษาเอ็มแอล, 224
ภาษาโพรเมลา, 7,69,137,212,228
ภาษาโอซีแอล, 208
ภาษาคาเฟโอบีเจ, 76, 116
ภาษาธรรมชาติ, 77
ภาษาวีดีเอ็ม-เอสแอล, 75

ม

ไม่เป็นที่พอใจได้, 42

โมเดล, 38,55, 62

โมเดลเช็กกิง, 179

โมเดลเบส, 73

มอดูลแบบตึง, 117

มอดูลแบบหย่อน, 117, 119

มาร์กกิง, 160

มาร์กกิงแรก, 158

ร

ระบบเชิงเวลาจริงแบบฝังตัว, 211

ระบบวิกฤติ, 5

ระบบสัญญาณไฟจราจร, 64,200

ว

วากยสัมพันธ์, 31, 35

วิธีเชิงรูปนัย, 4

วิธีเชิงรูปนัยบี, 76

วิธีพิสูจน์, 7

ส

n-สิ่งอันดับ, 48

สัจนิรันดร์, 39,63

สัญญาณนาฬิกา, 67

สายโยงใย, 7,139,147,198

สูตรเชิงเวลา, 56, 57

สูตรเชิงประพจน์, 36, 42

อ

เอเอ็มเอ็น, 108

เอ็มทีแอล, 66, 69

อนุมานแบบตรวจสอบ, 33

อนุมานแบบนิรนัย, 32, 39

อนุมานแบบอุปนัย, 32

อริสโตเติล, 29

ออโตมาตา, 143

ออโตมาตาเชิงไม่กำหนด, 148,183

ออโตมาตาเชิงกำหนด, 147, 148

Formal Verification
ISBN 978-616-474-543-8
9 786164 745438