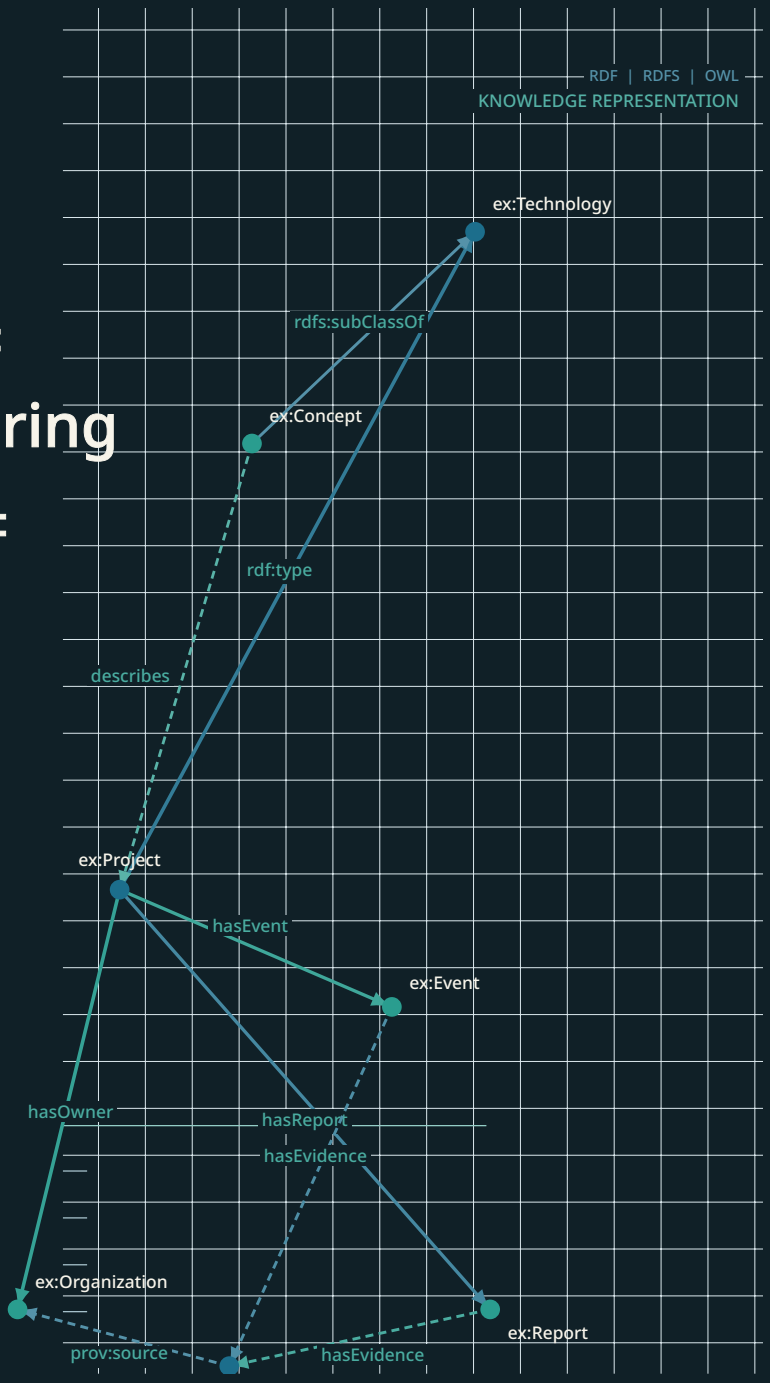
RDF | OWL | SPARQL | SHACL

Semantic Web และ Ontology Engineering หลักการออกแบบ และการสร้าง Knowledge Graph



ชินพงศ์ อังสุโชติเมธี

คณะวิทยาศาสตร์ มหาวิทยาลัยสงขลานครินทร์

RDF / OWL / SPARQL / SHACL

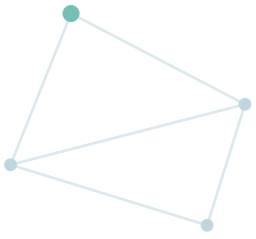
Semantic Web และ Ontology Engineering

หลักการออกแบบ และการสร้าง Knowledge Graph

ผู้เขียน

ชินพงศ์ อังสุโชติเมธี

คณะวิทยาศาสตร์
มหาวิทยาลัยสงขลานครินทร์



เผยแพร่ครั้งที่ 1

ตุลาคม พ.ศ. 2569

เผยแพร่ในรูปแบบ e-book

ลิขสิทธิ์และการเผยแพร่

Semantic Web และ Ontology Engineering: หลักการออกแบบและการสร้าง Knowledge Graph

จัดทำโดย ชินพงศ์ อังสุโชติเมธี

เจ้าของลิขสิทธิ์

© พ.ศ. 2569 ชินพงศ์ อังสุโชติเมธี
และคณะวิทยาศาสตร์ มหาวิทยาลัยสงขลานครินทร์

การใช้เพื่อการศึกษา

หนังสือฉบับนี้จัดทำและเผยแพร่ในรูปแบบ e-book เพื่อแจกจ่ายสำหรับการศึกษาและการเรียนรู้ โดยไม่คิดค่าใช้จ่าย

อนุญาตให้ทำซ้ำและแจกจ่ายหนังสือฉบับนี้ได้ทั้งเล่มหรือบางส่วนเพื่อการศึกษา โดยระบุชื่อผู้เขียน ชื่อหนังสือ และเจ้าของลิขสิทธิ์อย่างชัดเจน พร้อมคงข้อความแสดงความเป็นเจ้าของลิขสิทธิ์ไว้ในสำเนา

ลิขสิทธิ์ในหนังสือยังคงเป็นของชินพงศ์ อังสุโชติเมธี และคณะวิทยาศาสตร์ มหาวิทยาลัยสงขลานครินทร์ การอนุญาตให้ทำซ้ำและแจกจ่ายนี้มิใช่การโอนลิขสิทธิ์ ห้ามแอบอ้างว่าหนังสือหรือเนื้อหาเป็นผลงานของตน หรืออ้างตนเป็นเจ้าของลิขสิทธิ์เนื้อหาและภาพจากแหล่งอื่นที่ระบุที่มายังคงเป็นสิทธิของเจ้าของเดิม การใช้ส่วนดังกล่าวอยู่ภายใต้เงื่อนไขของเจ้าของสิทธินั้น

ข้อมูลทางบรรณานุกรมของหอสมุดแห่งชาติ

National Library of Thailand Cataloging in Publication data

ชินพงศ์ อังสุโชติเมธี.

Semantic Web และ Ontology Engineering: หลักการออกแบบและการสร้าง Knowledge Graph.-- สงขลา : [ม.ป.พ.], 2569.

155 หน้า.

1. เว็บเชิงความหมาย. I. ชื่อเรื่อง.

004.678

ISBN(e-book) 978-616-642-252-8

เผยแพร่ครั้งที่ 1 | ตุลาคม พ.ศ. 2569

รูปแบบ e-book

คำนำ

เหตุผลที่ผมเขียนหนังสือเล่มนี้เกิดจากเดิมผมเป็นโปรแกรมเมอร์ด้านระบบโทรคมนาคมผ่านเครือข่าย IP แต่ต้องมาทำวิจัยด้าน Semantic Web และ Ontology ซึ่งเป็นสาขาที่มีมาตรฐานมากมายให้เรียนรู้และทำความเข้าใจ ทั้ง RDF, RDFS, OWL, SPARQL และ SHACL การค่อย ๆ อ่านมาตรฐานและทำความเข้าใจทั้งทฤษฎีและการใช้งานใช้เวลานานมากสำหรับผม ผมจึงนำสิ่งที่ได้เรียนรู้มาสรุปเป็นหนังสือเล่มนี้ โดยหวังว่าจะช่วยให้ผู้เริ่มต้นเข้าสู่สายงานวิจัยด้านนี้ได้เร็วขึ้น

หนังสือเล่มนี้มุ่งช่วยนักศึกษาที่ไม่เคยเรียน Semantic Web นักศึกษาบัณฑิตศึกษาและนักวิจัยระยะเริ่มต้น ตลอดจนนักพัฒนาที่ต้องการเข้าใจหลักคิดก่อนลงมือ เนื้อหาพยายามตอบคำถามหลักสามข้อ

- Semantic Web และ Ontology คืออะไร และต่างจากฐานข้อมูลหรือ Class Diagram อย่างไร
- First-Order Logic, RDF, RDFS, OWL, SPARQL, SHACL และ Provenance แบ่งหน้าที่และเชื่อมต่อกันอย่างไร
- เมื่อออกแบบ Ontology หรือ Knowledge Graph ควรถามอะไร และจะหลีกเลี่ยงการกล่าวเกินหลักฐานได้อย่างไร

หนังสือเล่มนี้ไม่ใช่คู่มือพัฒนาระบบด้วยเครื่องมือใดเครื่องมือหนึ่ง ตัวอย่าง Turtle, OWL, SPARQL และ SHACL มีไว้ให้เห็นว่าหลักคิดที่อธิบายนำไปเขียนเป็นข้อมูลและข้อกำหนดได้อย่างไร โดยไม่จำเป็นต้องจดจำทุกบรรทัด เนื้อหาเริ่มจากการมองข้อมูลเป็นกราฟ แล้วค่อยเชื่อมไปยัง Ontology Engineering, Knowledge Graph, Ontology Alignment, การบันทึกที่มาของข้อมูล และการกำกับดูแลความหมายของข้อมูล

ผมไม่ได้คาดหวังว่าหนังสือเล่มเดียวจะทำให้ Ontology กลายเป็นเรื่องง่ายทั้งหมด เพียงหวังว่าเมื่ออ่านจบแล้ว ผู้อ่านจะเห็นว่าควรถามอะไร รู้ว่ามาตรฐานแต่ละส่วนทำหน้าที่ใด และเสียเวลากับความสับสนน้อยกว่าที่ผมเคยพบมาก่อน

ชินพงศ์ อังสุโชติเมธี
ผู้เขียน

สารบัญ

คำนำ iv

สารบัญภาพ ix

สารบัญตาราง xi

1 จากเว็บเอกสารสู่ Semantic Web 1

1.1 เว็บของเอกสารและข้อจำกัดของ Hyperlink . . . 1
1.2 จากลิงก์ระหว่างเอกสารสู่กราฟของข้อมูล . . . 2
1.3 Resource Description Framework (RDF) . . . 3
1.4 Namespace, Prefix และการเขียน Turtle เบื้องต้น . . . 4
1.5 จาก RDF Graph สู่ Ontology . . . 7
1.6 บริบทที่หนึ่ง: Multimedia Sensor Network . . . 7
1.7 บริบทที่สอง: ความรู้ของโครงการเปลี่ยนผ่านสู่ดิจิทัล . . . 8
1.8 Semantic Web ช่วยให้ข้อมูลทำงานร่วมกันอย่างไร . . . 10

2 RDF และ RDFS: โครงสร้างข้อมูลและคำศัพท์มาตรฐาน 12

2.1 จากประโยคข้อมูลสู่ RDF Graph . . . 12
2.2 แยก IRI, Literal และ Namespace ให้ถูกบทบาท . . . 14
2.3 เมื่อ RDF ยังสื่อความหมายได้ไม่เพียงพอ . . . 15
2.4 จากคำศัพท์มาตรฐานสู่ Vocabulary ของงาน . . . 18
2.5 ผลอนุมานของ RDFS และสิ่งที่ต้องระวัง . . . 19
2.6 ข้อผิดพลาดที่พบบ่อยเมื่อเริ่มใช้ RDF/RDFS . . . 20
2.7 ขยายตัวอย่างข้อมูลโครงการด้วย RDF/RDFS . . . 21
2.8 เมื่อ RDF Graph ใหญ่ขึ้นและต้องรวมข้อมูล . . . 23
2.9 การตั้งชื่อ Predicate ให้คนและระบบเข้าใจตรงกัน . . . 26
2.10 สรุป . . . 27

3 OWL และ Ontology ที่มีความหมายเชิงตรรกะ 29

3.1 จาก Vocabulary ไปสู่ Ontology . . . 29
3.2 OWL, OWL 2 และ OWL DL ต่างกันอย่างไร . . . 30
3.3 องค์ประกอบและคำศัพท์มาตรฐานของ OWL 2 . . . 31
3.4 First-Order Logic และฐานตรรกะของ OWL . . . 34

3.5 Open World, Restriction และ Consistency . . . 39
3.6 มองกราฟเดียวกันผ่าน RDFS, OWL และ SHACL . . . 40
3.7 Restriction จากโดเมนโครงการ . . . 41
3.8 Disjointness และการตรวจความขัดแย้ง . . . 41
3.9 Cardinality และข้อจำกัดด้านจำนวน . . . 42
3.10 Property Characteristic และ Identity . . . 42
3.11 OWL Profile และการเลือกความซับซ้อน . . . 44
3.12 การอ่านผลของ Reasoner . . . 45
3.13 กรณีศึกษาแบบต่อเนื่อง: เพิ่ม OWL ตามความต้องการของระบบ . . . 45
3.14 ข้อผิดพลาดที่พบบ่อยในการใช้ OWL . . . 47
3.15 อ่าน Description Logic เพิ่มเติมจากตัวอย่าง . . . 48
3.16 OWL กับ SHACL ควรแบ่งหน้าที่อย่างไร . . . 50
3.17 สรุป . . . 53

4 SPARQL: การค้นคืนข้อมูลและความรู้จากกราฟ 55

4.1 SPARQL กับการถามคำถามต่อ Graph . . . 55
4.2 มุมมองเชิงคณิตศาสตร์ของ Query . . . 56
4.3 SELECT, FILTER, OPTIONAL และ ASK . . . 57
4.4 CONSTRUCT และการสร้าง Graph ใหม่ . . . 57
4.5 Query Catalog และ Evidence Retrieval . . . 58
4.6 การเขียน Query จากคำถามของผู้ใช้ . . . 58
4.7 OPTIONAL กับข้อมูลที่ยังไม่ครบ . . . 59
4.8 Aggregation และการสรุปผล . . . 59
4.9 Named Graph Query และ Evidence . . . 60
4.10 การตรวจ Query ก่อนนำไปใช้ . . . 61
4.11 ประสิทธิภาพและขอบเขตของ Query . . . 61
4.12 กรณีศึกษาแบบต่อเนื่อง: Query Catalog ของข้อมูลโครงการ . . . 62
4.13 Query Test และการอ่านผลลัพธ์อย่างมีวินัย . . . 64
4.14 SPARQL Update และวงจรชีวิตของ Named Graph . . . 65
4.15 จากคำถามภาษาไทยสู่ Graph Pattern และกรณีศึกษา . . . 66
4.16 สรุป . . . 68

5 กระบวนการวิศวกรรม Ontology 70

5.1 Ontology Engineering เริ่มจากคำถาม . . . 70
5.2 วงจรของการออกแบบ Ontology . . . 71
5.3 จากคำถามไปสู่ขอบเขตด้วย Competency Questions . . . 71
5.4 Minimal Commitment และการควบคุม Scope . . . 74
5.5 การ Review, Documentation และ Release . . . 74

5.6 Documentation ของ Ontology 75
5.7 Release Package ของ Ontology 75
5.8 จากปัญหาเฉพาะโดเมนสู่ Pattern ที่ใช้ข้ามโดเมน 76
5.9 การประเมิน Ontology ในฐานะผลงานทางวิชาการและระบบใช้งาน 78
5.10 รูปแบบการออกแบบที่ใช้บ่อยในงาน Ontology 79
5.11 บันทึกการออกแบบสำหรับผู้เริ่มต้นทำ Ontology 80
5.12 สรุป 81

6 การออกแบบ Ontology จากกรณีศึกษา MSSN-Onto และ Dx-Onto 83
6.1 จากกระบวนการวิศวกรรมสู่การตัดสินใจออกแบบ 83
6.2 MSSN-Onto: จาก Sensor สู่ข้อมูลสื่อและเหตุการณ์ 84
6.3 Dx-Onto: การวางโครงการเป็นแกนของความรู้ 88
6.4 การตัดสินใจที่เหมือนและต่างกัน 90
6.5 ขอบเขต ข้อจำกัด และบทเรียนสำหรับการนำไปใช้ 91
6.6 สรุป 92

7 Upper Domain Ontology และ Ontology Alignment 94
7.1 เหตุใด Ontology จึงต้องเชื่อมความหมายกับผู้อื่น 94
7.2 Alignment ในรูปความสัมพันธ์ทางคณิตศาสตร์ 95
7.3 DOLCE และ DUL ในฐานะกรอบจำแนกความหมาย 96
7.4 จาก Candidate Mapping สู่ Axiom ที่ตรวจสอบได้ 97
7.5 กรณี MSSN-Onto: สิ่งที่มีอยู่จริงและสิ่งที่ต้องพิจารณาเพิ่ม 98
7.6 การประเมิน Versioning และ Governance ของ Alignment 100
7.7 สรุป 101

8 การสร้างแอปพลิเคชัน Knowledge Graph 102
8.1 จาก Ontology ไปสู่ระบบที่ใช้งานได้จริง 102
8.2 Pipeline พื้นฐานของ Knowledge Graph 103
8.3 Identity, Validation และ Storage 104
8.4 ชั้นข้อมูลต้นทางและการทำ Source Inventory 105
8.5 Extraction Layer และการควบคุมความหมาย 106
8.6 Semantic Mapping Layer 106
8.7 Validation Layer ในฐานะ Gate 107
8.8 API, UI และผู้ใช้ปลายทาง 108
8.9 Deployment และ Versioning 109
8.10 กรณีศึกษาแบบต่อเนื่อง: จาก Prototype ไปสู่ระบบจริง 110
8.11 สถาปัตยกรรมขั้นต่ำที่ควรมีในระบบหนึ่ง 111

8.12 Walkthrough: หนึ่งข้อมูลไหลผ่านทั้งระบบ . . . 113
8.13 ข้อผิดพลาดตอนนำระบบขึ้นใช้งาน . . . 117
8.14 การติดตามและประเมิน Pipeline ในระบบจริง . . . 119
8.15 เครื่องมือสำหรับทดลองสร้างระบบ . . . 120
8.16 สรุป . . . 122

9 SHACL, Provenance และ Semantic Governance 124
9.1 ทำไม Graph ต้องถูกตรวจสอบ . . . 124
9.2 Validation ในรูปฟังก์ชัน . . . 124
9.3 Provenance และ Named Graph . . . 126
9.4 Semantic Governance ในระบบจริง . . . 127
9.5 Node Shape และ Property Shape . . . 127
9.6 Validation Report ในฐานะ Evidence . . . 128
9.7 Provenance Pattern สำหรับ Claim . . . 128
9.8 Governance Metric . . . 129
9.9 Governance ช่วยควบคุมการเปลี่ยนแปลง . . . 129
9.10 กรณีศึกษาแบบต่อเนื่อง: Governance ของข้อมูลโครงการ . . . 130
9.11 การจัดการการเปลี่ยนแปลงของ Ontology, Shape และ Query . . . 132
9.12 Validation Message ที่ผู้ใช้เข้าใจได้ . . . 133
9.13 Severity และการตัดสินใจ Publish . . . 134
9.14 จาก Validation Policy สู่การกำกับข้อมูลจริง . . . 135
9.15 สรุป . . . 136

10 บทส่งท้าย: Semantic Web และระบบความรู้ในยุค LLM 137
10.1 เส้นทางที่หนังสือเล่มนี้วางไว้ . . . 137
10.2 ทำไม LLM ยังต้องการ Semantic Web . . . 138
10.3 สถาปัตยกรรมระบบแบบ Hybrid LLM-Ontology . . . 139
10.4 จากคำถามภาษาไทยสู่ SPARQL . . . 141
10.5 การประเมินคำตอบที่มีหลักฐาน . . . 142
10.6 บทเรียนที่เชื่อมโยงกันเมื่อมองทั้งเล่ม . . . 143
10.7 ถ้อยคำปิดท้าย . . . 144

อภิธานศัพท์ 146

บรรณานุกรมรวม 149

ดัชนีค้นคำ 153

สารบัญภาพ

1.1 เว็บทั่วไปในฐานะเครือข่ายของเอกสารที่เชื่อมกันด้วย Hyperlink 1
1.2 กราฟเชิงความหมายที่ Resource และความสัมพันธ์มี IRI ชัดเจน 3
1.3 ส่วนประกอบของ RDF Triple ซึ่งมี Subject, Predicate และ Object 4
1.4 การขยาย Prefix และ Local Part กลับเป็น IRI เต็ม 5
1.5 บริบทของ MSSN-Onto: จาก Sensor และ Multimedia Data ไปสู่ Event 8
1.6 บริบทของ Dx-Onto: การเชื่อมเอกสารหลายชนิดเข้ากับความรู้ของโครงการ 9

2.1 RDFS vocabulary สำหรับอธิบาย class, subclass, domain และ range 16
2.2 การรวม RDF Graph จากสองแหล่งด้วย IRI ของโครงการเดียวกัน 24

3.1 OWL reasoning และ SHACL validation มีบทบาทต่างกันแต่ต้องทำงานร่วมกัน 29
3.2 ความต่างระหว่าง **owl:someValuesFrom** และ **owl:allValuesFrom** 50

4.1 SPARQL ทำให้การตอบคำถามของระบบผูกกับ graph pattern และ evidence rows . . 55
4.2 การใช้ตัวแปรร่วมเป็นจุดเชื่อมและสร้าง binding ใน SPARQL 67

5.1 Ontology engineering method ในฐานะ pipeline ที่ตรวจสอบแต่ละขั้นได้ 70

6.1 องค์ประกอบของ SSN/SOSA ที่ใช้เป็นฐานสำหรับอธิบาย Sensor, Observation และ Result 85
6.2 การขยายชนิด Media Sensor และชนิดข้อมูลสื่อใน MSSN-Onto 86
6.3 การออกแบบ Multimedia Data, Media Segment และ Media Descriptor ใน MSSN-Onto 87
6.4 โครงสร้างแกนของ Dx-Onto ที่เชื่อมโครงการกับ Phase, Dimension, Resource และรายละเอียดที่เกี่ยวข้อง 88

7.1 กรอบพิจารณา alignment จาก MSSN-Onto ผ่าน SSN/SOSA ไปยัง DUL และ DOLCE 95

8.1 สถาปัตยกรรมปฏิบัติของ Knowledge Graph Application ตั้งแต่ Ingest ถึง Governance 103
8.2 เส้นทางจาก Extraction Candidate ไปสู่ Knowledge Graph ที่เผยแพร่ 108

9.1 เส้นทาง Provenance ระดับ Claim จากข้อสรุปกลับไปยังรายงานและกิจกรรมที่สร้างข้อมูล 131

10.1 Hybrid LLM-Ontology: LLM เป็น interface และ Knowledge Graph เป็นฐานของหลักฐาน 138
10.2 แนวทาง Hybrid LLM-Ontology ที่แบ่ง Ontology Indexing ออกจาก Semantic Recall 140

สารบัญตาราง

1.1 คำพื้นฐานที่ใช้ในการอ่าน RDF เบื้องต้น 6

2.1 คำศัพท์มาตรฐานของ RDF และ RDFS ที่ใช้บ่อยในบทนี้ 16
2.2 ตัวอย่างข้อมูลโครงการในรูปตารางก่อนแปลงเป็น RDF 18
2.3 ตัวอย่างข้อมูลโครงการที่มีรหัส สถานะ และรายงานอ้างอิง 21

3.1 การแยกรุ่นมาตรฐานออกจากภาษาย่อยและ Profile ของ OWL 30
3.2 คำศัพท์พื้นฐานของ OWL 2 ใน RDF/Turtle 32
3.3 คำศัพท์สำหรับ Class Expression และ Restriction 32
3.4 คำศัพท์สำหรับ Property, Identity และ Ontology Metadata 33
3.5 องค์ประกอบพื้นฐานของ First-Order Logic 35
3.6 ตัวอย่างการแปลงประโยคโดเมนเป็น First-Order Logic 37
3.7 สะพานเชื่อม First-Order Logic, Description Logic และ OWL 38
3.8 บทบาทที่ต่างกันของ RDFS, OWL และ SHACL 40

5.1 การแตก Competency Question หนึ่งข้อเป็นข้อกำหนดที่ตรวจได้ 73

6.1 การเปรียบเทียบการตัดสินใจออกแบบ MSSN-Onto และ Dx-Onto 91

7.1 แนวทางพิจารณาหมวดสำคัญของ DUL ก่อนทำ Alignment 97
7.2 การตัดสินใจ Alignment และ non-Alignment สำหรับ MSSN-Onto 99

8.1 Artifact สำคัญในแต่ละชั้นของ Knowledge Graph Application 104
8.2 Source Record ที่ใช้ตลอด Walkthrough 113
8.3 สถานะของข้อมูลในแต่ละ Checkpoint 114
8.4 การเลือกแนวทางจัดการข้อมูลตามลักษณะของปัญหา 118
8.5 การเลือก Tool ตาม Layer และ Artifact ที่ต้องการตรวจ 120

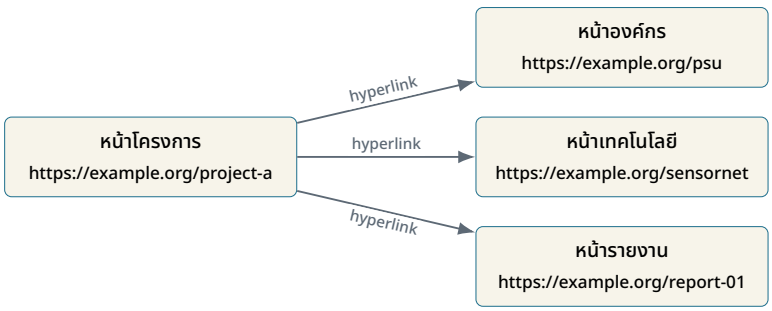
1

จากเว็บเอกสารสู่ Semantic Web

เว็บไซต์ทั่วไปนำเสนอข้อมูลผ่านข้อความ ภาพ ตาราง และลิงก์ไปยังหน้าอื่น ทำให้ผู้อ่านเข้าถึงเอกสารที่เกี่ยวข้องได้สะดวก แต่ลิงก์เพียงอย่างเดียวไม่ได้บอกความหมายของความสัมพันธ์ เช่น ลิงก์จากหน้าโครงการไปยังหน้ามหาวิทยาลัยอาจหมายถึงหน่วยงานผู้รับผิดชอบหรือผู้สนับสนุนทุน ผู้อ่านต้องอาศัยข้อความประกอบจึงจะแยกบทบาทเหล่านี้ได้ บทนี้เริ่มจากข้อจำกัดดังกล่าว แล้วอธิบายว่า Semantic Web ช่วยระบุสิ่งที่ข้อมูลกล่าวถึงและความสัมพันธ์ระหว่างสิ่งเหล่านั้นด้วย Resource, IRI และ Triple ได้อย่างไร

1.1 เว็บของเอกสารและข้อจำกัดของ Hyperlink

การทำความเข้าใจ Semantic Web ควรเริ่มจากสิ่งที่เว็บแบบเดิมทำได้ดีเสียก่อน จากนั้นจึงพิจารณาว่า Hyperlink ยังขาดข้อมูลชนิดใดเมื่อต้องนำเอกสารจากหลายแหล่งมาสืบค้นและวิเคราะห์รวมกัน หน้าเว็บแต่ละหน้ามีที่อยู่ซึ่งเรียกว่า URL หรือ Uniform Resource Locator ในตัวอย่างสมมติของหนังสือเล่มนี้ ใช้มหาวิทยาลัยสงขลานครินทร์ (Prince of Songkla University: PSU) เป็นหน่วยงานรับผิดชอบโครงการ และใช้ **https://www.psu.ac.th/research/project-a** เป็นที่อยู่สมมติของหน้าโครงการ ภายในหน้านั้นอาจมี hyperlink เชื่อมไปยังหน้าเว็บไซต์ของมหาวิทยาลัย หน้าของหน่วยงานผู้รับผิดชอบ หรือหน้าที่อธิบายเทคโนโลยีที่ใช้ในโครงการ เมื่อผู้ใช้กดลิงก์ เว็บเบราว์เซอร์ก็เปิดเอกสารปลายทางให้ ลักษณะดังกล่าวทำให้เว็บเป็นเครือข่ายขนาดใหญ่ของเอกสารที่เชื่อมโยงกัน จุดแข็งของเว็บอยู่ที่นักออกแบบไม่จำเป็นต้องนำเอกสารทุกฉบับมาไว้ในระบบเดียวกัน เพียงรู้ URL ก็สามารถสร้างลิงก์ไปยังข้อมูลที่อยู่คนละเว็บไซต์ได้ แนวคิดนี้ทำให้ข้อมูลจากมหาวิทยาลัย หน่วยงานรัฐ บริษัท และบุคคลทั่วไปเชื่อมถึงกันได้โดยไม่ต้องใช้ฐานข้อมูลเดียวกัน ลักษณะของเว็บในมุมของเอกสารแสดงในภาพที่ 1.1 หน้าโครงการเชื่อมไปยังหน้าองค์กร หน้าเทคโนโลยี และหน้ารายงาน แต่เส้นเชื่อมทุกเส้นเป็นเพียง hyperlink ที่ไม่ได้ระบุชนิดของความสัมพันธ์



ภาพที่ 1.1: เว็บทั่วไปในฐานะเครือข่ายของเอกสารที่เชื่อมกันด้วย Hyperlink

ภาพที่ 1.1 แสดงหน้าเว็บสมมติของโครงการ “Project A” ที่เชื่อมไปยังหน้าองค์กร เชื่อมไปยังหน้าเทคโนโลยี Sensor Network และเชื่อมไปยังรายงานสรุปโครงการ อย่างไรก็ตาม hyperlink แต่ละเส้นไม่ได้ระบุความหมายของความสัมพันธ์ไว้โดยตรง ทำให้การสืบค้นข้อมูลว่าผู้ใดเป็นเจ้าของโครงการ เป็นหน่วยงานผู้รับผิดชอบ เป็นผู้สนับสนุนทุน หรือเป็นเพียงเว็บไซต์ที่มีข้อมูลเกี่ยวข้องนั้นจำเป็นต้องให้มนุษย์อ่านประโยคครอบลิงก์แล้วจึงทำความเข้าใจได้ แต่โปรแกรมที่ต้องรวมข้อมูลจากหลายเว็บไซต์จะต้องวิเคราะห์ข้อความเพิ่มเติมและอาจตีความต่างกัน ปัญหานี้ชัดขึ้นเมื่อข้อมูลมีจำนวนมาก สมมุติว่าต้องการตอบคำถามว่า “มีโครงการใดบ้างที่มหาวิทยาลัยสงขลานครินทร์รับผิดชอบและใช้เทคโนโลยีด้าน Sensor” หากข้อมูลอยู่ในเว็บไซต์หลายแห่ง โปรแกรมต้องรู้ก่อนว่าหน้าใดกล่าวถึงโครงการ หน้าใดกล่าวถึงองค์กร และลิงก์ใดหมายถึงความสัมพันธ์ว่า “รับผิดชอบ” หรือ “ใช้เทคโนโลยี” การค้นเพียงคำที่ปรากฏในเอกสารอาจคืนหน้าที่มีคำใกล้เคียง แต่ไม่รับรองว่าความสัมพันธ์ในคำถามเป็นจริง ข้อจำกัดดังกล่าวเป็นหนึ่งในแรงผลักดันของ Semantic Web ซึ่งเสนอให้เว็บไม่เพียงเชื่อมเอกสาร แต่สามารถอธิบายสิ่งที่เอกสารกล่าวถึงและความสัมพันธ์ระหว่างสิ่งเหล่านั้นในรูปที่โปรแกรมนำไปใช้ต่อได้ แนวคิดนี้ไม่ต้องการยกเลิกเว็บเดิม หน้าเว็บและ hyperlink ยังทำงานตามปกติ เพียงเพิ่มชั้นข้อมูลที่บอกความหมายอย่างชัดเจนขึ้น (Berners-Lee et al., 2001; Bizer et al., 2009)

1.2 จากลิงก์ระหว่างเอกสารสู่กราฟของข้อมูล

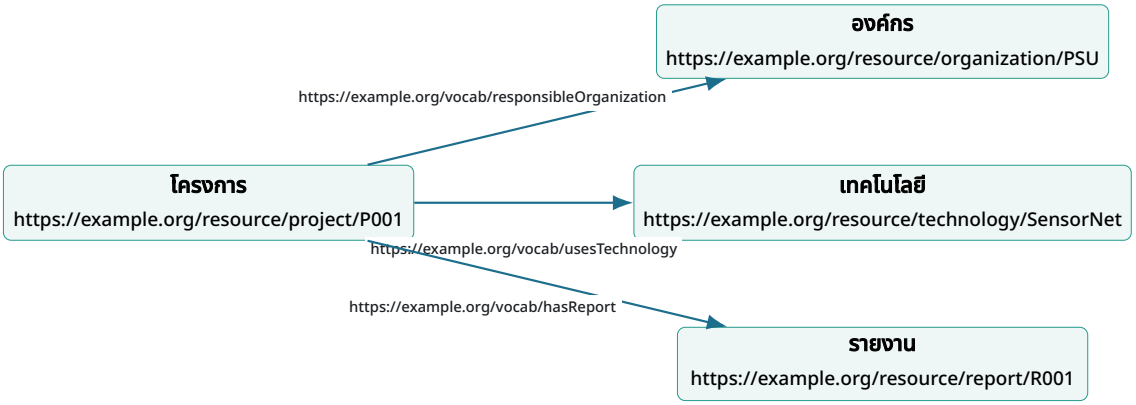
ก่อนกล่าวถึงมาตรฐานของ Semantic Web ควรทำความเข้าใจคำว่า *กราฟ* ในความหมายของโครงสร้างข้อมูลเสียก่อน กราฟประกอบด้วยจุดที่เรียกว่า Node และเส้นเชื่อมที่เรียกว่า Edge ในบริบทของข้อมูล Node ใช้แทนสิ่งที่กำลังกล่าวถึง เช่นโครงการ องค์กร เทคโนโลยี หรือรายงาน ส่วน Edge ใช้แทนความสัมพันธ์ เช่น “รับผิดชอบโดย” “ใช้เทคโนโลยี” หรือ “มีหลักฐานจากรายงาน”

ความต่างจากภาพเว็บทั่วไปคือเส้นในกราฟเชิงความหมายมีชื่อกำกับ ไม่ได้บอกเพียงว่าจุดสองจุดเชื่อมกัน แต่บอกด้วยว่าเชื่อมกันในลักษณะใด ตัวอย่างเช่น โครงการ A เชื่อมไปยัง PSU ด้วยความสัมพันธ์ว่า “มีหน่วยงานรับผิดชอบ” และเชื่อมไปยัง SensorNet ด้วยความสัมพันธ์ว่า “ใช้เทคโนโลยี” เมื่อโปรแกรมพบเส้นทั้งสอง ก็สามารถแยกบทบาทของ PSU ออกจากบทบาทของ SensorNet ได้ แม้ทั้งสองจะเป็นปลายทางของลิงก์จากโครงการเดียวกัน

ในหนังสือเล่มนี้ คำว่า *สิ่งที่ระบบกล่าวถึง* จะใช้ในความหมายกว้าง สิ่งนั้นอาจมีตัวตนทางกายภาพ เช่น อุปกรณ์ Sensor อาจเป็นองค์กร บุคคล หรือสถานที่ อาจเป็นสิ่งที่เกิดขึ้น เช่นเหตุการณ์หรือการสังเกต และอาจเป็นสิ่งที่สร้างขึ้นในระบบสารสนเทศ เช่นเอกสาร รายงาน หรือโครงการ แนวคิดของ Semantic Web เรียกสิ่งที่นำมาอธิบายในกราฟว่า *Resource* หรือทรัพยากรข้อมูล

เพื่อให้ Resource เดียวกันถูกอ้างถึงซ้ำได้ ระบบต้องมีตัวระบุที่ไม่กำกวม Semantic Web ใช้ตัวระบุที่เรียกว่า IRI หรือ Internationalized Resource Identifier ตัวอย่างเช่น **https://example.org/resource/project/P001** **https://example.org/resource/organization/PSU** หรือ **https://example.org/resource/technology/SensorNet** เป็นต้น IRI มีลักษณะคล้าย URL และมักเขียนโดยใช้รูปแบบ HTTPS เช่นเดียวกัน แต่หน้าที่หลักใน RDF คือใช้เป็น *ชื่อของ Resource* ไม่ใช่เพียงบอกตำแหน่งไฟล์หรือหน้าเว็บ IRI ที่ออกแบบดีควรคงที่และไม่ถูกนำกลับไปใช้แทนสิ่งอื่น ในงานจริงมักออกแบบให้เปิดผ่านเว็บได้ด้วย เพื่อให้ผู้ใช้หรือโปรแกรมค้นคำอธิบายเพิ่มเติมจาก IRI นั้นได้ (World Wide Web Consortium, 2014a)

การเปลี่ยนจากลิงก์ระหว่างเอกสารไปเป็นกราฟของสิ่งที่เอกสารกล่าวถึงแสดงในภาพที่ 1.2 โดยแต่ละ Node มี IRI เต็ม และแต่ละเส้นมี IRI ของความสัมพันธ์กำกับ จึงสามารถอ่านความหมายของเส้นแต่ละเส้นได้โดยไม่ต้องเดาจากตำแหน่งของลิงก์



ภาพที่ 1.2: กราฟเชิงความหมายที่ Resource และความสัมพันธ์มี IRI ชัดเจน

ภาพที่ 1.2 แสดงให้เห็นว่า IRI ทำให้ Resource และความสัมพันธ์ถูกอ้างถึงได้อย่างแน่นอน แม้การมี IRI จะไม่ได้ทำให้โปรแกรมเข้าใจทุกอย่างโดยอัตโนมัติ เพราะผู้เขียนข้อมูลยังต้องเลือกความสัมพันธ์ให้ตรงกับความหมายและชุมชนผู้ใช้ยังต้องมีคำศัพท์ที่เข้าใจร่วมกัน แต่ IRI เป็นจุดเริ่มต้นของการรวมข้อมูลจากหลายแหล่ง หากสองระบบใช้ IRI เดียวกันเพื่อกล่าวถึง PSU โปรแกรมก็มีพื้นฐานสำหรับรู้ว่าข้อมูลทั้งสองชุดกล่าวถึงองค์กรเดียวกัน ลักษณะการนำเสนอข้อมูลที่มีตัวระบุและการเชื่อมโยงเชิงความหมายชัดเจนเช่นนี้เป็นพื้นฐานของ Semantic Web ซึ่งเป็นเว็บของข้อมูลที่เชื่อมต่อกันในระดับความหมาย เทคโนโลยี Semantic Web ได้รับการวางเป็นมาตรฐานโดย W3C และมีกรอบสำหรับนำเสนอข้อมูลในรูปดังกล่าวชื่อว่า Resource Description Framework หรือ RDF

1.3 Resource Description Framework (RDF)

Resource Description Framework หรือ RDF เป็นมาตรฐานของ W3C สำหรับแทนข้อมูลในรูปกราฟ หลักพื้นฐานของ RDF คือการแบ่งข้อความหนึ่งเรื่องออกเป็นประโยคย่อยที่มีสามส่วน เรียกว่า *Triple* แต่ละ Triple ประกอบด้วย Subject, Predicate และ Object (World Wide Web Consortium, 2014a,b)

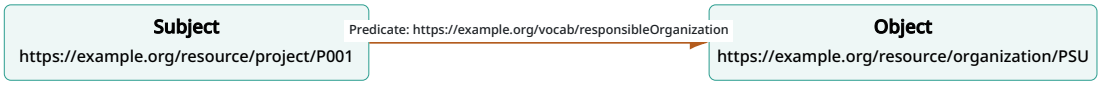
- *Subject* คือ Resource ที่ประโยคกำลังกล่าวถึง
- *Predicate* คือชื่อของความสัมพันธ์หรือคุณลักษณะที่ใช้กล่าวถึง Subject
- *Object* คือ Resource หรือค่าข้อมูลที่อยู่ปลายทางของความสัมพันธ์

เมื่อเขียนให้อยู่ในรูปย่อ มักใช้ตัวอักษร s, p และ o แทน Subject, Predicate และ Object แล้วเขียน Triple หนึ่งประโยคเป็น $\langle s, p, o \rangle$ ตัวอักษรทั้งสามเป็นเพียงสัญลักษณ์แทนตำแหน่ง ไม่ใช่ค่าที่ต้องพิมพ์ลงในข้อมูลจริง โดย s คือสิ่งที่ประโยคกำลังกล่าวถึง p คือความสัมพันธ์ที่เชื่อมออกจากสิ่งนั้น และ o คือสิ่งหรือค่าที่อยู่ปลายทาง ดังนั้นคำย่อ S/P/O ที่จะพบต่อจากนี้จึงหมายถึงโครงสร้างสามส่วนเดียวกันนี้ ส่วนข้อกำหนดอย่างเป็นทางการว่าแต่ละตำแหน่งรับ IRI, Blank Node หรือ Literal ได้อย่างไร จะอธิบายในบทที่ 2

ตัวอย่างเช่น ประโยคภาษาไทยว่า “โครงการ P001 มีหน่วยงานรับผิดชอบคือ PSU” สามารถแยกได้ว่าโครงการ P001 เป็น Subject คำว่า “มีหน่วยงานรับผิดชอบ” เป็น Predicate และ PSU เป็น Object หากใช้ IRI เต็ม จะเขียน Triple ได้ดังนี้

```
<https://example.org/resource/project/P001>
 <https://example.org/vocab/responsibleOrganization>
 <https://example.org/resource/organization/PSU> .
```

เครื่องหมายวงเล็บแหลม <...> ใช้ครอบ IRI ส่วนเครื่องหมายจุดใช้ปิดท้าย Triple หนึ่งประโยค เมื่ออ่านจากบนลงล่างจึงได้ Subject หนึ่งรายการ Predicate หนึ่งรายการ และ Object หนึ่งรายการตามลำดับโครงสร้างของ Triple เดียวกันในรูปกราฟแสดงในภาพที่ 1.3 โดย Subject และ Object เป็น Node ส่วน Predicate เป็นชื่อบนเส้นที่เชื่อม Node ทั้งสอง จุดสำคัญคือ Predicate ไม่ใช่ข้อความอธิบายที่วางไว้เพื่อให้ภาพสวยงาม แต่เป็น IRI ที่โปรแกรมใช้อ้างถึงความสัมพันธ์นั้นจริง



ภาพที่ 1.3: ส่วนประกอบของ RDF Triple ซึ่งมี Subject, Predicate และ Object

ภาพที่ 1.3 แสดงให้เห็นว่า Predicate ทำหน้าที่เป็นเส้นที่มีทิศทางจาก Subject ไปยัง Object อย่างไรก็ตาม Object ใน RDF ไม่จำเป็นต้องเป็น Resource ที่มี IRI เสมอไป แต่สามารถเป็นค่าข้อมูลโดยตรงได้ เช่น ชื่อ จำนวน วันที่ หรือข้อความ ค่าประเภทนี้เรียกว่า *Literal* ตัวอย่างต่อไปนี้บอกชื่อภาษาไทยของโครงการโดยเขียน Object ในรูป Literal

```
<https://example.org/resource/project/P001>
 <http://www.w3.org/2000/01/rdf-schema#label>
 "โครงการระบบเฝ้าระวังอัจฉริยะ"@th .
```

ข้อความในเครื่องหมายอัญประกาศคือ Literal ส่วน **@th** เป็น language tag ที่ระบุว่าข้อความนี้เป็นภาษาไทย การระบุภาษาแยกจากข้อความช่วยให้ Resource เดียวมีชื่อได้หลายภาษา เช่นเพิ่มชื่อภาษาอังกฤษโดยใช้ **@en** โดยยังคงเป็นโครงการเดียวกัน

ในระดับภาพรวมของบทนี้ เพียงเห็นว่า Triple หลายประโยคสามารถใช้ IRI เดียวกันร่วมกันจนเชื่อมเป็นกราฟก็เพียงพอแล้ว ส่วนแบบจำลอง RDF Graph ในเชิงคณิตศาสตร์ รวมทั้งข้อกำหนดว่า IRI, Blank Node และ Literal อยู่ในตำแหน่งใดของ Triple ได้บ้าง จะอธิบายอย่างเป็นระบบในบทที่ 2 ก่อนถึงรายละเอียดดังกล่าว ผู้อ่านควรอ่านและเขียน IRI ในตัวอย่างให้ได้เสียก่อน เพราะการเขียน IRI เต็มทุกครั้งแม้ชัดเจน แต่จะทำให้ข้อมูลยาวและมองโครงสร้างของ Triple ได้ยาก

1.4 Namespace, Prefix และการเขียน Turtle เบื้องต้น

การเขียน IRI เต็มทุกครั้งช่วยให้เห็นความหมายชัด แต่ทำให้ข้อมูลยาวและอ่านยาก W3C จึงกำหนดรูปแบบการเขียน RDF ที่เรียกว่า *Turtle* เพื่อให้เขียน Triple ได้สั้นและกระชับขึ้น Turtle อนุญาตให้ประกาศ *Prefix* เพื่อใช้ชื่อย่อแทนส่วนต้นของ IRI ที่ปรากฏซ้ำ ส่วนต้นที่ใช้ร่วมกันนี้มักเรียกว่า *Namespace* (World Wide Web Consortium, 2014c)

ตัวอย่างในบทนี้แยก Namespace ตามชนิดของ Resource เพื่อให้ชื่อย่ออ่านง่ายและเขียนได้ถูกต้องตามไวยากรณ์ Turtle โดยใช้ **proj:** สำหรับโครงการ **org:** สำหรับองค์กร **tech:** สำหรับเทคโนโลยี และ **voc:** สำหรับคำศัพท์ที่ใช้เป็น Class หรือ Predicate การประกาศ Prefix เขียนได้ดังนี้

```
@prefix proj: <https://example.org/resource/project/> .
@prefix org: <https://example.org/resource/organization/> .
@prefix tech: <https://example.org/resource/technology/> .
@prefix voc: <https://example.org/vocab/> .
```

หลังจากประกาศแล้ว โปรแกรมจะขยายชื่อย่อกลับเป็น IRI เต็ม ตัวอย่างการขยายชื่อย่อสองรายการแสดงได้ดังนี้

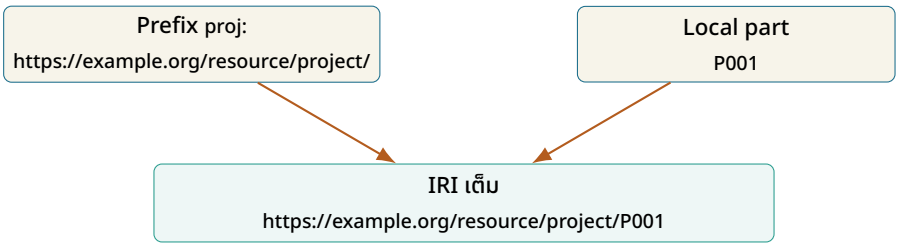
```
proj:P001
 -> <https://example.org/resource/project/P001>
voc:responsibleOrganization
 -> <https://example.org/vocab/responsibleOrganization>
```

เมื่อใช้ชื่อย่อเหล่านี้ Triple เดิมจึงเขียนให้สั้นลงได้ว่า

```
proj:P001 voc:responsibleOrganization org:PSU .
```

Prefix เป็นเพียงรูปย่อในเอกสาร Turtle ไม่ใช่ส่วนหนึ่งของตัวตนที่แท้จริงของ Resource โปรแกรมจะขยายชื่อย่อกลับเป็น IRI เต็มก่อนประมวลผล ดังนั้นสองไฟล์อาจใช้ Prefix ต่างกัน เช่นไฟล์หนึ่งใช้ **ex:** และอีกไฟล์ใช้ **data:** แต่ยังกล่าวถึง Resource เดียวกันได้ หากทั้งสอง Prefix ขยายกลับไปยัง Namespace เดียวกัน

ความสัมพันธ์ระหว่าง Prefix, Namespace และ IRI เต็มแสดงในภาพที่ 1.4 เครื่องหมาย colon หลังคำว่า **proj** ใช้แยกชื่อ Prefix ออกจากส่วนท้ายของ IRI ซึ่งมักเรียกว่า local part



ภาพที่ 1.4: การขยาย Prefix และ Local Part กลับเป็น IRI เต็ม

ภาพที่ 1.4 แสดงว่าชื่อย่อหนึ่งชื่อประกอบด้วย Prefix ที่ผูกกับ Namespace และ local part ซึ่งเมื่อนำมารวมกันจะได้ IRI เต็ม จากหลักนี้ ตัวอย่าง Turtle ที่สมบูรณ์ขึ้นสามารถเขียนข้อมูลของโครงการ องค์กร และเทคโนโลยีได้ดังนี้

```
@prefix proj: <https://example.org/resource/project/> .
@prefix org: <https://example.org/resource/organization/> .
@prefix tech: <https://example.org/resource/technology/> .
@prefix voc: <https://example.org/vocab/> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .

proj:P001
 rdfs:label "โครงการระบบเฝ้าระวังอัจฉริยะ"@th ;
 voc:responsibleOrganization org:PSU ;
 voc:usesTechnology tech:SensorNet .
```

เครื่องหมายอัฒภาคหมายความว่า Triple ถัดไปยังใช้ Subject เดิม คือ **proj:P001** แต่เปลี่ยน Predicate และ Object หากคลี่รูปย่อออก ข้อมูลส่วนนี้ประกอบด้วยสาม Triple ได้แก่โครงการมีชื่อภาษาไทย โครงการมีมหาวิทยาลัยสงขลานครินทร์เป็นหน่วยงานรับผิดชอบ และโครงการใช้ SensorNet เป็นเทคโนโลยี แม้เขียนหลายบรรทัดเพื่อให้อ่านง่าย ความหมายของกราฟยังคงเป็นประโยคสามส่วนเช่นเดิม คำว่า **rdfs:label** เป็นคำศัพท์มาตรฐานสำหรับให้ชื่อที่มนุษย์อ่านได้แก่ Resource Prefix **rdfs:** ขยายเป็น Namespace ของ RDF Schema ส่วนรายละเอียดว่า RDF Schema คืออะไร และคำศัพท์มาตรฐานอย่าง **rdf:type**, **rdfs:Class**, **rdfs:subClassOf**, **rdfs:domain** และ **rdfs:range** ใช้อย่างไร จะอธิบายในบทที่ 2 หลังจากผู้อ่านมีพื้นฐานเรื่อง IRI และ Prefix แล้ว

ก่อนนำแนวคิดเหล่านี้ไปใช้กับกรณีศึกษา ตารางที่ 1.1 รวบรวมคำที่ปรากฏแล้วในบทนี้ตามลำดับการใช้งาน ตารางนี้ไม่ได้ใช้แทนคำอธิบายในเนื้อหา แต่ทำหน้าที่เป็นจุดทบทวนว่าแต่ละคำอยู่คนละระดับกัน เช่น URL กล่าวถึงที่อยู่บนเว็บ IRI กล่าวถึงตัวระบุของ Resource และ Prefix เป็นเพียงรูปย่อสำหรับเขียน IRI

ตารางที่ 1.1: คำพื้นฐานที่ใช้ในการอ่าน RDF เบื้องต้น

คำศัพท์	ความหมายในบทนี้	ตัวอย่าง
URL	ที่อยู่สำหรับเข้าถึงหน้าเว็บหรือข้อมูลผ่านเครือข่าย	**https://example.org/project-a**
Resource	สิ่งใด ๆ ที่ระบบต้องการกล่าวถึง เช่น โครงการ องค์กร อุปกรณ์ หรือเอกสาร	โครงการ P001 หรือองค์กร PSU
IRI	ตัวระบุของ Resource ที่ใช้อ้างถึงซ้ำโดยไม่พึ่งชื่อที่มนุษย์ตั้ง	**https://example.org/resource/project/P001**
Triple	ประโยคข้อมูลหนึ่งประโยคที่ประกอบด้วย Subject, Predicate และ Object	โครงการ P001 มีองค์กรรับผิดชอบ PSU
Subject	Resource ที่ Triple กำลังกล่าวถึง	**proj:P001**
Predicate	IRI ของความสัมพันธ์หรือคุณลักษณะที่ใช้เชื่อม Subject ไปยัง Object	**voc:responsibleOrganization**
Object	ปลายทางของ Predicate ซึ่งเป็น Resource หรือ Literal ก็ได้	**org:PSU**
Literal	ค่าข้อมูลโดยตรง เช่นข้อความ ตัวเลข หรือวันที่	**"โครงการระบบเฝ้าระวังอัจฉริยะ"@th**
Namespace	ส่วนต้นของ IRI ที่คำศัพท์หรือ Resource กลุ่มหนึ่งใช้ร่วมกัน	**https://example.org/resource/**
Prefix	ชื่อย่อที่ผูกกับ Namespace เพื่อให้เขียน Turtle ได้กระชับ	**proj:**

วิธีใช้ตารางนี้เมื่อพบตัวอย่างใหม่คือเริ่มจากหา Subject ก่อน แล้วอ่าน Predicate เป็นคำเชื่อม จากนั้นจึงดูว่า Object เป็น Resource ที่มี IRI หรือเป็น Literal หากพบชื่อย่อที่มี colon เช่น **proj:P001** ให้ย้อนดูคำประกาศ Prefix เพื่อขยายกลับเป็น IRI เต็ม วิธีอ่านทีละขั้นนี้ช่วยแยกโครงสร้างของ RDF ออกจากความหมายเฉพาะของโดเมน และใช้ได้กับทั้งข้อมูลโครงการ ข้อมูล Sensor หรือข้อมูลประเภทอื่น

เมื่ออ่านโครงสร้างของ Triple และขยาย Prefix กลับเป็น IRI ได้แล้ว คำถามถัดไปไม่ใช่ว่าเขียนข้อมูลให้สั้นลงอย่างไร แต่เป็นคำถามว่าระบบต่าง ๆ จะเข้าใจ Class และ Property เหล่านี้ตรงกันได้อย่างไร ประเด็นดังกล่าวนำไปสู่บทบาทของ Ontology

1.5 จาก RDF Graph สู่ Ontology

RDF กำหนดรูปของประโยคข้อมูลว่าต้องมี Subject, Predicate และ Object แต่การเขียน Triple ได้ยังไม่เท่ากับการที่ระบบต่าง ๆ เข้าใจคำศัพท์ตรงกัน สมมุติว่าระบบหนึ่งใช้คำว่า **voc:Project** แทนโครงการ ขณะที่อีกระบบใช้ **voc:Initiative** โปรแกรมยังไม่สามารถสรุปเองได้ว่าสองคำนี้หมายถึงแนวคิดเดียวกัน หรือคำหนึ่งมีความหมายกว้างกว่าอีกคำหนึ่ง ปัญหาถัดมาจึงเป็นการกำหนดว่าในขอบเขตงานหนึ่งมีแนวคิดอะไรบ้าง แต่ละแนวคิดสัมพันธ์กันอย่างไร และข้อกำหนดใดต้องใช้ร่วมกัน

ในวิทยาการสารสนเทศ คำว่า *Ontology* หมายถึงข้อกำหนดอย่างชัดแจ้งของแนวคิดที่ใช้ร่วมกันในขอบเขตหนึ่ง โดยระบุทั้งกลุ่มของสิ่งที่สนใจ ความสัมพันธ์ระหว่างสิ่งเหล่านั้น และในบางกรณีรวมถึงข้อกำหนดเชิงตรรกะที่โปรแกรมนำไปใช้อนุมานได้ (Gruber, 1993; Noy and McGuinness, 2001) ตัวอย่างเช่น Ontology สำหรับข้อมูลโครงการอาจกำหนดแนวคิด Project, Organization และ Technology พร้อมความสัมพันธ์ responsibleOrganization และ usesTechnology การกำหนดเช่นนี้ไม่ได้สร้างข้อมูลของโครงการ P001 ขึ้นมา แต่สร้างคำศัพท์และความหมายที่ใช้เขียนข้อมูลของ P001 และโครงการอื่น ๆ ให้สอดคล้องกัน

จึงควรแยกสองสิ่งนี้ออกจากกันตั้งแต่ต้น RDF Graph คือชุดของ Triple ซึ่งอาจใช้บันทึกข้อเท็จจริงเฉพาะรายการ ส่วน Ontology คือแบบจำลองของคำศัพท์ ความสัมพันธ์ และข้อกำหนดที่ Graph เหล่านั้นอ้างอิง Ontology สามารถเขียนด้วยมาตรฐานในตระกูล Semantic Web โดยเริ่มจาก RDF Schema สำหรับ Class, Property และลำดับชั้นพื้นฐาน แล้วเพิ่มความหมายเชิงตรรกะด้วย OWL เมื่อจำเป็น รายละเอียดเหล่านี้จะค่อย ๆ ปรากฏในบทถัดไป เพราะผู้อ่านต้องเข้าใจ RDF Graph ก่อนจึงจะเห็นว่า Ontology เพิ่มความหมายส่วนใดให้แก่กราฟ

เพื่อให้เห็นว่า Ontology ใช้แก้ปัญหาข้อมูลจริงอย่างไร หนังสือเล่มนี้จะใช้สองบริบทประกอบในบทหลัง ได้แก่ Multimedia Sensor Network และการจัดการความรู้ของโครงการเปลี่ยนผ่านสู่ดิจิทัล ทั้งสองบริบทไม่ได้ใช้แทนนิยามของ RDF หรือ Ontology และผู้อ่านไม่จำเป็นต้องรู้รายละเอียดของแบบจำลองตั้งแต่บทนี้ จุดประสงค์มีเพียงให้เห็นล่วงหน้าว่าเมื่อข้อมูลมีอุปกรณ์ เหตุการณ์ โครงการ องค์กร และเอกสารจำนวนมาก ปัญหาเรื่องคำศัพท์และความสัมพันธ์ร่วมเกิดขึ้นอย่างไร

1.6 บริบทที่หนึ่ง: Multimedia Sensor Network

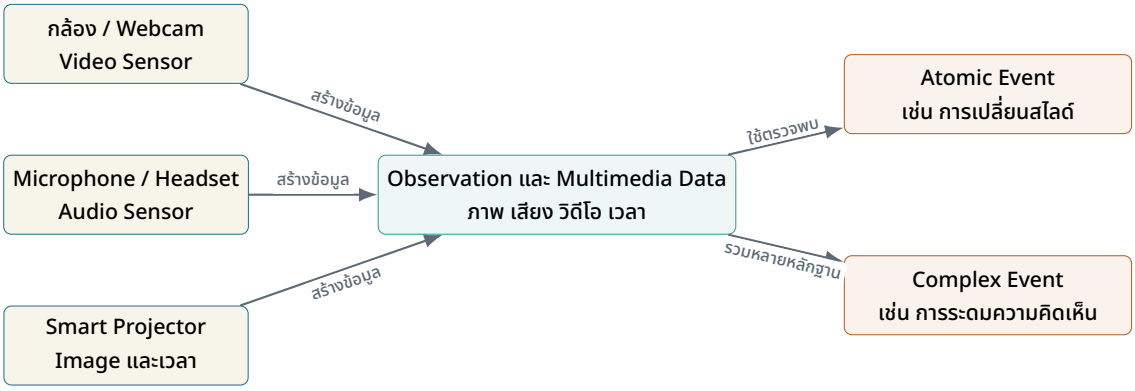
หลักของ RDF และ Ontology ที่กล่าวมาก่อนหน้านี้ยังเป็นแนวคิดค่อนข้างกว้าง กรณีของ Multimedia Sensor Network ช่วยให้เห็นปัญหาเป็นรูปธรรมขึ้น เพราะระบบต้องแยกอุปกรณ์ ข้อมูลที่อุปกรณ์สร้าง และ Event ที่ตีความจากข้อมูลออกจากกันอย่างชัดเจน

ห้องประชุมอัจฉริยะอาจติดตั้งกล้องเว็บแคม ไมโครโฟน เครื่องฉายภาพ และกระดานอัจฉริยะ กล้องบันทึกวิดีโอ ไมโครโฟนบันทึกเสียง ส่วนระบบที่ควบคุมการฉายภาพอาจบันทึกสไลด์และเวลาที่เปลี่ยนสไลด์

ขณะที่กระดานอัจฉริยะอาจบันทึกลายเส้นที่เขียน ข้อมูลจากอุปกรณ์เหล่านี้เป็นหลักฐานให้ระบบนำไปพิจารณาเหตุการณ์ เช่น การตอบว่า “มีการเปลี่ยนสไลด์เมื่อใด” อาจใช้บันทึกจากระบบฉายภาพ แต่การพิจารณาว่า “ผู้เข้าร่วมกำลังระดมความคิดเห็นหรือไม่” อาจต้องใช้เสียงจากหลายคน ช่วงเวลา และเนื้อหาของสไลด์ประกอบกัน ระบบจึงต้องแยกให้ชัดว่าอุปกรณ์ใดสร้างข้อมูลใด ข้อมูลเกิดขึ้นเมื่อใด และนำข้อมูลใดมาใช้ตีความเหตุการณ์ หากแต่ละระบบเรียกอุปกรณ์ ข้อมูล และเหตุการณ์ต่างกัน การรวมข้อมูลหรือนำนิยามเหตุการณ์ไปใช้กับอีกระบบจะทำได้ยาก

จากปัญหานี้ MSSN-Onto หรือ Multimedia Semantic Sensor Network Ontology เสนอแบบจำลองคำศัพท์สำหรับเชื่อม Sensor, Observation, ข้อมูล multimedia และเหตุการณ์ที่ตรวจพบ โดยเชื่อมกับมาตรฐานด้าน Sensor และเพิ่มรายละเอียดที่จำเป็นสำหรับภาพ เสียง และวิดีโอ (Angsuchotmetee et al., 2020) ในบทนี้ MSSN-Onto ทำหน้าที่เพียงเป็นบริบทให้เห็นว่าอุปกรณ์ ข้อมูลที่อุปกรณ์สร้าง และเหตุการณ์ที่ระบบตีความไม่ควรถูกใช้ปะปนกัน ส่วนเหตุผลของ Class, Property และการนำกลับมาใช้ มาตรฐานจะอธิบายอย่างเป็นระบบในบทที่ 6

บริบทของอุปกรณ์ ข้อมูลที่อุปกรณ์สร้าง และเหตุการณ์ที่ระบบตรวจพบแสดงในภาพที่ 1.5 ภาพนี้แยก Observation และข้อมูลสื่อออกจากกระบวนการตรวจเหตุการณ์ เพื่อไม่ให้สับสนระหว่าง Sensor ข้อมูลที่ Sensor สร้าง และเหตุการณ์ที่ระบบสรุปจากข้อมูล



ภาพที่ 1.5: บริบทของ MSSN-Onto: จาก Sensor และ Multimedia Data ไปสู่ Event

ภาพที่ 1.5 แสดงลำดับจาก Sensor ไปยัง Observation และ Multimedia Data ก่อนใช้ข้อมูลเหล่านั้นตรวจพบ Atomic Event หรือรวมหลักฐานเพื่อระบุ Complex Event จุดประสงค์ในที่นี้คือแนะนำปัญหาของโดเมนเท่านั้น ผู้อ่านยังไม่ต้องจำชื่อ Class หรือ Property ของ MSSN-Onto

กรณีนี้แสดงปัญหาของข้อมูลที่เกิดจาก Sensor และ Event ส่วนบริบทถัดไปเปลี่ยนไปพิจารณาข้อมูลเอกสารของโครงการ เพื่อให้เห็นว่าแนวคิดเดียวกันยังใช้ได้แม้ลักษณะข้อมูลและคำถามของโดเมนแตกต่างกัน

1.7 บริบทที่สอง: ความรู้ของโครงการเปลี่ยนผ่านสู่ดิจิทัล

เมื่อเปลี่ยนจาก Multimedia Sensor Networks มาสู่โครงการเปลี่ยนผ่านสู่ดิจิทัล ปัญหาไม่ได้อยู่ที่กระแสข้อมูลจากอุปกรณ์ แต่อยู่ที่ความรู้ซึ่งกระจายอยู่ในเอกสารและระบบหลายประเภท การพิจารณา

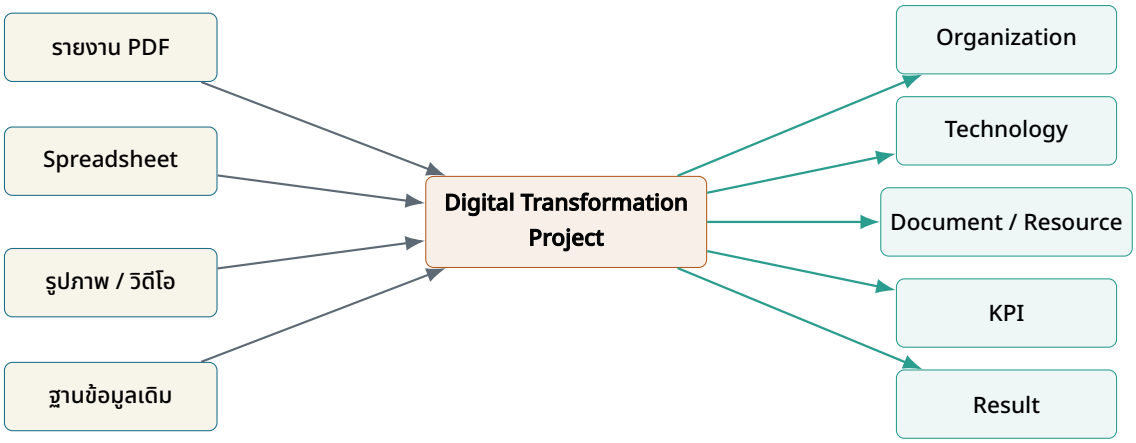
บริบทที่สองจึงช่วยแยกให้เห็นว่าส่วนใดเป็นหลักทั่วไปของ Semantic Web และส่วนใดเป็นการตัดสินใจเฉพาะโดเมน

องค์กรที่ดำเนินโครงการเปลี่ยนผ่านสู่ดิจิทัลมักมีข้อมูลกระจายอยู่ในรายงาน PDF, Spreadsheet, รูปภาพ เอกสารงบประมาณ และระบบของหลายหน่วยงาน เอกสารหนึ่งบอกชื่อและรายละเอียดโครงการ อีกไฟล์บอกองค์กรรับผิดชอบ อีกตารางบอกเทคโนโลยี งบประมาณ และ KPI ขณะที่ผลลัพธ์ของโครงการอาจอยู่ในรายงานสรุป หากคนด้วยชื่อไฟล์หรือคำสำคัญเพียงอย่างเดียว ผู้ใช้ต้องรู้ล่วงหน้าว่าข้อมูลอยู่ที่ใดและสะกดคำอย่างไร

ปัญหาหลักคือการทำให้เอกสารเหล่านี้กล่าวถึงโครงการ องค์กร เทคโนโลยี และผลลัพธ์ชุดเดียวกันอย่างสอดคล้อง รวมทั้งสามารถถามคำถามหลายเงื่อนไข เช่น “โครงการด้านการท่องเที่ยวใดใช้เทคโนโลยี drone และมีผลลัพธ์สำเร็จ” คำถามเช่นนี้ต้องเชื่อมข้อมูลจากหลายส่วน ไม่ใช่เพียงหาเอกสารที่มีคำว่า drone

จากปัญหานี้ Dx-Onto หรือ Digital Transformation Ontology เสนอคำศัพท์ที่ใช้ Project เป็นจุดเชื่อมไปยัง Organization, Technology, Resource document, phase, dimension, KPI และ result การแยกสิ่งเหล่านี้เป็น Resource และความสัมพันธ์ทำให้ระบบค้นจากความหมายของข้อมูลได้ แม้ไฟล์ต้นทางจะมาจากหลายรูปแบบ (Ben-arlee and Angsuchotmetee, 2026) บทนี้ยังไม่ใช้รายละเอียดของ Dx-Onto เพื่ออธิบาย RDF แต่แนะนำปัญหาและกลุ่มแนวคิดไว้ก่อน การตัดสินใจว่าสิ่งใดควรเป็น Class หรือ Property และเหตุใด Project จึงถูกวางเป็นจุดเชื่อม จะอธิบายในบทที่ 6

การเชื่อมเอกสารหลายชนิดเข้ากับความรู้ของโครงการแสดงในภาพที่ 1.6 เอกสารแต่ละแหล่งถูกนำเข้าสู่กระบวนการจัดระบุตัวตนและความสัมพันธ์ ก่อนรวมเป็นกราฟที่มีโครงการเป็นจุดกลาง โดย Project เชื่อมกับองค์กร เทคโนโลยี หลักฐาน และผลลัพธ์ที่เกี่ยวข้อง



ภาพที่ 1.6: บริบทของ Dx-Onto: การเชื่อมเอกสารหลายชนิดเข้ากับความรู้ของโครงการ

ภาพที่ 1.6 แสดงว่า Project เป็นจุดเชื่อมของข้อมูล ไม่ได้หมายความว่าข้อมูลทุกอย่างต้องถูกเก็บไว้ภายใน Node ของโครงการ สองบริบทในบทนี้จึงมีหน้าที่เพียงแสดงว่าโครงสร้าง Triple รองรับโดเมนที่มีคำศัพท์ต่างกันได้ โดยหลักของ Subject, Predicate, Object และ IRI ยังคงเหมือนเดิม

1.8 Semantic Web ช่วยให้ข้อมูลทำงานร่วมกันอย่างไร

สองบริบทก่อนหน้าใช้คำศัพท์ต่างกัน แต่ต่างเริ่มจากปัญหาเดียวกัน คือข้อมูลจากหลายแหล่งกล่าวถึงสิ่งเดียวกันโดยไม่มีตัวระบุและความสัมพันธ์ร่วม หัวข้อสุดท้ายของบทจึงสรุปว่า Semantic Web ช่วยการทำงานร่วมกันของข้อมูลในสามด้านใดบ้าง ก่อนที่บทถัดไปจะลงรายละเอียดของ RDF และ RDFS

1. **การรวมข้อมูลจากหลายแหล่ง** RDF ทำหน้าที่เป็นโครงสร้างพื้นฐานของประโยคข้อมูล เมื่อ Resource จากหลายแหล่งใช้ IRI ที่สอดคล้องกัน Triple จากคนละไฟล์หรือคนละระบบสามารถรวมเป็น Graph เดียวกันได้ โดยไม่ต้องบังคับให้ทุกแหล่งมีโครงสร้างตารางเหมือนกัน
2. **การตั้งคำถามตามความสัมพันธ์** ผู้ใช้สามารถค้นตามความสัมพันธ์ของข้อมูล เช่น ถามว่า “โครงการใดใช้เทคโนโลยีเซนเซอร์ มีองค์กรใดรับผิดชอบ และมีรายงานใดเป็นหลักฐาน” ภาษาสืบค้น SPARQL ซึ่งจะอธิบายในบทที่ 4 ใช้ระบุรูปแบบความสัมพันธ์เหล่านี้เพื่อค้นหาข้อมูลที่ตรงกับคำถาม
3. **การกำหนดคำศัพท์และความหมายร่วม** RDF บอกว่าประโยคข้อมูลเขียนอย่างไร แต่ยังไม่ได้กำหนดว่า Resource ชนิดใดอยู่ใน Class ใด หรือ Property หนึ่งเชื่อมสิ่งประเภทใด RDFS จึงเพิ่มคำศัพท์สำหรับอธิบาย Class, Property, Hierarchy, Domain และ Range ส่วน OWL เพิ่มความหมายเชิงตรรกะที่ละเอียดขึ้น

Semantic Web ไม่รับรองว่าข้อมูลทุก Triple เป็นความจริง การออกแบบ IRI อาจผิด การ Map ข้อมูลอาจผิด หรือแหล่งข้อมูลอาจขัดแย้งกันได้ ระบบที่นำกราฟไปใช้จริงจึงต้องมีการตรวจคุณภาพและบันทึกที่มาของข้อมูลด้วย ประเด็นเรื่อง SHACL, Provenance และ Governance จะปรากฏภายหลัง เมื่อผู้อ่านเข้าใจโครงสร้างและคำศัพท์ของกราฟแล้ว

บทนี้วางภาพรวมจากเว็บเอกสารไปสู่ Graph ของข้อมูล ส่วนบทถัดไปจะย้อนกลับมาพิจารณา RDF และ RDFS อย่างเป็นระบบ เพื่ออธิบายว่า IRI, Literal, Class และ Property ทำงานร่วมกันอย่างไร และการรวม Graph จากหลายแหล่งมีข้อควรระวังเรื่องใดบ้าง

เอกสารอ้างอิง

Angsuchotmetee, C., Chbeir, R., and Cardinale, Y. (2020). MSSN-Onto: An ontology-based approach for flexible event processing in multimedia sensor networks. *Future Generation Computer Systems*, 108:1140–1158.

Ben-arlee, S. and Angsuchotmetee, C. (2026). Dx-Onto: A core ontology for a semantic-based framework for managing digital transformation projects. *Applied System Innovation*, 9(7):146.

Berners-Lee, T., Hendler, J., and Lassila, O. (2001). The semantic web. *Scientific American*, 284(5):34–43.

Bizer, C., Heath, T., and Berners-Lee, T. (2009). Linked data: The story so far. *International Journal on Semantic Web and Information Systems*, 5(3):1–22.

Gruber, T. R. (1993). A translation approach to portable ontology specifications. *Knowledge Acquisition*, 5(2):199–220.

Heath, T. and Bizer, C. (2011). *Linked Data: Evolving the Web into a Global Data Space*. Morgan & Claypool.

Hogan, A., Blomqvist, E., Cochez, M., d'Amato, C., de Melo, G., Gutierrez, C., Kirrane, S., Gayo, J. E. L., Navigli, R., Neumaier, S., Ngomo, A.-C. N., Polleres, A., Rashid, S. M., Rula, A., Schmelzeisen, L., Sequeda, J., Staab, S., and Zimmermann, A. (2022). Knowledge graphs. *ACM Computing Surveys*, 54(4):1–37.

Noy, N. F. and McGuinness, D. L. (2001). Ontology development 101: A guide to creating your first ontology. Technical Report KSL-01-05, Stanford Knowledge Systems Laboratory.

World Wide Web Consortium (2014a). RDF 1.1 concepts and abstract syntax. W3C Recommendation. Accessed July 4, 2026, from **https://www.w3.org/TR/rdf11-concepts/**.

World Wide Web Consortium (2014b). RDF 1.1 semantics. W3C Recommendation. Accessed June 16, 2026, from **https://www.w3.org/TR/rdf11-mt/**.

World Wide Web Consortium (2014c). RDF 1.1 Turtle: Terse RDF triple language. W3C Recommendation. Accessed June 29, 2026, from **https://www.w3.org/TR/turtle/**.

2

RDF และ RDFS: โครงสร้างข้อมูลและคำศัพท์มาตรฐาน

บทที่แล้วเริ่มจากที่มาของ Semantic Web และอธิบายแนวคิดพื้นฐานว่า ข้อมูลสามารถถูกมองเป็นกราฟที่ประกอบด้วยประโยคแบบ Subject–Predicate–Object อย่างไรก็ตาม การเห็นภาพรวมดังกล่าวยังไม่เพียงพอสำหรับการสร้างกราฟขึ้นใช้งานจริง เพราะยังต้องรู้ว่าสิ่งใดมีสิทธิ์อยู่ในแต่ละตำแหน่งของประโยค ตัวระบุของสิ่งต่าง ๆ ควรเขียนอย่างไร และคำศัพท์ที่ปรากฏบน Node กับเส้นเชื่อมมีความหมายต่อระบบอย่างไร บทนี้จึงลงรายละเอียดของ Resource Description Framework (RDF) ก่อน แล้วจึงอธิบาย RDF Schema (RDFS) ในฐานะส่วนขยายที่ทำให้คำศัพท์ในกราฟมีโครงสร้างและอนุมานความหมายพื้นฐานได้

2.1 จากประโยคข้อมูลสู่ RDF Graph

บทแรกใช้ Triple เพื่ออธิบายภาพรวมของ Semantic Web แต่ยังไม่ได้พิจารณาข้อกำหนดของแต่ละตำแหน่งอย่างเป็นระบบ หัวข้อนี้จึงเริ่มจากโครงสร้างของ Triple แล้วแสดงให้เห็นว่า Triple หลายรายการเชื่อมกันเป็น RDF Graph ได้อย่างไร

RDF เป็นมาตรฐานสำหรับแทนข้อมูลเกี่ยวกับสิ่งต่าง ๆ ในรูปกราฟ สิ่งที่ RDF กำหนดไม่ใช่หัวข้อของข้อมูลและไม่ใช่ชุดคำศัพท์สำเร็จรูปสำหรับโดเมนใดโดเมนหนึ่ง แต่เป็นแบบจำลองร่วมสำหรับเขียนข้อความยืนยันหรือ Claim ให้มีโครงสร้างเดียวกัน กล่าวคือ แต่ละข้อความประกอบด้วยสิ่งที่กำลังกล่าวถึง ความสัมพันธ์ที่ต้องการบอก และสิ่งหรือค่าที่อยู่ปลายความสัมพันธ์นั้น องค์ประกอบทั้งสามเรียกว่า Subject, Predicate และ Object ตามลำดับ (World Wide Web Consortium, 2014a,c) ตัวอย่างเช่น ข้อความว่า “โครงการ A ใช้เทคโนโลยี SensorNet” สามารถแยกได้ว่าโครงการ A เป็น Subject คำว่า “ใช้เทคโนโลยี” เป็น Predicate และ SensorNet เป็น Object เมื่อนำข้อความลักษณะนี้หลายข้อความมารวมกัน Subject และ Object ที่อ้างถึงสิ่งเดียวกันจะกลายเป็น Node ร่วม ส่วน Predicate เป็นเส้นเชื่อมระหว่าง Node RDF Graph จึงไม่ได้เกิดจากการวาดแผนภาพก่อน แต่เกิดจากการรวม Triple ที่ใช้ตัวระบุและคำศัพท์อย่างสม่ำเสมอ

ใน RDF สิ่งที่สามารถถูกกล่าวถึงได้เรียกว่า Resource คำนี้มีความหมายกว้าง อาจเป็นบุคคล องค์กร โครงการ Sensor การสังเกต เหตุการณ์ เอกสาร หรือแม้แต่แนวคิดที่ไม่มีรูปร่างก็ได้ หาก Resource ต้องถูกอ้างถึงอย่างชัดเจนและใช้ซ้ำในกราฟ จะกำหนดชื่อด้วย Internationalized Resource Identifier หรือ IRI ซึ่งเป็นตัวระบุที่มีรูปคล้าย URL แต่หน้าที่หลักคือระบุว่าสิ่งใดกำลังถูกกล่าวถึง ไม่ได้หมายความว่า IRI ทุกค่าต้องเปิดเป็นหน้าเว็บได้เสมอ ส่วนค่าบรรยาย เช่น ชื่อภาษาไทย จำนวน วันที่ หรือข้อความ เรียกว่า Literal และ Node ที่ไม่ต้องการกำหนด IRI โดยตรงเรียกว่า Blank Node ถ้าให้ I เป็นเซตของ IRI, B เป็น

เซตของ Blank Node และ L เป็นเซตของ Literal แล้ว RDF Graph หนึ่งชุดสามารถเขียนในรูปคณิตศาสตร์ได้ว่า

$$G = \{\langle s_i, p_i, o_i\rangle \mid s_i \in I \cup B,\ p_i \in I,\ o_i \in I \cup B \cup L,\ 1 \leq i \leq n\}$$

สมการนี้บอกโครงสร้างของกราฟและข้อจำกัดของ Triple อย่างกระชับ โดย G คือ RDF Graph ทั้งชุด, n คือจำนวน Triple ใน Graph และ i คือเลขลำดับตั้งแต่ 1 ถึง n สำหรับ Triple ลำดับที่ i สัญลักษณ์ s_i, p_i และ o_i แทน Subject, Predicate และ Object ตามลำดับ เงื่อนไข $s_i \in I \cup B$ หมายความว่า Subject ต้องเป็น IRI หรือ Blank Node ส่วน $p_i \in I$ บังคับให้ Predicate เป็น IRI เพราะความสัมพันธ์ต้องมีชื่อที่อ้างถึงได้อย่างแน่นอน ขณะที่ $o_i \in I \cup B \cup L$ เปิดให้ Object เป็น Resource หรือเป็น Literal ก็ได้

ข้อกำหนดนี้สะท้อนการทำงานของกราฟโดยตรง Subject และ Object ทำหน้าที่เป็น Node ส่วน Predicate ทำหน้าที่เป็นเส้นที่มีชื่อกำกับ หาก Object เป็น IRI หรือ Blank Node ระบบสามารถเชื่อมไปยังข้อมูลของ Resource นั้นได้ แต่หาก Object เป็น Literal การเชื่อมโยงจะสิ้นสุดที่ค่าบรรยาย ดังนั้นการออกแบบ RDF จึงไม่ได้จบที่การเขียนเครื่องหมายให้ถูกต้อง แต่ต้องตัดสินใจด้วยว่าอะไรควรมีตัวตนและมีความสัมพันธ์กับ Resource อื่น และอะไรควรเป็นเพียงค่าที่ใช้บรรยาย Resource ตัวอย่างเช่น โครงการหนึ่งที่ใช้เทคโนโลยี SensorNet สามารถเขียนเป็น Turtle ได้ดังนี้

```
@prefix proj: <https://example.org/resource/project/> .
@prefix org: <https://example.org/resource/organization/> .
@prefix tech: <https://example.org/resource/technology/> .
@prefix voc: <https://example.org/vocab/> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .

proj:A voc:usesTechnology tech:SensorNet .
proj:A voc:responsibleOrg org:PSU .
proj:A rdfs:label "ระบบเฝ้าระวังอัจฉริยะ"@th .
```

สามบรรทัดนี้เป็นสาม Claim ที่ระบบคอมพิวเตอร์อ่านได้ โดยบรรทัดแรกบอกว่าโครงการใช้เทคโนโลยีใด บรรทัดที่สองบอกว่าองค์กรใดรับผิดชอบ และบรรทัดที่สามให้ชื่อภาษาไทยของโครงการ ความต่างจากการเก็บเป็นข้อความธรรมดาคือ RDF แยกสิ่งที่กล่าวถึง ความสัมพันธ์ และค่าบรรยายออกจากกัน ทำให้ข้อมูลถูกสืบค้น รวม และตรวจสอบต่อได้ง่ายกว่า รูปเขียนในตัวอย่างเรียกว่า Turtle ซึ่งเป็น Syntax สำหรับเขียน RDF ให้มนุษย์อ่านได้สะดวก แต่ไม่ใช่แบบจำลองข้อมูลอีกชนิดหนึ่ง ไม่ว่าจะจัดเก็บ RDF ด้วย Turtle, RDF/XML หรือรูปแบบอื่น Triple ที่ได้ยังคงมีความหมายเดียวกัน

IRI แบบเต็มทำให้ตัวระบุไม่ปะปนกันระหว่างแหล่งข้อมูล แต่การเขียน IRI เต็มซ้ำทุกบรรทัดทำให้เอกสารอ่านยาก Turtle จึงให้ประกาศ Prefix ซึ่งเป็นชื่อย่อของ Namespace หรือส่วนต้นร่วมของ IRI บรรทัดที่ขึ้นต้นด้วย **@prefix** ต้องลงท้ายด้วยจุด เมื่อประกาศแล้ว **proj:A** จะขยายกลับเป็น **https://example.org/resource/project/A**, **voc:usesTechnology** จะขยายกลับเป็น **https://example.org/vocab/usesTechnology** และ **rdfs:label** จะขยายกลับไปยัง IRI ของคำศัพท์มาตรฐาน RDFS Prefix จึงเป็นเพียงกลไกย่อชื่อ ไม่ใช่ชนิดของ Resource และไม่ได้เปลี่ยนความหมายของ IRI เครื่องหมายจุดท้าย Triple หมายถึงจบประโยค หาก Subject เดิมมีหลาย Predicate สามารถใช้เครื่องหมายอัฒภาคเพื่อเขียน Subject เพียงครั้งเดียว และคำว่า **a** เป็นรูปย่อของ **rdf:type** ตัวอย่างเช่น

```
proj:A a voc:Project ;
 rdfs:label "ระบบเฝ้าระวังอัจฉริยะ"@th .
```

ข้อความนี้มีสอง Triple กล่าวคือโครงการ A เป็นสมาชิกของ Class **voc:Project** และมีชื่อภาษาไทยตามที่ระบุไว้ การเยื้องบรรทัดเป็นเพียงการจัดให้อ่านง่าย ไม่ได้สร้างลำดับชั้นเพิ่มขึ้นในข้อมูล หากขยายรูปย่อออกมา ทั้งสองประโยคยังคงมี Subject เดียวกัน การเข้าใจความต่างระหว่างแบบจำลอง RDF กับรูปเขียน Turtle ช่วยให้ผู้อ่านมองเห็น Triple ที่ซ่อนอยู่โดยไม่ต้องท่องจำเครื่องหมายทั้งหมด

เมื่อมองเห็น Triple ภายใต้รูปเขียน Turtle ได้แล้ว การตัดสินใจที่สำคัญกว่าการจำเครื่องหมายคือการแยกบทบาทของ IRI, Literal และ Namespace ซึ่งเป็นหัวข้อถัดไป

2.2 แยก IRI, Literal และ Namespace ให้ถูกบทบาท

โครงสร้าง Triple บอกตำแหน่งของข้อมูล แต่ยังไม่บอกว่าสิ่งหนึ่งควรเป็น Resource ที่อ้างซ้ำได้หรือเป็นเพียงค่าบรรยาย หัวข้อนี้จึงเชื่อมจากรูปประโยคของ RDF มาสู่การตัดสินใจเรื่อง Identity และการจัด Namespace ซึ่งมีผลต่อการรวมข้อมูลในระยะยาว

เมื่อเริ่มเขียน RDF จริง สิ่งแรกที่ต้องตัดสินใจไม่ใช่ว่าจะใช้ software ตัวใด แต่เป็นการตัดสินใจว่าอะไรควรมีตัวตนใน Graph การตัดสินใจนี้มีผลต่อการใช้ข้อมูลระยะยาว หากสิ่งหนึ่งถูกเก็บเป็น Literal ระบบจะใช้สิ่งนั้นเป็นค่าบรรยายได้ แต่จะเชื่อมความสัมพันธ์ต่อจากค่านั้นได้ยาก ในทางกลับกัน หากสิ่งหนึ่งถูกกำหนดเป็น IRI สิ่งนั้นสามารถถูกอ้างซ้ำ เชื่อมกับข้อมูลอื่น และใช้เป็นจุดเริ่มต้นของ Query ได้ ตัวอย่างเช่น ถ้าเขียนว่า

```
proj:A voc:usesTechnology "SensorNet" .
```

ระบบจะรู้เพียงว่าโครงการ A มีข้อความคำว่า SensorNet อยู่ในตำแหน่ง Object แต่จะไม่รู้ว่า SensorNet เป็นเทคโนโลยีเดียวกับ Resource อื่นในระบบหรือไม่ ถ้าในอีกแหล่งข้อมูลหนึ่งเขียนว่า **"Sensor Network"** หรือ **"SensorNet Platform"** ระบบจะไม่สามารถรู้ได้เองว่าคำเหล่านี้อาจหมายถึงสิ่งเดียวกัน ดังนั้นในกรณีที่เทคโนโลยีต้องถูกอ้างซ้ำ ควรเขียนเป็น Resource ดังนี้

```
proj:A voc:usesTechnology tech:SensorNet .
tech:SensorNet rdfs:label "SensorNet"@en .
tech:SensorNet rdfs:label "เทคโนโลยี SensorNet"@th .
```

การแยก IRI ออกจาก Label ทำให้ระบบรองรับหลายภาษา รองรับชื่อย่อ และรองรับการเปลี่ยนชื่อในอนาคตได้ดีขึ้น จุดนี้สำคัญในบริบทภาษาไทย เพราะชื่อเดียวกันอาจถูกสะกดหลายแบบ เช่น ภาษาไทยเต็ม ภาษาอังกฤษเต็ม ชื่อย่อ หรือชื่อทางการในเอกสารราชการ หากทุกอย่างถูกเก็บเป็น Literal ตรง ๆ การรวมข้อมูลภายหลังจะลำบากมาก

Namespace ก็มีบทบาทในเชิงการอ่านและการกำกับความหมายเช่นเดียวกัน Prefix เช่น **ex:**, **voc:** หรือ **rdfs:** เป็นเพียงชื่อย่อของ IRI เต็ม แต่ชื่อย่อนี้ช่วยให้มนุษย์อ่าน Graph ได้ง่ายขึ้น อย่างไรก็ตาม Prefix ไม่ใช่นิยามของคำศัพท์และไม่ใช่หลักฐานว่าความหมายถูกต้อง การเห็น **voc:Project** บอกเพียงว่า IRI นี้เกิดจากการนำ Namespace ที่ผูกกับ **voc:** มารวมกับ local part ว่า **Project** สิ่งที่ต้องตรวจต่อคือ IRI เต็มถูกกำหนดไว้ที่ใด มีคำอธิบายหรือไม่ และถูกใช้สม่ำเสมอหรือไม่

ในระบบจริง Prefix Policy ควรถูกกำหนดเป็นเอกสารตั้งแต่ต้น เพราะชื่อย่อที่ใช้ไม่ตรงกันจะสร้างปัญหาเมื่อมีผู้ร่วมงานมากขึ้น ตัวอย่างในหนังสือใช้ **voc:** สำหรับคำศัพท์ และแยก Prefix ของข้อมูลตาม

ชนิดของ Resource เช่น **proj:**, **org:** และ **tech:** ที่สำคัญคือ Namespace ของคำศัพท์ควรแยกจาก Namespace ของข้อมูล เช่น **https://example.org/vocab/Project** ใช้เป็น Class ขณะที่ **https://example.org/resource/project/P001** ใช้ระบุโครงการหนึ่งรายการ ระบบจริงสามารถเลือกชื่อ Prefix อื่นได้ แต่ทุกไฟล์ต้องประกาศให้ชัดว่าชื่อย่อแต่ละชื่อขยายกลับไปยัง Namespace ใด ตัวอย่างย่อยในบทนี้ใช้ชุด Prefix ที่ประกาศไว้ตอนต้น ส่วนตัวอย่างที่ตั้งใจให้เป็นเอกสาร Turtle สมบูรณ์จะประกาศ Prefix ซ้ำภายใน Code block

การแยก Namespace เช่นนี้ช่วยให้ Graph อ่านง่ายและดูแลระยะยาวได้ดีขึ้น เมื่อเห็น IRI ใน Namespace ของคำศัพท์ ผู้อ่านจะคาดหวังว่าเป็น Class หรือ Property ส่วน IRI ใน Namespace ของข้อมูลควรเป็น Resource ที่กล่าวถึงสิ่งหนึ่งในข้อมูลจริง หากสองส่วนนี้ปนกัน Query อาจยังทำงานได้ แต่คนอ่านจะต้องเดาว่า IRI ใดเป็นคำศัพท์และ IRI ใดเป็นข้อมูล ซึ่งทำให้การรวม Graph จากหลายแหล่งผิดพลาดได้ง่ายขึ้น

อีกเรื่องหนึ่งคือ Namespace ไม่ควรถูกตั้งตามโครงสร้างไฟล์หรือระบบภายในมากเกินไป เช่น **/db/table1/row45** อาจสะดวกตอน Mapping จากฐานข้อมูล แต่ถ้าโครงสร้างฐานข้อมูลเปลี่ยน IRI อาจเสียความหมายทันที แนวทางที่มั่นคงกว่าคือออกแบบ IRI ตามสิ่งในโดเมน เช่น **/project/**, **/org/**, **/technology/** และ **/report/** เพราะสิ่งเหล่านี้มักคงอยู่แม้ระบบจัดเก็บภายในจะเปลี่ยน การคิดเช่นนี้ทำให้ IRI เป็น Identity ของความรู้ ไม่ใช่เพียงตำแหน่งของ record ในระบบใดระบบหนึ่ง

อย่างไรก็ตาม การมี IRI และ Namespace ที่ชัดเจนยังรับรองได้เพียงว่าสิ่งต่าง ๆ ถูกอ้างถึงอย่างสม่ำเสมอ ยังไม่เพียงพอที่จะบอกว่า Class และ Property เหล่านั้นสัมพันธ์กันอย่างไร ข้อจำกัดนี้นำไปสู่ RDFS

2.3 เมื่อ RDF ยังสื่อความหมายได้ไม่เพียงพอ

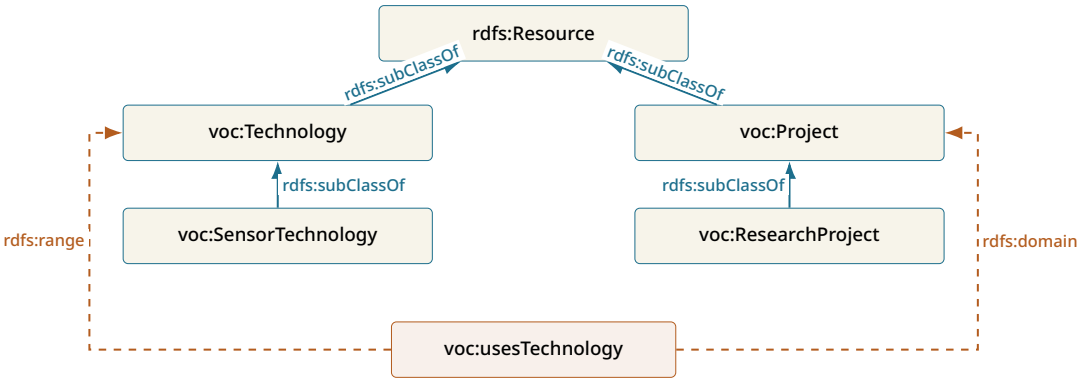
สองหัวข้อก่อนหน้าทำให้เขียน Triple และกำหนด Identity ของ Resource ได้แล้ว ขั้นต่อไปคือพิจารณาว่าโครงสร้างดังกล่าวสื่อความหมายได้เพียงใด เพราะการมี Graph ยังไม่ได้แปลว่าระบบเข้าใจลำดับชั้นของ Class หรือบทบาทของ Property

สมมุติว่ากราฟใช้คำว่า **voc:Project**, **voc:ResearchProject** และ **voc:usesTechnology** RDF ยอมให้ใช้ IRI เหล่านี้ใน Triple ได้ แต่ RDF เพียงลำพังไม่ได้บอกว่า Project เป็นกลุ่มของสิ่งต่าง ๆ ไม่รู้ว่า ResearchProject เป็นชนิดย่อยของ Project และไม่รู้ว่า usesTechnology ควรเชื่อมโครงการไปยังเทคโนโลยี

ข้อจำกัดนี้เป็นเรื่องของความสามารถในการสื่อความหมาย หรือ expressivity กล่าวคือ RDF มีโครงสร้างเพียงพอสำหรับบอกว่า “มีความสัมพันธ์อะไรเกิดขึ้นระหว่างสิ่งใด” แต่ยังมีคำศัพท์ไม่เพียงพอสำหรับบอกว่า Class ต่าง ๆ จัดลำดับกันอย่างไร Property ใดเป็นชนิดย่อยของ Property อื่น หรือเมื่อพบ Property หนึ่งแล้วควรอนุมานชนิดของ Subject และ Object อย่างไร หากแต่ละระบบสร้างคำศัพท์ขึ้นเองโดยไม่ประกาศความหมายเหล่านี้ คนอาจพอเดาได้จากชื่อ แต่โปรแกรมไม่ควรต้องเดาจากการสะกดคำ

RDF Schema หรือ RDFS จึงเพิ่มคำศัพท์มาตรฐานสำหรับอธิบาย Vocabulary ของกราฟ โดยเปิดให้ประกาศ Class, Property, Subclass, Subproperty, Domain และ Range ได้ (World Wide Web Consortium, 2014d,b) RDFS ไม่ได้เปลี่ยน Triple ให้เป็นโครงสร้างชนิดใหม่ เพราะคำประกาศของ RDFS เองก็เขียนเป็น RDF Triple เช่นกัน สิ่งที่เพิ่มขึ้นคือความหมายที่เครื่องมือซึ่งรองรับ RDFS สามารถนำไปอนุมานต่อได้ หากเปรียบ RDF เป็นรูปประโยค RDFS ก็คือคำอธิบายว่าคำในประโยคนั้นทำหน้าที่อะไรและ

สัมพันธ์กับคำอื่นอย่างไร ความสัมพันธ์ระหว่างลำดับชั้นของ Class กับการกำหนด Domain และ Range ของ Property แสดงในภาพที่ 2.1 เพื่อให้เห็นว่า RDFS เพิ่มโครงสร้างของคำศัพท์ให้กับ Triple อย่างไร



ภาพที่ 2.1: RDFS vocabulary สำหรับอธิบาย class, subclass, domain และ range

ภาพที่ 2.1 แสดงลำดับชั้นของ Class ในส่วนบน โดย **voc:ResearchProject** และ **voc:SensorTechnology** เชื่อมไปยัง Class ที่กว้างกว่าด้วย **rdfs:subClassOf** ส่วนล่างแสดงว่า Property **voc:usesTechnology** มี **rdfs:domain** เป็น **voc:Project** และมี **rdfs:range** เป็น **voc:Technology** เมื่อ Vocabulary ระบุโครงสร้างเหล่านี้ ระบบจึงมองเห็นมากกว่าชุดชื่อที่วางอยู่ใน Triple แยกกัน คำศัพท์สำคัญที่ใช้สร้างโครงสร้างดังกล่าวสรุปไว้ในตารางที่ 2.1 คำว่า Class หมายถึงกลุ่มของ Resource ที่มีลักษณะร่วมกัน เช่น กลุ่มโครงการหรือกลุ่มเทคโนโลยี ส่วน Property หมายถึง Resource ที่ใช้เป็น Predicate เพื่อบอกชนิดของความสัมพันธ์ เช่น “ใช้เทคโนโลยี” หรือ “มีหน่วยงานรับผิดชอบ” การประกาศ Class และ Property ไม่ได้สร้างข้อมูลโครงการขึ้นมาเอง แต่ทำให้คำศัพท์ที่ใช้ใน Triple มีนิยามร่วมกัน

ตารางที่ 2.1 แสดงตัวอย่างเป็น Triple ที่เขียนด้วย Turtle โดยใช้ Prefix ที่ประกาศไว้ตอนต้นบท จุดที่ควรแยกให้ชัดคือ **rdf:type** ใช้เป็น Predicate ส่วน **rdfs:Class** และ **rdf:Property** เป็นชื่อ Class และอยู่ในตำแหน่ง Object ของตัวอย่างที่ใช้ประกาศชนิดของคำศัพท์ เช่น **voc:Project** และ **voc:usesTechnology** ตามลำดับ (World Wide Web Consortium, 2014d)

ตารางที่ 2.1: คำศัพท์มาตรฐานของ RDF และ RDFS ที่ใช้บ่อยในบทนี้

คำศัพท์	ความหมาย	ตัวอย่าง Triple ใน Turtle
rdf:type	ระบุว่า Resource เป็นสมาชิกของ Class ใด	**proj:A rdf:type voc:Project .**
rdfs:Class	Class ที่มีสมาชิกเป็น Class ต่าง ๆ ตัวอย่างนี้ประกาศว่า **voc:Project** เป็น Class	**voc:Project rdf:type rdfs:Class .**
rdf:Property	Class ที่มีสมาชิกเป็น Property ต่าง ๆ ตัวอย่างนี้ประกาศว่า **voc:usesTechnology** เป็น Property	**voc:usesTechnology rdf:type rdf:Property .**

ตารางที่ 2.1: คำศัพท์มาตรฐานของ RDF และ RDFS ที่ใช้บ่อยในบทนี้ (ต่อ)

คำศัพท์	ความหมาย	ตัวอย่าง Triple ใน Turtle
rdfs:subClassOf	ระบุว่าสมาชิกทุกตัวของ Class แรกเป็นสมาชิกของ Class ที่สองด้วย	**voc:ResearchProject rdfs:subClassOf voc:Project .**
rdfs:subPropertyOf	ระบุว่าคู่ Subject และ Object ที่สัมพันธ์กันด้วย Property แรก สัมพันธ์กันด้วย Property ที่สองด้วย	**voc:mainPartner rdfs:subPropertyOf voc:partner .**
rdfs:domain	ระบุ Class ที่อนุมานให้ Subject ของ Triple ที่ใช้ Property นี้	**voc:usesTechnology rdfs:domain voc:Project .**
rdfs:range	ระบุ Class ที่อนุมานให้ Object ของ Triple ที่ใช้ Property นี้	**voc:usesTechnology rdfs:range voc:Technology .**
rdfs:label	ให้ชื่อที่มนุษย์อ่านได้ โดยใช้ Literal และอาจระบุภาษา	**proj:A rdfs:label "โครงการ A"@th .**
rdfs:comment	ให้คำอธิบาย Resource โดยใช้ Literal และอาจระบุภาษา	**voc:usesTechnology rdfs:comment "เทคโนโลยีที่โครงการใช้"@th .**

ตัวอย่างต่อไปประกาศ Class สำหรับโครงการและเทคโนโลยี พร้อมประกาศ Property ที่ใช้เชื่อมโครงการไปยังเทคโนโลยี

```
voc:Project rdf:type rdfs:Class .
voc:Technology rdf:type rdfs:Class .
voc:usesTechnology rdf:type rdf:Property .
voc:usesTechnology rdfs:domain voc:Project .
voc:usesTechnology rdfs:range voc:Technology .
```

เมื่อมี Triple ว่า

```
proj:A voc:usesTechnology tech:SensorNet .
```

คำประกาศ **voc:usesTechnology rdfs:domain voc:Project** กล่าวถึง Property **voc:usesTechnology** ว่าเมื่อใช้เป็น Predicate แล้ว Subject ของ Triple นั้นจะเป็นสมาชิกของ **voc:Project** ส่วนคำประกาศ **rdfs:range voc:Technology** ทำให้อนุมานว่า Object เป็นสมาชิกของ **voc:Technology** ดังนั้น เมื่อนำคำประกาศทั้งสองมาใช้กับ Triple ข้างต้น ระบบที่รองรับ RDFS จะอนุมานข้อความเพิ่มได้ดังนี้

```
proj:A rdf:type voc:Project .
tech:SensorNet rdf:type voc:Technology .
```

จึงควรแยกคำประกาศ Domain/Range ออกจากผลการอนุมาน: คำประกาศใช้ Property เป็น Subject ส่วนผลการอนุมานใช้ Resource ในข้อมูลเป็น Subject และใช้ **rdf:type** ระบุ Class ที่ได้ การประกาศ

Domain/Range ไม่ได้หมายความว่าระบบจะปฏิเสธข้อมูลที่ไม่มี **rdf:type** ระบุไว้ล่วงหน้า หากใช้ Property ผิด ระบบอาจอนุมานชนิดที่ไม่ต้องการเพิ่มขึ้นแทนที่จะรายงานข้อผิดพลาดทันที (World Wide Web Consortium, 2014d; Allemang et al., 2020)

ในเชิงตรรกะ กฎการอนุมานของ RDFS สามารถเขียนเป็น Horn Rule อย่างง่าย เช่น หาก p มี Domain C และมี Triple $p(s, o)$ แล้วสามารถอนุมานได้ว่า $C(s)$ หรือเขียนได้ว่า

$$p(s, o) \wedge \mathit{Domain}(p, C) \rightarrow C(s)$$

ในกฎนี้ p แทน Property หนึ่ง, s แทน Subject, o แทน Object และ C แทน Class ที่ถูกประกาศเป็น Domain ของ Property นั้น นิพจน์ $p(s, o)$ หมายถึงมี Triple ที่เชื่อม s ไปยัง o ด้วย Property p ส่วน $\mathit{Domain}(p, C)$ หมายถึง p มี Domain เป็น C ดังนั้นผลลัพธ์ $C(s)$ จึงอ่านได้ว่า s เป็นสมาชิกของ Class C หลักนี้ทำให้ RDFS เป็นเครื่องมือที่เหมาะกับการจัดระเบียบ Vocabulary เบื้องต้น และช่วยให้ Query ครอบคลุม Entity ที่อยู่ภายใต้ Class Hierarchy ได้มากขึ้น แต่ยังไม่รองรับเงื่อนไขเชิงตรรกะที่ละเอียดเท่า OWL

เมื่อทราบคำศัพท์มาตรฐานของ RDFS แล้ว ขั้นต่อไปคือเลือกเฉพาะ Class และ Property ที่จำเป็นต่อคำถามของงาน และจัดคำเหล่านั้นเป็น Vocabulary ของโดเมน

2.4 จากคำศัพท์มาตรฐานสู่ Vocabulary ของงาน

RDFS มีคำศัพท์สำหรับประกาศ Class, Property, Domain และ Range แต่ไม่ได้กำหนดว่าโดเมนหนึ่งควรมีคำใดบ้าง การเปลี่ยนจากมาตรฐานทั่วไปสู่ Vocabulary ของงานจึงต้องเริ่มจากคำถามและข้อมูลที่ระบบต้องรองรับ

ในงานปฏิบัติ การออกแบบ RDFS Vocabulary ที่ดีควรเริ่มจากคำถามของระบบ ไม่ใช่เริ่มจากการสร้าง Class ให้ครบทุกคำที่พบในเอกสาร หากระบบต้องตอบคำถามว่าโครงการใดใช้เทคโนโลยีใด ใครรับผิดชอบโครงการ และโครงการนั้นมีชื่อภาษาไทยว่าอะไร Vocabulary รุ่นแรกอาจมีเพียง **voc:Project**, **voc:Technology**, **voc:Organization**, **voc:usesTechnology**, **voc:responsibleOrg** และ **rdfs:label** ก็เพียงพอแล้ว การเริ่มจากเฉพาะคำศัพท์ที่จำเป็นต่อคำถามช่วยให้ตรวจได้ง่ายว่าทุก Class และ Property มีหน้าที่อะไร แล้วจึงค่อยขยายเมื่อมีคำถามหรือข้อมูลชนิดใหม่เข้ามา วิธีนี้ลดคำศัพท์ที่ยังไม่มีการใช้งานและช่วยให้ผู้ที่เพิ่งเริ่มอ่าน RDF มองเห็นความสัมพันธ์หลักของกราฟได้ชัดเจน

ผู้อ่านจำนวนมากเริ่มต้นจากข้อมูลแบบตาราง ดังนั้นการอธิบาย RDF ควรเริ่มจากรูปแบบที่คุ้นเคยก่อน สมมติว่ามีข้อมูลโครงการดังตารางที่ 2.2

ตารางที่ 2.2: ตัวอย่างข้อมูลโครงการในรูปตารางก่อนแปลงเป็น RDF

Project	org	Technology	Label
A	PSU	SensorNet	ระบบเฝ้าระวังอัจฉริยะ

ถ้ามองแบบฐานข้อมูลทั่วไป ตารางนี้มีหนึ่งแถวและสี่คอลัมน์ แต่ถ้ามองแบบ RDF ต้องถามต่อว่าแต่ละคอลัมน์ควรถูกแปลงเป็น Resource, Property หรือ Literal อย่างไร คอลัมน์ **project** อาจใช้สร้าง IRI ของ

โครงการ คอลัมน์ **org** อาจเป็น Organization Resource คอลัมน์ **technology** อาจเป็น Technology Resource และคอลัมน์ **label** ควรเป็น Literal ภาษาไทย เมื่อแปลงเป็น RDF จึงอาจได้ดังนี้

```
proj:A a voc:Project ;
 rdfs:label "ระบบเฝ้าระวังอัจฉริยะ"@th ;
 voc:responsibleOrg org:PSU ;
 voc:usesTechnology tech:SensorNet .

org:PSU a voc:Organization .
tech:SensorNet a voc:Technology .
```

สิ่งที่เพิ่มขึ้นจากตารางเดิมไม่ใช่เพียง Syntax แต่เป็นการตัดสินใจเชิงความหมาย ต้องตัดสินว่า Project, Organization และ Technology เป็นสิ่งที่ควรถูกอ้างซ้ำใน Graph ส่วน Label เป็นคำบรรยาย ไม่ใช่ Resource ใหม่ การแปลงเช่นนี้ทำให้ข้อมูลจากหลายตารางหรือหลายระบบรวมกันได้ง่ายขึ้น เพราะ Resource เดียวกันสามารถมี IRI เดียวกันได้ หากใช้ R2RML หรือ Mapping language อื่น หลักคิดก็ยังเหมือนกัน คือกำหนด Mapping จากคอลัมน์ไปสู่ RDF Pattern ที่ชัดเจน (World Wide Web Consortium, 2012) สิ่งที่ไม่ควรทำคือแปลงทุก column เป็น string แล้วประกาศว่าข้อมูลนั้นเป็น RDF เพราะผลลัพธ์อาจถูกต้องตาม Syntax แต่ไม่ช่วยให้เกิด Graph ที่มีความหมายจริง

Vocabulary ที่กำหนดขึ้นไม่ได้มีผลเฉพาะต่อการอ่าน Graph แต่ยังเป็นฐานของการอนุมานด้วย หัวข้อถัดไปจึงพิจารณาทั้งประโยชน์และความเสี่ยงของ RDFS Inference

2.5 ผลอนุมานของ RDFS และสิ่งที่ต้องระวัง

เมื่อ Mapping ใช้ Vocabulary ที่มี Domain และ Range ระบบสามารถสรุปข้อมูลบางส่วนเพิ่มจาก Triple ที่มีอยู่ได้ อย่างไรก็ตาม ผลอนุมานที่ถูกต้องตามกฎอาจยังผิดจากเจตนาของข้อมูล หาก Property ถูกใช้ผิดบริบท จึงต้องอ่านทั้งประโยชน์และข้อควรระวังควบคู่กัน

RDFS ทำให้ระบบ Infer ข้อมูลใหม่จาก Vocabulary ที่กำหนดไว้ได้ ตัวอย่างเช่น หากกำหนดว่า **voc:usesTechnology** มี Domain เป็น **voc:Project** และ Range เป็น **voc:Technology** แล้วมี Triple ว่า **proj:A voc:usesTechnology tech:SensorNet** ระบบสามารถ Infer ได้ว่า **proj:A** เป็น **voc:Project** และ **tech:SensorNet** เป็น **voc:Technology** ข้อดีคือ Query สามารถดึงข้อมูลได้กว้างขึ้น เช่น แม้ไม่ได้เขียน type ของโครงการไว้โดยตรง ระบบก็อาจ Infer type ได้จาก Property ที่ใช้ แต่ข้อเสียคือหากใช้ Property ผิด ระบบจะ Infer type ผิดตามไปด้วย ตัวอย่างเช่น ถ้าใส่ Object เป็น Organization โดยผิดพลาด

```
proj:A voc:usesTechnology org:PSU .
```

ระบบ RDFS อาจ Infer ว่า **org:PSU** เป็น **voc:Technology** เพราะ Range ของ Property บอกไว้เช่นนั้น นี่คือเหตุผลที่ต้องเข้าใจว่า RDFS Domain และ Range ไม่ใช่ Validation Rule แต่เป็น Inference Rule หากต้องการตรวจว่าค่า Object ต้องเป็น Technology จริงหรือไม่ ควรใช้ SHACL ในบทที่ว่าด้วย Validation ลำดับเหตุผลของกรณีนี้ประกอบด้วย

1. **voc:usesTechnology** มี Range เป็น **voc:Technology**
2. Graph มี Triple ที่เชื่อมโครงการ A ไปยัง **org:PSU** ด้วย Property ดังกล่าว

3. RDFS Rule จึงสรุปว่า **org:PSU** เป็น **voc:Technology**

ผลดังกล่าวถูกต้องตามกฎ RDFS แต่ผิดเจตนาของข้อมูลต้นทาง ปัญหาจึงไม่ได้อยู่ที่ Reasoner ทำงานผิด แต่อยู่ที่ข้อมูลหรือ Mapping ใช้ Property ผิดชั้นความหมาย

ความเข้าใจนี้สำคัญมากสำหรับผู้อ่านที่มาจากฐานข้อมูลสัมพันธ์ เพราะในฐานข้อมูลสัมพันธ์ Foreign Key หรือ Constraint มักถูกใช้เพื่อปฏิเสธข้อมูลที่ผิด แต่ RDFS Domain/Range มีหน้าที่หลักในการเพิ่มความหมาย เมื่อเห็นผล Infer ที่แปลก จึงควรถามก่อนว่า Property ถูกใช้ถูกบริบทหรือไม่ Instance ที่เป็น Object ควรเป็น Resource ชนิดใด และถ้าต้องการตรวจความผิดควรเพิ่ม SHACL Shape ใดใน Pipeline การแยกคำถามเหล่านี้ช่วยไม่ให้นำ RDFS ไปใช้แทน Validation และเตรียมฐานให้เข้าใจ OWL กับ SHACL ในบทถัดไปได้ดีขึ้น

ตัวอย่างนี้แสดงความผิดพลาดหนึ่งรูปแบบ แต่การเริ่มใช้ RDF/RDFS ยังมีข้อผิดพลาดด้านการตั้งชื่อ Identity และขอบเขตของ Class อีกหลายประการ จึงควรรวบรวมไว้พิจารณาก่อนขยาย Graph

2.6 ข้อผิดพลาดที่พบบ่อยเมื่อเริ่มใช้ RDF/RDFS

หัวข้อก่อนหน้าแสดงให้เห็นว่าความผิดพลาดใน RDF/RDFS ไม่จำเป็นต้องทำให้ Syntax ผิดหรือระบบหยุดทำงาน Graph อาจประมวลผลได้ตามปกติแต่สื่อความหมายคลาดเคลื่อน หัวข้อนี้จึงรวบรวมข้อผิดพลาดที่ควรตรวจตั้งแต่ Graph ยังมีขนาดเล็ก

1. **ใช้ Property กว้างเกินไป** เช่น ใช้ **voc:relatedTo** สำหรับทุกความสัมพันธ์ แม้ Graph จะดูเชื่อมกันมากขึ้น แต่ระบบไม่สามารถแยกได้ว่าความสัมพันธ์นั้นหมายถึงการใช้เทคโนโลยี การเป็นเจ้าของ การเกิดขึ้นในเวลา หรือการมีหลักฐานรองรับ Property ที่สำคัญต่อ Query จึงควรมีความหมายเฉพาะเจาะจงเพียงพอ
2. **ใช้ Label เป็น Identity** เช่น ตั้ง IRI จากชื่อเต็มของโครงการซึ่งอาจเปลี่ยนแปลง เมื่อชื่อโครงการเปลี่ยน IRI จะเปลี่ยนตามและทำให้ Link เดิมเสีย แนวทางที่มั่นคงกว่าคือใช้ Identifier ที่คงที่ แล้วใช้ **rdfs:label** สำหรับชื่อที่มนุษย์อ่าน
3. **สร้าง Class มากเกินไปในรุ่นแรก** จำนวน Class ที่มากไม่ได้ทำให้ Vocabulary สมบูรณ์ขึ้นโดยอัตโนมัติ Class ที่ไม่รองรับ Query หรือการอนุมานทำให้ Graph อ่านยากโดยไม่เพิ่มประโยชน์ รุ่นแรกจึงควรเริ่มจาก Class ที่จำเป็นต่อข้อมูลและคำถาม
4. **ขาด Label และ Comment ที่ชัดเจน** คำศัพท์ RDF/RDFS ไม่ได้ถูกใช้โดยเครื่องเท่านั้น แต่ยังถูกอ่านโดยผู้พัฒนา นักวิจัย และผู้เชี่ยวชาญโดเมน การกำหนด **rdfs:label** และ **rdfs:comment** จึงช่วยลดความคลาดเคลื่อนของความหมายระหว่างผู้เกี่ยวข้อง

เมื่อทราบข้อผิดพลาดเหล่านี้แล้ว ขั้นต่อไปคือกลับไปยังข้อมูลโครงการและขยาย Graph อย่างมีหลักเกณฑ์ โดยตรวจ Identity, Datatype และความสัมพันธ์ไปพร้อมกัน

2.7 ขยายตัวอย่างข้อมูลโครงการด้วย RDF/RDFS

หลักการก่อนหน้านี้จะเห็นผลชัดขึ้นเมื่อใช้กับข้อมูลที่มีหลาย Resource และหลายความสัมพันธ์ หัวข้อนี้จึงขยายตัวอย่างโครงการจากบทแรก เพื่อแสดงลำดับการตัดสินใจตั้งแต่ข้อมูลตารางไปจนถึง RDF/RDFS ที่นำไป Query ต่อได้

เพื่อให้เห็นภาพชัดขึ้น สมมติว่ามีข้อมูลโครงการวิจัยหรือโครงการพัฒนาระบบอยู่ชุดหนึ่ง ข้อมูลเริ่มต้นอาจอยู่ใน Spreadsheet ธรรมดา มีคอลัมน์ชื่อโครงการ หน่วยงานรับผิดชอบ เทคโนโลยีที่ใช้ วันที่เริ่มต้น สถานะ และเอกสารอ้างอิง หากดูในมุมของผู้ใช้ทั่วไป ข้อมูลชุดนี้ก็คือตารางรายการโครงการ แต่ถ้าดูในมุม Semantic Web ต้องถามต่อว่าแต่ละคอลัมน์ควรกลายเป็นอะไรใน Graph เพราะไม่ใช่ทุกช่องในตารางควรถูกแปลงเป็น Literal ทั้งหมด ตารางที่ 2.3 เป็นตัวอย่างข้อมูลเริ่มต้นที่มีรหัสโครงการ สถานะ และรายงานอ้างอิง

ตารางที่ 2.3: ตัวอย่างข้อมูลโครงการที่มีรหัส สถานะ และรายงานอ้างอิง

Code	org	tech	status	Report
P001	PSU	SensorNet	Running	R2026-01

หากแปลงทุกค่าเป็นข้อความโดยไม่แยกสิ่งที่ต้องอ้างซ้ำ อาจเขียน RDF ได้ว่าโครงการ P001 มี Organization เป็นข้อความ “PSU” มี Technology เป็นข้อความ “SensorNet” และมี Report เป็นข้อความ “R2026-01” การเขียนแบบนี้ทำได้และไม่ผิด Syntax แต่ Graph ที่ได้จะเชื่อมโยงและสืบค้นต่อได้จำกัด เพราะสิ่งที่ควรมี IRI กลับถูกเก็บเป็นข้อความธรรมดา หากวันหนึ่งต้องการถามต่อว่า PSU มีโครงการใดบ้าง หรือ SensorNet ถูกใช้ในโครงการอื่นด้วยหรือไม่ ระบบจะทำงานยากขึ้นมาก ดังนั้นการแปลงเป็น RDF ที่เหมาะสมกว่าคือให้สิ่งสำคัญมี IRI ของตนเอง เช่น โครงการ องค์กร เทคโนโลยี และรายงาน ส่วนค่าที่เป็นชื่ออ่านง่ายหรือวันที่จึงค่อยเก็บเป็น Literal ตัวอย่างเช่น

```
@prefix proj: <https://example.org/resource/project/> .
@prefix org: <https://example.org/resource/organization/> .
@prefix tech: <https://example.org/resource/technology/> .
@prefix status: <https://example.org/resource/status/> .
@prefix report: <https://example.org/resource/report/> .
@prefix voc: <https://example.org/vocab/> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .

proj:P001 a voc:Project ;
 rdfs:label "โครงการระบบเฝ้าระวังอัจฉริยะ"@th ;
 voc:responsibleOrg org:PSU ;
 voc:usesTechnology tech:SensorNet ;
 voc:hasStatus status:Running ;
 voc:hasReport report:R2026-01 ;
 voc:startDate "2026-01-15"^^xsd:date .

org:PSU a voc:Organization ;
 rdfs:label "มหาวิทยาลัยสงขลานครินทร์"@th ;
```

```
    rdfs:label "Prince of Songkla University"@en .

tech:SensorNet a voc:Technology ;
 rdfs:label "SensorNet"@en .
```

ตัวอย่างนี้ดูยาวกว่าตารางเดิมมาก แต่ความยาวที่เพิ่มขึ้นคือความชัดเจนของความหมาย โครงการ องค์กร เทคโนโลยี สถานะ และรายงานกลายเป็น Resource ที่อ้างซ้ำได้ทั้งหมด หากภายหลังมีข้อมูลจากอีกระบบหนึ่งที่กล่าวถึง **org:PSU** Resource เดียวกันนี้ก็สามารถเชื่อมกับข้อมูลเดิมได้ทันที ไม่ต้องเดาจากข้อความว่า “PSU”, “P.S.U.” หรือ “Prince of Songkla University” หมายถึงสิ่งเดียวกันหรือไม่ เมื่อได้ Resource พื้นฐานแล้ว RDFS จะช่วยจัดระเบียบคำศัพท์ให้ชัดขึ้น เช่น

```
voc:Project rdf:type rdfs:Class .
voc:Organization rdf:type rdfs:Class .
voc:Technology rdf:type rdfs:Class .
voc:Report rdf:type rdfs:Class .

voc:ResearchProject rdfs:subClassOf voc:Project .
voc:DigitalTransformationProject rdfs:subClassOf voc:Project .

voc:responsibleOrg rdf:type rdf:Property ;
 rdfs:domain voc:Project ;
 rdfs:range voc:Organization .

voc:usesTechnology rdf:type rdf:Property ;
 rdfs:domain voc:Project ;
 rdfs:range voc:Technology .
```

จากนิยามนี้ หาก Graph มี Triple ว่า **proj:P001 voc:usesTechnology tech:SensorNet** ระบบที่ใช้ RDFS Reasoning สามารถช่วยเติมความรู้พื้นฐานได้ว่า **proj:P001** เป็น **voc:Project** และ **tech:SensorNet** เป็น **voc:Technology** แม้ Triple เหล่านี้อาจไม่ได้ถูกเขียนไว้ตรง ๆ ในข้อมูลดิบ จุดนี้ทำให้ RDFS มีประโยชน์ต่อการสืบค้น เพราะ Query สามารถพึ่ง Class และ Property ที่นิยามไว้ใน Vocabulary ได้ อย่างไรก็ตาม ตัวอย่างเดียวกันนี้ก็แสดงความเสี่ยงของ RDFS ด้วย หาก Mapping ผิดและเขียนว่า

```
proj:P001 voc:usesTechnology org:PSU .
```

ระบบจะไม่เข้าใจว่าผู้เขียนทำผิดแบบ Validation แต่จะ Infer ตาม Range ของ **voc:usesTechnology** ว่า **org:PSU** เป็น **voc:Technology** ด้วย เหตุนี้ผู้พัฒนาต้องเข้าใจให้ชัดว่า RDFS Domain และ Range คือกฎสำหรับการอนุมาน ไม่ใช่ Constraint สำหรับปฏิเสธข้อมูลผิด หากต้องการให้ระบบรายงานว่า Object ของ **voc:usesTechnology** ต้องเป็น Technology จริง ต้องใช้ SHACL ในบทที่ว่าด้วย Validation

ในงานที่มีข้อมูลจากหลายตาราง การออกแบบ IRI Policy ยิ่งสำคัญ เช่น ตารางแรกอาจมีรหัสโครงการ **P001** ตารางถัดมาอาจมีรหัสรายงาน **R2026-01** และอีกตารางหนึ่งมีชื่อหน่วยงาน ถ้าทุกตารางสร้าง IRI ตามใจตนเอง Graph จะรวมกันยากมาก จึงควรกำหนดรูปแบบ IRI ตั้งแต่ต้น เช่น **/project/**, **/org/**, **/tech/**, **/report/** และ **/status/** ให้สอดคล้องกันทั้งระบบ การกำหนดเช่นนี้ไม่ใช่เรื่องความสวยงามของ URL เท่านั้น แต่เป็นโครงสร้างของ Identity ในระบบความรู้

การใช้ R2RML หรือเครื่องมือ Mapping อื่นจะช่วยทำให้การแปลงจากฐานข้อมูลสัมพันธ์ไปเป็น RDF ทำซ้ำได้มากขึ้น (World Wide Web Consortium, 2012) แต่เครื่องมือเหล่านี้ไม่สามารถตัดสินความหมาย

แทนผู้ออกแบบทั้งหมดได้ ผู้เขียน Mapping ยังต้องตอบคำถามเดิมอยู่เสมอว่า field ใดควรเป็น Resource field ใดควรเป็น Literal และ Relation ใดควรเป็น Predicate การมีมาตรฐาน Mapping จึงช่วยให้กระบวนการเป็นระบบ แต่ไม่แทนที่การพิจารณาความหมายของข้อมูล

หากมองกลับไปยังตัวอย่างในบทแรก จะเห็นว่าหลักของ Semantic Web ไม่ได้ซับซ้อนเกินไป จุดตั้งต้นคือการเปลี่ยนข้อมูลที่เคยอยู่ในตารางหรือเอกสาร ให้กลายเป็น Claim ที่มี Subject, Predicate และ Object ชัดเจน จากนั้นใช้ RDFS เพื่อจัดระเบียบคำศัพท์พื้นฐานให้ Graph อ่านได้ทั้งโดยมนุษย์และเครื่อง เมื่อฐานนี้มั่นคง บทถัดไปจึงค่อยเพิ่ม OWL เพื่อให้นิยามความหมายได้ละเอียดขึ้นและใช้ Reasoning ได้มากขึ้น

2.8 เมื่อ RDF Graph ใหญ่ขึ้นและต้องรวมข้อมูล

ตัวอย่างโครงการหนึ่งชุดยังไม่สะท้อนปัญหาของระบบจริงทั้งหมด เมื่อ Graph รับ Claim จากหลายแหล่ง ประเด็นเรื่อง IRI ร่วม Resource ซ้ำ และบริบทของข้อมูลจะมีผลชัดเจนขึ้น หัวข้อนี้จึงขยายจากการสร้าง Graph หนึ่งชุดไปสู่การรวม Graph หลายชุด

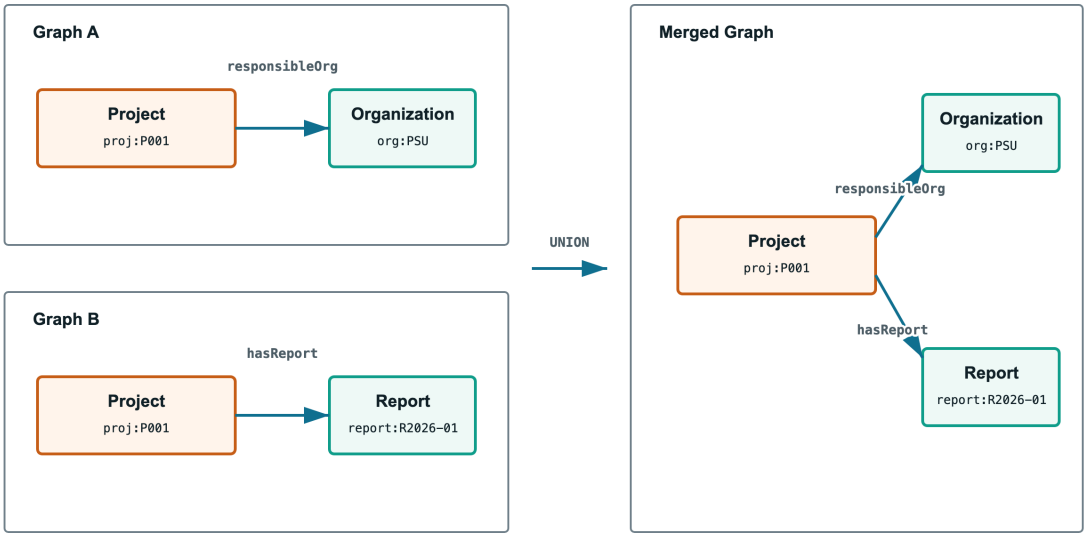
เมื่อข้อมูลเริ่มมีมากขึ้น RDF Graph จะไม่ได้มีเพียง Triple สองสามบรรทัด แต่จะมี Claim จากหลายแหล่งมารวมกัน เช่น ข้อมูลโครงการจากระบบบริหารงาน ข้อมูลรายงานจากคลังเอกสาร ข้อมูลเทคโนโลยีจาก Taxonomy และข้อมูลหน่วยงานจากฐานข้อมูลบุคลากร หลักของ RDF คือ Graph หลายชุดสามารถถูกรวมกันได้ด้วยการ union ของ Triple หาก IRI ของสิ่งเดียวกันตรงกัน ข้อมูลก็จะเชื่อมกันโดยธรรมชาติ สมมติว่า Graph แรกมาจากระบบโครงการ

```
proj:P001 voc:responsibleOrg org:PSU .
proj:P001 voc:usesTechnology tech:SensorNet .
```

Graph ที่สองมาจากระบบรายงาน

```
proj:P001 voc:hasReport report:R2026-01 .
report:R2026-01 rdfs:label "รายงานความก้าวหน้าไตรมาส 1"@th .
```

เมื่อรวมสอง Graph นี้เข้าด้วยกัน ระบบจะเห็นว่า **proj:P001** เป็น Resource เดียวกัน และสามารถถามต่อได้ว่าโครงการ P001 ใช้เทคโนโลยีอะไร มีรายงานอะไร และอยู่ภายใต้หน่วยงานใด การรวมเช่นนี้เป็นเหตุผลสำคัญที่ RDF ให้ความสำคัญกับ IRI เพราะ IRI ทำหน้าที่เป็นตัวระบุร่วมของ Resource ใน Graph ที่มาจากหลายแหล่ง กลไกที่ Graph สองชุดเชื่อมกันผ่าน IRI เดียวกันแสดงในภาพที่ 2.2



ภาพที่ 2.2: การรวม RDF Graph จากสองแหล่งด้วย IRI ของโครงการเดียวกัน

ภาพที่ 2.2 แสดงว่าข้อมูลของโครงการจากระบบหนึ่งและข้อมูลรายงานจากอีกระบบหนึ่งไม่ได้เชื่อมกัน เพราะข้อความชื่อโครงการคล้ายกัน แต่เชื่อมกันเพราะทั้งสอง Graph ใช้ **proj:P001** เป็น Subject เดียวกัน อย่างไรก็ตาม การ union Graph ไม่ได้แก้ปัญหาทุกอย่าง หากแหล่งหนึ่งใช้ **proj:P001** แต่อีกแหล่งใช้ **proj:001** เพื่อหมายถึงโครงการเดียวกัน ระบบจะมองเป็นคนละ Resource ทันที เว้นแต่จะมีข้อมูลเพิ่มเติมระบุว่าตัวระบุทั้งสองหมายถึงสิ่งเดียวกัน ดังนั้นการกำหนด IRI Policy ตั้งแต่ต้นจึงมีผลมากต่อความสำเร็จของกราฟข้อมูล

อีกเรื่องที่พบบ่อยคือการใช้ Blank Node ซึ่งเหมาะกับกรณีที่มีสิ่งหนึ่งอยู่ใน Graph แต่ไม่ต้องการตั้ง IRI ให้ เช่น address ชั่วคราวหรือโครงสร้างข้อมูลภายใน แต่สำหรับสิ่งที่ต้องอ้างซ้ำ ค้นหา หรือทำ Provenance ไม่ควรใช้ Blank Node เป็นหลัก เพราะจะอ้างกลับยาก ตัวอย่างเช่น Report, Claim, Project และ Organization ควรมี IRI ของตนเองมากกว่าเป็น Blank Node ในกรณี Claim ที่ต้องมี Metadata ควรสร้าง Claim เป็น Resource แยกต่างหากแทนที่จะพยายามใส่ข้อมูลทุกอย่างลงใน Triple เดียว เช่น

```
claim:C001 a voc:OutcomeClaim ;
 voc:aboutProject proj:P001 ;
 voc:assertedOutcome outcome:ReducedDelay ;
 voc:hasConfidence "0.82"^^xsd:decimal .
```

การทำ Claim เป็น Resource ทำให้สามารถเชื่อม Provenance, Confidence, Reviewer และ Validation status ได้ง่ายขึ้น นี่เป็นตัวอย่างที่แสดงว่า RDF Triple เรียบง่าย แต่การออกแบบ Graph ที่ดีต้องคิดเรื่องสิ่งที่ต้องอ้างอิงในอนาคตด้วย หาก Claim มีบทบาทต่อการตัดสินใจ การให้ Claim มี IRI เป็นการลงทุนที่คุ้มค่า Datatype และ language tag ก็มีผลต่อคุณภาพของ Graph เช่น วันที่ควรใช้ **xsd:date** หรือ **xsd:dateTime** ไม่ใช่ string ธรรมดา ตัวเลขควรใช้ Datatype ที่เหมาะสม และ Label ภาษาไทยควรมี **@th** เพื่อให้ Query เลือกภาษาได้ การละเลยรายละเอียดเหล่านี้อาจไม่ทำให้ RDF ผิด Syntax แต่ทำให้ Query และการแสดงผลภายหลังลำบากขึ้น

เมื่อ Graph ขยายตัวขึ้น ผู้เขียนควรคิดเสมอว่า RDF ไม่ได้เป็นเพียง format สำหรับเก็บข้อมูล แต่เป็นสัญญาระหว่างแหล่งข้อมูลหลายชุด หากสัญญาเรื่อง IRI, Datatype, Label, Class และ Property ไม่ชัด

Graph จะรวมกันได้ในเชิง Syntax แต่ไม่รวมกันในเชิงความหมาย ดังนั้น RDF/RDFS จึงเป็นบทพื้นฐานที่สำคัญมากก่อนเข้าสู่ OWL, SPARQL และ SHACL

เพื่อให้เห็นความต่อเนื่องของทั้งบท ลองย้อนจากข้อมูลต้นทางหนึ่งแถวก่อน สมมุติว่าตารางโครงการมีรหัสโครงการ ชื่อโครงการ หน่วยงานรับผิดชอบ และเทคโนโลยีที่ใช้ การแปลงข้อมูลไม่ได้เริ่มจากการเปลี่ยนชื่อคอลัมน์ให้เป็น Prefix แต่เริ่มจากการถามว่าค่าใดแทนสิ่งที่มีตัวตน รหัสโครงการใช้สร้าง IRI ของโครงการ ชื่อโครงการเป็น Label หน่วยงานรับผิดชอบควรถูกเชื่อมไปยัง Resource ขององค์กร และเทคโนโลยีควรถูกเชื่อมไปยัง Resource ที่ใช้อ้างซ้ำได้ การตอบคำถามเช่นนี้ทำให้ Mapping เป็นผลจากการเข้าใจข้อมูล ไม่ใช่การแปลงรูปแบบเชิงกลไก

เมื่อข้อมูลแต่ละแถวถูกแปลงเป็น Triple แล้ว RDFS จึงเข้ามาช่วยอธิบายคำศัพท์ที่ Mapping ใช้ กำหนดว่า Resource ประเภทโครงการและองค์กรอยู่ใน Class ใด Property ที่เชื่อมกันมีความหมายอะไร และ Class ใดอยู่ใต้ Class ที่กว้างกว่า ลำดับเช่นนี้ช่วยให้ผู้อ่านเห็นว่า RDFS ไม่ได้สร้างข้อเท็จจริงของโครงการขึ้นมาเอง แต่เพิ่มกรอบสำหรับอ่านข้อเท็จจริงที่อยู่ใน RDF Graph หากข้อมูลต้นทาง Map ผิด RDFS อาจอนุมานต่อจากความผิดนั้นได้ จึงต้องตรวจทั้ง Mapping และ Vocabulary ควบคู่กัน

จุดนี้ยังช่วยอธิบายความต่างระหว่างการอนุมานกับการตรวจความครบถ้วน สมมุติว่ากำหนด Domain ของ **voc:responsibleOrg** เป็น **voc:Project** เมื่อพบ Triple ที่ใช้ Property นี้ Reasoner สามารถอนุมานว่า Subject เป็นโครงการ แต่ถ้าโครงการหนึ่งไม่มี **voc:responsibleOrg** เลย RDFS จะไม่รายงานว่าเป็นข้อมูลผิด การอนุมานตอบคำถามว่า “จากข้อความที่มี สรุปอะไรเพิ่มได้” ส่วนการตรวจความครบถ้วนตอบคำถามว่า “ข้อมูลที่ระบบงานต้องการมีครบหรือไม่” ซึ่งเป็นคนละหน้าที่กัน และจะนำไปสู่ SHACL ในบทหลัง

สำหรับข้อมูลที่มาจากหลายหน่วยงาน ควรเลือกข้อมูลตัวอย่างจากอย่างน้อยสองแหล่งมาทดลองรวมตั้งแต่ระยะแรก ไม่ควรรอให้ Mapping ทุกแหล่งเสร็จแล้วจึงค่อยตรวจ ตัวอย่างเช่นให้สองแหล่งข้อมูลสร้างข้อมูลของโครงการและองค์กรเดียวกัน แล้วตรวจว่า IRI เชื่อมถึงกันจริงหรือเกิด Resource ซ้ำ จากนั้นลองเขียน Query ที่ครอบคลุมความสัมพันธ์หลัก เพื่อดูว่าเส้นทางจากโครงการไปองค์กรและเทคโนโลยีต่อกันได้หรือไม่ การทดสอบด้วยข้อมูลตัวอย่างที่ตรวจผลได้มักเปิดเผยปัญหา Namespace และ Identity ได้เร็วกว่าการอ่านเอกสาร Mapping เพียงอย่างเดียว

ภาษาและ Datatype ก็ควรถูกตรวจในรอบเดียวกัน หากชื่อภาษาไทยไม่มี language tag ระบบอาจแยกไม่ได้ว่าควรแสดง Label ใดให้ผู้ใช้ หากจำนวนงบประมาณถูกเก็บเป็นข้อความ การกรองและคำนวณจะผิดจากที่คาด หากวันที่ใช้รูปแบบไม่สม่ำเสมอ การเรียงตามเวลาก็จะไม่แน่นอน ปัญหาเหล่านี้ดูเหมือนรายละเอียดของข้อมูล แต่สะท้อนหลักเดียวกันว่า Object ของ Triple ต้องถูกเลือกให้เหมาะกับสิ่งที่ระบบจะทำต่อ ไม่ใช่เลือกเพียงให้เขียนได้สะดวกในวันนี้

เมื่อ Vocabulary เริ่มคงที่ ทีมควรมี canonical Graph ที่ครอบคลุมกรณีหลักและใช้ร่วมกัน ตัวอย่างนี้ควรมี Instance ที่สำคัญ Class และ Property หลัก Label ภาษาไทยและอังกฤษ รวมทั้งกรณีที่แสดง Inference ที่ตั้งใจไว้ ตัวอย่างดังกล่าวทำหน้าที่เป็นภาษากลางระหว่างผู้เชี่ยวชาญโดเมน ผู้ทำ Mapping และนักพัฒนา คนในทีมสามารถชี้ไปที่ Triple เดียวกันแล้วอภิปรายได้ว่าความหมายตรงหรือไม่ โดยไม่ต้องเริ่มจากเอกสารมาตรฐานทั้งฉบับ

เมื่อพิจารณาตัวอย่างทั้งหมดในบทนี้ร่วมกัน คุณภาพของ RDF/RDFS ไม่ได้วัดจากจำนวน Triple หรือจำนวน Class แต่ดูว่ากราฟรักษาตัวตนของสิ่งสำคัญ ใช้คำศัพท์สม่ำเสมอ และรองรับคำถามที่ต้องการได้หรือไม่ จุดที่มีผลต่อความชัดเจนอย่างมากคือ Predicate เพราะ Predicate เป็นส่วนที่บอกว่า Resource

สองรายการเกี่ยวข้องกันอย่างไร เนื้อหาส่วนถัดไปจึงพิจารณาหลักการตั้งชื่อและอธิบาย Predicate ให้ทั้งคนและระบบอ่านได้ตรงกัน

2.9 การตั้งชื่อ Predicate ให้คนและระบบเข้าใจตรงกัน

หลังจากพิจารณาการรวมข้อมูลแล้ว จุดที่กำหนดว่าการเชื่อมนั้นมีความหมายเพียงใดคือ Predicate เพราะ Predicate เป็นส่วนที่อธิบายบทบาทของเส้นเชื่อม หัวข้อนี้จึงพิจารณาการตั้งชื่อและเขียนคำอธิบายให้ทั้งผู้เชี่ยวชาญโดเมนและระบบอ่านตรงกัน

Predicate เป็นส่วนที่ทำให้ Graph มีความหมาย เพราะเป็นชื่อของความสัมพันธ์บนเส้นเชื่อมระหว่าง Resource หาก Predicate กว้างเกินไป Graph จะเชื่อมกันจริงแต่ไม่บอกความหมายเพียงพอ เช่น **voc:relatedTo** อาจใช้เชื่อมได้ทุกอย่าง แต่เมื่อ Query ต้องการถามว่าโครงการใดใช้เทคโนโลยีใด Predicate นี้จะไม่ช่วยมากนัก ในทางกลับกัน Predicate ที่เฉพาะเจาะจง เช่น **voc:usesTechnology** หรือ **voc:responsibleOrg** ทำให้ทั้งคนและเครื่องเข้าใจ Pattern ของ Graph ได้ดีกว่า

การตั้งชื่อ Predicate ควรอ่านเป็นประโยคได้ เช่น **proj:P001 voc:usesTechnology tech:SensorNet** อ่านได้ว่า โครงการ P001 ใช้เทคโนโลยี SensorNet ถ้า Predicate อ่านแล้วไม่เป็นธรรมชาติ อาจแปลว่าความสัมพันธ์ยังไม่ชัด หรือกำลังพยายามยัดความหมายหลายอย่างไว้ใน Property เดียว เช่น **voc:projectInfo** ซึ่งไม่บอกว่าข้อมูลนั้นคือ owner, status, Report หรือ Outcome อีกหลักหนึ่งคือหลีกเลี่ยงการตั้ง Predicate ตามชื่อคอลัมน์ต้นทางโดยตรงเสมอไป ชื่อคอลัมน์ในระบบเดิมอาจเกิดจากข้อจำกัดของฐานข้อมูล ไม่ใช่ความหมายของโดเมน เช่นคอลัมน์ **owner** อาจหมายถึงหน่วยงานรับผิดชอบในระบบหนึ่ง แต่หมายถึงผู้สร้าง record ในอีกระบบหนึ่ง หากนำมาเป็น Predicate โดยไม่ทบทวน ความหมายจะปนกันใน Graph RDFS ช่วยได้ด้วย **rdfs:label** และ **rdfs:comment** สำหรับ Property เช่น

```
voc:responsibleOrg a rdf:Property ;
 rdfs:label "หน่วยงานรับผิดชอบ"@th ;
 rdfs:comment "เชื่อมโครงการกับองค์กรที่รับผิดชอบหลัก"@th ;
 rdfs:domain voc:Project ;
 rdfs:range voc:Organization .
```

comment ที่ดีควรบอกเจตนาและขอบเขตของ Property ไม่ใช่แปลชื่อซ้ำ เช่นถ้า Label คือ “หน่วยงานรับผิดชอบ” comment ไม่ควรเขียนเพียงว่า “เป็นหน่วยงานรับผิดชอบ” แต่ควรบอกว่าหมายถึงองค์กรที่รับผิดชอบหลัก ไม่รวมผู้ร่วมดำเนินงานหรือผู้สนับสนุน หากมีความสัมพันธ์อื่น เช่น **voc:partnerOrg** หรือ **voc:fundingOrg** ก็ควรแยก Property ให้ชัด เมื่อ Vocabulary มีหลาย Predicate ควรตรวจว่า Property เหล่านั้นทับซ้อนกันเกินไปหรือไม่ หาก **voc:hasOwner**, **voc:responsibleOrg** และ **voc:managedBy** ถูกใช้ปนกันโดยไม่มีนิยามต่างกันชัดเจน Query จะยากและข้อมูลจะไม่สม่ำเสมอ บางครั้งการลดจำนวน Predicate และนิยามให้ชัดดีกว่าการเพิ่มคำใหม่ทุกครั้งที่เจอคำในเอกสาร

ดังนั้นการออกแบบ RDFS Vocabulary จึงมีทั้งด้านภาษาและด้านวิศวกรรม ด้านภาษาคือคำต้องอ่านเข้าใจ ด้านวิศวกรรมคือคำต้องช่วยให้การแปลงและค้นข้อมูลทำงานได้ หาก Predicate หนึ่งไม่มีใครใช้และไม่มีเหตุผลว่าช่วยสื่อความหมายอะไร ก็ควรตั้งคำถามว่าจำเป็นต้องมีหรือไม่ เมื่อ Vocabulary เริ่มคงที่ควรทดสอบด้วยตัวอย่าง RDF ที่มีข้อมูลเท่าที่จำเป็นต่อคำถาม วิธีง่ายที่สุดคือเลือกคำถามหนึ่งข้อ แล้วเขียน Graph ที่มีข้อมูลพอให้ตอบคำถามนั้น จากนั้นตรวจว่า Predicate และ Class ที่ออกแบบทำให้เส้นทางใน

กราฟอ่านได้เป็นธรรมชาติหรือไม่ หากคำถามพื้นฐานต้องผ่านความสัมพันธ์ที่ซับซ้อนผิดปกติ นั่นอาจเป็นสัญญาณว่า Vocabulary ยังไม่ตรงกับวิธีคิดของโดเมน

การทดสอบเช่นนี้ยังช่วยให้ผู้เชี่ยวชาญโดเมนเข้ามาทบทวนได้ง่ายขึ้น เพราะไม่จำเป็นต้องอ่าน Vocabulary ทั้งไฟล์ แต่สามารถดูตัวอย่างประโยค RDF และคำถามที่ระบบตอบได้ เช่น “โครงการนี้ใช้เทคโนโลยีใด” หรือ “รายงานใดกล่าวถึงโครงการนี้” หากผู้เชี่ยวชาญอ่านตัวอย่างแล้วบอกว่าคำที่ใช้ไม่ตรงกับงานจริง ควรแก้ Vocabulary ตั้งแต่ตอนนี้ ดีกว่าปล่อยให้ข้อมูลจำนวนมากถูกแปลงด้วยคำศัพท์ที่ผิดแล้วค่อยแก้ในภายหลัง

ในด้านความทันสมัยของมาตรฐาน ตัวอย่างในหนังสือเล่มนี้ใช้ RDF 1.1 เป็นฐานหลัก เพราะเครื่องมือและคลังซอฟต์แวร์จำนวนมากยังรองรับแนวคิดนี้อย่างมั่นคง แต่ในปี ค.ศ. 2026 W3C ได้เผยแพร่ RDF 1.2 Concepts เป็น Candidate Recommendation Snapshot ซึ่งเพิ่มประเด็นสำคัญ เช่น Triple term, directional language-tagged string และกลไกประกาศ Version ของ RDF (World Wide Web Consortium, 2026) สำหรับผู้อ่านที่เริ่มทำระบบใหม่ ประเด็นเหล่านี้ควรถูกติดตาม โดยเฉพาะงานที่ต้องอธิบาย Claim เกี่ยวกับ Triple หรือ Metadata ของ statement อย่างละเอียด อย่างไรก็ตาม หลักพื้นฐานในบทนี้ยังคงใช้ได้ เพราะ RDF 1.2 ยังคงตั้งอยู่บนความคิดเรื่อง Graph ของ statement ที่มี Subject, Predicate และ Object เป็นแกนกลาง

เมื่อ Predicate มีนิยามชัดเจน RDF/RDFS จะทำหน้าที่เป็นฐานความหมายได้มั่นคงขึ้น แต่ยังไม่รองรับเงื่อนไขเชิงตรรกะหลายชนิดที่จำเป็นต่อ Ontology หัวข้อสรุปจึงเชื่อมฐานนี้ไปสู่ OWL ในบทถัดไป

2.10 สรุป

บทนี้เริ่มจากโครงสร้างภายในของ Triple แล้วขยายไปสู่ Identity, Vocabulary, Inference และการรวม Graph จากหลายแหล่ง ประเด็นเหล่านี้สามารถสรุปเป็นฐานความเข้าใจสองชั้น ได้แก่ RDF สำหรับโครงสร้างข้อมูล และ RDFS สำหรับความหมายพื้นฐานของคำศัพท์

RDF และ RDFS เป็นฐานที่ทำให้ Semantic Web ใช้งานได้จริงในระดับเริ่มต้น RDF ทำให้ข้อมูลถูกเขียนเป็น Triple ที่มี Subject, Predicate และ Object ส่วน RDFS ทำให้คำที่ใช้ใน Triple มีโครงสร้างพื้นฐาน เช่น Class, Property และ hierarchy หากกำหนด IRI และ Vocabulary ให้สอดคล้องกับข้อมูลและคำถาม ทั้งสองชั้นนี้จะเป็นฐานสำหรับบทถัดไป ซึ่งจะเข้าสู่ OWL และการใช้ตรรกะเพื่อกำหนดความหมายที่ละเอียดกว่า RDFS

ตัวอย่างที่ตั้งใจให้รันได้จากบทนี้และบทถัดไปถูกรวบรวมเป็น regression fixture ในโฟลเดอร์ **companion/** ของ Source Repository โดยมีสคริปต์ตรวจ Turtle, OWL, SPARQL และ SHACL แยกจากการ compile PDF การแยกสองขั้นนี้ช่วยให้ข้อผิดพลาดของ Code example ถูกตรวจด้วย Parser ไม่ใช่ตรวจจากรูปลักษณ์ในหน้าหนังสือเพียงอย่างเดียว

เอกสารอ้างอิง

Allemang, D., Hendler, J., and Gandon, F. (2020). *Semantic Web for the Working Ontologist: Effective Modeling for Linked Data, RDFS, and OWL*. ACM Books, 3 edition.

Antoniou, G., Groth, P., van Harmelen, F., and Hoekstra, R. (2012). *A Semantic Web Primer*. MIT Press, 3 edition.

World Wide Web Consortium (2012). R2RML: RDB to RDF mapping language. W3C Recommendation. Accessed July 9, 2026, from **https://www.w3.org/TR/r2rml/**.

World Wide Web Consortium (2014a). RDF 1.1 concepts and abstract syntax. W3C Recommendation. Accessed July 4, 2026, from **https://www.w3.org/TR/rdf11-concepts/**.

World Wide Web Consortium (2014b). RDF 1.1 semantics. W3C Recommendation. Accessed June 16, 2026, from **https://www.w3.org/TR/rdf11-mt/**.

World Wide Web Consortium (2014c). RDF 1.1 Turtle: Terse RDF triple language. W3C Recommendation. Accessed June 29, 2026, from **https://www.w3.org/TR/turtle/**.

World Wide Web Consortium (2014d). RDF schema 1.1. W3C Recommendation. Accessed July 10, 2026, from **https://www.w3.org/TR/rdf-schema/**.

World Wide Web Consortium (2026). RDF 1.2 concepts and abstract data model. W3C Candidate Recommendation Snapshot. Accessed October 18, 2026, from **https://www.w3.org/TR/2026/CR-rdf12-concepts-20260407/**.

3

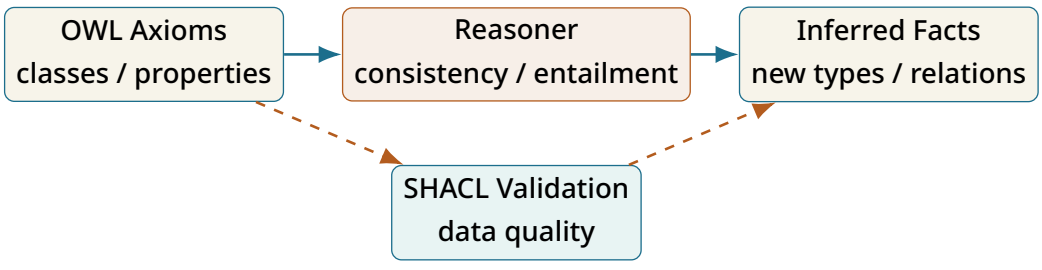
OWL และ Ontology ที่มีความหมายเชิงตรรกะ

เมื่อ RDF และ RDFS ทำให้มี Graph และคำศัพท์พื้นฐานแล้ว คำถามถัดไปคือจะระบุความหมายที่ลึกกว่า Subclass หรือ Domain/Range ได้อย่างไร บทนี้จึงเข้าสู่ OWL ในฐานะภาษาสำหรับเขียน Ontology ที่มีนัยทางตรรกะมากขึ้น โดยยังคงยึดหลักเดิมของเล่มนี้ คืออธิบายให้ผู้อ่านเห็นก่อนว่าแต่ละ Axiom ช่วยงานจริงอย่างไร แล้วจึงค่อยเชื่อมกับสมการเชิงตรรกะเท่าที่จำเป็น

3.1 จาก Vocabulary ไปสู่ Ontology

RDFS จากบทก่อนทำให้คำศัพท์มี Class Hierarchy, Domain และ Range แต่ยังไม่สามารถอธิบายเงื่อนไขเชิงตรรกะได้ครบ หัวข้อนี้จึงวางเส้นแบ่งระหว่าง Vocabulary กับ Ontology และชี้ให้เห็นว่า OWL เพิ่มความหมายชนิดใดจาก RDFS

RDFS ช่วยให้จัดระเบียบคำศัพท์พื้นฐานของ Graph ได้ แต่ในงานจริงหลายกรณีต้องการมากกว่านั้น เช่น ต้องการบอกว่า Class สองชุดไม่ควรทับกัน ต้องการบอกว่าความสัมพันธ์บางชนิดต้องมีอย่างน้อยหนึ่งค่า หรือต้องการให้ Reasoner ตรวจว่าคำอธิบายของ Ontology ขัดแย้งกันหรือไม่ OWL หรือ Web Ontology Language จึงถูกออกแบบขึ้นเพื่อเพิ่มพลังของการนิยามความหมายบนเว็บให้ละเอียดกว่า RDFS (World Wide Web Consortium, 2012e,i) ความสัมพันธ์ระหว่าง Axiom, Reasoner, Inferred Facts และการตรวจข้อมูลด้วย SHACL แสดงในภาพที่ 3.1



OWL ให้ความหมายและ reasoning ส่วน SHACL ตรวจเงื่อนไขข้อมูลใน workflow

ภาพที่ 3.1: OWL reasoning และ SHACL validation มีบทบาทต่างกันแต่ต้องทำงานร่วมกัน

ภาพที่ 3.1 แสดงว่า OWL Reasoning รับ Axiom ไปตรวจความสอดคล้องและอนุมานข้อเท็จจริง ส่วน SHACL Validation ตรวจข้อมูลตามเงื่อนไขของระบบงาน บทบาททั้งสองจึงสัมพันธ์กันแต่ไม่ใช่สิ่งเดียวกัน ในเชิงปฏิบัติ OWL ไม่ได้มีไว้เพื่อทำให้ Ontology ดูซับซ้อน แต่มีไว้เพื่อระบุความหมายที่จำเป็นต่อการใช้

เหตุผล ตัวอย่างเช่น หากกำหนดว่า **ex:Sensor** และ **ex:Organization** เป็น Class ที่ Disjoint กัน แล้วข้อมูลระบุว่า Resource เดียวกันเป็นทั้ง Sensor และ Organization ระบบ Reasoner จะชี้ให้เห็นความไม่สอดคล้องได้ การตรวจพบความขัดแย้งแบบนี้เป็นประโยชน์มากในงานที่ข้อมูลมาจากหลายแหล่งและมีโอกาสใช้คำไม่ตรงกัน

3.2 OWL, OWL 2 และ OWL DL ต่างกันอย่างไร

เมื่อเห็นหน้าที่ของ OWL ในภาพรวมแล้ว คำที่ต้องแยกให้ชัดก่อนอ่าน Axiom คือ OWL, OWL 2 และ OWL DL เพราะคำเหล่านี้ไม่ได้อยู่บนแกนเดียวกันทั้งหมด หัวข้อนี้จึงแยก “รุ่นของมาตรฐาน” ออกจาก “ภาษาย่อยที่จำกัดความสามารถ” และ “Syntax ที่ใช้เขียนเอกสาร”

มาตรฐานที่ W3C ประกาศในปี 2004 มักเรียกย้อนหลังว่า **OWL 1** ภายในมาตรฐานรุ่นนั้นมี OWL Lite, OWL DL และ OWL Full ส่วน **OWL 2** คือมาตรฐานรุ่นถัดมาที่ขยาย OWL 1 และรักษาความเข้ากันได้กับแนวคิดเดิมเป็นส่วนใหญ่ ดังนั้น OWL DL ไม่ใช่ Version ที่อยู่ระหว่าง OWL กับ OWL 2 แต่เป็นภาษาย่อยของ OWL 1 ที่อนุญาตให้ใช้ Construct ของ OWL ภายใต้ข้อจำกัดบางประการ เพื่อรักษา Computational Completeness และ Decidability ของงาน Reasoning (World Wide Web Consortium, 2004, 2012d)

ความสัมพันธ์ของชื่อที่พบบ่อยสรุปได้ดังตารางที่ 3.1

ตารางที่ 3.1: การแยกรุ่นมาตรฐานออกจากภาษาย่อยและ Profile ของ OWL

คำที่พบ	สถานะ	ความหมายที่ควรอ่าน
OWL หรือ OWL 1	มาตรฐานรุ่นปี 2004	ชื่อรวมของภาษาที่แบ่งเป็น OWL Lite, OWL DL และ OWL Full
OWL DL	ภาษาย่อยของ OWL 1	ใช้ Construct ของ OWL ภายใต้ข้อจำกัดที่ทำให้การอนุมานยังคง Complete และ Decidable
OWL Full	รูปแบบที่เปิดกว้างของ OWL 1	ผสม OWL กับความยืดหยุ่นของ RDF/RDFS ได้มากกว่า แต่ไม่ให้หลักประกันด้าน Reasoning แบบเดียวกับ OWL DL
OWL 2	มาตรฐานรุ่นถัดมา	ขยาย OWL 1 ด้วยความสามารถด้าน Property, Datatype, Key, Annotation และ Profile เพิ่มเติม
OWL 2 DL	ภาษาย่อยของ OWL 2	เป็น Subset ที่ผ่าน Typing Constraint และ Global Restriction เพื่อใช้ Direct Semantics และรักษา Decidability
OWL 2 EL/QL/RL	Profile ของ OWL 2	ลด Expressivity บางส่วนเพื่อให้เหมาะกับรูปแบบข้อมูลและงาน Reasoning คนละประเภท

ในเอกสาร OWL 2 คำว่า **OWL 2 DL** ใช้กับ Ontology ที่ผ่านข้อจำกัดเชิงโครงสร้างและอ่านด้วย Direct Semantics ส่วนคำว่า **OWL 2 Full** มักใช้โดยนัยกับ RDF Graph ที่อ่านด้วย RDF-Based Semantics ความหมายสองแบบนี้เชื่อมโยงกันผ่านการ Mapping ระหว่างโครงสร้าง OWL กับ RDF Graph แต่ไม่ควรสรุปว่า Semantics ทั้งสองเหมือนกันในทุกกรณี (World Wide Web Consortium, 2012b,a,h)

อีกเรื่องที่มักทำให้สับสนคือชื่อ Syntax เช่น Turtle, RDF/XML, OWL/XML, Functional-Style Syntax และ Manchester Syntax ชื่อเหล่านี้บอกวิธีเขียนหรือแลกเปลี่ยน Ontology ไม่ได้บอกว่า Ontology เป็น OWL Version ใดหรืออยู่ใน Profile ใด ไฟล์ Turtle หนึ่งไฟล์อาจแทน OWL 2 DL ได้ หาก Graph

ที่ได้เป็นไปตามข้อจำกัดของ OWL 2 DL ขณะเดียวกันไฟล์ RDF/XML ก็ไม่ได้เป็น OWL DL โดยอัตโนมัติเพียงเพราะใช้ Syntax ที่พบในเครื่องมือ Ontology

OWL 2 ไม่ได้ทิ้งแนวคิดของ OWL 1 แล้วสร้างภาษาใหม่ทั้งหมด แต่เพิ่มความสามารถที่จำเป็นต่อกรณีใช้งานมากขึ้น ตัวอย่างสำคัญ ได้แก่

- Property Chain, Reflexive, Irreflexive, Asymmetric และ Disjoint Property
- Qualified Cardinality ที่ระบุได้ทั้งจำนวนและ Class หรือ Data Range ของค่าปลายทาง
- Key และ Negative Property Assertion
- Datatype Restriction และการประกอบ Data Range ที่ละเอียดขึ้น
- Punning สำหรับ Meta-Modeling แบบจำกัดภายใต้ OWL 2 DL
- Annotation บน Axiom และ Profile แบบ OWL 2 EL, QL และ RL

รายการนี้มีหน้าที่ช่วยอ่านความต่างของรุ่น ไม่ใช่ Checklist ว่า Ontology ทุกชุดต้องใช้ความสามารถเหล่านี้ทั้งหมด การเลือก Construct ยังคงต้องย้อนกลับไปยังคำถามของระบบและผลอนุมานที่ต้องการ

3.3 องค์ประกอบและคำศัพท์มาตรฐานของ OWL 2

เมื่อแยกชื่อรุ่นและภาษาย่อยแล้ว ขั้นต่อไปคือมอง OWL 2 เป็นภาษาหนึ่งชุดก่อนเข้าสู่ตรรกะอย่างละเอียด Structural Specification จัดองค์ประกอบหลักเป็น Entity, Expression และ Axiom พร้อมส่วน Annotation, Import และ Ontology Metadata ส่วนตัวอย่างในหนังสือเล่มนี้เขียนด้วย Turtle จึงเห็นองค์ประกอบเหล่านั้นผ่าน RDF Triple และคำศัพท์ใน Namespace **rdf:**, **rdfs:** และ **owl:** (World Wide Web Consortium, 2012i,c,g)

คำว่า “Predicate ของ OWL” ควรใช้ด้วยความระมัดระวัง เพราะคำมาตรฐานทุกคำไม่ได้ทำหน้าที่เป็น Predicate ใน Triple ตัวอย่างเช่น **owl:Class**, **owl:ObjectProperty** และ **owl:Restriction** เป็น Resource ที่มักอยู่ตำแหน่ง Object ของ **rdf:type** ขณะที่ **owl:onProperty** และ **owl:someValuesFrom** จึงเป็น Predicate จริงใน RDF Serialization นอกจากนี้ Axiom บางชนิดใช้คำจาก RDF/RDFS เช่น **rdf:type**, **rdfs:subClassOf**, **rdfs:domain** และ **rdfs:range** ไม่ได้ขึ้นต้นด้วย **owl:**

คำศัพท์สำหรับประกาศ Entity และความสัมพันธ์พื้นฐานแสดงในตารางที่ 3.2

ตารางที่ 3.2: คำศัพท์พื้นฐานของ OWL 2 ใน RDF/Turtle

หมวด	คำมาตรฐาน	หน้าที่
Declaration	**owl:Class**, **owl:ObjectProperty**, **owl:DatatypeProperty**, **owl:AnnotationProperty**, **owl:NamedIndividual**	ใช้เป็น Object ของ **rdf:type** เพื่อประกาศบทบาทของ Entity
Class Axiom	**rdfs:subClassOf**, **owl:equivalentClass**, **owl:disjointWith**	ระบุ Subclass, Equivalence และ Disjointness ระหว่าง Class
Property Axiom	**rdfs:subPropertyOf**, **owl:equivalentProperty**, **owl:inverseOf**	อธิบายลำดับชั้น ความเท่ากัน และทิศกลับของ Property
Domain/Range	**rdfs:domain**, **rdfs:range**	ระบุความหมายของ Subject และ Object ที่ตามมาจากการใช้ Property
Built-in Class	**owl:Thing**, **owl:Nothing**	แทน Class ของ Individual ทั้งหมดและ Class ว่างตามลำดับ

Class Expression และ Restriction ใช้คำศัพท์อีกกลุ่มหนึ่งดังตารางที่ 3.3 คำในกลุ่มนี้มักปรากฏร่วมกันเป็นโครงสร้าง Blank Node มากกว่าจะใช้แยกเพียง Predicate เดียว

ตารางที่ 3.3: คำศัพท์สำหรับ Class Expression และ Restriction

หมวด	คำมาตรฐาน	หน้าที่
Restriction Core	**owl:Restriction**, **owl:onProperty**	ประกาศว่า Node เป็น Restriction และระบุ Property ที่ Restriction กล่าวถึง
Value Restriction	**owl:someValuesFrom**, **owl:allValuesFrom**, **owl:hasValue**, **owl:hasSelf**	กำหนดเงื่อนไขว่าค่าบางค่า ทุกค่า ค่าหนึ่งที่ระบุ หรือค่าที่สัมพันธ์กับตนเองต้องเป็นอย่างไร
Cardinality	**owl:minCardinality**, **owl:maxCardinality**, **owl:cardinality**	กำหนดจำนวนโดยยังไม่จำกัดชนิดของค่าที่ถูกนับ
Qualified Cardinality	**owl:minQualifiedCardinality**, **owl:maxQualifiedCardinality**, **owl:qualifiedCardinality**, **owl:onClass**, **owl:onDataRange**	กำหนดจำนวนพร้อมระบุ Class หรือ Data Range ของค่าที่นับ
Boolean Class Expression	**owl:intersectionOf**, **owl:unionOf**, **owl:complementOf**, **owl:oneOf**	สร้าง Class จาก Intersection, Union, Complement หรือ Enumeration

Property Characteristic, Identity และ Metadata ที่พบได้บ่อยสรุปไว้ในตารางที่ 3.4

ตารางที่ 3.4: คำศัพท์สำหรับ Property, Identity และ Ontology Metadata

หมวด	คำมาตรฐาน	หน้าที่
Property Characteristic	owl:FunctionalProperty, owl:InverseFunctionalProperty, owl:TransitiveProperty, owl:SymmetricProperty, owl:AsymmetricProperty, owl:ReflexiveProperty, owl:IrreflexiveProperty	ใช้เป็น Object ของ rdf:type เพื่อกำหนดลักษณะเชิงตรรกะของ Property
Property Composition	owl:propertyChainAxiom, owl:propertyDisjointWith	ระบุ Property Chain และ Property ที่ไม่ควรมีคู่ค่าเดียวกัน
Identity	owl:sameAs, owl:differentFrom, owl:AllDifferent, owl:hasKey	ระบุ Identity ความแตกต่างเป็นรายคู่หรือเป็นกลุ่ม และ Key ของ Class
Negative Assertion	owl:NegativePropertyAssertion, owl:sourceIndividual, owl:assertionProperty, owl:targetIndividual, owl:targetValue	ยืนยันอย่างชัดเจนว่า Object Property หรือ Data Property หนึ่งไม่มีค่าคู่ที่ระบุ
Ontology Header	owl:Ontology, owl:imports, owl:versionIRI	ประกาศ Ontology, Import Ontology อื่น และระบุ IRI ของ Version
Version Annotation	owl:versionInfo, owl:priorVersion, owl:backwardCompatibleWith, owl:incompatibleWith, owl:deprecated	บันทึกข้อมูลและความสัมพันธ์ระหว่าง Version โดยไม่เปลี่ยนตรรกะหลักของ Class
Axiom Annotation	owl:Axiom, owl:annotatedSource, owl:annotatedProperty, owl:annotatedTarget	ผูก Annotation กับ Axiom หรือ Triple ที่ต้องการอธิบาย โดย Annotation ไม่เปลี่ยนผลตรรกะภายใต้ Direct Semantics

ตัวอย่างต่อไปนี้แสดงคำศัพท์หลายหมวดในบริบทเดียวกัน บทนี้ใช้ Prefix **ex:** แทน Namespace **https://example.org/vocab/** ซึ่งเป็น Namespace เดียวกับ **voc:** ในบทก่อน ชื่อ Prefix เป็นเพียงชื่อย่อที่กำหนดในแต่ละเอกสาร

```
@prefix ex: <https://example.org/vocab/> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix owl: <http://www.w3.org/2002/07/owl#> .

ex:Project a owl:Class .
ex:Technology a owl:Class .

ex:usesTechnology a owl:ObjectProperty ;
 rdfs:domain ex:Project ;
 rdfs:range ex:Technology .

ex:DigitalProject rdfs:subClassOf ex:Project , [
 a owl:Restriction ;
 owl:onProperty ex:usesTechnology ;
 owl:someValuesFrom ex:Technology
] .
```

ในตัวอย่างนี้ **a** เป็นรูปย่อของ **rdf:type** ส่วน **ex:usesTechnology** เป็น Domain Property ที่ทำหน้าที่เป็น Predicate ในข้อมูลจริง แม้ไม่ได้อยู่ใน Namespace **owl:** การรู้ชื่อคำมาตรฐานจึงเป็นเพียงขั้นแรก สิ่งที่ต้องอ่านต่อคือคำเหล่านั้นประกอบกันเป็น Axiom ใดและทำให้ Reasoner สรุปอะไรได้ ซึ่งเป็นหน้าที่ของหัวข้อถัดไป

3.4 First-Order Logic และฐานตรรกะของ OWL

เมื่อเห็นภาพแล้วว่า OWL เพิ่ม Axiom ให้แก่ Vocabulary ขั้นต่อไปคือทำความเข้าใจว่าความหมายของ Axiom เหล่านั้นตั้งอยู่บนฐานตรรกะแบบใด หากข้ามพื้นฐานนี้ OWL อาจดูเหมือนรายการคำศัพท์และ Syntax ที่ต้องจำ ทั้งที่สาระสำคัญอยู่ที่เงื่อนไขว่า Interpretation แบบใดทำให้ Axiom เป็นจริง Section นี้จึงอธิบาย First-Order Logic เท่าที่จำเป็นต่อการอ่าน Description Logic และ OWL โดยไม่มุ่งพัฒนาการพิสูจน์เชิงตรรกะเต็มรูปแบบ

First-Order Logic หรือ FOL เป็นภาษาสำหรับกล่าวถึงสิ่งต่าง ๆ ในโดเมน คุณสมบัติของสิ่งเหล่านั้น และความสัมพันธ์ระหว่างสิ่งสองสิ่งหรือมากกว่า OWL 2 DL ไม่ได้อนุญาตให้เขียน FOL ได้ทุกประโยคอย่างอิสระ แต่ใช้ Description Logic ซึ่งจำกัดรูปประโยคบางส่วนเพื่อให้ Reasoner ตัดสินปัญหาสำคัญได้ ภาพรวมที่เหมาะสำหรับบทนี้จึงเป็น

$$\text{First-Order Logic} \supset \text{Description Logic} \longrightarrow \text{OWL 2 DL}$$

เครื่องหมาย $\supset$ ในบรรทัดนี้ใช้สื่อความสัมพันธ์เชิงขอบเขตอย่างไม่เป็นทางการว่า Description Logic ใช้แนวคิดจาก FOL แต่จำกัดรูปแบบของประโยค ส่วนลูกศรด้านขวาสื่อว่า OWL 2 DL นำ Description Logic มาใช้ผ่าน Syntax และ Semantics ของมาตรฐาน ไม่ได้หมายความว่าทุกภาษา Description Logic จะเท่ากับ OWL 2 DL โดยตรง (Baader et al., 2003; Hitzler et al., 2009; World Wide Web Consortium, 2012a)

องค์ประกอบพื้นฐานของ FOL เริ่มจากกำหนดโดเมนของการพิจารณาเป็นเซต Δ ของสิ่งทั้งหมดที่กำลังกล่าวถึง เช่น Sensor, Observation, Project, Organization และ Report จากนั้นจึงใช้สัญลักษณ์หลายชนิดเพื่อสร้างประโยค ตารางที่ 3.5 สรุปองค์ประกอบที่ต้องใช้ต่อในบทนี้

ตารางที่ 3.5: องค์ประกอบพื้นฐานของ First-Order Logic

องค์ประกอบ	รูปทั่วไป	ความหมายในตัวอย่าง Ontology
Constant	a, b, p_1	ชื่อที่ชี้ Individual หนึ่งตัว เช่น Project P001 หรือ Sensor S1
Variable	x, y	ตัวแทนของ Individual ใด ๆ ในโดเมน ไม่ใช่ชื่อ Resource จริง
Unary Predicate	$C(x)$	การเป็นสมาชิกของ Class เช่น *Project*(x)
Binary Predicate	$R(x, y)$	ความสัมพันธ์ระหว่าง Individual สองตัว เช่น *usesTechnology*(x, y)
Equality	$a = b$	Constant สองชื่ออ้างถึง Individual เดียวกัน
Connective	$\neg, \wedge, \vee, \rightarrow, \leftrightarrow$	ไม่, และ, หรือ, ถ้าแล้ว และก็ต่อเมื่อ
Universal Quantifier	$\forall x$	สำหรับทุก x ในโดเมน
Existential Quantifier	$\exists x$	มี x อย่างน้อยหนึ่งตัวในโดเมน

FOL ทั่วไปอาจมี Function Symbol สำหรับสร้าง Term ใหม่ด้วย แต่แบบจำลอง RDF/OWL ที่ใช้ในหนังสือเล่มนี้เน้น Constant, Predicate และ Quantifier เป็นหลัก IRI เช่น **proj:P001** ทำหน้าที่เป็นชื่อของ Resource ไม่ใช่ Function ที่คำนวณค่าจาก Input การแยกบทบาทนี้ช่วยไม่ให้สับสนระหว่างสัญลักษณ์เชิงตรรกะกับ Code ของ Application

เมื่อรู้บทบาทของ Constant, Variable และ Predicate แล้ว ขั้นต่อไปคืออ่าน Formula พื้นฐานสองแบบ หาก C เป็น Unary Predicate ที่แทน Class และ a เป็น Constant ที่แทน Individual สามารถเขียนว่า a เป็นสมาชิกของ C ได้เป็น

$$C(a)$$

และเขียนว่า a มีความสัมพันธ์ R กับ b ได้เป็น

$$R(a, b)$$

Formula สองแบบนี้เป็น Atomic Formula เพราะยังไม่มี Connective หรือ Quantifier เชื่อมอยู่ หาก p_1 แทน Project P001 และ t_1 แทน SensorNet สามารถเขียนข้อเท็จจริงพื้นฐานได้ว่า

$$\textit{Project}(p_1) \qquad \textit{Technology}(t_1) \qquad \textit{usesTechnology}(p_1, t_1)$$

เมื่อนำ Atomic Formula มาเชื่อมด้วย $\wedge$, $\vee$, $\neg$ หรือ $\rightarrow$ จะได้ Formula ที่ซับซ้อนขึ้น ตัวอย่างเช่น

$$\textit{Project}(p_1) \wedge \textit{usesTechnology}(p_1, t_1)$$

อ่านว่า P001 เป็น Project และ P001 ใช้ SensorNet ส่วน Formula ต่อไปนี้ยังมี x เป็น Free Variable เพราะไม่มี Quantifier ครอบอยู่

$$\mathit{Project}(x) \rightarrow \mathit{usesTechnology}(x, t_1)$$

หากต้องการกล่าวถึง Project ทุกตัว ต้องเขียนเป็น

$$\forall x(\mathit{Project}(x) \rightarrow \mathit{usesTechnology}(x, t_1))$$

ประโยคนี้มีความหมายแรงมาก เพราะประกาศว่า Project ทุกตัวใช้ SensorNet จึงไม่ควรเขียนเพียงเพราะพบ SensorNet ในข้อมูลตัวอย่างหนึ่งรายการ การเห็นขอบเขตของ Quantifier ช่วยป้องกันการยกระดับข้อเท็จจริงเฉพาะกรณีให้กลายเป็นกฎทั่วไปโดยไม่ตั้งใจ

สัญลักษณ์สองบรรทัดนี้ยังเป็นเพียงรูปทั่วไปเท่านั้น กล่าวคือ C แทน Class ใด ๆ, R แทน Relation ใด ๆ และ a, b แทน Individual สองตัว เพื่อให้เห็นภาพมากขึ้น สมมติว่ากำลังพูดถึงโดเมน Sensor ซึ่งเป็นโดเมนที่มักถูกใช้เป็นตัวอย่างใน Semantic Web เพราะมีทั้งอุปกรณ์ การสังเกต และเหตุการณ์ หากให้ C คือ Class **ex:Sensor** และให้ a คือ **sensor:S1** ประโยค $C(a)$ ก็อ่านได้ว่า Sensor S1 เป็นสมาชิกของ Class Sensor ส่วนถ้าให้ R คือ Property **ex:hasObservation** และให้ b คือ **obs:O1** ประโยค $R(a, b)$ ก็อ่านได้ว่า Sensor S1 มี Observation O1 เมื่อเข้าใจพื้นฐานนี้แล้ว จะอ่าน OWL Axiom ได้ต่อเนื่องขึ้น ตัวอย่างเช่น ถ้ากำหนดว่า **ex:SmartSensor** เป็น Subclass ของ **ex:Sensor** ข้อความใน RDF/OWL อาจเขียนได้ว่า **ex:SmartSensor rdfs:subClassOf ex:Sensor** และสามารถอ่านเป็นตรรกะอย่างง่ายได้ว่า

$$\forall x\ (\mathit{SmartSensor}(x) \rightarrow \mathit{Sensor}(x))$$

สมการนี้หมายความว่า ถ้า x ใดเป็น SmartSensor แล้ว x นั้นย่อมเป็น Sensor ด้วย ประโยคนี้ไม่ใช่การพูดถึง Sensor ตัวใดตัวหนึ่ง แต่เป็นการกำหนดความสัมพันธ์ระหว่าง Class ทั้งสองใน Ontology จากนั้นถ้าเพิ่ม Property **ex:hasObservation** เพื่อเชื่อม Sensor ไปยัง Observation อาจเขียน Restriction แบบ Existential อย่างง่ายได้ว่า Sensor หนึ่งมี Observation อย่างน้อยหนึ่งรายการ หรืออ่านเป็นตรรกะได้ว่า

$$\forall x(\mathit{Sensor}(x) \rightarrow \exists y\,(\mathit{hasObservation}(x, y) \wedge \mathit{Observation}(y)))$$

ในสมการนี้ x แทน Individual ใด ๆ ที่เป็น Sensor และ y แทน Individual อย่างน้อยหนึ่งตัวที่ต้องเป็นสมาชิกของ Class Observation ด้วย เงื่อนไข $\mathit{hasObservation}(x, y)$ บอกว่าทั้งสอง Individual เชื่อมกันด้วยความสัมพันธ์ดังกล่าว ส่วน $\mathit{Observation}(y)$ ระบุชนิดของค่าปลายทางให้ตรงกับความหมายของ **someValuesFrom Observation** เครื่องหมาย $\forall x$ ทำให้ข้อความครอบคลุม Sensor ทุกตัว ไม่ใช่เฉพาะตัวอย่างหนึ่งตัว และ $\exists y$ ระบุว่าต้องมี Observation อย่างน้อยหนึ่งตัวโดยไม่ได้กำหนดชื่อของ Observation นั้น สมการนี้เป็นการถอดความ Axiom เฉพาะกรณีให้เห็นองค์ประกอบของตรรกะ ไม่ใช่คำอธิบาย Formal Semantics ทั้งระบบของ OWL แต่เพียงพอสำหรับเห็นว่า OWL ไม่ได้เป็นเพียงการวาด Class Diagram หรือจัดหมวดหมู่คำเท่านั้น (Horrocks et al., 2003; World Wide Web Consortium, 2012a)

หากแยกส่วนของสมการนี้อีกเล็กน้อย เครื่องหมาย $\forall$ อ่านว่า “สำหรับทุกสิ่ง” และใช้เมื่อต้องการบอกกฎที่ครอบคลุมสมาชิกทุกตัวของ Class ส่วนเครื่องหมาย $\exists$ อ่านว่า “มีอย่างน้อยหนึ่งสิ่ง” และใช้เมื่อต้องการบอกว่ามีความสัมพันธ์บางอย่างเกิดขึ้นอย่างน้อยหนึ่งครั้ง ตัวแปร x และ y ไม่ได้เป็นชื่อของ Resource จริงใน Graph แต่เป็นตัวแทนของ Individual ใด ๆ ที่อาจอยู่ใน Graph ได้ เมื่อเขียน $\mathit{Sensor}(x)$ จึงไม่ได้พูดถึง Sensor ตัวชื่อ x แต่พูดถึงสิ่งใดก็ตามที่เป็นสมาชิกของ Class Sensor

การแปลงประโยคธรรมชาติเป็น FOL ไม่ได้มีเป้าหมายเพื่อเปลี่ยนเอกสารทุกประโยคให้เป็นสมการ แต่ใช้ตรวจว่าผู้ออกแบบกำลังกล่าวถึง Individual ตัวใด Class ใด และใช้เงื่อนไขทั่วไปหรือเงื่อนไขการมีอยู่ ตารางที่ 3.6 แสดงรูปประโยคที่ควรอ่านได้ก่อนเข้าสู่ Restriction ของ OWL

ตารางที่ 3.6: ตัวอย่างการแปลงประโยคโดเมนเป็น First-Order Logic

ประโยคโดเมน	Formula อย่างย่อ
P001 เป็น Project	$\mathit{Project}(p_1)$
P001 ใช้ SensorNet	$\mathit{usesTechnology}(p_1, t_1)$
SmartSensor ทุกตัวเป็น Sensor	$\forall x(\mathit{SmartSensor}(x) \rightarrow \mathit{Sensor}(x))$
Project ทุกตัวใช้ Technology อย่างน้อยหนึ่งรายการ	$\forall x(\mathit{Project}(x) \rightarrow \exists y(\mathit{Technology}(y) \land \mathit{usesTechnology}(x, y)))$
ทุกสิ่งที่ P001 ใช้ผ่าน *usesTechnology* ต้องเป็น Technology	$\forall y(\mathit{usesTechnology}(p_1, y) \rightarrow \mathit{Technology}(y))$
ไม่มี Report ใดเป็น Technology	$\forall x(\mathit{Report}(x) \rightarrow \neg \mathit{Technology}(x))$

Existential Quantifier ในแถวที่สี่กำหนดว่าสำหรับ Project แต่ละตัว ต้องมี Individual อย่างน้อยหนึ่งตัวที่เป็น Technology และเชื่อมด้วย *usesTechnology* แต่ไม่ได้บอกว่า Technology นั้นชื่ออะไร และไม่ได้บังคับว่า Project ทุกตัวต้องใช้ Technology ตัวเดียวกัน การเขียน $\exists y$ จึงต่างจากการเพิ่ม IRI ที่ระบุชื่อไว้แล้ว

ส่วนแถวที่ห้าเป็น Universal Restriction ซึ่งกำหนดชนิดของค่าปลายทางทุกตัวที่พบผ่าน Relation แต่ไม่ได้ยืนยันว่าต้องมีค่าปลายทางอย่างน้อยหนึ่งตัว หากไม่มี Triple *usesTechnology* เลย Formula แบบ Universal อาจยังเป็นจริงได้ ความต่างระหว่าง “ค่าทุกตัวต้องเป็น Technology” กับ “ต้องมี Technology อย่างน้อยหนึ่งตัว” เป็นจุดที่ผู้อ่าน OWL มักสับสน และเป็นเหตุผลที่ **owl:allValuesFrom** ไม่ควรถูกอ่านแทน **owl:someValuesFrom**

Inference เกิดจากกฎและข้อเท็จจริงร่วมกัน สมมติให้ Ontology มี Axiom ว่า SmartSensor ทุกตัวเป็น Sensor และ Data Graph ระบุว่า S1 เป็น SmartSensor

$$\forall x(\mathit{SmartSensor}(x) \rightarrow \mathit{Sensor}(x))$$

$$\mathit{SmartSensor}(s_1)$$

เมื่อแทน x ด้วย s_1 จะได้ $\mathit{SmartSensor}(s_1) \rightarrow \mathit{Sensor}(s_1)$ และจากข้อเท็จจริงที่มีอยู่ Reasoner จึงสรุปได้ว่า

$$\mathit{Sensor}(s_1)$$

ผลลัพธ์นี้เป็น Entailment กล่าวคือไม่จำเป็นต้องมี Triple ที่เขียน $\mathit{Sensor}(s_1)$ ไว้โดยตรง แต่ข้อเท็จจริงดังกล่าวเป็นจริงในทุก Interpretation ที่ทำให้ Axiom และข้อมูลตั้งต้นเป็นจริง คำว่า Interpretation หมายถึงการกำหนดว่า Constant ชี้สิ่งใด Unary Predicate ครอบคลุมสมาชิกใด และ Binary Predicate

ครอบคลุมคู่ความสัมพันธ์ใด Formula ไม่ได้เป็นจริงเพราะรูปสัญลักษณ์เพียงอย่างเดียว แต่เป็นจริงเมื่อการตีความองค์ประกอบเหล่านี้สอดคล้องกับ Formula ทั้งชุด

สะพานจาก FOL ไปสู่ Description Logic และ OWL แสดงในตารางที่ 3.7 การเทียบนี้มีไว้ช่วยอ่านความหมาย ไม่ได้หมายความว่า Turtle Syntax เป็นการแสดง FOL แบบหนึ่งต่อหนึ่งทุกกรณี

ตารางที่ 3.7: สะพานเชื่อม First-Order Logic, Description Logic และ OWL

First-Order Logic	Description Logic	OWL ที่ใช้สื่อความหมาย
$C(a)$	$a : C$	Class Assertion หรือ Triple **a rdf:type C**
$R(a, b)$	$(a, b) : R$	Object Property Assertion
$\forall x(A(x) \rightarrow B(x))$	$A \sqsubseteq B$	**rdfs:subClassOf**
$\forall x(A(x) \rightarrow \exists y(R(x, y) \wedge B(y)))$	$A \sqsubseteq \exists R.B$	**owl:someValuesFrom**
$\forall x(A(x) \rightarrow \forall y(R(x, y) \rightarrow B(y)))$	$A \sqsubseteq \forall R.B$	**owl:allValuesFrom**
$\forall x(A(x) \rightarrow \neg B(x))$	$A \sqcap B \sqsubseteq \bot$	**owl:disjointWith**
$a = b$	$a = b$	**owl:sameAs**

Description Logic ใช้สัญลักษณ์ $A \sqsubseteq B$ เพื่อบอกว่า Class A เป็น Subclass ของ B, ใช้ $\exists R.B$ เพื่อบอกว่ามีความสัมพันธ์ R ไปยังสมาชิกของ B อย่างน้อยหนึ่งตัว และใช้ $\forall R.B$ เพื่อกำหนดว่าค่าทุกตัวที่เชื่อมผ่าน R ต้องอยู่ใน B รูปเหล่านี้ปรากฏใน OWL ผ่าน Class Expression และ Restriction ที่อธิบายต่อในบทนี้ (Horrocks et al., 2003; World Wide Web Consortium, 2012i,a)

เหตุที่ OWL 2 DL ไม่เปิดให้เขียน FOL ได้ทุก Formula คือความสามารถในการตัดสินผลของ Reasoner ต้องแลกกับข้อจำกัดของภาษา FOL ทั่วไปสามารถสร้างประโยคที่การตรวจ Satisfiability หรือ Entailment ไม่มีวิธีรับประกันว่าจะจบสำหรับทุก Input ขณะที่ Description Logic แต่ละตระกูลกำหนดรูป Constructor และเงื่อนไขที่ควบคุมความซับซ้อน OWL 2 DL จึงมีข้อจำกัดเชิงโครงสร้างต่อการใช้ Class, Property และ Individual บางกรณี การรู้ข้อจำกัดนี้ช่วยให้ไม่สรุปว่า OWL เป็นเพียง FOL ที่เขียนด้วย RDF Syntax

ก่อนอ่าน OWL Axiom ผู้อ่านสามารถใช้คำถามสั้น ๆ ต่อไปนี้ตรวจความหมายของ Formula ได้

- Formula กล่าวถึง Individual ที่ระบุชื่อ หรือกล่าวถึงทุก Individual ผ่าน $\forall$
- Formula ยืนยันว่าต้องมีบางสิ่งผ่าน $\exists$ หรือเพียงกำหนดชนิดของค่าที่อาจมี
- Negation ถูกระบุไว้อย่างชัดเจน หรือเป็นข้อสรุปจากการไม่มีข้อมูล
- Axiom ทำให้เกิด Entailment ใด และ Entailment นั้นเป็นสิ่งที่โดเมนยอมรับหรือไม่

ความแตกต่างระหว่างการพูดถึง Class กับการพูดถึง Individual ตรงนี้เป็นหัวใจของ OWL หากเขียนว่า **sensor:S1 a ex:Sensor** กำลังพูดถึง Individual ตัวหนึ่ง แต่ถ้าเขียนว่า **ex:SmartSensor rdfs:subClassOf ex:Sensor** กำลังพูดถึงความสัมพันธ์ระหว่าง Class สองชุด ไม่ได้พูดถึง Sensor ตัวใดตัวหนึ่งโดยตรง Reasoner จึงทำงานโดยใช้ Axiom ระดับ Class ไปช่วยตีความข้อมูลระดับ Individual เช่นถ้ารู้ว่า S1 เป็น SmartSensor ก็สามารถสรุปต่อได้ว่า S1 เป็น Sensor ด้วย

ในงานปฏิบัติ ไม่จำเป็นต้องเขียน Formula ทุกครั้งที่ออกแบบ Ontology แต่การเข้าใจโครงสร้างของ Formula ช่วยให้ไม่ใช้ OWL ผิดหน้าที่ ตัวอย่างเช่น Restriction แบบ ∃ บอกว่า Class หนึ่งเกี่ยวข้องกับ Property บางอย่างในเชิงความหมาย แต่ไม่ได้ให้ข้อความ Error เมื่อข้อมูลยังไม่ครบทันที ส่วนการตรวจว่า Record ใดขาดค่าใดเป็นงานของ Validation Layer ดังนั้น Logic Primer ในบทนี้มีหน้าที่ทำให้ผู้อ่านรู้ว่า Axiom พูดในระดับใด ผลของ Reasoner ควรถูกอ่านอย่างไร และเหตุใดการไม่มี Triple จึงยังไม่เท่ากับการยืนยันว่า Formula นั้นเป็นเท็จ

3.5 Open World, Restriction และ Consistency

ฐานตรรกะของ OWL ทำให้การไม่มีข้อมูลและการมีข้อมูลที่ขัดแย้งกันต้องถูกตีความต่างจากฐานข้อมูลทั่วไป หัวข้อนี้จึงเชื่อมจากสัญลักษณ์เชิงตรรกะไปสู่สามแนวคิดที่มีผลโดยตรงต่อการออกแบบ ได้แก่ Open World Assumption, Restriction และ Consistency

จุดที่ผู้อ่านใหม่มักสับสนคือ OWL ใช้หลัก Open World Assumption กล่าวคือถ้า Graph ไม่มีข้อมูลบางอย่าง ระบบจะไม่สรุปทันทีว่าสิ่งนั้นไม่มีอยู่จริง ตัวอย่างเช่น ถ้าโครงการหนึ่งยังไม่มี Triple ที่บอกว่าใช้เทคโนโลยีใด OWL จะไม่ถือว่าโครงการนั้นไม่ใช้เทคโนโลยี แต่จะถือว่ายังไม่มีข้อมูลที่บอกเรื่องนี้ ความคิดแบบนี้ต่างจากฐานข้อมูลเชิงธุรกิจจำนวนมากที่มักใช้ Closed World Assumption Restriction ใน OWL จึงควรถูกใช้ด้วยความเข้าใจ ไม่ใช่ใช้แทน Validation ทุกกรณี หากต้องการตรวจว่าข้อมูลกรอกครบหรือไม่ SHACL อาจเหมาะกว่า OWL เพราะ SHACL ถูกออกแบบเพื่อ Validation แบบปิดในระดับข้อมูล ส่วน OWL เหมาะกับการนิยามความหมายและตรวจ Consistency ในระดับแบบจำลอง (World Wide Web Consortium, 2017; Labra Gayo et al., 2017)

ตัวอย่างเช่น การกำหนดว่า **ex:Project** ควรมี **ex:hasOwner** อย่างน้อยหนึ่งค่าใน OWL ไม่ได้แปลว่าข้อมูล Project ที่ไม่มี Owner จะเป็น Error ในทันที หากต้องการ Error Report สำหรับ Workflow การกรอกข้อมูล ควรใช้ SHACL Shape ในบทหลัง ดังนั้นการใช้ OWL ที่ดีคือใช้เพื่อบอกความหมายเชิงตรรกะเท่าที่จำเป็น และไม่ใช้ OWL เป็นเครื่องมือ Validation แทนทุกอย่าง

ตัวอย่างนี้อาจดูขัดกับสัญชาตญาณของคนที่คุ้นกับฐานข้อมูล เพราะในฐานข้อมูลหาก Column ถูกกำหนดว่า Not Null แล้ว Record ที่ไม่มีค่าจะผิดทันที แต่ OWL ไม่ได้มอง Graph เป็นแบบฟอร์มที่ต้องกรอกครบ OWL มอง Graph เป็นความรู้บางส่วนของโลกที่อาจยังไม่สมบูรณ์ หากยังไม่มีข้อมูลเจ้าของโครงการ ระบบจึงไม่สรุปว่าผิด แต่สรุปเพียงว่ายังไม่มีข้อมูลใน Graph ที่ระบุเจ้าของนั้น ความคิดนี้ทำให้ OWL เหมาะกับเว็บและข้อมูลกระจายตัว เพราะโลกจริงมักมีข้อมูลที่ยังไม่ครบอยู่เสมอ

อย่างไรก็ตาม Open World ไม่ได้แปลว่าปล่อยข้อมูลขาดได้ทุกกรณี หากระบบงานต้องใช้ข้อมูล Owner เพื่ออนุมัติโครงการ การไม่มี Owner ย่อมเป็นปัญหา เพียงแต่ปัญหานี้ควรถูกจัดการในชั้น SHACL หรือ Workflow ของ Application ไม่ใช่คาดหวังให้ OWL รายงาน Error แบบเดียวกับฐานข้อมูล การแยกหน้าที่เช่นนี้ทำให้สถาปัตยกรรมชัดขึ้น OWL บอกความหมายของ Model ส่วน SHACL บอกความพร้อมของข้อมูลในบริบทการใช้งานหนึ่ง

3.6 มองกราฟเดียวกันผ่าน RDFS, OWL และ SHACL

เมื่อเข้าใจ Open World Assumption แล้ว ยังต้องแยกให้ได้ว่าคำถามหนึ่งควรตอบด้วย RDFS, OWL หรือ SHACL การมอง Graph ชุดเดียวกันผ่านสามมาตรฐานช่วยให้เห็นว่าการอนุมานความหมาย การตรวจความสอดคล้อง และการตรวจคุณภาพข้อมูลเป็นคนละหน้าที่กัน

จุดที่ทำให้ Semantic Web ใช้ยากในงานจริงไม่ใช่เพียง Syntax แต่คือการเข้าใจว่าเครื่องมือแต่ละชั้นอ่าน Graph คนละแบบ RDFS อ่าน Graph เพื่อเติมความหมายพื้นฐาน เช่น Type ที่ตามมาจาก Domain/Range หรือ Subclass OWL อ่าน Graph ภายใต้ Open World Semantics และใช้ Axiom เพื่ออนุมานหรือค้นความไม่สอดคล้อง ส่วน SHACL อ่าน Graph ในฐานะข้อมูลที่ต้องผ่านเงื่อนไขของระบบใน Workflow หนึ่ง ความต่างนี้ทำให้ Axiom เดียวกันไม่ควรถูกใช้แทนกันทั้งหมด ตารางที่ 3.8 สรุปความต่างของสามชั้นนี้ในมุมที่เกี่ยวกับคำถามหลักและสิ่งที่ต้องระวังในการใช้งานจริง

ตารางที่ 3.8: บทบาทที่ต่างกันของ RDFS, OWL และ SHACL

ชั้นความหมาย	คำถามหลัก	สิ่งที่ต้องระวัง
RDFS	คำศัพท์นี้บอก Type หรือ Hierarchy อะไรเพิ่มได้บ้าง	Domain/Range เป็นกฎ Infer ไม่ใช่ Validation Rule
OWL	Ontology นี้มีความหมายเชิงตรรกะสอดคล้องกันหรือไม่	การไม่มีข้อมูลไม่ใช่ความผิดทันทีภายใต้ Open World
SHACL	ข้อมูลชุดนี้ผ่านเงื่อนไขที่ระบบต้องการหรือไม่	Shape เป็น Contract ของ Workflow ไม่ใช่ความจริงทั้งหมดของโดเมน

ผลที่ได้จากแต่ละมาตรฐานอาจแตกต่างกันแม้พิจารณาข้อมูลชุดเดียวกัน ตัวอย่างเช่น หาก Triple หนึ่งเชื่อม **proj:A** ไปยัง **org:PSU** ด้วย Property **ex:usesTechnology** และกำหนดให้ Property นี้มี Range เป็น **ex:Technology** ระบบ RDFS อาจ Infer ว่า **org:PSU** เป็น Technology ส่วน OWL จะตรวจพบ Inconsistency ได้เมื่อ Ontology กำหนดเพิ่มเติมว่า Organization และ Technology เป็น Disjoint Class ขณะที่ SHACL สามารถรายงาน Validation Error ได้โดยตรง หาก Shape ระบุว่า Object ของ **ex:usesTechnology** ต้องเป็น Instance ของ **ex:Technology** แต่ข้อมูลไม่เป็นไปตามเงื่อนไขดังกล่าว

การออกแบบจึงควรพิจารณาหน้าที่ของแต่ละมาตรฐาน มากกว่าพิจารณาเพียงว่ามาตรฐานใดมีความสามารถมากที่สุด RDFS เหมาะสำหรับจัด Vocabulary และ Hierarchy พื้นฐาน OWL ใช้เมื่อแบบจำลองต้องรองรับ Reasoning หรือ Consistency Checking ส่วน SHACL ใช้เมื่อระบบต้องการ Validation Report สำหรับตรวจและแก้ไขข้อมูล การแบ่งหน้าที่ดังกล่าวเป็นองค์ประกอบหนึ่งของ Knowledge Graph ที่ตรวจสอบได้ เนื่องจากทำให้ทั้งความหมายและคุณภาพข้อมูลมี Artifact ที่พิจารณาแยกจากกันได้

ในงาน Ontology Engineering ควรเริ่มจาก RDFS Vocabulary ที่อ่านง่ายก่อน แล้วค่อยเพิ่ม OWL เมื่อมีเหตุผลชัดเจน เช่น ต้องการ Disjointness ต้องการ Equivalence ต้องการ Restriction หรือต้องการ Reasoner ตรวจความสอดคล้องของแบบจำลอง การเพิ่ม Axiom ทุกอย่างตั้งแต่ต้นมักทำให้ Ontology เข้าใจยากและดูแลยาก โดยเฉพาะในทีมที่มีทั้งนักวิจัย นักพัฒนา และผู้เชี่ยวชาญโดเมน Workflow ที่ใช้ได้จริงคือสร้าง Ontology ใน Protégé หรือเครื่องมือใกล้เคียง ตรวจชื่อ Class และ Property ให้คนอ่าน

เข้าใจก่อน จากนั้นเพิ่ม Axiom เฉพาะจุดที่จำเป็น แล้วรัน Reasoner เพื่อดูว่า Ontology ยัง Consistent หรือไม่ หาก Reasoner ให้ผลที่แปลก เช่น Class กลายเป็น Unsatisfiable นั่นคือสัญญาณว่ามีนิยามบางส่วนขัดแย้งกันและต้องกลับไปทบทวนความหมาย

3.7 Restriction จากโดเมนโครงการ

การเปรียบเทียบมาตรฐานในหัวข้อก่อนทำให้เห็นหน้าที่ของ OWL ในภาพรวม หัวข้อนี้จึงกลับมายังโดเมนโครงการและใช้ Restriction แสดงว่า Axiom หนึ่งรายการเพิ่มความหมายให้ Class อย่างไร โดยไม่เปลี่ยน Restriction ให้เป็นกฎตรวจความครบถ้วนของข้อมูล

เพื่อให้เห็นตัวอย่างที่เป็นรูปธรรม สมมติว่ามี Ontology ของโครงการที่มี Class สำคัญคือ **ex:Project**, **ex:Technology**, **ex:Risk** และ **ex:Organization** ในระดับ RDFS อาจบอกได้ว่า Property **ex:usesTechnology** มี Domain เป็น Project และ Range เป็น Technology แต่ถ้าต้องการบอกความหมายที่ลึกขึ้น เช่น โครงการดิจิทัลทุกโครงการควรใช้เทคโนโลยีอย่างน้อยหนึ่งชนิด อาจใช้ Restriction ของ OWL ได้

```
ex:DigitalProject rdfs:subClassOf ex:Project .
ex:DigitalProject rdfs:subClassOf [
 a owl:Restriction ;
 owl:onProperty ex:usesTechnology ;
 owl:someValuesFrom ex:Technology
] .
```

ข้อความนี้ไม่ได้หมายความว่า Data Record ทุกตัวต้องกรอก Technology แล้วจึงจะผ่าน Validation แต่หมายความว่าใน Model ดังกล่าว Digital Project เป็นชนิดของ Project ที่มีความสัมพันธ์กับ Technology อย่างน้อยหนึ่งชนิดภายใต้ Open World Semantics หากข้อมูลยังไม่มี Triple ดังกล่าว ระบบอาจตีความว่า “ยังไม่รู้” ไม่ใช่ตีความว่า Ontology ผิดทันที ในทางกลับกัน ถ้าต้องการกำหนดว่าข้อมูล Project ในระบบ Production ต้องมี Technology อย่างน้อยหนึ่งรายการ ควรใช้ SHACL Shape แยกต่างหาก บทเรียนสำคัญคือ OWL ใช้เพื่อบอกความหมายของโลกที่ Ontology จำลอง ส่วน SHACL ใช้เพื่อบอกเงื่อนไขของข้อมูลที่ระบบต้องการรับเข้า

3.8 Disjointness และการตรวจความขัดแย้ง

Restriction อธิบายเงื่อนไขการเป็นสมาชิกของ Class ส่วน Disjointness ใช้ระบุว่า Class ใดไม่ควรมีสมาชิกร่วมกัน การเปลี่ยนมาพิจารณา Disjointness จึงช่วยให้เห็นอีกด้านหนึ่งของ OWL คือการเปิดเผยความขัดแย้งที่เกิดจากข้อมูลหรือ Mapping หลายแหล่ง

หนึ่งใน Axiom ที่ใช้บ่อยและมีประโยชน์มากคือ Disjointness เช่น หากกำหนดว่า **ex:Organization** และ **ex:Technology** เป็น Class ที่แยกกันอย่างเด็ดขาด แล้วข้อมูลบางส่วนบอกว่า **org:PSU** เป็นทั้ง Organization และ Technology Reasoner จะตรวจพบความไม่สอดคล้องของ Ontology หรือข้อมูลได้

```
ex:Organization owl:disjointWith ex:Technology .
org:PSU a ex:Organization .
org:PSU a ex:Technology .
```

กรณีนี้มีประโยชน์ในระบบที่รวมข้อมูลจากหลายแหล่ง เพราะความผิดพลาดมักเกิดจาก Mapping หรือการตั้ง IRI ซ้ำ เช่น IRI หนึ่งถูกใช้แทนองค์กรใน Source หนึ่ง แต่ถูกใช้แทนระบบเทคโนโลยีในอีก Source หนึ่ง หากไม่มี Axiom ที่ช่วยตรวจ ความผิดพลาดนี้อาจหลุดเข้าไปใน Graph และส่งผลต่อ Query ภายหลัง อย่างไรก็ตาม Disjointness ต้องใช้ด้วยความระมัดระวัง เพราะโลกจริงมีกรณีที่ Category ทับซ้อนกันได้ เช่น หน่วยงานหนึ่งอาจทำหน้าที่เป็นผู้ให้บริการระบบด้วย หาก Ontology ประกาศว่า Class แยกจากกันโดยไม่ตรวจขอบเขตของโดเมน ข้อมูลที่ถูกต้องอาจถูกตีความว่า Inconsistency ดังนั้นการใช้ OWL ต้องมาจากความเข้าใจโดเมน ไม่ใช่ใช้เพื่อทำให้ Ontology ดูเป็นทางการเท่านั้น

3.9 Cardinality และข้อจำกัดด้านจำนวน

Disjointness ตรวจความขัดแย้งระหว่างประเภทของสิ่งที่กล่าวถึง แต่บางโดเมนยังต้องอธิบายจำนวนความสัมพันธ์ด้วย หัวข้อนี้จึงเพิ่ม Cardinality เข้ามา พร้อมแยกความหมายเชิง Ontology ออกจากข้อกำหนดกรอกข้อมูลของระบบงาน

Cardinality Restriction ช่วยบอกจำนวนของความสัมพันธ์ เช่น Project หนึ่งมี Owner ได้อย่างน้อยหนึ่งองค์กร หรือ Sensor หนึ่งมี Serial Number ได้ไม่เกินหนึ่งค่า ตัวอย่างเช่น

```
ex:Project rdfs:subClassOf [
 a owl:Restriction ;
 owl:onProperty ex:responsibleOrg ;
 owl:minCardinality "1"^^xsd:nonNegativeInteger
] .
```

ควรกำหนด Cardinality เมื่อมีหลักฐานจากโดเมนว่าจำนวนดังกล่าวเป็นข้อกำหนดจริง เพราะข้อจำกัดจำนวนใน Ontology อาจไม่ตรงกับสถานการณ์ของข้อมูล เช่น โครงการหนึ่งอาจมีหลายหน่วยงานร่วมรับผิดชอบ หรือ Sensor หนึ่งอาจเปลี่ยนหมายเลขในกระบวนการซ่อมบำรุง หากกำหนด Cardinality โดยยังไม่ตรวจกรณีเหล่านี้ Ontology จะปรับเปลี่ยนได้ยาก ในงานเชิงปฏิบัติจึงควรเริ่มจาก Axiom ที่อธิบายความหมายพื้นฐาน เช่น Subclass, Disjointness ที่จำเป็น และ Property Hierarchy แล้วเพิ่ม Cardinality หลังจากตรวจข้อมูลจริงและยืนยันข้อกำหนดกับผู้เชี่ยวชาญโดเมน วิธีนี้ลดความเสี่ยงที่ข้อจำกัดใน Ontology จะขัดกับการใช้งานจริง

3.10 Property Characteristic และ Identity

Restriction และ Cardinality อธิบายเงื่อนไขที่ระดับ Class แต่ OWL ยังสามารถกำหนดลักษณะของ Property และความเป็นสิ่งเดียวกันของ Individual ได้ หัวข้อนี้จึงขยายจากจำนวนความสัมพันธ์ไปสู่ผลอนุมานที่เกิดจาก Functional, Inverse, Transitive และ Symmetric Property รวมถึงข้อควรระวังของ **owl:sameAs**

คำว่า Functional Property หมายความว่า Subject หนึ่งมีค่าได้ไม่เกินหนึ่ง Object สำหรับ Property นั้น ไม่ได้หมายความว่า Object หนึ่งถูกใช้ได้โดย Subject เดียว ส่วน Inverse Functional Property พิจารณาในทิศกลับ คือ Object หนึ่งชี้กลับไปยัง Subject ได้ไม่เกินหนึ่งรายการ ตัวอย่างการประกาศเขียนได้ดังนี้

```
ex:hasPrimaryOwner a owl:ObjectProperty ,
        owl:FunctionalProperty .

ex:hasRegistrationCode a owl:DatatypeProperty ,
        owl:FunctionalProperty .

ex:hasNationalIdentifier a owl:ObjectProperty ,
        owl:InverseFunctionalProperty .
```

ภายใต้ Open World Semantics หาก Functional Property ของ Subject เดียวกันชี้ไปยัง Individual สองรายการ Reasoner อาจอนุมานว่า Individual ทั้งสองเป็นสิ่งเดียวกัน แทนที่จะรายงานข้อมูลซ้ำเป็น Error ทันที หาก Ontology ประกาศเพิ่มเติมว่า Individual ทั้งสอง **owl:differentFrom** กัน จึงจะเกิด Inconsistency ประเด็นนี้แสดงอีกครั้งว่า OWL ใช้ให้ความหมายและอนุมาน ไม่ได้ทำหน้าที่แทน Unique Constraint ของฐานข้อมูลหรือ SHACL

Property บางชนิดทำให้เกิด Triple ใหม่ตามรูปแบบของความสัมพันธ์ ตัวอย่างที่พบบ่อย ได้แก่

- **Transitive Property** หาก a สัมพันธ์กับ b และ b สัมพันธ์กับ c จะอนุมานว่า a สัมพันธ์กับ c เช่น Relation *part of* ในโดเมนที่นิยามให้ถ่ายทอดได้
- **Symmetric Property** หาก a สัมพันธ์กับ b จะอนุมานว่า b สัมพันธ์กับ a เช่น Relation การร่วมมือที่ตั้งใจให้อ่านได้สองทิศ
- **Inverse Property** กำหนดชื่อสองชื่อให้แทนความสัมพันธ์คนละทิศ เช่น *has part* กับ *part of*
- **Reflexive/Irreflexive Property** ระบุว่าทุก Individual ต้องสัมพันธ์กับตนเอง หรือห้ามสัมพันธ์กับตนเองภายใต้ขอบเขตของความหมายที่กำหนด

ตัวอย่างใน Turtle เขียนได้ดังนี้

```
ex:partOf a owl:ObjectProperty , owl:TransitiveProperty .
ex:collaboratesWith a owl:ObjectProperty , owl:SymmetricProperty .
ex:hasPart owl:inverseOf ex:partOf .

ex:hasEvidence owl:propertyChainAxiom
 ( ex:hasClaim ex:derivedFrom ) .
```

Property Chain บรรทัดสุดท้ายอ่านว่า หาก Resource หนึ่งมี Claim และ Claim นั้น Derived From Source ใด ระบบสามารถอนุมานว่า Resource แรกมี Evidence จาก Source นั้นได้ การใช้ Property Characteristic หรือ Property Chain ต้องยืนยันว่าผลอนุมานสอดคล้องกับโดเมนจริง เพราะคำที่ฟังดูคล้าย Transitive หรือ Symmetric ในภาษาทั่วไปอาจไม่ได้มีคุณสมบัตินั้นในทุกบริบท นอกจากนี้ OWL 2 DL ยังมี Global Restriction ต่อการใช้ Property ที่ซับซ้อนร่วมกับ Axiom บางชนิด จึงควรตรวจด้วยเครื่องมือที่รองรับ OWL 2 DL เมื่อเพิ่ม Chain หรือ Transitivity (World Wide Web Consortium, 2012i)

ด้าน Identity นั้น OWL ไม่มี Unique Name Assumption กล่าวคือ IRI สองชื่อไม่ได้หมายความว่าเป็น Individual คนละตัวโดยอัตโนมัติ หากยืนยันว่าเป็นสิ่งเดียวกันใช้ **owl:sameAs** และหากยืนยันว่าเป็นคนละสิ่งใช้ **owl:differentFrom** ตัวอย่างเช่น

```
@prefix legacy: <https://example.org/legacy/project/> .
@prefix proj: <https://example.org/resource/project/> .
@prefix ex: <https://example.org/vocab/> .
@prefix owl: <http://www.w3.org/2002/07/owl#> .

legacy:P001 owl:sameAs proj:P001 .
proj:P001 owl:differentFrom proj:P002 .
ex:Project owl:hasKey ( ex:projectCode ) .
```

owl:sameAs เป็นข้ออ้างด้าน Identity ที่แรง เพราะ Axiom และ Assertion ของทั้งสอง IRI สามารถถ่ายทอดถึงกันได้ จึงไม่ควรใช้เพียงเพราะ Label คล้ายกัน หากต้องการบอกว่าแนวคิดสองรายการใกล้เคียงหรือเกี่ยวของกัน แต่ยังไม่ยืนยันว่าเป็นสิ่งเดียวกัน ควรใช้ Relation ที่อ่อนกว่าและตรงกับวัตถุประสงค์ของ Alignment ส่วน **owl:hasKey** ช่วยระบุชุด Property ที่ใช้จำแนก Instance ของ Class ในความหมายของ OWL แต่ไม่ได้แทน Primary Key ของฐานข้อมูลทุกด้าน

เมื่อ Ontology ใช้ Property Characteristic, Property Chain, Identity และ Key มากขึ้น ผลอนุมานและข้อจำกัดเชิงโครงสร้างจะซับซ้อนตามไปด้วย ประเด็นนี้นำไปสู่การเลือก OWL 2 DL และ Profile ให้เหมาะกับระบบในหัวข้อถัดไป

3.11 OWL Profile และการเลือกความซับซ้อน

เมื่อ Ontology มี Restriction, Disjointness และ Cardinality มากขึ้น ความสามารถในการแสดงความหมายย่อมแลกมากับภาระของ Reasoning OWL Profile จึงเป็นกรอบสำหรับเลือกขอบเขตความซับซ้อนให้เหมาะกับชนิดข้อมูลและ Query ของระบบ

OWL 2 DL ไม่ใช่ Profile ที่เลือกเพื่อเร่งระบบ แต่เป็น Subset ของ OWL 2 ที่ต้องผ่าน Typing Constraint และ Global Restriction เพื่อรักษา Decidability ภายใต้ Direct Semantics ภายในขอบเขตดังกล่าว OWL 2 ยังนิยาม Profile สามแบบซึ่งลด Expressivity บางส่วนเพื่อให้กระบวนการ Reasoning เฉพาะลักษณะทำงานได้มีประสิทธิภาพขึ้น (World Wide Web Consortium, 2012f,i)

- **OWL 2 EL** เหมาะกับ Ontology ที่มี Class และ Property จำนวนมาก โดยเฉพาะงานที่เน้น Class Hierarchy และการจำแนกประเภทขนาดใหญ่
- **OWL 2 QL** ออกแบบให้ตอบ Query ผ่านข้อมูลปริมาณมากได้โดยอาศัยแนวคิด Query Rewriting และเหมาะกับระบบที่เชื่อมกับฐานข้อมูลเชิงสัมพันธ์
- **OWL 2 RL** รองรับการประมวลผลแบบ Rule-Based ซึ่งสามารถนำไปใช้กับ Rule Engine หรือการคำนวณบน RDF Triple ได้ค่อนข้างตรงไปตรงมา

Profile ทั้งสามไม่ได้เรียงจากง่ายไปยากและไม่ได้เป็น Subset ซ้อนกันทั้งหมด การเลือกจึงต้องพิจารณาโครงสร้าง Ontology, งาน Reasoning และเครื่องมือที่ใช้ ไม่ใช่เลือกจากชื่อ Profile เพียงอย่างเดียว สำหรับหนังสือเล่มนี้ จุดสำคัญไม่ใช่การจำข้อกำหนดทุกบรรทัด แต่คือการเข้าใจว่า Expressivity มีทั้งต้นทุนเชิงคำนวณและต้นทุนในการอธิบายผล ยิ่ง Ontology มี Axiom ซับซ้อน การตรวจหาข้อผิดพลาด และ Review ผล Reasoning ก็ยิ่งต้องมี Test Case ที่ชัดเจน (Krötzsch, 2012)

3.12 การอ่านผลของ Reasoner

การเลือก OWL Profile มีความหมายก็ต่อเมื่อสามารถอ่านผลที่ Reasoner คืนมาได้อย่างถูกต้อง หัวข้อนี้จึงเปลี่ยนจากการเขียน Axiom มาสู่การตีความ Inference, Unsatisfiable Class และ Inconsistency โดยไม่สรุปเกินสิ่งที่ Reasoner ตรวจจริง

เมื่อรัน Reasoner แล้วได้ผลลัพธ์ใหม่ เช่น Instance หนึ่งถูกจัดเป็น Class ใหม่ หรือ Class หนึ่งกลายเป็น Subclass ของอีก Class หนึ่ง ผู้พัฒนาไม่ควรรับผลนั้นโดยไม่ตรวจสอบ Reasoner ทำงานตาม Axiom ที่กำหนดไว้ หากผลลัพธ์แปลก แสดงว่า Axiom หรือข้อมูลบางส่วนอาจสื่อความหมายต่างจากเจตนาเดิม แนวทางที่ดีคือเก็บตัวอย่าง Inference สำคัญไว้เป็น Test Case เช่น หาก **ex:SmartSensor** เป็น Subclass ของ **ex:Sensor** ระบบควร Infer ว่า Smart Sensor ทุกตัวเป็น Sensor หากไม่เกิดผลนี้ Hierarchy อาจไม่ได้รับการประกาศอย่างถูกต้อง แต่หากเกิดผลมากเกินไป เช่น Organization ถูก Infer เป็น Technology ปัญหาอาจอยู่ที่ Domain/Range หรือ Mapping จุดนี้ทำให้ Ontology Engineering มีลักษณะใกล้กับ Software Engineering เพราะต้องมี Test, Review และ Version Control

การอ่านผล Reasoner ควรเริ่มจากคำถามที่ระบุผลคาดหวังชัดเจน ไม่ควรเปิด Ontology ทั้งหมดแล้วตรวจผลทุกชนิดพร้อมกัน เช่นสร้าง Graph ตัวอย่างที่มี Project หนึ่งตัว Technology หนึ่งตัว Organization หนึ่งตัว และ Claim หนึ่งตัว จากนั้นเพิ่ม Axiom ทีละชุดแล้วดูว่าผล Inference เปลี่ยนอย่างไร วิธีนี้ช่วยให้เห็นว่า Axiom ใดทำให้เกิดผลใด หากเพิ่ม Disjointness แล้วเกิด Inconsistency ก็จะรู้ว่าต้องตรวจ Resource ที่ถูก Type ซ้ำ หากเพิ่ม Restriction แล้วไม่ได้ผลตามคาด ก็จะรู้ว่าต้องตรวจ Property หรือ Class ที่เกี่ยวข้อง

ในงานวิจัย การรายงานผลของ OWL ไม่ควรหยุดที่คำว่า Ontology นี้ Consistent เท่านั้น แต่ควรบอกด้วยว่า Reasoner ตรวจอะไรบ้าง และมีตัวอย่าง Inference สำคัญใดที่สะท้อนความหมายของ Ontology เช่น Sensor เฉพาะประเภทถูกจัดเข้า Class Sensor ระดับบนได้ หรือ Claim บางชนิดถูกเชื่อมกับ Evidence ผ่าน Property ที่กำหนดไว้ การเขียนเช่นนี้ทำให้ผู้อ่านเห็นประโยชน์ของ OWL ในฐานะเครื่องมือความหมาย ไม่ใช่เพียงเครื่องหมายว่าระบบใช้เทคโนโลยี Semantic Web แล้ว

3.13 กรณีศึกษาแบบต่อเนื่อง: เพิ่ม OWL ตามความต้องการของระบบ

แนวคิดก่อนหน้าถูกอธิบายแยกเป็นส่วน ๆ เพื่อให้เห็นหน้าที่ของแต่ละ Axiom หัวข้อนี้นำแนวคิดเหล่านั้นกลับมารวมใน Graph โครงการ และเพิ่ม OWL เฉพาะเมื่อคำถามของระบบต้องการความหมายที่ RDFS ยังให้ไม่ได้

จากบทก่อน มี RDF Graph ของโครงการ องค์กร เทคโนโลยี สถานะ และรายงานแล้ว หากใช้เพียง RDFS สามารถบอกได้ว่าโครงการเป็น Class หนึ่ง เทคโนโลยีเป็น Class หนึ่ง และ Property บางตัวมี Domain/Range อย่างไร แต่เมื่อระบบเริ่มต้องการความหมายที่ละเอียดขึ้น RDFS อาจไม่พอ ตัวอย่างเช่น อาจต้องการบอกว่าโครงการดิจิทัลเป็นโครงการชนิดหนึ่งที่ต้องเกี่ยวข้องกับเทคโนโลยีอย่างน้อยหนึ่งอย่าง หรือองค์กรกับเทคโนโลยีไม่ควรถูกมองเป็นสิ่งชนิดเดียวกัน หรือสถานะของโครงการควรอยู่ในชุดสถานะที่กำหนดไว้ จุดนี้คือพื้นที่ของ OWL แต่การเพิ่ม OWL ควรทำอย่างระมัดระวัง ไม่ใช่ใส่ Axiom ให้มากที่สุดตั้งแต่แรก ตัวอย่างเริ่มจาก Ontology Header และ Class พื้นฐานดังนี้

```
<https://example.org/ontology/project> a owl:Ontology ;
 owl:versionIRI <https://example.org/ontology/project/1.0> ;
 owl:versionInfo "1.0" ;
 rdfs:label "Project ontology"@en .

ex:Project a owl:Class .
ex:DigitalTransformationProject a owl:Class ;
 rdfs:subClassOf ex:Project .

ex:Organization a owl:Class .
ex:Technology a owl:Class .
ex:Report a owl:Class .
```

owl:versionIRI ระบุ IRI ของ Ontology Version ในโครงสร้างของ OWL ขณะที่ **owl:versionInfo** เป็น Annotation สำหรับบอกข้อมูล Version ที่คนหรือเครื่องมืออ่านได้ ทั้งสองคำจึงเกี่ยวกับ Version เหมือนกันแต่ทำหน้าที่ไม่เหมือนกัน ในระดับนี้ส่วนตรรกะของ OWL ยังดูคล้าย RDFS มาก เพราะใช้ Class และ Subclass เป็นหลัก แต่เมื่อเพิ่ม Restriction ความหมายจะเริ่มละเอียดขึ้น เช่น

```
ex:DigitalTransformationProject rdfs:subClassOf [
 a owl:Restriction ;
 owl:onProperty ex:usesTechnology ;
 owl:someValuesFrom ex:Technology
] .
```

ประโยคนี้อ่านได้ว่า โครงการ Digital Transformation เป็นโครงการที่มีความสัมพันธ์ **ex:usesTechnology** ไปยัง Technology อย่างน้อยหนึ่งอย่าง แต่ต้องระวังอีกครั้งว่าใน Open World Semantics หากข้อมูลของโครงการหนึ่งยังไม่มี Technology ระบบไม่ได้สรุปทันทีว่าข้อมูลผิด ระบบอาจมองว่ายังไม่รู้ Technology ของโครงการนั้น ดังนั้น Axiom นี้เหมาะสำหรับนิยามความหมายของ Class มากกว่าการทำ Data-Entry Validation หากต้องการตรวจว่าข้อมูลทุกโครงการในระบบต้องกรอก Technology จริง ๆ ควรใช้ SHACL Shape แยกต่างหาก บทเรียนนี้สำคัญมาก เพราะผู้เริ่มต้นมักพยายามใช้ OWL ทำทุกอย่าง ตั้งแต่นิยามความหมาย ตรวจฟอร์ม ตรวจคุณภาพข้อมูล ไปจนถึงกำกับ Workflow ผลคือ Ontology ซับซ้อนขึ้น แต่ระบบก็ยังไม่ได้ Validation Report ที่คนอ่านเข้าใจอยู่ดี

อีกตัวอย่างหนึ่งคือ Disjointness หากองค์กรและเทคโนโลยีเป็นสิ่งที่แยกกันในโดเมนของระบบ อาจเขียนว่า

```
ex:Organization owl:disjointWith ex:Technology .
```

ถ้าข้อมูลบอกว่า **org:PSU** เป็นทั้ง Organization และ Technology Reasoner จะเห็นความขัดแย้งได้ การใช้ Axiom แบบนี้มีประโยชน์เมื่อระบบรวมข้อมูลจากหลายแหล่ง เพราะช่วยจับกรณีที่ IRI ถูกใช้ผิดประเภท แต่ Disjointness ก็ต้องเกิดจากความเข้าใจโดเมนจริง หากในโดเมนหนึ่งมีสิ่งที่เป็นทั้งหน่วยงานและผู้ให้บริการเทคโนโลยี การประกาศ Disjointness แบบกว้างเกินไปอาจทำให้ข้อมูลจริงกลายเป็น Inconsistency โดยไม่จำเป็น OWL ยังช่วยเรื่อง Equivalence ได้ เช่น ถ้ามี Ontology อีกชุดหนึ่งใช้คำว่า **ex:ICTProject** ในความหมายเดียวกับ **ex:DigitalTransformationProject** จริง ๆ อาจประกาศว่า

```
ex:DigitalTransformationProject owl:equivalentClass ex:ICTProject .
```

แต่คำว่า “จริง ๆ” ในประโยคนี้สำคัญมาก เพราะ Equivalence เป็น Axiom ที่แรง หากใช้เพียงเพราะชื่อคล้ายกัน ระบบจะรวมความหมายของ Class สองตัวเข้าหากัน และ Inference ที่เกิดตามมาอาจกว้างเกินไป ในงาน Alignment บทหลังจะกลับมาพูดเรื่องนี้อีกครั้งว่า Mapping ระหว่าง Ontology ควรแยก Equivalent, Subclass, Close Match และ Related ให้ชัด สำหรับ Property ก็เช่นเดียวกัน อาจประกาศ Property Hierarchy ได้ เช่น

```
ex:hasPrimaryTechnology rdfs:subPropertyOf ex:usesTechnology .
```

หากโครงการหนึ่งมี Primary Technology เป็น SensorNet ระบบสามารถ Infer ได้ว่าโครงการนั้นใช้ SensorNet ด้วย การใช้ Subproperty เช่นนี้ช่วยให้ Query ระดับกว้างยังทำงานได้ แม้ข้อมูลบางส่วนจะใช้ Property ที่เฉพาะกว่า แต่ต้องออกแบบลำดับชั้นของ Property อย่างมีเหตุผล ไม่ใช่ให้ Property ทุกตัวเป็น Subproperty ของ Property กว้างเพียงตัวเดียว เพราะจะทำให้ความหมายหายไป

ในเชิงการทำงานกับเครื่องมือ ผู้พัฒนาควรเพิ่ม OWL ทีละส่วนแล้วรัน Reasoner เป็นระยะ ไม่ควรรอให้ Ontology ใหญ่แล้วค่อยตรวจครั้งเดียว ขั้นตอนง่าย ๆ คือหนึ่ง สร้าง Class Hierarchy ก่อน สอง เพิ่ม Property Hierarchy สาม เพิ่ม Disjointness เฉพาะจุดที่มั่นใจ สี่ เพิ่ม Restriction เท่าที่จำเป็น ห้า รัน Reasoner แล้วตรวจ Class ที่ Unsatisfiable หรือ Inference ที่ไม่คาดคิด การทำแบบนี้ทำให้ปัญหาถูกจับได้เร็วและอธิบายให้ทีมฟังง่ายขึ้น ตัวอย่างการตรวจแบบง่ายคือสร้าง Resource ทดสอบไว้ชุดหนึ่ง

```
proj:P001 a ex:DigitalTransformationProject ;
 ex:usesTechnology tech:SensorNet .

tech:SensorNet a ex:Technology .
```

หลัง Reasoner ทำงาน ระบบควรเห็นว่า **proj:P001** เป็น **ex:Project** ด้วย เพราะ Class ของโครงการดิจิทัลถูกประกาศให้เป็น Subclass ของ **ex:Project** หากไม่เกิด Inference นี้ แปลว่า Hierarchy หรือ Reasoning Configuration อาจมีปัญหา ในทางกลับกัน หากระบบ Infer ว่า **org:PSU** เป็น Technology โดยไม่ตั้งใจ ต้องกลับไปตรวจว่า Property หรือ Mapping ส่วนใดทำให้เกิดผลนั้น

ในงานวิจัยและงานระบบจริง OWL จึงไม่ใช่สิ่งที่ใช้เพื่อทำให้ Ontology ดูเป็นทางการเท่านั้น แต่เป็นเครื่องมือสำหรับทำให้ความหมายบางส่วนถูกตรวจด้วยตรรกะได้ คุณภาพของ Ontology ไม่ได้วัดจากจำนวน Axiom แต่พิจารณาจากว่า Axiom เหล่านั้นช่วยตอบคำถาม ตรวจความขัดแย้ง หรืออธิบายความหมายที่จำเป็นได้หรือไม่ หาก Axiom หนึ่งยังไม่มีหน้าที่ดังกล่าว ก็ยังไม่จำเป็นต้องใส่ในรุ่นแรก

3.14 ข้อผิดพลาดที่พบบ่อยในการใช้ OWL

กรณีศึกษาแสดงวิธีเพิ่ม Axiom ตามความจำเป็น แต่อีกด้านหนึ่งของการออกแบบคือการรู้ว่า Axiom ใดไม่ควรถูกเพิ่มเพียงเพราะภาษา OWL รองรับ หัวข้อนี้รวบรวมข้อผิดพลาดที่ทำให้ Ontology กำหนดความหมายมากกว่าเจตนาของโดเมนหรือทำให้ Reasoning ซับซ้อนโดยไม่จำเป็น

ข้อผิดพลาดที่พบบ่อยมักไม่ได้เกิดจาก Syntax ของ OWL แต่เกิดจากการเลือก Axiom ที่กำหนดความหมายมากกว่าที่ข้อมูลและผู้เชี่ยวชาญโดเมนยืนยันได้

- **ใช้ owl:sameAs กว้างเกินไป** คำนี้หมายถึงสอง IRI อ้างถึงสิ่งเดียวกันจริงในเชิงตรรกะ จึงไม่ควรใช้กับสิ่งที่เพียงคล้ายหรือเกี่ยวข้องกัน เช่นองค์กรกับหน่วยงานย่อย หรือโครงการกับรายงานของโครงการ

- **ประกาศ Equivalence เพราะชื่อคล้ายกัน** **ex:DigitalProject** กับ **ex:ICTProject** อาจสัมพันธ์กันแต่ไม่จำเป็นต้องครอบคลุมสมาชิกชุดเดียวกัน ต้องตรวจนิยามและข้อมูลตัวอย่างก่อนเสมอ
- **กำหนด Cardinality ก่อนยืนยันข้อกำหนด** การกำหนดว่า Project มี Responsible Organization ได้เพียงหนึ่งเดียวจะผิดทันทีเมื่อข้อมูลจริงมีหลายหน่วยงานร่วมรับผิดชอบ ข้อจำกัดด้านจำนวนจึงต้องมีหลักฐานจากโดเมนรองรับ
- **อ่าน owl:allValuesFrom เป็นข้อบังคับว่าต้องมีค่า** Universal Restriction บอกว่าทุกค่าที่มีต้องอยู่ใน Class ที่กำหนด แต่ไม่ได้บอกว่าต้องมีค่าอย่างน้อยหนึ่งค่า ความหมายหลังเป็นหน้าที่ของ **someValuesFrom**
- **ใช้ OWL แทน Validation ทั้งหมด** ภายใต้ Open World Semantics การไม่มี Label หรือ Owner ไม่ได้ทำให้ข้อมูลผิดทันที หากต้องการตรวจความครบถ้วนของข้อมูลใน Workflow ควรเขียน SHACL Shape
- **ไม่ทดสอบ Reasoner กับข้อมูลตัวอย่าง** Axiom ที่ดูถูกต้องอาจทำให้ Class เป็น Unsatisfiable หรือจัด Instance เข้ากลุ่มที่ไม่ต้องการ การทดสอบ Graph ที่ครอบคลุมทั้งกรณีปกติและกรณีขัดแย้งช่วยให้เห็นผลก่อนนำ Axiom ไปใช้กับข้อมูลทั้งหมด
- **ไม่บันทึก Rationale ของ Axiom** ผู้ดูแลรุ่นถัดไปควรทราบว่าทำไม Class จึง Disjoint ทำไมต้องมี Restriction หรือทำไม Property หนึ่งเป็น Subproperty ของอีก Property มิฉะนั้นอาจแก้ Axiom ผิดทิศทาง

ในงานปฏิบัติ วิธีลดข้อผิดพลาดเหล่านี้คือเริ่มจาก OWL Profile และ Axiom ที่ทีมเข้าใจก่อน ใช้ Reasoner ตรวจเป็นระยะ แยก Validation ไปอยู่ใน SHACL และเขียนตัวอย่าง RDF ควบคู่กับ Ontology เสมอ หาก Ontology หนึ่งอธิบายด้วยตัวอย่างไม่ได้ แสดงว่า Ontology นั้นอาจยังไม่ชัดพอสำหรับผู้ใช้จริง

3.15 อ่าน Description Logic เพิ่มเติมจากตัวอย่าง

เมื่อทราบข้อผิดพลาดที่พบบ่อยแล้ว การกลับมาอ่าน Description Logic อีกครั้งจะช่วยอธิบายว่าเหตุใด Axiom บางชนิดจึงให้ผลกว้างกว่าที่คาด หัวข้อนี้จึงเพิ่มรายละเอียดเชิงสัญลักษณ์จากตัวอย่างเดิม โดยยังยึดการตัดสินใจออกแบบเป็นแกน

เพื่อเข้าใจ OWL ให้ลึกขึ้นอีกเล็กน้อย สามารถมอง OWL บางส่วนผ่าน Description Logic ได้ โดยให้ Concept คือ Class, Role คือ Property และ Individual คือ Resource เฉพาะตัว เช่น **ex:Project**, **ex:usesTechnology** และ **proj:P001** ตามลำดับ ในมุมนี้ Class ไม่ใช่เพียงกล่องใน Diagram แต่เป็นเซตของ Individual ที่มีคุณสมบัติตามนิยาม ถ้าเขียนว่า

$$\mathit{DigitalTransformationProject} \sqsubseteq \mathit{Project}$$

ความหมายคือทุก Individual ที่อยู่ใน Class *DigitalTransformationProject* ต้องอยู่ใน Class *Project* ด้วย ใน Description Logic ไม่ได้เขียนตัวแปร x ออกมาตรง ๆ ทุกครั้ง แต่สามารถอ่านในใจได้ว่า “สำหรับสิ่งใด ๆ หากสิ่งนั้นเป็น DigitalTransformationProject สิ่งนั้นก็เป็น Project” สัญลักษณ์ $\sqsubseteq$ จึงสอดคล้องกับความคิดเรื่อง Subclass ใน Ontology ส่วนถ้าเขียนว่า

$$Organization \sqcap Technology \sqsubseteq \bot$$

จะหมายถึง Organization และ Technology เป็น Class ที่ไม่ควรมีสมาชิกซ้ำกัน โดย $\sqcap$ คือการตัดกันของ Class และ $\bot$ คือ Class ว่าง หากอ่านแบบมีตัวแปรคือไม่มี Individual ใดที่เป็นทั้ง Organization และ Technology พร้อมกัน ใน OWL แนวคิดนี้มักเขียนด้วย **owl:disjointWith** Restriction สามารถมองในรูปตรรกะได้เช่นกัน เช่น

$$DigitalTransformationProject \sqsubseteq \exists usesTechnology.Technology$$

หมายความว่า Digital Transformation Project ทุกตัวมีความสัมพันธ์ *usesTechnology* ไปยัง Technology อย่างน้อยหนึ่งตัว ในสมการนี้ *usesTechnology* คือ Role หรือ Property และ *Technology* คือ Class ของ Object ปลายทาง สัญลักษณ์ $\exists$ คือ Existential Restriction ซึ่งอ่านว่า “มีอย่างน้อยหนึ่งค่า” และใกล้กับ **owl:someValuesFrom** ตัวอย่างนี้ช่วยให้เห็นว่า OWL ไม่ใช่เพียง Syntax ของ RDF แต่เป็นภาษาที่มีความหมายเชิงตรรกะรองรับ ในอีกด้านหนึ่ง

$$Project \sqsubseteq \forall usesTechnology.Technology$$

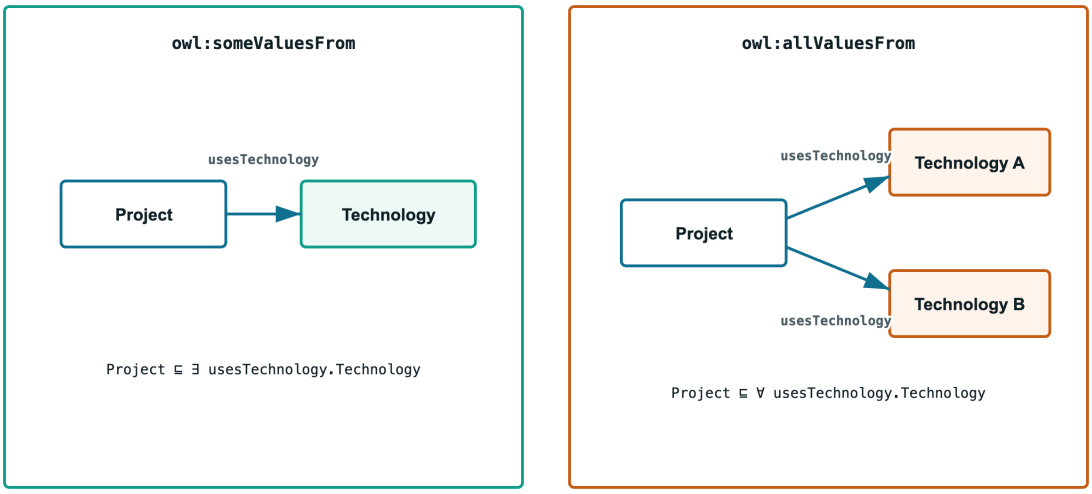
หมายความว่าหาก Project ใช้ *usesTechnology* ไปยังสิ่งใด สิ่งนั้นต้องเป็น Technology ทั้งหมด ในสมการนี้ $\forall$ อ่านว่า “ทุกค่า” และใกล้กับ **owl:allValuesFrom** ความต่างระหว่าง $\exists$ และ $\forall$ สำคัญมาก เพราะ $\exists$ บอกว่าต้องมีบางค่า ส่วน $\forall$ บอกว่าทุกค่าที่มีต้องอยู่ใน Class ที่กำหนด แต่ไม่ได้บังคับว่าต้องมีค่า หากนำมาสู่ตัวอย่าง Turtle จะเห็นความสัมพันธ์ดังนี้

```
ex:DigitalTransformationProject rdfs:subClassOf [
 a owl:Restriction ;
 owl:onProperty ex:usesTechnology ;
 owl:someValuesFrom ex:Technology
] .
```

และ

```
ex:Project rdfs:subClassOf [
 a owl:Restriction ;
 owl:onProperty ex:usesTechnology ;
 owl:allValuesFrom ex:Technology
] .
```

ความต่างระหว่าง Restriction สองแบบเมื่ออ่านผ่านโครงสร้างของ Graph แสดงในภาพที่ 3.2 โดยภาพเน้นสิ่งที่ Axiom แต่ละชนิดรับรองและสิ่งที่ไม่ได้รับรอง



ภาพที่ 3.2: ความต่างระหว่าง **owl:someValuesFrom** และ **owl:allValuesFrom**

ภาพที่ 3.2 แสดงว่า **someValuesFrom** กำหนดให้มีค่าปลายทางอย่างน้อยหนึ่งค่าที่อยู่ใน Class ที่ระบุ ส่วน **allValuesFrom** กำหนดชนิดของทุกค่าที่มีอยู่ แต่ไม่ได้บังคับว่าต้องมีค่า ความต่างนี้เป็นเหตุผลที่ไม่ควรใช้ Universal Restriction แทนกฎตรวจความครบถ้วนของข้อมูล

การเข้าใจตรรกะพื้นฐานนี้ช่วยให้ผู้เขียน Ontology เลือก Axiom ได้ตรงเจตนา ถ้าต้องการบอกว่าโครงการดิจิทัลเกี่ยวข้องกับเทคโนโลยีอย่างน้อยหนึ่งชนิด ใช้ **someValuesFrom** ถ้าต้องการบอกว่าค่าทุกค่าของ Property นี้ต้องเป็น Technology ใช้ **allValuesFrom** ถ้าต้องการตรวจว่าข้อมูลกรอกครบหรือไม่ ใช้ SHACL แทน ในเชิงการสื่อสารกับผู้เชี่ยวชาญโดเมน ไม่จำเป็นต้องแสดงสัญลักษณ์เหล่านี้ทั้งหมด แต่ Ontology Engineer ควรเข้าใจเบื้องหลัง เพราะสัญลักษณ์เหล่านี้เป็นเหตุผลว่าทำไม Reasoner จึงให้ผลลัพธ์บางอย่างออกมา เมื่อผล Reasoning แปลก ผู้เขียนจะย้อนกลับไปดู Axiom ได้ถูกจุด ไม่ใช่แก้ Ontology ด้วยการลองผิดลองถูก

3.16 OWL กับ SHACL ควรแบ่งหน้าที่อย่างไร

Description Logic ทำให้ความหมายของ OWL ชัดขึ้น แต่ระบบใช้งานจริงยังต้องตรวจความครบถ้วนและรูปแบบของข้อมูล หัวข้อนี้จึงกลับมาเปรียบเทียบ OWL กับ SHACL เพื่อกำหนดขอบเขตของ Ontology Axiom และ Validation Shape ให้ไม่ทำหน้าที่ทับซ้อนกัน

คำถามที่พบบ่อยคือถ้า OWL สามารถเขียน Restriction ได้ แล้วทำไมยังต้องใช้ SHACL คำตอบคือ OWL และ SHACL ตอบโจทย์คนละแบบ OWL ใช้เพื่ออธิบายความหมายของโลกที่ Ontology จำลอง ส่วน SHACL ใช้เพื่อตรวจว่าข้อมูลใน Graph หนึ่งตรงตามเงื่อนไขของระบบหรือไม่ ความต่างนี้อาจดูละเอียด แต่มีผลต่อการออกแบบอย่างมาก ตัวอย่างเช่น หากเขียน OWL ว่า Digital Transformation Project มี **usesTechnology** บางค่าเป็น Technology กำลังนิยามว่า Project ประเภทนี้เกี่ยวข้องกับ Technology ในเชิงความหมาย แต่ถ้า Graph ของ Project หนึ่งยังไม่มี Technology ระบบ OWL ไม่จำเป็นต้องรายงาน Error เพราะอาจตีความว่ายังไม่รู้เทคโนโลยีของ Project นั้น

หากต้องการให้ระบบรายงานว่า “โครงการนี้ยังไม่ได้กรอกเทคโนโลยี” นั่นเป็นหน้าที่ของ SHACL Shape เช่นกำหนด **sh:minCount 1** บน Property **ex:usesTechnology** ความหมายของ SHACL จึงใกล้กับ

Data Contract ของ Application มากกว่า Ontology Axiom ในเชิงตรรกะ อีกตัวอย่างคือ Range ของ Property ใน RDFS/OWL หากกำหนดว่า **ex:usesTechnology** มี Range เป็น **ex:Technology** แล้วใส่ Object เป็นองค์กร ระบบอาจ Infer ว่าองค์กรนั้นเป็น Technology แทนที่จะบอกว่าข้อมูลผิดทันที แต่ SHACL สามารถเขียนว่า Object ของ Property นี้ต้องเป็น Class **ex:Technology** และหากไม่ใช่ให้สร้าง Validation Report ได้

ในงานจริงจึงควรตั้งคำถามก่อนว่ากำลังเขียนกฎเพื่ออะไร หากต้องการให้ Reasoner เข้าใจความสัมพันธ์ของ Class ใช้ OWL หากต้องการให้ระบบรับข้อมูลหรือปฏิเสธข้อมูลตามเงื่อนไข ใช้ SHACL หากต้องการดึงคำตอบ ใช้ SPARQL และหากต้องการบอกที่มาของ Claim ใช้ Provenance การแยกหน้าที่เช่นนี้ทำให้ระบบไม่บวมและ Debug ง่ายขึ้น การใช้ OWL และ SHACL ร่วมกันให้ดีคือให้ OWL เป็นชั้นความหมายที่ค่อนข้างมั่นคง ส่วน SHACL เป็นชั้นข้อกำหนดของระบบที่อาจเปลี่ยนตาม Workflow ได้ เช่น Ontology อาจบอกว่า Project มี Relation กับ Organization ได้หลายแบบ แต่ Shape ของระบบรายงานประจำปีอาจบังคับว่า Project ที่ Publish ในรายงานต้องมี Responsible Organization อย่างน้อยหนึ่งรายการ

เมื่อทีมเข้าใจการแบ่งหน้าที่นี้ การสนทนาระหว่างนักวิจัยกับนักพัฒนาจะง่ายขึ้น นักวิจัยสามารถอธิบายว่า Axiom หนึ่งหมายถึงอะไรในเชิง Ontology นักพัฒนาสามารถอธิบายว่า Shape หนึ่งจำเป็นต่อ Workflow ใด และผู้เชี่ยวชาญโดเมนสามารถตรวจว่าเงื่อนไขเหล่านั้นตรงกับงานจริงหรือไม่

เมื่ออ่านทั้งบทต่อเนื่องกัน จะเห็นว่าการเพิ่ม OWL ควรเริ่มจากปัญหาที่ RDFS ตอบไม่ได้ ไม่ใช่เริ่มจากรายการคำสั่งของภาษา ตัวอย่างเช่น หากต้องการเพียงจัดลำดับว่า SmartSensor เป็น Sensor การใช้ **rdfs:subClassOf** ก็เพียงพอ แต่ถ้าต้องการบอกว่า Sensor และ Organization ไม่ควรมีสมาชิกตัวเดียวกัน หรือต้องการนิยาม Class จากความสัมพันธ์ที่สมาชิกต้องมี จึงมีเหตุผลที่จะใช้ Axiom ของ OWL การเลือกเช่นนี้ทำให้ Ontology อ่านง่ายและลดผลอนุมานที่ทีมไม่ได้ตั้งใจ

ในมุมของตรรกะ ทุก Axiom ควรถูกอ่านกลับเป็นประโยคภาษาธรรมชาติได้ก่อนนำไปใช้ หากเขียน $\forall x\,(\mathit{SmartSensor}(x) \rightarrow \mathit{Sensor}(x))$ ทีมควรอธิบายได้ว่า สำหรับทุกสิ่ง x หาก x เป็น SmartSensor แล้ว x ต้องเป็น Sensor สัญลักษณ์ $\forall$ หมายถึง “สำหรับทุกค่า” ตัวแปร x ใช้แทน Individual ใด ๆ ในโดเมน และลูกศร $\rightarrow$ บอกเงื่อนไขจากด้านซ้ายไปด้านขวา การอ่านทีละส่วนเช่นนี้ช่วยป้องกันการมองสมการเป็นเพียงของตกแต่งใน Ontology

หากมีความสัมพันธ์สองตำแหน่ง เช่น $\mathit{Observes}(x, y)$ ตัวแปร x และ y มีบทบาทต่างกัน โดย x แทนสิ่งที่ทำการสังเกตและ y แทนสิ่งที่ถูกสังเกต เมื่อเขียน $\mathit{Sensor}(x) \rightarrow \exists y\, \mathit{Observes}(x, y)$ เครื่องหมาย $\exists$ หมายถึง “มีอย่างน้อยหนึ่งค่า” ประโยคทั้งหมดจึงอ่านว่า Sensor ทุกตัวต้องมีบางสิ่งที่มันสังเกตอยู่ ความหมายนี้ต่างจากการกำหนด Shape ว่าข้อมูล Sensor แต่ละระเบียนต้องกรอก Property **observes** เพราะ OWL กล่าวถึงโลกที่แบบจำลองยอมรับ ส่วน SHACL ตรวจ Graph ที่ส่งเข้ามาในรอบนั้น

ความต่างดังกล่าวเห็นได้ชัดเมื่อข้อมูลไม่ครบ หาก Graph ระบุว่า **sensor:S01** เป็น Sensor แต่ไม่มี Triple **observes** OWL ภายใต้ Open World Assumption อาจถือว่าวัตถุที่ถูกสังเกตมีอยู่แต่ยังไม่ถูกระบุ ไม่ได้ตัดสินทันทีว่าข้อมูลผิด หากระบบงานต้องปฏิเสธระเบียนนี้ก่อนเผยแพร่ ต้องเขียน SHACL Shape เพิ่ม การแยกคำถามว่า “แบบจำลองขัดแย้งหรือไม่” ออกจาก “ข้อมูลพร้อมใช้หรือไม่” เป็นทักษะสำคัญกว่าการจำชื่อ Restriction จำนวนมาก

ในงานจริงควรทดสอบ Axiom ด้วยตัวอย่างที่คาดว่าจะผ่านและตัวอย่างที่คาดว่าจะขัดแย้ง สมมุติประกาศ Project และ Person เป็น Class ที่ Disjoint กัน ควรมี Graph ตัวอย่างที่ Resource หนึ่งเป็น Project อย่างเดียว และอีก Graph ที่จงใจประกาศ Resource เดียวเป็นทั้ง Project และ Person จากนั้น

จึงดูว่า Reasoner รายงานอย่างไร วิธีนี้ทำให้ทีมเห็นผลของ Axiom บนข้อมูลจริง และช่วยตรวจว่าเครื่องมือเปิด Reasoning Mode ตามที่คิดไว้หรือไม่

การอธิบายผลของ Reasoner ก็ควรย้อนกลับไปยัง Axiom ต้นทาง ไม่ควรหยุดที่ข้อความว่า Class Unsatisfiable หรือ Ontology Inconsistent หาก Resource หนึ่งถูกอนุมานเป็นสอง Class ที่ Disjoint กัน ต้องตามให้ได้ว่า Triple ใดและ Axiom ใดนำไปสู่ข้อสรุปนั้น การเก็บ Graph ตัวอย่างที่ระบุผลคาดหวังและเพิ่ม Axiom ทีละกลุ่มช่วยให้ตรวจที่มาของผลลัพธ์ได้ง่ายกว่าการเปิด Ontology ทั้งหมดแล้วพยายามหาสาเหตุพร้อมกัน

สำหรับผู้เริ่มต้น ควรเริ่มจาก RDFS ให้ตอบคำถามพื้นฐานก่อน เพิ่ม OWL เฉพาะความหมายที่มีผลต่อการอนุมานหรือความสอดคล้อง และเก็บข้อกำหนดเรื่องความครบถ้วนไว้ใน SHACL หากยังอธิบายไม่ได้ว่า Axiom หนึ่งช่วยคำถามใด ก็ยังไม่ควรใส่ Axiom นั้น Ontology ที่เหมาะกับงานจึงไม่จำเป็นต้องใช้ความสามารถของ OWL ครบทุกชนิด แต่ต้องมีตรรกะที่ผู้ใช้และทีมดูแลสามารถอธิบาย ทดสอบ และนำไปใช้ได้จริง

โดเมนการตรวจเหตุการณ์ช่วยอธิบายเหตุผลของการใช้ OWL ด้วยกรณีที่มีข้อมูลรองรับ การแยก AtomicEvent และ ComplexEvent ไม่ได้มีไว้เพื่อจัดหมวดหมู่ชื่อเท่านั้น แต่สะท้อนวิธีนิยามเหตุการณ์ Atomic Event อาจอาศัยหลักฐานหนึ่งรายการเป็นหลัก ส่วน Complex Event ประกอบจาก Sub-Event หลายรายการซึ่งอาจเป็น Atomic หรือ Complex ต่ออีกชั้น หากต้องการให้ Reasoner รู้ว่าเหตุการณ์ทุกชนิดยังเป็น Event การใช้ Subclass Hierarchy ก็เพียงพอ แต่หากต้องการนิยามว่า ComplexEvent ต้องมีองค์ประกอบบางอย่าง จึงค่อยพิจารณา Existential Restriction สมมุติให้ $\mathit{ComplexEvent}(x)$ หมายถึง x เป็นเหตุการณ์ซับซ้อน และ $\mathit{hasSubEvent}(x, y)$ หมายถึง x มี y เป็นเหตุการณ์ย่อย ความหมายขั้นต่ำอาจเขียนได้ว่า

$$\forall x \, (\mathit{ComplexEvent}(x) \rightarrow \mathit{Event}(x))$$

และถ้าขอบเขตของ Ontology ต้องการยืนยันว่า Complex Event มี Sub-Event อย่างน้อยหนึ่งรายการ อาจเขียนอย่างง่ายว่า

$$\forall x \, (\mathit{ComplexEvent}(x) \rightarrow \exists y \, (\mathit{Event}(y) \wedge \mathit{hasSubEvent}(x, y)))$$

ในสมการแรก x แทน Individual ใด ๆ และบอกว่า Complex Event ทุกตัวเป็น Event ในสมการที่สอง y แทนเหตุการณ์ย่อยอย่างน้อยหนึ่งตัว เครื่องหมาย $\wedge$ หมายถึงเงื่อนไขสองด้านเป็นจริงร่วมกัน คือ y เป็น Event และมีความสัมพันธ์ *hasSubEvent* จาก x ไปยัง y การเขียนเช่นนี้ทำให้ทีมอภิปรายได้ตรงว่าต้องการเพียง Hierarchy หรือต้องการข้อผูกพันเพิ่ม อย่างไรก็ตาม Axiom นี้ยังไม่รับรองว่า Graph ของเหตุการณ์ซับซ้อนแต่ละรายการกรอก Sub-Event ครบ เพราะ Open World Assumption ยอมรับว่า Sub-Event อาจมีอยู่แต่ยังไม่ถูกระบุ หากระบบตรวจจับเหตุการณ์ต้องปฏิเสธ Event Definition ที่ไม่มีองค์ประกอบต้องใช้ SHACL กำหนด **minCount** เพิ่ม นี่เป็นตัวอย่างจากโดเมนจริงที่แสดงว่า OWL กับ SHACL ไม่ควรถูกแทนกัน

ตัวอย่างเหตุการณ์ซับซ้อนนี้สรุปหลักของบทได้ชัดเจนว่า OWL มีคุณค่าเมื่อเปลี่ยนความเข้าใจของโดเมนให้เป็น Axiom ที่อธิบายและทดสอบได้ แต่ Axiom ทุกตัวต้องมีคำถาม ตัวอย่างข้อมูล และผลอนุมานที่คาดหวังรองรับ มิฉะนั้นความซับซ้อนที่เพิ่มขึ้นจะไม่ช่วยให้ Ontology มีความหมายหรือใช้งานได้ดีขึ้น

3.17 สรุป

บทนี้ขยายจาก Vocabulary ของ RDFS ไปสู่ Ontology ที่มีความหมายเชิงตรรกะ โดยแยก OWL 1, OWL 2, OWL DL และ OWL 2 DL ออกจากกัน แล้วใช้ First-Order Logic อธิบาย Predicate, Connective, Quantifier, Interpretation และ Entailment ก่อนเชื่อมไปยัง Description Logic, Class Expression, Restriction, Property Characteristic, Identity, Profile และการแบ่งหน้าที่กับ SHACL ประเด็นสำคัญจึงไม่ใช่จำนวน Axiom หรือจำนวนคำใน Namespace **owl:** แต่เป็นเหตุผลว่าความหมายใดจำเป็นต่อคำถามและการอนุมานของระบบ

OWL เป็นชั้นของ Semantic Web ที่ช่วยให้ Ontology มีความหมายเชิงตรรกะมากขึ้น แต่พลังของ OWL มาพร้อมภาระในการออกแบบ จึงควรใช้ OWL เพื่อระบุความหมายที่จำเป็นจริง และแยกบทบาทของ OWL ออกจาก SHACL ให้ชัดเจน บทถัดไปจะเปลี่ยนจากการนิยามความหมายไปสู่การสืบค้น Graph ด้วย SPARQL ซึ่งเป็นเครื่องมือหลักในการถามคำถามกับ Knowledge Graph

เอกสารอ้างอิง

Baader, F., Calvanese, D., McGuinness, D. L., Nardi, D., and Patel-Schneider, P. F., editors (2003). *The Description Logic Handbook: Theory, Implementation, and Applications*. Cambridge University Press.

Hitzler, P., Krötzsch, M., and Rudolph, S. (2009). *Foundations of Semantic Web Technologies*. CRC Press.

Horrocks, I., Patel-Schneider, P. F., and van Harmelen, F. (2003). From SHIQ and RDF to OWL: The making of a web ontology language. *Journal of Web Semantics*, 1(1):7–26.

Krötzsch, M. (2012). OWL 2 profiles: An introduction to lightweight ontology languages. In Eiter, T. and Krennwallner, T., editors, *Reasoning Web. Semantic Technologies for Advanced Query Answering*, volume 7487 of *Lecture Notes in Computer Science*, pages 112–183. Springer.

Labra Gayo, J. E., Prud'hommeaux, E., Boneva, I., and Kontokostas, D. (2017). *Validating RDF Data*. Morgan & Claypool.

World Wide Web Consortium (2004). OWL web ontology language overview. W3C Recommendation. Accessed July 12, 2026, from **https://www.w3.org/TR/owl-features/**.

World Wide Web Consortium (2012a). OWL 2 web ontology language direct semantics. W3C Recommendation. Accessed June 10, 2026, from **https://www.w3.org/TR/owl2-direct-semantics/**.

World Wide Web Consortium (2012b). OWL 2 web ontology language document overview. W3C Recommendation. Accessed July 12, 2026, from **https://www.w3.org/TR/owl2-overview/**.

World Wide Web Consortium (2012c). OWL 2 web ontology language mapping to RDF graphs. W3C Recommendation. Accessed July 12, 2026, from **https://www.w3.org/TR/owl2-mapping-to-rdf/**.

World Wide Web Consortium (2012d). OWL 2 web ontology language new features and rationale. W3C Recommendation. Accessed July 12, 2026, from **https://www.w3.org/TR/owl2-new-features/**.

World Wide Web Consortium (2012e). OWL 2 web ontology language primer. W3C Recommendation. Accessed June 11, 2026, from **https://www.w3.org/TR/owl2-primer/**.

World Wide Web Consortium (2012f). OWL 2 web ontology language profiles. W3C Recommendation. Accessed July 5, 2026, from **https://www.w3.org/TR/owl2-profiles/**.

World Wide Web Consortium (2012g). OWL 2 web ontology language quick reference guide. W3C Recommendation. Accessed July 12, 2026, from **https://www.w3.org/TR/owl2-quick-reference/**.

World Wide Web Consortium (2012h). OWL 2 web ontology language RDF-based semantics. W3C Recommendation. Accessed July 12, 2026, from **https://www.w3.org/TR/owl2-rdf-based-semantics/**.

World Wide Web Consortium (2012i). OWL 2 web ontology language structural specification and functional-style syntax. W3C Recommendation. Accessed July 7, 2026, from **https://www.w3.org/TR/owl2-syntax/**.

World Wide Web Consortium (2017). Shapes constraint language (SHACL). W3C Recommendation. Accessed July 3, 2026, from **https://www.w3.org/TR/shacl/**.

4

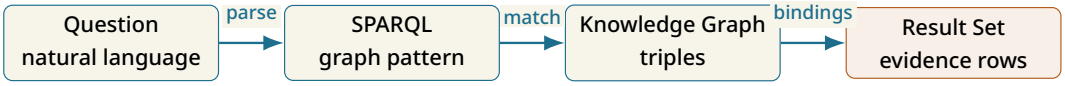
SPARQL: การค้นคืนข้อมูลและความรู้จากกราฟ

เมื่อข้อมูลถูกออกแบบเป็น RDF Graph และมีความหมายจาก RDFS/OWL แล้ว ขั้นต่อไปคือการถามคำถามกับ Graph นั้นอย่างเป็นระบบ บทนี้จึงเปลี่ยนจากมุมของการสร้างความหมายไปสู่มุมของการเรียกคืนความรู้ โดยอธิบาย SPARQL ตั้งแต่หลักการเขียนรูปแบบความสัมพันธ์ที่ต้องการหา แล้วให้ระบบจับคู่รูปแบบนั้นกับ Triple ที่มีอยู่จริงใน Graph

4.1 SPARQL กับการถามคำถามต่อ Graph

สามบทแรกอธิบายวิธีสร้าง Graph และกำหนดความหมายของคำศัพท์ แต่คุณค่าของโครงสร้างเหล่านั้นจะปรากฏเมื่อระบบนำมาใช้ตอบคำถาม หัวข้อนี้จึงเริ่มจากบทบาทของ SPARQL และเปลี่ยนคำถามของผู้ใช้ให้เป็นรูปแบบความสัมพันธ์ที่ค้นใน Graph ได้

เมื่อข้อมูลถูกจัดให้อยู่ในรูป RDF Graph แล้ว คำถามสำคัญต่อมาคือจะเรียกคืนความรู้จาก Graph ได้อย่างไร SPARQL เป็นภาษามาตรฐานของ W3C สำหรับสืบค้น RDF Graph โดยมีแนวคิดหลักคือการเขียน Pattern ของ Triple ที่ต้องการค้นหา แล้วให้ระบบจับคู่ Pattern นั้นกับข้อมูลจริงใน Graph (World Wide Web Consortium, 2013c,a) เส้นทางจากคำถามไปสู่ผลลัพธ์ที่มีหลักฐานแสดงในภาพที่ 4.1



ภาพที่ 4.1: SPARQL ทำให้การตอบคำถามของระบบผูกกับ graph pattern และ evidence rows

ภาพที่ 4.1 แสดงว่าคำถามต้องถูกแปลงเป็น SPARQL Graph Pattern ก่อนนำไปจับคู่กับ Knowledge Graph และคืนค่าเป็นแถวผลลัพธ์ที่ผูกกับหลักฐานในกราฟ ถ้าฐานข้อมูลสัมพันธ์ใช้ SQL เพื่อถามตาราง SPARQL ก็ทำหน้าที่คล้ายกันกับ RDF Graph แต่สิ่งที่ต่างคือ SPARQL ไม่ได้ถามแถวและคอลัมน์โดยตรง หากถามความสัมพันธ์ระหว่าง Resource เช่น โครงการใดใช้เทคโนโลยีใด องค์กรใดรับผิดชอบโครงการใด หรือ Evidence ใดรองรับ Claim ใด ดังนั้นการเขียน SPARQL ที่ดีจึงขึ้นอยู่กับการออกแบบ RDF และ Ontology ก่อนหน้าอย่างมาก

ในด้านความทันสมัยของมาตรฐาน หนังสือเล่มนี้ใช้ SPARQL 1.1 เป็นฐานในการอธิบาย เพราะยังเป็น Recommendation ที่เครื่องมือส่วนใหญ่รองรับอย่างแพร่หลาย แต่ W3C ได้เผยแพร่ SPARQL 1.2 Query Language เป็น Working Draft ในวันที่ 25 มิถุนายน 2026 (World Wide Web Consortium, 2026) ดังนั้นผู้อ่านที่ต้องทำระบบใหม่ควรติดตาม SPARQL 1.2 ในฐานะงานมาตรฐานที่กำลังพัฒนา โดยเฉพาะ

ประเด็น Version announcement และความสอดคล้องกับชุดเอกสาร RDF/SPARQL 1.2 ทั้งนี้ แนวคิดหลักของบทนี้ยังอยู่ที่การจับคู่ Graph Pattern และการดึง Evidence ซึ่งเป็นรากฐานร่วมของ SPARQL ทั้งสองรุ่น

4.2 มุมมองเชิงคณิตศาสตร์ของ Query

ตัวอย่าง SPARQL เบื้องต้นช่วยให้เห็นรูปเขียนของภาษา ส่วนมุมมองเชิงคณิตศาสตร์ช่วยอธิบายว่าเหตุใดตัวแปรที่ใช้ร่วมกันจึงทำให้ Triple Pattern เชื่อมกัน หัวข้อนี้จึงมอง Query เป็นเงื่อนไขหลายชุดที่ต้องมี Binding สอดคล้องกัน

SPARQL พื้นฐานสามารถมองเป็น conjunctive Query ได้ กล่าวคือ Query หนึ่งประกอบด้วยเงื่อนไขหลายชุดที่ต้องเป็นจริงพร้อมกัน หากเขียน Pattern ว่า

```
PREFIX ex: <https://example.org/vocab/>

SELECT ?project ?tech ?org
WHERE {
 ?project ex:usesTechnology ?tech .
 ?project ex:responsibleOrg ?org .
}
```

กำลังถามหา binding ของตัวแปร **?project**, **?tech** และ **?org** ที่ทำให้ทั้งสอง Triple Pattern ตรงกับข้อมูลใน Graph ในเชิงตรรกะสามารถมองได้ว่า

$$\mathit{usesTechnology}(p, t) \wedge \mathit{responsibleOrg}(p, o)$$

ในสูตรนี้ p แทน Project ที่ระบบกำลังหา, t แทน Technology และ o แทน Organization ส่วน $\mathit{usesTechnology}(p, t)$ หมายถึงมีความสัมพันธ์ว่า Project p ใช้ Technology t และ $\mathit{responsibleOrg}(p, o)$ หมายถึง Project เดียวกันมี Organization o รับผิดชอบ ผลลัพธ์ของ Query จึงคือชุดค่าของ p, t, o ที่ทำให้เงื่อนไขทั้งสองเป็นจริงพร้อมกันใน Graph ที่กำลังสืบค้น แนวคิดนี้อธิบายว่าทำไม SPARQL จึงเหมาะกับข้อมูลที่สัมพันธ์กัน เพราะ Query ไม่ได้ค้นจากคำที่เหมือนกันเท่านั้น แต่ค้นจาก Pattern ของความสัมพันธ์ที่ถูกนิยามไว้ใน Graph (Pérez et al., 2009, 2006)

คำว่า binding ใน SPARQL หมายถึงการจับค่าจริงจาก Graph มาแทนตัวแปรใน Query เช่นถ้า Graph มี Triple ว่า **proj:P001 ex:usesTechnology tech:SensorNet** และ **proj:P001 ex:responsibleOrg org:PSU** ระบบอาจสร้าง binding ได้ว่า $p =$ **proj:P001**, $t =$ **tech:SensorNet** และ $o =$ **org:PSU** หากมี Project อีกตัวที่ใช้เทคโนโลยีเดียวกันและมีองค์กรรับผิดชอบอีกองค์กรหนึ่ง ระบบก็จะสร้าง binding อีกแถวหนึ่ง ผลลัพธ์ของ SPARQL จึงมีรูปแบบคล้ายตาราง แต่ข้อมูลแต่ละแถวเกิดจากการจับคู่ Graph Pattern กับข้อมูล ไม่ใช่การอ่านจากตารางต้นทางโดยตรง

มุมมองนี้ช่วยให้เข้าใจการ join ของ SPARQL ได้ดีขึ้น ตัวแปรที่ชื่อเหมือนกันในหลาย Triple Pattern คือจุดเชื่อมของเงื่อนไข เช่น **?project** ในตัวอย่างข้างต้นต้องเป็น Project เดียวกันทั้งในความสัมพันธ์ **ex:usesTechnology** และ **ex:responsibleOrg** หาก Query ใช้ตัวแปรคนละชื่อโดยไม่ตั้งใจ ระบบจะมองว่าเป็นเงื่อนไขคนละชุดและผลลัพธ์อาจกว้างเกินไป ดังนั้นการตั้งชื่อตัวแปรให้สื่อความหมาย เช่น **?project**, **?tech**, **?org**, **?claim**, **?report** เป็นเรื่องสำคัญต่อทั้งคนอ่านและความถูกต้องของ Query

4.3 SELECT, FILTER, OPTIONAL และ ASK

เมื่อเข้าใจ Pattern และ Binding แล้ว ขั้นต่อไปคือเลือกชนิดผลลัพธ์และเงื่อนไขให้ตรงกับคำถาม SELECT, FILTER, OPTIONAL และ ASK จึงถูกนำมาเปรียบเทียบตามหน้าที่ไม่ใช่ในฐานะ Syntax ที่ต้องจดจำแยกกัน

รูปแบบที่ใช้บ่อยที่สุดคือ **SELECT** ซึ่งระบุตัวแปรที่ต้องการแสดงผล ตัวอย่างเช่น

```
PREFIX ex: <https://example.org/vocab/>

SELECT ?project ?tech
WHERE {
 ?project ex:usesTechnology ?tech .
}
```

ถ้าต้องการจำกัดผลลัพธ์ สามารถใช้ **FILTER** เช่น เลือกเฉพาะชื่อภาษาไทย หรือเลือกเฉพาะวันที่หลังช่วงเวลาหนึ่ง ส่วน **OPTIONAL** ใช้เมื่อข้อมูลบางส่วนอาจมีหรือไม่มีก็ได้ เช่น บางโครงการอาจมี Label ภาษาไทย แต่บางโครงการยังไม่มี หากไม่ใช้ **OPTIONAL** ผลลัพธ์ที่ไม่มี Label จะถูกตัดออกทั้งหมด สำหรับการตรวจสอบว่า Claim บางอย่างมีอยู่หรือไม่ **ASK** เป็นรูปแบบที่เหมาะสม เพราะผลลัพธ์เป็น true หรือ false ตัวอย่างเช่นถามว่าโครงการ A ใช้ SensorNet หรือไม่ Query แบบนี้มีประโยชน์ในระบบที่ต้องตรวจว่ากราฟมีหลักฐานรองรับข้อความหนึ่งหรือไม่ หรือใช้ตรวจ Policy ก่อนอนุญาตให้แสดงข้อมูล (Mishra et al., 2022)

แต่ผลของ **ASK** ต้องอ่านอย่างระมัดระวัง หากได้ผลลัพธ์เป็น false ไม่ได้แปลว่า Claim นั้นเป็นเท็จเสมอไป อาจแปลว่า Graph ยังไม่มีข้อมูล หรือ Query ใช้ IRI ผิด หรือข้อมูลอยู่ใน Named Graph อื่นที่ไม่ได้ถูก Query หลักการนี้ใกล้กับ Open World ที่กล่าวในบท OWL กล่าวคือการไม่พบข้อมูลไม่เท่ากับการพิสูจน์ว่าไม่มีข้อมูล ดังนั้น **ASK** จึงเหมาะกับการตรวจว่า Graph ปัจจุบันมีหลักฐานรองรับหรือไม่ มากกว่าการประกาศความจริงทั้งหมดของโลก

4.4 CONSTRUCT และการสร้าง Graph ใหม่

คำสั่งก่อนหน้าคืนผลลัพธ์เป็นตารางหรือค่าความจริง แต่บางงานต้องการส่งต่อโครงสร้างความสัมพันธ์เป็น RDF Graph หัวข้อนี้จึงเพิ่ม CONSTRUCT เพื่อสราง Evidence Subgraph ที่ยังรักษาความเชื่อมโยงของ Resource ไว้

นอกจากการดึงตารางผลลัพธ์ SPARQL ยังสามารถใช้ **CONSTRUCT** เพื่อสร้าง RDF Graph ใหม่จาก Pattern เดิมได้ ตัวอย่างเช่น หากต้องการสร้าง Graph เฉพาะความสัมพันธ์ระหว่างโครงการ เทคโนโลยี และองค์กรที่เกี่ยวข้อง สามารถเขียน Query ที่ดึงเฉพาะ Triple ที่จำเป็นออกมาเป็น subgraph สำหรับส่งต่อให้ระบบอื่น แนวคิดนี้สำคัญเมื่อ Application ต้องส่งต่อหลักฐานให้ Dashboard, API หรือระบบรายงาน เพราะไม่จำเป็นต้องส่ง Graph ทั้งหมดไปยังทุกส่วน SPARQL สามารถดึง Evidence subgraph ที่เกี่ยวข้องกับคำถาม โดยยังคง IRI และชื่อ Predicate ไว้ครบ ทำให้ผลลัพธ์ตรวจสอบย้อนกลับไปยัง Graph ต้นทางได้

ตัวอย่างเช่น ถ้าต้องการส่งหลักฐานเรื่องโครงการหนึ่งให้ระบบอธิบายเป็นภาษาไทย อาจไม่ต้องส่งข้อมูลทุก Triple ขององค์กรและเทคโนโลยี แต่สร้าง subgraph ที่มีเฉพาะ Project, Technology, Outcome

Claim และ Report ที่เกี่ยวข้อง การใช้ **CONSTRUCT** ทำให้ Evidence ถูกส่งต่อในรูป RDF เหมือนเดิม ไม่สูญเสียชื่อ Predicate และ IRI ระหว่างทาง เมื่อผู้ใช้ต้องการตรวจคำตอบ ระบบยังย้อนกลับไปยัง Triple ต้นทางได้ เพราะคำตอบไม่ได้สร้างจากข้อความที่ถูกตัดบริบทออกจาก Graph

4.5 Query Catalog และ Evidence Retrieval

เมื่อ Query เริ่มถูกใช้ซ้ำ การเก็บ Query ไว้เป็นไฟล์กระจัดกระจายทำให้ตรวจสอบและดูแลได้ยาก Query Catalog จึงเชื่อมการเขียน SPARQL เข้ากับการจัดการ Artifact ของระบบ และทำให้ Evidence Retrieval มีคำอธิบายกับผลลัพธ์ที่คาดไว้

ในระบบจริงไม่ควรปล่อยให้ทุกคนเขียน SPARQL ใหม่ตั้งแต่ต้นเสมอไป ควรมี Query Catalog ที่รวบรวม Query สำคัญของระบบ เช่น Query สำหรับค้นโครงการตามเทคโนโลยี Query สำหรับค้น Evidence ของ Outcome และ Query สำหรับตรวจความครบถ้วนของข้อมูล การมี Query Catalog ทำให้ทีมทำงานซ้ำได้ ตรวจสอบผลลัพธ์ได้ และใช้เป็นเอกสารทางวิชาการของระบบได้ด้วย Query Catalog ที่ดีควรมีมากกว่าไฟล์ **.rq** หลายไฟล์ ควรบอกชื่อ Query วัตถุประสงค์ ตัวแปรนำเข้า ผลลัพธ์ที่คาด Ontology Version ที่รองรับ และตัวอย่าง Graph ที่ใช้ทดสอบได้ เช่น Query “Project by Technology” ควรบอกว่ารับ IRI ของ Technology เป็น input และคืน Project, Label, Organization และ Evidence Graph ใดบ้าง หาก Catalog ระบุข้อมูลเหล่านี้ไว้ชัดเจน Query จะกลายเป็น Artifact ที่ Review ได้ ไม่ใช่ script ส่วนตัวของผู้พัฒนาคนหนึ่ง

SPARQL จึงไม่ใช่เพียงภาษา Query แต่เป็นสัญญาระหว่าง Ontology, data Pipeline และ Application ถ้า Query ที่จำเป็นเขียนไม่ได้ แปลว่า Ontology หรือ RDF Mapping อาจยังไม่ตอบโจทย์จริง เนื้อหาส่วนถัดไปจึงนำหลักนี้ไปใช้กับคำถามของผู้ใช้และรูปแบบ Query ที่พบบ่อย ก่อนเชื่อมกลับไปสู่กระบวนการออกแบบ Ontology อย่างเป็นระบบในบทถัดไป

4.6 การเขียน Query จากคำถามของผู้ใช้

Query Catalog ควรเริ่มจากคำถามที่ผู้ใช้ต้องการตอบ ไม่ใช่เริ่มจากรายการ Syntax ของ SPARQL หัวข้อนี้จึงแยก Entity, Relation และเงื่อนไขจากภาษาธรรมชาติ ก่อนแปลงเป็น Triple Pattern ทีละส่วน

จุดเริ่มต้นของ SPARQL ที่ดีคือคำถามของผู้ใช้ ไม่ใช่ Syntax ของภาษา ตัวอย่างเช่น หากผู้ใช้ถามว่า “โครงการใดใช้ SensorNet และอยู่ภายใต้หน่วยงานใด” ควรแยกคำถามนี้ออกเป็นความสัมพันธ์ใน Graph ก่อน คือ Project ใช้ Technology และ Project มี responsible Organization จากนั้นจึงเขียน SPARQL ให้ตรงกับ Pattern ดังกล่าว

```
PREFIX ex: <https://example.org/vocab/>
PREFIX tech: <https://example.org/resource/technology/>

SELECT ?project ?org
WHERE {
 ?project ex:usesTechnology tech:SensorNet .
 ?project ex:responsibleOrg ?org .
}
```

Query นี้สั้น แต่แสดงหลักสำคัญของ SPARQL คือไม่ได้ค้นคำว่า SensorNet ในเอกสาร แต่ค้น Resource ที่เชื่อมกับ **tech:SensorNet** ผ่าน Predicate ที่กำหนดไว้ หาก RDF Mapping เปลี่ยนจาก **ex:usesTechnology** ไปใช้ Property อื่น Query นี้ก็ต้องเปลี่ยนด้วย ดังนั้น Query Catalog จึงเป็นหลักฐานว่าระบบใช้ Vocabulary อย่างไร สำหรับผู้อ่านที่คุ้นกับ SQL สิ่งที่ต้องปรับความคิดคือ SPARQL Pattern ไม่จำเป็นต้องรู้ว่าข้อมูลอยู่ในตารางใด เพราะ Graph ถูกมองเป็นชุดของ Triple ทั้งหมด การ join เกิดจากตัวแปรร่วม เช่น **?project** ในตัวอย่างข้างต้น ตัวแปรเดียวกันปรากฏในสอง Pattern ทำให้ระบบเชื่อมผลลัพธ์ของสองความสัมพันธ์เข้าด้วยกัน

4.7 OPTIONAL กับข้อมูลที่ยังไม่ครบ

Query ก่อนหน้าใช้ Pattern ที่ต้องตรงครบทุกส่วน แต่ Graph ในระบบจริงมักมีข้อมูลบางรายการไม่สมบูรณ์ OPTIONAL จึงถูกเพิ่มเพื่อรักษาผลลัพธ์หลักไว้ พร้อมแสดงข้อมูลเสริมเมื่อมีอยู่จริง

ข้อมูลใน Knowledge Graph มักไม่ครบตั้งแต่ต้น หากกำหนดทุก Triple Pattern เป็นเงื่อนไขบังคับ ผลลัพธ์ที่มีประโยชน์อาจถูกตัดออกทั้งหมด เช่น ต้องการแสดง Project พร้อม Label ภาษาไทย แต่บาง Project ยังไม่มี Label ภาษาไทย หากเขียนแบบบังคับทั้งหมด ระบบจะคืนเฉพาะ Project ที่มี Label เท่านั้น

```
PREFIX ex: <https://example.org/vocab/>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>

SELECT ?project ?label
WHERE {
 ?project a ex:Project .
 OPTIONAL {
 ?project rdfs:label ?label .
 FILTER(lang(?label) = "th")
 }
}
```

การใช้ **OPTIONAL** ทำให้ Project ที่ไม่มี Label ยังปรากฏในผลลัพธ์ เพียงแต่ค่า **?label** ว่างอยู่ วิธีนี้เหมาะกับระบบที่ข้อมูลกำลังถูกเติมเต็มทีละส่วน และยังช่วยให้ทีมเห็นว่าข้อมูลส่วนใดขาดหาย หากต้องการตรวจว่าข้อมูลขาดแล้วต้องแก้จริง ควรใช้ SHACL ไม่ใช่ทำให้ SPARQL Query ล้มเหลว

4.8 Aggregation และการสรุปผล

OPTIONAL ช่วยจัดการข้อมูลที่ขาดเป็นราย Resource ส่วนคำถามเชิงรายงานต้องรวมหลาย Resource เข้าด้วยกัน หัวข้อนี้จึงพิจารณา Aggregation สำหรับนับ จัดกลุ่ม และเรียงผล โดยยังต้องรักษาความหมายของหน่วยที่นำมารวม

SPARQL รองรับการ aggregation เช่น **COUNT**, **GROUP BY** และ **ORDER BY** ซึ่งจำเป็นเมื่อ Graph ถูกใช้เพื่อทำรายงาน ตัวอย่างเช่น ต้องการนับจำนวนโครงการตามเทคโนโลยีที่ใช้

```
PREFIX ex: <https://example.org/vocab/>

SELECT ?tech (COUNT(?project) AS ?count)
WHERE {
 ?project ex:usesTechnology ?tech .
}
GROUP BY ?tech
ORDER BY DESC(?count)
```

Query แบบนี้ทำให้ Knowledge Graph ไม่ใช่เพียงฐานข้อมูลเชิงความหมาย แต่เป็นแหล่งข้อมูลสำหรับ Dashboard และ analysis ได้ด้วย อย่างไรก็ตาม aggregation จะมีความหมายก็ต่อเมื่อ RDF Mapping มีคุณภาพ หากเทคโนโลยีเดียวกันถูกสร้างเป็นหลาย IRI ผลนับจะถูกแยกออกเป็นหลายกลุ่มอย่างผิดพลาด ดังนั้นคุณภาพของ IRI Policy ส่งผลโดยตรงต่อผลลัพธ์ของ Query

4.9 Named Graph Query และ Evidence

ผลสรุปจาก Graph มีความหมายมากขึ้นเมื่อทราบว่าข้อมูลมาจากแหล่งหรือ Version ใด Named Graph จึงเชื่อมการ Query เข้ากับบริบทของ Evidence และเปิดให้ผลลัพธ์ระบุทั้ง Claim กับ Graph ที่รองรับ Claim นั้น

เมื่อระบบใช้ Named Graph เพื่อแยกข้อมูลตาม Source หรือ Version SPARQL สามารถระบุได้ว่าต้องการค้นใน Graph ใด ตัวอย่างเช่นต้องการค้น Claim ที่อยู่ใน Graph ของรายงานไตรมาสแรก

```
PREFIX ex: <https://example.org/vocab/>
PREFIX graph: <https://example.org/resource/graph/>

SELECT ?project ?outcome
WHERE {
 GRAPH graph:Q1Report {
 ?project ex:hasOutcome ?outcome .
 }
}
```

หากต้องการคืนทั้งคำตอบและแหล่งที่มา สามารถใช้ตัวแปร Graph ได้

```
PREFIX ex: <https://example.org/vocab/>

SELECT ?g ?project ?outcome
WHERE {
 GRAPH ?g {
 ?project ex:hasOutcome ?outcome .
 }
}
```

รูปแบบนี้สำคัญมากต่อระบบที่ต้องการอธิบายคำตอบ เพราะผลลัพธ์ไม่ได้มีแค่ Project และ Outcome แต่มี Graph ที่เป็นบริบทของ Claim ด้วย เมื่อเชื่อมกับ PROV-O ระบบสามารถบอกต่อได้ว่า Graph นั้น derived from เอกสารใด สร้างเมื่อไร และใครเป็นผู้รับผิดชอบ

4.10 การตรวจ Query ก่อนนำไปใช้

เมื่อ Query สามารถระบุ Graph และดึง Evidence ได้ ความเสี่ยงจะย้ายจากการเขียน Pattern ไปสู่การควบคุม Query ที่ระบบยอมให้รัน หัวข้อนี้จึงพิจารณาทั้งความปลอดภัย ความสอดคล้องกับ Ontology และขอบเขตการเข้าถึงข้อมูลก่อนส่ง Query ไปยัง Endpoint

Query ที่มาจากส่วนติดต่อผู้ใช้หรือถูกประกอบขึ้นจาก Parameter ไม่ควรถูกส่งไปยัง Endpoint โดยไม่มีการตรวจ เพราะอาจใช้ Property ผิด ใช้ Prefix ที่ไม่มีอยู่ หรือมี Pattern กว้างเกินความจำเป็น แม้ Query จะผ่านการตรวจ Syntax ก็ยังอาจไม่ตรงกับความหมายของคำถามได้ แนวทางที่ควบคุมได้คือให้ Application เลือก Query Template จาก Catalog แล้วเติม IRI หรือค่าพารามิเตอร์ที่ผ่านการตรวจ วิธีนี้รักษาโครงสร้างของ Query ที่เคยทดสอบแล้วและแยกการตีความคำถามออกจากความหมายของ Graph หากคำถามไม่มี Template รองรับ ระบบควรส่งกลับไปยังขั้นออกแบบ Query แทนการประกอบ Pattern ที่ไม่มีหลักประกัน

ก่อนรัน Query ระบบควรตรวจว่า Prefix ที่ใช้มีอยู่จริง Property ทุกตัวอยู่ใน Ontology Version ปัจจุบัน ชนิด Query อยู่ในรายการที่อนุญาต และ Query อ่านเฉพาะ Named Graph ที่ผู้ใช้มีสิทธิ์เข้าถึง การตรวจเช่นนี้ทำให้ Query Catalog เป็นส่วนหนึ่งของการกำกับระบบ ไม่ใช่เพียงคลังตัวอย่างสำหรับคัดลอก

4.11 ประสิทธิภาพและขอบเขตของ Query

Query ที่ผ่านการตรวจด้านโครงสร้างและสิทธิ์อาจยังใช้ทรัพยากรมากเกินไปเมื่อ Graph มีขนาดใหญ่ หัวข้อนี้จึงเชื่อมความถูกต้องของ Query กับประสิทธิภาพ โดยพิจารณาความกว้างของ Pattern, Join, Property Path และขอบเขตผลลัพธ์

SPARQL Query ที่อ่านง่ายไม่ได้แปลว่าจะทำงานเร็วเสมอไป ใน Graph ขนาดใหญ่ Query ที่มี Pattern กว้างมาก หรือใช้ Property path โดยไม่จำกัดขอบเขต อาจทำให้ระบบทำงานช้าอย่างมาก งานด้าน RDF store เช่น RDF-3X แสดงให้เห็นว่าการจัดเก็บและ optimization มีผลต่อ performance ของ RDF Query โดยตรง (Neumann and Weikum, 2010) ในงานปฏิบัติ ควรเขียน Query ให้เริ่มจาก Pattern ที่จำกัดผลลัพธ์ได้ดี เช่นระบุ type หรือระบุ Resource ที่รู้แน่ก่อน แล้วค่อย join ไปยัง Relation อื่น นอกจากนี้ควรทดสอบ Query กับข้อมูลจริงและเก็บ Query ที่สำคัญไว้ใน Catalog พร้อมคำอธิบายว่าคาดหวังผลลัพธ์อย่างไร Query ที่ไม่มีเจ้าของและไม่มีคำอธิบายมักกลายเป็นหนี้ทางเทคนิคของระบบ Knowledge Graph

เมื่อ Graph ใหญ่ขึ้น performance ไม่ได้ขึ้นกับ Query อย่างเดียว แต่ขึ้นกับวิธีโหลดข้อมูล การเปิด Reasoning การใช้ Named Graph และ index ของ Triple Store ด้วย Query เดียวกันอาจเร็วใน Graph ตัวอย่างแต่ช้าใน Graph จริงที่มีหลายล้าน Triple ดังนั้น Query สำคัญควรถูกทดสอบกับข้อมูลขนาดใกล้เคียงงานจริง และควรบันทึกเวลาประมาณการไว้ใน Catalog หาก Query หนึ่งใช้เป็นส่วนหนึ่งของ API ที่ผู้ใช้เรียกบ่อย ก็ควรมีเกณฑ์เวลาและแผนปรับปรุง เช่นลด Pattern ที่กว้างเกินไป หรือสร้าง Graph materialized เฉพาะสำหรับรายงานบางประเภท

4.12 กรณีศึกษาแบบต่อเนื่อง: Query Catalog ของข้อมูลโครงการ

แนวคิดเรื่อง Catalog, Evidence และประสิทธิภาพจะเห็นได้ชัดขึ้นเมื่อกลับมายัง Graph โครงการ หัวข้อนี้จึงจัดคำถามสำคัญของโดเมนเป็น Query Catalog และแสดงความสัมพันธ์ระหว่าง Competency Questions, Ontology และผลลัพธ์ที่คาดหวัง

เพื่อให้ SPARQL ไม่เป็นเพียงภาษา Query ที่อยู่ในเอกสาร ลองกลับมาที่ Graph โครงการจากบทก่อน สมมติว่าระบบมีข้อมูลโครงการ องค์กร เทคโนโลยี รายงาน Outcome และ Provenance แล้ว คำถามของผู้ใช้อาจไม่ใช่ภาษาเทคนิค แต่เป็นคำถามธรรมดา เช่น “โครงการใดใช้ SensorNet”, “โครงการใดมี Outcome ที่มีรายงานรองรับ”, หรือ “ข้อมูลใดที่ยังไม่มีหน่วยงานรับผิดชอบ” หน้าที่ของ Query Catalog คือ เก็บรูปแบบ Query เหล่านี้ให้เป็นทรัพย์สินของระบบ Query แรกอาจเป็น Query สำหรับค้นโครงการตามเทคโนโลยี

```
PREFIX ex: <https://example.org/vocab/>
PREFIX tech: <https://example.org/resource/technology/>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>

SELECT ?project ?label ?org
WHERE {
 ?project a ex:Project ;
 ex:usesTechnology tech:SensorNet ;
 ex:responsibleOrg ?org .
 OPTIONAL {
 ?project rdfs:label ?label .
 FILTER(lang(?label) = "th")
 }
}
```

Query นี้ดูตรงไปตรงมา แต่บอกอะไรหลายอย่างเกี่ยวกับ Ontology ของระบบ หนึ่ง ระบบถือว่า Project ต้องมี Relation กับ Technology ผ่าน **ex:usesTechnology** สอง ระบบถือว่าหน่วยงานรับผิดชอบอยู่ที่ **ex:responsibleOrg** สาม Label ภาษาไทยเป็นข้อมูลเสริมที่อาจมีหรือไม่มีก็ได้ จึงใช้ **OPTIONAL** การอ่าน Query ในลักษณะนี้ช่วยให้ผู้พัฒนาเห็นว่า Query Catalog เป็นเอกสารของ semantic Contract ด้วย ไม่ใช่เพียงไฟล์สำหรับรันคำสั่ง Query ที่สองอาจใช้ดึง Outcome พร้อมหลักฐาน

```
PREFIX ex: <https://example.org/vocab/>
PREFIX prov: <http://www.w3.org/ns/prov#>

SELECT ?project ?outcome ?report ?time
WHERE {
 ?claim a ex:OutcomeClaim ;
 ex:aboutProject ?project ;
 ex:assertedOutcome ?outcome ;
 prov:wasDerivedFrom ?report ;
 prov:generatedAtTime ?time .
}
```

รูปแบบนี้ต่างจากการเก็บ Outcome เป็น Literal ตรง ๆ เพราะ Outcome ถูกผูกกับ Claim และ Claim ถูกผูกกับ Report เมื่อ Application นำผลลัพธ์ไปแสดง คำตอบจึงมีฐานหลักฐานชัดเจน ผู้ใช้ไม่ได้

เห็นเพียงข้อความ Outcome แต่เห็นด้วยว่า Outcome นั้นมาจากรายงานใดและถูกสร้างเมื่อไร Query ที่สามอาจเป็น Query เพื่อหาข้อมูลที่ยังไม่สมบูรณ์

```
PREFIX ex: <https://example.org/vocab/>

SELECT ?project
WHERE {
 ?project a ex:Project .
 FILTER NOT EXISTS { ?project ex:responsibleOrg ?org . }
}
```

Query นี้ไม่ใช่ Validation Rule เต็มรูปแบบ แต่ช่วยให้ทีมตรวจข้อมูลเบื้องต้นได้ การมี Query แบบนี้ใน Catalog ทำให้ทีมเห็นปัญหาข้อมูลก่อนที่ข้อมูลจะถูกใช้ในรายงานหรือระบบตอบคำถาม ในระบบ production Query ประเภทนี้มักทำงานคู่กับ SHACL กล่าวคือ SPARQL ใช้สำรวจและวิเคราะห์ ส่วน SHACL ใช้สร้าง Validation Report ที่เป็นมาตรฐานกว่า เมื่อต้องการส่งหลักฐานให้ระบบอื่น เช่น Dashboard, API หรือระบบรายงาน อาจใช้ **CONSTRUCT** เพื่อสร้าง subgraph เฉพาะส่วนที่ต้องการได้ เช่น

```
PREFIX ex: <https://example.org/vocab/>
PREFIX tech: <https://example.org/resource/technology/>
PREFIX prov: <http://www.w3.org/ns/prov#>

CONSTRUCT {
 ?project ex:usesTechnology ?tech .
 ?project ex:hasOutcome ?outcome .
 ?claim ex:aboutProject ?project ;
 ex:assertedOutcome ?outcome ;
 prov:wasDerivedFrom ?report .
}
WHERE {
 ?project ex:usesTechnology ?tech .
 ?claim a ex:OutcomeClaim ;
 ex:aboutProject ?project ;
 ex:assertedOutcome ?outcome ;
 prov:wasDerivedFrom ?report .
 FILTER(?tech = tech:SensorNet)
}
```

ผลลัพธ์ของ **CONSTRUCT** ไม่ใช่ตาราง แต่เป็น Graph ใหม่ที่เล็กลงและเกี่ยวข้องกับคำถามโดยตรง ระบบจึงไม่จำเป็นต้องส่ง Graph ทั้งฐานไปยังชั้นถัดไป แต่ส่งเฉพาะ subgraph ที่มี Relation สำคัญครบถ้วน ชั้นแสดงผลยังสามารถใช้ Label ภาษาไทยได้ ขณะเดียวกันก็รักษา IRI และเส้นทางของหลักฐานไว้สำหรับการตรวจสอบ

Query Catalog ที่ดีควรเก็บ Metadata ของ Query ด้วย เช่น ชื่อ Query วัตถุประสงค์ ผู้รับผิดชอบ Ontology Version ที่รองรับ Shape ที่เกี่ยวข้อง ตัวอย่างผลลัพธ์ และข้อควรระวัง หาก Query หนึ่งพึ่งพา Property **ex:hasOutcome** แต่ Ontology Version ใหม่เปลี่ยนชื่อ Property เป็น **ex:assertedOutcome** Catalog ต้องบอกให้ทีมรู้ว่า Query นี้ต้องแก้ ไม่เช่นนั้นระบบอาจยังรันได้บางส่วนแต่ให้ผลลัพธ์ผิดหรือไม่ครบ ในทางปฏิบัติ Query ที่รับค่าจากผู้ใช้ควรทำงานบน Catalog มากกว่าประกอบ Query จากข้อความทุกส่วนโดยตรง ระบบควรจำแนกก่อนว่าคำถามต้องการค้น Project ตาม Technology หรือค้น Outcome ตาม

Report จากนั้นระบบเลือก Template ที่ผ่านการทดสอบแล้วและเติมค่า เช่น **tech:SensorNet** หรือช่วงเวลา วิธีนี้ลดโอกาสการใช้ Property ที่ไม่มีอยู่จริง และทำให้ Query ที่รันยังอยู่ในขอบเขตที่ตรวจสอบได้

ดังนั้น SPARQL ในหนังสือเล่มนี้ควรถูกมองเป็นภาษาที่เชื่อม Ontology เข้ากับการใช้งานจริง หาก Ontology ออกแบบดีแต่ไม่มี Query ที่ตอบคำถามได้ Ontology นั้นก็ยังไม่ทำหน้าที่ในระบบเต็มที่ ในทางกลับกัน หาก Query ต้องซับซ้อนเกินไปเพื่อดึงข้อมูลพื้นฐาน แปลว่า Ontology หรือ RDF Mapping อาจยังไม่สะท้อนวิธีคิดของโดเมนอย่างเหมาะสม

4.13 Query Test และการอ่านผลลัพธ์อย่างมีวินัย

การมี Query Catalog ยังไม่รับรองว่า Query จะทำงานถูกต้องหลัง Ontology หรือ Mapping เปลี่ยนแปลง หัวข้อนี้จึงเพิ่ม Test Graph และ Expected Result เพื่อให้ Query เป็น Artifact ที่ตรวจซ้ำได้ และเพื่อแยกผลลัพธ์ว่างออกจากข้อสรุปว่า Claim เป็นเท็จ

เมื่อ Query Catalog เริ่มมีหลายรายการ ปัญหาต่อมาคือจะรู้ได้อย่างไรว่า Query ยังทำงานถูกต้องหลัง Ontology หรือ Mapping เปลี่ยนไป วิธีหนึ่งคือสร้าง Query test สำหรับ Query สำคัญ คล้ายกับการทดสอบ software แต่เปลี่ยนจากการตรวจ function มาเป็นการตรวจผลลัพธ์ของ SPARQL บน Graph ตัวอย่าง เช่น Query “Project by Technology” ควรมี Graph ตัวอย่างที่รู้คำตอบล่วงหน้า หาก Graph ตัวอย่างมี Project สองตัวที่ใช้ **tech:SensorNet** Query ก็ควรคืนสองแถว หากคืนหนึ่งแถว แปลว่า Query หรือ Mapping มีปัญหา หากคืนสามแถว แปลว่า Graph หรือ Ontology อาจมีข้อมูลเกินหรือ Inference ที่ไม่คาดคิด การมี Expected Result ทำให้ทีมไม่ต้องตรวจด้วยสายตาทุกครั้ง รูปแบบ test อย่างง่ายอาจประกอบด้วยสามไฟล์

```
tests/project-by-technology/input.ttl
tests/project-by-technology/query.rq
tests/project-by-technology/expected.csv
```

ไฟล์ **input.ttl** เก็บ Graph ตัวอย่าง ไฟล์ **query.rq** เก็บ SPARQL และไฟล์ **expected.csv** เก็บผลลัพธ์ที่คาด หาก Ontology Version ใหม่ทำให้ Query fail ทีมจะรู้ทันทีว่าต้องแก้ Query, Mapping หรือ Ontology การทดสอบแบบนี้ช่วยให้ Query Catalog เป็น Artifact ที่มีคุณภาพ ไม่ใช่คลัง Query ที่ค่อย ๆ เสื่อมสภาพตามเวลา นอกจากนี้ควรแยก Query ตามบทบาท เช่น Query สำหรับผู้ใช้ปลายทาง Query สำหรับตรวจคุณภาพข้อมูล Query สำหรับสร้าง Evidence subgraph และ Query สำหรับ Monitoring performance แต่ละกลุ่มมีเกณฑ์ต่างกัน Query สำหรับผู้ใช้ปลายทางต้องให้ผลลัพธ์อ่านง่าย Query สำหรับ Validation support ต้องจับข้อมูลผิดได้ Query สำหรับ Evidence subgraph ต้องไม่ดึงข้อมูลมากเกินจำเป็น และ Query สำหรับ Monitoring ต้องรันเร็วพอที่จะใช้เป็นประจำ

การอ่านผลลัพธ์ของ SPARQL ต้องระวังด้วยว่าไม่มีผลลัพธ์ไม่ได้แปลว่าโลกไม่มีสิ่งนั้นเสมอไป อาจแปลว่า Graph ยังไม่มีข้อมูล Mapping ยังไม่ครอบคลุม หรือ Query เขียนแคบเกินไป เช่น หาก Query ถาม Project ที่มี Label ภาษาไทยแบบบังคับ ผลลัพธ์อาจหายไปเพียงเพราะ Label ภาษาไทยยังไม่ถูกกรอก ไม่ใช่เพราะ Project ไม่มีอยู่จริง จึงต้องใช้ **OPTIONAL** และ Validation Report ให้เหมาะสม ในระบบที่มี Reasoning ผลลัพธ์ของ Query ยังขึ้นกับ entailment regime ด้วย หาก Triple Store เปิด RDFS Reasoning Query ที่ถามหา **ex:Project** อาจคืน Resource ที่ไม่ได้ประกาศ type โดยตรง แต่ Infer จาก

Domain/Range หากปิด Reasoning Query เดียวกันอาจคืนผลลัพธ์น้อยลง ดังนั้น Query Catalog ควรระบุด้วยว่า Query คาดหวัง Reasoning แบบใด ไม่เช่นนั้นผลลัพธ์ระหว่าง environment อาจต่างกันโดยไม่มีใครเข้าใจ

อีกประเด็นหนึ่งคือการใช้ **LIMIT** เพื่อแก้ปัญหา performance แบบผิดวิธี หาก Query กว้างเกินไป การใส่ **LIMIT 100** อาจทำให้ Query เร็วขึ้น แต่ไม่ได้แก้ความหมายของ Query และอาจทำให้ผลลัพธ์ไม่ครบโดยไม่มีสัญญาณเตือน วิธีที่ดีกว่าคือออกแบบ Pattern ให้เฉพาะเจาะจงขึ้น ใช้ type หรือ Resource ที่รู้แน่เป็นจุดเริ่มต้น และตรวจ index หรือ store configuration เมื่อ Graph ใหญ่ขึ้น SPARQL จึงมีทั้งด้านภาษา ด้านตรรกะ และด้านวิศวกรรมระบบ Query ที่ดีต้องอ่านแล้วเข้าใจว่ากำลังถามอะไร ใช้ Vocabulary สอดคล้องกับ Ontology คืนผลลัพธ์ที่ตรวจได้ และทำงานได้ในเวลาที่เหมาะสม หากทำได้เช่นนี้ SPARQL จะเป็นสะพานสำคัญระหว่าง Ontology ที่เป็นแบบจำลองความหมาย กับ Application ที่ผู้ใช้สัมผัสจริง

4.14 SPARQL Update และวงจรชีวิตของ Named Graph

หัวข้อก่อนมุ่งตรวจ Query ที่อ่านข้อมูล แต่ Graph ในระบบจริงยังมีวงจรการนำเข้า แก้ไข และเผยแพร่ SPARQL Update จึงถูกพิจารณาร่วมกับ Staging Graph, Published Graph และ Versioning มากกว่ามองเป็นเพียงคำสั่งเพิ่มหรือลบ Triple

นอกจากการ Query แล้ว SPARQL ยังมีส่วนของ SPARQL Update สำหรับเพิ่ม ลบ หรือแก้ไขข้อมูลใน Graph (World Wide Web Consortium, 2013d) ในระบบทดลอง ผู้พัฒนาอาจโหลดไฟล์ Turtle เข้า Triple Store ด้วยมือ แต่ในระบบจริงข้อมูลจะถูกเพิ่มเป็นรอบ ๆ และต้องควบคุมบทบาทของแต่ละ Graph ว่าเป็น Staging Graph, Published Graph หรือ Provenance Graph การเข้าใจ SPARQL Update จึงช่วยให้ Pipeline มีโครงสร้างมากขึ้น ตัวอย่างเช่น เมื่อมีข้อมูลโครงการชุดใหม่ ระบบอาจโหลดเข้า Staging Graph ก่อน

```
PREFIX ex: <https://example.org/vocab/>
PREFIX proj: <https://example.org/resource/project/>
PREFIX tech: <https://example.org/resource/technology/>
PREFIX graph: <https://example.org/resource/graph/>

INSERT DATA {
 GRAPH graph:staging-2026-07-01 {
 proj:P001 a ex:Project ;
 ex:usesTechnology tech:SensorNet .
 }
}
```

หลังจากนั้นระบบรัน SHACL เพื่อตรวจ Graph นี้ หากผ่านจึงค่อย Publish โดยอาจ copy หรือ move Triple ไปยัง Graph ที่ผู้ใช้ปลายทางอ่าน เช่น **graph:projects-current** การทำเช่นนี้ช่วยให้ข้อมูลที่ยังไม่ผ่าน Validation ไม่ปนกับข้อมูลที่ระบบใช้ตอบคำถามจริง ในบางกรณี update ไม่ควรลบข้อมูลเดิมทิ้งทันที แต่ควรสร้าง Graph Version ใหม่ เช่น **graph:projects-2026-07-01** และให้ **graph:projects-current** ชี้หรือรวมข้อมูล Version ล่าสุด วิธีนี้ช่วยให้ระบบย้อนกลับไปดูข้อมูลเดิมได้ หาก Release ใหม่มีปัญหา การเก็บ Version ของ Graph จึงสัมพันธ์กับ Provenance และ Governance โดยตรง ตัวอย่างการลบและเพิ่มอย่างระมัดระวังคือการเปลี่ยนสถานะโครงการ

```
PREFIX ex: <https://example.org/vocab/>
PREFIX proj: <https://example.org/resource/project/>
PREFIX status: <https://example.org/resource/status/>
PREFIX graph: <https://example.org/resource/graph/>

DELETE {
 GRAPH graph:projects-current {
 proj:P001 ex:hasStatus ?oldStatus .
 }
}
INSERT {
 GRAPH graph:projects-current {
 proj:P001 ex:hasStatus status:Closed .
 }
}
WHERE {
 GRAPH graph:projects-current {
 proj:P001 ex:hasStatus ?oldStatus .
 }
}
```

Query นี้แสดงให้เห็นว่า update ควรระบุ Graph และ Pattern ให้ชัด หากเขียน update กว้างเกินไป อาจลบ status จากหลาย Graph หรือหลาย Project โดยไม่ตั้งใจ ดังนั้น SPARQL Update ควรถูกใช้ด้วย Policy และ test เช่นเดียวกับ SELECT Query ในระบบที่มี Provenance การ update ควรสร้าง Activity record ด้วย เช่นการเปลี่ยนสถานะเกิดจาก Activity ใด ใครเป็น Agent และใช้ Source ใด หาก update เปลี่ยนข้อมูลแต่ไม่บันทึก Provenance คำตอบของระบบจะบอกสถานะใหม่ได้ แต่ไม่สามารถอธิบายได้ว่าสถานะนั้นเปลี่ยนเมื่อไรและเพราะอะไร

สิ่งที่ควรหลีกเลี่ยงคือการแก้ข้อมูลใน Published Graph โดยตรงจากหลายช่องทางโดยไม่มี log เพราะเมื่อข้อมูลผิด ทีมจะไม่รู้ว่าเกิดจาก script ใด ผู้ใช้ใด หรือ Mapping run ใด แนวทางที่ดีกว่าคือให้ update ผ่าน Pipeline กลาง เช่น import, validate, Publish และ record Provenance ทุกครั้ง แม้ระบบจะมีข้อมูลไม่มากในช่วงเริ่มต้น การมีวินัยนี้ช่วยลดปัญหาเมื่อปริมาณข้อมูลเพิ่มขึ้น SPARQL Update จึงเป็นส่วนหนึ่งของการดูแล Knowledge Graph ที่มีการเพิ่ม แก้ ตรวจ และ Publish ข้อมูลเป็นรอบ หาก Query คือวิธีอ่าน Graph update ก็คือวิธีเปลี่ยน Graph และทั้งสองส่วนต้องอยู่ภายใต้ Ontology, Validation และ Governance เดียวกัน

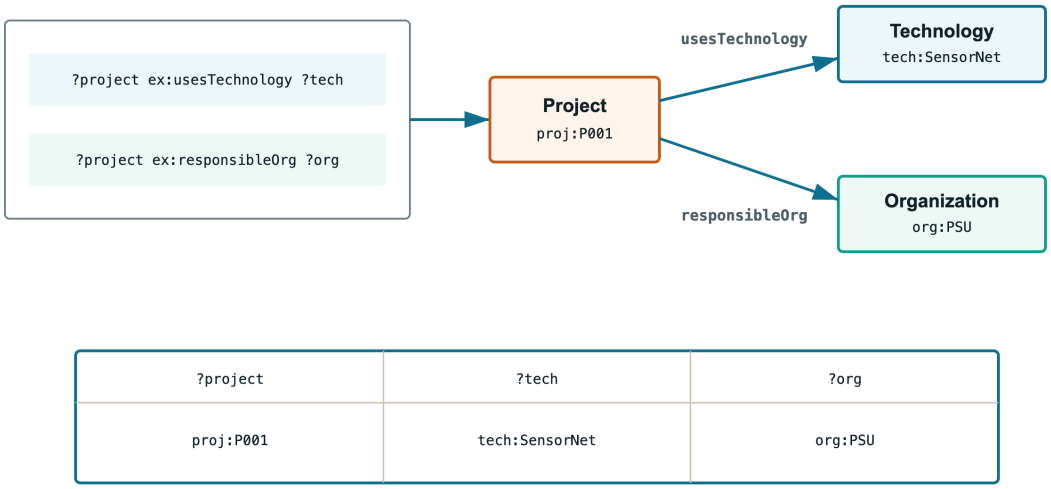
4.15 จากคำถามภาษาไทยสู่ Graph Pattern และกรณีศึกษา

เมื่อองค์ประกอบของ SPARQL ถูกพิจารณาครบแล้ว ขั้นสุดท้ายคือกลับไปยังจุดเริ่มต้นของผู้ใช้ ซึ่งอาจถามด้วยภาษาไทยโดยไม่รู้ชื่อ Class หรือ Property หัวข้อนี้จึงสรุปลำดับจากคำถามภาษาธรรมชาติไปสู่ Graph Pattern และใช้เป็นสะพานไปยังกรณีศึกษา Ontology ในบทถัดไป

หากมองการใช้ SPARQL จากมุมของผู้อ่านที่ไม่เคยเขียน Query มาก่อน จุดเริ่มต้นไม่ควรเป็นการจำรูปแบบคำสั่ง แต่เป็นการวาดเส้นทางคำตอบบนกราฟก่อน สมมุติคำถามว่า “โครงการใดใช้ SensorNet และมีองค์กรใดรับผิดชอบ” ต้องเห็นเส้นทางจาก Project ผ่าน **usesTechnology** ไปยัง SensorNet และจาก Project เดียวกันผ่าน **responsibleOrg** ไปยัง Organization เมื่อเส้นทางชัดแล้วจึงแทน Resource ที่ต้องการ

ค้นด้วยตัวแปร เช่น **?project** และ **?org** วิธีคิดนี้ทำให้ WHERE clause เป็นภาพของคำถาม ไม่ใช่ชุดสัญลักษณ์ที่ต้องท่องจำ

ตัวแปรใน SPARQL ควรถูกอธิบายด้วยบทบาทเช่นเดียวกับตัวแปรในสมการ หากเขียน Pattern $\langle ?p,$ **ex:usesTechnology**, **tech:SensorNet**$\rangle$ ตัวแปร $?p$ แทน Resource ใด ๆ ที่ทำให้ Triple Pattern นี้ตรงกับ Graph เมื่อเพิ่ม Pattern $\langle ?p,$ **ex:responsibleOrg**, $?o\rangle$ ตัวแปร $?p$ ตัวเดิมบังคับให้สอง Pattern กล่าวถึงโครงการเดียวกัน ส่วน $?o$ แทนองค์กรที่พบจากเส้นอีกด้าน การใช้ชื่อตัวแปรซ้ำจึงไม่ใช่เรื่องรูปแบบ แต่เป็นการระบุจุดเชื่อมของเงื่อนไข การจับค่าตัวแปรจากสอง Triple Patterns ที่มีตัวแปรร่วมกันแสดงในภาพที่ 4.2



ภาพที่ 4.2: การใช้ตัวแปรร่วมเป็นจุดเชื่อมและสร้าง binding ใน SPARQL

ภาพที่ 4.2 แสดงว่า **?project** ตัวเดียวกันในสอง Pattern ถูกแทนด้วย **proj:P001** แล้วจึงพาไปยังค่าของ **?tech** และ **?org** ในแถวผลลัพธ์เดียวกัน หากใช้ชื่อตัวแปร Project ต่างกันโดยไม่ตั้งใจ เงื่อนไขทั้งสองจะไม่ถูกบังคับให้กล่าวถึงโครงการเดียวกัน

เมื่อ Query มีหลาย Pattern ผลลัพธ์เกิดจากการรวมชุด binding ที่เข้ากันได้ หาก Pattern แรกคืนโครงการสามรายการ และ Pattern ที่สองมีองค์กรของเพียงสองโครงการ ผลลัพธ์ของการเชื่อมแบบบังคับจะเหลือสองรายการ ส่วน **OPTIONAL** อนุญาตให้โครงการที่ยังไม่มีองค์กรปรากฏต่อไปโดยปล่อยค่าตัวแปรว่าง ความต่างนี้ควรถูกเลือกจากความหมายของคำถาม หากถามหาเฉพาะโครงการที่มีผู้รับผิดชอบ การบังคับ Pattern เหมาะสม แต่ถ้าต้องการรายชื่อโครงการทั้งหมดพร้อมข้อมูลผู้รับผิดชอบเท่าที่มี **OPTIONAL** จะตรงเจตนากว่า

ผลลัพธ์ว่างก็ต้องถูกอ่านอย่างระมัดระวัง เพราะอาจเกิดได้อย่างน้อยสามกรณี คือ Graph ไม่มีข้อเท็จจริงนั้น Query ใช้คำศัพท์ไม่ตรงกับ Ontology หรือข้อมูลอยู่คนละ Named Graph ที่ Query ไม่ได้ระบุ การตรวจ Query จึงควรเริ่มจาก Pattern ที่มีเงื่อนไขเดียวและทราบว่าต้องพบผล แล้วเพิ่มเงื่อนไขทีละส่วน หากเพิ่ม FILTER แล้วผลหายทั้งหมด จะรู้ว่าปัญหาเกี่ยวข้องกับเงื่อนไขนั้น วิธีนี้ตรวจสาเหตุได้ง่ายกว่าการเขียน Query ยาวครั้งเดียวแล้วคาดเดาว่าทำไมไม่พบคำตอบ

สำหรับคำถามภาษาไทย ขั้นก่อน SPARQL มีความสำคัญเพิ่มขึ้น เพราะคำที่ผู้ใช้พิมพ์อาจเป็น Label ไม่ใช่ IRI เช่นคำว่า “มหาวิทยาลัยสงขลานครินทร์” ต้องถูก resolve ไปยัง Resource ขององค์กรก่อน ระบบอาจค้นผ่าน **rdfs:label**, alias หรือดัชนีข้อความ แล้วจึงนำ IRI ที่ยืนยันได้ไปเติมใน Query Template หากนำข้อความจากผู้ใช้ไปประกอบ Query โดยตรงโดยไม่แยก Identity ออกจาก Label ระบบอาจเลือก Entity ผิดหรือเปิดช่องให้ Query มีโครงสร้างไม่ปลอดภัย

Query Catalog จึงควรถูกมองเป็นส่วนหนึ่งของการออกแบบความหมาย ไม่ใช่เพียงที่เก็บโค้ด ตัว Query บอกว่าระบบคาดหวังให้ Class และ Property เชื่อมกันอย่างไร ตัวอย่างผลลัพธ์บอกว่าคำตอบควรมีรูปร่างแบบใด และ test Graph บอกว่ากรณีใดต้องคืนหรือไม่คืนผล เมื่อ Ontology เปลี่ยน ทีมสามารถรัน Catalog เพื่อดูผลกระทบได้ทันที SPARQL จึงทำหน้าที่ทั้งค้นข้อมูลและเปิดเผยว่าการออกแบบ Ontology รองรับคำถามจริงเพียงใด

ลำดับจากคำถามไปหลักฐานที่ควรรักษาไว้คือ คำถามของผู้ใช้ถูกแยกเป็น Entity และเงื่อนไข จากนั้นแปลงเงื่อนไขเป็น Graph Pattern และตรวจว่า Query ใช้ Vocabulary ที่มีจริง ผลลัพธ์ถูกดึงพร้อม Provenance และชั้นแสดงผลจึงค่อยเรียบเรียงเป็นข้อความ หากคำตอบผิด การแยกลำดับนี้ช่วยให้ตรวจได้ว่าผิดตั้งแต่การระบุ Entity การสราง Pattern ขอมูลใน Graph หรือการสรุปผล

เมื่อ Ontology ถูกออกแบบจากโดเมนจริง Query อาจมีเส้นทางต่างจากตัวอย่างโครงการอย่างมาก ระบบตรวจเหตุการณ์อาจต้องเชื่อม Event, Observation, Sensor, result และเวลา ขณะที่ระบบความรู้โครงการอาจต้องเชื่อม Project เดียวกับ phase, Technology, location และ result อย่างไรก็ตามหลักของ SPARQL ยังเหมือนเดิม คือใช้ตัวแปรร่วมบังคับให้เงื่อนไขกล่าวถึง Resource เดียวกัน และใช้ Graph Pattern แสดงเส้นทางของคำถาม

บทนี้จึงหยุดที่หลักของการสร้างและตรวจ Query ก่อน รายละเอียดว่า Ontology สองโดเมนวาง Node กับ Edge อย่างไรจะอธิบายในบทที่ 6 หลังจากผู้อ่านเห็นโครงสร้างนั้นแล้ว จะสามารถย้อนกลับมาอ่าน SPARQL ได้ชัดว่าคุณภาพของ Query ไม่ได้ขึ้นกับ Syntax เพียงอย่างเดียว แต่ขึ้นกับการตัดสินใจว่าแนวคิดใดเป็น Resource และ Property ใดเชื่อมเส้นทางไปยังคำตอบ

4.16 สรุป

บทนี้เริ่มจากการจับคู่ Triple Pattern แล้วค่อยเพิ่มชนิดผลลัพธ์ เงื่อนไข Evidence, Test และวงจรชีวิตของ Graph ภาพรวมดังกล่าวทำให้ SPARQL เป็นส่วนเชื่อมระหว่างความหมายใน Ontology กับคำถามที่ระบบต้องตอบ

SPARQL เป็นเครื่องมือที่ทำให้ RDF Graph ถูกถามและนำไปใช้ได้จริง หลักสำคัญคือการมองคำถามของผู้ใช้ให้เป็น Pattern ของความสัมพันธ์ใน Graph จากนั้นออกแบบ Query ให้ดึงคำตอบและหลักฐานออกมาอย่างตรวจสอบได้ เมื่อ SPARQL ถูกจัดเป็น Query Catalog ร่วมกับ Ontology, SHACL และ Provenance ระบบ Knowledge Graph จะไม่ใช่เพียงคลังข้อมูล แต่เป็นระบบความรู้ที่ตอบคำถามได้อย่างมีโครงสร้าง

เอกสารอ้างอิง

Angles, R., Arenas, M., Barceló, P., Hogan, A., Reutter, J., and Vrgoč, D. (2017). Foundations of modern query languages for graph databases. *ACM Computing Surveys*, 50(5):68:1–68:40.

Hogan, A., Blomqvist, E., Cochez, M., d'Amato, C., de Melo, G., Gutierrez, C., Kirrane, S., Gayo, J. E. L., Navigli, R., Neumaier, S., Ngomo, A.-C. N., Polleres, A., Rashid, S. M., Rula, A., Schmelzeisen, L., Sequeda, J., Staab, S., and Zimmermann, A. (2022). Knowledge graphs. *ACM Computing Surveys*, 54(4):1–37.

Mishra, D. S., Agarwal, A., Swathi, B. P., and Akshay, K. C. (2022). Natural language query formalization to SPARQL for querying knowledge bases using Rasa. *Progress in Artificial Intelligence*, 11(3):193–206.

Neumann, T. and Weikum, G. (2010). The RDF-3X engine for scalable management of RDF data. *The VLDB Journal*, 19(1):91–113.

Pérez, J., Arenas, M., and Gutierrez, C. (2006). Semantics and complexity of SPARQL. In *The Semantic Web – ISWC 2006*, pages 30–43. Springer.

Pérez, J., Arenas, M., and Gutierrez, C. (2009). Semantics and complexity of SPARQL. *ACM Transactions on Database Systems*, 34(3):1–45.

World Wide Web Consortium (2013a). SPARQL 1.1 overview. W3C Recommendation. Accessed June 13, 2026, from **https://www.w3.org/TR/sparql11-overview/**.

World Wide Web Consortium (2013b). SPARQL 1.1 protocol. W3C Recommendation. Accessed July 8, 2026, from **https://www.w3.org/TR/sparql11-protocol/**.

World Wide Web Consortium (2013c). SPARQL 1.1 query language. W3C Recommendation. Accessed June 24, 2026, from **https://www.w3.org/TR/sparql11-query/**.

World Wide Web Consortium (2013d). SPARQL 1.1 update. W3C Recommendation. Accessed June 22, 2026, from **https://www.w3.org/TR/sparql11-update/**.

World Wide Web Consortium (2026). SPARQL 1.2 query language. W3C Working Draft. Accessed October 18, 2026, from **https://www.w3.org/TR/2026/WD-sparql12-query-20261011/**.

5

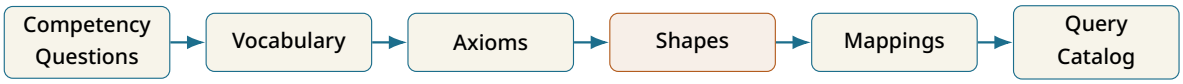
กระบวนการวิศวกรรม Ontology

หลังจากบทก่อนอธิบายหน้าที่ของ RDF, OWL, SPARQL และ SHACL แยกส่วนแล้ว บทนี้จะนำมาตรฐานเหล่านั้นมาใช้ในกระบวนการ Ontology Engineering ตั้งแต่การกำหนดคำถามของระบบ การออกแบบ การทดสอบ จนถึงการดูแล Ontology โดยใช้ตัวอย่างจากงาน Sensor และ Governance เพื่อสรุปเป็น Pattern ที่นำไปปรับใช้กับโดเมนอื่นได้

5.1 Ontology Engineering เริ่มจากคำถาม

บทก่อนแสดงว่า Query ที่ดีขึ้นอยู่กับ Class, Property และเส้นทางใน Graph ที่ออกแบบไว้แล้ว คำถามของผู้ใช้จึงไม่ใช่เพียงผลลัพธ์ปลายทาง แต่เป็นจุดเริ่มต้นของ Ontology Engineering หัวข้อนี้อธิบายเหตุผลที่ Competency Questions ควรมาก่อนการเปิดเครื่องมือและสร้าง Class

Ontology Engineering ไม่ควรเริ่มจากการเปิดเครื่องมือแล้วสร้าง Class จำนวนมาก แต่ควรเริ่มจากคำถามว่าระบบต้องรู้อะไร ต้องตอบคำถามใด และข้อมูลใดจำเป็นต่อการตัดสินใจนั้น แนวคิดนี้สอดคล้องกับหลักการพัฒนา Ontology ที่เน้น Competency Questions หรือคำถามที่ Ontology ต้องตอบได้ (Noy and McGuinness, 2001; Corcho et al., 2003) กระบวนการตั้งแต่คำถามไปจนถึง Mapping และ Query Catalog แสดงในภาพที่ 5.1



ภาพที่ 5.1: Ontology engineering method ในฐานะ pipeline ที่ตรวจสอบแต่ละขั้นได้

ภาพที่ 5.1 แสดงว่า Ontology Engineering เป็นกระบวนการต่อเนื่องที่เริ่มจากคำถาม ออกแบบ Vocabulary และ Axiom ตรวจด้วย Shape แล้วจึงนำไปสู่ Mapping และ Query Catalog ตัวอย่างเช่น หากระบบต้องตอบว่าโครงการใดใช้เทคโนโลยีใด ใครรับผิดชอบ และ Outcome ใดมีหลักฐานรองรับ Ontology ก็ต้องมี Resource และ Property ที่รองรับคำถามเหล่านี้ก่อน ไม่จำเป็นต้องสร้าง Class ทุกชนิดที่พบในเอกสารตั้งแต่เริ่มออกแบบ วิธีนี้ช่วยให้ Ontology มีขอบเขตที่ชัดเจนและลดการออกแบบเกินความจำเป็น

5.2 วงจรของการออกแบบ Ontology

การเริ่มจากคำถามช่วยกำหนดทิศทาง แต่ Ontology ไม่ได้เสร็จสมบูรณ์หลังออกแบบครั้งเดียว หัวข้อนี้จึงขยายจากจุดเริ่มต้นไปสู่วงจรที่เชื่อม Scope, Vocabulary, Mapping, Validation, Query และการทบทวนผลเข้าด้วยกัน

ในทางปฏิบัติ สามารถมอง Ontology Engineering เป็นวงจร ไม่ใช่เส้นตรง โดยเริ่มจาก Scope และ Competency Questions จากนั้นออกแบบ Vocabulary สร้าง RDF Mapping เพิ่ม Axiom เท่าที่จำเป็น ตรวจด้วย SHACL ทดสอบด้วย SPARQL และกลับไปปรับ Ontology ตามผลการใช้งานจริง วงจรนี้สามารถเขียนอย่างย่อได้ว่า

$$Q \rightarrow Vocab \rightarrow Mapping \rightarrow Graph \rightarrow Validation \rightarrow Query \rightarrow Review$$

ในสมการนี้ Q คือชุด Competency Questions หรือคำถามที่ระบบต้องตอบได้, *Vocab* คือคำศัพท์ของ Ontology, *Mapping* คือกติกาแปลงข้อมูลต้นทางเป็น RDF, *Graph* คือข้อมูล RDF ที่เกิดขึ้น, *Validation* คือการตรวจด้วย Shape หรือ Rule ที่กำหนดไว้, *Query* คือคำถามที่ใช้ทดสอบ Graph และ *Review* คือขั้นตอนที่มนุษย์ตรวจว่าคำถามเดิมถูกตอบได้จริงหรือไม่ หากตอบไม่ได้ ปัญหาอาจอยู่ที่ Ontology, Mapping, ข้อมูลต้นทาง หรือ Query ก็ได้ การมีวงจรเช่นนี้ช่วยให้ทีมไม่เข้าใจผิดว่า Ontology เสร็จสมบูรณ์ตั้งแต่ Version แรก

วงจรนี้ทำให้การพัฒนา Ontology ผูกอยู่กับสิ่งที่ตรวจได้ในทุกระยะ คำถามกำหนดขอบเขต Vocabulary และ Mapping กำหนดว่าข้อมูลจะถูกแทนอย่างไร ส่วน Validation, Query และ Review ตรวจว่าแบบจำลองรองรับการใช้งานจริงหรือไม่ เนื้อหาต่อไปจึงแยกพิจารณาการกำหนดคำถาม การควบคุมขอบเขต การทบทวน และการเผยแพร่ เพื่อให้แต่ละการตัดสินใจมีเหตุผลและมีหลักฐานรองรับ

5.3 จากคำถามไปสู่ขอบเขตด้วย Competency Questions

เมื่อมอง Ontology Engineering เป็นวงจร จุดตั้งต้นที่ต้องทำให้ตรวจสอบได้คือการเปลี่ยนความต้องการกว้าง ๆ ให้เป็น Competency Questions หัวข้อนี้จึงพิจารณาลักษณะของคำถามที่ดีและวิธีใช้คำถามกำหนดขอบเขตของ Class, Property และข้อมูลตัวอย่าง

Competency Question ที่ดีควรเป็นคำถามที่ระบบต้องตอบได้จริง ไม่ใช่คำถามกว้างจนตรวจไม่ได้ เช่น “ระบบต้องเข้าใจโครงการทั้งหมด” เป็นคำถามที่กว้างเกินไป แต่ “โครงการใดใช้เทคโนโลยีใด และมี Outcome ใดที่อ้างอิงรายงานฉบับใด” สามารถแปลงไปเป็น Ontology, RDF Mapping และ SPARQL Query ได้ คำถามที่พร้อมใช้กำหนดขอบเขตควรมีองค์ประกอบต่อไปนี้

- **Entity ที่ชัดเจน** เช่น Project, Technology, Outcome และ Report เพื่อให้ทราบว่าสิ่งใดต้องมี Identity ใน Graph
- **Relation ที่ชัดเจน** เช่น `usesTechnology`, `hasOutcome` และ `prov:wasDerivedFrom` เพื่อให้ทราบว่าต้องเชื่อม Resource ด้วยความหมายใด
- **เงื่อนไขหรือบริบทที่ตรวจได้** เช่นช่วงเวลา หน่วยงาน หรือประเภทของ Evidence เพื่อให้คำถามถูกแปลงเป็น Query ที่มีขอบเขตแน่นอน

องค์ประกอบข้างต้นจะเห็นชัดขึ้นเมื่อเปรียบเทียบคำถามที่ยังไม่พร้อมใช้กับคำถามที่ปรับแล้ว ตัวอย่างต่อไปนี้ไม่ได้เสนอว่ามีถ้อยคำที่ถูกต้องเพียงแบบเดียว แต่แสดงว่าการแก้คำถามควรทำให้ชนิดของคำตอบ ขอบเขต และหลักฐานตรวจสอบได้ชัดขึ้น

1. **คำถามกว้างจนไม่ทราบว่าจะยืนยันคำตอบอย่างไร**
    - **ยังไม่พร้อมใช้:** “ระบบต้องเข้าใจโครงการทั้งหมดหรือไม่”
    - **ปัญหา:** คำว่า “เข้าใจ” ไม่มีหน่วยของคำตอบ และคำว่า “ทั้งหมด” ไม่ได้บอกว่าครอบคลุม Project, Organization, Technology, Outcome หรือช่วงเวลาใด
    - **ปรับเป็น:** “ในปีงบประมาณ 2026 โครงการใดใช้ SensorNet และ Report ฉบับใดเป็น Evidence ของข้อมูลนั้น”
2. **คำถามใช้ถ้อยคำเชิงคุณค่าที่ยังไม่มีเกณฑ์รองรับ**
    - **ยังไม่พร้อมใช้:** “Technology ใดดีที่สุดสำหรับโครงการดิจิทัล”
    - **ปัญหา:** คำว่า “ดีที่สุด” ต้องมี Metric, กลุ่มเปรียบเทียบ และบริบทการตัดสิน หาก Ontology ไม่มีองค์ประกอบเหล่านี้ ระบบจะเสี่ยงตอบเกินข้อมูลที่มี
    - **ปรับเป็น:** “โครงการดิจิทัลใดใช้ Technology ประเภท Sensor Network และรายงาน Outcome ด้วย Metric ใด”
3. **คำถามถามความครบถ้วนของ Ontology โดยไม่ระบุ Use Case**
    - **ยังไม่พร้อมใช้:** “Ontology นี้ครบถ้วนหรือยัง”
    - **ปัญหา:** ไม่มี Ontology ใดครบถ้วนต่อทุกคำถาม ความครบถ้วนต้องประเมินเทียบกับขอบเขตและงานที่ต้องการใช้
    - **ปรับเป็น:** “สำหรับ Project P001 ระบบสามารถคืน Responsible Organization, Technology, Outcome และ Report ที่รองรับ Outcome ได้ครบหรือไม่”

ตัวอย่างแรกสามารถแตกเป็น Contract สำหรับการออกแบบและทดสอบได้ดังตารางที่ 5.1 การแตกคำถามเช่นนี้ช่วยให้ทีมไม่หยุดเพียงการเขียนประโยคที่ดูชัด แต่เห็นต่อไปว่าต้องสร้าง Artifact ใดเพื่อพิสูจน์ว่า Ontology รองรับคำถามแล้ว

ตารางที่ 5.1: การแตก Competency Question หนึ่งข้อเป็นข้อกำหนดที่ตรวจได้

องค์ประกอบ	สิ่งที่ต้องระบุ
Competency Question	ในปีงบประมาณ 2026 โครงการใดใช้ SensorNet และ Report ฉบับใดเป็น Evidence ของข้อมูลนั้น
หน่วยของคำตอบ	หนึ่งแถวประกอบด้วย Project, Technology และ Report ที่เชื่อมกันผ่าน Claim เดียวกัน
Identity ที่จำเป็น	Project, Technology, Claim และ Report ต้องมี IRI ที่อ้างซ้ำได้
Relation ที่จำเป็น	Project ใช้ Technology, Claim กล่าวถึง Project, Claim ยืนยัน Technology และ Claim มี Provenance ไปยัง Report
เงื่อนไข	Project อยู่ในปีงบประมาณ 2026 และ Technology ถูก Resolve เป็น SensorNet เดียวกัน
Expected Result	ชุดข้อมูลทดสอบต้องระบุล่วงหน้าว่าควรพบ Project และ Report ใด ไม่ใช่ตรวจเพียงว่า Query ทำงานโดยไม่ Error
เกณฑ์ผ่าน	Query คืนเฉพาะคำตอบที่มี Report รองรับ และสามารถตามจากผลลัพธ์กลับไปยัง Source Record ได้

Competency Question ควรเขียนด้วยภาษาของปัญหาก่อน ไม่จำเป็นต้องใส่ชื่อ Property เช่น **ex:usesTechnology** ตั้งแต่ประโยคแรก เพราะชื่อนั้นเป็นการตัดสินใจในขั้นออกแบบ Ontology อย่างไรก็ตาม เมื่อทีมยอมรับคำถามแล้ว ต้องสามารถ Map คำนามและความสัมพันธ์ในคำถามไปยัง Class, Property, Source Field และ Expected Result ได้ หาก Map ไม่ได้ คำถามอาจยังกว้างเกินไปหรือข้อมูลต้นทางยังไม่รองรับ

หากองค์ประกอบเหล่านี้ยังไม่ชัด Ontology ที่สร้างขึ้นก็มักจะไม่ชัดตามไปด้วย เพราะผู้ออกแบบยังไม่รู้ว่าคำศัพท์ใดจำเป็นและผลลัพธ์แบบใดใช้ยืนยันว่าแบบจำลองตอบโจทย์แล้ว

อีกคุณสมบัติหนึ่งที่มักถูกมองข้ามคือคำถามที่ดีควรบอกขอบเขตของคำตอบด้วย เช่นถามว่า “โครงการใดใช้เทคโนโลยี SensorNet ในปีงบประมาณ 2026” จะชัดกว่า “โครงการใดใช้เทคโนโลยี Sensor” เพราะมีทั้ง Resource ที่ต้องจับคู่ ช่วงเวลา และเงื่อนไขการกรองข้อมูล เมื่อคำถามมีขอบเขต ผู้ออกแบบจะเห็นทันทีว่าต้องมี Class หรือ Property ใดบ้าง เช่น Project, Technology, Fiscal Year และอาจต้องมี Provenance เพื่อรู้ว่าข้อมูลมาจากรายงานใด

คำถามบางข้อควรใช้ตรวจว่า Ontology ยังไม่รองรับอะไรด้วย เช่น “ระบบสามารถบอกได้หรือไม่ว่า Outcome นี้มีหลักฐานจากรายงานใด” หากตอบไม่ได้ อาจแปลว่าขาด Claim Resource หรือขาด Property ที่เชื่อม Claim กับ Report คำถามลักษณะนี้ช่วยระบุส่วนที่ Ontology ยังไม่รองรับโดยไม่ต้องเพิ่ม Class จากการเดา ทีมออกแบบจึงควรเก็บทั้งคำถามที่ตอบได้และคำถามที่ยังตอบไม่ได้ไว้ในเอกสาร เพื่อให้ Version ถัดไปมีเหตุผลในการขยาย

ในงานวิจัย การเก็บ Competency Questions เป็นส่วนหนึ่งของเอกสาร Ontology มีประโยชน์มาก เพราะผู้อ่านจะเห็นว่า Class และ Property ที่ออกแบบมีเหตุผลอะไร ไม่ใช่เกิดจากการรวมคำศัพท์ตามความรู้สึกเท่านั้น นอกจากนี้ยังช่วยให้ทบทวนได้ว่า Ontology ตอบคำถามเดิมได้ครบหรือไม่เมื่อมีการเปลี่ยน Version

5.4 Minimal Commitment และการควบคุม Scope

Competency Questions บอกสิ่งที่ Ontology ต้องรองรับ แต่ไม่ได้หมายความว่าต้องนิยามทุกสิ่งที่เกี่ยวข้องกับโดเมน Minimal Commitment จึงช่วยควบคุมไม่ให้แบบจำลองประกาศความหมายเกินหลักฐานและวัตถุประสงค์ของรุ่นปัจจุบัน

Minimal Commitment หมายถึงการไม่ประกาศความหมายมากเกินความจำเป็นในช่วงแรก หลักนี้สำคัญเพราะ Ontology ที่กำหนดข้อจำกัดมากกว่าที่โดเมนยืนยันได้อาจรองรับข้อมูลจริงไม่ได้ ตัวอย่างเช่น หากประกาศว่า Project ต้องมี Responsible Organization เพียงหนึ่งเดียว แต่ในโลกจริงมีหลายหน่วยงานร่วมรับผิดชอบ Ontology จะกลายเป็นอุปสรรคแทนที่จะช่วยจัดความหมาย การควบคุม Scope จึงควรใช้ Competency Questions เป็นตัวตั้ง หากคำถามของระบบยังไม่ต้องแยก Technology เป็น AI Model, AI Service และ AI Platform ก็อาจเริ่มด้วย Class **ex:Technology** ก่อน แล้วค่อยขยายเมื่อ Query ต้องการความละเอียดเพิ่ม การออกแบบเช่นนี้ทำให้ Ontology เติบโตไปพร้อมกับการใช้งานจริง

แต่ละโดเมนมี Scope ของตนเอง และ Scope ที่ดีเกิดจากโจทย์ของระบบ ไม่ใช่จากการเลียนแบบ Ontology อื่นโดยตรง Minimal Commitment ไม่ได้หมายถึง Ontology ต้องเล็กตลอดไป แต่หมายถึงการประกาศความหมายเท่าที่มีเหตุผลรองรับใน Version นั้น หากระบบเริ่มจากคำถามเรื่อง Project, Technology และ Report ก็ยังไม่จำเป็นต้องสร้าง Ontology ขนาดใหญ่ของทุกกระบวนการในองค์กร เมื่อมีคำถามใหม่ เช่นต้องการวิเคราะห์ Risk, Budget หรือ Stakeholder Role จึงค่อยขยาย Ontology ด้วยหลักฐานจาก Use Case จริง วิธีนี้ทำให้ Ontology เติบโตตามการใช้งาน ไม่ใช่โตจากความตั้งใจให้ครบตั้งแต่แรก

หลักนี้ยังช่วยป้องกันการใช้ OWL Axiom เกินจำเป็น หากยังไม่มีหลักฐานว่า Class สองชุดแยกจากกันจริงในโดเมน ก็ไม่ควรประกาศ Disjointness และหากยังยืนยันไม่ได้ว่า Property หนึ่งต้องมีค่าเดียวเสมอก็ไม่ควรกำหนด Cardinality ดังกล่าว การระบุเฉพาะความหมายที่ตรวจสอบได้ในรุ่นแรกช่วยลดความเสี่ยงของการนิยามผิด และเปิดให้เพิ่ม Axiom ภายหลังเมื่อมีข้อมูลรองรับ

5.5 การ Review, Documentation และ Release

เมื่อขอบเขตเริ่มคงที่ Ontology ต้องถูกตรวจทั้งความหมาย ตรรกะ และการใช้งานจริง หัวข้อนี้จึงเชื่อมการออกแบบเข้ากับ Review, Documentation และ Release ซึ่งทำให้การตัดสินใจของทีมตรวจย้อนกลับและนำกลับมาใช้ได้

การทบทวน Ontology ควรมองสามด้านร่วมกัน เพราะแต่ละด้านตรวจปัญหาคนละชนิด

- **Domain Review** ให้ผู้เชี่ยวชาญตรวจว่าคำศัพท์ ความสัมพันธ์ และตัวอย่างตรงกับความเข้าใจของโดเมนหรือไม่
- **Logical Review** ให้ Ontology Engineer ตรวจว่า Class Hierarchy, Axiom และ Restriction สอดคล้องกันและไม่สร้างผลอนุมานที่ผิดเจตนาหรือไม่
- **Application Review** ให้ผู้พัฒนาตรวจว่า Ontology รองรับ Mapping, Validation และ Query ที่ระบบต้องใช้จริงหรือไม่

หาก Review เฉพาะด้านโดเมน Ontology อาจอ่านถูกแต่ใช้กับระบบไม่ได้ หากตรวจเฉพาะตรรกะ Ontology อาจ Consistent แต่ไม่ตอบคำถามของผู้ใช้ และหากตรวจเฉพาะ Application ระบบอาจทำงาน

ได้เร็วแต่ความหมายไม่มั่นคง การมีสามมุมมองพร้อมกันจึงช่วยให้ Ontology เป็นทั้งงานวิชาการและงานวิศวกรรม

การทบทวนที่ดีควรใช้ Artifact จริง ไม่ใช่คุยจาก Diagram อย่างเดียว เช่นควรมี RDF Example หนึ่งชุด SPARQL Query ที่ตอบ Competency Questions และ Validation Report ตัวอย่าง เมื่อผู้เชี่ยวชาญโดเมนเห็น RDF Example จะตรวจได้ว่าความสัมพันธ์อ่านตรงกับงานจริงหรือไม่ เมื่อ Ontology Engineer เห็น Axiom จะตรวจ Consistency ได้ และเมื่อผู้พัฒนาระบบเห็น Query กับ Shape จะตรวจได้ว่าระบบนำไปใช้จริงได้หรือไม่ การใช้ Artifact ร่วมกันทำให้การทบทวนเป็นรูปธรรมและตรวจสอบได้ชัดเจนขึ้น

อีกเรื่องหนึ่งคือควรบันทึก Decision ของ Review ไว้เสมอ เช่นยอมรับ Property นี้เพราะตอบคำถามใด ปฏิเสธ Class นี้เพราะอยู่นอก Scope หรือเลื่อน Alignment นี้ไป Version ถัดไปเพราะยังไม่มีข้อมูลรองรับ บันทึกเหล่านี้ช่วยให้ Ontology ไม่สูญเสียเหตุผลเมื่อเวลาผ่านไป และช่วยให้ผู้อ่านงานวิจัยเห็นว่าการออกแบบเกิดจากกระบวนการที่มีเหตุผล ไม่ใช่เลือกตามความรู้สึกของบุคคลเพียงคนเดียว

5.6 Documentation ของ Ontology

ผลของ Review จะสูญหายได้ง่ายหากเหตุผลและขอบเขตของคำศัพท์ไม่ถูกบันทึก Documentation จึงไม่ได้เป็นงานที่ทำหลัง Ontology เสร็จ แต่เป็นส่วนหนึ่งของการรักษาความหมายให้ผู้พัฒนาและผู้เชี่ยวชาญโดเมนเข้าใจตรงกัน

Ontology ที่ไม่มี Documentation มักใช้ซ้ำยาก แม้ไฟล์ OWL จะถูกต้องก็ตาม Documentation ขั้นต่ำควรมีคำอธิบายของ Class และ Property, ตัวอย่าง RDF, Competency Questions, SPARQL Query ตัวอย่าง และข้อจำกัดของ Ontology เช่น สิ่งใดอยู่นอก Scope สิ่งใดยังไม่รองรับ และสิ่งใดเป็น Extension ในอนาคต สำหรับผู้อ่านภาษาไทย การมี Label และ Comment ภาษาไทยช่วยให้ผู้เชี่ยวชาญโดเมนเข้าร่วม Review ได้ดีขึ้น แต่ควรมี Label ภาษาอังกฤษควบคู่หาก Ontology ต้องเชื่อมกับมาตรฐานสากล การทำหลายภาษาไม่ควรถูกมองเป็นเรื่องตกแต่ง แต่เป็นส่วนหนึ่งของการทำ Ontology ให้ใช้ได้จริงในบริบทไทยและสากลพร้อมกัน

Documentation ควรบอกตัวอย่างการใช้ Property ที่อาจสับสนด้วย เช่น **ex:responsibleOrg**, **ex:partnerOrg** และ **ex:fundingOrg** อาจเชื่อม Project กับ Organization เหมือนกัน แต่ความหมายต่างกัน หาก Documentation อธิบายเพียงชื่อ Property โดยไม่มีตัวอย่าง ผู้ใช้จะมีโอกาสใช้ปนกันสูง การให้ตัวอย่าง Triple อย่างน้อยหนึ่งกรณีพร้อมคำอธิบายว่าใช้เมื่อไรและไม่ใช้เมื่อไรช่วยลดความคลุมเครือได้มาก ในงานที่ต้องเผยแพร่ Ontology สู่ผู้อื่น Documentation ยังควรระบุ Namespace, License, Version, Contact, Changelog และวิธี Cite Ontology ด้วย เพราะ Ontology ไม่ใช่ไฟล์เทคนิคอย่างเดียว แต่เป็นผลงานที่ผู้อื่นอาจนำไปอ้างอิง หากไม่มีข้อมูลเหล่านี้ Ontology อาจถูกใช้ซ้ำยาก แม้โครงสร้างภายในจะดีมากก็ตาม

5.7 Release Package ของ Ontology

Documentation อธิบายความหมายของ Ontology ส่วน Release Package ทำให้ไฟล์และ Artifact ที่เกี่ยวข้องถูกส่งต่อเป็นชุดเดียวกัน หัวข้อนี้จึงพิจารณาว่าการเผยแพร่ควรรวม Ontology, Shape, Mapping, Query, Example และ Version Information อย่างไร

เมื่อ Ontology พร้อมเผยแพร่ ควร Release เป็น Package ไม่ใช่ปล่อยไฟล์เดี่ยว Package หนึ่งควรประกอบด้วยไฟล์ Ontology, Shape, Mapping, Example Data, Query Catalog และ Changelog หากมีการเปลี่ยน Version ควรระบุว่าเพิ่ม Class ใด ลบ Property ใด หรือเปลี่ยน Domain/Range ใด เพราะการเปลี่ยนเล็กน้อยใน Ontology อาจกระทบข้อมูลจำนวนมาก สำหรับงานวิจัย Release Package ยังเป็นหลักฐานของคุณค่าที่ตรวจสอบได้ เพราะผู้อื่นสามารถตรวจว่า Ontology ตอบ Competency Questions อย่างไร ใช้กับข้อมูลตัวอย่างได้อย่างไร และมีข้อจำกัดอะไร การทำ Release ที่ดีจึงช่วยยกระดับ Ontology จาก Artifact ภายในทีมให้กลายเป็นผลงานที่ตรวจสอบและใช้ซ้ำได้

Release Package ควรมี Manifest ที่ระบุว่ามีไฟล์ใดบ้างและแต่ละไฟล์ทำหน้าที่อะไร เช่น Ontology Core, Ontology Extension, Alignment File, Shape File, Example Data และ Query Catalog หากไม่มี Manifest ผู้อ่านต้องเดาเองว่าไฟล์ใดคือ Entry Point และควรรันอะไรเพื่อเห็นผลลัพธ์ การมี Manifest ยังช่วยให้การทบทวนง่ายขึ้น เพราะ Reviewer สามารถตรวจตามลำดับที่ทีมออกแบบกำหนดไว้ ในระบบที่ใช้ Ontology จริง Release Package ควรมีตัวอย่างข้อมูลที่ตั้งใจให้ผ่าน Validation และตัวอย่างข้อมูลที่ตั้งใจให้ไม่ผ่าน Validation ด้วย ตัวอย่างที่ผ่านช่วยแสดงวิธีใช้ Ontology ที่ถูกต้อง ส่วนตัวอย่างที่ไม่ผ่านช่วยแสดงว่า Shape ตรวจอะไรได้บ้าง ทั้งสองอย่างทำให้คุณค่าของ Ontology ตรวจสอบได้ชัดกว่าไฟล์ Class/Property อย่างเดียว เพราะผู้อ่านเห็นทั้งความหมายและเงื่อนไขการใช้งาน

5.8 จากปัญหาเฉพาะโดเมนสู่ Pattern ที่ใช้ข้ามโดเมน

เมื่อ Ontology ถูกจัดเป็น Release Package แล้ว จะเห็นชัดว่าส่วนใดเป็นคำศัพท์เฉพาะโดเมนและส่วนใดเป็น Pattern ที่นำกลับมาใช้ได้ หัวข้อนี้จึงแยกโครงสร้างทั่วไปออกจากรายละเอียดของ Multimedia Sensor Networks และโครงการเปลี่ยนผ่านสู่ดิจิทัล โดยไม่อ้างว่า Pattern หนึ่งเหมาะกับทุกโดเมน

เมื่องานวิจัยหนึ่งเริ่มจากโดเมนเฉพาะ เช่น Multimedia Sensor Network สิ่งแรกที่ควรทำไม่ใช่การสร้าง Ontology ให้ใหญ่ที่สุด แต่คือการแยกให้ชัดว่าโดเมนนั้นมีปัญหาอะไรที่ Ontology ช่วยได้ ในกรณี Sensor Network ปัญหาไม่ได้อยู่เพียงการรู้ว่า Sensor ชื่ออะไร แต่อยู่ที่การเชื่อม Sensor, Observation, Event, Media Data, Time และ Context เข้าด้วยกัน เพื่อให้ระบบประมวลผลเหตุการณ์ได้อย่างมีความหมาย

หากมองจากวิธีวิศวกรรม Ontology ปัญหานี้ให้ Pattern สำคัญสามแบบ

- **ทำ Event ให้เป็น Resource** แทนการเก็บเป็นข้อความใน Log เพราะ Event ต้องถูกอ้างถึงซ้ำ มีเวลาเชื่อมกับ Sensor หรือ Observation และอาจมี Evidence หลายชนิดรองรับ
- **แยก Observation ออกจาก Result** เพราะกิจกรรมการสังเกตหนึ่งครั้งกับค่าหรือข้อมูลที่ได้จากการสังเกตไม่ใช่สิ่งเดียวกัน การแยกสองส่วนทำให้บันทึกอุปกรณ์ เวลา และวิธีสังเกตได้โดยไม่ปนกับผลลัพธ์
- **เชื่อม Context เข้ากับ Event** เพื่อรักษาข้อมูลเวลา สถานที่ กิจกรรม และสถานการณ์ที่เกี่ยวข้องกับเหตุการณ์ไว้ในการประมวลผล

Pattern เหล่านี้ไม่จำเป็นต้องใช้เฉพาะ Sensor Network เท่านั้น หากย้ายมาที่ระบบโครงการดิจิทัล อาจเปลี่ยน Event เป็น Project Activity เปลี่ยน Observation เป็น Progress Record เปลี่ยน Media Evidence เป็น Report หรือ Document และเปลี่ยน Context เป็น Phase, Organization หรือ Policy Environment โครงสร้างโดเมนเปลี่ยน แต่หลักคิดเรื่อง Resource, Relation, Evidence และ Context ยังใช้ต่อได้ โครงการจึงไม่ควรถูกเก็บเป็นแถวข้อมูลอย่างเดียว แต่ควรเป็น Resource ที่เชื่อมกับ Stakeholder,

Technology, Risk, KPI, Outcome และ Evidence หาก Project Outcome ถูกใช้ในการรายงาน ทีมออกแบบ Ontology ต้องถามต่อว่า Outcome นั้นมาจาก Report ใด เกิดใน Phase ใด และมีผู้รับผิดชอบคือใคร คำถามเหล่านี้ทำให้ Ontology ไม่ใช่เพียง Class Diagram แต่เป็นโครงสร้างของการอธิบายและตรวจสอบความรู้

เมื่อต้องออกแบบ Ontology ใหม่ ทีมงานจึงควรเริ่มจาก Competency Questions ที่เจาะจง เช่น

- โครงการใดใช้เทคโนโลยีใดและมีหน่วยงานใดรับผิดชอบ
- Outcome ใดของโครงการมีหลักฐานจากรายงานฉบับใด
- ความเสี่ยงใดสัมพันธ์กับ Capability หรือ Phase ใดของโครงการ
- ข้อมูลใดผ่าน Validation แล้วและข้อมูลใดยังขาด Provenance

คำถามเหล่านี้ช่วยบอกว่า Ontology ต้องมี Resource และ Relation ใดบ้าง หากไม่มีคำถามเรื่อง Risk ก็อาจยังไม่จำเป็นต้องสร้าง Risk Hierarchy ใน Version แรก หากไม่มีคำถามเรื่อง Capability ก็อาจยังไม่ต้องแยก Capability เป็นหลายระดับ การออกแบบลักษณะนี้ช่วยให้ Ontology เติบโตจากการใช้งานจริง ไม่ใช่เติบโตจากความต้องการให้ Schema ดูครบทุกอย่าง หลังได้คำถามแล้ว ขั้นต่อไปคือสร้าง Conceptual Sketch ที่คนในทีมอ่านเข้าใจ อาจเริ่มจากกลุ่ม Resource หลัก เช่น Project, Organization, Technology, Report, OutcomeClaim และ Risk จากนั้นเขียน Relation หลัก เช่น responsibleOrg, usesTechnology, wasDerivedFrom, hasRisk และ hasOutcome เมื่อ Sketch นี้ชัดจึงค่อยแปลงเป็น RDF/RDFS และพิจารณาว่า Class หรือ Property ใดต้องใช้ OWL เพิ่มเติม

ในขั้น Review ผู้เชี่ยวชาญโดเมนควรถามว่า Relation เหล่านี้ตรงกับภาษาของงานหรือไม่ ผู้เชี่ยวชาญ Ontology ควรถามว่า Hierarchy และ Axiom สอดคล้องหรือไม่ และผู้พัฒนาควรถามว่า Mapping และ Query ทำได้จริงหรือไม่ หากทั้งสามกลุ่มตอบคนละทาง Ontology ยังไม่พร้อม Release เพราะอาจอ่านถูกทางวิชาการแต่ใช้ในระบบไม่ได้ หรือใช้ในระบบได้แต่ความหมายไม่มั่นคง ตัวอย่างปัญหาที่มักพบคือ Class หนึ่งถูกสร้างขึ้นเพราะคำในเอกสารฟังดูสำคัญ แต่ไม่มี Query ใดใช้ Class นั้น ไม่มี Shape ใดตรวจ และไม่มีข้อมูลต้นทางที่ Map เข้ามาได้จริง Class ดังกล่าวอาจยังไม่ควรอยู่ใน Version แรก ในทางกลับกัน Property บางตัวอาจดูธรรมดา เช่น **prov:wasDerivedFrom** แต่มีความสำคัญมาก เพราะทำให้ Claim เชื่อมกับหลักฐานได้และทำให้ระบบตอบคำถามอย่างรับผิดชอบได้

การทำ Ontology Engineering ที่ดีจึงต้องยอมรับว่าบางส่วนของ Ontology เป็นความรู้โดเมน บางส่วนเป็น Pattern ข้ามโดเมน และบางส่วนเป็นข้อกำหนดของระบบงาน เช่น Validation และ API ไม่ควรนำทุกอย่างมาปนกันโดยไม่มีขอบเขต เพราะจะทำให้ Ontology อ่านยากและดูแลยาก การแยกชั้นเหล่านี้ไม่ได้ทำให้ Ontology แยกขาดจากกัน แต่ทำให้แต่ละชั้นมีหน้าที่ชัดเจน สุดท้าย Release Package ควรบอกสิ่งที่ Ontology ทำได้และทำไม่ได้อย่างตรงไปตรงมา เช่น Version แรกอาจรองรับ Project, Technology, Organization และ Evidence แต่ยังไม่รองรับ Budget Model หรือ Impact Assessment เต็มรูปแบบ การบอกข้อจำกัดเช่นนี้ไม่ใช่ข้อเสียของงาน แต่เป็นความซื่อสัตย์ทางวิชาการและช่วยให้ผู้อื่นต่อยอดได้ถูกทิศทาง

5.9 การประเมิน Ontology ในฐานะผลงานทางวิชาการและระบบใช้งาน

การระบุ Pattern ที่นำกลับมาใช้ได้ยังไม่เพียงพอที่จะยืนยันคุณภาพของ Ontology การประเมินจึงต้องพิจารณาทั้งความสอดคล้องเชิงตรรกะ ความสามารถในการตอบ Competency Questions ความเหมาะสมต่อข้อมูล และความชัดเจนของ Artifact ที่เผยแพร่

เมื่อ Ontology ถูกสร้างเสร็จ คำถามที่สำคัญคือจะรู้ได้อย่างไรว่า Ontology นั้นเหมาะกับงาน คำตอบไม่ควรอยู่ที่จำนวน Class หรือ Property เพียงอย่างเดียว เพราะ Ontology ที่มีคำศัพท์จำนวนมากอาจไม่ตอบคำถามของระบบ ขณะที่ Ontology ที่กำหนดขอบเขตชัดเจนและมีข้อมูลจริง Query และ Validation ที่ตรวจได้อาจมีคุณค่ามากกว่า การประเมินจึงควรพิจารณาอย่างน้อยเจ็ดด้าน

- **Coverage** ตรวจว่า Ontology ครอบคลุม Competency Questions ที่ตั้งไว้หรือไม่ หากคำถามต้องเชื่อม Project, Outcome Claim และ Report แต่ Ontology ไม่มี Relation ไปยัง Report คำถามนั้นย่อมตอบไม่ได้ แม้จะมี Class อื่นจำนวนมาก
- **Clarity** ตรวจว่า Class และ Property มี Label, Comment และตัวอย่างที่อ่านตรงกันหรือไม่ ในบริบทไทยควรมีคำอธิบายภาษาไทยเพื่อให้ผู้เชี่ยวชาญโดเมน Review ได้ และมี Label ภาษาอังกฤษควบคู่เมื่อจำเป็นต้องเชื่อมมาตรฐานสากล
- **Logical Consistency** ใช้ Reasoner ตรวจ Class ที่ Unsatisfiable, Axiom ที่ให้ผลอนุมานผิดเจตนา และ Disjointness ที่ไม่สอดคล้องกับข้อมูลจริง เพื่อยืนยันว่า Ontology เป็นแบบจำลองเชิงตรรกะ ไม่ใช่เพียงแผนภาพ
- **Data fitness** ตรวจว่าข้อมูลจริงสามารถ Map เข้ากับ Ontology ได้หรือไม่ หากข้อมูลต้นทางไม่มี Field ที่รองรับหรือ Extraction ทำไม่ได้ แบบจำลองที่ละเอียดมากอาจยังไม่เหมาะกับระบบปัจจุบัน
- **Query fitness** ใช้ SPARQL Query และ Expected Results แสดงว่า Ontology ตอบคำถามหลักได้จริง Query เหล่านี้ควรอยู่ใน Release Package เพื่อให้ผู้อื่นตรวจลำดับตั้งแต่คำถาม Graph Pattern จนถึงผลลัพธ์ได้
- **Validation readiness** จัดเตรียม SHACL Shapes สำหรับเงื่อนไขขั้นต่ำของข้อมูล เช่น Project ต้องมี Label, Organization และ Status หรือ Outcome Claim ต้องมี Provenance เพื่อไม่ให้ข้อมูลที่ผิดหรือขาดเข้าสู่ Graph โดยไม่มีรายงาน
- **Reusability** ตรวจ Documentation, Namespace Policy, Alignment, License, Example Data และ Changelog หากเผยแพร่เพียงไฟล์ Ontology โดยไม่มีตัวอย่าง Query หรือคำอธิบาย ผู้อื่นจะใช้ซ้ำได้ยาก แม้แนวคิดภายในจะดี

ในงานวิจัย การเขียนส่วน Evaluation ของ Ontology จึงควรตอบมากกว่า “ ได้สร้าง Ontology แล้ว” แต่ควรบอกว่า Ontology ตอบ Competency Questions ใด ใช้กับข้อมูลตัวอย่างได้อย่างไร ผ่าน Reasoner หรือไม่ มี Shape ตรวจอะไรบ้าง Query หลักคืออะไร และมี Alignment กับมาตรฐานใด การอธิบายเช่นนี้ทำให้ขอบเขตของผลงานชัดและไม่กล่าวเกินหลักฐาน เพราะไม่ต้องประกาศว่า Ontology ครอบคลุมทุกอย่าง แต่แสดงให้เห็นว่า Ontology ทำงานตามบทบาทที่ตั้งใจไว้ได้อย่างไร

สำหรับผู้อ่านที่ต้องสร้าง Ontology ของตนเอง เกณฑ์เหล่านี้สามารถใช้เป็น Checklist ระหว่างทำงานได้ หากยังไม่มี Competency Questions ให้กลับไปกำหนดคำถาม หากมีคำถามแต่ไม่มี Query ให้เขียน

Query หากมี Query แต่ข้อมูล Map ไม่ได้ ให้กลับไปดู Mapping หรือ Scope หากข้อมูล Map ได้แต่ไม่มี Validation ให้เพิ่ม Shape หากทุกอย่างทำได้แล้วจึงค่อยพิจารณา Alignment และ Release Package วิธีนี้ทำให้ Ontology Engineering เดินหน้าอย่างมีเหตุผลมากกว่าการเพิ่ม Class ไปเรื่อย ๆ

5.10 รูปแบบการออกแบบที่ใช้บ่อยในงาน Ontology

เกณฑ์ประเมินจากหัวข้อก่อนช่วยบอกว่า Ontology ทำหน้าที่ได้เพียงใด ส่วน Design Pattern ช่วยอธิบายว่าโครงสร้างใดทำให้หน้าที่นั้นเกิดขึ้น หัวข้อนี้จึงรวบรวม Pattern ที่พบซ้ำ พร้อมระบุบริบทและข้อจำกัดของการนำไปใช้

เมื่อทำ Ontology หลายโดเมน จะพบว่า Pattern บางอย่างปรากฏซ้ำอยู่เสมอ คำว่า Pattern ในที่นี้ไม่ได้หมายถึงสูตรสำเร็จที่ต้องใช้เหมือนกันทุกงาน แต่หมายถึงรูปแบบการตัดสินใจที่ใช้แก้ปัญหาเดิมซ้ำ ๆ เช่นจะทำสิ่งใดให้เป็น Resource จะผูก Claim กับหลักฐานอย่างไร จะบันทึกบทบาทของคนและองค์กรอย่างไร หรือจะกำหนดขอบเขตระหว่าง Ontology เฉพาะโดเมนกับมาตรฐานภายนอกอย่างไร จุดสำคัญคือ Pattern ต้องช่วยให้ผู้ออกแบบคิดเป็นระบบ ไม่ใช่เพิ่ม Class ตามคำที่เห็นในเอกสารไปเรื่อย ๆ

- **Entity หรือ Identity Pattern** กำหนด Resource ให้สิ่งที่ต้องอ้างซ้ำ เช่น Project, Organization, Sensor, Technology, Report และ Person หากสิ่งหนึ่งต้องถูก Query, Link, Review หรือทำ Provenance สิ่งนั้นควรมี IRI ของตนเอง ส่วนชื่อที่มนุษย์อ่านควรเป็น Label หรือ Literal ที่เปลี่ยนภาษาได้ ไม่ควรใช้แทน Identity หลัก
- **Process-Claim หรือ Claim-Evidence Pattern** ใช้เมื่อข้อมูลหนึ่งเป็นคำกล่าวอ้างที่เกิดจากกระบวนการและต้องมีหลักฐาน เช่น Outcome ของโครงการ เหตุการณ์ที่ตรวจพบ หรือผลประเมิน KPI แทนที่จะเขียนเพียงว่า Project มี Outcome อาจสร้าง Claim เป็น Resource แล้วเชื่อมกับ Process, Outcome และ Report เพื่อให้บันทึก Provenance, Reviewer, Version หรือสถานะการอนุมัติได้ ตัวอย่างเช่น **ex:OutcomeClaim001** อาจเชื่อมกับ **proj:P001**, **ex:ImproveServiceQuality** และ **report:R2026-01** หากภายหลังมีรายงานใหม่หรือ Reviewer ไม่เห็นด้วย ระบบยังจัดการ Claim เดิมได้โดยไม่ต้องลบ Graph ทั้งชุด
- **Event-Observation Pattern** แยก Observation ซึ่งเป็นกิจกรรมการเก็บข้อมูล Result ซึ่งเป็นค่าหรือข้อมูลที่ได้ และ Event ซึ่งเป็นสิ่งที่ระบบตีความจาก Result การแยกสามส่วนช่วยอธิบายได้ว่าค่าใดเกิดจากกระบวนการใด เวลาใด อุปกรณ์ใด และเหตุการณ์ที่สรุปมีบริบทอะไร โครงสร้างนี้ช่วยไม่ให้ Event, Media Processing และ Result ถูกปนกันจน Query หรือ Alignment ทำได้ยาก
- **Role-Agent Pattern** ใช้เมื่อคนหรือองค์กรหนึ่งมีหลายบทบาท เช่นเจ้าของโครงการ ผู้ตรวจรายงาน ผู้ให้บริการ หรือผู้ได้รับผลกระทบ หาก Role มีเวลา ขอบเขต หรือเงื่อนไข การแยก Role เป็น Resource หรืออย่างน้อยแยก Relation ให้ชัดจะรองรับสถานการณ์ได้ดีกว่า เช่นองค์กรเดียวกันอาจเป็น Funder ของโครงการหนึ่งและ Evaluator ของอีกโครงการหนึ่ง
- **Status-lifecycle Pattern** ใช้กับสิ่งที่มีสถานะเปลี่ยนตามเวลา เช่น Project, Report, Claim หรือ Validation Run การเก็บเฉพาะสถานะปัจจุบันอาจพอสำหรับระบบเล็ก แต่ระบบที่ต้อง Audit ควรเก็บ History หรือ Activity ที่ทำให้สถานะเปลี่ยน เพื่อให้ทราบว่าเปลี่ยนจากอะไร เมื่อไร และโดยใคร

- **Controlled Vocabulary Pattern** ใช้กับค่าที่ควรอยู่ในชุดที่กำหนด เช่นสถานะโครงการ ประเภทเทคโนโลยี ระดับความเสี่ยง หรือประเภทหลักฐาน ค่าเหล่านี้อาจสร้างเป็น Resource แทน String เพื่อให้มี Label หลายภาษา มี Hierarchy และ Query ได้สม่ำเสมอ มิฉะนั้นค่าที่หมายถึงสิ่งเดียวกันอาจแตกเป็น “เสร็จแล้ว”, “Completed”, “Complete” และ “Done”
- **Alignment Boundary Pattern** กำหนดว่าคำใดอยู่ใน Ontology เฉพาะโดเมน คำใด Reuse จากมาตรฐาน และคำใด Align กับ Ontology ภายนอก หากใช้ SSN/SOSA สำหรับ Sensor และใช้ Vocabulary เฉพาะสำหรับ Multimedia Event ต้องระบุให้ชัดว่าเชื่อมกันตรงไหน และเป็น Subclass, Equivalence หรือเพียงความสัมพันธ์แบบ Related ขอบเขตนี้ทำให้เห็นตำแหน่งของ Ontology เฉพาะโดเมนใน Ecosystem ที่กว้างกว่า

Pattern เหล่านี้ไม่ใช่กฎตายตัว แต่เป็นจุดเริ่มต้นของการออกแบบ หากเจอปัญหาใหม่ ผู้เขียน Ontology ควรถามว่าเป็นกรณีของ Pattern ใดหรือไม่ หากใช่ ก็สามารถนำโครงสร้างเดิมมาปรับได้ หากไม่ใช่ อาจต้องสร้าง Pattern ใหม่พร้อม Rationale และตัวอย่าง RDF การทำ Catalog ของ Pattern ช่วยให้ทีมเรียนรู้จากงานเดิมและไม่ต้องออกแบบใหม่ทุกครั้ง การใช้ Pattern ควรเริ่มจากปัญหาที่เจอจริง เช่นถ้าระบบเริ่มมีชื่อองค์กรซ้ำหลายแบบ ควรกลับไปใช้ Entity หรือ Identity Pattern ถ้าคำตอบของระบบเริ่มมี Claim ที่ตรวจที่มาไม่ได้ ควรเพิ่ม Claim-Evidence Pattern ถ้าข้อมูลเริ่มมีเหตุการณ์หรือการสังเกตที่ปนกับผลลัพธ์ ควรแยก Event-Observation Pattern วิธีนี้ทำให้ Pattern เป็นเครื่องมือแก้ปัญหา ไม่ใช่รายการคำศัพท์ที่ใส่เพิ่มเพื่อให้ Ontology ดูซับซ้อนขึ้น

สำหรับงานวิจัย การอธิบาย Pattern ยังช่วยให้ขอบเขตของผลงานชัดเจนขึ้น แทนที่จะกล่าวว่า Ontology ครอบคลุมโดเมนทั้งหมด สามารถระบุได้ว่า Ontology นี้ใช้ Pattern สำหรับ Identity, Claim-Evidence, Event-Observation หรือ Alignment Boundary ในบริบทใด และแสดงตัวอย่างว่าช่วยตอบ Competency Questions ใด การเขียนแบบนี้ทำให้ผลงานเป็นวิศวกรรมความหมายที่ตรวจได้ มากกว่าการประกาศความสมบูรณ์ของ Ontology โดยไม่มีหลักฐาน

5.11 บันทึกการออกแบบสำหรับผู้เริ่มต้นทำ Ontology

Pattern หลายแบบอาจทำให้ผู้เริ่มต้นรู้สึกว่าต้องออกแบบทั้งระบบให้เสร็จก่อนลงมือ หัวข้อนี้จึงย่อกระบวนการให้เป็นบันทึกการตัดสินใจที่เริ่มจากคำถามหนึ่งข้อ ตัวอย่างข้อมูลขนาดเล็ก และ Query ที่ตรวจผลได้ แล้วค่อยขยายเมื่อมีหลักฐานเพิ่ม

ผู้เริ่มต้นไม่จำเป็นต้องวาด Ontology ให้ครบก่อนจึงเริ่มเขียน RDF ได้ ในงานจริงสามารถเลือกคำถามหนึ่งข้อ ใช้ข้อมูลตัวอย่างที่ตรวจผลได้ เขียน RDF ให้ตอบคำถามนั้น แล้วค่อยเพิ่ม Class, Property และ Shape ในรอบถัดไป บันทึกต่อไปนี้ช่วยให้การเริ่มต้นยังคงอยู่ในขอบเขตที่อธิบายและตรวจสอบได้

- **เริ่มจากขอบเขตที่ตอบคำถามได้** หากรุ่นแรกมีเพียง Project, Organization, Technology, Report และ OutcomeClaim แต่ตอบคำถามหลักได้จริง ย่อมมีหลักฐานการใช้งานชัดกว่า Ontology ที่มีคำศัพท์จำนวนมากแต่ยังไม่มีข้อมูลหรือ Query รองรับ
- **ใช้ OWL Axiom เมื่อทีมอธิบายผลได้** Axiom ที่ไม่มีใครอ่านกลับเป็นภาษาธรรมชาติหรือแสดงผลที่คาดหวังได้จะกลายเป็นภาระเมื่อ Reasoner ให้ผลที่ไม่ตรงเจตนา

- **เขียนตัวอย่าง RDF ควบคู่กับทุก Class และ Property** หาก Property หนึ่งไม่มีตัวอย่างการใช้ อาจแปลว่ายังไม่จำเป็นหรือยังนิยามไม่ชัด
- **ให้ผู้เชี่ยวชาญโดเมนอ่าน Label และ Comment ก่อนดู Diagram** คำอธิบายภาษาคนมักเผยปัญหาความหมายได้เร็วกว่าการเริ่มจากโครงสร้างเชิงเทคนิค
- **เก็บคำถามที่ Ontology ยังตอบไม่ได้ไว้ด้วย** ไม่จำเป็นต้องแก้ทุกอย่างในรุ่นแรก แต่ต้องระบุให้ได้ว่าเรื่องใดอยู่นอก Scope เช่น Budget, Impact Assessment หรือ Alignment บางส่วน การบอกขอบเขตทำให้งานน่าเชื่อถือกว่าการกล่าวว่ารองรับทุกอย่าง
- **มอง Ontology Engineering เป็นงานวนรอบ** เมื่อข้อมูลจริงเข้ามา Query ถูกใช้ และผู้ใช้ตั้งคำถามใหม่ Ontology ย่อมต้องปรับ Versioning, Changelog, Test Graph และ Query Catalog จึงเป็นเครื่องมือพื้นฐานของการดูแล ไม่ใช่งานเสริมหลังส่งมอบ

เมื่อมองบันทึกทั้งหกข้อนี้ร่วมกัน Competency Question ทำหน้าที่กำหนดสิ่งที่ Ontology ต้องช่วยตอบ ตัวอย่างข้อมูลทำให้คำถามจับต้องได้ Vocabulary และ Axiom แทนความหมายอย่างเป็นระบบ ส่วน Query และ Shape ตรวจว่าแบบจำลองรองรับทั้งการค้นคืนและคุณภาพข้อมูลจริงหรือไม่ ลำดับนี้ช่วยให้การออกแบบเริ่มจากภารกิจของระบบและหลักฐานที่ใช้ตรวจความสำเร็จ แทนการเริ่มจากรายการ Class ที่ทีมงานนึกออก

5.12 สรุป

บทนี้เชื่อมมาตรฐานจากบทก่อนให้เป็นกระบวนการ Ontology Engineering ตั้งแต่ Competency Questions ไปจนถึง Review, Release และ Evaluation ภาพรวมดังกล่าวจะถูกนำไปใช้กับ MSSN-Onto และ Dx-Onto ในบทถัดไป เพื่อแสดงว่ากระบวนการเดียวกันนำไปสู่การตัดสินใจที่ต่างกันได้อย่างไร

วิธีวิศวกรรม Ontology ที่ดีต้องเริ่มจากปัญหาและคำถาม ไม่ใช่เริ่มจาก Class Diagram เพียงอย่างเดียว ผู้ออกแบบต้องกำหนดสิ่งสำคัญเป็น Resource ระบุความสัมพันธ์ให้ชัด เพิ่ม Axiom เท่าที่อธิบายได้ ทดสอบด้วยข้อมูลและ Query จริง แล้วเผยแพร่พร้อม Artifact ที่ผู้อื่นตรวจและใช้ซ้ำได้ บทถัดไปจะนำหลักเหล่านี้ไปอ่านผ่าน MSSN-Onto และ Dx-Onto เพื่อให้เห็นว่าข้อมูลและคำถามคนละโดเมนนำไปสู่การตัดสินใจออกแบบที่ต่างกันอย่างไร

เอกสารอ้างอิง

Corcho, O., Fernández-López, M., and Gómez-Pérez, A. (2003). Methodologies, tools and languages for building ontologies. where is their meeting point? *Data & Knowledge Engineering*, 46(1):41–64.

Gruber, T. R. (1993). A translation approach to portable ontology specifications. *Knowledge Acquisition*, 5(2):199–220.

Guarino, N. (1998). Formal ontology and information systems. In *Proceedings of FOIS'98*, pages 3–15. IOS Press.

Hogan, A., Blomqvist, E., Cochez, M., d'Amato, C., de Melo, G., Gutierrez, C., Kirrane, S., Gayo, J. E. L., Navigli, R., Neumaier, S., Ngomo, A.-C. N., Polleres, A., Rashid, S. M., Rula, A., Schmelzeisen, L.,

Sequeda, J., Staab, S., and Zimmermann, A. (2022). Knowledge graphs. *ACM Computing Surveys*, 54(4):1–37.

Noy, N. F. and McGuinness, D. L. (2001). Ontology development 101: A guide to creating your first ontology. Technical Report KSL-01-05, Stanford Knowledge Systems Laboratory.

Sequeda, J. and Lassila, O. (2021). *Designing and Building Enterprise Knowledge Graphs*. Morgan & Claypool.

6

การออกแบบ Ontology จากกรณีศึกษา MSSN-Onto และ Dx-Onto

บทก่อนหน้าอธิบายว่าการวิศวกรรม Ontology ต้องเริ่มจากคำถาม ขอบเขต และความหมายที่ผู้ใช้ในโดเมนยอมรับร่วมกัน ไม่ใช่เริ่มจากการสร้าง Class จำนวนมาก บทนี้นำหลักดังกล่าวมาวิเคราะห์ Ontology สองชุดที่เกิดจากปัญหาต่างกันอย่างชัดเจน ชุดแรกคือ MSSN-Onto ซึ่งออกแบบเพื่อแทน Multimedia Sensor Network ให้เชื่อม Sensor, Observation, ข้อมูลสื่อ และเหตุการณ์ที่ระบบตรวจพบได้ ส่วนชุดที่สองคือ Dx-Onto ซึ่งออกแบบเพื่อเชื่อมความรู้ของโครงการเปลี่ยนผ่านสู่ดิจิทัลที่กระจายอยู่ในเอกสารและระบบหลายชนิด การวางสองกรณีไว้ด้วยกันไม่ได้หมายความว่า Ontology ทั้งสองมีโครงสร้างเหมือนกัน แต่ช่วยให้เห็นว่าหลักวิศวกรรมเดียวกันนำไปสู่การตัดสินใจคนละแบบเมื่อคำถามและข้อมูลของโดเมนต่างกัน (Angsuchotmetee et al., 2020; Ben-arlee and Angsuchotmetee, 2026)

เนื้อหาในบทนี้มุ่งอธิบายเหตุผลของการออกแบบตามที่ปรากฏในผลงานต้นทาง ไม่ได้เสนอ Class, Property หรือผลการทดลองชุดใหม่ ผู้อ่านจึงควรอ่านภาพและตัวอย่างในฐานะกรณีศึกษาว่า Ontology engineer แยกสิ่งใดออกจากกัน เลือก reuse มาตรฐานตรงไหน และเพิ่มคำศัพท์เฉพาะโดเมนเมื่อใด หลังจากเข้าใจโครงสร้างภายในแล้ว บทถัดไปจึงค่อยพิจารณาว่า Ontology เฉพาะโดเมนเช่น MSSN-Onto จะเชื่อมกับ SSN/SOSA, DUL และ Ontology ภายนอกอย่างระมัดระวังได้อย่างไร

เพื่อให้ IRI ในภาพและตัวอย่างอ่านได้กระชับ บทนี้ใช้ Prefix **mssn:** แทน Namespace **https://mssn.sigappfr.org/mssn-onto#** ตาม MSSN-Onto และใช้ Prefix **dx:** แทน Namespace **https://dxonto.105app.site/dx#** ตาม Dx-Onto Prefix เป็นเพียงชื่อย่อในเอกสาร ส่วน Identity ของ Class และ Property ยังคงเป็น IRI เต็มภายใต้ Namespace ดังกล่าว

6.1 จากกระบวนการวิศวกรรมสู่การตัดสินใจออกแบบ

บทก่อนอธิบาย Ontology Engineering ในฐานะกระบวนการทั่วไป ส่วนกรณีศึกษาจะมีคุณค่าเมื่อสามารถชี้ได้ว่ากระบวนการนั้นนำไปสู่การตัดสินใจใด หัวข้อนี้จึงกำหนดกรอบสำหรับอ่าน Class, Property, Axiom และส่วนที่ Reuse เพื่อใช้เปรียบเทียบ MSSN-Onto กับ Dx-Onto บนหลักเดียวกัน

Ontology หนึ่งชุดสามารถมองอย่างย่อเป็นโครงสร้าง

$$O_D = \langle C_D, P_D, A_D \rangle$$

โดย O_D แทน Ontology ของโดเมนหนึ่ง C_D คือเซตของ Class ที่ใช้จำแนกสิ่งในโดเมน P_D คือเซตของ Property ที่ใช้แทนคุณลักษณะและความสัมพันธ์ และ A_D คือเซตของ Axiom ที่กำหนดความหมายเชิงตรรกะระหว่าง Class กับ Property ตัวแปรทั้งสามส่วนไม่ได้บอกว่า Ontology ที่ดีต้องมีขนาดเท่าใด แต่ช่วยให้ตรวจการตัดสินใจได้เป็นลำดับว่า Class ใดจำเป็นต่อคำถาม Property ใดสร้างเส้นทางไปยังคำตอบ และ Axiom ใดมีเหตุผลเพียงพอที่จะให้ Reasoner ใช้อนุมาน

ในงานที่มีมาตรฐานหรือ Ontology เดิมอยู่แล้ว ส่วนประกอบดังกล่าวอาจแยกต่อได้เป็น

$$C_D = C_R \cup C_L, \qquad P_D = P_R \cup P_L, \qquad A_D = A_R \cup A_L \cup A_B$$

สัญลักษณ์กำกับ R หมายถึงส่วนที่ reuse จาก Ontology ภายนอก ส่วน L หมายถึงคำศัพท์หรือ Axiom ที่เพิ่มขึ้นเพื่อรองรับความต้องการเฉพาะของโดเมน และ A_B หมายถึง Axiom เชื่อมระหว่างส่วนที่ reuse กับส่วนที่เพิ่มใหม่ สมการนี้เป็นเพียงกรอบอ่านการออกแบบ ไม่ได้หมายความว่าไฟล์ Ontology ต้องแบ่งเป็นสามไฟล์เสมอไป ประโยชน์ของกรอบนี้คือทำให้ถามได้ว่าแนวคิดหนึ่งถูกสร้างใหม่เพราะมาตรฐานเดิมไม่ครอบคลุมจริง หรือเพียงเพราะทีมยังไม่ได้สำรวจคำศัพท์ที่มีอยู่ การนำคำศัพท์เดิมกลับมาใช้มากเกินไปอาจทำให้แบบจำลองรับความหมายจาก Ontology ภายนอกที่ไม่ตรงกับงาน ขณะที่การสร้างใหม่ทั้งหมดทำให้ข้อมูลเชื่อมกับชุมชนภายนอกได้ยาก การออกแบบจึงต้องพิจารณาทั้งความสามารถในการทำงานร่วมกันและรายละเอียดที่โดเมนต้องใช้ (Noy and McGuinness, 2001; Corcho et al., 2003)

กรอบเดียวกันยังช่วยแยก Ontology ออกจากข้อมูล Instance ได้ชัดขึ้น ในกรณี MSSN-Onto คำว่า VideoSensor หรือ MediaSegment เป็นแนวคิดของแบบจำลอง ส่วนกล้องตัวหนึ่งและช่วงวิดีโอที่บันทึกจริงเป็น Instance ในกรณี Dx-Onto คำว่า DxProject, DxPhase และ DxDimension เป็นแนวคิด ส่วนโครงการหนึ่ง รายงานหนึ่ง หรือหน่วยงานหนึ่งเป็น Resource ที่ใช้แนวคิดเหล่านั้นอธิบาย หากไม่แยกสองระดับนี้ ผู้ออกแบบอาจนำชื่อข้อมูลเฉพาะรายการไปสร้างเป็น Class หรือกลับกันอาจเก็บแนวคิดที่ต้องใช้ร่วมกันเป็นข้อความ Literal จน Query และ Alignment ทำได้ยาก

6.2 MSSN-Onto: จาก Sensor สู่ข้อมูลสื่อและเหตุการณ์

กรณีแรกมาจากประสบการณ์วิจัยด้าน Multimedia Semantic Sensor Networks ซึ่งพบว่าคำศัพท์ Sensor และ Observation ที่มีอยู่ยังไม่เพียงพอสำหรับอธิบายข้อมูลหลายสื่อ Segment, Descriptor และ Event ที่ตรวจพบ การตัดสินใจของ MSSN-Onto จึงเริ่มจากการรักษาฐาน SSN/SOSA แล้วเพิ่มเฉพาะส่วนที่โจทย์ Multimedia ต้องใช้

จุดเริ่มต้นของ MSSN-Onto คือข้อสังเกตว่า Multimedia Sensor Network มีรายละเอียดมากกว่าการบอกว่า Sensor หนึ่งสร้างค่าหนึ่งออกมา กล้องสร้างภาพหรือวิดีโอ microphone สร้างเสียง และอุปกรณ์บางชนิดสร้างข้อมูลหลายสื่อร่วมกัน ข้อมูลเหล่านี้ยังมีช่วงเวลา ตำแหน่งภายในสื่อ และ descriptor ที่ได้จากการประมวลผล ก่อนที่ระบบจะใช้หลักฐานหนึ่งรายการหรือหลายรายการร่วมกันเพื่อตรวจเหตุการณ์ หากแบบจำลองปน Sensor, ข้อมูลที่บันทึก, ผลการสังเกต และเหตุการณ์ที่ตีความไว้เป็นสิ่งเดียวกัน ระบบจะอธิบายได้ยากว่าเหตุการณ์หนึ่งมาจากข้อมูลใดและผ่านกระบวนการใด

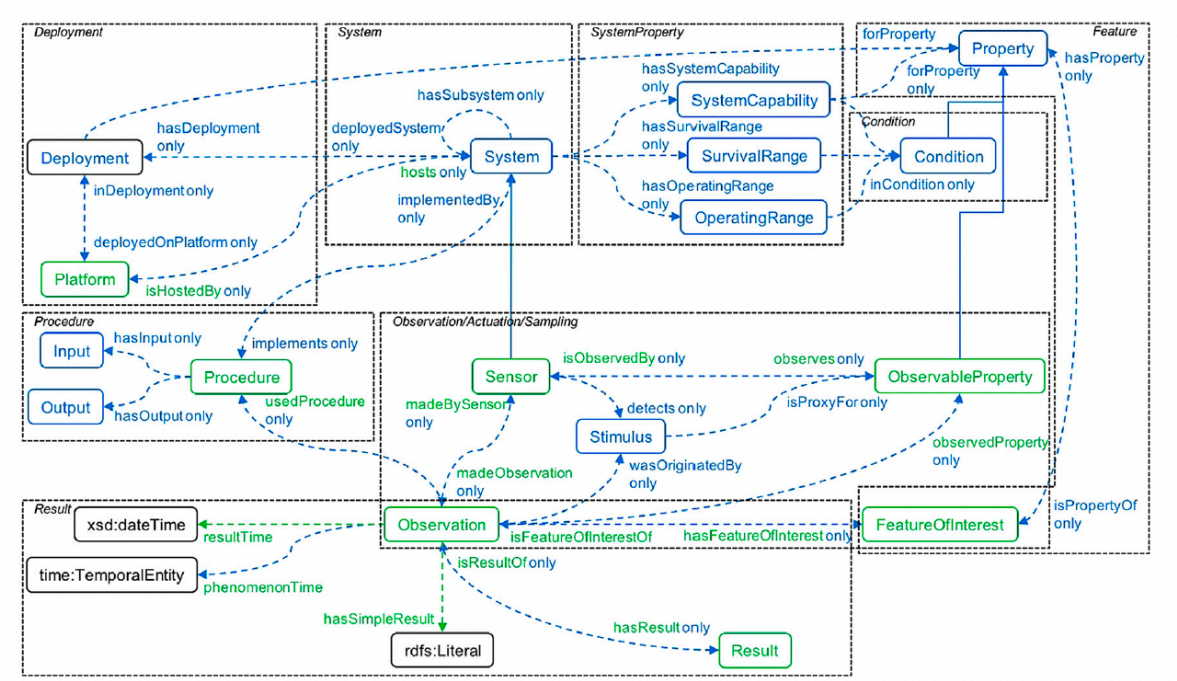
งาน MSSN-Onto กำหนดความต้องการสำคัญไว้สามด้าน

- แทน Sensor Network และข้อมูลสื่อในระดับที่รองรับการอธิบาย Event

- ทำให้ข้อมูลจากระบบต่างกันใช้คำศัพท์ร่วมกันได้
- รองรับนิยามและการตรวจจับ Event โดยไม่ผูก Ontology ไว้กับห้องประชุมหรืออาคารชนิดใดชนิดหนึ่ง

จากความต้องการเหล่านี้ การตัดสินใจแรกจึงไม่ใช่การสร้างแนวคิด Sensor ใหม่ทั้งหมด แต่เริ่มจากฐานของ SSN/SOSA ซึ่งมีคำศัพท์สำหรับ System, Sensor, Observation, Result, Feature of Interest, Observable Property และบริบทของการติดตั้งหรือขั้นตอนการสังเกตอยู่แล้ว (World Wide Web Consortium, 2017; Haller et al., 2019; Angsuchotmetee et al., 2020)

องค์ประกอบหลักของฐาน SSN/SOSA ที่ MSSN-Onto ใช้เป็นจุดตั้งต้นแสดงในภาพที่ 6.1 ภาพนี้ควรอ่านจากบริเวณ Observation ซึ่งเชื่อม Sensor, Result, Feature of Interest และ Observable Property ก่อน แล้วจึงมองออกไปยัง System, Deployment, Procedure และ Condition การวางองค์ประกอบเช่นนี้ทำให้การสังเกตหนึ่งรายการไม่ถูกลดเหลือเพียงค่าตัวเลข แต่มีทั้งผู้สร้างผล สิ่งที่ถูกสังเกต คุณสมบัติที่สนใจ และบริบทที่เกี่ยวข้อง



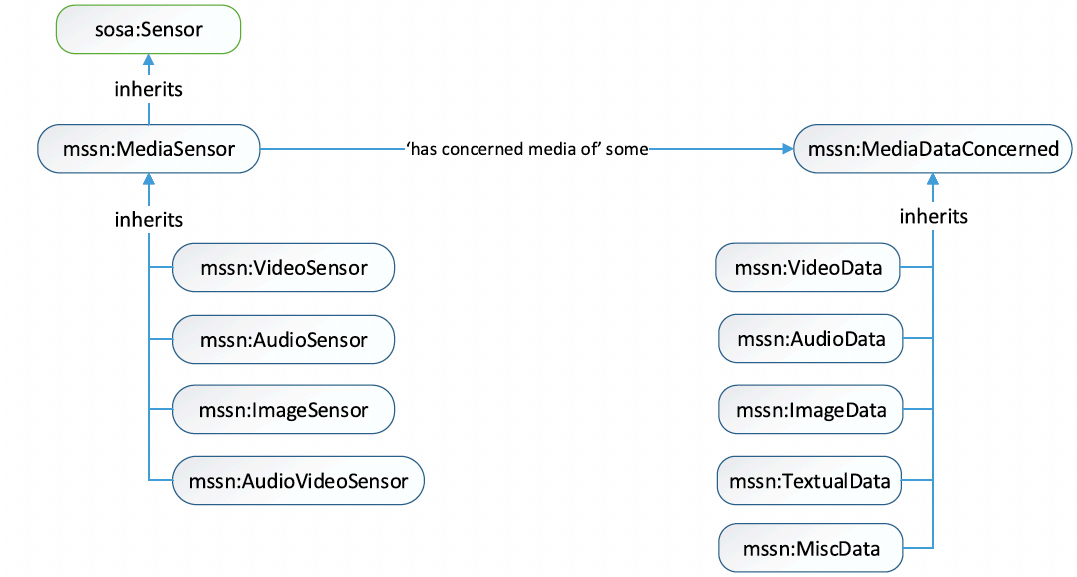
ภาพที่ 6.1: องค์ประกอบของ SSN/SOSA ที่ใช้เป็นฐานสำหรับอธิบาย Sensor, Observation และ Result (ที่มา: Angsuchotmetee et al., 2020, ภาพที่ 3)

ภาพที่ 6.1 แสดงให้เห็นว่างาน Sensor มี Vocabulary มาตรฐานรองรับอยู่มากแล้ว MSSN-Onto จึงไม่จำเป็นต้องนิยามความหมายพื้นฐานของ Sensor หรือ Observation ซ้ำทั้งหมด ช่องว่างที่ต้องเติมอยู่ที่ชนิดของ multimedia Sensor รูปแบบข้อมูลสื่อ การแบ่งข้อมูลเป็น segment และ descriptor ที่ Event processing ต้องใช้ การเลือก extend เฉพาะช่องว่างนี้ช่วยให้ข้อมูล MSSN ยังเชื่อมกับระบบที่เข้าใจ SOSA/SSN ได้ ขณะเดียวกันก็มีรายละเอียดพอสำหรับงาน multimedia (Angsuchotmetee et al., 2020)

การตัดสินใจถัดมาคือไม่ใช้ Class Sensor เพียง Class เดียวครอบอุปกรณ์ทุกชนิด MSSN-Onto เพิ่มแนวคิด **mssn:MediaSensor** ใต้ **sosa:Sensor** แล้วแยกชนิดสำคัญ เช่น VideoSensor, AudioSensor, ImageSensor และ AudioVideoSensor ความสัมพันธ์เชิงลำดับชั้นนี้อ่านแบบ Description Logic อย่างย่อได้ว่า

$$MediaSensor \sqsubseteq Sensor$$

สัญลักษณ์ $\sqsubseteq$ หมายความว่า Instance ทุกตัวของ MediaSensor เป็น Sensor ด้วย แต่ Sensor ใน SOSA ไม่จำเป็นต้องเป็น MediaSensor ทุกตัว การเลือก Subclass แทน Equivalence จึงรักษาความหมายของมาตรฐานเดิมและเพิ่มความเฉพาะเพียงทิศทางเดียว นอกจากนี้ MediaSensor ยังเชื่อมไปยัง **mssn:MediaDataConcerned** เพื่อบอกชนิดสื่อที่ Sensor นั้นเกี่ยวข้อง การออกแบบส่วนนี้แสดงในภาพที่ 6.2



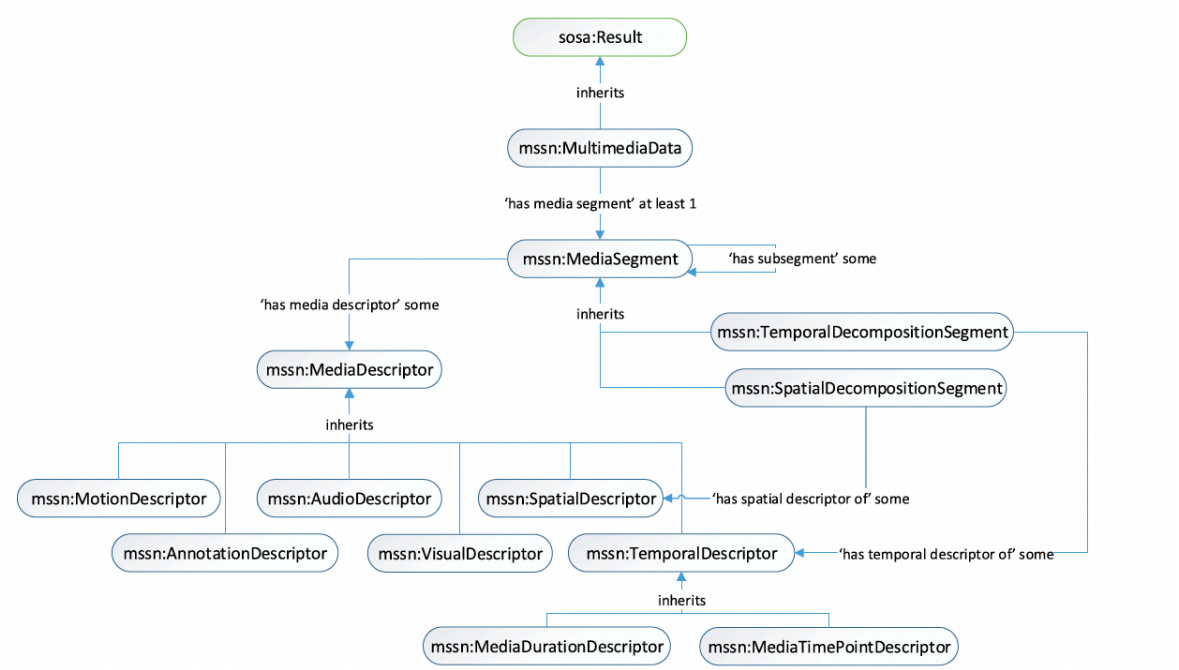
ภาพที่ 6.2: การขยายชนิด Media Sensor และชนิดข้อมูลสื่อใน MSSN-Onto (ที่มา: Angsuchotmetee et al., 2020, ภาพที่ 4)

ภาพที่ 6.2 แสดงการตัดสินใจสองด้านพร้อมกัน ด้านซ้ายเป็น Taxonomy ของ Sensor ส่วนด้านขวาเป็น Taxonomy ของข้อมูลที่เกี่ยวข้อง ได้แก่ VideoData, AudioData, ImageData, TextualData และ MiscData การแยก Taxonomy สองฝั่งสำคัญเพราะชนิดอุปกรณ์กับชนิดข้อมูลไม่ใช่เรื่องเดียวกัน กล้องหนึ่งอาจสร้างทั้งวิดีโอ ภาพนิ่ง และ Metadata ขณะที่ข้อมูล TextualData อาจมาจากกระบวนการถอดเสียง ไม่ได้หมายความว่ามีอุปกรณ์ชนิด TextSensor โดยอัตโนมัติ ความสัมพันธ์ระหว่างสองฝั่งจึงควรถูกเขียนเป็น Property แทนการบังคับให้ลำดับชั้นของ Sensor และ data มีรูปเหมือนกัน

เมื่อพิจารณาข้อมูลที่ Sensor สร้าง MSSN-Onto วาง **mssn:MultimediaData** ไว้ใต้แนวคิด Result ของ SOSA การตัดสินใจนี้ทำให้ข้อมูลสื่อถูกอ่านในฐานะผลของ Observation ได้ แต่ยังต้องเพิ่มโครงสร้างภายในที่ Result ทั่วไปไม่ได้กำหนด ข้อมูลวิดีโอหรือเสียงหนึ่งรายการอาจยาวเกินกว่าจะใช้ตรวจเหตุการณ์โดยตรง จึงต้องแบ่งเป็น MediaSegment และบางกรณีแบ่งต่อเป็น subsegment การแบ่งอาจอาศัยเวลา เช่นช่วง

วินาทีที่ 20 ถึง 35 หรืออาศัยพื้นที่ เช่นบริเวณหนึ่งภายในภาพ การแยก segment เป็น Resource ทำให้แต่ละช่วงมี IRI มี descriptor และเชื่อมกลับไปยังข้อมูลต้นฉบับได้

โครงสร้างของข้อมูลสื่อ segment และ descriptor แสดงในภาพที่ 6.3 โดยส่วนบนเริ่มจาก **sosa:Result** ลงมายัง **mssn:MultimediaData** และ **mssn:MediaSegment** ส่วนล่างแยก descriptor ตามหน้าที่ เช่น motion, audio, visual, spatial และ temporal



ภาพที่ 6.3: การออกแบบ Multimedia Data, Media Segment และ Media Descriptor ใน MSSN-Onto (ที่มา: Angsuchotmetee et al., 2020, ภาพที่ 6)

ภาพที่ 6.3 แสดงว่า MSSN-Onto ไม่เก็บ descriptor ทุกชนิดเป็น Literal ที่ติดอยู่กับไฟล์สื่อโดยตรง แต่แยกแนวคิด MediaDescriptor เพื่อให้ descriptor แต่ละกลุ่มมีความหมายและขยายต่อได้ TemporalDescriptor ยังแยกเป็น MediaDurationDescriptor กับ MediaTimePointDescriptor เพราะคำว่า “เกิดเมื่อใด” และ “กินเวลานานเท่าใด” เป็นคนละคุณสมบัติ สำหรับการตรวจเหตุการณ์ ความต่างนี้มีผลโดยตรง เช่นเหตุการณ์เปลี่ยนสไลด์อาจอาศัย time point หลายจุด ส่วนกิจกรรมการอภิปรายอาจอาศัย duration และข้อมูลเสียงหลาย segment ร่วมกัน

การแยกชั้นดังกล่าวนำไปสู่เส้นทางความรู้ที่ตรวจได้จาก Sensor ไปยัง Observation จาก Observation ไปยัง multimedia result จาก result ไปยัง segment และ descriptor แล้วจึงไปยัง Event definition หรือเหตุการณ์ที่ตรวจพบ ประโยชน์ไม่ได้อยู่ที่การทำกราฟให้มี Node มากขึ้น แต่อยู่ที่การตอบคำถามได้ว่าเหตุการณ์หนึ่งอาศัยข้อมูลช่วงใด ข้อมูลนั้นสร้างโดย Sensor ใด และ feature ใดถูกใช้ประกอบการตัดสิน ถ้าเก็บเพียงว่า “กล้องตรวจพบเหตุการณ์” ระบบจะไม่มีโครงสร้างเพียงพอสำหรับเปลี่ยนวิธีประมวลผล ตรวจข้อผิดพลาด หรือใช้ข้อมูลเดิมกับ Event definition รุ่นใหม่

จุดที่ต้องระวังคือ Ontology ไม่ได้ทำให้เครื่องมือรู้จำภาพหรือเสียงแม่นยำขึ้นเอง MSSN-Onto กำหนดความหมายและเส้นทางเชื่อมหลักฐาน แต่ค่าที่สกัดจากสื่อยังขึ้นกับ algorithm และคุณภาพข้อมูลต้นทาง หากระบบระบุ Event ผิด อาจผิดที่ descriptor ผิดที่ Mapping ผิดที่ time window หรือผิดที่ Event

definition การออกแบบ Ontology ที่แยกขั้นเหล่านี้ช่วยให้หาต้นเหตุได้ แต่ไม่ควรนำผลของ classifier มาใช้เป็นหลักฐานว่า Class Hierarchy ถูกต้องโดยอัตโนมัติ

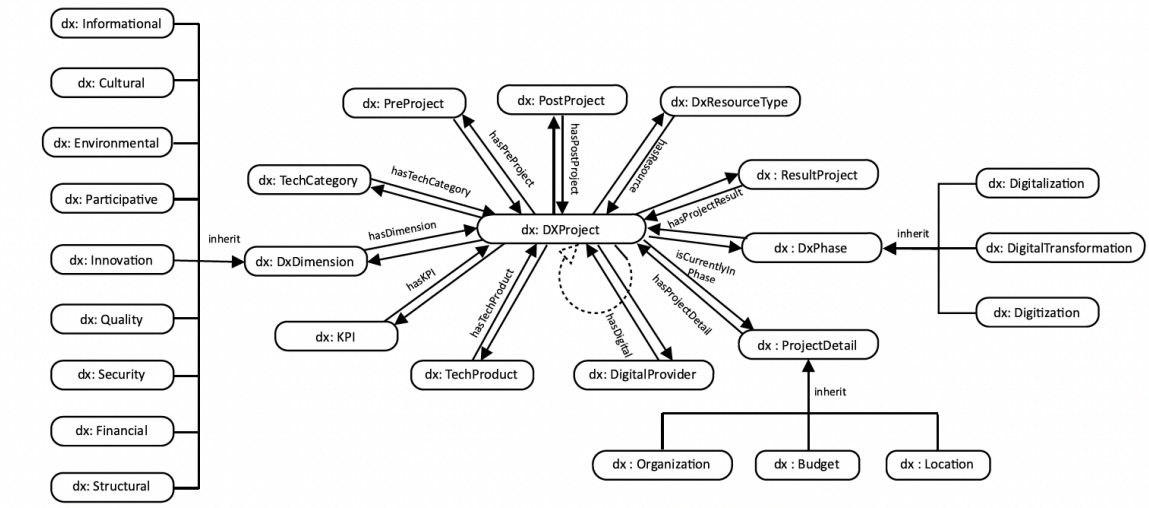
6.3 Dx-Onto: การวางโครงการเป็นแกนของความรู้

ประสบการณ์จาก MSSN-Onto เริ่มด้วยมาตรฐาน Sensor ที่มีฐานค่อนข้างชัด ส่วน Dx-Onto เผชิญโจทย์อีกลักษณะหนึ่ง คือความรู้ของโครงการกระจายอยู่ในเอกสารและระบบหลายชนิด การออกแบบจึงต้องหา Resource แกนที่ทำให้ข้อมูลเหล่านั้นกล่าวถึงโครงการเดียวกันได้ก่อนพิจารณารายละเอียดส่วนอื่น

Dx-Onto เริ่มจากปัญหาคนละแบบกับ MSSN-Onto ข้อมูลโครงการเปลี่ยนผ่านสู่ดิจิทัลไม่ได้ไหลเข้ามาเป็น stream ต่อเนื่อง แต่กระจายอยู่ในรายงาน PDF, Spreadsheet, รูปภาพ, วิดีโอ และฐานข้อมูลของหลายหน่วยงาน เอกสารหนึ่งบอกชื่อกับรายละเอียดโครงการ อีกเอกสารบอกเทคโนโลยี งบประมาณ หรือผู้ให้บริการ ขณะที่ผลลัพธ์และตัวชี้วัดอาจอยู่ในรายงานคนละฉบับ การค้นด้วยชื่อไฟล์หรือคำสำคัญจึงไม่เพียงพอสำหรับคำถามหลายเงื่อนไข เช่นโครงการใดอยู่ในระยะหนึ่ง ใช้เทคโนโลยีประเภทหนึ่ง และมีผลลัพธ์ด้านหนึ่งพร้อมหลักฐานรองรับ

กระบวนการพัฒนา Dx-Onto เริ่มจาก requirement และ Competency Questions แล้วจึงออกแบบ Vocabulary, นำข้อมูลตัวอย่างเข้ากราฟ และประเมินด้วย Query ตามคำถามที่กำหนดไว้ วิธีคิดนี้ทำให้ Class ไม่ได้เกิดจากการรวบรวมคำนามทุกคำในเอกสาร แต่เกิดจากสิ่งที่ต้องใช้เชื่อมข้อมูลและตอบคำถาม โครงสร้างถูกออกแบบให้เป็น modular เพื่อให้ส่วนแกนของโครงการแยกจาก phase, dimension, Technology, provider, Resource และรายละเอียดประกอบ การแบ่งเช่นนี้ช่วยให้เพิ่มคำศัพท์ใน module หนึ่งโดยไม่ต้องเปลี่ยนความหมายของทั้ง Ontology (Ben-arlee and Angsuchotmetee, 2026)

โครงสร้างหลักของ Dx-Onto แสดงในภาพที่ 6.4 โดยมี **dx:DxProject** อยู่กลางภาพ การวาง Project เป็นศูนย์กลางไม่ได้หมายความว่าทุกข้อมูลเป็นคุณลักษณะภายใน Project แต่หมายถึง Project เป็น Resource ที่เชื่อมไปยัง Resource อื่นผ่าน Property ที่บอกความหมายชัดเจน



ภาพที่ 6.4: โครงสร้างแกนของ Dx-Onto ที่เชื่อมโครงการกับ Phase, Dimension, Resource และรายละเอียดที่เกี่ยวข้อง (ที่มา: ภาพที่ 3 ในต้นฉบับผู้เขียนของงาน Ben-arlee and Angsuchotmetee, 2026)

ภาพที่ 6.4 แสดงกลุ่มการตัดสินใจสำคัญของ Dx-Onto กลุ่มแรกคือวงจรชีวิตของโครงการ ซึ่งแยก PreProject, PostProject และ DxPhase ออกจากกัน คำว่า phase อธิบายสถานะหรือช่วงของการเปลี่ยนผ่าน เช่น Digitization, Digitalization และ Digital Transformation ส่วน PreProject กับ PostProject ใช้เชื่อมรายละเอียดก่อนและหลังช่วงดำเนินงาน การแยกกลุ่มนี้ช่วยไม่ให้ข้อมูลทุกช่วงถูกตีความว่าเป็นสถานะชนิดเดียวกัน

กลุ่มที่สองคือ DxDimension ซึ่งจัดแนวคิดเชิงกลยุทธ์เก้าด้าน เช่น Informational, Cultural, Environmental, Participative, Innovation, Quality, Security, Financial และ Structural ไว้ในลำดับชั้นเดียวกัน Dimension ไม่ใช่ KPI และไม่ใช่ผลลัพธ์ของโครงการ แต่เป็นมุมที่ใช้จัดหมวดหรือวิเคราะห์ลักษณะของการเปลี่ยนผ่าน หากเก็บชื่อ dimension เป็นข้อความอิสระ แต่ละเอกสารอาจสะกดหรือแบ่งหมวดต่างกันจน Query เปรียบเทียบข้ามโครงการไม่ได้ การทำให้ dimension มี IRI และตำแหน่งใน Vocabulary จึงเป็นการสร้างจุดอ้างอิงร่วม ไม่ใช่เพียงการตกแต่งข้อมูลด้วยคำศัพท์

กลุ่มที่สามคือ Technology และทรัพยากร ภาพแยก TechCategory, TechProduct, DigitalProvider และ DxResourceType ออกจากกัน เพราะประเภทเทคโนโลยี ผลิตภัณฑ์ ผู้ให้บริการ และเอกสารหรือทรัพยากรที่เกี่ยวข้องมีบทบาทคนละแบบ ตัวอย่างเช่น drone อาจเป็นผลิตภัณฑ์หรือเทคโนโลยีที่โครงการใช้ บริษัทหนึ่งเป็นผู้ให้บริการ และไฟล์ภาพจาก drone เป็น Resource ของโครงการ การรวมทั้งสามอย่างไว้ในช่อง “Technology” เดียวจะทำให้ Query ไม่ทราบว่าค่าหนึ่งหมายถึงชนิด ผู้ขาย หรือหลักฐาน

กลุ่มที่สี่คือ ProjectDetail, KPI และ ResultProject ซึ่งแยกบริบทการดำเนินงาน ตัวชี้วัด และผลของโครงการออกจากกัน ProjectDetail เชื่อมต่อไปยัง Organization, Budget และ Location เพื่อให้รายละเอียดที่อาจมาจากคนละเอกสารยังอ้างกลับมายังโครงการเดียวกันได้ KPI บอกสิ่งที่ใช้วัด ส่วน ResultProject บอกผลที่ถูกรายงาน ความแตกต่างนี้สำคัญต่อการตรวจหลักฐาน เพราะค่าเป้าหมายของ KPI ไม่ใช่ผลสำเร็จที่เกิดขึ้นจริง และข้อความสรุปผลไม่ควรกลายเป็นข้อเท็จจริงที่ยืนยันแล้วหากยังไม่มี Source รองรับ

หากมองภาพเป็น Graph Pattern แกนของ Dx-Onto สามารถเขียนอย่างย่อได้ว่า

$$\textit{DxProject} \xrightarrow{\textit{hasDimension}} \textit{DxDimension}, \qquad \textit{DxProject} \xrightarrow{\textit{isCurrentlyInPhase}} \textit{DxPhase}$$

และ

$$\textit{DxProject} \xrightarrow{\textit{hasResource}} \textit{DxResourceType}, \qquad \textit{DxProject} \xrightarrow{\textit{hasProjectResult}} \textit{ResultProject}$$

ลูกศรในที่นี้ใช้ช่วยอ่านเส้นทางของ Property ไม่ใช่การประกาศ cardinality ว่า Project ทุกตัวต้องมีค่าครบทุกเส้น การตัดสินว่าข้อมูลระยะ draft หรือข้อมูลที่เผยแพร่ต้องมี Organization, phase, KPI หรือ result อย่างน้อยกี่ค่าควรเป็นหน้าที่ของ data Contract และ SHACL ในบทหลัง การแยกความหมายของ Ontology ออกจากข้อกำหนดความครบถ้วนของ Workflow ทำให้ Dx-Onto ใช้ได้กับโครงการที่มีข้อมูลต่างระดับกันโดยไม่บังคับว่า Graph ที่ยังไม่ครบเป็นโลกที่ขัดแย้งทางตรรกะทันที

การวาง Project เป็นศูนย์กลางยังทำให้เห็นบทบาทของ Identity ชัดเจน เอกสารสองฉบับอาจกล่าวถึงชื่อโครงการต่างรูป องค์กรเดียวกันอาจมีชื่อย่อหลายแบบ และไฟล์หนึ่งอาจถูกย้ายหรือเปลี่ยนชื่อ หาก Mapping สร้าง IRI ใหม่ตามข้อความทุกครั้ง กราฟจะมีโครงการหรือองค์กรซ้ำ การออกแบบ Class และ Property จึงต้องทำคู่กับ IRI Policy และ Entity Resolution แม้สองเรื่องหลังไม่ปรากฏในภาพ Ontology โดยตรง แต่เป็นเงื่อนไขที่ทำให้โครงสร้างนั้นใช้กับข้อมูลจริงได้

เช่นเดียวกับ MSSN-Onto ขอบเขตของ Dx-Onto ไม่ได้ครอบคลุมระบบบริหารโครงการทุกหน้าที่ Ontology ทำหน้าที่เป็น semantic core สำหรับเชื่อมความรู้ ไม่ได้แทนระบบเก็บไฟล์ ระบบบัญชี Workflow การอนุมัติ หรือเครื่องมือวิเคราะห์ทุกชนิด การนำไปใช้จึงต้องมี Mapping จาก Schema ต้นทาง มี Provenance ของข้อมูลที่สกัด และมี Query ที่ตรวจว่าความสัมพันธ์ซึ่งออกแบบไว้ตอบ Competency Questions ได้จริง

6.4 การตัดสินใจที่เหมือนและต่างกัน

เมื่อพิจารณาเหตุผลของแต่ละกรณีแยกกันแล้ว การวางสอง Ontology ไว้ในกรอบเดียวกันช่วยแยกหลัก Ontology Engineering ที่ใช้ร่วมกันออกจากการตัดสินใจเฉพาะโดเมน ความเหมือนและความต่างจึงควรถูกอ่านจากปัญหาและคำถาม ไม่ใช่เปรียบเทียบเพียงจำนวน Class หรือรูปของแผนภาพ

เมื่ออ่านสอง Ontology ผ่านกรอบ $O_D = \langle C_D, P_D, A_D \rangle$ จะเห็นส่วนร่วมสองประการ

- ทั้งสองกรณีไม่ได้เริ่มจาก Syntax ของ OWL แต่เริ่มจากความแตกต่างที่ระบบเดิมอธิบายไม่ได้ MSSN-Onto ต้องแยก Sensor, Observation, Multimedia Segment, Descriptor และ Event ส่วน Dx-Onto ต้องแยก Project, Phase, Dimension, Technology, Provider, Resource, KPI และ Result
- ทั้งสองกรณีทำให้สิ่งที่ต้องอ้างซ้ำมี Identity ของตนเอง เพื่อให้ข้อมูลจากหลายแหล่งเชื่อมกันและ Query ใช้ Relation เหล่านั้นค้นคำตอบได้

ความต่างสำคัญคือ MSSN-Onto มีมาตรฐาน SSN/SOSA เป็นฐานที่ชัดและเพิ่มคำศัพท์ multimedia บนโครงเดิม ขณะที่ Dx-Onto วาง core Vocabulary ของโดเมนโครงการขึ้นจาก Competency Questions และจัด module รอบ DxProject MSSN-Onto จึงให้ความสำคัญมากกับเวลา segment และเส้นทางจาก Observation ไปยัง Event ส่วน Dx-Onto ให้ความสำคัญกับการรวม Artifact ต่างรูป ตัวตนของโครงการ และความสัมพันธ์หลายมิติ ความต่างเหล่านี้สรุปในตารางที่ 6.1

ตารางที่ 6.1: การเปรียบเทียบการตัดสินใจออกแบบ MSSN-Onto และ Dx-Onto

ประเด็น	MSSN-Onto	Dx-Onto
ปัญหาหลัก	เชื่อม Sensor, Observation, ข้อมูลหลายสื่อ และ Event	เชื่อมความรู้โครงการจากเอกสารและระบบหลายชนิด
ฐานที่ reuse	SSN/SOSA และคำศัพท์ด้าน Metadata ของสื่อ	มาตรฐาน RDF/OWL และ Vocabulary แกนที่ออกแบบจากโดเมนโครงการ
Resource แกน	Sensor, Observation, MultimediaData, MediaSegment และ Event	DxProject, DxPhase, DxDimension, Resource, Technology, KPI และ Result
รายละเอียดเฉพาะโดเมน	ชนิดสื่อ segment descriptor เวลา และการรวมหลักฐานของ Event	ระยะโครงการ มิติเชิงกลยุทธ์ ผู้ให้บริการ ทรัพยากร และรายละเอียดโครงการ
ความเสี่ยงในการออกแบบ	ปน Sensor, result, descriptor และ detected Event เป็นสิ่งเดียวกัน	ปนชนิดเทคโนโลยี ผลิตภัณฑ์ ผู้ให้บริการ เอกสาร และผลลัพธ์
คำถามที่แบบจำลองต้องรองรับ	ข้อมูลช่วงใดจาก Sensor ใดสนับสนุน Event ใด	โครงการใดอยู่ใน phase ใด ใช้เทคโนโลยีใด และมีผลลัพธ์พร้อมหลักฐานใด

ตารางที่ 6.1 ชี้ว่าการออกแบบ Ontology ไม่ควรถูกประเมินด้วยจำนวน Class หรือ Property เพียงอย่างเดียว Ontology ที่มี Class น้อยอาจตอบคำถามได้ครบหากเลือกขอบเขตเหมาะสม ขณะที่ Ontology ขนาดใหญ่อาจใช้งานไม่ได้หากเส้นทาง Relation ไม่สอดคล้องกับข้อมูลต้นทาง การประเมินจึงต้องย้อนกลับไปยัง Competency Questions ตัวอย่าง Instance และ Query เสมอ ใน MSSN-Onto Query ต้องแสดงว่าสามารถค้นจาก Event ย้อนกลับไปยัง Observation, segment และ Sensor ได้ ใน Dx-Onto Query ต้องแสดงว่าเงื่อนไขด้าน Project, phase, Technology, dimension และ result เชื่อมอยู่บน Resource ที่ถูกต้อง ไม่ใช่เพียงพบคำเหล่านั้นกระจายในเอกสาร

อีกบทเรียนหนึ่งคือการนำกลับมาใช้ ไม่ได้หมายถึงการประกาศ Equivalence ให้มากที่สุด MSSN-Onto เลือก Subclass เพื่อบอกว่า MediaSensor เป็น Sensor ชนิดหนึ่ง โดยไม่บอกว่า Sensor ทุกตัวเป็น MediaSensor ส่วนแนวคิดเฉพาะเช่น MediaSegment และ MediaDescriptor ถูกเพิ่มเพราะมีบทบาทที่ฐาน Sensor Ontology ไม่ได้กำหนด ใน Dx-Onto การแยก module ทำหน้าที่คล้ายกัน กล่าวคือเก็บ core ของ Project ไว้มั่นคงและเปิดให้ส่วน phase, dimension หรือ Resource ขยายตามข้อมูลกับคำถาม การตัดสินใจทั้งสองแบบใช้หลัก minimal commitment คือประกาศเท่าที่มีเหตุผลรองรับและรักษาทางสำหรับการขยายในอนาคต

6.5 ขอบเขต ข้อจำกัด และบทเรียนสำหรับการนำไปใช้

การเปรียบเทียบก่อนหน้าชี้ให้เห็นเหตุผลของโครงสร้าง แต่การนำ Ontology ไปใช้ยังต้องทราบด้วยว่าแบบจำลองไม่ได้รับผิดชอบส่วนใด หัวข้อนี้จึงระบุขอบเขตของ MSSN-Onto และ Dx-Onto พร้อม Artifact ที่ต้องมีเพิ่มเติมเมื่อเชื่อม Ontology เข้ากับข้อมูลและระบบจริง

กรณีศึกษาทั้งสองแสดงว่า Ontology ที่ดีต้องบอกสิ่งที่ตนไม่ได้ทำด้วย MSSN-Onto อธิบายโครงสร้างความหมายของข้อมูล Sensor และ multimedia แต่ไม่ได้แทน algorithm สำหรับรู้จำภาพ เสียง หรือวิดีโอ และไม่ได้รับรองว่า Event ที่ตรวจพบถูกต้องทุกครั้ง Dx-Onto อธิบายความสัมพันธ์ของความรู้โครงการ แต่ไม่ได้แทนระบบบริหารเอกสาร การอนุมัติงบประมาณ หรือกระบวนการจัดการโครงการทั้งหมด การเขียนข้อจำกัดเช่นนี้ช่วยให้ผู้ใช้ไม่คาดหวังว่า Ontology จะชดเชยข้อมูลต้นทางที่ผิดหรือทำหน้าที่ของ Application Layer ได้ทุกอย่าง

ในทางปฏิบัติ การนำ Ontology ไปใช้ต้องมี Artifact มากกว่าไฟล์ OWL อย่างน้อยต้องมีตัวอย่างข้อมูลที่แสดงการใช้ Class กับ Property มี Mapping Specification จากข้อมูลต้นทาง มี Query ที่ผูกกับ Competency Questions และมีชุดทดสอบสำหรับผลที่คาดหวัง MSSN-Onto ต้องมีตัวอย่างเส้นทางจาก Sensor ไปยัง Observation, multimedia segment และ Event ส่วน Dx-Onto ต้องมีตัวอย่าง Project ที่เชื่อม phase, dimension, Technology, Resource และ result โดยไม่สร้าง IRI ซ้ำ เมื่อ Ontology เปลี่ยนตัวอย่างและ Query เหล่านี้ต้องเปลี่ยนตาม มิฉะนั้นทีมจะทราบเพียงว่าไฟล์ยัง parse ได้ แต่ไม่ทราบว่าความหมายที่ระบบใช้ยังทำงานเหมือนเดิมหรือไม่

ภาพทั้งสี่ในบทนี้จึงไม่ควรถูกอ่านเป็น Class diagram ที่ต้องลอกไปใช้ทั้งหมด แต่เป็นบันทึกการตัดสินใจ MSSN-Onto แสดงการ extend มาตรฐานเมื่อโดเมนต้องการรายละเอียดใหม่ ส่วน Dx-Onto แสดงการสร้าง core Vocabulary รอบ Resource ที่เป็นศูนย์กลางของคำถาม หากผู้อ่านออกแบบ Ontology ในโดเมนอื่น คำถามสำคัญไม่ใช่ว่าควรมี Class เหมือนสองกรณีนี้หรือไม่ แต่ควรถามว่ามาตรฐานใด reuse ได้ แนวคิดใดเป็นช่องว่างของโดเมน Relation ใดจำเป็นต่อคำตอบ และขอบเขตใดควรถูกปล่อยไว้นอก Ontology

การออกแบบภายในเป็นเพียงขั้นหนึ่ง เมื่อ Ontology ต้องแลกเปลี่ยนข้อมูลกับผู้อื่น ความหมายของ Class และ Property ต้องถูกเปรียบเทียบกับมาตรฐานหรือ Ontology ภายนอก การประกาศ Axiom ที่ก่อผลอนุมานเกินหลักฐานอาจทำให้ได้ข้อสรุปผิด ขณะที่การไม่มี Mapping ทำให้ข้อมูลยังไม่เชื่อมกันในระดับความหมาย บทถัดไปจึงใช้ MSSN-Onto ที่อธิบายแล้วเป็นกรณีศึกษาสำหรับพิจารณา Upper Domain Ontology และ Ontology Alignment โดยเน้นการเชื่อมกับ SSN/SOSA และ DUL พร้อม Rationale ที่ตรวจสอบได้

6.6 สรุป

บทนี้ใช้ผลงานวิจัยสองชุดเพื่อแสดงว่าตัวตนของผู้เขียนในงาน Ontology ไม่ได้ปรากฏจากการประกาศว่าแนวทางหนึ่งดีที่สุด แต่ปรากฏจากเหตุผลของการเลือก Reuse, การแยกแนวคิด และการกำหนดขอบเขตของแบบจำลอง ข้อสรุปจึงย้อนกลับไปยังคำถามว่าแต่ละการตัดสินใจช่วยตอบโจทย์ใด

MSSN-Onto และ Dx-Onto แสดงเส้นทางจากปัญหาโดเมนไปสู่โครงสร้างความหมายสองแบบ MSSN-Onto นำฐาน SSN/SOSA กลับมาใช้แล้วเพิ่มชนิด multimedia Sensor, ข้อมูลสื่อ, segment และ descriptor เพื่อรักษาเส้นทางจาก Observation ไปยัง Event ส่วน Dx-Onto วาง DxProject เป็นจุดเชื่อมของ phase, dimension, Technology, provider, Resource, KPI, detail และ result เพื่อรวมความรู้โครงการจากหลายแหล่ง ทั้งสองกรณียืนยันหลักเดียวกันว่าการออกแบบ Ontology ต้องอธิบายได้ว่าแต่ละแนวคิดมีไว้ตอบคำถามใด เชื่อมกับข้อมูลจริงอย่างไร และมีขอบเขตตรงไหน Class, Property และเหตุผล

การออกแบบที่อธิบายในบทนี้จึงเป็นพื้นฐานให้ผู้อ่านย้อนมาตรวจสอบได้ เมื่อพิจารณา Ontology Alignment การตรวจสอบข้อมูล การกำกับดูแล และการใช้ Ontology ร่วมกับ LLM ในบทถัดไป

เอกสารอ้างอิง

Angsuchotmetee, C., Chbeir, R., and Cardinale, Y. (2020). MSSN-Onto: An ontology-based approach for flexible event processing in multimedia sensor networks. *Future Generation Computer Systems*, 108:1140–1158.

Ben-arlee, S. and Angsuchotmetee, C. (2026). Dx-Onto: A core ontology for a semantic-based framework for managing digital transformation projects. *Applied System Innovation*, 9(7):146.

Compton, M., Barnaghi, P., Bermudez, L., García-Castro, R., Corcho, O., Cox, S., Graybeal, J., Hauswirth, M., Henson, C., Herzog, A., Huang, V., Janowicz, K., Kelsey, W. D., Le Phuoc, D., Lefort, L., Leggieri, M., Neuhaus, H., Nikolov, A., Page, K., Passant, A., Sheth, A., and Taylor, K. (2012). The SSN ontology of the W3C semantic sensor network incubator group. *Journal of Web Semantics*, 17:25–32.

Corcho, O., Fernández-López, M., and Gómez-Pérez, A. (2003). Methodologies, tools and languages for building ontologies. where is their meeting point? *Data & Knowledge Engineering*, 46(1):41–64.

Gruber, T. R. (1993). A translation approach to portable ontology specifications. *Knowledge Acquisition*, 5(2):199–220.

Haller, A., Janowicz, K., Cox, S. J. D., Lefrançois, M., Taylor, K., Le Phuoc, D., Lieberman, J., García-Castro, R., Atkinson, R., and Stadler, C. (2019). The modular SSN ontology: A joint W3C and OGC standard specifying the semantics of sensors, observations, sampling, and actuation. *Semantic Web*, 10(1):9–32.

Noy, N. F. and McGuinness, D. L. (2001). Ontology development 101: A guide to creating your first ontology. Technical Report KSL-01-05, Stanford Knowledge Systems Laboratory.

World Wide Web Consortium (2017). Semantic sensor network ontology. W3C Recommendation. Accessed June 18, 2026, from **https://www.w3.org/TR/vocab-ssn/**.

7

Upper Domain Ontology และ Ontology Alignment

บทก่อนหน้าแสดงให้เห็นว่าการออกแบบ Ontology เฉพาะโดเมนต้องเริ่มจากปัญหาและคำถามที่ต้องการตอบ แล้วจึงเลือกว่าจะนำคำศัพท์เดิมมาใช้หรือสร้างส่วนขยายใหม่ เมื่อ Ontology ถูกนำไปใช้ร่วมกับแหล่งข้อมูลอื่น คำถามสำคัญจึงเปลี่ยนจาก “แบบจำลองนี้ครอบคลุมโดเมนหรือไม่” ไปเป็น “ความหมายในแบบจำลองนี้สอดคล้องกับความหมายที่ระบบอื่นใช้อย่างไร” บทนี้อธิบาย Upper Domain Ontology และ Ontology Alignment ในฐานะเครื่องมือสำหรับตอบคำถามดังกล่าว โดยใช้ DOLCE, DUL และ SSN/SOSA เป็นฐานทางวิชาการ และนำ MSSN-Onto จากบทที่ 6 กลับมาพิจารณาเฉพาะในมิติของการเชื่อมความหมายกับ Ontology ภายนอก

7.1 เหตุใด Ontology จึงต้องเชื่อมความหมายกับผู้อื่น

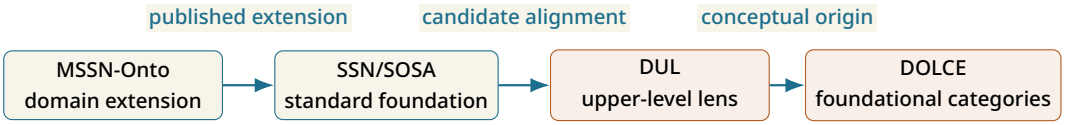
บทก่อนแสดงการออกแบบ Ontology ภายในขอบเขตของ MSSN-Onto และ Dx-Onto แต่ Ontology เฉพาะโดเมนไม่ได้ทำงานอยู่ลำพัง เมื่อข้อมูลต้องแลกเปลี่ยนกับระบบอื่น ความหมายที่ชัดภายในยังต้องมีจุดเชื่อมกับ Vocabulary ภายนอก หัวข้อนี้จึงเริ่มจากเหตุผลของการทำ Alignment ก่อนพิจารณาวิธีประกาศ Mapping

Ontology ที่สร้างขึ้นสำหรับงานหนึ่งอาจมีนิยามภายในที่ชัดเจนและตอบคำถามของระบบนั้นได้ครบ แต่ยังไม่รับประกันว่าจะทำงานร่วมกับข้อมูลจากระบบอื่นได้ ตัวอย่างเช่น ระบบสองแห่งอาจใช้คำว่า **Sensor** เหมือนกัน แห่งแรกหมายถึงอุปกรณ์ทางกายภาพ ส่วนอีกแห่งรวมซอฟต์แวร์ที่ประมวลผลสัญญาณไว้ด้วย หากรวมข้อมูลโดยดูจากชื่อเพียงอย่างเดียว ระบบอาจสรุปว่าสองคำนี้มีความหมายเท่ากันทั้งที่ขอบเขตต่างกัน ในทางกลับกัน คำที่ใช้ชื่อต่างกันอาจมีนิยามตรงกันและควรเชื่อมถึงกัน ปัญหานี้ทำให้การทำงานร่วมกันเชิงความหมายต้องอาศัยทั้งนิยาม ตัวอย่างข้อมูล และคำถามที่ต้องการใช้ข้ามระบบ ไม่ใช่อาศัยความคล้ายของคำเท่านั้น (Kalfoglou and Schorlemmer, 2003; Euzenat and Shvaiko, 2013)

การเชื่อมดังกล่าวเรียกว่า Ontology Alignment ซึ่งเป็นกระบวนการระบุความสัมพันธ์ระหว่าง Entity ใน Ontology ต่างชุด Entity ในที่นี้อาจเป็น Class, Property หรือ Individual ก็ได้ ส่วนผลที่ได้จากกระบวนการเรียกว่า Mapping หรือ correspondence เช่น Class หนึ่งเป็น Subclass ของอีก Class หนึ่ง Property สองตัวมีความหมายเท่ากัน หรือ Individual สองรายการอ้างถึงสิ่งเดียวกัน จุดประสงค์ของ Alignment ไม่ใช่ทำให้ Ontology ทุกชุดกลายเป็นแบบจำลองเดียวกัน แต่ทำให้ระบบรู้ว่าควรแปลและเชื่อมความหมายตรงใด ส่วนใดควรรักษาความแตกต่างไว้ และข้อสรุปใดอนุมานได้อย่างปลอดภัย

ในบางกรณี Ontology เฉพาะโดเมนสองชุดไม่มีคำระดับกลางร่วมกัน การเชื่อมโดยตรงจึงทำได้ยาก Upper Ontology หรือ top-level Ontology ช่วยจัดหมวดความหมายพื้นฐาน เช่น Object, Event, quality,

role และ situation เพื่อให้ concept จากต่างโดเมนมีจุดอ้างอิงร่วมกัน เส้นทางจาก Ontology เฉพาะโดเมนไปยังมาตรฐานโดเมนและหมวดระดับบนแสดงในภาพที่ 7.1



ภาพที่ 7.1: กรอบพิจารณา alignment จาก MSSN-Onto ผ่าน SSN/SOSA ไปยัง DUL และ DOLCE

ภาพที่ 7.1 แสดงกรอบพิจารณาการเชื่อม MSSN-Onto ซึ่งเป็น Ontology เฉพาะโดเมนเข้ากับ SSN/SOSA ซึ่งเป็นมาตรฐานด้าน Sensor ก่อนพิจารณาหมวดอ้างอิงระดับบนจาก DUL และ DOLCE เส้นทางนี้ไม่ได้หมายความว่าทุก Class ของ MSSN-Onto ต้องมี Mapping ไปยัง DUL หรือ DOLCE และไม่ได้อ้างว่า Mapping ทุกเส้นถูกประกาศไว้ในงาน MSSN-Onto เดิม ภาพนี้ใช้แยกให้เห็นสามระดับของการตัดสินใจ ได้แก่ การรักษารายละเอียดเฉพาะโดเมน การใช้คำศัพท์ร่วมของโดเมน และการวิเคราะห์ category ระดับบนเมื่อจำเป็นต่อการทำงานร่วมกัน

7.2 Alignment ในรูปความสัมพันธ์ทางคณิตศาสตร์

เมื่อทราบเหตุผลของการเชื่อม Ontology แล้ว ขั้นต่อไปคือต้องระบุให้ชัดว่าการเชื่อมนั้นกล่าวอ้างความสัมพันธ์ชนิดใด มุมมองเชิงคณิตศาสตร์ช่วยแยก Equivalence, Subsumption และความสัมพันธ์ที่มีทิศทางออกจากการจับคู่ชื่อที่ดูคล้ายกัน

หากให้ O_1 และ O_2 แทน Ontology ต้นทางและปลายทาง และให้ E_1 กับ E_2 เป็นเซตของ Entity ที่อยู่ใน Ontology ทั้งสองชุดตามลำดับ ผลของ Alignment สามารถเขียนเป็นเซต A ได้ว่า

$$A \subseteq E_1 \times E_2 \times R \times [0, 1]$$

สมาชิกหนึ่งรายการของ A เขียนได้เป็น $\langle e_1, e_2, r, c \rangle$ โดย $e_1 \in E_1$ คือ Entity ต้นทาง, $e_2 \in E_2$ คือ Entity ปลายทาง, $r \in R$ คือชนิดของความสัมพันธ์ และ $c \in [0, 1]$ คือค่าความมั่นใจเมื่อกระบวนการ Alignment ต้องใช้คะแนนประกอบการพิจารณา ค่า c ไม่ใช่ส่วนจำเป็นของ Axiom ใน OWL และไม่ควรถูกตีความเป็นความจริงทางตรรกะโดยอัตโนมัติ แต่มีประโยชน์ในขั้นเสนอ Candidate Mapping เพื่อจัดลำดับให้ผู้เชี่ยวชาญตรวจ ส่วนที่มีผลต่อ Reasoning โดยตรงคือชนิดของความสัมพันธ์ r

สมมติว่า C เป็น Class ใน Ontology ต้นทางและ D เป็น Class ใน Ontology ปลายทาง การประกาศว่า C เป็น Subclass ของ D เขียนเป็น $C \sqsubseteq D$ และอ่านในรูป First-Order Logic ได้ว่า

$$\forall x \left(C(x) \rightarrow D(x) \right)$$

ตัวแปร x แทน Individual ใด ๆ ในโดเมนของการตีความ, $C(x)$ หมายถึง x เป็นสมาชิกของ Class C, และ $D(x)$ หมายถึง x เป็นสมาชิกของ Class D สมการจึงกล่าวว่า Individual ทุกตัวที่เป็นสมาชิกของ C ต้องเป็นสมาชิกของ D ด้วย แต่สมาชิกของ D ไม่จำเป็นต้องเป็นสมาชิกของ C หากประกาศว่า Class ทั้งสองเทียบเท่ากันด้วย $C \equiv D$ ความหมายจะเข้มกว่าและเขียนได้ว่า

$$\forall x \left(C(x) \leftrightarrow D(x) \right)$$

เครื่องหมาย $\leftrightarrow$ ทำให้ข้ออ้างมีผลทั้งสองทิศ กล่าวคือสมาชิกของ C ต้องเป็นสมาชิกของ D และสมาชิกของ D ตองเป็นสมาชิกของ C ดวย ความแตกต่างนี้เป็นเหตุผลว่าทำไม **owl:equivalentClass** จึงต้องใช้เมื่อมีหลักฐานว่านิยามและขอบเขตตรงกันจริงเท่านั้น ส่วน **owl:sameAs** ยิ่งต้องระวัง เพราะเป็นการกล่าวว่า Individual สอง IRI อ้างถึงสิ่งเดียวกัน ไม่ใช่เพียงสิ่งที่คล้ายหรือเกี่ยวข้องกัน

7.3 DOLCE และ DUL ในฐานะกรอบจำแนกความหมาย

ความสัมพันธ์ทางคณิตศาสตร์บอกความแรงของ Mapping แต่ยังต้องมีเหตุผลว่า Entity สองรายการอยู่ใน Category เดียวกันหรือไม่ DOLCE และ DUL จึงถูกนำมาใช้เป็นกรอบตั้งคำถามต่อความหมายระดับบนก่อนประกาศ Axiom ที่มีผลต่อ Reasoning

DOLCE หรือ Descriptive Ontology for Linguistic and Cognitive Engineering เป็น foundational Ontology ที่เสนอหมวดพื้นฐานสำหรับอธิบายสิ่งต่าง ๆ ในโลก เช่นสิ่งที่คงเอกลักษณ์ของตนขณะเวลาเปลี่ยน เหตุการณ์หรือกระบวนการที่ดำเนินไปตามเวลา และคุณภาพที่ต้องอาศัยสิ่งอื่นเป็นฐาน หมวดเหล่านี้มักอธิบายด้วยคำว่า endurant, perdurant และ quality ตามลำดับ จุดประสงค์ของการแบ่งไม่ได้อยู่ที่การเพิ่มศัพท์ปรัชญาให้ Ontology แต่ช่วยป้องกันการนำอุปกรณ์ เหตุการณ์ บทบาท และค่าที่สังเกตได้มาปะปนเป็นสิ่งชนิดเดียวกัน (Masolo et al., 2003; Gangemi et al., 2002)

DOLCE+DnS Ultralite หรือ DUL เป็น Ontology ที่ลดความซับซ้อนของแนวคิดดังกล่าวและเชื่อมกับแนวทาง Descriptions and Situations ทำให้สามารถอธิบาย Object, Event, situation, role, quality และ information Object ในกรอบเดียวกันได้ ตัวอย่างเช่น บุคคลหนึ่งเป็น Object หรือ Agent แต่คำว่า “ผู้ตรวจ” เป็น role ที่บุคคลนั้นทำหน้าที่ภายใต้ situation หนึ่ง การแยก Agent ออกจาก role ช่วยให้บุคคลเดียวกันมีบทบาทต่างกันตามกระบวนการและช่วงเวลาโดยไม่ต้องกำหนดว่าบุคคลนั้นเป็นผู้ตรวจอย่างถาวร ในทำนองเดียวกัน กล้องเป็น Object การสังเกตเป็นเหตุการณ์หรือกิจกรรม ผลภาพเป็นข้อมูลที่เกิดจากการสังเกต และคำอธิบายรูปแบบเหตุการณ์เป็น information Object การแยกเช่นนี้ทำให้การ Query และการตรวจ Provenance ชัดขึ้น (Gangemi, n.d., 2008)

หมวดของ DUL ที่ใช้บ่อยและคำถามที่ควรถามก่อน Alignment สรุปในตารางที่ 7.1

ตารางที่ 7.1: แนวทางพิจารณาหมวดสำคัญของ DUL ก่อนทำ Alignment

หมวดระดับบน	ความหมายที่ต้องแยกให้ชัด	คำถามสำหรับตรวจ concept ต้นทาง
dul:Object	สิ่งที่คงเอกลักษณ์และเข้าร่วมในเหตุการณ์ได้	เป็นตัวอุปกรณ์ บุคคล หรือวัตถุ ไม่ใช่กิจกรรมที่สิ่งนั้นทำใช่หรือไม่
dul:Event	สิ่งที่เกิดขึ้นหรือดำเนินไปตามเวลา	มีช่วงเวลาและผู้เข้าร่วม หรือเป็นเพียงระเบียนที่บรรยายเหตุการณ์
dul:Situation	บริบทที่ทำให้สิ่งและเหตุการณ์ชุดหนึ่งถูกอธิบายร่วมกัน	concept นี้เป็นเหตุการณ์เอง หรือเป็นบริบทสำหรับตีความเหตุการณ์
dul:Role	บทบาทที่ Object หรือ Agent ทำหน้าที่ภายใต้บริบทหนึ่ง	เป็นบทบาทที่เปลี่ยนได้ หรือเป็นตัวบุคคลและหน่วยงาน
dul:InformationObject	เนื้อหาข้อมูลที่แยกจากสื่อหรือสิ่งที่ใช้แสดงเนื้อหานั้น	กำลังกล่าวถึงเนื้อหารายงานหรือนิยาม แทนที่จะเป็นตัวสื่อที่จัดเก็บใช่หรือไม่

ตารางที่ 7.1 แสดงว่า upper Ontology มีประโยชน์เมื่อช่วยให้ทีมตั้งคำถามต่อ category ของ concept ก่อนประกาศ Mapping แต่ไม่ได้ให้คำตอบตายตัวจากชื่อเพียงอย่างเดียว คำว่า Event ในระบบหนึ่งอาจหมายถึงเหตุการณ์ในโลกจริง ขณะที่อีกระบบใช้แทนข้อความแจ้งเตือนที่บรรยายเหตุการณ์ หากจับคู่ทั้งสองคำกับ **dul:Event** โดยไม่ดูนิยามและ Instance ระบบจะรวมสิ่งที่อยู่คนละ category เข้าด้วยกัน DUL จึงควรถูกใช้เป็นกรอบวิเคราะห์ ไม่ใช่รายการ Superclass ที่ต้องเติมให้ครบทุก Class

7.4 จาก Candidate Mapping สู่ Axiom ที่ตรวจสอบได้

กรอบจาก DOLCE/DUL ช่วยเสนอ Candidate Mapping แต่ Candidate ยังไม่ควรถูกนำเข้า Ontology ทันที หัวข้อนี้จึงอธิบายขั้น Review ตั้งแต่ Competency Questions, ตัวอย่างบวกและลบ ไปจนถึง Rationale ที่ทำให้ Mapping ตรวจสอบและเปลี่ยนแปลงภายหลังได้

การทำ Alignment ควรเริ่มจาก Competency Question ของการรวมข้อมูล เช่น ต้องการค้น Observation จากหลายระบบร่วมกันหรือไม่ ต้องการรวมอุปกรณ์จากหลายผู้ผลิตภายใต้คำว่า Sensor หรือไม่ หรือต้องการเชื่อมเหตุการณ์กับเอกสารหลักฐานข้ามโดเมน คำถามเหล่านี้กำหนดขอบเขตของ Alignment และช่วยตัด Mapping ที่ไม่ก่อประโยชน์ต่อการใช้งานออก หลังจากนั้นจึงรวบรวม Candidate Mapping จากชื่อ คำนิยาม Axiom ตัวอย่าง Instance และ Query ที่ต้องการรองรับ เครื่องมืออัตโนมัติอาจช่วยเสนอคู่ที่น่าจะสัมพันธ์กัน แต่การอนุมัติ Mapping ที่มีผลต่อ Reasoning ยังต้องอาศัยผู้เชี่ยวชาญโดเมนและ Ontology engineer ร่วมกัน (Euzenat and Shvaiko, 2013)

หลักฐานขั้นต่ำของ Mapping หนึ่งรายการควรมีสามส่วน

- คำนิยามของ Entity ทั้งสองฝั่ง
- ตัวอย่างข้อมูลที่แสดงว่าแต่ละ Entity ถูกใช้อย่างไร
- Query หรือกรณีใช้งานที่ได้รับประโยชน์จาก Mapping

หากหลักฐานบอกได้เพียงว่าชื่อคล้ายกัน ความสัมพันธ์เชิงคำศัพท์อย่าง **skos:closeMatch** อาจเหมาะกว่าการประกาศ Axiom เชิงตรรกะ หากสมาชิกของ Class ต้นทางทุกตัวอยู่ในขอบเขตของ Class ปลายทาง แต่ทิศกลับไม่จริง ควรใช้ **rdfs:subClassOf** ส่วน **owl:equivalentClass** เหมาะเฉพาะเมื่อความหมายตรงกันทั้งสองทิศ การเลือก Relation ที่ไม่ก่อผลอนุมานเกินหลักฐานช่วยลดความเสี่ยงจาก Mapping ที่ยังยืนยันไม่ได้

ผลการพิจารณาควรถูกบันทึกใน Mapping Specification ซึ่งมีอย่างน้อย Source Entity, target Entity, Relation, Rationale, ผู้ตรวจ และสถานะการอนุมัติ หากระบบใช้คะแนนความมั่นใจ คะแนนนั้นควรใช้จัดลำดับ Candidate สำหรับการทบทวน มากกว่าใช้แทนคำตัดสินของผู้เชี่ยวชาญ ที่สำคัญคือ Specification ควรบันทึกเหตุผลของการไม่ align ด้วย เช่น Entity หนึ่งไม่มีคู่ที่มีขอบเขตตรงกัน หรือ Mapping นั้นไม่ช่วยตอบ Competency Question ใด การตัดสินใจรักษา concept ไว้ใน Ontology เฉพาะโดเมนจึงเป็นผลลัพธ์ที่ถูกต้องได้ ไม่ใช่ช่องว่างที่ต้องเติมเสมอ

7.5 กรณี MSSN-Onto: สิ่งที่มีอยู่จริงและสิ่งที่ต้องพิจารณาเพิ่ม

ขั้นตอนจาก Candidate ไปสู่ Axiom จะเห็นได้ชัดเมื่อกลับมายัง MSSN-Onto กรณีนี้ต้องแยกระหว่าง Mapping ที่ผลงานเดิมประกาศไว้จริง กับ Mapping ที่เป็นเพียงข้อเสนอสำหรับการวิเคราะห์เพิ่มเติม เพื่อไม่ให้การอธิบายในหนังสือกลายเป็นการกล่าวอ้างเกินหลักฐานของงานต้นทาง

บทที่ 6 อธิบายแล้วว่า MSSN-Onto ถูกออกแบบเพื่อรองรับข้อมูลและการประมวลผลเหตุการณ์ใน Multimedia Sensor Network โดยนำ SOSA/SSN และ MA-Ont มาใช้เป็นฐาน ส่วนสำคัญของแบบจำลองจึงมีจุดเชื่อมกับมาตรฐานตั้งแต่ขั้นออกแบบ ไม่ได้เริ่มจากการสร้าง Ontology ปิดแล้วจึงค่อยจับคู่ชื่อในภายหลัง จากแบบจำลองที่เผยแพร่ สามารถยืนยันความสัมพันธ์หลักได้ว่า **mssn:MediaSensor** เป็น Sensor ชนิดหนึ่งภายใต้ SOSA และ **mssn:MultimediaData** เป็น result ชนิดหนึ่ง ความสัมพันธ์สองรายการเขียนเป็น Turtle ได้ดังนี้ (Angsuchotmetee et al., 2020; World Wide Web Consortium, 2017; Haller et al., 2019)

```
@prefix mssn: <https://mssn.sigappfr.org/mssn-onto#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix sosa: <http://www.w3.org/ns/sosa/> .

mssn:MediaSensor rdfs:subClassOf sosa:Sensor .
mssn:MultimediaData rdfs:subClassOf sosa:Result .
```

บรรทัดแรกทำให้ Instance ของ **mssn:MediaSensor** ถูกอ่านเป็น **sosa:Sensor** ได้ด้วย จึงสามารถใช้ Query ที่เขียนบนคำศัพท์มาตรฐานค้น Sensor เฉพาะของ MSSN-Onto ได้โดยไม่ต้องประกาศว่า Sensor ทุกชนิดเป็น media Sensor บรรทัดที่สองทำหน้าที่ในลักษณะเดียวกันสำหรับผลข้อมูลสื่อ ความสัมพันธ์ทั้งสองจึงรักษารายละเอียดเฉพาะโดเมนไว้และทำให้ระบบภายนอกอ่านข้อมูลผ่าน SOSA ได้ อย่างไรก็ตาม Class อย่าง **mssn:MediaSegment** และ **mssn:MediaDescriptor** มีหน้าที่เฉพาะต่อการอธิบายส่วนของสื่อและคุณลักษณะของสื่อ การไม่บังคับให้ Class เหล่านี้มีคู่เทียบใน SSN/SOSA เป็นการรักษาขอบเขตของแต่ละมาตรฐาน ไม่ใช่ข้อบกพร่องของ Alignment

เมื่อพิจารณา DUL ต้องแยก “ข้อเท็จจริงจาก Ontology ที่เผยแพร่” ออกจาก “ข้อเสนอสำหรับการวิเคราะห์เพิ่มเติม” ให้ชัด งาน MSSN-Onto แสดงการต่อยอดจาก SOSA/SSN แต่ไม่ได้ทำให้ทุก concept

เฉพาะของ MSSN-Onto กลายเป็น Subclass ของ DUL โดยอัตโนมัติ หากระบบใหม่ต้องเชื่อมเหตุการณ์ บริบท บทบาท หรือวัตถุข้อมูลกับโดเมนอื่น ผู้พัฒนาจึงอาจใช้หมวดของ DUL เป็นกรอบตรวจว่า concept แต่ละตัวอยู่ใน category ใด แล้วสร้าง Alignment module แยกต่างหากหลังจากตรวจนิยามและข้อมูลจริง การกล่าวว่า concept “อาจพิจารณาเทียบกับ **dul:Event**” จึงต่างจากการประกาศ Axiom ว่า concept นั้นเป็น Subclass ของ **dul:Event** อย่างสิ้นเชิง ข้อแรกเป็นคำถามออกแบบ ส่วนข้อหลังเป็นข้ออ้างทางตรรกะที่ต้องมีหลักฐานและการทดสอบรองรับ

การตัดสินใจที่ยืนยันได้และการตัดสินใจที่ยังควรรักษาไว้ในโดเมนสรุปในตารางที่ 7.2

ตารางที่ 7.2: การตัดสินใจ Alignment และ non-Alignment สำหรับ MSSN-Onto

Source	Target	Relation	Rationale
mssn:MediaSensor	**sosa:Sensor**	Subclass	เป็น Sensor เฉพาะสำหรับสื่อ จึงแคบกว่า Sensor ตาม SOSA
mssn:MultimediaData	**sosa:Result**	Subclass	เป็นผลข้อมูลจากกระบวนการสังเกตที่เก็บรายละเอียดของสื่อเพิ่มขึ้น
mssn:MediaSegment	ไม่มีคู่ตรง	local Class	ใช้อธิบายส่วนของข้อมูลสื่อ ซึ่งไม่ใช่ขอบเขตหลักของ SSN/SOSA
mssn:MediaDescriptor	ไม่มีคู่ตรง	local Class	เก็บ descriptor เฉพาะสื่อและควรรักษาความหมายในโดเมนเดิม

ตารางที่ 7.2 แสดงว่าคุณภาพของ Alignment ไม่ได้วัดจากจำนวน Mapping แต่ดูจากความถูกต้องและประโยชน์ของแต่ละรายการ สองแถวแรกทำให้ข้อมูล MSSN-Onto เชื่อมกับคำศัพท์มาตรฐานได้ ส่วนสองแถวหลังบอกอย่างเปิดเผยว่ารายละเอียดเฉพาะสื่อยังไม่มีเหตุผลพอที่จะบังคับเข้ากับ SSN/SOSA หากในอนาคตต้องเชื่อม MediaSegment กับ Ontology ด้านสื่ออื่น จึงค่อยสร้าง Mapping ใหม่ใน module ที่ระบุ Version และ Rationale ของตนเอง แนวทางนี้ช่วยให้ Ontology หลักไม่รับ Axiom ที่ยังอยู่ระหว่างการทดลอง

การทดสอบประโยชน์ของ Subclass Mapping ทำได้ด้วย Query บน Class ระดับมาตรฐาน เช่น Query ต่อไปนี้ค้น Sensor ผ่าน **sosa:Sensor** โดยไม่ต้องรู้ล่วงหน้าว่า Instance นั้นถูกระบุเป็น **mssn:MediaSensor**

```
PREFIX sosa: <http://www.w3.org/ns/sosa/>

SELECT ?sensor
WHERE {
 ?sensor a sosa:Sensor .
}
```

หาก Reasoner ประมวลผล **rdfs:subClassOf** หรือระบบ materialize Inference ไว้แล้ว Query นี้ควรคืน Instance ของ **mssn:MediaSensor** ด้วย การทดสอบกรณีบวกจึงตรวจว่า Mapping เพิ่ม coverage ได้ตามเป้าหมาย ขณะเดียวกันต้องมีกรณีลบเพื่อตรวจว่า Instance ของ **sosa:Sensor** ทั่วไปไม่ได้ถูกอนุมานย้อนกลับให้เป็น **mssn:MediaSensor** เพราะ Subclass มีผลเพียงทิศเดียว หากผลทดสอบไม่เป็นเช่นนี้ ต้องตรวจทั้ง Axiom, การตั้งค่า Reasoner และข้อมูลที่โหลดเข้าระบบก่อนสรุปว่า Alignment ใช้งานได้

7.6 การประเมิน Versioning และ Governance ของ Alignment

แม้ Mapping จะผ่านการตรวจใน Version หนึ่ง ความหมายของ Ontology ต้นทางและปลายทางยังเปลี่ยนได้ Alignment จึงต้องมี Versioning, Test และ Governance ของตนเอง หัวข้อนี้เชื่อมความถูกต้องของ Axiom เข้ากับวงจรดูแล Artifact ในระยะยาว

Alignment ที่ผ่านการทบทวน แล้วควรถูกจัดเก็บเป็น Artifact แยกจาก Ontology หลัก เพราะวงจรการเปลี่ยนของสองส่วนนี้อาจไม่เท่ากัน Ontology เฉพาะโดเมนอาจคงเดิม แต่ Ontology ปลายทางออก Version ใหม่และเปลี่ยนนิยามของ Class ที่ใช้เป็น target หรือทีมอาจพบจากข้อมูลจริงว่า Mapping เดิมกว้างเกินไป การแยกไฟล์ช่วยให้ทบทวนหรือย้อน Version ของ Alignment ได้โดยไม่ต้องเปลี่ยน Ontology หลักทุกครั้ง แต่ละ Release ควรระบุ Ontology ต้นทางและปลายทางพร้อม Version IRI, รายการ Mapping ที่เพิ่มหรือถอน, ผู้อนุมัติ และผลการทดสอบที่เกี่ยวข้อง

การประเมินควรครอบคลุมอย่างน้อยสี่ด้าน

- **ความถูกต้องของความหมาย** ให้ผู้เชี่ยวชาญตรวจคำนิยามและ Instance
- **ความสอดคล้องทางตรรกะ** ใช้ Reasoner ตรวจ Inconsistency หรือ Unsatisfiable Class
- **ผลต่อ Query** เปรียบเทียบผลก่อนและหลังโหลด Alignment
- **ความครอบคลุมของกรณีใช้งาน** ย้อนกลับไปยัง Competency Question ว่า Mapping ช่วยตอบคำถามข้ามระบบได้จริงหรือไม่

Alignment ที่ไม่สร้าง Inconsistency แต่ไม่มี Query หรือระบบใดใช้ อาจถูกต้องทางเทคนิคแต่ยังไม่มีเหตุผลพอสำหรับการดูแลระยะยาว

ในระบบหลายโดเมน ไม่จำเป็นต้องให้ Ontology ทุกชุดอ้าง upper Ontology โดยตรง องค์กรอาจมี core Vocabulary ระดับกลาง เช่น Project, Organization, Document, Evidence, Activity, Location และ Agent แล้วให้ Ontology เฉพาะโดเมนเชื่อมกับ Vocabulary นี้ ส่วน core Vocabulary จึงพิจารณาเชื่อมกับ DUL อีกชั้นหนึ่ง วิธีดังกล่าวลดการพึ่งพา Ontology ระดับบนในทุก module และทำให้ทีมโดเมนยังอ่านแบบจำลองของตนเองได้ง่าย อย่างไรก็ตาม จุดเชื่อมทุกชั้นยังต้องมี owner, Version, Rationale และ test เช่นเดิม ไม่เช่นนั้น Triple Store อาจเก็บ Graph หลายโดเมนไว้ด้วยกันได้ แต่ความหมายยังไม่เชื่อมกันและ Query ข้ามโดเมนไม่ได้อย่างน่าเชื่อถือ

ข้อผิดพลาดที่ควรหลีกเลี่ยงจึงมีทั้งการจับคู่จากชื่อ การใช้ Equivalence มากเกินหลักฐาน การ align ทุก Class เพื่อให้ตารางดูครบ การไม่ทดสอบกับ Instance จริง และการไม่มีผู้รับผิดชอบเมื่อ Ontology ปลายทางเปลี่ยน อีกกรณีที่พบได้บ่อยคือใช้ Mapping สำหรับอธิบายความใกล้เคียงเชิงเอกสารเป็น Axiom สำหรับ Reasoner ทั้งที่ต้องการเพียงช่วยให้คนค้นคำศัพท์ การแยก Mapping เชิงคำศัพท์ออกจาก Mapping เชิงตรรกะ และเริ่มจากเฉพาะ Mapping ที่จำเป็นต่อคำถาม จะทำให้ Alignment ตรวจสอบและบำรุงรักษาได้มากกว่าการสร้างความสัมพันธ์จำนวนมากในครั้งเดียว

7.7 สรุป

บทนี้เริ่มจากความจำเป็นในการเชื่อม Ontology แล้วค่อยเพิ่มชนิดความสัมพันธ์ กรอบ Category, การทบทวน และ Versioning ลำดับดังกล่าวชี้ว่า Alignment ที่น่าเชื่อถือเกิดจาก Rationale และ Test มากกว่าจำนวน Mapping ที่ประกาศไว้

Upper Domain Ontology และ Ontology Alignment ช่วยให้ Ontology เฉพาะโดเมนทำงานร่วมกับ Ontology และข้อมูลจากระบบอื่นโดยไม่ต้องละทิ้งรายละเอียดของตนเอง DOLCE และ DUL ให้กรอบสำหรับจำแนก category ระดับบน ส่วนมาตรฐานโดเมนอย่าง SSN/SOSA ให้คำศัพท์ร่วมที่ใกล้กับข้อมูลใช้งานจริง กรณี MSSN-Onto แสดงให้เห็นว่าการต่อยอดจากมาตรฐานด้วย Subclass ที่มีความหมายชัดสามารถเพิ่ม interoperability ได้ ขณะเดียวกัน Class เฉพาะอย่าง MediaSegment และ MediaDescriptor ยังควรอยู่ในขอบเขตของโดเมนจนกว่าจะมีเหตุผลและหลักฐานสำหรับ Mapping เพิ่มเติม หลักสำคัญจึงไม่ใช่การเชื่อมให้มากที่สุด แต่เป็นการเชื่อมเท่าที่นิยาม ข้อมูล และ Query รองรับ พร้อมบันทึก Rationale และผลทดสอบทุกครั้ง บทถัดไปจะนำ Ontology, Mapping และ Query ที่ได้จากกระบวนการเหล่านี้ไปประกอบเป็นแอปพลิเคชัน Knowledge Graph ที่มีข้อมูลไหลผ่านอย่างเป็นระบบ

เอกสารอ้างอิง

Angsuchotmetee, C., Chbeir, R., and Cardinale, Y. (2020). MSSN-Onto: An ontology-based approach for flexible event processing in multimedia sensor networks. *Future Generation Computer Systems*, 108:1140–1158.

Euzenat, J. and Shvaiko, P. (2013). *Ontology Matching*. Springer, 2 edition.

Gangemi, A. (2008). Norms and plans as unification criteria for social collectives. *Autonomous Agents and Multi-Agent Systems*, 17(1):70–112.

Gangemi, A. (n.d.). DOLCE+DnS Ultralite (version 3.34). Ontology Design Patterns, ontology specification. Archived June 7, 2021, by DBpedia Archivo. Accessed October 18, 2026, from **https://akswnc7.informatik.uni-leipzig.de/dstreitmatter/archivo/ontologydesignpatterns.org/ont--dul--DUL--owl/2021.06.07-182648/ont--dul--DUL--owl_type=pyLodeDoc.html**.

Gangemi, A., Guarino, N., Masolo, C., Oltramari, A., and Schneider, L. (2002). Sweetening ontologies with DOLCE. In Gómez-Pérez, A. and Benjamins, V. R., editors, *Knowledge Engineering and Knowledge Management: Ontologies and the Semantic Web*, pages 166–181. Springer.

Haller, A., Janowicz, K., Cox, S. J. D., Lefrançois, M., Taylor, K., Le Phuoc, D., Lieberman, J., García-Castro, R., Atkinson, R., and Stadler, C. (2019). The modular SSN ontology: A joint W3C and OGC standard specifying the semantics of sensors, observations, sampling, and actuation. *Semantic Web*, 10(1):9–32.

Kalfoglou, Y. and Schorlemmer, M. (2003). Ontology mapping: The state of the art. *The Knowledge Engineering Review*, 18(1):1–31.

Masolo, C., Borgo, S., Gangemi, A., Guarino, N., and Oltramari, A. (2003). WonderWeb deliverable D18: Ontology library (final). Technical report, IST Project 2001-33052 WonderWeb.

World Wide Web Consortium (2017). Semantic sensor network ontology. W3C Recommendation. Accessed June 18, 2026, from **https://www.w3.org/TR/vocab-ssn/**.

8

การสร้างแอปพลิเคชัน Knowledge Graph

หลังจากรู้วิธีสร้าง Graph และถาม Graph แล้ว คำถามที่สำคัญกว่าสำหรับงานจริงคือจะทำให้สิ่งเหล่านี้กลายเป็นระบบที่ใช้งานได้อย่างไร บทนี้จึงเชื่อมมาตรฐานก่อนหน้ากับสถาปัตยกรรมของแอปพลิเคชัน โดยมอง Knowledge Graph ไม่ใช่ไฟล์ RDF เพียงชุดเดียว แต่เป็น Pipeline ที่เริ่มจากข้อมูลต้นทาง ผ่าน การ Mapping, Validation, Storage และ Query ก่อนจะกลายเป็นบริการที่ผู้ใช้ปลายทางใช้ได้จริง

เพื่อให้เห็นหน้าที่ของแต่ละส่วนและวิธีทำงานร่วมกันอย่างชัดเจน บทนี้แบ่งการอธิบายออกเป็นสามช่วง

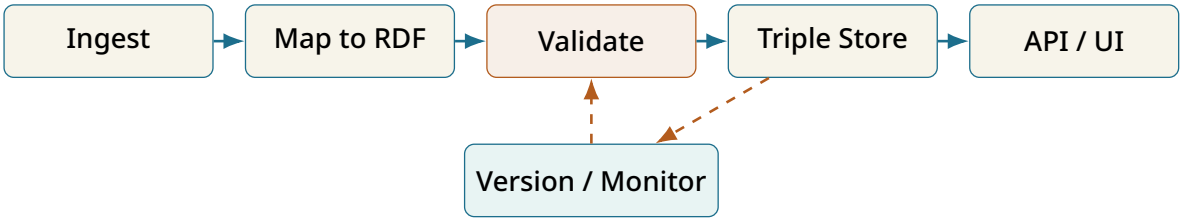
1. **ส่วนประกอบของ Pipeline** อธิบายหน้าที่ของ Source, Extraction, Mapping, Validation, Storage และส่วนติดต่อผู้ใช้
2. **การประกอบเป็นระบบ** พิจารณาการยกระดับ Prototype, Artifact ขั้นต่ำ และติดตาม Record หนึ่งรายการตั้งแต่ Source จนถึงคำตอบ
3. **การนำขึ้นใช้และดูแลต่อเนื่อง** ตรวจข้อผิดพลาด, Metric, Regression Test และ Compatibility ระหว่าง Version

สามช่วงนี้ตอบคำถามคนละระดับ ช่วงแรกตอบว่าแต่ละ Layer ทำอะไร ช่วงที่สองตอบว่า Layer เหล่านั้นทำงานต่อกันอย่างไร และช่วงสุดท้ายตอบว่าจะรู้ได้อย่างไรว่าระบบยังทำงานถูกต้องหลัง Deployment

8.1 จาก Ontology ไปสู่ระบบที่ใช้งานได้จริง

บทก่อนอธิบาย Ontology Alignment เพื่อเชื่อมคำศัพท์และความหมายระหว่าง Ontology ต่างชุด ส่วนการสร้างระบบที่ใช้งานจริงต้องพิจารณาต่อว่าข้อมูลจะเดินทางจากแหล่งต้นทางไปสู่คำตอบได้อย่างไร หัวข้อนี้จึงขยายจากไฟล์ Ontology ไปสู่แอปพลิเคชันที่มีการแปลงข้อมูล การตรวจสอบข้อมูล การจัดเก็บ และบริการสืบค้นทำงานร่วมกัน

หลังจากเข้าใจ RDF, RDFS, OWL และ SPARQL แล้ว คำถามสำคัญคือจะนำส่วนเหล่านี้ไปสร้างระบบจริงได้อย่างไร แอปพลิเคชัน Knowledge Graph ไม่ได้เกิดจากการมี Ontology อย่างเดียว แต่เกิดจากการออกแบบ Pipeline ที่ทำให้ข้อมูลจากหลายแหล่งถูกแปลง ตรวจสอบ จัดเก็บ สืบค้น และนำไปใช้ได้อย่างต่อเนื่อง (Hogan et al., 2022; Sequeda and Lassila, 2021) ภาพรวมของ Pipeline ตั้งแต่ข้อมูลต้นทางจนถึงบริการสำหรับผู้ใช้แสดงในภาพที่ 8.1



Knowledge Graph Application คือ Pipeline ที่ต้องดูแลข้อมูลและ Ontology ไปพร้อมกัน

ภาพที่ 8.1: สถาปัตยกรรมปฏิบัติของ Knowledge Graph Application ตั้งแต่ Ingest ถึง Governance

ภาพที่ 8.1 แสดงว่า Knowledge Graph Application ต้องพิจารณา Ingestion, Mapping, Validation, Storage และ Service Layer เป็น Pipeline เดียวกัน ไม่ใช่ดูเฉพาะไฟล์ Ontology ระบบหนึ่งอาจเริ่มจากไฟล์ CSV รายงาน PDF ฐานข้อมูล SQL หรือข้อมูลจาก Sensor จากนั้นต้องมีขั้นตอน Mapping เพื่อแปลงข้อมูลเหล่านั้นเป็น RDF ต้องมีขั้นตอน Validation เพื่อตรวจว่าข้อมูลเป็นไปตามรูปแบบที่ต้องการ ต้องมี Triple Store สำหรับจัดเก็บ Graph และต้องมี Query หรือ API สำหรับนำข้อมูลไปใช้ใน Application หากขาดขั้นตอนใดขั้นตอนหนึ่ง ระบบอาจมี Ontology ที่ดีแต่ไม่สามารถใช้งานจริงได้

8.2 Pipeline พื้นฐานของ Knowledge Graph

เมื่อมอง Application เป็นกระบวนการหลายชั้น ขั้นต่อไปคือกำหนด Input, Output และ Artifact ของแต่ละชั้นให้ตรวจสอบได้ หัวข้อนี้จึงใช้สมการอย่างย่อเพื่อแสดงการไหลของข้อมูล ไม่ได้ใช้สมการแทนรายละเอียดของ Implementation

Pipeline พื้นฐานสามารถมองเป็นฟังก์ชันหลายขั้นตอนต่อกัน หาก D คือข้อมูลดิบ, M คือ Mapping, V คือ Validation และ Q คือ Query แล้วระบบหนึ่งอาจเขียนอย่างย่อได้ว่า

$$D \xrightarrow{M} G \xrightarrow{V} G' \xrightarrow{Q} R$$

ในสมการนี้ D คือข้อมูลดิบจากแหล่งต้นทาง เช่น ตาราง เอกสาร หรือ Sensor Log, M คือกระบวนการ Mapping ที่แปลงข้อมูลเหล่านั้นเป็น RDF, G คือ RDF Graph ที่ได้หลัง Mapping, V คือกระบวนการ Validation, G' คือ Graph ที่ผ่านการตรวจสอบหรือถูกจัดให้อยู่ในสถานะพร้อมใช้, Q คือ Query หรือ API ที่เรียกใช้ Graph และ R คือผลลัพธ์ที่ Application นำไปใช้ได้ ลูกศรในสมการจึงไม่ได้หมายถึงการคำนวณทางคณิตศาสตร์อย่างเดียว แต่หมายถึงขั้นตอนการทำงานที่ข้อมูลไหลผ่าน สมการนี้ช่วยให้เห็นว่า Knowledge Graph ไม่ใช่เพียงไฟล์ RDF แต่เป็นกระบวนการผลิตความรู้ที่ต้องควบคุมตั้งแต่ข้อมูลต้นทางจนถึงผลลัพธ์ปลายทาง หากแปลงสมการนี้เป็น Artifact ของระบบจริง จะเห็นชั้นงานชัดขึ้นดังตารางที่ 8.1

ใน Walkthrough ช่วงท้ายบท สัญลักษณ์เหล่านี้จะมีค่าที่จับต้องได้: D คือ Source Record ของ Project **P001**, G คือ Staging Graph, G' คือ Graph ที่ผ่าน SHACL, และ R คือคำตอบพร้อม Evidence Bundle การระบุเช่นนี้ช่วยไม่ให้สมการถูกอ่านเป็น Pipeline นามธรรมที่ไม่สัมพันธ์กับข้อมูลจริง

ตารางที่ 8.1: Artifact สำคัญในแต่ละชั้นของ Knowledge Graph Application

ชั้นงาน	Artifact ที่ควรมี	คำถามที่ใช้ตรวจ
Source	Source Inventory, Data Owner, Field Description	ข้อมูลนี้มาจากไหนและใครรับผิดชอบ
Mapping	Mapping Specification, IRI Policy	Field ใดกลายเป็น Resource, Literal หรือ Relation
Validation	SHACL Shape และ Validation Report	Graph ผ่านเงื่อนไขขั้นต่ำก่อน Publish หรือไม่
Storage	Named Graph, Version, Provenance	Graph ชุดนี้อยู่ในสถานะใดและมาจาก Source ใด
Query	Query Catalog และ Expected Result	Query สำคัญตอบ Competency Question ได้หรือไม่
Application	API Response, UI View, Evidence Bundle	ผู้ใช้เห็นคำตอบพร้อมหลักฐานที่ตรวจย้อนกลับได้หรือไม่

ตารางที่ 8.1 แสดงความต่างระหว่าง Knowledge Graph Application กับการสาธิต RDF ที่ยังไม่มีขั้นตอนควบคุมข้อมูล เพราะทุกชั้นของ Application มี Artifact ที่ Review ได้ หาก Query ให้คำตอบผิด ทีมสามารถตรวจย้อนจากผลลัพธ์กลับไปยัง Query, Graph, Mapping และ Source ได้ ไม่ต้องเดาว่าความผิดเกิดจาก Ontology หรือข้อมูลต้นทาง ในกรณีข้อมูลตาราง R2RML สามารถใช้เป็นมาตรฐานสำหรับ Mapping จาก Relational Database ไปยัง RDF ได้ (World Wide Web Consortium, 2012) ส่วนในกรณีที่ข้อมูลมาจากเอกสารหรือระบบกึ่งโครงสร้าง อาจต้องมี Extraction Layer เพิ่มเติม แต่ไม่ว่าต้นทางจะเป็นแบบใด การตัดสินใจสำคัญยังเหมือนเดิม คืออะไรควรเป็น IRI อะไรควรเป็น Literal และ Relation ใดควรถูกใช้เพื่อเชื่อมข้อมูล

8.3 Identity, Validation และ Storage

Pipeline พื้นฐานแสดงลำดับของงาน แต่สามจุดที่มีผลต่อความน่าเชื่อถือของ Graph โดยตรงคือ Identity, Validation และการแยกพื้นที่จัดเก็บ หัวข้อนี้จึงพิจารณาสามส่วนร่วมกันก่อนแยกลงรายละเอียดของแต่ละ Layer

ปัญหาที่พบมากในระบบ Knowledge Graph คือ Identity ไม่คงที่ เช่น โครงการเดียวกันถูกสร้างเป็นหลาย IRI หรือองค์กรเดียวกันถูกเขียนด้วยหลายชื่อโดยไม่มี Mapping เข้าหากัน ดังนั้น IRI Policy จึงเป็นส่วนหนึ่งของสถาปัตยกรรมระบบ ไม่ใช่เพียงกติกาการตั้งชื่อ หาก Identity ไม่ชัด การรวม Graph ภายหลังจะให้ผลลัพธ์ที่ผิดหรือไม่ครบ หลังจากสร้าง RDF แล้วควรตรวจด้วย SHACL เพื่อดูว่าข้อมูลสำคัญครบตามที่ระบบต้องการหรือไม่ เช่น Project ต้องมี Label, ต้องมี Owner, และ Technology ต้องเป็น Resource ไม่ใช่ข้อความธรรมดา SHACL ทำหน้าที่เหมือน Contract ระหว่างข้อมูลกับ Application เพราะบอกว่าข้อมูลขั้นต่ำที่ระบบต้องการคืออะไร (World Wide Web Consortium, 2017; Labra Gayo et al., 2017)

ส่วนการจัดเก็บ Graph อาจใช้ Triple Store เช่น Apache Jena Fuseki, RDF4J, GraphDB หรือระบบอื่นที่รองรับ SPARQL จุดสำคัญคือการแยก Graph สำหรับข้อมูลจริง Graph สำหรับ Staging และ

Graph สำหรับ Evidence หรือ Provenance ออกจากกันให้ชัดเจน เมื่อข้อมูลผิดพลาดจะได้ตรวจย้อนกลับได้ว่ามาจาก Source ใดและเกิดจาก Mapping ขั้นตอนไหน (Carroll et al., 2005; World Wide Web Consortium, 2013)

จากภาพรวมข้างต้นจะเห็นได้ว่าแอปพลิเคชัน Knowledge Graph ไม่ใช่การนำ Ontology ไปวางไว้ข้างระบบเดิม แต่เป็นการออกแบบให้ Ontology, RDF Mapping, Validation, Storage และ Query ทำงานร่วมกันเป็นกระบวนการเดียวกัน รายละเอียดต่อไปนี้จะลงลึกขึ้นในแต่ละชั้นของ Pipeline เพื่อให้เห็นว่าเมื่อเริ่มทำระบบจริง ทีมควรเตรียม Artifact อะไร และควรตัดสินใจเรื่องใดก่อนหลัง

8.4 ชั้นข้อมูลต้นทางและการทำ Source Inventory

ก่อนทำ Mapping ต้องทราบว่าข้อมูลที่ Ontology ต้องการมีอยู่จริงใน Source ใด และ Source แต่ละแห่งมี Authority ต่อข้อมูลชนิดใด Source Inventory จึงเป็นจุดเชื่อมระหว่าง Competency Questions กับข้อจำกัดของข้อมูลต้นทาง

งานสร้าง Knowledge Graph มักเริ่มจากข้อมูลที่ไม่ได้เรียบร้อยเท่าที่ออกแบบไว้ใน Ontology ข้อมูลอาจอยู่ใน Spreadsheet, Database, PDF, Log File, API หรือเอกสารรายงาน การทำ Source Inventory จึงเป็นขั้นตอนแรกที่สำคัญมาก เพราะช่วยให้ทีมรู้ว่าข้อมูลใดมีอยู่จริง ข้อมูลใดต้องสกัด และข้อมูลใดเป็นเพียงความต้องการที่ยังไม่มีแหล่งรองรับ Source Inventory ที่ดีควรบันทึกอย่างน้อยสี่อย่าง คือชื่อแหล่งข้อมูล เจ้าของข้อมูล โครงสร้างของข้อมูล และความสัมพันธ์กับ Competency Questions ตัวอย่างเช่น ถ้าคำถามคือ “โครงการใดมี Outcome ใดและมีหลักฐานจากรายงานใด” Source Inventory ต้องชี้ได้ว่า Outcome อยู่ในระบบใด รายงานอยู่ที่ใด และมี Identifier ใดที่เชื่อม Project กับ Report ได้ หากไม่มีจุดเชื่อมนี้ Ontology ที่ออกแบบไว้จะยังไม่สามารถทำงานจริงได้

ในทางปฏิบัติ Source Inventory ควรถูกเก็บเป็น Artifact ของโครงการ ไม่ใช่เป็นไฟล์ชั่วคราว เพราะเมื่อข้อมูลต้นทางเปลี่ยน Mapping และ Validation Rule ก็ต้องเปลี่ยนตาม หากไม่มี Inventory การตรวจหาข้อผิดพลาด Graph ที่ผิดจะยากมาก เพราะทีมจะไม่รู้ว่า Triple หนึ่งเกิดจาก Source ใดและผ่าน Rule ใดมาก่อน Source Inventory ยังควรบอกระดับความน่าเชื่อถือของแต่ละแหล่งข้อมูลด้วย เช่นข้อมูลจากระบบทะเบียนหลักอาจเป็น Authority สำหรับรหัสโครงการ ขณะที่รายงาน PDF อาจเป็นหลักฐานของ Outcome แต่ไม่ใช่แหล่งกำหนดชื่อองค์กรหลัก หากไม่ระบุบทบาทเช่นนี้ Mapping Layer อาจใช้ข้อมูลจากเอกสารรองไปสร้าง IRI ใหม่ ทั้งที่ควร Map กลับไปยัง Authority List เดิม การรู้ว่า Source ใดเป็น Authoritative สำหรับ Entity ใดช่วยลด Resource ซ้ำ และทำให้ Provenance ของ Claim ชัดขึ้น

ในงานที่มีหลายทีม Source Inventory ควรอ่านได้โดยทั้งนักพัฒนาและผู้เชี่ยวชาญโดเมน ไม่ควรมีแต่ชื่อไฟล์หรือชื่อ Table แต่ควรมีคำอธิบายภาษาคนว่า Source นี้ใช้ตอบคำถามใด มีข้อจำกัดอะไร และข้อมูลใดไม่ควรถูกตีความเกินจริง เช่น Field “Status” ในระบบหนึ่งอาจหมายถึงสถานะเอกสาร ไม่ใช่สถานะโครงการ หากไม่บันทึกไว้ตั้งแต่ต้น Graph ที่ได้อาจสื่อความหมายผิด แม้ Mapping จะถูกต้องตาม Syntax

8.5 Extraction Layer และการควบคุมความหมาย

Source Inventory ทำให้ทราบตำแหน่งของข้อมูล แต่เอกสารและข้อความกึ่งโครงสร้างยังไม่สามารถกลายเป็น RDF ได้โดยตรง Extraction Layer จึงทำหน้าที่เสนอ Candidate พร้อมบริบทจากต้นฉบับ โดยยังไม่ประกาศ Candidate นั้นเป็นข้อเท็จจริงของ Published Graph

เมื่อข้อมูลต้นทางเป็นเอกสารหรือข้อความกึ่งโครงสร้าง ต้องมี Extraction Layer เพื่อดึง Entity และ Relation ออกมา ขั้นตอนนี้อาจใช้ Rule, Regular Expression, แบบจำลองประมวลผลภาษาหรือเครื่องมือสกัดข้อมูลชนิดอื่น แต่ไม่ควรปล่อยให้ Extraction Layer สร้าง Ontology หรือ IRI อย่างอิสระทั้งหมด เพราะจะทำให้ Graph ขาดเสถียรภาพ แนวทางที่เหมาะสมคือให้ Extraction Layer ส่งผลลัพธ์ในรูป Candidate เช่น Candidate Project, Candidate Organization, Candidate Technology และ Candidate Evidence จากนั้น Mapping Layer จึงตัดสินว่าควร Map Candidate เหล่านี้ไปยัง IRI ใด หากไม่พบ IRI เดิม ระบบอาจสร้าง IRI ใหม่ตาม Policy ที่กำหนดไว้ พร้อม Provenance ว่ามาจาก Extraction ครั้งใด

การแยก Extraction ออกจาก Mapping ทำให้ระบบตรวจสอบได้มากขึ้น หากเครื่องมือสกัดชื่อองค์กรผิด สามารถแก้ที่ Extraction Candidate ได้โดยไม่ทำลาย Ontology หาก Mapping ผิด สามารถแก้ Mapping Rule ได้โดยไม่ต้องเปลี่ยนเอกสารต้นทาง วิธีนี้ทำให้ Knowledge Graph Application มีโครงสร้างที่ดูแลระยะยาวได้ ผลลัพธ์จาก Extraction จึงควรมีสถานะเป็น Candidate ที่ยังต้องผ่านการตรวจ ไม่ใช่ความจริงของ Graph ทันที ตัวอย่าง Candidate อาจมีชื่อ Entity ข้อความต้นทาง ตำแหน่งในเอกสาร และ Confidence ของการสกัด แต่ยังไม่ควรประกาศเป็น RDF ใน Published Graph จนกว่าจะผ่าน Mapping และ Validation ขั้นต่อไป เพราะเครื่องมืออัตโนมัติทุกชนิดอาจสกัดความสัมพันธ์ผิดหรือเติมความหมายเกินข้อความที่ปรากฏจริงได้

```
candidate:
text: "PSU พัฒนาโครงการระบบเฝ้าระวังอัจฉริยะ"
entity: "PSU"
candidate_type: Organization
source: report-q1.pdf
span: page 3, line 12
status: pending-mapping
```

ตัวอย่างนี้ยังไม่ใช่ RDF Graph ที่พร้อมใช้ แต่เป็นวัตถุดิบสำหรับ Mapping Layer ขั้นถัดไป การเก็บ Span หรือหน้าของเอกสารช่วยให้ผู้ทบทวน ย้อนกลับไปตรวจได้ว่า Candidate นั้นมาจากข้อความใด หากภายหลังพบว่าการสกัดผิด ทีมสามารถแก้ Candidate หรือปรับ Prompt/Rule ได้โดยไม่ต้องเปลี่ยน Ontology หลัก

8.6 Semantic Mapping Layer

ผลจาก Extraction ยังเป็นเพียงข้อความหรือ Candidate ที่ไม่มี IRI และความหมายตาม Ontology Semantic Mapping จึงเป็นขั้นที่ตัดสินว่า Candidate ใดตรงกับ Resource เดิม ควรสร้าง Resource ใหม่หรือไม่ และใช้ Class กับ Property ใดในการสร้าง Triple

Semantic Mapping เป็นขั้นที่เปลี่ยนข้อมูลหรือ Candidate ให้กลายเป็น RDF ตาม Vocabulary ที่กำหนดไว้ สมมติว่ามี Record หนึ่งประกอบด้วย Project Code, Organization Name, Technology

Name และ Start Date Mapping Layer ต้องกำหนดว่า Project Code สร้าง IRI อย่างไร Organization Name ถูก Resolve เป็น Organization ใด Technology Name ถูก Map ไปยัง Technology Resource ใด และ Start Date เป็น Literal Datatype ใด ตัวอย่างต่อไปนี้เป็น Pseudocode ของ Mapping Rule ไม่ใช่ไวยากรณ์ Turtle

```
row.project_code -> IRI(project_namespace + row.project_code)
row.org_code -> IRI(org_namespace + row.org_code)
row.tech_code -> IRI(tech_namespace + row.tech_code)
row.start_date -> "YYYY-MM-DD"^^xsd:date
```

แต่ในระบบจริง Mapping มักซับซ้อนกว่านี้ เพราะข้อมูลต้นทางอาจไม่มี Code ที่สะอาด ชื่อองค์กรอาจสะกดหลายแบบ หรือเทคโนโลยีหนึ่งอาจอยู่ในหลาย Taxonomy จึงต้องมี Entity Resolution และ Authority List เข้ามาช่วย หากไม่มีขั้นนี้ Graph จะเต็มไปด้วย Resource ซ้ำและ Query จะคืนผลลัพธ์ไม่ครบ Mapping Specification ที่ดีควรมี Rationale ของแต่ละ Rule ด้วย เช่นทำไม **row.org_code** จึงสร้าง IRI ของ Organization ได้โดยตรง แต่ **row.org_name** ต้องผ่าน Authority List ก่อน หรือทำไม **row.technology_name** บางกรณีสร้าง Technology Resource ใหม่ไม่ได้จนกว่าจะมี Reviewer ยืนยัน เหตุผลเหล่านี้ช่วยให้ทีมเข้าใจว่า Mapping ไม่ใช่การแปลง Column เป็น Triple แบบกลไกเท่านั้น แต่เป็นการตัดสินความหมายของข้อมูลต้นทาง

หาก Mapping ไม่มี Rationale เมื่อเวลาผ่านไปผู้ดูแลระบบอาจไม่กล้าแก้ Rule แม้รู้ว่าผลลัพธ์ไม่เหมาะสม เพราะไม่ทราบว่ากฎเดิมเกิดจากข้อจำกัดใด การบันทึก Rationale จึงเป็นส่วนหนึ่งของ Semantic Governance ตั้งแต่ชั้น Application ไม่ใช่รอไปทำเฉพาะตอน Release Ontology

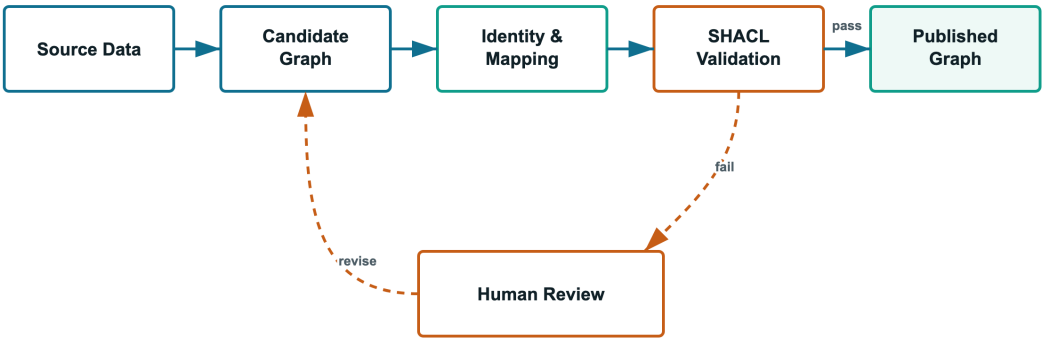
8.7 Validation Layer ในฐานะ Gate

Mapping ทำให้ข้อมูลมีรูปเป็น RDF แต่ไม่ได้รับรองว่าข้อมูลครบตามเงื่อนไขของ Application Validation Layer จึงทำหน้าที่เป็น Gate ระหว่าง Candidate Graph กับ Published Graph และต้องคืนรายงานที่ชี้กลับไปยัง Source หรือ Mapping Rule ที่เป็นต้นเหตุได้

หลัง Mapping แล้ว Graph ไม่ควรถูก Publish ทันที แต่ควรผ่าน Validation Layer ก่อน SHACL สามารถทำหน้าที่เป็น Gate ที่ตรวจว่าข้อมูลสำคัญครบตามข้อกำหนดหรือไม่ เช่น Project ทุกตัวต้องมี Label, Owner และ Status ส่วน Technology ต้องมี Type และ Label อย่างน้อยหนึ่งภาษา หากไม่ผ่านระบบควรเก็บ Validation Report ไว้ให้ผู้ดูแลแก้ไข Validation Report ควรถูกมองเป็นข้อมูลชนิดหนึ่ง ไม่ใช่เพียงข้อความ Error ชั่วคราว เพราะรายงานเหล่านี้บอกคุณภาพของ Pipeline ได้ เช่น Source ใดทำให้ Violation บ่อย Mapping Rule ใดสร้างข้อมูลผิด และ Field ใดมักขาด การเก็บ Report อย่างเป็นระบบช่วยให้ทีมปรับปรุง Pipeline จากหลักฐานจริง ไม่ใช่จากความรู้สึก

ในระบบ Production Validation Layer ควรมีผลต่อการตัดสินใจของ Pipeline อย่างชัดเจน เช่น Violation ระดับ Error ทำให้ Graph ไม่ถูก Publish ส่วน Warning อาจให้ Publish ได้แต่ต้องมี Issue สำหรับ Data Steward และ Info ใช้เป็นข้อเสนอแนะในการปรับ Documentation การแยกระดับเช่นนี้ทำให้ Warning ที่ไม่กระทบคำตอบไม่หยุด Pipeline ทั้งหมด ขณะเดียวกันก็ป้องกันข้อมูลสำคัญที่ผิดไม่ให้เข้าสู่ Graph ที่ผู้ใช้เชื่อถือ นอกจากนี้ Validation Report ควรถูกเชื่อมกับ Source และ Mapping Run เสมอ หาก Report บอกเพียงว่า **proj:P001** ไม่มี **ex:responsibleOrg** แต่ไม่บอกว่า Triple นี้มาจากไฟล์ใดหรือ

Mapping Rule ใด ผู้ดูแลข้อมูลจะต้องเสียเวลาค้นย้อนกลับเอง การออกแบบ Validation Layer ที่ดีจึงต้องคิดต่อจาก SHACL Syntax ไปถึง Workflow การแก้ข้อมูลจริง เส้นทางจากข้อมูลต้นทางไปสู่ Graph ที่เผยแพร่ รวมทั้งวงจรส่งกลับเมื่อข้อมูลไม่ผ่าน แสดงในภาพที่ 8.2



ภาพที่ 8.2: เส้นทางจาก Extraction Candidate ไปสู่ Knowledge Graph ที่เผยแพร่

ภาพที่ 8.2 แสดงว่า Candidate จากเครื่องมือสกัดยังไม่ใช่ข้อเท็จจริงที่ระบบเผยแพร่ ขั้น Identity Resolution และ Mapping ต้องกำหนด IRI, Class และ Property ก่อน จากนั้น SHACL Gate จึงตรวจ Graph หากไม่ผ่าน ข้อมูลต้องย้อนกลับไปยังการทบทวน หรือแก้ Rule ที่เป็นต้นเหตุ ไม่ควรแก้เฉพาะค่าปลายทางใน Published Graph แล้วปล่อยให้การนำเข้ารอบต่อไปสร้างความผิดเดิม

8.8 API, UI และผู้ใช้ปลายทาง

เมื่อ Graph ผ่าน Validation แล้ว ข้อมูลยังต้องถูกนำเสนอในรูปที่ผู้ใช้เข้าใจได้ API และ UI จึงไม่ควรคืนเพียงข้อความสรุป แต่ควรรักษา Identifier, Evidence และบริบทที่จำเป็นต่อการตรวจสอบคำตอบไว้ด้วย

ผู้ใช้ปลายทางส่วนใหญ่ไม่ได้ต้องการเห็น RDF หรือ SPARQL โดยตรง พวกเขาต้องการคำตอบ รายงาน Dashboard หรือคำอธิบาย ดังนั้นแอปพลิเคชัน Knowledge Graph ต้องมีชั้น API หรือ UI ที่แปล Graph ให้เป็นรูปแบบที่ผู้ใช้ทำงานได้ เช่น หน้าแสดงความสัมพันธ์ของโครงการ หน้าค้นหา Evidence หรือ Dashboard ที่นับจำนวนโครงการตาม Technology อย่างไรก็ตาม UI ไม่ควรซ่อน Provenance จนหมด หากผู้ใช้เห็น Outcome ของโครงการ ควรคลิกกลับไปดูได้ว่าข้อมูลมาจากรายงานใด Graph ใด และผ่าน Validation เมื่อไร จุดนี้ทำให้ Knowledge Graph แตกต่างจาก Dashboard ทั่วไป เพราะไม่เพียงแสดงตัวเลขหรือข้อความ แต่แสดงเส้นทางของความหมายและหลักฐานด้วย

API ที่ดีควรส่งคืนทั้งค่าที่ผู้ใช้ต้องการและ Identifier ที่ทำให้ตรวจต่อได้ เช่น Endpoint ที่คืนข้อมูลโครงการควรมี Project IRI, Label, Organization IRI, Technology IRI, Claim IRI และ Report IRI ตามความจำเป็น ไม่ควรส่งคืนเฉพาะข้อความที่ถูก Render แล้ว เพราะ Application ชั้นบนจะสูญเสียความสามารถในการเชื่อมต่อและตรวจสอบ ส่วนติดต่อใดที่นำข้อมูลไปสรุปเป็นภาษาธรรมชาติก็ควรถูกผูกกลับ

มายัง Evidence Bundle ชุดเดียวกัน ตัวอย่างโครงสร้าง Response อาจไม่ต้องซับซ้อน แต่ควรมีแนวคิดเรื่อง Evidence เช่น

```
project: proj:P001
label: "ระบบเฝ้าระวังอัจฉริยะ"
technology: tech:SensorNet
evidence:
 claim: claim:C001
 report: report:R2026-01
 graph: graph:projects-current
```

รูปแบบนี้ทำให้ UI แสดงคำตอบอ่านง่ายได้ ขณะเดียวกันนักวิจัยหรือผู้ตรวจสอบยังสามารถตามกลับไปยัง Graph และ Source ได้ การออกแบบ API จึงเป็นส่วนหนึ่งของ Knowledge Graph Engineering ไม่ใช่เรื่องแยกจาก Ontology

8.9 Deployment และ Versioning

ส่วนติดต่อผู้ใช้จะให้ผลลัพธ์สม่ำเสมอได้เมื่อ Ontology, Mapping, Shape และ Query Catalog ใช้ Version ที่เข้ากัน หัวข้อนี้จึงเชื่อม Deployment เข้ากับ Versioning และ Migration แทนการมองการนำระบบขึ้นใช้เป็นเพียงการเปิด Endpoint

Knowledge Graph Application ที่ใช้จริงต้องมี Versioning ของ Ontology, Mapping, Shapes และ Query Catalog หาก Ontology เปลี่ยนแต่ Mapping ยังใช้ Version เดิม ระบบอาจสร้าง Triple ที่ไม่สอดคล้องกับ Vocabulary ใหม่ หาก Shape เปลี่ยนแต่ Validation Report เก่ายังถูกนำมาใช้ อาจทำให้คุณภาพข้อมูลถูกตีความผิด แนวทางหนึ่งคือกำหนด Release Bundle ของระบบ เช่น Release หนึ่งประกอบด้วย Ontology Version, SHACL Version, Mapping Version, Query Catalog Version และตัวอย่าง RDF Graph ที่ผ่าน Validation แล้ว เมื่อ Release ใหม่ออกมา ทีมควรรู้ว่าอะไรเปลี่ยนและผลกระทบต่อ Data Pipeline คืออะไร วิธีนี้ทำให้ Knowledge Graph Application ถูกจัดการเหมือน Software System ไม่ใช่เอกสารความรู้ที่แก้ไขแบบไม่มีประวัติ

Release Bundle ควรมี Migration Note ด้วย โดยเฉพาะเมื่อเปลี่ยนชื่อ Property เปลี่ยน Class Hierarchy หรือเปลี่ยน Shape ที่เข้มขึ้น เช่นถ้าเปลี่ยน **ex:owner** เป็น **ex:responsibleOrg** ต้องบอกว่าข้อมูลเก่าจะถูกแปลงอย่างไร Query ใดต้องปรับ และ API ใดได้รับผลกระทบ หากไม่มี Migration Note ระบบอาจมีข้อมูลสองรูปแบบอยู่พร้อมกัน ทำให้ Query บางตัวเห็นข้อมูลเก่าแต่ไม่เห็นข้อมูลใหม่ หรือกลับกัน ในระบบที่ทำงานกับข้อมูลจริง การ Rollback ก็สำคัญไม่แพ้ Release หาก Graph Version ใหม่ทำให้ Query สำคัญให้ผลผิด ทีมควรรู้ว่าจะกลับไปใช้ Published Graph Version ก่อนหน้าอย่างไร และ Validation Report ของ Version นั้นอยู่ที่ไหน การคิดเรื่อง Rollback ตั้งแต่ต้นทำให้การนำ Knowledge Graph ไปใช้ในองค์กรน่าเชื่อถือขึ้น เพราะทีมไม่ได้พึ่งการแก้ไขด้วยมือเมื่อเกิดปัญหา แต่มีเส้นทางกลับที่ตรวจได้

8.10 กรณีศึกษาแบบต่อเนื่อง: จาก Prototype ไปสู่ระบบจริง

แนวคิดแต่ละ Layer จะเห็นความสัมพันธ์ชัดขึ้นเมื่อพิจารณาการเปลี่ยนจาก Prototype ไปสู่ระบบจริง หัวข้อนี้มองในระดับ *โครงสร้างของระบบและความรับผิดชอบของทีม* ว่าต้องเพิ่มสิ่งใดเมื่อข้อมูลเริ่มเปลี่ยน มีผู้ใช้หลายกลุ่ม และต้องตรวจย้อนกลับได้ ยังไม่ได้ติดตาม Record ใด Record หนึ่ง ซึ่งจะทำแยกต่างหากใน Walkthrough หลังจากกำหนดสถาปัตยกรรมขั้นต่ำแล้ว

สมมติว่าทีมต้องการสร้างระบบ Knowledge Graph สำหรับติดตามโครงการดิจิทัลในองค์กร จุดเริ่มต้นอาจเป็น Proof of Concept ที่ใช้ไฟล์ Spreadsheet หนึ่งชุด Ontology หนึ่งไฟล์ และ SPARQL Query ที่ครอบคลุมคำถามหลัก วิธีนี้ช่วยตรวจว่าการแทน Project, Technology, Organization, Report และ Outcome เป็น RDF ทำให้ตอบคำถามได้จริงหรือไม่ แต่ระบบที่นำไปใช้งานต้องมีโครงสร้างมากกว่านั้น เพราะข้อมูลจะเปลี่ยน มีผู้ใช้หลายกลุ่ม และต้องตรวจย้อนกลับได้

การยกระดับ Prototype ไปสู่ระบบจริงควรพิจารณาเป็นลำดับดังนี้

1. **แยก Raw Data ออกจาก Published Graph** Raw Data คือข้อมูลต้นทางที่ยังไม่ได้ผ่านการตีความ เช่น CSV, PDF, API Response หรือเอกสารรายงาน ส่วน Published Graph คือ RDF Graph ที่ผ่าน Mapping และ Validation แล้ว การแยกสองส่วนนี้ช่วยให้ตรวจได้ว่า Triple ที่ผิดเกิดจาก Source, Extraction หรือ Mapping
2. **สร้าง Staging Graph ก่อน Publish** เมื่อข้อมูลชุดใหม่เข้ามา Mapping Layer ควรเขียน RDF ไปยัง Staging Graph แล้วจึงตรวจด้วย Validation Layer หากผ่านจึงย้ายหรือ Merge เข้าสู่ Published Graph หากไม่ผ่าน ระบบควรเก็บ Validation Report และไม่ปล่อยข้อมูลไปใช้กับ Dashboard หรือ Application โดยตรง รูปแบบนี้ทำให้ Knowledge Graph มีการตรวจอัตโนมัติก่อน Release ตัวอย่าง Pipeline อย่างย่อเขียนได้ดังนี้

```
raw/project.csv
 -> extraction candidates
 -> mapping rules
 -> staging RDF graph
 -> SHACL validation report
 -> published named graph
 -> SPARQL API / dashboard / user interface
```

   ในแต่ละขั้นควรมี Log และ Identifier ของตนเอง เช่น Mapping Run ครั้งที่เท่าไร ใช้ Ontology Version ใด ใช้ Shape Version ใด และ Source File ใด ข้อมูลเหล่านี้อาจดูเหมือนรายละเอียดเชิงระบบ แต่มีผลต่อความน่าเชื่อถือของคำตอบอย่างมาก หากผู้ใช้ถามว่า Outcome หนึ่งมาจากไหน ระบบควรตอบได้ไม่ใช่เพียงว่ามาจาก Graph แต่ควรบอกได้ว่า Graph นั้นสร้างจาก Source ใด ผ่าน Mapping Version ใด และผ่าน Validation เมื่อไร
3. **ทำ Entity Resolution** ประเด็นนี้สำคัญเป็นพิเศษในบริบทภาษาไทย เพราะชื่อองค์กรหรือโครงการอาจสะกดได้หลายแบบ เช่นชื่อเต็มภาษาไทย ชื่อย่อภาษาอังกฤษ ชื่อในเอกสารราชการ หรือชื่อที่พิมพ์คลาดเคลื่อน หากสร้าง IRI ใหม่ทุกครั้งที่พบชื่อใหม่ Graph จะมี Resource ซ้ำจำนวนมาก แนวทางที่เหมาะสมคือจัดทำ Authority List หรือ Registry ของ Entity สำคัญ แล้วให้ Mapping Layer ใช้รายการดังกล่าวตัดสินว่าควร Map ไปยัง IRI เดิมหรือสร้าง IRI ใหม่

ตัวอย่างเช่น "PSU", "Prince of Songkla University" และ "มหาวิทยาลัยสงขลานครินทร์" ควร Map ไปยัง Resource เดียวกัน หากหมายถึงองค์กรเดียวกัน การเก็บ Label หลายภาษาและ Synonym ไว้ใน Graph ช่วยให้ผู้ใช้ค้นด้วยคำต่างกันและยังเชื่อมกลับไปยัง IRI เดียวกันได้

4. **ออกแบบ API ตามกลุ่มผู้ใช้** ระบบไม่จำเป็นต้องเปิด SPARQL โดยตรงให้ผู้ใช้ทุกคน แม้ SPARQL Endpoint จะเหมาะกับนักพัฒนาและนักวิจัย แต่ผู้ใช้ทั่วไปอาจต้องการ REST API หรือหน้า UI ที่แสดงข้อมูลพร้อมหลักฐาน เช่น `/projects/P001`, `/projects?technology=SensorNet` หรือหน้าแสดง Evidence ของ Outcome หนึ่ง API อาจใช้ SPARQL อยู่ด้านหลังโดยซ่อนความซับซ้อนของ Query จากผู้ใช้ปลายทาง แต่ไม่ควรซ่อนความหมายจนตรวจสอบไม่ได้ หาก Endpoint ส่งคืน Outcome ของโครงการ ก็ควรส่งคืน IRI ของ Outcome, IRI ของ Claim, รายงานที่เป็น Provenance และสถานะ Validation ด้วย
5. **ติดตามคุณภาพของ Graph** ระบบควรวัดอย่างน้อยว่ามี Resource ที่ไม่มี Label กี่ตัว มี Violation กี่รายการต่อ Source มี IRI ซ้ำที่ต้อง Merge กี่กรณี Query สำคัญยังคืนผลลัพธ์ตามที่คาดหรือไม่ และเวลาตอบ Query อยู่ในระดับที่ผู้ใช้ยอมรับได้หรือไม่ Metric เหล่านี้ทำให้ทีมประเมินคุณภาพจากหลักฐาน แทนการสรุปจากความรู้สึกว่า Graph น่าจะดีแล้ว

ในบริบทงานวิจัย การสร้างระบบเช่นนี้ยังช่วยให้ Contribution ชัดขึ้น เพราะผู้อ่านไม่ได้เห็นเพียง Ontology Diagram แต่เห็นวิธีนำ Ontology ไปใช้กับข้อมูลจริง เห็น Validation เห็น Query และเห็น Provenance หากงานวิจัยเสนอ Ontology ใหม่แต่ไม่มีตัวอย่าง RDF, ไม่มี Query และไม่มีวิธีตรวจข้อมูล Contribution จะประเมินยากกว่า เพราะไม่รู้ว่า Ontology นั้นใช้ตอบคำถามจริงได้อย่างไร ดังนั้นการสร้างแอปพลิเคชัน Knowledge Graph จึงควรถูกมองเป็นงานวิศวกรรมเต็มรูปแบบ ไม่ใช่การแปลงข้อมูลเป็น RDF เพียงครั้งเดียว ระบบที่ดีต้องมี Lifecycle ของข้อมูล มีการตรวจสอบ มี Versioning มีช่องทางให้ผู้ใช้เข้าถึง และมีวิธีอธิบายคำตอบย้อนกลับไปยังหลักฐานต้นทางได้

8.11 สถาปัตยกรรมขั้นต่ำที่ควรมีในระบบหนึ่ง

กรณีศึกษาก่อนหน้าอาจทำให้ดูเหมือนว่าระบบต้องมีองค์ประกอบจำนวนมากตั้งแต่เริ่มต้น หัวข้อนี้จึงลดโครงสร้างลงเหลือสถาปัตยกรรมขั้นต่ำที่ยังรักษา Artifact, Validation, Query Test และ Provenance ไว้ได้

จากประสบการณ์ออกแบบระบบ แนวทางที่เหมาะสมคือเริ่มจากสถาปัตยกรรมขั้นต่ำที่ไม่ซับซ้อนเกินไป แต่มีองค์ประกอบเพียงพอสำหรับการดูแลระยะยาว ได้แก่

- **Ontology Repository** เก็บไฟล์ Ontology, Prefix, Label, Comment และ Changelog ของ Vocabulary
- **Mapping Repository** เก็บ Rule ที่แปลงข้อมูลต้นทางเป็น RDF
- **Validation Repository** เก็บ SHACL Shape และตัวอย่าง Validation Report
- **Query Catalog** เก็บ Query ที่ผ่านการตรวจ พร้อมวัตถุประสงค์และเงื่อนไขการใช้
- **Graph Store** แยก Staging, Published และ Provenance Graph ออกจากกัน

การแยก Repository เช่นนี้ไม่ได้แปลว่าระบบต้องมีหลายระบบตั้งแต่วันแรก แต่ช่วยให้ทีมรู้ว่า Artifact แต่ละชนิดอยู่ตรงไหน หาก Ontology เปลี่ยน ทีมจะรู้ว่าต้องตรวจ Mapping และ Query Catalog หาก Shape เปลี่ยน ทีมจะรู้ว่าต้องตรวจ Validation Report หาก Query เปลี่ยน ทีมจะรู้ว่าต้องตรวจผลลัพธ์กับ Graph ตัวอย่าง การจัดการเช่นนี้ป้องกันไม่ให้ Knowledge Graph กลายเป็นไฟล์เดียวที่รวม Artifact ต่างชนิดโดยไม่มีโครงสร้าง ในระบบที่มีผู้ใช้และข้อมูลไม่มาก Artifact เหล่านี้อาจอยู่ใน Git Repository เดียวกันก็ได้ เช่นมีโฟลเดอร์ **ontology/**, **mappings/**, **shapes/**, **queries/**, **examples/** และ **docs/** สิ่งสำคัญไม่ใช่เครื่องมือ แต่คือการทำให้การเปลี่ยนแปลงถูกติดตามได้ หาก Property หนึ่งถูกเปลี่ยนชื่อ ควรเห็นได้ว่า Query ใดต้องแก้ หาก Shape หนึ่งเข้มขึ้น ควรเห็นได้ว่าข้อมูลตัวอย่างชุดใดไม่ผ่าน

ตัวอย่างโครงสร้าง Repository อย่างง่ายคือ

```
knowledge-graph/
|-- ontology/
|   |-- core.ttl
|   `-- project.ttl
|-- mappings/
|   |-- project-csv.mapping.ttl
|   `-- report-extraction.mapping.ttl
|-- shapes/
|   |-- project-shapes.ttl
|   `-- outcome-claim-shapes.ttl
|-- queries/
|   |-- project-by-technology.rq
|   `-- outcome-with-evidence.rq
|-- examples/
|   |-- sample-projects.ttl
|   `-- expected-results/
`-- docs/
    |-- competency-questions.md
    `-- release-notes.md
```

โครงสร้างนี้ช่วยให้ผู้อ่านใหม่เข้าใจระบบได้เร็วขึ้น เพราะเห็นทันทีว่า Ontology อยู่ที่ใด Mapping อยู่ที่ใด และ Query ใดใช้ทดสอบความสามารถของระบบ เมื่อมีผู้ร่วมงานใหม่เข้ามา เขาไม่ต้องเริ่มจากการเปิด Triple Store แล้วเดาว่าข้อมูลมาจากไหน แต่เริ่มจาก Artifact ที่อธิบายระบบได้ ในส่วนของการ Deploy ระบบควรมีอย่างน้อยสอง Environment คือ Staging และ Production โดย Staging ใช้รับข้อมูลใหม่และทดสอบ Mapping/Validation ส่วน Production ใช้ให้ Application หรือผู้ใช้ปลายทางอ่านข้อมูล หากทีมมีทรัพยากรมากขึ้น อาจเพิ่ม Development Environment สำหรับทดลอง Ontology Version ใหม่ การมีหลาย Environment ช่วยลดความเสี่ยงที่ข้อมูลทดลองจะปนกับข้อมูลที่ผู้ใช้เชื่อถือแล้ว

อีกเรื่องที่ควรทำตั้งแต่ต้นคือ Regression Test ของ Query สำคัญ เช่น Query ที่ค้นโครงการตามเทคโนโลยีควรคืนจำนวนผลลัพธ์ขั้นต่ำ Query ที่ดึง Outcome พร้อม Evidence ควรมี Report ทุกแถว และ Query ที่ตรวจข้อมูลขาดควรไม่เพิ่มขึ้นผิดปกติหลัง Release ใหม่ Test เหล่านี้ไม่จำเป็นต้องซับซ้อน แต่ช่วยให้ทีมตรวจพบว่า การเปลี่ยน Ontology หรือ Mapping มีผลกระทบต่อผลการสืบค้นที่คาดหวังหรือไม่ สำหรับระบบที่มีส่วนสรุปหรือรายงานอัตโนมัติ ควรมีชั้น Policy เพิ่มเติม เช่นห้ามแสดง Claim ที่ไม่มี Provenance ห้ามใช้ Graph ที่ Validation Error ยังไม่ถูกแก้ และต้อง Log Query ที่ใช้สร้างผลลัพธ์ทุกครั้ง Policy เหล่านี้ช่วยให้ข้อความที่ระบบแสดงไม่กล่าวเกินหลักฐานใน Knowledge Graph และทำให้ Audit ได้ภายหลังว่าผลลัพธ์หนึ่งเกิดจากข้อมูลใด

ดังนั้นสถาปัตยกรรมขั้นต่ำของ Knowledge Graph ไม่ได้เริ่มจากเครื่องมือราคาแพงหรือระบบขนาดใหญ่ แต่เริ่มจากการจัด Artifact ให้เป็นระบบ มี Graph ที่แยกบทบาท มี Validation ก่อน Publish มี Query ที่ทดสอบได้ และมี Provenance ที่ตรวจย้อนกลับได้ หัวข้อนี้ตอบว่า “ระบบต้องมีอะไร” ส่วนหัวข้อถัดไปจะตอบว่า “เมื่อ Record หนึ่งเข้ามา Artifact เหล่านี้ถูกใช้ตามลำดับอย่างไร”

8.12 Walkthrough: หนึ่งข้อมูลไหลผ่านทั้งระบบ

สถาปัตยกรรมขั้นต่ำบอกว่าระบบควรมีส่วนใดบ้าง แต่ยังไม่แสดงว่า Record หนึ่งเปลี่ยนทั้ง *รูปแทน* และ *สถานะความน่าเชื่อถือ* อย่างไร คำว่า “ไหลผ่านระบบ” ในที่นี้จึงไม่ได้หมายถึงการส่งไฟล์เดิมต่อกันไปเรื่อย ๆ แต่หมายถึงการสร้าง Artifact ใหม่ในแต่ละจุด พร้อมการตัดสินใจว่าข้อมูลพร้อมเข้าสู่ขั้นถัดไปหรือไม่

Walkthrough นี้ใช้ Record เดียวตลอดทาง และตั้งเป้าตอบคำถามว่า

> “โครงการใดใช้ SensorNet และรายงานใดเป็นหลักฐานของข้อมูลนั้น”

ข้อมูลต้นทางเป็นหนึ่งแถวจาก Spreadsheet ทะเบียนโครงการ ดังตารางที่ 8.2

ตารางที่ 8.2: Source Record ที่ใช้ตลอด Walkthrough

Field	ค่าที่พบใน Source	ความหมายที่คาดว่าจะใช้
project_code	**P001**	รหัสสำหรับสร้าง Identity ของ Project
project_name	ระบบเฝ้าระวังอัจฉริยะ	Label ภาษาไทยของ Project
org_name	**PSU**	ชื่อดิบที่ต้อง Resolve กับทะเบียน Organization
technology_name	**SensorNet**	ชื่อดิบที่ต้อง Resolve กับ Technology Vocabulary
report_id	**R2026-01**	Identifier ของ Report ที่ Source ระบุเป็นหลักฐาน
status	ว่าง	ข้อมูลที่ Shape ของระบบกำหนดให้ต้องมี

จุดสำคัญคือแถวนี้ยังไม่ใช่ข้อเท็จจริงใน Knowledge Graph แม้ค่าบางช่องจะดูถูกต้องก็ตาม แต่ละ Checkpoint ให้สถานะต่างกันดังตารางที่ 8.3

ตารางที่ 8.3: สถานะของข้อมูลในแต่ละ Checkpoint

Checkpoint	Artifact ที่มีอยู่	ใช้ตอบผู้ใช้ได้หรือไม่
Source	แถวข้อมูลดิบพร้อม Source Identifier	ยังไม่ได้ เพราะยังไม่ผ่านการตีความ
Candidate	ค่าที่ Parse แล้วพร้อมตำแหน่งต้นทาง	ยังไม่ได้ เพราะ Identity ยังไม่ถูก Resolve
Staging Graph	RDF ที่ผ่าน Mapping	ยังไม่ได้ เพราะยังไม่ผ่าน Validation
Validated Graph	RDF และ Validation Report ที่ Conform	ยังไม่ได้ จนกว่าจะได้รับการ Publish เป็น Version ที่อนุมัติ
Published Graph	Named Graph ที่ระบุ Version และ Provenance	ใช้เป็นฐานของ Query ได้
Evidence Bundle	Query Result พร้อม Source และ Validation Run	ใช้แสดงคำตอบที่ตรวจย้อนกลับได้

เส้นทางของ Record **P001** มีรายละเอียดดังนี้

1. **ระบุ Source และขอบเขตความน่าเชื่อถือ**

   Source Inventory กำหนดว่า Spreadsheet ชุดนี้เป็น Authority สำหรับ **project_code, project_name** และ **status** แต่ชื่อ Organization กับ Technology ต้องตรวจเทียบ Registry ภายนอก Record นี้ได้รับ Source Identifier เช่น **source:projects-q1-row-P001** เพื่อให้ Artifact ในขั้นหลังอ้างกลับมาที่แถวเดิมได้

2. **Parse เป็น Candidate โดยยังไม่สรุป Identity**

   Parsing Layer อ่านค่าจากแต่ละ Field และเก็บข้อความดิบไว้พร้อม Source Pointer ผลลัพธ์อย่างย่อคือ

   candidate_id: candidate:P001-2026Q1
   project_code: P001
   organization_text: PSU
   technology_text: SensorNet
   report_id: R2026-01
   status: null
   source_record: source:projects-q1-row-P001

   สถานะ **Candidate** หมายความว่า Parser อ่านข้อความได้แล้ว แต่ระบบยังไม่ได้ยืนยันว่า “PSU” หรือ “SensorNet” หมายถึง Resource ใด

3. **Resolve ข้อความดิบให้เป็น IRI**

   IRI Policy ใช้รหัสโครงการสร้าง **proj:P001** ส่วน Entity Resolution ตรวจ Registry แล้วได้ Mapping ดังนี้

   - **P001** → **proj:P001**

- PSU → org:PSU
- SensorNet → tech:SensorNet
- R2026-01 → report:R2026-01

หากชื่อ “PSU” ตรงกับ Organization มากกว่าหนึ่งรายการ ขั้นนี้ต้องหยุดเพื่อ Review ไม่ควรเลือก IRI จากความคล้ายของ Label แล้วปล่อยความกำกวมไปยัง Graph

4. **Map ลง Staging Graph**

Mapping Specification เลือก Class และ Property จาก Ontology แล้วสร้าง RDF เพื่อนำเข้า Named Graph สำหรับ Mapping Run นี้ ตัวอย่าง Turtle ต่อไปนี้แสดงเนื้อหาของกราฟ ส่วนบรรทัดที่ขึ้นต้นด้วยเครื่องหมาย # เป็นหมายเหตุบอกชื่อกราฟ การนำเข้าต้องระบุชื่อ Named Graph ให้ระบบจัดเก็บอีกครั้ง เพราะ Turtle เพียงอย่างเดียวไม่ได้กำหนดชื่อกราฟ

```
@prefix ex: <https://example.org/vocab/> .
@prefix proj: <https://example.org/resource/project/> .
@prefix org: <https://example.org/resource/organization/> .
@prefix tech: <https://example.org/resource/technology/> .
@prefix report: <https://example.org/resource/report/> .
@prefix claim: <https://example.org/resource/claim/> .
@prefix graph: <https://example.org/resource/graph/> .
@prefix run: <https://example.org/resource/run/> .
@prefix source: <https://example.org/resource/source/> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix prov: <http://www.w3.org/ns/prov#> .

# Named graph: graph:staging-run-042
proj:P001 a ex:Project ;
 rdfs:label "ระบบเฝ้าระวังอัจฉริยะ"@th ;
 ex:responsibleOrg org:PSU ;
 ex:usesTechnology tech:SensorNet ;
 ex:hasReport report:R2026-01 .

claim:C001 a ex:TechnologyUseClaim ;
 ex:aboutProject proj:P001 ;
 ex:assertsTechnology tech:SensorNet ;
 prov:wasDerivedFrom report:R2026-01 .

graph:staging-run-042
 prov:wasGeneratedBy run:mapping-042 ;
 prov:wasDerivedFrom source:projects-q1 .
```

Triple **ex:usesTechnology** ทำให้ Query อ่านความสัมพันธ์โดยตรง ส่วน **claim:C001** ทำให้ความสัมพันธ์นั้นมีจุดเชื่อมกลับไปยัง Report ที่ Source ระบุเป็นหลักฐาน อย่างไรก็ตาม Graph นี้ยังไม่มี **ex:status** เพราะ Source Field เดิมเป็นค่าว่าง จึงยังห้าม Query สำหรับผู้ใช้ปลายทางอ่าน Graph นี้

5. **ให้ SHACL Gate ตัดสินว่าข้อมูลพร้อมหรือไม่**

Shape ของระบบกำหนดว่า Project ที่จะ Publish ต้องมี Status อย่างน้อยหนึ่งค่า Validation Run จึงคืนผลโดยย่อในรูปแบบรายงานต่อไปนี้ ซึ่งเป็นข้อความสำหรับอ่านและไม่ใช่ Turtle

```
# Illustrative report (not Turtle)
focus_node: proj:P001
result_path: ex:status
severity: sh:Violation
message: "Project ที่เผยแพร่ต้องมี Status"
source_record: source:projects-q1-row-P001
```

ผลลัพธ์นี้ไม่ได้บอกว่า Ontology ผิด แต่บอกว่า Candidate Graph ยังไม่ผ่าน Contract ของ Application Pipeline จึงหยุดที่ Staging และไม่แก้ Graph ให้ผ่านด้วยการเติมค่าที่เดาเอง

6. **แก้ที่ต้นเหตุแล้วรัน Pipeline ใหม่**

   Data Owner ตรวจทะเบียนและยืนยันว่า Project มีสถานะ **Active** จึงแก้ Field **status** ใน Source จากนั้น Mapping Run ใหม่เพิ่ม Triple **proj:P001 ex:status status:Active** และ Validation Run ใหม่คืนค่า **sh:conforms true** วิธีนี้รักษาความสอดคล้องระหว่าง Source กับ Graph และป้องกันไม่ให้การนำเข้ารอบถัดไปสร้าง Violation เดิมซ้ำ

7. **Publish เป็น Graph Version ที่ระบุได้**

   เมื่อผ่าน Gate แล้ว ระบบจึงนำข้อมูลเข้าสู่ **graph:projects-2026-q1-v3** พร้อมบันทึก Ontology Version, Mapping Version, Shape Version และ Validation Run ที่ใช้ Query Catalog สำหรับผู้ใช้ถูกกำหนดให้อ่านเฉพาะ Published Graph ไม่อ่าน **graph:staging-run-042** หรือ Graph ชั่วคราวอื่น

8. **Query แล้วคืนคำตอบพร้อม Evidence**

   คำถามตั้งต้นสามารถแปลงเป็น Query ที่อ่าน Published Graph ได้ดังนี้

```
PREFIX ex: <https://example.org/vocab/>
PREFIX tech: <https://example.org/resource/technology/>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX prov: <http://www.w3.org/ns/prov#>

SELECT ?project ?label ?claim ?report
FROM <https://example.org/resource/graph/projects-2026-q1-v3>
WHERE {
 ?project ex:usesTechnology tech:SensorNet ;
        rdfs:label ?label .
 ?claim a ex:TechnologyUseClaim ;
       ex:aboutProject ?project ;
       ex:assertsTechnology tech:SensorNet ;
       prov:wasDerivedFrom ?report .
 FILTER(LANG(?label) = "th")
}
```

ผล Query ไม่ควรถูกส่งต่อเป็นข้อความชื่อโครงการเพียงค่าเดียว แต่ควรจัดเป็น Evidence Bundle ที่รักษาเส้นทางตรวจสอบไว้

```
answer: "โครงการ P001 ใช้ SensorNet"
project: proj:P001
technology: tech:SensorNet
claim: claim:C001
report: report:R2026-01
graph_version: graph:projects-2026-q1-v3
```

```
mapping_run: run:mapping-043
validation_run: run:validation-043
source_record: source:projects-q1-row-P001
```

Walkthrough นี้ต้องการชี้ให้เห็นสามประเด็น

- ข้อมูลแต่ละขั้นไม่ใช่ Artifact เดียวกัน: Source, Candidate, Staging Graph และ Published Graph มีสถานะต่างกัน
- แต่ละ Layer ตอบคำถามคนละข้อ: Parsing อ่านค่า, Entity Resolution เลือก Identity, Mapping เลือกความหมาย และ Validation ตัดสินความพร้อม
- เมื่อพบความผิด ต้องย้อนกลับไปแก้ Source, Registry หรือ Mapping Rule ที่เป็นต้นเหตุ ไม่ควร Patch Published Graph แล้วปล่อยให้ Pipeline สร้างความผิดเดิมอีก

เมื่อเส้นทางปกติและเส้นทางที่ Validation ไม่ผ่านชัดเจนแล้ว หัวข้อถัดไปจึงพิจารณาข้อผิดพลาดที่เกิดขึ้นเมื่อระบบจริงข้าม Checkpoint เหล่านี้หรือปล่อยให้แต่ละ Layer ใช้ Artifact คนละ Version

8.13 ข้อผิดพลาดตอนนำระบบขึ้นใช้งาน

Walkthrough ก่อนหน้าแสดงเส้นทางที่ควรเกิดขึ้น ส่วนการนำระบบขึ้นใช้จริงมักล้มเหลวเมื่อข้ามการตัดสินใจบางชั้น หัวข้อนี้จึงรวบรวมข้อผิดพลาดด้าน Architecture, Artifact, Environment และ Governance ไว้เป็นรายการตรวจ

1. **เริ่มจากเครื่องมือก่อน Architecture** การเลือก Triple Store หรือ Graph Database ไม่สามารถชดเชย Ontology, Mapping และ Query ที่ไม่มีขอบเขตชัดเจนได้
2. **ไม่มีข้อมูลตัวอย่างที่ผ่านทั้ง Pipeline** หากไม่มี Test Graph และ Expected Result จะระบุได้ยากว่าความผิดเกิดจาก Extraction, Mapping, Validation หรือ Query
3. **ให้ Application อ่าน Staging Graph โดยตรง** ข้อมูลที่ยังไม่ผ่าน Validation อาจถูกนำไปใช้ใน Dashboard หรือ API และทำให้ผลลัพธ์ผิดกระจายออกไป
4. **ไม่มี Owner ของ Ontology และ Query Catalog** การเปลี่ยน Property หรือ Query โดยไม่มี Review ทำให้คำศัพท์ซ้ำและ Artifact ใช้คนละ Version
5. **ไม่มี Migration Plan เมื่อ Ontology เปลี่ยน** Graph รุ่นเก่าและใหม่อาจปะปนกันจน Query มองเห็นข้อมูลไม่ครบหรืออ่านความหมายต่างกัน
6. **เปิดให้ระบบอัตโนมัติเข้าถึงข้อมูลโดยไม่จำกัดขอบเขต** Retrieval Policy ควรบังคับให้ใช้ Published Graph และ Query Template ที่ได้รับอนุญาต
7. **เลือก Knowledge Graph เพียงเพราะข้อมูลมีความสัมพันธ์** ชนิดคำถาม ความต้องการด้าน Interoperability และภาระการดูแลควรเป็นเหตุผลของการเลือก RDF/OWL ไม่ใช่การมี Relation เพียงอย่างเดียว

ข้อผิดพลาดเหล่านี้พบได้บ่อยเพราะ Knowledge Graph เกี่ยวข้องทั้งงานข้อมูล งานวิศวกรรมซอฟต์แวร์ และงานวิชาการด้านความหมาย การนำระบบขึ้นใช้จริงจึงต้องมีทั้งความเข้าใจ Ontology และวินัยของระบบ Production หากขาดด้านใดด้านหนึ่ง ระบบอาจทำงานได้ในการสาธิตแต่ยังไม่พร้อม

สำหรับการใช้งานระยะยาว การตรวจ Pipeline ครบทุกขั้นช่วยแก้ความเข้าใจผิดว่าแอปพลิเคชัน Knowledge Graph คือ Triple Store ที่มีส่วนติดต่อผู้ใช้เท่านั้น ความจริงแล้ว Triple Store เป็นเพียงส่วนเก็บและ Query ข้อมูล ความน่าเชื่อถือของระบบเริ่มก่อนหน้านั้นตั้งแต่การระบุแหล่งข้อมูล การรักษาไฟล์ต้นฉบับ การสร้าง IRI การ Map ข้อมูล และการตรวจผลลัพธ์ หากขั้นก่อนจัดเก็บไม่ชัด ฐานกราฟอาจทำงานเร็วแต่เก็บข้อเท็จจริงที่อธิบายที่มาไม่ได้

การมี Relation ไม่ได้ทำให้ RDF/OWL เป็นคำตอบที่เหมาะโดยอัตโนมัติ ฐานข้อมูลสัมพันธ์ยังเหมาะกับ Transaction และ Constraint ที่โครงสร้างคงที่ ระบบเอกสารหรือ JSON เหมาะกับข้อมูลที่อ่านและส่งต่อเป็นเอกสารหรือ Record ทั้งชุด Property Graph เหมาะกับการค้นตามเส้นทางภายในระบบที่ไม่ต้องแลกเปลี่ยนความหมายผ่านมาตรฐาน RDF ส่วน Vector Retrieval เหมาะกับการค้นข้อความที่คล้ายกันเมื่อยังไม่มี Vocabulary ที่คงที่ การเลือกระหว่างทางเลือกเหล่านี้จึงควรเริ่มจากชนิดของคำถามและภาระการดูแล (Angles and Gutierrez, 2008; Angles et al., 2017; Hogan et al., 2022; Fan et al., 2024)

เกณฑ์ตัดสินใจในระดับสถาปัตยกรรมสรุปไว้ในตารางที่ 8.4 ตารางนี้ไม่ได้แบ่งว่าเทคโนโลยีใดดีกว่าโดยรวม แต่แสดงว่าความได้เปรียบของ Semantic Web เกิดชัดเมื่อระบบต้องรวม Identity และ Vocabulary ข้ามแหล่ง ต้องแลกเปลี่ยนข้อมูลโดยรักษาความหมาย หรือต้องใช้ Ontology, Reasoning และ Provenance เป็นส่วนของคำตอบ

ตารางที่ 8.4: การเลือกแนวทางจัดการข้อมูลตามลักษณะของปัญหา

แนวทาง	เหมาะเมื่อ	ข้อจำกัดที่ต้องยอมรับ
Relational Database	Schema คงที่มี Transaction และ Constraint ชัด	การรวมความหมายข้ามระบบต้องออกแบบเพิ่ม
Document/JSON	ข้อมูลถูกส่งต่อเป็นเอกสารหรือ Record ที่เปลี่ยนรูปได้	Identity และ Relation ข้ามเอกสารไม่เกิดขึ้นเอง
Property Graph	เน้น Traversal และ Application ภายในระบบเดียว	Vocabulary และ Interoperability ไม่ได้อาศัยมาตรฐาน RDF โดยตรง
RDF/OWL Knowledge Graph	ต้องใช้ IRI ร่วม แลกเปลี่ยน Vocabulary และอธิบายความหมายเชิงตรรกะ	มีต้นทุนด้าน Ontology, Mapping, Reasoning และ Governance
Vector Retrieval	ต้องค้นบริบทข้อความกว้างและยังไม่มี Ontology ที่ครอบคลุม	Similarity ไม่รับรองว่า Relation และเงื่อนไขในคำถามเป็นจริง
Hybrid Graph–Vector	มีทั้งข้อเท็จจริงที่ต้องตรวจและข้อความประกอบที่ยังไม่ถูกแทนเป็น Triple	ต้องประเมินสอง Retrieval Path และรวม Provenance ให้เป็นระบบเดียว

ตารางที่ 8.4 ยังชี้กรณีที่ไม่ควรเริ่มจาก Ontology ได้แก่ระบบ CRUD ภายในที่ Schema ชัดและไม่มีการแลกเปลี่ยนความหมาย งานค้นเอกสารชั่วคราวที่ยังไม่มีผู้รับผิดชอบ Vocabulary หรือโครงการที่ไม่สามารถจัดสรรคนดูแล Mapping และ Version ได้ ในกรณีเหล่านี้การเพิ่ม Ontology อาจสร้างภาระมากกว่าคุณค่า ส่วนระบบที่ต้องตอบคำถามหลายเงื่อนไขข้ามเอกสารและต้องชี้หลักฐาน อาจใช้ RDF/OWL เป็นแกนแล้วเสริม Vector Retrieval เฉพาะข้อความที่ยังไม่อยู่ใน Graph การตัดสินเช่นนี้ทำให้ Semantic Web เป็นทางเลือกทางวิศวกรรมที่มีขอบเขต ไม่ใช่คำตอบสากลของทุกระบบข้อมูล

8.14 การติดตามและประเมิน Pipeline ในระบบจริง

รายการข้อผิดพลาดช่วยป้องกันปัญหาก่อน Deployment แต่ระบบที่ใช้งานแล้วต้องมีสัญญาณสำหรับตรวจว่าคุณภาพกำลังเปลี่ยนที่ Layer ใด หัวข้อนี้จึงเปลี่ยนจากการออกแบบ Pipeline ไปสู่ Monitoring, Regression Test และ Compatibility ระหว่าง Version

Walkthrough ก่อนหน้าแสดงเส้นทางของข้อมูลหนึ่งรายการครบแล้ว ส่วนการติดตามระบบหลังนำขึ้นใช้ต้องเปลี่ยนมุมมองจาก “ข้อมูลเดินผ่านขั้นใด” เป็น “แต่ละขั้นให้สัญญาณอะไรเมื่อคุณภาพเริ่มเปลี่ยน” การเฝ้าดูจึงต้องแยก Ingestion, Extraction, Identity Resolution, Mapping, Validation, Publish และ Query ออกจากกัน เพื่อให้ความผิดพลาดหนึ่งรายการย้อนกลับไปยังชั้นต้นเหตุได้

หาก Extraction อ่านผิดว่า SensorNet เป็นชื่อองค์กร ทีมไม่ควรแก้ Triple ในฐานข้อมูลแล้วจบ เพราะการนำเข้าเอกสารรอบถัดไปจะสร้างความผิดเดิมอีก ต้องแก้ Prompt, Rule หรือ Mapping ที่เป็นต้นเหตุ พร้อมเก็บตัวอย่างนี้เป็น Regression Test ในทำนองเดียวกัน หาก Entity Resolution รวมองค์กรคนละแห่งเข้าด้วยกัน ต้องแก้กฎการระบุตัวตน ไม่ใช่เพิ่ม Label เพื่อกลบปัญหา การแก้ที่ชั้นต้นเหตุเป็นหลักของระบบที่ดูแลได้

เมื่อใช้เครื่องมือสกัดข้อมูลอัตโนมัติ ชั้น Extraction ควรคืน Candidate ที่มีข้อความต้นทาง ตำแหน่งในเอกสาร และค่าความมั่นใจเท่าที่ใช้ประกอบการทบทวน ได้ Candidate ยังไม่ควรกลายเป็นข้อเท็จจริงที่เผยแพร่ทันที เพราะเครื่องมืออาจขาดข้อความ อ่านความสัมพันธ์กลับด้าน หรือสร้าง Class/Property ที่ไม่มีใน Ontology การบังคับให้ผลลัพธ์ผ่าน Schema Mapping และ SHACL ก่อน Publish ทำให้ระบบรักษาขอบเขตของความหมายได้

ส่วน API ไม่ควรคืนเฉพาะข้อความคำตอบ หากระบบออกแบบเพื่อความตรวจสอบได้ Response ควรมีอย่างน้อย Identifier ของ Resource, Graph Version, Source หรือ Claim ที่รองรับ และสถานะ Validation ตามความเหมาะสม ผู้ใช้ทั่วไปอาจเห็นเพียงคำตอบสรุป แต่ผู้ตรวจสอบควรเปิดดูหลักฐานได้จากข้อมูลชุดเดียวกัน การมี Evidence Bundle เช่นนี้ยังทำให้ส่วนติดต่อทุกแบบไม่ต้องสร้างคำอธิบายจากข้อมูลที่ไม่มีที่มา

เมื่อระบบถูกนำขึ้นใช้งาน การเฝ้าดูควรครอบคลุมมากกว่าสถานะ Server ต้องดูจำนวนเอกสารที่นำเข้า จำนวน Candidate ที่ถูกปฏิเสธ อัตรา SHACL Violation จำนวน Query ที่ล้มเหลว และเวลาที่ใช้ในแต่ละชั้น ตัวเลขเหล่านี้ช่วยแยกได้ว่าปัญหาเกิดจากคุณภาพข้อมูล ความเปลี่ยนแปลงของ Ontology หรือประสิทธิภาพของฐานกราฟ หากดูเพียงว่า Endpoint ตอบ HTTP ได้ ระบบอาจยังทำงานแต่คำตอบมีความครบถ้วนลดลง Versioning จึงต้องเชื่อม Artifact หลายชนิดเข้าด้วยกัน Release หนึ่งควรระบุ Ontology Version, Mapping Version, Shape Version, Query Catalog Version และ Graph Snapshot ที่เข้ากันได้ หากแก้ Ontology แต่ไม่แก้ Mapping หรือ Query ระบบอาจไม่แสดง Error ทันทีแต่คืนผลลัพธ์ผิด การบันทึก Compatibility และมีขั้น Rollback ช่วยให้การเปลี่ยน Semantic Layer มีวินัยเช่นเดียวกับการเปลี่ยน Software

เมื่อพิจารณาตั้งแต่ Source ถึงคำตอบ จะเห็นว่าแอปพลิเคชัน Knowledge Graph ที่ดีไม่ได้ซ่อนความซับซ้อนทั้งหมดไว้หลังคำว่า AI หรือ Graph แต่จัดความซับซ้อนให้เป็นชั้นที่ตรวจได้ ผู้เชี่ยวชาญโดเมนตรวจความหมาย ผู้ทำข้อมูลตรวจ Mapping นักพัฒนาตรวจ Pipeline และผู้ใช้ตรวจหลักฐานของคำตอบได้ การแบ่งความรับผิดชอบเช่นนี้เป็นสิ่งที่ทำให้ Ontology เคลื่อนจากเอกสารเชิงแนวคิดไปเป็นส่วนหนึ่งของระบบจริง

8.15 เครื่องมือสำหรับทดลองสร้างระบบ

หลักการในบทนี้ไม่ผูกกับ Tool ใด Tool หนึ่ง แต่การเลือก Tool ยังเป็นการตัดสินใจทางสถาปัตยกรรม เพราะ Tool แต่ละชนิดทำงานกับ Artifact คนละแบบและอาจรองรับ Reasoning, Validation, Versioning หรือการให้บริการข้อมูลไม่เท่ากัน หัวข้อนี้จึงไม่ได้ตอบว่า “ควรใช้ซอฟต์แวร์ใดดีที่สุด” แต่เสนอวิธีเลือก Tool จาก Layer ที่ต้องรับผิดชอบและผลลัพธ์ที่ต้องตรวจได้

จุดเริ่มต้นที่เหมาะสมคือแยกบทบาทของ Tool ออกจากกันก่อน เครื่องมือสร้าง Ontology ไม่จำเป็นต้องเป็น Triple Store และ Triple Store ก็ไม่ควรถูกใช้แทน Mapping Specification หรือ Validation Report ตารางที่ 8.5 สรุปบทบาทของกลุ่มเครื่องมือที่ใช้ทดลอง Pipeline ขนาดเล็กได้

ตารางที่ 8.5: การเลือก Tool ตาม Layer และ Artifact ที่ต้องการตรวจ

บทบาท	ตัวอย่าง Tool	Artifact และคำถามที่ใช้ตรวจ
Ontology Authoring	Protégé Desktop หรือ WebProtégé	ไฟล์ OWL, Axiom และผลจาก Reasoner; Vocabulary และ Class Expression ตรงตามความหมายที่ต้องการหรือไม่
RDF Processing	Apache Jena หรือ RDFLib	Test Graph, Mapping Output และ Query Result; ข้อมูลถูก Parse, สร้าง Triple และ Query ได้ตาม Expected Result หรือไม่
SHACL Validation	Apache Jena SHACL หรือ pySHACL	Shape Graph และ Validation Report; Candidate Graph ผ่าน Gate ใดและ Violation ชี้กลับไปยัง Source ได้หรือไม่
Graph Service	Jena Fuseki หรือ Triple Store ที่รองรับ SPARQL	Dataset, Named Graph, Endpoint และ Access Policy; Application อ่านเฉพาะ Published Graph ที่อนุญาตหรือไม่
Application Integration	OWLAPI, Jena API, RDFLib หรือ Library ของภาษาโปรแกรมที่ใช้	Code ที่เรียก Ontology, Graph และ Query; Application รักษา IRI, Version และ Evidence ไว้ครบหรือไม่
Version และ Test	Git Repository และ Test Runner ของโครงการ	Ontology Version, Mapping Version, Shape Version และ Regression Test; การเปลี่ยน Artifact หนึ่งกระทบ Query ใดบ้าง

Protégé เหมาะกับการเปิด แก้ และตรวจ OWL Ontology ในระดับแนวคิด ขณะที่ Apache Jena ครอบคลุมงาน RDF, SPARQL, SHACL และการให้บริการ SPARQL ผ่าน Fuseki ส่วน RDFLib เหมาะกับการทดลอง RDF และ SPARQL ในระบบนิเวศ Python และ pySHACL ใช้ตรวจ Data Graph กับ SHACL Shape ได้ การแบ่งบทบาทเช่นนี้อ้างอิงจากขอบเขตที่โครงการเหล่านี้ประกาศไว้ในเอกสารทางการ ไม่ได้หมายความว่าแต่ละระบบต้องใช้ Tool ชุดเดียวกันทั้งหมด (Stanford Center for Biomedical Informatics Research, n.d.; Apache Software Foundation, n.d.; RDFLib Team, n.d.-b, n.d.-a)

สำหรับผู้อ่านที่ต้องการทดลองแนวคิดของบทนี้โดยไม่สร้างระบบขนาดใหญ่ สามารถเริ่มจากเส้นทางขั้นต่ำหนึ่งรอบดังนี้

1. **สร้าง Ontology ขนาดเล็ก** กำหนดเพียง Project, Organization, Technology และ Report พร้อม Property ที่จำเป็นต่อ Competency Question หนึ่งข้อ
2. **เตรียม Source Record และ Expected Result** ใช้ข้อมูลตัวอย่างไม่กี่รายการ แต่กำหนดไว้ล่วงหน้าว่า Query ควรคืน Project และ Report ใด
3. **Map ข้อมูลไปยัง Test Graph** สร้าง IRI ตาม Policy เดียวกัน และเก็บ Source Pointer ของแต่ละ Record ไว้เพื่อให้ตรวจย้อนกลับได้
4. **ตรวจด้วย SHACL ก่อน Publish** จงใจเว้น Field สำคัญหนึ่งค่าเพื่อดูว่า Validation Report ระบุ Focus Node, Result Path และ Source Record ได้จริงหรือไม่
5. **Query Graph ที่ผ่าน Validation** เปรียบเทียบผลกับ Expected Result และตรวจว่า Evidence Bundle ยังมี Graph Version, Mapping Run และ Report ครบ
6. **เปลี่ยน Artifact หนึ่งรายการแล้ว Run ซ้ำ** เช่นเปลี่ยนชื่อ Property หรือทำ Shape ให้เข้มขึ้น จากนั้นตรวจว่า Regression Test ชี้ผลกระทบได้หรือไม่

เส้นทางนี้ตั้งใจให้ผู้อ่านเห็นวงจรของระบบมากกว่าความสามารถเฉพาะของ Tool หากทดลองเฉพาะการสร้าง Class ใน Ontology Editor จะยังไม่เห็นปัญหา Mapping และ Validation แต่หากเริ่มจาก Triple Store อย่างเดียวก็จะยังไม่ทราบว่า Vocabulary และ IRI Policy มาจากการตัดสินใจใด การทดลองที่มี Source, Mapping, Validation, Query และ Evidence ครบแม้ข้อมูลมีเพียงไม่กี่ Record จึงให้ภาพของ Knowledge Graph Application ได้ดีกว่าการสาธิตข้อมูลจำนวนมากแต่ตรวจที่มาไม่ได้

ก่อนเลือก Tool สำหรับระบบจริง ควรถามอย่างน้อยห้าข้อ

- **ทีมทำงานในระบบนิเวศใด** ภาษาโปรแกรมและทักษะของทีมมีผลต่อต้นทุนการดูแล Library และ Deployment ระยะยาว
- **ต้องใช้ Reasoning ระดับใด** ระบบที่ต้องตรวจ OWL 2 DL แตกต่างจากระบบที่ใช้เพียง RDFS หรือ OWL 2 RL จึงไม่ควรเลือก Reasoner จากชื่อ Tool เพียงอย่างเดียว
- **Validation อยู่ตรงไหนของ Pipeline** ต้องกำหนดว่า Tool ใดสร้าง Validation Report ใครเป็นผู้แก้ Source และเงื่อนไขใดอนุญาตให้ Publish
- **ข้อมูลและ Query มีขนาดเท่าใด** การทดลองใน Memory, การเก็บ Named Graph และการให้บริการ Endpoint มีข้อกำหนดด้าน Resource และการเข้าถึงต่างกัน
- **จะทำให้ผลลัพธ์ทำซ้ำได้อย่างไร** Version ของ Ontology, Mapping, Shape, Query และข้อมูลตัวอย่างต้องถูกบันทึกร่วมกัน ไม่เช่นนั้นการเปลี่ยน Tool อาจทำให้แยกไม่ออกว่าผลลัพธ์ต่างเพราะ Software หรือเพราะ Artifact

หนังสือเล่มนี้จึงไม่ลงรายละเอียดคำสั่งติดตั้งหรือ API ที่เปลี่ยนตาม Version ของซอฟต์แวร์ แต่ให้เกณฑ์ที่ใช้เปรียบเทียบ Tool ได้ เมื่อ Layer, Artifact และ Expected Result ถูกกำหนดชัด ทีมสามารถเปลี่ยน Tool ได้โดยไม่ต้องออกแบบความหมายของระบบใหม่ทั้งหมด และยังตรวจได้ว่าระบบใหม่รักษา Contract เดิมไว้หรือไม่ ประเด็นนี้นำไปสู่ข้อสรุปของบทว่า Knowledge Graph Application ที่ดูแลได้เกิดจากวินัยของ Pipeline ไม่ได้เกิดจากชื่อผลิตภัณฑ์เพียงอย่างเดียว

8.16 สรุป

บทนี้ขยาย Ontology จากแบบจำลองความหมายไปสู่ Application ที่ควบคุมเส้นทางของข้อมูลได้ ตั้งแต่ Source Inventory, Extraction และ Mapping ไปจนถึง Validation, Query, API และ Monitoring ภาพรวมนี้เป็นฐานสำหรับบทถัดไปซึ่งลงรายละเอียดเรื่อง SHACL, Provenance และ Semantic Governance

การสร้างแอปพลิเคชัน Knowledge Graph ต้องออกแบบทั้งชั้นความหมายและชั้นปฏิบัติพร้อมกัน โดยต้องแยก Source, Candidate, Staging Graph, Validated Graph และ Published Graph ออกจากกัน เพราะแต่ละ Artifact ผ่านการตัดสินใจไม่เท่ากัน Ontology ช่วยให้ระบบมีภาษากลาง RDF ทำให้ข้อมูลเชื่อมเป็น Graph SPARQL ทำให้ค้นคืนได้ SHACL ทำหน้าที่เป็น Gate และ Provenance ทำให้คำตอบย้อนกลับไปหาแหล่งที่มาได้ บทถัดไปจะลงรายละเอียดเรื่อง SHACL, Provenance และ Semantic Governance ซึ่งเป็นส่วนที่ทำให้ Graph ไม่ใช่เพียงข้อมูลที่เชื่อมกัน แต่เป็นความรู้ที่รับผิดชอบได้

เอกสารอ้างอิง

Angles, R., Arenas, M., Barceló, P., Hogan, A., Reutter, J., and Vrgoč, D. (2017). Foundations of modern query languages for graph databases. *ACM Computing Surveys*, 50(5):68:1–68:40.

Angles, R. and Gutierrez, C. (2008). Survey of graph database models. *ACM Computing Surveys*, 40(1):1–39.

Apache Software Foundation (n.d.). Apache Jena documentation. Official project documentation. Accessed October 18, 2026, from **https://jena.apache.org/documentation/**.

Carroll, J. J., Bizer, C., Hayes, P., and Stickler, P. (2005). Named graphs, provenance and trust. In *Proceedings of the 14th International Conference on World Wide Web*, pages 613–622. ACM.

Fan, W., Ding, Y., Ning, L., Wang, S., Li, H., Yin, D., Chua, T.-S., and Li, Q. (2024). A survey on RAG meeting LLMs: Towards retrieval-augmented large language models. In *Proceedings of the 30th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, pages 6491–6501. ACM.

Hogan, A., Blomqvist, E., Cochez, M., d'Amato, C., de Melo, G., Gutierrez, C., Kirrane, S., Gayo, J. E. L., Navigli, R., Neumaier, S., Ngomo, A.-C. N., Polleres, A., Rashid, S. M., Rula, A., Schmelzeisen, L., Sequeda, J., Staab, S., and Zimmermann, A. (2022). Knowledge graphs. *ACM Computing Surveys*, 54(4):1–37.

Labra Gayo, J. E., Prud'hommeaux, E., Boneva, I., and Kontokostas, D. (2017). *Validating RDF Data*. Morgan & Claypool.

Neumann, T. and Weikum, G. (2010). The RDF-3X engine for scalable management of RDF data. *The VLDB Journal*, 19(1):91–113.

RDFLib Team (n.d.-a). pySHACL: A Python validator for SHACL. Official project repository. Accessed October 18, 2026, from **https://github.com/RDFLib/pySHACL**.

RDFLib Team (n.d.-b). RDFLib: Python software for working with RDF. Official project documentation. Accessed October 18, 2026, from **https://rdflib.readthedocs.io/**.

Sequeda, J. and Lassila, O. (2021). *Designing and Building Enterprise Knowledge Graphs*. Morgan & Claypool.

Stanford Center for Biomedical Informatics Research (n.d.). Protégé: A free, open-source ontology editor. Official project documentation. Accessed October 18, 2026, from **https://protege.stanford.edu/**.

World Wide Web Consortium (2012). R2RML: RDB to RDF mapping language. W3C Recommendation. Accessed July 9, 2026, from https://www.w3.org/TR/r2rml/.

World Wide Web Consortium (2013). PROV-O: The PROV ontology. W3C Recommendation. Accessed June 27, 2026, from https://www.w3.org/TR/prov-o/.

World Wide Web Consortium (2017). Shapes constraint language (SHACL). W3C Recommendation. Accessed July 3, 2026, from https://www.w3.org/TR/shacl/.

9

SHACL, Provenance และ Semantic Governance

เมื่อ Knowledge Graph ถูกนำไปใช้ในระบบจริง คำถามจะเปลี่ยนจาก “มี Graph หรือยัง” ไปเป็น “Graph นี้เชื่อถือได้แค่ไหน” บทนี้จึงเชื่อมงาน Ontology กับการควบคุมคุณภาพข้อมูล โดยอธิบาย SHACL ในฐานะเครื่องมือ Validation, Provenance ในฐานะหลักฐานย้อนกลับ และ Semantic Governance ในฐานะวินัยของการดูแล Ontology, Mapping, Query และข้อมูลให้เปลี่ยนแปลงอย่างตรวจสอบได้

9.1 ทำไม Graph ต้องถูกตรวจสอบ

บทก่อนแสดงเส้นทางจาก Source ไปยัง Published Graph แต่การผ่านแต่ละ Layer ยังต้องมีเกณฑ์ว่าข้อมูลพร้อมใช้เมื่อใด หัวข้อนี้จึงเริ่มจากความแตกต่างระหว่าง Graph ที่เขียนเป็น RDF ได้กับ Graph ที่มีคุณภาพเพียงพอสำหรับตอบคำถามของระบบ

เมื่อ Knowledge Graph ถูกใช้เป็นฐานความรู้ของระบบจริง ปัญหาไม่ได้อยู่เพียงว่าข้อมูลถูกเขียนเป็น RDF แล้วหรือไม่ แต่อยู่ที่ข้อมูลนั้นถูกต้อง ครบถ้วน และตรวจสอบย้อนกลับได้หรือไม่ RDF และ OWL ช่วยอธิบายความหมายของข้อมูล แต่ไม่ได้ถูกออกแบบมาเพื่อรายงานข้อผิดพลาดของข้อมูลในลักษณะเดียวกับ Validation Rule ของระบบงาน ดังนั้น SHACL จึงมีบทบาทสำคัญในการกำหนดเงื่อนไขว่าข้อมูลใน Graph ควรมีรูปแบบอย่างไร (World Wide Web Consortium, 2017; Labra Gayo et al., 2017) ในทางปฏิบัติ ข้อมูล Candidate Graph ต้องผ่าน SHACL Validation พร้อมบันทึก Provenance และการตัดสินใจด้าน Governance ก่อนเผยแพร่เป็น Knowledge Graph ที่ใช้ตอบคำถามได้ ตัวอย่างเช่น ในระบบจัดการโครงการ อาจต้องการบอกว่า Resource ที่เป็น **ex:Project** ต้องมี Label อย่างน้อยหนึ่งค่า ต้องมีหน่วยงานรับผิดชอบ และต้องมีสถานะปัจจุบัน เงื่อนไขเหล่านี้ไม่ได้เป็นความหมายเชิง Ontology ทั้งหมด แต่เป็นข้อกำหนดเชิงคุณภาพของข้อมูลที่ Application ต้องการ หากข้อมูลไม่ผ่าน Shape ระบบควรรายงานได้ว่าผิดตรงไหนและต้องแก้อย่างไร

9.2 Validation ในรูปฟังก์ชัน

เมื่อเห็นเหตุผลของการตรวจ Graph แล้ว ขั้นต่อไปคือกำหนด Input และ Output ของ Validation ให้ชัด มุมมองแบบฟังก์ชันช่วยแยก Data Graph, Shape Graph และ Validation Report ออกจากกัน ก่อนพิจารณารายละเอียดของ SHACL

ในเชิงคณิตศาสตร์ สามารถมอง Validation เป็นฟังก์ชันที่รับ Graph และชุด Shape แล้วคืนผลลัพธ์เป็นรายงานการตรวจสอบได้

$$V(G, S) \to R$$

ในสมการนี้ V คือกระบวนการตรวจสอบ, G คือ RDF Graph ที่ต้องการตรวจ, S คือชุดของ SHACL Shapes ที่ทำหน้าที่เป็นเงื่อนไขหรือ Contract ของข้อมูล และ R คือ Validation Report ที่บอกว่าข้อมูลผ่านหรือมี Violation ใดบ้าง หาก R ไม่มี Violation Graph นั้นอาจถือว่าผ่านเงื่อนไขของระบบในรอบนั้น แต่ไม่ได้แปลว่า Graph เป็นความจริงสมบูรณ์ของโลก เพียงแปลว่าผ่านข้อกำหนดที่กำหนดไว้เท่านั้น ความเข้าใจนี้สำคัญ เพราะ Validation เป็นการตรวจตาม Contract ไม่ใช่การพิสูจน์ความจริงทั้งหมด

หากแยกความหมายของสมการนี้ให้ละเอียดขึ้น G ไม่จำเป็นต้องเป็น Graph ทั้งระบบเสมอไป แต่อาจเป็น Staging Graph ของข้อมูลที่กำลังจะ Publish, Named Graph ของรายงานหนึ่งชุด หรือ Graph ตัวอย่างที่ใช้ทดสอบ Mapping ก็ได้ ส่วน S อาจมีหลาย Shape สำหรับหลายบทบาท เช่น Shape สำหรับ Project, Shape สำหรับ Claim, Shape สำหรับ Technology และ Shape สำหรับ Provenance เมื่อรัน $V(G, S)$ ระบบจึงไม่ได้ถามว่า Graph นี้ดีในทุกความหมายหรือไม่ แต่ถามว่า Graph นี้ผ่านเงื่อนไขที่ชุด Shape นี้กำหนดไว้หรือไม่

ส่วน R ไม่ควรถูกมองเป็นผลข้างเคียงของโปรแกรม แต่ควรถูกมองเป็น Evidence ของ Pipeline หากรอบหนึ่ง Graph ผ่าน Validation ควรเก็บได้ว่าใช้ Graph Version ใด Shape Version ใด และเครื่องมือใดตรวจ หากรอบหนึ่งไม่ผ่าน ควรรู้ว่า Violation อยู่ที่ Node ใด path ใด และควรแก้ที่ Source หรือ Mapping ใด การมอง Validation Report เป็น Artifact ทำให้ SHACL เชื่อมกับ Governance ได้จริง ไม่ใช่เป็นเพียงคำสั่งที่รันก่อนโหลดข้อมูลเข้า Triple Store ตัวอย่าง SHACL Shape อย่างง่ายคือ

```
ex:ProjectShape a sh:NodeShape ;
 sh:targetClass ex:Project ;
 sh:property [
 sh:path rdfs:label ;
 sh:minCount 1 ;
 ] ;
 sh:property [
 sh:path ex:responsibleOrg ;
 sh:minCount 1 ;
 sh:class ex:Organization ;
 ] .
```

Shape นี้อ่านได้ว่า Project ทุกตัวควรมี Label อย่างน้อยหนึ่งค่า และควรมี Responsible Organization อย่างน้อยหนึ่งค่า โดย Organization นั้นควรเป็น Instance ของ Class ที่กำหนดไว้ การอ่าน Shape ควรแยกเป็นสามส่วน

- **sh:targetClass ex:Project** ระบุว่าระบบจะตรวจ Node ใดบ้าง
- **sh:path** ระบุ Property ของ Node ที่ต้องตรวจ
- Constraint เช่น **sh:minCount** หรือ **sh:class** ระบุเงื่อนไขของค่านั้น

หาก Graph มี Resource ที่เป็น **ex:Project** แต่ไม่มี **rdfs:label** อย่างน้อยหนึ่งค่า Report จะเกิด Violation ที่ Path **rdfs:label** หากมี **ex:responsibleOrg** แต่ Object ไม่ใช่ **ex:Organization** ก็จะเกิด Violation อีกแบบหนึ่ง

การแยกเช่นนี้ทำให้ผู้ดูแลข้อมูลอ่าน Report ได้ตรงจุดมากขึ้น ไม่ใช่เห็นคำว่า Validation Error แล้วไม่รู้ว่าต้องแก้ตรงไหน หาก Violation เกิดจาก **sh:minCount** อาจต้องเติมข้อมูล หากเกิดจาก **sh:class** อาจต้องตรวจ type ของ Object หรือ Mapping ที่สร้าง Object นั้น หากเกิดจาก Datatype อาจต้องแก้ Parser หรือ format ของข้อมูลต้นทาง ดังนั้น SHACL ไม่ได้มีประโยชน์แค่บอกผ่านหรือไม่ผ่าน แต่ช่วยชี้ทิศทางการตรวจหาข้อผิดพลาด Graph ด้วย

9.3 Provenance และ Named Graph

Validation ตอบว่าข้อมูลผ่านเงื่อนไขหรือไม่ แต่ยังไม่ตอบว่า Claim มาจาก Source ใดหรือเกิดจาก Activity ใด Provenance และ Named Graph จึงเพิ่มบริบทที่ทำให้ผลการตรวจและข้อเท็จจริงใน Graph ย้อนกลับไปยังที่มาได้

การตรวจรูปแบบข้อมูลยังไม่พอ หากระบบต้องใช้ Graph เพื่ออธิบายหรือสนับสนุนการตัดสินใจ ต้องรู้ด้วยว่า Claim แต่ละชุดมาจากที่ใด ใครสร้าง เมื่อไร และผ่านกระบวนการใด Provenance จึงเป็นชั้นสำคัญของ Semantic Governance โดยมาตรฐาน PROV-O ช่วยให้บันทึกความสัมพันธ์ระหว่าง Entity, Activity และ Agent ได้เป็น RDF (World Wide Web Consortium, 2013) ในงานจริงการใช้ Named Graph ช่วยแยกบริบทของ Claim ได้ดี เช่น Graph หนึ่งอาจเก็บข้อมูลจากรายงานไตรมาสแรก อีก Graph หนึ่งเก็บข้อมูลจาก Sensor และอีก Graph หนึ่งเก็บข้อมูลที่ผ่านการตรวจแล้ว การแยกเช่นนี้ป้องกันไม่ให้ Claim ต่างบริบทปะปนอยู่ใน Graph เดียว และช่วยให้ตรวจสอบย้อนกลับได้ว่า Query หนึ่งใช้หลักฐานจาก Graph ใด (Carroll et al., 2005)

แนวคิดนี้มีความสำคัญต่อระบบรายงานและระบบสนับสนุนการตัดสินใจ หากหน้าจอระบุว่าโครงการหนึ่งลดเวลาการตอบสนองลง ระบบควรสามารถชี้กลับไปได้ว่า Claim นี้มาจาก Report ใด Metric ใด และถูกตรวจเมื่อไร หากไม่มี Provenance ข้อความอาจอ่านดูน่าเชื่อแต่ไม่สามารถตรวจสอบได้ PROV-O มีคำพื้นฐานที่ช่วยอธิบายเส้นทางนี้อย่างเป็นระบบ ได้แก่ Entity, Activity และ Agent โดย Entity คือสิ่งที่ถูกใช้หรือถูกสร้าง เช่น Report, Graph หรือ Claim Activity คือกระบวนการที่สร้างหรือแปลงสิ่งนั้น เช่น Mapping run, Validation run หรือ Review Activity และ Agent คือคน ทีม หรือระบบที่เกี่ยวข้องกับ Activity นั้น เมื่อ Claim หนึ่งถูกบอกว่า **prov:wasGeneratedBy** Activity ใด และ Activity นั้น **prov:used** Source ใด ระบบจึงสามารถเล่าเส้นทางของ Claim ได้ตั้งแต่เอกสารต้นทางจนถึง Graph ที่ใช้ตอบคำถาม

การใช้ Named Graph ช่วยแยก Claim, Provenance และ Validation Report ออกจากกันตามบริบท เช่น Graph หนึ่งอาจเก็บ Claim จากรายงานไตรมาสแรก อีก Graph หนึ่งเก็บ Provenance ของการ import และอีก Graph หนึ่งเก็บ Validation Report ของรอบนั้น การแยกเช่นนี้ทำให้ Query สามารถถามได้ทั้งคำตอบและบริบท เช่น Claim นี้อยู่ใน Graph ใด Graph นี้ถูกสร้างจาก Source ใด และ Source นั้นผ่าน Validation ด้วย Shape Version ใด หากระบบต้องตอบคำถามที่มีความเสี่ยงสูง บริบทเหล่านี้ไม่ใช่รายละเอียดเสริม แต่เป็นส่วนหนึ่งของคำตอบ

9.4 Semantic Governance ในระบบจริง

เมื่อระบบมีทั้ง Validation Report และ Provenance แล้ว ยังต้องกำหนดว่าใครตัดสินใจแก้หรือเผยแพร่ข้อมูล และ Artifact แต่ละชนิดเปลี่ยนร่วมกันอย่างไร Semantic Governance จึงเชื่อมหลักฐานทางเทคนิคเข้ากับความรับผิดชอบและวงจรการเปลี่ยนแปลงของระบบ

Semantic Governance หมายถึงการจัดการให้ Ontology, Mapping, Shapes, Query และข้อมูลเปลี่ยนแปลงอย่างมีวินัย ไม่ใช่เพียงการตั้งคณะกรรมการดูแลคำศัพท์ หาก Ontology เปลี่ยนแต่ Shape ไม่เปลี่ยน หรือ Mapping เปลี่ยนแต่ Query Catalog ไม่ถูกปรับ ระบบจะคืนผลลัพธ์ที่ไม่น่าเชื่อถือได้โดยไม่มี Error ที่ชัดเจน Artifact ขั้นต่ำที่ควรมีคือ Ontology Version, SHACL Shape Version, Mapping Specification, Query Catalog, Validation Report และ Provenance Policy แต่ละ Artifact ควรถูกเก็บเป็นไฟล์หรือ Graph ที่ตรวจสอบได้ ไม่ใช่พึ่งความรู้ที่ไม่ได้บันทึกของผู้พัฒนาเพียงคนเดียว แนวทางนี้สอดคล้องกับงานวิจัยด้านคุณภาพของ Linked Data และ Knowledge Graph refinement ที่เน้นว่าคุณภาพของ Graph ต้องถูกวัดและปรับปรุงอย่างต่อเนื่อง (Zaveri et al., 2016; Paulheim, 2017)

เมื่อเข้าใจบทบาทของ SHACL และ Provenance แล้ว ขั้นต่อไปคือการนำสองส่วนนี้ไปใช้เป็นระบบกำกับความรู้ที่ทำซ้ำได้ ไม่ใช่ใช้ตรวจเฉพาะตอนสร้าง Graph ครั้งแรก รายละเอียดต่อไปนี้จึงลงลึกที่รูปแบบของ Shape, Validation Report, Provenance Pattern และ Metric ที่ช่วยให้ทีมดูแล Knowledge Graph ได้ในระยะยาว

9.5 Node Shape และ Property Shape

ภาพรวมของ Semantic Governance อาศัย Rule ที่ตรวจได้จริง หัวข้อนี้จึงกลับมาที่โครงสร้างพื้นฐานของ SHACL และแยกหน้าที่ของ Node Shape ซึ่งเลือก Resource ที่ต้องตรวจ ออกจาก Property Shape ซึ่งกำหนดเงื่อนไขของค่าบนเส้นทางหนึ่ง

SHACL มีแนวคิดหลักคือ Node Shape และ Property Shape โดย Node Shape ใช้กำหนดว่า Resource กลุ่มใดต้องถูกตรวจ และ Property Shape ใช้กำหนดว่า Property ใดต้องมีเงื่อนไขอย่างไร ตัวอย่างเช่น **ex:ProjectShape** อาจ target ไปที่ทุก Resource ที่เป็น **ex:Project** แล้วกำหนด Property Shape สำหรับ **rdfs:label**, **ex:responsibleOrg** และ **ex:status** ตัวอย่างที่ละเอียดขึ้นคือ

```
ex:ProjectShape a sh:NodeShape ;
 sh:targetClass ex:Project ;
 sh:property [
 sh:path rdfs:label ;
 sh:datatype rdf:langString ;
 sh:minCount 1 ;
 ] ;
 sh:property [
 sh:path ex:responsibleOrg ;
 sh:class ex:Organization ;
 sh:minCount 1 ;
 ] ;
 sh:property [
 sh:path ex:status ;
```

```
sh:in (ex:Planning ex:Running ex:Closed) ;
sh:minCount 1 ;
] .
```

Shape นี้ช่วยตรวจสามเรื่องพร้อมกัน คือ Project ต้องมี Label, Organization ที่รับผิดชอบต้องเป็น Class ที่กำหนด และ status ต้องอยู่ในชุดค่าที่อนุญาต ความสำคัญของ Shape ไม่ได้อยู่ที่ Syntax แต่คือการทำให้ข้อกำหนดของข้อมูลถูกเขียนเป็น Artifact ที่ตรวจซ้ำได้

9.6 Validation Report ในฐานะ Evidence

Shape จะมีประโยชน์ต่อ Governance เมื่อผลการตรวจถูกเก็บและอ้างอิงได้ Validation Report จึงควรถูกมองเป็น Evidence ของการตัดสินใจ Publish ไม่ใช่ข้อความ Error ชั่วคราวที่หายไปหลัง Pipeline ทำงานเสร็จ

เมื่อ SHACL ตรวจ Graph แล้ว ผลลัพธ์จะออกมาเป็น Validation Report ซึ่งสามารถเขียนเป็น RDF ได้เช่นกัน รายงานนี้มักบอกว่า Graph ผ่านหรือไม่ Violation เกิดที่ Node ใด path ใด และข้อความอธิบายคืออะไร หากเก็บ Report เป็น RDF หรืออย่างน้อยเป็นไฟล์ที่กำหนดรุ่นได้ จะมีหลักฐานว่าข้อมูลชุดหนึ่งผ่านการตรวจ ณ เวลาใด ในระบบที่นำ Graph ไปสร้างรายงานหรือคำตอบอัตโนมัติ Validation Report มีบทบาทเพิ่มขึ้น เพราะสามารถใช้เป็นเงื่อนไขก่อนส่ง Graph ไปยังชั้นแสดงผล หาก Graph ยังไม่ผ่าน Shape สำคัญ ระบบอาจไม่ควรใช้ Graph นั้น หรือควรแสดงคำเตือนว่าผลลัพธ์อ้างอิงข้อมูลที่ยังไม่สมบูรณ์ วิธีนี้ทำให้ Semantic Governance เชื่อมกับความน่าเชื่อถือของ Application โดยตรง

9.7 Provenance Pattern สำหรับ Claim

Validation Report เป็นหลักฐานว่าข้อมูลผ่านเงื่อนไข ส่วน Provenance ของ Claim อธิบายว่าข้อความหนึ่งมาจาก Source และ Activity ใด หัวข้อนี้จึงเสนอ Pattern สำหรับทำให้ Claim ที่สำคัญมี Identity และเชื่อมกลับไปยังหลักฐานได้

Claim ที่สำคัญควรมี Provenance เสมอ ตัวอย่างเช่น Claim ว่าโครงการ A มี Outcome X ไม่ควรถูกเก็บเป็น Triple ที่ไม่มีข้อมูลที่มา หาก Outcome นี้ใช้ในการประเมินหรือรายงาน ควรมีข้อมูลว่า Claim มาจากเอกสารใด ถูกสร้างโดย Activity ใด และถูกสร้างเมื่อไร

```
claim:001 a ex:Claim ;
ex:aboutProject proj:A ;
ex:assertedOutcome outcome:ReducedDelay ;
prov:wasDerivedFrom report:Q1 ;
prov:generatedAtTime "2026-04-01"^^xsd:date .
```

การสร้าง Claim เป็น Resource อาจดู verbose แต่มีประโยชน์มากเมื่อ Claim ต้องมี Reviewer, Confidence, Version หรือสถานะการอนุมัติ หากใช้เพียง Triple พื้นฐาน จะบันทึก Metadata เหล่านี้ได้ยากกว่า ดังนั้นการออกแบบ Provenance จึงควรถูกตัดสินจากความสำคัญของ Claim ไม่ใช่ใช้ Pattern เดียวกับทุกข้อมูลเสมอไป

9.8 Governance Metric

เมื่อ Claim และ Validation Report ถูกเก็บเป็น Artifact ระบบสามารถวัดคุณภาพจากข้อมูลจริงแทนการประเมินด้วยความรู้สึก หัวข้อนี้จึงเลือก Metric ที่สัมพันธ์กับคำถามและความเสี่ยงของระบบ โดยไม่ใช้จำนวน Triple เป็นตัวแทนคุณภาพทั้งหมด

Semantic Governance ควรมี Metric ที่วัดได้ เช่น จำนวน Violation ต่อ Source, จำนวน Resource ที่ไม่มี Label, จำนวน IRI ซ้ำที่ถูก merge, จำนวน Query สำคัญที่ผ่าน test และจำนวน Mapping Rule ที่เปลี่ยนในแต่ละ Release Metric เหล่านี้ช่วยให้ทีมรู้ว่าคุณภาพของ Graph ดีขึ้นหรือแย่ลง ตัวอย่างเช่น หาก Release ใหม่ทำให้ Violation ลดลง แต่ Query สำคัญบางตัวคืนผลลัพธ์น้อยลงมาก อาจแปลว่า Shape เข้มขึ้นจนข้อมูลบางส่วนถูกตัดออก หรือ Mapping เปลี่ยนจน Resource เดิมหายไป การดู Metric หลายตัวร่วมกันจึงสำคัญ เพราะคุณภาพของ Knowledge Graph ไม่ได้มีมิติเดียว

Metric ที่ดีควรผูกกับคำถามของระบบ เช่นถ้าระบบใช้ Graph เพื่อตอบคำถามเรื่อง Outcome ที่มีหลักฐาน Metric สำคัญอาจไม่ใช่จำนวน Triple ทั้งหมด แต่เป็นสัดส่วนของ Outcome Claim ที่มี Report รองรับ จำนวน Claim ที่ไม่มี Provenance และจำนวน Query หลักที่ผ่าน Expected Result หาก Metric สนใจแต่ขนาด Graph ทีมอาจเพิ่มข้อมูลจำนวนมากแต่ไม่ได้เพิ่มความน่าเชื่อถือของคำตอบ อีกด้านหนึ่ง Metric ควรถูกใช้เพื่อเรียนรู้ ไม่ใช่เพื่อโทษคนทำข้อมูล หาก Source หนึ่งสร้าง Violation บ่อย อาจแปลว่า Template ของรายงานไม่ชัด หรือ Mapping Rule ยังไม่รองรับรูปแบบข้อมูลจริง ไม่จำเป็นต้องแปลว่าผู้กรอกข้อมูลผิดเสมอไป การใช้ Metric อย่างสร้างสรรค์ทำให้ Governance เป็นกระบวนการปรับปรุงระบบ ไม่ใช่การตรวจจับความผิดอย่างเดียว

9.9 Governance ช่วยควบคุมการเปลี่ยนแปลง

Metric ช่วยบอกว่าสถานะของระบบเปลี่ยนไปอย่างไร แต่การตอบสนองต่อการเปลี่ยนแปลงต้องมีผู้รับผิดชอบและขั้น Review หัวข้อนี้จึงอธิบาย Governance ในฐานะกลไกลดผลกระทบของการเปลี่ยน Ontology, Mapping, Shape และ Query

หลายทีมมอง Governance เป็นภาระเพราะเพิ่มขั้นตอนอนุมัติ แต่ในบริบทของ Semantic Web Governance ที่ออกแบบร่วมกับ Pipeline ช่วยลดเวลาค้นหาสาเหตุของปัญหา ลดความสับสนของคำศัพท์ และทำให้ทีมทราบว่า Ontology แต่ละ Version ใช้กับข้อมูลชุดใด หากไม่มี Governance การเปลี่ยนชื่อ Property หนึ่งรายการอาจทำให้ Query หลายตัวคืนผลผิดโดยไม่มีผู้รับผิดชอบ หากไม่มี Provenance การแก้ข้อมูลผิดหนึ่งจุดอาจต้องค้นเอกสารย้อนหลังทั้งหมด และหากไม่มี Validation ข้อมูลผิดอาจเข้าไปใน Graph และถูกใช้สร้างคำตอบต่อผู้ใช้ Governance จึงเป็นกลไกควบคุมความน่าเชื่อถือของระบบ ไม่ใช่เพียงงานเอกสาร

ในงานจริง Governance ควรระบุให้ชัดว่าการเปลี่ยน Artifact แต่ละชนิดต้องตรวจอะไร เช่นถ้าเปลี่ยน Ontology ต้องรัน Query test ชุดใด ถ้าเปลี่ยน Mapping ต้องดู Validation Report ใด และถ้าเพิ่ม Source ใหม่ต้องปรับ Provenance Policy ส่วนใด ขั้นตอนที่กำหนดไว้ล่วงหน้าช่วยลดเวลาค้นหาสาเหตุเมื่อระบบผิดพลาด เพราะแสดงความสัมพันธ์ระหว่าง Artifact แต่ละชนิด Semantic Governance จึงควรอยู่ในรูปกติกาที่ตรวจได้ใน Pipeline เช่น naming Policy, IRI Policy, Shape Release Policy, Query

test Policy และ Provenance Policy เมื่อกติกาเหล่านี้อยู่ใน Repository เดียวกับ Ontology, Mapping และ Query ทีมจะใช้ Governance เป็นส่วนหนึ่งของงานประจำแทนการแยกไว้ในเอกสารอีกชุดหนึ่ง

9.10 กรณีศึกษาแบบต่อเนื่อง: Governance ของข้อมูลโครงการ

แนวคิด Governance อาจดูเป็นนามธรรมหากไม่ผูกกับการตัดสินใจ Publish หัวข้อนี้จึงกลับมายังข้อมูลโครงการและแสดงว่าข้อมูลจากหลาย Source ต้องผ่าน Validation, Provenance และ Policy ใดก่อนเข้าสู่ Graph ที่ผู้ใช้เชื่อถือ

กลับมาที่ระบบโครงการดิจิทัล สมมติว่าองค์กรมีข้อมูลโครงการหลายสิบหรือหลายร้อยโครงการ ข้อมูลเหล่านี้มาจากหลายแหล่ง เช่น ระบบบริหารโครงการ รายงานความก้าวหน้า เอกสารงบประมาณ และไฟล์ประกอบการประชุม หากไม่มี Governance โครงการบางรายการอาจไม่มีหน่วยงานรับผิดชอบ บางรายการใช้ชื่อสถานะไม่ตรงกัน บางรายการมี Outcome แต่ไม่มีหลักฐาน และบางรายการถูกอัปเดตโดยไม่บันทึก Source ต้นทาง

SHACL Shape สำหรับระบบนี้ควรเริ่มจากข้อกำหนดขั้นต่ำที่ Application ต้องใช้จริง ไม่ใช่เขียน Shape ให้ครอบคลุมทุกความเป็นไปได้ตั้งแต่วันแรก ตัวอย่างเช่น Project ที่จะ Publish ได้อาจต้องมี Label ภาษาไทย หน่วยงานรับผิดชอบ สถานะ และวันที่เริ่มต้น ส่วน Outcome Claim ต้องมี Project ที่เกี่ยวข้อง Outcome ที่กล่าวอ้าง และ Provenance ไปยัง Report อย่างน้อยหนึ่งรายการ

```
ex:OutcomeClaimShape a sh:NodeShape ;
 sh:targetClass ex:OutcomeClaim ;
 sh:property [
 sh:path ex:aboutProject ;
 sh:class ex:Project ;
 sh:minCount 1 ;
 ] ;
 sh:property [
 sh:path ex:assertedOutcome ;
 sh:minCount 1 ;
 ] ;
 sh:property [
 sh:path prov:wasDerivedFrom ;
 sh:class ex:Report ;
 sh:minCount 1 ;
 ] .
```

Shape นี้ไม่ได้พิสูจน์ว่า Outcome เป็นความจริงทั้งหมด แต่ตรวจว่าหากระบบจะยอมรับ Outcome Claim หนึ่ง Claim นั้นต้องผูกกับ Project และต้องมี Report รองรับอย่างน้อยหนึ่งรายการ ความต่างนี้สำคัญมาก เพราะ Validation ไม่ใช่การตัดสินความจริงของโลก แต่เป็นการตรวจว่า Claim มีโครงสร้างและหลักฐานขั้นต่ำตามที่ระบบกำหนดหรือไม่ เมื่อ Validation ทำงาน ระบบควรเก็บ Report ไว้เป็น Artifact เช่น

```
validation:run-2026-07-01 a ex:ValidationRun ;
 ex:validatedGraph graph:staging-2026-07-01 ;
 ex:usedShapeVersion "shape-1.2.0" ;
```

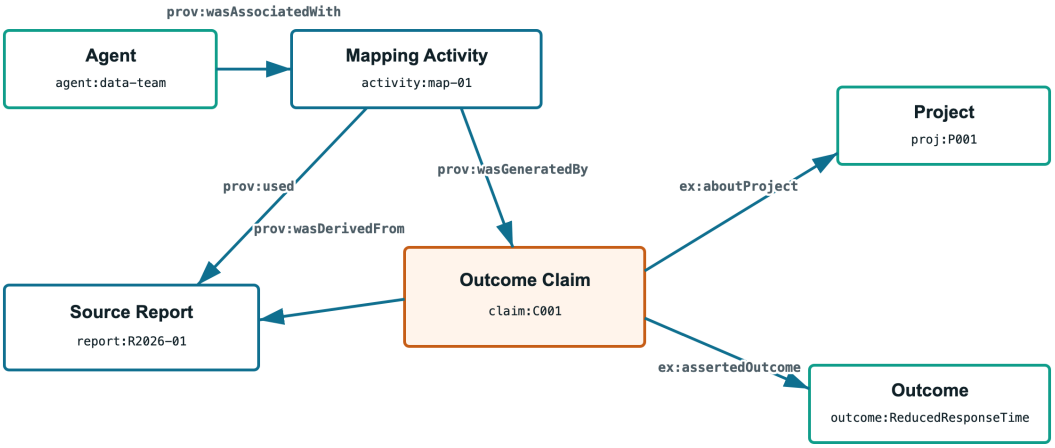
```
ex:hasViolationCount 14 ;
prov:generatedAtTime "2026-07-01T10:30:00"^^xsd:dateTime .
```

การเก็บ Validation run ทำให้ทีมเห็นประวัติคุณภาพข้อมูล หากเดือนก่อนมี Violation 80 รายการ และเดือนนี้เหลือ 14 รายการ แปลว่าคุณภาพข้อมูลดีขึ้นในมิติหนึ่ง แต่ต้องดูต่อด้วยว่า Query สำคัญยังคืนผลลัพธ์ครบหรือไม่ เพราะการลด Violation อาจเกิดจากการตัดข้อมูลออกมากเกินไปก็ได้ Provenance ของระบบโครงการควรบอกทั้งระดับ Source และระดับ Activity เช่น Outcome Claim หนึ่งอาจ derived from รายงานไตรมาสที่ 1 และ generated by Mapping Activity ครั้งหนึ่ง หาก Claim ถูกแก้ภายหลัง ควรมี Activity ใหม่ที่บอกว่าใครแก้ แก้เมื่อไร และอ้างอิง Source ใด การเก็บเช่นนี้ช่วยให้ระบบไม่สูญเสียประวัติของความหมาย

```
claim:C001 a ex:OutcomeClaim ;
 ex:aboutProject proj:P001 ;
 ex:assertedOutcome outcome:ReducedResponseTime ;
 prov:wasDerivedFrom report:R2026-01 ;
 prov:wasGeneratedBy activity:mapping-2026-07-01 .

activity:mapping-2026-07-01 a prov:Activity ;
 prov:used file:project-report-q1.pdf ;
 prov:wasAssociatedWith agent:data-team ;
 prov:endedAtTime "2026-07-01T10:20:00"^^xsd:dateTime .
```

ความสัมพันธ์ระหว่าง Claim, Source, Activity, Agent, Project และ Outcome ในตัวอย่างนี้แสดงในภาพที่ 9.1 โดยชื่อ Predicate บนเส้นช่วยบอกว่าทรัพยากรแต่ละรายการเกี่ยวข้องกันในบทบาทใด



ภาพที่ 9.1: เส้นทาง Provenance ระดับ Claim จากข้อสรุปกลับไปยังรายงานและกิจกรรมที่สร้างข้อมูล

ภาพที่ 9.1 แสดงว่า Claim ไม่ได้เชื่อมกับ Source Report ด้วยลูกศรที่หมายถึง “มาก่อน–มาทีหลัง” แต่เชื่อมด้วย Predicate ที่กำหนดความหมาย เช่น **prov:wasDerivedFrom** และ **prov:wasGeneratedBy** ขณะเดียวกัน Activity ระบุ Agent และ Source ที่ใช้ ส่วน Claim ระบุ Project กับ Outcome ที่กล่าวถึง โครงสร้างนี้ทำให้ระบบเชื่อมข้อความคำตอบย้อนกลับไปยังหลักฐานและกระบวนการได้อย่างชัดเจน

เมื่อระบบมีส่วนสรุปผลอัตโนมัติ Provenance ยิ่งสำคัญขึ้น เพราะคำตอบภาษาไทยที่ผู้ใช้อ่านควรสามารถขยายดูได้ว่าข้อความใดมาจาก Claim ใดและ Claim นั้นอ้างรายงานใด หากระบบสรุปว่า “โครงการ

P001 ลดเวลาตอบสนอง” แต่ไม่มี Claim และ Report รองรับ ข้อความนั้นควรถูกมองว่าเป็นคำสรุปที่ยังไม่มีหลักฐาน ไม่ใช่ความรู้ที่ยืนยันแล้วของระบบ Governance ยังต้องกำหนด Release gate ด้วย เช่น Graph จะถูก Publish ได้เมื่อ Violation ระดับ Error เป็นศูนย์ Violation ระดับ Warning ไม่เกินจำนวนที่กำหนด Query Regression Test ผ่านทั้งหมด และ Ontology Version ตรงกับ Shape Version ที่ใช้อยู่ การกำหนด gate แบบนี้ทำให้ระบบมีวินัย แต่ยังยืดหยุ่นพอที่จะยอมรับ Warning บางอย่างที่ไม่กระทบการใช้งานหลัก

ตัวอย่าง Policy อย่างง่ายคือ

```
publish graph if:
error violations = 0
critical queries pass
ontology version matches shape version
provenance graph exists for all imported sources
```

Policy นี้ไม่จำเป็นต้องซับซ้อน แต่ต้องชัดเจนและตรวจได้ หากทีมไม่รู้ว่า Graph ชุดหนึ่ง Publish ด้วยเงื่อนไขใด ความน่าเชื่อถือของระบบจะลดลง เพราะไม่มีใครบอกได้ว่าข้อมูลผ่านเกณฑ์ใดมาก่อนที่จะถูกนำไปใช้ สุดท้าย Governance ควรทำให้คนทำงานร่วมกันได้ง่ายขึ้น ไม่ใช่สร้างเอกสารที่ไม่มีใครใช้ ผู้เชี่ยวชาญโดเมนควรอ่าน Label, comment และ Validation message ได้ ผู้พัฒนาควรเห็น Shape และ Query test ได้ นักวิจัยควรเห็น Provenance และ Release history ได้ หาก Artifact เหล่านี้ถูกจัดการดี Knowledge Graph จะกลายเป็นระบบความรู้ที่ทั้งมนุษย์และเครื่องร่วมกันดูแลได้

9.11 การจัดการการเปลี่ยนแปลงของ Ontology, Shape และ Query

กรณีศึกษาก่อนหน้าอธิบายการ Publish ข้อมูลหนึ่งรุ่น แต่ระบบจริงต้องรับมือกับการเปลี่ยน Vocabulary และ Requirement อย่างต่อเนื่อง หัวข้อนี้จึงกำหนดผลกระทบระหว่าง Ontology, Shape, Mapping และ Query พร้อม Test ที่ควรรันเมื่อ Artifact หนึ่งเปลี่ยน

ระบบ Knowledge Graph ที่ใช้งานจริงจะเปลี่ยนอยู่เสมอ โดเมนมีคำใหม่ ข้อมูลต้นทางมี field ใหม่ ผู้ใช้ถามคำถามใหม่ และ Application ต้องการผลลัพธ์ใหม่ ปัญหาไม่ได้อยู่ที่การเปลี่ยนแปลง แต่อยู่ที่การเปลี่ยนโดยไม่มีร่องรอย หาก Ontology เปลี่ยนแต่ Shape ไม่เปลี่ยน ระบบอาจ validate ด้วยเงื่อนไขเก่า หาก Query เปลี่ยนแต่ Mapping ไม่เปลี่ยน Query อาจคืนผลลัพธ์ว่าง หาก Mapping เปลี่ยนแต่ Provenance ไม่ถูกบันทึก ทีมจะไม่รู้ว่าข้อมูลชุดใหม่เกิดจาก Rule ใด ดังนั้น Governance ควรมอง Artifact เหล่านี้เป็นชุดที่ต้องเผยแพร่รุ่นร่วมกัน เช่น Ontology Version 1.2.0 ควรมี Shape Version ที่เข้ากันได้ Query Catalog Version ที่ทดสอบกับ Ontology Version นั้น และ Mapping Version ที่สร้าง RDF ตาม Vocabulary เดียวกัน หากมีการเปลี่ยนแปลงที่ breaking change เช่นเปลี่ยนชื่อ Property หรือลบ Class ควรระบุใน Changelog อย่างชัดเจน

ตัวอย่าง breaking change คือเปลี่ยน **ex:hasOwner** เป็น **ex:responsibleOrg** หาก Ontology เปลี่ยนชื่อ Property แต่ Mapping ยังสร้าง **ex:hasOwner** Query ใหม่ที่ใช้ **ex:responsibleOrg** จะไม่พบข้อมูล ในทางกลับกัน หาก Mapping เปลี่ยนแล้วแต่ Query เก่ายังใช้ **ex:hasOwner** Dashboard อาจแสดงผลว่าง ทั้งที่ข้อมูลยังมีอยู่ ปัญหานี้ไม่ใช่ปัญหาเชิงตรรกะ แต่เป็นปัญหา Governance ของ Artifact วิธีลดปัญหาคือมี deprecation Policy เช่น Property เก่าอาจถูกประกาศว่า deprecated ในหนึ่ง Release ก่อนลบจริง

พร้อม Mapping ชั่วคราวให้ Query เก่ายังทำงานได้ หรือมี SPARQL Query สำหรับแปลงข้อมูลเก่าไปยัง Property ใหม่ แนวทางนี้ช่วยให้ระบบเปลี่ยนอย่างค่อยเป็นค่อยไป และลดความเสี่ยงที่ Application ปลายทางจะพังโดยไม่ทันรู้ตัว

ในกรณี SHACL Shape การเปลี่ยน **sh:minCount** จาก 0 เป็น 1 อาจทำให้ข้อมูลจำนวนมากไม่ผ่าน Validation ทันที ดังนั้นควรทดสอบ Shape ใหม่กับ Staging Graph ก่อน และดู Violation Report ว่าเพิ่มขึ้นเท่าไร หาก Violation เพิ่มขึ้นมาก ต้องตัดสินใจว่า Shape เข้มเกินไปหรือข้อมูลต้องถูกเติมจริง การตัดสินใจนี้ควรมีผู้เชี่ยวชาญโดเมนร่วมด้วย ไม่ใช่ให้ผู้พัฒนาระบบตัดสินฝ่ายเดียว สำหรับ Query Catalog การเปลี่ยน Query ควรมี Test Case เสมอ หาก Query หนึ่งใช้ในรายงานสำคัญ ควรมี Expected Result กับ Graph ตัวอย่าง และควรมีคำอธิบายว่าผลลัพธ์แต่ละ column หมายถึงอะไร Query ที่ไม่มีคำอธิบายอาจยังรันได้ แต่เมื่อเวลาผ่านไปทีมจะไม่รู้ว่าทำไม Query นั้นถูกเขียนแบบนั้น และจะไม่กล้าแก้แม้ Ontology เปลี่ยนไปแล้ว

ส่วนประกอบที่สร้าง Query หรือรายงานโดยอัตโนมัติก็ต้องอยู่ภายใต้การควบคุมเดียวกัน เพราะ Template อาจอ้าง Property เดิม หาก Ontology เปลี่ยนแต่ Template ไม่เปลี่ยน ระบบอาจสร้าง Query ผิดหรือเรียก Property ที่ไม่มีอยู่จริง ดังนั้น Semantic Governance ต้องครอบคลุม Template, Retrieval Plan และ Policy ของการแสดงผลด้วย กล่าวโดยสรุป Governance ไม่ใช่การหยุดการเปลี่ยนแปลง แต่เป็นการทำให้การเปลี่ยนแปลงมีหลักฐาน มีผลกระทบที่ประเมินได้ และมีทางย้อนกลับหากเกิดปัญหา หนังสือเล่มนี้จึงวาง SHACL และ Provenance ไว้คู่กับ Ontology Engineering เพราะในระบบจริง ความหมายที่ดีต้องมาพร้อมกระบวนการดูแลความหมายด้วย

9.12 Validation Message ที่ผู้ใช้เข้าใจได้

การตรวจพบ Violation ยังไม่ช่วยให้ข้อมูลดีขึ้นหากข้อความรายงานมีแต่ Technical Path ที่ผู้รับผิดชอบอ่านไม่เข้าใจ หัวข้อนี้จึงเชื่อม SHACL Result เข้ากับภาษาที่ชี้สาเหตุ แนวทางแก้ และ Source Record ได้อย่างเหมาะสม

SHACL Shape ที่ดีไม่ควรสร้างรายงานที่มีแต่ path และ Node ที่ผู้ใช้ทั่วไปอ่านไม่ออก หากระบบต้องให้ผู้เชี่ยวชาญโดเมนหรือ data steward แก้ข้อมูล Validation message ควรถูกเขียนให้เข้าใจได้ เช่นแทนที่จะบอกเพียงว่า Violation ที่ path **ex:responsibleOrg** ควรมีข้อความว่า “โครงการนี้ยังไม่มีหน่วยงานรับผิดชอบหลัก” และควรบอกด้วยว่าต้องแก้ที่ field ใดหรือ Source ใด ตัวอย่าง Shape สามารถใส่ message ได้ดังนี้

```
ex:ProjectShape a sh:NodeShape ;
 sh:targetClass ex:Project ;
 sh:property [
 sh:path ex:responsibleOrg ;
 sh:minCount 1 ;
 sh:message "โครงการต้องมีหน่วยงานรับผิดชอบหลัก"@th ;
 ] .
```

ข้อความภาษาไทยใน Validation Report ช่วยให้ผู้เชี่ยวชาญโดเมนเข้ามามีส่วนร่วมได้มากขึ้น เพราะไม่จำเป็นต้องอ่าน RDF หรือ SHACL ทั้งหมดจึงจะรู้ว่าข้อมูลผิดตรงไหน นี่เป็นตัวอย่างของการทำ Semantic

Web ให้ทำงานร่วมกับมนุษย์ ไม่ใช่ทำงานกับเครื่องเท่านั้น ในระบบที่มีหลายภาษา Shape อาจมี message ทั้งภาษาไทยและอังกฤษ เพื่อให้ผู้ใช้งานภาษาไทยเข้าใจผลการตรวจสอบได้สะดวก และยังรองรับการทำงานร่วมกับผู้ใช้ภาษาอื่น เช่นข้อความภาษาอังกฤษอาจใช้ใน log หรือ issue tracker ส่วนข้อความภาษาไทยใช้ในหน้าจอแก้ข้อมูล การมีสองภาษาไม่ใช่เรื่องตกแต่ง แต่เป็นส่วนหนึ่งของ Governance ที่ทำให้คนหลายกลุ่มเข้าใจ Rule เดียวกัน

Validation Report ควรจัดลำดับ Severity ด้วย เช่น Error, Warning และ Info Error คือข้อมูลที่ห้าม Publish เพราะกระทบความถูกต้องหลัก Warning คือข้อมูลที่ควรแก้แต่ยังอาจใช้งานได้บางบริบท Info คือข้อสังเกตหรือคำแนะนำ การแยก Severity ทำให้ทีมไม่ติดอยู่กับ Violation จำนวนมากโดยไม่รู้ว่าข้อใดสำคัญที่สุด ตัวอย่างเช่น Outcome Claim ที่ไม่มี Report รองรับควรเป็น Error เพราะกระทบความน่าเชื่อถือโดยตรง ส่วน Project ที่ไม่มี Label ภาษาอังกฤษอาจเป็น Warning หากระบบหลักใช้ภาษาไทย และ Technology ที่ยังไม่มีคำอธิบายอาจเป็น Info เพื่อเตือนให้ปรับ Documentation ในอนาคต การจัด Severity เช่นนี้ช่วยให้ Governance ไม่แข็งทื่อเกินไป

สุดท้าย Validation Report ควรเชื่อมกับ Workflow การแก้ไข เช่น Violation หนึ่งควรบอก Source record, Mapping Rule, Shape Version และผู้รับผิดชอบ หาก Report บอกแค่ว่า Node ใดผิด แต่ไม่บอกว่าต้องแก้ที่ไหน คนดูแลข้อมูลจะเสียเวลามาก การออกแบบ Validation message จึงเป็นส่วนหนึ่งของ user experience ของ Knowledge Graph

9.13 Severity และการตัดสินใจ Publish

ข้อความที่ชัดเจนช่วยอธิบาย Violation แต่ Pipeline ยังต้องตัดสินใจว่า Violation ระดับใดควรหยุดการ Publish Severity จึงเป็นภาษากลางระหว่าง Rule ทางเทคนิค ความเสี่ยงของข้อมูล และ Policy ขององค์กร

ใน SHACL ระดับมาตรฐานเขียนด้วย **sh:Violation**, **sh:Warning** และ **sh:Info** ส่วนคำว่า Error ในตัวอย่างนโยบายของบทนี้ใช้เรียกระดับที่ระบบกำหนดให้หยุดการเผยแพร่ ไม่ใช่ชื่อคำศัพท์มาตรฐาน **sh:Error** การตัดสินใจเผยแพร่ตามนโยบายต้องแยกจากค่า **sh:conforms** ในรายงาน SHACL ซึ่งเป็นเท็จเมื่อมี Validation Result แม้ผลนั้นจะมีระดับ Warning หรือ Info ก็ตาม (World Wide Web Consortium, 2017)

เมื่อระบบมี Validation Report จำนวนมาก การตัดสินใจว่าจะ Publish Graph หรือไม่ควรอาศัย Severity ที่กำหนดไว้ล่วงหน้า ไม่ควรตัดสินจากจำนวน Violation รวมเพียงอย่างเดียว เพราะ Violation หนึ่งรายการอาจร้ายแรงกว่า Warning หลายสิบรายการ เช่น Outcome Claim ที่ไม่มี Report รองรับหนึ่งรายการอาจกระทบความน่าเชื่อถือมากกว่า Label ภาษาอังกฤษที่ขาดหลายรายการ ตัวอย่าง Policy อาจกำหนดว่า Error ทุกประเภทต้องเป็นศูนย์ก่อน Publish ส่วน Warning สามารถมีได้หากมีผู้รับผิดชอบและมีแผนแก้ ส่วน Info ใช้เพื่อปรับปรุง Documentation ในรอบถัดไป Policy นี้ไม่บังคับให้ Warning ทุกประเภทหยุดการ Publish แต่ยังป้องกันข้อมูลผิดที่กระทบคำตอบไม่ให้เข้าสู่ระบบ

Severity ควรถูกกำหนดจากผลกระทบต่อคำตอบของระบบ หาก Violation ทำให้คำตอบอ้างหลักฐานไม่ได้ ควรเป็น Error หาก Violation ทำให้คำตอบแสดงผลไม่สมบูรณ์แต่ยังไม่ผิด ควรเป็น Warning และหาก Violation เป็นเพียงคำแนะนำเรื่องรูปแบบหรือเอกสารของ Vocabulary อาจเป็น Info การกำหนดเช่นนี้ทำให้ Validation เชื่อมกับ Use Case จริง ไม่ใช่ Rule ที่ไม่มีบริบทการใช้งาน Policy นี้ใช้เป็นเงื่อนไข

ของ Application ได้ เช่นระบบแสดงผลได้เฉพาะจาก Graph ที่ไม่มี Error และหาก Evidence มี Warning ต้องแจ้งผู้ใช้ว่าข้อมูลบางส่วนยังไม่สมบูรณ์ แนวทางนี้ป้องกันไม่ให้ส่วนแสดงผลสร้างความมั่นใจเกินหลักฐาน และทำให้ผู้ใช้เห็นระดับความน่าเชื่อถือของคำตอบ

ดังนั้น Severity จึงมีผลต่อการตัดสินใจของระบบและเป็นภาษากลางระหว่าง data engineer, Ontology engineer, Domain expert และ Application ผู้ใช้แต่ละกลุ่มอาจไม่อ่าน RDF เหมือนกัน แต่ทุกกลุ่มเข้าใจได้ว่าข้อมูลระดับ Error ไม่ควรถูกใช้ตอบคำถามสำคัญ เมื่อนำแนวคิดของบทนี้ไปใช้กับ Pipeline จากบทก่อน SHACL จะตรวจว่าข้อมูลที่สร้างขึ้นพร้อมใช้หรือไม่ ส่วน Provenance จะระบุหลักฐานและที่มาของข้อเท็จจริงที่ระบบเก็บ สองส่วนนี้จึงต้องอยู่ในกระบวนการก่อนนำ Graph ไปใช้สนับสนุนการตัดสินใจ

9.14 จาก Validation Policy สู่การกำกับข้อมูลจริง

เมื่อมี Severity และผู้รับผิดชอบแล้ว การ Publish Graph ควรถูกบันทึกเป็นการตัดสินใจที่ตรวจย้อนกลับได้ หัวข้อนี้จึงรวม Shape, Report, Provenance, Approval และ Version ให้เป็นวงจร Semantic Governance หนึ่งรอบ

เมื่อนำหลักจากส่วนก่อนมารวมกัน การ Publish Graph หนึ่งรุ่นควรเป็นการตัดสินใจที่มี record ไม่ใช่ผลข้างเคียงของการรันโปรแกรม Record นั้นต้องระบุ Graph และ Shape Version ที่ใช้ ผล Validation ระดับ Error/Warning แหล่งข้อมูลหรือ Activity ที่สร้าง Graph และผู้รับผิดชอบการตัดสินใจ เงื่อนไขเหล่านี้มีขอบเขตเฉพาะ Workflow เช่น Ontology อาจยอมรับ Project ที่ยังไม่ทราบ owner ภายใต้ Open World แต่ระบบเผยแพร่รายงานอาจปฏิเสธ Project เดียวกันด้วย SHACL เพราะคำตอบของ Application จำเป็นต้องแสดงองค์กรรับผิดชอบ ความต่างนี้ต้องปรากฏในคำอธิบาย Policy เพื่อไม่ให้ Validation Rule ถูกอ่านเป็นความจริงสากลของโดเมน

เมื่อไม่ผ่าน gate รายงานต้องระบุต้นเหตุให้ผู้ดูแลตรวจได้ โดยเชื่อม focus Node และ path เข้ากับ Source record, Mapping run และ Shape Version ข้อความสำหรับผู้ใช้โดเมนควรบอกสิ่งที่ขาดหรือค่าที่ผิด ส่วน Constraint component เชิงเทคนิคเก็บไว้สำหรับนักพัฒนา ในทำนองเดียวกัน Provenance ต้องละเอียดตามผลกระทบของ Claim: Label ทั่วไปอาจต้องการเพียง Source file แต่ผลลัพธ์โครงการที่ใช้ตัดสินใจต้องตามกลับไปยังรายงาน ตำแหน่ง กระบวนการ ผู้ตรวจ และ Graph Release ได้ Named Graph ช่วยแบ่งบริบทเหล่านี้ แต่ไม่แทน Policy เรื่องสิทธิ์เขียน สถานะเผยแพร่ หรือความหมายของแต่ละ Graph

วงจร Governance จึงมีหน่วยตัดสินใจที่ชัดเจน: ข้อเสนอเปลี่ยน Ontology หรือ Shape ระบุเหตุผล ตัวอย่างข้อมูล Query ที่ได้รับผล และแผน Migration จากนั้นจึง Review ออก Version และรัน Regression Test ตัวชี้วัด เช่น Violation rate, Provenance coverage และ Query failure rate มีค่าก็ต่อเมื่อผูกกับ threshold และ owner ที่ต้องดำเนินการ SHACL บอกว่าข้อมูลพร้อมตาม Contract หรือไม่ Provenance บอกว่าข้อมูลเกิดขึ้นอย่างไร และ Governance ทำให้การเปลี่ยน Contract กับข้อมูลถูกตรวจและย้อนกลับได้ ทั้งสามส่วนจึงประกอบเป็นกลไกเดียวกันโดยไม่ต้องกล่าวหลักปฏิบัติชุดเดิมซ้ำในทุกขั้นของ Pipeline

9.15 สรุป

บทนี้เริ่มจากคำถามว่า Graph เชื่อถือได้เพียงใด แล้วค่อยเพิ่ม Validation, Provenance, Metric, Severity และ Governance แต่ละส่วนตอบคำถามคนละด้านและต้องทำงานร่วมกันเพื่อให้การ Publish กับการเปลี่ยนแปลงมีหลักฐานรองรับ

SHACL, Provenance และ Semantic Governance ทำให้ Knowledge Graph มีความรับผิดชอบมากขึ้น SHACL บอกว่าข้อมูลผ่านเงื่อนไขของระบบหรือไม่ Provenance บอกว่าข้อมูลมาจากไหน และ Governance ทำให้ Ontology กับข้อมูลเปลี่ยนแปลงอย่างควบคุมได้ เมื่อเข้าใจหน้าที่เหล่านี้แล้ว บทสุดท้ายจะนำมาพิจารณาร่วมกันในการใช้ Semantic Web และ Ontology กับแบบจำลองภาษาขนาดใหญ่หรือ LLM

เอกสารอ้างอิง

Boneva, I., Labra Gayo, J. E., and Prud'hommeaux, E. G. (2014). Semantics and validation of shapes schemas for RDF. *arXiv*.

Carroll, J. J., Bizer, C., Hayes, P., and Stickler, P. (2005). Named graphs, provenance and trust. In *Proceedings of the 14th International Conference on World Wide Web*, pages 613–622. ACM.

Hogan, A., Blomqvist, E., Cochez, M., d'Amato, C., de Melo, G., Gutierrez, C., Kirrane, S., Gayo, J. E. L., Navigli, R., Neumaier, S., Ngomo, A.-C. N., Polleres, A., Rashid, S. M., Rula, A., Schmelzeisen, L., Sequeda, J., Staab, S., and Zimmermann, A. (2022). Knowledge graphs. *ACM Computing Surveys*, 54(4):1–37.

Labra Gayo, J. E., Prud'hommeaux, E., Boneva, I., and Kontokostas, D. (2017). *Validating RDF Data*. Morgan & Claypool.

Paulheim, H. (2017). Knowledge graph refinement: A survey of approaches and evaluation methods. *Semantic Web*, 8(3):489–508.

World Wide Web Consortium (2013). PROV-O: The PROV ontology. W3C Recommendation. Accessed June 27, 2026, from **https://www.w3.org/TR/prov-o/**.

World Wide Web Consortium (2017). Shapes constraint language (SHACL). W3C Recommendation. Accessed July 3, 2026, from **https://www.w3.org/TR/shacl/**.

Zaveri, A., Rula, A., Maurino, A., Pietrobon, R., Lehmann, J., and Auer, S. (2016). Quality assessment for linked data: A survey. *Semantic Web*, 7(1):63–93.

10

บทส่งท้าย: Semantic Web และระบบความรู้ในยุค LLM

บทก่อนหน้าอธิบาย Semantic Governance ซึ่งเป็นชั้นที่ทำให้ Knowledge Graph ตรวจสอบและดูแลการเปลี่ยนแปลงได้ บทส่งท้ายนี้นำแนวคิดทั้งหมดกลับมาพิจารณาร่วมกันในบริบทที่แบบจำลองภาษาขนาดใหญ่ หรือ Large Language Model (LLM) ถูกใช้เป็นส่วนติดต่อของระบบสารสนเทศมากขึ้น ประเด็นหลักไม่ใช่การเปรียบเทียบประสิทธิภาพของโมเดล แต่เป็นคำถามว่า Semantic Web ควรรับผิดชอบส่วนใด และ LLM ควรช่วยงานส่วนใดเมื่อระบบต้องตอบด้วยภาษาธรรมชาติพร้อมหลักฐานที่ตรวจย้อนกลับได้

เนื้อหาในบทนี้จึงนำเสนอแนวคิดและ Architecture จากประสบการณ์วิจัยของผู้เขียน โดยไม่ใช้ผลการทดลองเชิงตัวเลขเพื่อสรุปว่าแนวทางหนึ่งเหนือกว่าวิธีอื่น จุดมุ่งหมายคือแสดงลำดับการคิดตั้งแต่ Ontology Indexing ไปจนถึง Prompt-to-SPARQL, Grounding และ Auditability เพื่อให้ผู้อ่านพิจารณาความเหมาะสมของแนวทางดังกล่าวกับบริบทของตนเอง

10.1 เส้นทางที่หนังสือเล่มนี้วางไว้

ก่อนพิจารณาบทบาทของ LLM ควรย้อนทบทวนว่าแต่ละมาตรฐานในเล่มนี้เพิ่มสิ่งใดให้ระบบความรู้ การทบทวนตามลำดับช่วยป้องกันไม่ให้ Hybrid Architecture กลายเป็นเพียงการนำ LLM ไปวางหน้า Graph โดยไม่มีขอบเขตความหมายหรือกระบวนการตรวจสอบรองรับ

เส้นทางของหนังสือเริ่มจากแนวคิดว่าข้อมูลสามารถถูกมองเป็น Graph ของ Statement ได้ RDF ทำให้ Statement หนึ่งถูกเขียนเป็น Triple ที่มี Subject, Predicate และ Object ส่วน RDFS ทำให้คำศัพท์มี Class Hierarchy, Domain และ Range จากนั้น OWL เพิ่ม Axiom เชิงตรรกะที่ Reasoner ใช้อนุมานหรือตรวจ Consistency ได้ SPARQL ทำให้ Graph ถูกถามด้วย Pattern ที่ชัดเจน และ SHACL ทำให้ข้อมูลถูกตรวจตามเงื่อนไขของ Workflow

เมื่อมาตรฐานเหล่านี้ถูกใช้ร่วมกัน ลำดับของงานอาจสรุปได้ดังนี้

Identity → *Vocabulary* → *Axiom* → *Mapping* → *Validation* → *Evidence* → *Governance*

ลำดับดังกล่าวเป็นกรอบคิด ไม่ใช่สูตรคำนวณ โดยแต่ละองค์ประกอบมีหน้าที่ต่างกัน

- **Identity** ทำให้ Resource สำคัญมีตัวระบุที่อ้างซ้ำและเชื่อมข้ามแหล่งข้อมูลได้
- **Vocabulary** กำหนด Class และ Property ที่ใช้เป็นภาษาร่วมของโดเมน

- **Axiom** เพิ่มความหมายเชิงตรรกะเมื่อคำถามของระบบต้องอาศัย Reasoning
- **Mapping** นำข้อมูลจริงเข้าสู่ RDF Graph ตาม Vocabulary และ IRI Policy
- **Validation** ตรวจว่าข้อมูลพร้อมใช้ตามเงื่อนไขของ Workflow หรือไม่
- **Evidence** เชื่อม Claim และผลของ Query กลับไปยัง Source
- **Governance** ดูแล Version และการเปลี่ยนแปลงของ Ontology, Mapping, Shape และ Query

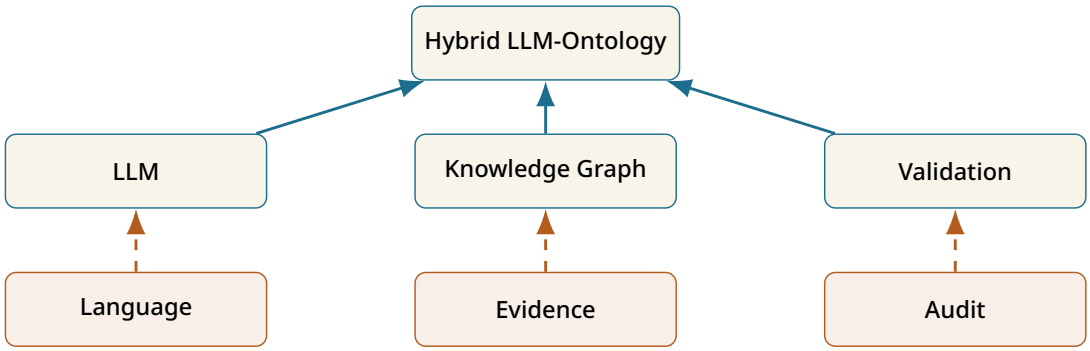
กรอบนี้ชี้ว่า Semantic Web ไม่จำเป็นต้องเริ่มจาก Ontology ขนาดใหญ่หรือ Reasoner ที่ซับซ้อน การเริ่มจาก Graph ตัวอย่างและคำถามที่ตรวจผลได้ช่วยให้เห็นก่อนว่า Resource ใดต้องมี Identity และ Relation ใดจำเป็นต่อคำตอบ เมื่อความต้องการเพิ่มขึ้นจึงค่อยเพิ่ม RDFS, OWL, SHACL, Provenance และ Alignment ตามหน้าที่ของแต่ละส่วน

สำหรับข้อมูลภาษาไทย จุดเริ่มต้นมักเป็นเอกสาร รายงาน Spreadsheet หรือฐานข้อมูลเดิม มากกว่าระบบที่ออกแบบเพื่อ Semantic Web ตั้งแต่ต้น การเพิ่มชั้นความหมายจึงไม่จำเป็นต้องย้ายข้อมูลทุกอย่างพร้อมกัน แต่ควรเลือกคำถามที่มีประโยชน์ กำหนด Resource และ Relation ที่เกี่ยวข้อง แล้วสร้างเส้นทางกลับไปยังหลักฐานให้ตรวจสอบได้ ลำดับนี้เป็นฐานสำหรับพิจารณาว่า LLM ควรถูกวางไว้ตรงใดในระบบ

10.2 ทำไม LLM ยังต้องการ Semantic Web

เมื่อผู้ใช้ถามระบบด้วยภาษาธรรมชาติได้โดยตรง อาจเกิดคำถามว่า Ontology และ SPARQL ยังจำเป็นหรือไม่ คำตอบขึ้นอยู่กับสิ่งที่ระบบต้องรับผิดชอบ หากระบบมีหน้าที่เพียงสนทนาทั่วไป LLM อาจทำงานได้โดยไม่ต้องมี Knowledge Graph แต่หากคำตอบต้องอ้างข้อมูลขององค์กร แยก Entity ที่มีชื่อคล้ายกัน หรือชี้ Source ของ Claim ระบบยังต้องมีชั้นความหมายและหลักฐานรองรับ (Ji et al., 2023; Pan et al., 2024)

การแบ่งบทบาทระหว่าง LLM, Knowledge Graph และ Validation แสดงในภาพที่ 10.1



ภาพที่ 10.1: Hybrid LLM-Ontology: LLM เป็น interface และ Knowledge Graph เป็นฐานของหลักฐาน

ภาพที่ 10.1 วาง LLM ไว้ในฐานะส่วนติดต่อเพื่อรับคำถามและเรียบเรียงคำอธิบาย ส่วน Knowledge Graph ทำหน้าที่รักษา Identity, Relation และ Evidence ขณะที่ Validation กำหนดขอบเขตว่าข้อมูลใดพร้อมให้ระบบนำไปใช้ การแบ่งบทบาทนี้ไม่ได้ทำให้ส่วนใดแทนที่อีกส่วนหนึ่ง แต่ทำให้ข้อดีและข้อจำกัดของแต่ละเทคโนโลยีถูกจัดวางอย่างชัดเจน

LLM มีความเหมาะสมกับงานด้านภาษา เช่น

- แยก Entity, Intent และเงื่อนไขจากคำถามภาษาธรรมชาติ
- ช่วยสกัด Candidate Entity และ Relation จากเอกสาร
- ช่วยสร้างหรือเลือก Query ภายใต้ Vocabulary ที่กำหนด
- เรียบเรียงผลจาก Graph ให้เป็นคำตอบที่ผู้อ่านเข้าใจได้

ส่วน Semantic Web เหมาะกับงานที่ต้องการความหมายและการตรวจสอบ ได้แก่

- กำหนด IRI ให้ Entity เดียวกันมี Identity ร่วม แม้มีหลาย Label
- กำหนด Class, Property และ Axiom ที่ระบบยอมรับ
- ใช้ SPARQL ดึงความสัมพันธ์ตามเงื่อนไขของคำถาม
- ใช้ SHACL และ Provenance ตรวจข้อมูลและเชื่อมคำตอบกลับไปยัง Source

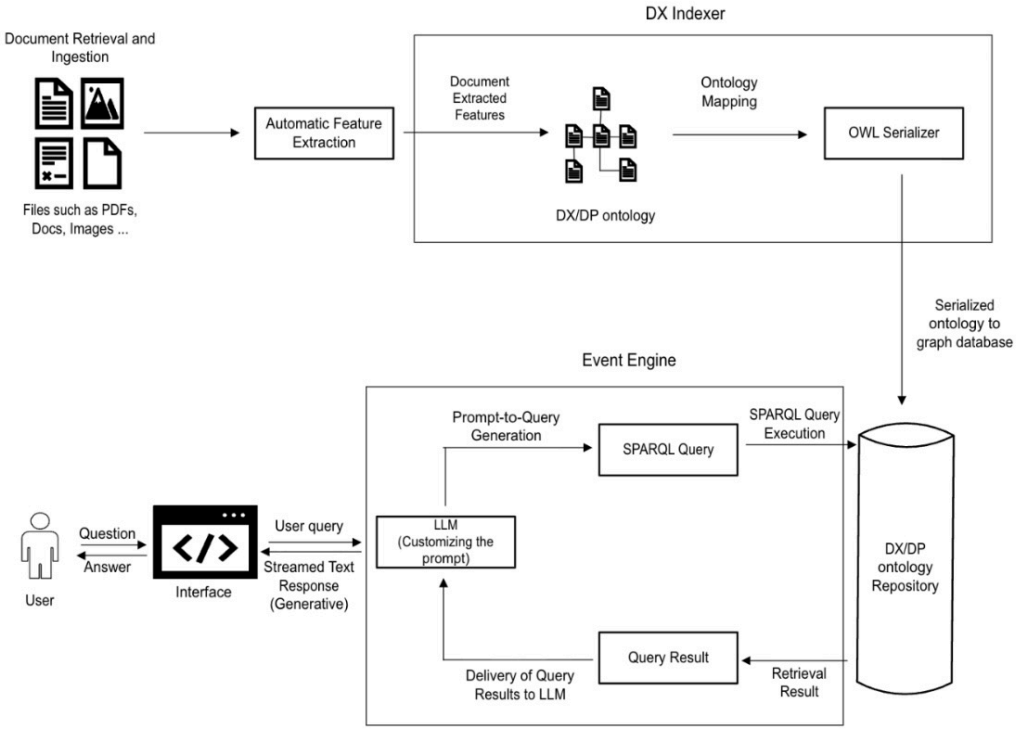
ความแตกต่างนี้มีผลชัดในภาษาไทย เพราะชื่อองค์กรหรือโครงการหนึ่งอาจมีชื่อไทย ชื่ออังกฤษ ชื่อย่อ และรูปที่ปรากฏในเอกสารหลายแบบ หากไม่มี Identity กลาง การจับคู่ด้วยข้อความเพียงอย่างเดียวอาจแยก Entity เดียวกันออกเป็นหลายรายการหรือรวม Entity คนละรายการเข้าด้วยกัน Ontology ไม่ได้แก้ Entity Resolution โดยอัตโนมัติ แต่ให้จุดอ้างอิงที่ระบบใช้บันทึกและตรวจการตัดสินใจดังกล่าวได้

ในอีกด้านหนึ่ง Graph ที่มีความหมายครบถ้วนอาจยังเข้าถึงยากหากผู้ใช้ต้องเขียน SPARQL เอง LLM จึงช่วยลดระยะห่างระหว่างภาษาของผู้ใช้กับ Vocabulary ของระบบได้ เมื่อแบ่งหน้าที่เช่นนี้ Semantic Web รักษาฐานข้อเท็จจริง ขณะที่ LLM ทำให้ฐานดังกล่าวถูกถามและอธิบายด้วยภาษาธรรมชาติได้สะดวกขึ้น

10.3 สถาปัตยกรรมระบบแบบ Hybrid LLM-Ontology

เมื่อกำหนดบทบาทของแต่ละส่วนแล้ว ขั้นต่อไปคือจัดวางองค์ประกอบให้ความผิดพลาดถูกตรวจพบก่อนกลายเป็นคำตอบ Hybrid LLM-Ontology Architecture ในหนังสือเล่มนี้เป็นแนวทางหนึ่งที่ผู้เขียนพัฒนาจากประสบการณ์ด้าน MSSN-Onto, Dx-Onto และระบบค้นคืนความรู้ ไม่ได้เสนอให้เป็น Architecture สากลสำหรับทุกโดเมน

แนวทางนี้แบ่งการทำงานออกเป็นสองช่วง ได้แก่ Ontology Indexing และ Semantic Recall สถาปัตยกรรมโดยสรุปแสดงในภาพที่ 10.2



ภาพที่ 10.2: แนวทาง Hybrid LLM-Ontology ที่แบ่ง Ontology Indexing ออกจาก Semantic Recall

ช่วงแรกคือ **Ontology Indexing** ซึ่งเปลี่ยนเอกสารหรือข้อมูลต้นทางให้เป็น Knowledge Graph ที่ผ่านการตรวจสอบ ลำดับหลักประกอบด้วย

1. **Document Ingestion** รับเอกสารและรักษา Document Identifier, ตำแหน่ง, Version และ Metadata ที่จำเป็นต่อ Provenance
2. **Schema-guided Extraction** ให้ LLM สกัด Candidate Entity, Attribute และ Relation ภายใต้ Class กับ Property ที่ Ontology กำหนด
3. **Entity Resolution** จับชื่อหลายรูปกลับไปยัง IRI เดิม หรือส่งกรณีที่ยังไม่แน่ชัดให้ทบทวน
4. **Ontology Mapping** ตรวจว่า Candidate ใช้ Class, Property, Datatype และทิศทางของ Relation ตรงตาม Ontology
5. **RDF Serialization** สร้าง Triple พร้อม Provenance ของ Source และกิจกรรมที่สร้างข้อมูล
6. **Validation and Publication** ใช้ SHACL ตรวจ Candidate Graph ก่อนเผยแพร่เป็น Graph ที่ระบบถามตอบนำไปใช้ได้

ช่วงที่สองคือ **Semantic Recall** ซึ่งรับคำถามและดึง Evidence จาก Graph ลำดับหลักประกอบด้วย

1. แยก Intent, Entity และเงื่อนไขจากคำถาม
2. Resolve ชื่อในคำถามกลับไปยัง IRI และ Vocabulary ของ Ontology
3. เลือก Query Template หรือสร้าง SPARQL ภายใต้ Schema ที่กำหนด
4. ตรวจ Syntax, Vocabulary, ชนิด Query และขอบเขต Graph ก่อน Execute

5. ดึง Evidence Bundle ที่มี Resource, Relation, Source และ Graph Version
6. ให้ LLM เรียบเรียงคำตอบจาก Evidence ที่ได้รับ พร้อมระบุข้อจำกัดเมื่อหลักฐานไม่เพียงพอ

สองช่วงนี้อาจเขียนเป็นฟังก์ชันย่อได้ว่า

$$G = \mathit{Publish}(\mathit{Validate}(\mathit{Serialize}(\mathit{Map}(\mathit{Extract}(D, O), O))))$$

$$a = \mathit{Generate}(q, \mathit{Execute}(\mathit{Validate}(\mathit{ToSPARQL}(q, O)), G))$$

ในสมการแรก D คือชุดเอกสารต้นทางและ O คือ Ontology Schema ฟังก์ชัน *Extract* สร้าง Candidate จากเอกสาร *Map* เชื่อม Candidate เข้ากับ Ontology *Serialize* สร้าง Graph และ *Validate* ตรวจ Graph ก่อน *Publish* ส่วนสมการที่สองเริ่มจากคำถาม q แล้วสร้าง SPARQL ภายใต้ Ontology ตรวจ Query และ Execute กับ Graph G ก่อนเรียบเรียงคำตอบ a จากผลลัพธ์

จุดสำคัญของกรอบนี้ไม่ได้อยู่ที่จำนวนครั้งที่เรียก LLM แต่อยู่ที่ขอบเขตซึ่งส่วน Deterministic ตรวจการทำงานของโมเดล Ontology จำกัด Vocabulary, Parser ตรวจ Query, SHACL ตรวจข้อมูล และ SPARQL Endpoint คืนผลจาก Graph หากโมเดลเสนอ Class หรือ Property ที่ไม่มีอยู่ ระบบจึงสามารถปฏิเสธหรือส่งให้ทบทวน ก่อนข้อผิดพลาดนั้นกลายเป็นคำตอบ

แนวทางดังกล่าวใกล้กับ Retrieval-Augmented Generation หรือ RAG แต่ Graph-grounded Retrieval เน้น Relation ที่นิยามไว้ใน Ontology มากกว่าความคล้ายของข้อความเพียงอย่างเดียว (Lewis et al., 2020; Fan et al., 2024) ทั้งสองแนวทางไม่จำเป็นต้องแข่งขันกัน ระบบอาจใช้ Graph เป็นแกนของ Claim ที่ต้องตรวจสอบ และใช้ Vector Retrieval เสริมบริบทจากข้อความซึ่งยังไม่ได้ถูกแทนเป็น Triple โดยต้องรวม Provenance ของ Retrieval ทั้งสองเส้นทางให้ตรวจย้อนกลับได้

10.4 จากคำถามภาษาไทยสู่ SPARQL

Architecture ก่อนหน้าแสดงตำแหน่งของ Prompt-to-SPARQL ในภาพรวม หัวข้อนี้พิจารณารายละเอียดของจุดเชื่อมระหว่างภาษาของผู้ใช้กับ Vocabulary ของ Ontology โดยเฉพาะคำถามภาษาไทยที่มีชื่อเฉพาะ คำย่อ และเงื่อนไขหลายชนิดอยู่ในประโยคเดียวกัน

ตัวอย่างคำถามว่า “โครงการใดของหน่วยงานนี้ใช้งบประมาณมากกว่าสิบล้านบาทและมีผลลัพธ์ด้านการลดเวลา” มีองค์ประกอบอย่างน้อยสี่ส่วน

- Entity ของหน่วยงานซึ่งต้อง Resolve ไปยัง IRI
- Class ของโครงการที่ระบบยอมรับ
- เงื่อนไขตัวเลขของงบประมาณพร้อม Datatype และหน่วย
- Relation ที่เชื่อม Project กับ Result หรือ Outcome Claim

การสร้าง SPARQL ที่ควบคุมได้จึงควรมีกติกาขั้นต่ำดังนี้

- ใช้เฉพาะ Class และ Property ที่มีอยู่จริงใน Ontology Schema

- ตรวจ Prefix, Syntax และทิศทางของ Relation ก่อนส่ง Query ไปยัง Endpoint
- จำกัดชนิด Query, Named Graph, เวลาในการประมวลผล และขนาดผลลัพธ์ตาม Policy
- คืนสถานะว่าไม่สามารถสร้าง Query ได้เมื่อคำถามอยู่นอกขอบเขต แทนการสร้าง Vocabulary ใหม่ขึ้นเอง

หากคำถามตรงกับ Query Catalog ที่มีอยู่ การเลือก Template ที่ผ่าน Test แล้วและเติม IRI หรือ Parameter มักควบคุมผลได้ง่ายกว่าการสร้าง Query ใหม่ทั้งหมด ส่วนคำถามที่จำเป็นต้องสร้าง Query ใหม่ควรถูกบันทึกเป็น Test Case เพื่อให้การเปลี่ยน Ontology หรือ Prompt ในภายหลังไม่ทำให้คำถามเดิมคืนผลต่างไปโดยไม่มีสัญญาณ

เมื่อ SPARQL คืนผล ระบบควรจัดผลลัพธ์เป็น Evidence Bundle ก่อนส่งให้ LLM ตัวอย่างเช่น Bundle อาจมี Project IRI, Label, Relation ที่ตอบคำถาม, Claim, Source Document และ Graph Version หากผลลัพธ์ไม่เพียงพอ คำตอบควรระบุว่า Graph ปัจจุบันยังไม่มีหลักฐาน ไม่ควรให้ LLM เติมข้อมูลจากความรู้ทั่วไปจนผู้ใช้แยกไม่ออกว่าส่วนใดมาจากระบบ

การแยก Query Generation ออกจาก Answer Generation ทำให้ตรวจตำแหน่งของความผิดได้ชัดขึ้น Query อาจผิดเพราะ Resolve Entity ไม่ถูก ใช้ Property กลับทิศ หรือขาดเงื่อนไข ส่วนคำตอบอาจผิดเพราะ LLM ละข้อมูลหรือกล่าวเกิน Evidence การประเมินจึงต้องแยกสองชั้นนี้ออกจากกัน ซึ่งเป็นประเด็นของหัวข้อถัดไป

10.5 การประเมินคำตอบที่มีหลักฐาน

เมื่อระบบตอบด้วยภาษาที่อ่านเป็นธรรมชาติ การประเมินเฉพาะความลื่นไหลของข้อความไม่เพียงพอ คำตอบที่อ่านดีอาจยังอ้าง Resource ผิดหรือเพิ่ม Claim ที่ไม่มีใน Graph หัวข้อนี้จึงเสนอกรอบประเมินเชิงหลักการ โดยไม่ใช้คะแนนจากการทดลองหนึ่งชุดเป็นข้อสรุปทั่วไป

การประเมินควรแยกอย่างน้อยสี่มิติ

1. **Retrieval Correctness** Query หรือ Retrieval Plan ดึง Resource และ Relation ที่ตรงกับคำถามหรือไม่
2. **Grounding** Claim ทุกข้อในคำตอบมี Triple, Source Span หรือ Provenance รองรับหรือไม่
3. **Explanation Quality** การเรียบเรียงรักษาความหมายของ Ontology และระบุข้อจำกัดของข้อมูลอย่างถูกต้องหรือไม่
4. **Auditability** คำตอบย้อนกลับไปยัง Query, Graph Version, Resource, Source และ Validation Status ได้หรือไม่

ตัวอย่างเช่น หากระบบตอบว่าโครงการ **P001** ใช้ SensorNet และมีผลลัพธ์ด้านการลดเวลาตอบสนอง Evidence ขั้นต่ำควรประกอบด้วย

- Triple ที่เชื่อม Project กับ Technology
- Claim ที่เชื่อม Project กับ Result ซึ่งปรากฏในคำตอบ
- Provenance ที่เชื่อม Claim กลับไปยัง Report หรือ Source

หากขาดส่วนใดส่วนหนึ่ง คำตอบควรระบุข้อจำกัดหรือไม่แสดง Claim นั้นในฐานะข้อเท็จจริงที่ยืนยันแล้ว หลักนี้ต่างจากการตรวจความคลาดเคลื่อนของคำตอบทั้งย่อหน้า เพราะคำตอบหนึ่งอาจมีหลาย Claim และต้องตรวจหลักฐานเป็นราย Claim

เมื่อพบข้อผิดพลาด การตรวจควรย้อนกลับไปยัง Stage ที่เป็นต้นเหตุ

- ตรวจ Document Ingestion ว่าไฟล์ หน้า ภาษา และ Metadata ถูกอ่านครบ
- ตรวจ Extraction ด้วย Candidate ที่ผู้เชี่ยวชาญกำกับไว้
- ตรวจ Mapping ว่า IRI, Class, Property และ Datatype ตรงกับ Ontology
- ตรวจ SHACL และ Duplicate Detection ก่อนใช้ Graph ตอบคำถาม
- ตรวจ Prompt-to-SPARQL ด้วย Query และ Expected Result บน Graph ควบคุม
- ตรวจคำตอบระดับ Claim ว่าไม่กล่าวเกิน Evidence Bundle

การแยก Stage ช่วยป้องกันการแก้ปัญหาผิดจุด การเพิ่ม Prompt ไม่สามารถชดเชยเอกสารที่ไม่ถูกนำเข้า และการเพิ่ม Axiom ไม่สามารถแก้ Query ที่ใช้ Property ผิดได้ เมื่อกรณีผิดพลาดถูกพบ ควรนำกลับมาเป็น Regression Test พร้อม Version ของ Ontology, Mapping, Shape และ Query Catalog ที่เกี่ยวข้อง

Auditability จึงเป็นเกณฑ์ที่เชื่อมการประเมินทั้งหมด คำตอบหนึ่งไม่จำเป็นต้องแสดงรายละเอียดทางเทคนิคทุกอย่างแก่ผู้ใช้ทันที แต่ระบบควรสร้าง Evidence View ได้เมื่อร้องขอ และควรบอกได้ว่าคำตอบใช้ Query ใดกับ Graph Version ใด การประเมินในลักษณะนี้ให้ความสำคัญกับความรับผิดชอบของระบบมากกว่าการหาคะแนนเฉลี่ยเพียงค่าเดียว

10.6 บทเรียนที่เชื่อมโยงกันเมื่อมองทั้งเล่ม

กรอบการประเมินก่อนหน้าทำให้เห็นว่าแนวคิดในแต่ละบทไม่ได้จบลงเมื่อสร้างไฟล์มาตรฐานสำเร็จ หัวข้อนี้จึงรวบรวมบทเรียนของทั้งเล่มในฐานะชุดการตัดสินใจที่เชื่อมกัน ตั้งแต่การกำหนด Identity ไปจนถึงคำตอบภาษาธรรมชาติที่ตรวจสอบได้

บทเรียนสำคัญสรุปได้เป็นสี่ข้อ

- **มาตรฐานแต่ละส่วนมีหน้าที่ต่างกัน** Semantic Web ไม่ใช่รายการเทคโนโลยีที่นำมาวางต่อกันโดยอัตโนมัติ การเลือกใช้ RDF, RDFS, OWL, SPARQL, SHACL, Provenance หรือ Alignment ควรเริ่มจากคำถามและความเสี่ยงของระบบ ไม่ใช่จากความต้องการทำให้ Architecture มีองค์ประกอบครบตามชื่อมาตรฐาน
- **คุณค่าของ Ontology อยู่ที่การตัดสินใจเชิงความหมาย** Ontology ไม่ควรถูกประเมินจากจำนวน Class, Property หรือ Axiom MSSN-Onto และ Dx-Onto แสดงให้เห็นว่าการตัดสินใจออกแบบเกิดจากช่องว่างของโดเมนและข้อมูลที่มีอยู่ Class หนึ่งมีคุณค่าเมื่อช่วยแยกความหมายที่ระบบต้องใช้ส่วน Property หนึ่งมีคุณค่าเมื่อสร้างเส้นทางไปยังคำตอบหรือ Evidence ที่ต้องการ

- **Code ทำหน้าที่เปิดให้ตรวจเหตุผล** ตัวอย่าง Turtle ช่วยแยก Resource ออกจาก Literal ตัวอย่าง OWL ช่วยอ่านผลของ Axiom ตัวอย่าง SPARQL ช่วยติดตามเส้นทางของคำถาม และตัวอย่าง SHACL ช่วยแยก Ontology Semantics ออกจากเงื่อนไขของ Workflow หนังสือจึงไม่ได้มุ่งสอน Implementation ตาม Tool แต่ใช้ Code เป็นภาษาประกอบการอธิบายแนวคิด
- **LLM ทำให้คำถามเรื่องความหมายและหลักฐานเด่นขึ้น** หากระบบตอบได้คล่องแต่ไม่รู้ว่าคำตอบอิง Source ใด ความสะดวกจะแลกมากับความเสี่ยง ในทางกลับกัน หาก Graph มีความหมายครบแต่ผู้ใช้เข้าถึงไม่ได้ ระบบก็ไม่สร้างประโยชน์ตามที่ออกแบบไว้ Hybrid LLM-Ontology จึงเป็นแนวทางหนึ่งสำหรับเชื่อมสองด้านนี้ โดยต้องรักษาขอบเขตของโมเดลและหลักฐานให้ชัดเจน

10.7 ถ้อยคำปิดท้าย

บทเรียนจากทั้งเล่มนำกลับมาสู่แนวคิดตั้งต้นของ Semantic Web คือการทำให้สิ่งที่ระบบกล่าวถึงมี Identity และทำให้ความสัมพันธ์มีความหมายในรูปที่โปรแกรมนำไปใช้ต่อได้ เครื่องมืออาจเปลี่ยนแปลง แต่คำถามพื้นฐานยังคงเดิมว่า Resource นี้คืออะไร Relation นี้หมายถึงอะไร และ Claim นี้อ้างหลักฐานจากที่ใด

สิ่งที่ควรนำติดตัวออกจากหนังสือจึงไม่ใช่เพียงชื่อมาตรฐาน แต่เป็นลำดับการพิจารณา

- สิ่งใดควรมี IRI และสิ่งใดควรเป็น Literal
- ความสัมพันธ์ใดต้องมี Property ที่กำหนดความหมายชัดเจน
- Axiom ใดจำเป็นต่อ Reasoning และเงื่อนไขใดควรอยู่ใน SHACL
- Query ใดตอบคำถามของงานและต้องคืน Evidence อะไร
- Claim ใดต้องมี Provenance และผ่าน Governance แบบใด

เมื่อ LLM เข้ามาช่วยรับคำถามหรือเรียบเรียงคำตอบ ลำดับนี้ยังคงใช้ได้ Ontology และ Knowledge Graph ช่วยกำหนดขอบเขตของความหมาย ขณะที่ LLM ช่วยเชื่อมภาษาของผู้ใช้เข้ากับโครงสร้างดังกล่าว ระบบที่ออกแบบอย่างระมัดระวังจึงไม่ต้องเลือกระหว่างคำตอบที่อ่านง่ายกับคำตอบที่ตรวจสอบได้ แต่สามารถใช้แต่ละเทคโนโลยีในบทบาทที่เหมาะสมและเปิดเผยข้อจำกัดของข้อมูลอย่างตรงไปตรงมา

Semantic Web ในยุค LLM จึงไม่ได้มีความหมายเพราะมาตรฐานเดิมยังคงอยู่เท่านั้น แต่มีความหมายเพราะระบบความรู้ยังต้องตอบคำถามเดิมเสมอว่า สิ่งใดคืออะไร เกี่ยวข้องกันอย่างไร และรู้ได้อย่างไรว่าคำตอบนั้นมีหลักฐานรองรับ

เอกสารอ้างอิง

Ben-arlee, S. and Angsuchotmetee, C. (2022). DP-Onto: A semantic-based approach for governing digital provider information. In *2022 37th International Technical Conference on Circuits/Systems, Computers and Communications (ITC-CSCC)*, pages 1–4. IEEE.

Ben-arlee, S. and Angsuchotmetee, C. (2026). Dx-Onto: A core ontology for a semantic-based framework for managing digital transformation projects. *Applied System Innovation*, 9(7):146.

Berners-Lee, T., Hendler, J., and Lassila, O. (2001). The semantic web. *Scientific American*, 284(5):34–43.

Fan, W., Ding, Y., Ning, L., Wang, S., Li, H., Yin, D., Chua, T.-S., and Li, Q. (2024). A survey on RAG meeting LLMs: Towards retrieval-augmented large language models. In *Proceedings of the 30th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, pages 6491–6501. ACM.

Hogan, A., Blomqvist, E., Cochez, M., d'Amato, C., de Melo, G., Gutierrez, C., Kirrane, S., Gayo, J. E. L., Navigli, R., Neumaier, S., Ngomo, A.-C. N., Polleres, A., Rashid, S. M., Rula, A., Schmelzeisen, L., Sequeda, J., Staab, S., and Zimmermann, A. (2022). Knowledge graphs. *ACM Computing Surveys*, 54(4):1–37.

Ji, Z., Lee, N., Frieske, R., Yu, T., Su, D., Xu, Y., Ishii, E., Bang, Y. J., Madotto, A., and Fung, P. (2023). Survey of hallucination in natural language generation. *ACM Computing Surveys*, 55(12):1–38.

Lewis, P., Perez, E., Piktus, A., Petroni, F., Karpukhin, V., Goyal, N., Küttler, H., Lewis, M., Yih, W.-t., Rocktäschel, T., Riedel, S., and Kiela, D. (2020). Retrieval-augmented generation for knowledge-intensive NLP tasks. In *Advances in Neural Information Processing Systems*, volume 33, pages 9459–9474.

Pan, S., Luo, L., Wang, Y., Chen, C., Wang, J., and Wu, X. (2024). Unifying large language models and knowledge graphs: A roadmap. *IEEE Transactions on Knowledge and Data Engineering*, 36(7):3580–3599.

World Wide Web Consortium (2012). OWL 2 web ontology language primer. W3C Recommendation. Accessed June 11, 2026, from **https://www.w3.org/TR/owl2-primer/**.

World Wide Web Consortium (2013). SPARQL 1.1 query language. W3C Recommendation. Accessed June 24, 2026, from **https://www.w3.org/TR/sparql11-query/**.

World Wide Web Consortium (2014a). RDF 1.1 concepts and abstract syntax. W3C Recommendation. Accessed July 4, 2026, from **https://www.w3.org/TR/rdf11-concepts/**.

World Wide Web Consortium (2014b). RDF schema 1.1. W3C Recommendation. Accessed July 10, 2026, from **https://www.w3.org/TR/rdf-schema/**.

World Wide Web Consortium (2017). Shapes constraint language (SHACL). W3C Recommendation. Accessed July 3, 2026, from **https://www.w3.org/TR/shacl/**.

อภิธานศัพท์

Class	กลุ่มของสิ่งที่มีความหมายร่วมกันใน Ontology เช่นโครงการ องค์กร เทคโนโลยี หรือเหตุการณ์
Competency Question	คำถามที่ Ontology หรือ Knowledge Graph ต้องตอบได้ ใช้กำหนดว่า Class, Property และ Query ใดจำเป็นต่อระบบจริง
Constraint	เงื่อนไขที่ข้อมูลต้องผ่านในกระบวนการทำงานหนึ่ง เช่น ต้องมีหน่วยงานรับผิดชอบหรือค่าต้องเป็นตัวเลข โดยมักตรวจด้วย SHACL
Description Logic	กลุ่มตรรกะที่เป็นฐานสำคัญของ OWL ใช้อธิบาย Class, Property, Restriction และการอนุมานอย่างเป็นระบบ
DOLCE	Descriptive Ontology for Linguistic and Cognitive Engineering ซึ่งเป็น Upper Ontology สำหรับจำแนกหมวดพื้นฐาน เช่น Endurant, Perdurant และ Quality ตามความต่างของการคงเอกลักษณ์ การดำเนินไปตามเวลา และคุณภาพของสิ่งต่าง ๆ
DUL	DOLCE+DnS Ultralite ซึ่งปรับแนวคิดของ DOLCE ให้นำไปใช้กับ OWL และ Semantic Web ได้สะดวกขึ้น
Entailment	ความสัมพันธ์เชิงตรรกะที่ข้อความหนึ่งเป็นจริงในทุกแบบจำลองที่ทำให้ข้อมูลและ Axiom ที่กำหนดเป็นจริง เช่น การเป็น Instance ของ Subclass ย่อมทำให้เป็น Instance ของ Superclass ด้วย
Entity Resolution	กระบวนการระบุว่าระเบียนหรือชื่อหลายรูปหมายถึง Entity เดียวกัน เช่น ชื่อหน่วยงานภาษาไทย ชื่ออังกฤษ และชื่อย่อ
Evidence	หลักฐานที่รองรับ Claim เช่นเอกสารต้นทาง ผลของ Query หรือระเบียน Provenance ที่ตรวจย้อนกลับได้
First-Order Logic	ตรรกะอันดับหนึ่ง ใช้ Predicate พจน์ ตัวบ่งปริมาณ และการตีความ เพื่ออธิบายเงื่อนไขความจริงของข้อความ
Hybrid LLM-Ontology	แนวทางที่ใช้ LLM ช่วยงานภาษา เช่นการรับคำถามและเรียบเรียงคำตอบ ขณะที่ Ontology, RDF, SPARQL, SHACL และ Provenance รับผิดชอบความหมายและหลักฐาน
Inference	กระบวนการสรุปข้อความใหม่จากข้อมูลและกฎหรือ Axiom ที่มีอยู่ เช่น การอนุมานตาม RDFS หรือ OWL

IRI	Internationalized Resource Identifier ตัวระบุที่ใช้เรียก Resource ใน RDF Graph อย่างไม่กำกวม โดยมีรูปคล้าย URL แต่ไม่จำเป็นต้องเปิดเป็นหน้าเว็บ
Knowledge Graph	กราฟความรู้ที่เชื่อม Entity ความสัมพันธ์ ประเภทของสิ่งที่กล่าวถึง หลักฐาน และข้อมูลประกอบ เพื่อให้สืบค้น นำกลับมาใช้ และตรวจสอบได้
Large Language Model (LLM)	แบบจำลองภาษาขนาดใหญ่ที่รับและสร้างข้อความภาษาธรรมชาติ ในระบบแบบผสมควรใช้ช่วยรับคำถามและเรียบเรียงคำตอบ โดยให้กราฟและหลักฐานเป็นฐานของข้อเท็จจริง
Literal	ค่าข้อมูลใน RDF เช่นข้อความ วันที่ หรือตัวเลข ซึ่งต่างจาก IRI ที่ใช้ระบุ Resource
Mapping Specification	ข้อกำหนดที่บอกว่าเขตข้อมูลจากแหล่งต้นทางถูกแปลงเป็น RDF Class หรือ Property ใด ใช้ Datatype ใด และกำหนด IRI อย่างไร
Named Graph	กลุ่มของ Triples ที่มีชื่อกำกับ ใช้แยกบริบท แหล่งที่มา รุ่นของข้อมูล หรือสถานะของข้อมูล
Ontology	แบบจำลองคำศัพท์และสัจพจน์ของโดเมนหนึ่ง ซึ่งกำหนดว่า Entity และความสัมพันธ์ใดมีความหมายอย่างไร
Ontology Alignment	การประกาศความสัมพันธ์ระหว่างคำศัพท์หรือ Axiom ของ Ontology ต่างชุด เพื่อให้ข้อมูลจากหลายระบบทำงานร่วมกันได้
Ontology Engineering	กระบวนการวิเคราะห์ความต้องการ ออกแบบ สร้าง ทดสอบ เผยแพร่ และดูแล Ontology อย่างเป็นระบบ
Open World Assumption	สมมติฐานที่ว่าการไม่มี statement ใน Graph ไม่ได้แปลว่า statement นั้นเป็นเท็จ แต่หมายถึงระบบยังไม่มีข้อมูลยืนยัน
OWL	Web Ontology Language กลุ่มภาษาสำหรับนิยาม Ontology ที่เพิ่ม Class Expression, Restriction, Equivalence, Disjointness และ Reasoning จาก RDFS โดย OWL 1 กับ OWL 2 เป็นคนละรุ่นมาตรฐาน ส่วน OWL DL และ OWL 2 DL เป็นภาษาย่อยที่มีข้อจำกัดเพื่อรักษาคุณสมบัติของ Reasoning
Predicate	ความสัมพันธ์หรือคุณสมบัติที่เชื่อม Subject กับ Object ใน RDF และต้องระบุด้วย IRI
Prefix	ชื่อย่อที่ผูกกับส่วนต้นของ IRI เพื่อให้เขียน RDF หรือ SPARQL ได้กระชับขึ้น โดยไม่เปลี่ยน Identity ของ Resource
Property	คำศัพท์ที่ใช้แทนความสัมพันธ์หรือ Attribute ซึ่งเชื่อม Resource กับ Resource อื่นหรือกับ Literal

Provenance	ข้อมูลเกี่ยวกับที่มา กระบวนการ ผู้สร้าง เวลา หรือรุ่นของข้อมูล ใช้ตรวจสอบย้อนกลับและอธิบายความน่าเชื่อถือของข้อมูล
Query Catalog	ชุด Query สำคัญของระบบที่บันทึกชื่อ วัตถุประสงค์ ข้อมูลเข้า ผลลัพธ์ ผู้ดูแล และรุ่น เพื่อให้ทดสอบซ้ำได้
RDF	Resource Description Framework มาตรฐานสำหรับแทนข้อมูลเป็น Triple แบบ Subject–Predicate–Object
RDFS	RDF Schema ภาษาสำหรับนิยามคำศัพท์พื้นฐานของ RDF เช่น Class, Subclass, Domain และ Range
Reasoner	โปรแกรมที่ใช้ Ontology และข้อมูลเพื่ออนุมานข้อความใหม่ หรือตรวจความสอดคล้องตาม Semantics ที่กำหนด
Resource	สิ่งที่กราฟกำลังกล่าวถึง เช่น โครงการ เอกสาร องค์กร เซนเซอร์ เหตุการณ์ หรือเทคโนโลยี
Semantic Governance	วินัยและกระบวนการกำกับการเปลี่ยนแปลงของ Ontology, Mapping, Shape, Query และข้อมูล เพื่อให้ระบบความรู้ตรวจสอบและดูแลต่อได้
Semantic Web	แนวทางและชุดมาตรฐานที่ทำให้ข้อมูลบนเว็บมีตัวระบุ ความหมาย และความสัมพันธ์ในรูปที่โปรแกรมตีความและเชื่อมโยงต่อได้
SHACL	Shapes Constraint Language ภาษาสำหรับตรวจว่าข้อมูล RDF ผ่าน Shape และเงื่อนไขคุณภาพที่ระบบกำหนดหรือไม่
SPARQL	ภาษาสำหรับ Query RDF Graph ด้วย Pattern รวมทั้งการกรอง การสรุปผล และการสร้าง Graph ใหม่
Triple	statement พื้นฐานของ RDF ที่ประกอบด้วย Subject, Predicate และ Object
Triple Store	ระบบฐานข้อมูลหรือเอนจินที่เก็บ RDF และให้บริการ SPARQL Endpoint
Upper Domain Ontology	คำที่ใช้ในบทที่ 7 เพื่ออภิปราย Ontology อ้างอิงที่ช่วยเชื่อมความหมาย ทั้ง Upper Ontology ซึ่งให้หมวดพื้นฐานข้ามโดเมน เช่น DOLCE/DUL และคำศัพท์ร่วมเฉพาะโดเมน เช่น SSN/SOSA โดยสองระดับนี้มีขอบเขตต่างกัน
Validation	การตรวจว่าข้อมูลเป็นไปตามเงื่อนไขที่กำหนด เช่น Shape, Datatype และเขตข้อมูลที่ต้องมี ซึ่งต่างจากการอนุมานความหมายด้วย OWL
Versioning	การกำหนดรุ่นของ Ontology, Mapping, Shape และ Query เพื่อให้การเปลี่ยนแปลงตรวจสอบและทำซ้ำได้
Vocabulary	ชุด Class และ Property ที่ใช้เป็นภาษากลางสำหรับอธิบายโดเมน

บรรณานุกรมรวม

Allemang, D., Hendler, J., and Gandon, F. (2020). *Semantic Web for the Working Ontologist: Effective Modeling for Linked Data, RDFS, and OWL*. ACM Books, 3 edition.

Angles, R., Arenas, M., Barceló, P., Hogan, A., Reutter, J., and Vrgoč, D. (2017). Foundations of modern query languages for graph databases. *ACM Computing Surveys*, 50(5):68:1–68:40.

Angles, R. and Gutierrez, C. (2008). Survey of graph database models. *ACM Computing Surveys*, 40(1):1–39.

Angsuchotmetee, C. and Chbeir, R. (2016). A survey on complex event definition languages in multimedia sensor networks. In *Proceedings of the 8th International Conference on Management of Digital EcoSystems*, pages 99–108. ACM.

Angsuchotmetee, C., Chbeir, R., and Cardinale, Y. (2020). MSSN-Onto: An ontology-based approach for flexible event processing in multimedia sensor networks. *Future Generation Computer Systems*, 108:1140–1158.

Angsuchotmetee, C., Chbeir, R., Cardinale, Y., and Yokoyama, S. (2017). A dynamic event detection framework for multimedia sensor networks. In *2017 23rd Asia-Pacific Conference on Communications (APCC)*, pages 1–6. IEEE.

Angsuchotmetee, C., Chbeir, R., Cardinale, Y., and Yokoyama, S. (2018). A pipelining-based framework for processing events in multimedia sensor networks. In *Proceedings of the 33rd Annual ACM Symposium on Applied Computing*, pages 247–250. ACM.

Antoniou, G., Groth, P., van Harmelen, F., and Hoekstra, R. (2012). *A Semantic Web Primer*. MIT Press, 3 edition.

Apache Software Foundation (n.d.). Apache Jena documentation. Official project documentation. Accessed October 18, 2026, from **https://jena.apache.org/documentation/**.

Baader, F., Calvanese, D., McGuinness, D. L., Nardi, D., and Patel-Schneider, P. F., editors (2003). *The Description Logic Handbook: Theory, Implementation, and Applications*. Cambridge University Press.

Ben-arlee, S. and Angsuchotmetee, C. (2022). DP-Onto: A semantic-based approach for governing digital provider information. In *2022 37th International Technical Conference on Circuits/Systems, Computers and Communications (ITC-CSCC)*, pages 1–4. IEEE.

Ben-arlee, S. and Angsuchotmetee, C. (2026). Dx-Onto: A core ontology for a semantic-based framework for managing digital transformation projects. *Applied System Innovation*, 9(7):146.

Berners-Lee, T., Hendler, J., and Lassila, O. (2001). The semantic web. *Scientific American*, 284(5):34–43.

Bizer, C., Heath, T., and Berners-Lee, T. (2009). Linked data: The story so far. *International Journal on Semantic Web and Information Systems*, 5(3):1–22.

Boneva, I., Labra Gayo, J. E., and Prud'hommeaux, E. G. (2014). Semantics and validation of shapes schemas for RDF. *arXiv*.

Carroll, J. J., Bizer, C., Hayes, P., and Stickler, P. (2005). Named graphs, provenance and trust. In *Proceedings of the 14th International Conference on World Wide Web*, pages 613–622. ACM.

Compton, M., Barnaghi, P., Bermudez, L., García-Castro, R., Corcho, O., Cox, S., Graybeal, J., Hauswirth, M., Henson, C., Herzog, A., Huang, V., Janowicz, K., Kelsey, W. D., Le Phuoc, D., Lefort, L., Leggieri, M., Neuhaus, H., Nikolov, A., Page, K., Passant, A., Sheth, A., and Taylor, K. (2012). The SSN ontology of the W3C semantic sensor network incubator group. *Journal of Web Semantics*, 17:25–32.

Corcho, O., Fernández-López, M., and Gómez-Pérez, A. (2003). Methodologies, tools and languages for building ontologies. where is their meeting point? *Data & Knowledge Engineering*, 46(1):41–64.

Euzenat, J. and Shvaiko, P. (2013). *Ontology Matching*. Springer, 2 edition.

Fan, W., Ding, Y., Ning, L., Wang, S., Li, H., Yin, D., Chua, T.-S., and Li, Q. (2024). A survey on RAG meeting LLMs: Towards retrieval-augmented large language models. In *Proceedings of the 30th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, pages 6491–6501. ACM.

Gangemi, A. (2008). Norms and plans as unification criteria for social collectives. *Autonomous Agents and Multi-Agent Systems*, 17(1):70–112.

Gangemi, A. (n.d.). DOLCE+DnS Ultralite (version 3.34). Ontology Design Patterns, ontology specification. Archived June 7, 2021, by DBpedia Archivo. Accessed October 18, 2026, from **https://akswnc7.informatik.uni-leipzig.de/dstreitmatter/archivo/ontologydesignpatterns.org/ont--dul--DUL--owl/2021.06.07-182648/ont--dul--DUL--owl_type=pyLodeDoc.html.**

Gangemi, A., Guarino, N., Masolo, C., Oltramari, A., and Schneider, L. (2002). Sweetening ontologies with DOLCE. In Gómez-Pérez, A. and Benjamins, V. R., editors, *Knowledge Engineering and Knowledge Management: Ontologies and the Semantic Web*, pages 166–181. Springer.

Gruber, T. R. (1993). A translation approach to portable ontology specifications. *Knowledge Acquisition*, 5(2):199–220.

Guarino, N. (1998). Formal ontology and information systems. In *Proceedings of FOIS'98*, pages 3–15. IOS Press.

Haller, A., Janowicz, K., Cox, S. J. D., Lefrançois, M., Taylor, K., Le Phuoc, D., Lieberman, J., García-Castro, R., Atkinson, R., and Stadler, C. (2019). The modular SSN ontology: A joint W3C and OGC standard specifying the semantics of sensors, observations, sampling, and actuation. *Semantic Web*, 10(1):9–32.

Heath, T. and Bizer, C. (2011). *Linked Data: Evolving the Web into a Global Data Space*. Morgan & Claypool.

Hitzler, P., Krötzsch, M., and Rudolph, S. (2009). *Foundations of Semantic Web Technologies*. CRC Press.

Hogan, A., Blomqvist, E., Cochez, M., d'Amato, C., de Melo, G., Gutierrez, C., Kirrane, S., Gayo, J. E. L., Navigli, R., Neumaier, S., Ngomo, A.-C. N., Polleres, A., Rashid, S. M., Rula, A., Schmelzeisen, L., Sequeda, J., Staab, S., and Zimmermann, A. (2022). Knowledge graphs. *ACM Computing Surveys*, 54(4):1–37.

Horrocks, I., Patel-Schneider, P. F., and van Harmelen, F. (2003). From SHIQ and RDF to OWL: The making of a web ontology language. *Journal of Web Semantics*, 1(1):7–26.

Ji, Z., Lee, N., Frieske, R., Yu, T., Su, D., Xu, Y., Ishii, E., Bang, Y. J., Madotto, A., and Fung, P. (2023). Survey of hallucination in natural language generation. *ACM Computing Surveys*, 55(12):1–38.

Kalfoglou, Y. and Schorlemmer, M. (2003). Ontology mapping: The state of the art. *The Knowledge Engineering Review*, 18(1):1–31.

Krötzsch, M. (2012). OWL 2 profiles: An introduction to lightweight ontology languages. In Eiter, T. and Krennwallner, T., editors, *Reasoning Web. Semantic Technologies for Advanced Query Answering*, volume 7487 of *Lecture Notes in Computer Science*, pages 112–183. Springer.

Labra Gayo, J. E., Prud'hommeaux, E., Boneva, I., and Kontokostas, D. (2017). *Validating RDF Data*. Morgan & Claypool.

Lewis, P., Perez, E., Piktus, A., Petroni, F., Karpukhin, V., Goyal, N., Küttler, H., Lewis, M., Yih, W.-t., Rocktäschel, T., Riedel, S., and Kiela, D. (2020). Retrieval-augmented generation for knowledge-intensive NLP tasks. In *Advances in Neural Information Processing Systems*, volume 33, pages 9459–9474.

Masolo, C., Borgo, S., Gangemi, A., Guarino, N., and Oltramari, A. (2003). WonderWeb deliverable D18: Ontology library (final). Technical report, IST Project 2001-33052 WonderWeb.

Mishra, D. S., Agarwal, A., Swathi, B. P., and Akshay, K. C. (2022). Natural language query formalization to SPARQL for querying knowledge bases using Rasa. *Progress in Artificial Intelligence*, 11(3):193–206.

Neumann, T. and Weikum, G. (2010). The RDF-3X engine for scalable management of RDF data. *The VLDB Journal*, 19(1):91–113.

Noy, N. F. and McGuinness, D. L. (2001). Ontology development 101: A guide to creating your first ontology. Technical Report KSL-01-05, Stanford Knowledge Systems Laboratory.

Pan, S., Luo, L., Wang, Y., Chen, C., Wang, J., and Wu, X. (2024). Unifying large language models and knowledge graphs: A roadmap. *IEEE Transactions on Knowledge and Data Engineering*, 36(7):3580–3599.

Paulheim, H. (2017). Knowledge graph refinement: A survey of approaches and evaluation methods. *Semantic Web*, 8(3):489–508.

Pérez, J., Arenas, M., and Gutierrez, C. (2006). Semantics and complexity of SPARQL. In *The Semantic Web – ISWC 2006*, pages 30–43. Springer.

Pérez, J., Arenas, M., and Gutierrez, C. (2009). Semantics and complexity of SPARQL. *ACM Transactions on Database Systems*, 34(3):1–45.

RDFLib Team (n.d.-a). pySHACL: A Python validator for SHACL. Official project repository. Accessed October 18, 2026, from **https://github.com/RDFLib/pySHACL**.

RDFLib Team (n.d.-b). RDFLib: Python software for working with RDF. Official project documentation. Accessed October 18, 2026, from **https://rdflib.readthedocs.io/**.

Sequeda, J. and Lassila, O. (2021). *Designing and Building Enterprise Knowledge Graphs*. Morgan & Claypool.

Stanford Center for Biomedical Informatics Research (n.d.). Protégé: A free, open-source ontology editor. Official project documentation. Accessed October 18, 2026, from **https://protege.stanford.edu/**.

World Wide Web Consortium (2004). OWL web ontology language overview. W3C Recommendation. Accessed July 12, 2026, from **https://www.w3.org/TR/owl-features/**.

World Wide Web Consortium (2012a). OWL 2 web ontology language direct semantics. W3C Recommendation. Accessed June 10, 2026, from **https://www.w3.org/TR/owl2-direct-semantics/**.

World Wide Web Consortium (2012b). OWL 2 web ontology language document overview. W3C Recommendation. Accessed July 12, 2026, from **https://www.w3.org/TR/owl2-overview/**.

World Wide Web Consortium (2012c). OWL 2 web ontology language mapping to RDF graphs. W3C Recommendation. Accessed July 12, 2026, from **https://www.w3.org/TR/owl2-mapping-to-rdf/**.

World Wide Web Consortium (2012d). OWL 2 web ontology language new features and rationale. W3C Recommendation. Accessed July 12, 2026, from **https://www.w3.org/TR/owl2-new-features/**.

World Wide Web Consortium (2012e). OWL 2 web ontology language primer. W3C Recommendation. Accessed June 11, 2026, from **https://www.w3.org/TR/owl2-primer/**.

World Wide Web Consortium (2012f). OWL 2 web ontology language profiles. W3C Recommendation. Accessed July 5, 2026, from **https://www.w3.org/TR/owl2-profiles/**.

World Wide Web Consortium (2012g). OWL 2 web ontology language quick reference guide. W3C Recommendation. Accessed July 12, 2026, from **https://www.w3.org/TR/owl2-quick-reference/**.

World Wide Web Consortium (2012h). OWL 2 web ontology language RDF-based semantics. W3C Recommendation. Accessed July 12, 2026, from **https://www.w3.org/TR/owl2-rdf-based-semantics/**.

World Wide Web Consortium (2012i). OWL 2 web ontology language structural specification and functional-style syntax. W3C Recommendation. Accessed July 7, 2026, from **https://www.w3.org/TR/owl2-syntax/**.

World Wide Web Consortium (2012j). R2RML: RDB to RDF mapping language. W3C Recommendation. Accessed July 9, 2026, from **https://www.w3.org/TR/r2rml/**.

World Wide Web Consortium (2013a). PROV-O: The PROV ontology. W3C Recommendation. Accessed June 27, 2026, from **https://www.w3.org/TR/prov-o/**.

World Wide Web Consortium (2013b). SPARQL 1.1 overview. W3C Recommendation. Accessed June 13, 2026, from **https://www.w3.org/TR/sparql11-overview/**.

World Wide Web Consortium (2013c). SPARQL 1.1 protocol. W3C Recommendation. Accessed July 8, 2026, from **https://www.w3.org/TR/sparql11-protocol/**.

World Wide Web Consortium (2013d). SPARQL 1.1 query language. W3C Recommendation. Accessed June 24, 2026, from **https://www.w3.org/TR/sparql11-query/**.

World Wide Web Consortium (2013e). SPARQL 1.1 update. W3C Recommendation. Accessed June 22, 2026, from **https://www.w3.org/TR/sparql11-update/**.

World Wide Web Consortium (2014a). RDF 1.1 concepts and abstract syntax. W3C Recommendation. Accessed July 4, 2026, from **https://www.w3.org/TR/rdf11-concepts/**.

World Wide Web Consortium (2014b). RDF 1.1 semantics. W3C Recommendation. Accessed June 16, 2026, from **https://www.w3.org/TR/rdf11-mt/**.

World Wide Web Consortium (2014c). RDF 1.1 Turtle: Terse RDF triple language. W3C Recommendation. Accessed June 29, 2026, from **https://www.w3.org/TR/turtle/**.

World Wide Web Consortium (2014d). RDF schema 1.1. W3C Recommendation. Accessed July 10, 2026, from **https://www.w3.org/TR/rdf-schema/**.

World Wide Web Consortium (2017a). Semantic sensor network ontology. W3C Recommendation. Accessed June 18, 2026, from **https://www.w3.org/TR/vocab-ssn/**.

World Wide Web Consortium (2017b). Shapes constraint language (SHACL). W3C Recommendation. Accessed July 3, 2026, from **https://www.w3.org/TR/shacl/**.

World Wide Web Consortium (2026a). RDF 1.2 concepts and abstract data model. W3C Candidate Recommendation Snapshot. Accessed October 18, 2026, from **https://www.w3.org/TR/2026/CR-rdf12-concepts-20260407/**.

World Wide Web Consortium (2026b). SPARQL 1.2 query language. W3C Working Draft. Accessed October 18, 2026, from **https://www.w3.org/TR/2026/WD-sparql12-query-20261011/**.

Zaveri, A., Rula, A., Maurino, A., Pietrobon, R., Lehmann, J., and Auer, S. (2016). Quality assessment for linked data: A survey. *Semantic Web*, 7(1):63–93.

ดัชนีค้นคำ

คำค้นภาษาไทย

กราฟความรู้ (Knowledge Graph), 1
กราฟที่มีชื่อ (Named Graph), 60, 126
การกำกับดูแลเชิงความหมาย (Semantic Governance), 124
การตรวจสอบข้อมูล (Validation), 124
การตีความ (Interpretation), 34
การสืบค้น (Query), 55
การอนุมาน (Inference), 19
ตรรกะอันดับหนึ่ง (First-Order Logic), 34
ตัวบ่งปริมาณ (Quantifier), 34
ตัวระบุ (IRI), 14
ทรัพยากรข้อมูล (Resource), 2
แบบจำลองภาษาขนาดใหญ่ (LLM), 137
เว็บเชิงความหมาย (Semantic Web), 1
สัจพจน์ (Axiom), 29
สมมติฐานโลกเปิด (Open World Assumption), 39
หลักฐาน (Evidence), 60, 128
แหล่งที่มาของข้อมูล (Provenance), 90, 124
ออนโทโลยี (Ontology), 7

A

Aggregation, 59
Alignment Axiom, 97
Alignment Governance, 100
Alignment Relation, 95
Alignment Versioning, 100
Annotation Property, 31
Answer Evaluation, 142
Apache Jena, 120
Application Architecture, 102
Application Review, 74
Artifact, 103
ASK, 57
Audit Trail, 135
Auditability, 138
Authority List, 106
Axiom, 29

B

Basic Graph Pattern, 55
Blank Node, 14

C

Candidate, 106
Candidate Mapping, 97
Cardinality, 42
Change Management, 132
Claim, 128
Class, 15
Class Expression, 31
Compatibility, 119
Competency Question, 70, 71
Conformance, 124
Consistency, 39
CONSTRUCT, 57
COUNT, 59

D

Data Authority, 105
Data Quality Metric, 129
Data Steward, 135
Dataset, 60
Datatype Property, 31
Decision Log, 80
Deployment, 109
Description Logic, 34
Design Decision, 80
Design Rationale, 75
Digital Transformation, 88
Direct Semantics, 30
Disjointness, 41
DOLCE, 96
Domain, 15
Domain Ontology, 94
Domain Review, 74
DUL, 96
Dx-Onto, 83

E

Entailment, 19, 34
Entity Resolution, 106
Event Modeling, 84
Evidence, 60, 128
Evidence Bundle, 113, 142
Evidence Chain, 128
Evidence Modeling, 88
Evidence Retrieval, 58

Existential Restriction, 41
Expected Result, 61, 71
Extraction, 106

F
FILTER, 57
First-Order Logic, 34

G
Governance Metric, 129
Graph Integration, 23
Graph Lifecycle, 65
Graph Pattern, 55
Graph RAG, 139
Graph Transformation, 57
Grounded Answer, 142
Grounding, 138
GROUP BY, 59

H
Hallucination, 138
Hybrid LLM-Ontology, 137

I
Identity, 23, 42
Inconsistency, 41
Inference, 19
Interoperability, 1
Interpretation, 34
IRI, 14
IRI Policy, 23

J
Jena Fuseki, 120

K
Knowledge Graph, 1
    Pipeline, 103
Knowledge Graph Application, 102

L
Literal, 14
LLM, 137
Logic
    First-Order Logic, 34
Logical Review, 74

M
Mapping Review, 97
Mapping Run, 113
Mapping Specification, 102
Migration, 109
Minimal Commitment, 74
Modular Ontology, 90
Monitoring, 119
MSSN-Onto, 83
Multimedia Semantic Sensor Networks, 84

N
Named Graph, 60, 126
Namespace, 4, 14
Node Shape, 127

O
Object, 3
Object Property, 31
Ontology, 7
    Logical Semantics, 29
    Modularity, 90
    Scope, 74
Ontology Alignment, 94
Ontology Case Study, 83
Ontology Design Pattern, 76
Ontology Documentation, 75
Ontology Engineering, 70
    Competency Question, 71
Ontology Indexing, 139
Ontology Mapping, 95
Ontology Versioning, 75
Open World Assumption, 39
OPTIONAL, 57
OWL, 29, 40
    Restriction, 41
OWL 2, 30
OWL 2 EL, 44
OWL 2 QL, 44
OWL 2 RL, 44
OWL DL, 30
OWL Profile, 44
owl:sameAs, 42

P
Pattern Reuse, 76
Pipeline, 102
Predicate, 3
Predicate Design, 26
Predicate Logic, 34
Prefix, 4
Project-Centric Ontology, 88
Prompt-to-SPARQL, 141
Property, 15

Property Characteristic, 42
Property Path, 66
Property Shape, 127
Protégé, 120
PROV-O, 126
Provenance, 90, 124
    Named Graph, 126
Provenance Pattern, 128
Publish Policy, 134
Published Graph, 113
pySHACL, 120

Q
Quantifier, 34
Query, 55
Query Catalog, 58
Query Design, 66
Query Generation, 141
Query Test, 61

R
Range, 15
RDF, 3, 12
RDF Graph, 12
RDF-Based Semantics, 30
RDFLib, 120
RDFS, 12, 40
    Entailment, 19
Reasoner, 45
Regression Test, 119
Release Package, 75
Resource, 2
Restriction, 39
Retrieval-Augmented Generation, 139
Rollback, 109

S
Satisfiability, 45
Scope, 74
SELECT, 57
Semantic Governance, 124
Semantic Mapping, 106
Semantic Recall, 139
Semantic Versioning, 132
Semantic Web, 1
    LLM, 137
Sensor Ontology, 84
Severity, 134
SHACL, 40, 124
    Shape, 127
SHACL Gate, 107
Source Inventory, 105
Source Pointer, 106
Source Record, 105
SPARQL, 55
SPARQL Update, 65
Staging Graph, 113
Subclass, 15
Subject, 3

T
Thai Question Processing, 141
Tool Selection, 120
Triple, 3
Turtle, 4, 13

U
Universal Restriction, 41
Unsatisfiable Class, 45
Upper Ontology, 94
    DOLCE and DUL, 96

V
Validation, 124
Validation Function, 124
Validation Gate, 107
Validation Policy, 135
Validation Report, 107, 128
Validation Run, 113
Versioning, 109
Violation, 134
Vocabulary, 7
Vocabulary Design, 26

W
Warning, 134