รายงานวิจัยฉบับสมบูรณ์

โครงการ การกำหนดความสัมพันธ์ของไดอะแกรมภาษา UML โดยใช้ทฤษฎีโปรแกรมเชิงประกาศสำหรับ XMI/XML เป็นพื้นฐาน

(On the Relationships between UML Diagrams
Based-on XMI/XML Declarative Program Theory)

โดย เอกวิชญ์ นันทจิวรวัฒน์ และคณะ

มิถุนายน 2548

สัญญาเลขที่ PDF/31/2543

รายงานวิจัยฉบับสมบูรณ์

โครงการ การกำหนดความสัมพันธ์ของไดอะแกรมภาษา UML โดยใช้ทฤษฎีโปรแกรมเชิงประกาศสำหรับ XML/XMI เป็นพื้นฐาน

(On the Relationships between UML Diagrams
Based-on XMI/XML Declarative Program Theory)

ผศ. ดร. เอกวิชญ์ นันทจีวรวัฒน์	สถาบันเทคโนโลยีนานาชาติสิรินธร มหาวิทยาลัยธรรมศาสตร์
ศ. ดร. วิลาศ วูวงศ์	สถาบันเทคโนโลยีแห่งเอเชีย

สนับสนุนโดยสำนักงานกองทุนสนับสนุนการวิจัย

(ความเห็นในรายงานนี้เป็นของผู้วิจัย สกว.ไม่จำเป็นต้องเห็นด้วยเสมอไป)

สำนักงานกองทุนสนับสนุนการวิจัย (สกว.)
ชั้น 14 อาคาร เอส เอ็ม ที ทาวเวอร์
เลขที่ 979/17-21 ถนนพหลโยธิน แขวงสามเสนใน
เขตพญาไท กรุงเทพฯ 10400
โทร. 298-0455 โทรสาร 298-0476
Home page : http://www.trf.or.th
E-mail : trfinfo@trf.or.th

กิตติกรรมประกาศ

แนวความคิดหลักที่นำเสนอในโครงการนี้ เกิดขึ้นมาจากคำแนะนำอย่างต่อเนื่องที่หัวหน้าโครงการได้รับจาก ศาสตราจารย์ วิลาศ วูวงศ์ ซึ่งนอกจากจะทำหน้าที่เป็นอาจารย์ที่ปรึกษาสำหรับวิทยานิพนธ์ปริญญาโทและเอกของหัวหน้าโครงการในระหว่างปี พ.ศ. 2533-2540 แล้ว ยังให้ความกรุณาทำหน้าที่เป็นนักวิจัยพี่เลี้ยงของโครงการด้วย หัวหน้าโครงการได้รับการสนับสนุนในเรื่องรายละเอียดของกลไกการประมวลผลแบบสมมูล ซึ่งเป็นกลไกการประมวลผลหลักที่ใช้ในโครงการนี้ จากศาสตราจารย์ คิโยชิ อะคามะ และได้มีโอกาสทำงานร่วมกันในด้านรากฐานทางทฤษฎีสำหรับกลไกการประมวลผลแบบสมมูล ในช่วงเวลาสามปีที่ผ่านมา ซอฟท์แวร์ต้นแบบของระบบฐานความรู้ในโครงการนี้ถูกพัฒนาขึ้น โดยได้รับความช่วยเหลือจาก ชุติพร อนุตริยะ สุรภา เทียมจรัส และ หทัยชนก อุ่นผล โครงการนี้ได้รับการสนับสนุนจากสำนักงานกองทุนสนับสนุนการวิจัย (ทุนวิจัยหลังปริญญาเอก สกว.)

Abstract

Project Code : **PDF/31/2543**

Project Title : **On the Relationships between UML Diagrams Based-on XMI/XML Declarative Program Theory**

Investigator : Ekawit Nantajeewarat
Sirindhorn International Institute of Technology
Thammasat University

E-mail Address : ekawit@siit.tu.ac.th

Project Period : 1 July 2000 – 30 June 2002

Objective: To establish a foundation for representing knowledge and reasoning in the domain of UML based on XML declarative descriptions.

Methods: Graphical diagrams in a UML model are encoded as XML elements, which are regarded as facts about a specific problem instance in a knowledge base, and the general knowledge in the UML domain, such as inherent interrelationships among diagram components and implicit properties of diagrams, is represented as a set of XML definite clauses. Equivalent transformation is employed as the underlying computation mechanism for reasoning with the represented diagrams.

Results: A framework for knowledge representation and reasoning in the domain of UML, based on XML Declarative Description Theory, is proposed. To represent UML diagrams in a standard way, the XML Document Type Definition (DTD) specified by XML Metadata Interchange Format (XMI), a technology recommended lately by the Object Management Group (OMG), is employed. Representation of general rules in the UML domain using XML definite clauses is demonstrated. The framework has been applied to the representation of transformation rules for generating relational database schemas from the static parts of UML models. A prototype UML knowledge-based system under the proposed framework has been developed and tested, and satisfactory results have been obtained.

Conclusion: Since XMI/XML is becoming a standard textual representation of UML diagrams, it is expected that the presented framework has several promising applications, such as forward and reverse engineering, consistency verification of models, and automatic generation of database schemas. Integration of the proposed framework into other UML-based software modeling tools and techniques is also possible inasmuch as virtually every tool supporting UML is capable of reading and writing models in XMI format.

Keywords : UML, XML, Knowledge Representation, Knowledge-Base Systems, Declarative Descriptions, Deduction, Software Engineering

บทคัดย่อ

รหัสโครงการ : **PDF/31/2543**

ชื่อโครงการ : **การกำหนดความสัมพันธ์ของไดอะแกรมภาษา UML โดยใช้ทฤษฎีโปรแกรมเชิงประกาศสำหรับ XMI/XML เป็นพื้นฐาน**

ชื่อนักวิจัย และสถาบัน : เอกวิชญ์ นันทจีวรวัฒน์
สถาบันเทคโนโลยีนานาชาติสิรินธร มหาวิทยาลัยธรรมศาสตร์

E-mail Address : ekawit@siit.tu.ac.th

ระยะเวลาโครงการ : 1 กรกฎาคม 2543 – 30 มิถุนายน 2545

วัตถุประสงค์ : เพื่อสร้างองค์ความรู้ใหม่สำหรับการจัดเก็บความรู้ และการอนุมาน เกี่ยวกับแบบจำลองของระบบงานที่สร้างขึ้นในภาษา Unified Modeling Language (UML) โดยใช้ทฤษฎีโปรแกรมเชิงประกาศสำหรับ Extensible Markup Language (XML) เป็นพื้นฐาน

วิธีการ : ไดอะแกรม UML ประเภทต่างๆ ในแบบจำลองของระบบงาน จะถูกนำไปจัดเก็บในฐานความรู้ในลักษณะของข้อมูลในรูปแบบ XML ส่วนความรู้เกี่ยวกับความสัมพันธ์ระหว่างไดอะแกรมประเภทต่างๆ จะถูกจัดเก็บในลักษณะของกฎเกณฑ์ในรูปแบบของ Definite Clause สำหรับ XML โดยจะใช้การประมวลผลแบบสมมูลเป็นกลไกหลักในการนิรนัยและการอนุมานเกี่ยวกับเนื้อหาของฐานความรู้

ผลของโครงการ : วิธีการสำหรับการจัดเก็บความรู้ และการนิรนัย เกี่ยวกับแบบจำลองของระบบงานที่เขียนขึ้นในภาษา UML ได้ถูกเสนอขึ้นบนรากฐานของทฤษฎีโปรแกรมเชิงประกาศสำหรับ XML โดยมีการนำ XML Document Type Definition (DTD) ที่ถูกกำหนดขึ้นโดย XML Metadata Interchange Format (XMI) มาใช้เป็นมาตรฐานในการจัดเก็บไดอะแกรมต่างๆ และได้มีการแสดงอย่างชัดเจนถึงการใช้ Definite Clause สำหรับ XML ในการจัดเก็บความสัมพันธ์ระหว่างส่วนประกอบของไดอะแกรมและกฎเกณฑ์ทั่วไปในโดเมนของ UML รวมถึงมีการแสดงการนำวิธีการที่เสนอขึ้นไปประยุกต์ใช้ในการจัดเก็บกฎเกณฑ์การสร้างโครงสร้างฐานข้อมูลแบบสัมพันธ์ (Relational Database Schema) จากแบบจำลองระบบงานที่อยู่ในรูปของ UML ต้นแบบของระบบฐานความรู้ได้ถูกพัฒนาขึ้นเพื่อการทดสอบขั้นพื้นฐาน

สรุป : เป็นที่คาดหวังว่าวิธีการที่เสนอขึ้นนี้จะนำไปประยุกต์ใช้ได้ กับงานทางด้านการจัดเก็บกฎเกณฑ์การสร้างโปรแกรมประยุกต์จากแบบจำลองระบบงาน และงานทางด้านการตรวจสอบความถูกต้องสอดคล้องของแบบจำลองระบบงาน นอกจากนี้ยังมีความเป็นไปได้ที่จะนำวิธีการนี้ไปประยุกต์ใช้ร่วมกับวิธีการอื่นๆที่ใช้อยู่ในซอฟท์แวร์สำหรับช่วยการพัฒนาแบบจำลองระบบงานต่างๆ เนื่องจากซอฟท์แวร์เหล่านี้ส่วนใหญ่สามารถอ่าน และบันทึกแบบจำลองระบบงานในรูปแบบ XMI/XML

คำหลัก : UML, XML, การจัดเก็บความรู้, ระบบฐานความรู้, โปรแกรมเชิงประกาศ, การนิรนัย, วิศวกรรมซอฟท์แวร์

เนื้อหางานวิจัย

บทนำ

การสร้างแบบจำลอง (Model) ของระบบงานเป็นสิ่งจำเป็นอย่างยิ่งในการพัฒนาซอฟท์แวร์โดยเฉพาะอย่างยิ่งในการพัฒนาซอฟท์แวร์ขนาดใหญ่ แบบจำลองของระบบจะช่วยทำให้ผู้วิเคราะห์ระบบและผู้ใช้ระบบมีความเข้าใจตรงกันในระบบงานที่ต้องการ และจะทำหน้าที่เป็นข้อกำหนดรายละเอียด (Specifications) เพื่อให้ทีมเขียนโปรแกรมสร้างโปรแกรมได้อย่างถูกต้องตามความต้องการของผู้ใช้ระบบ ความสำคัญของแบบจำลองระบบงานในการพัฒนาซอฟท์แวร์เปรียบได้กับความสำคัญของพิมพ์เขียว (Blueprint) ในการก่อสร้างอาคาร ตั้งแต่ปลายปี 1997 เป็นต้นมา ภาษา Unified Modeling Language (UML) [1, 2, 3] ได้รับการรับรองจาก Object Management Group (OMG)[1] ซึ่งเป็นองค์การที่ทำหน้าที่กำหนดมาตรฐานในอุตสาหกรรมซอฟท์แวร์ ให้เป็นภาษามาตรฐานสำหรับเขียนแบบจำลองเชิงวัตถุของระบบงาน (Object-Oriented Model) ภาษาUML ประกอบด้วยไดอะแกรมเชิงภาพประเภทต่างๆ ซึ่งสามารถนำไปใช้ทั้งในการบรรยายโครงสร้างของระบบ (Static Model) และการกำหนดการทำงานร่วมกันขององค์ประกอบย่อยต่างๆ ของระบบ (Behavioral Model) ในแง่มุมต่างๆ

เนื่องจากภาษา UML ได้รับการพัฒนาขึ้นอย่างรวดเร็วโดยมีแรงผลักดันจากภาคอุตสาหกรรมซอฟท์แวร์เป็นหลัก ปัญหาสำคัญอย่างหนึ่งของภาษา UML ในปัจจุบันก็คือ การขาดรากฐานทางทฤษฎีเกี่ยวกับความหมายที่ชัดเจน (Precise Formal Semantics) และความสัมพันธ์ระหว่างกันของไดอะแกรมประเภทต่างๆ รากฐานทางทฤษฎีดังกล่าวนี้มีความสำคัญอย่างยิ่งในการวิเคราะห์และตรวจสอบความถูกต้องตรงกันของไดอะแกรมต่างๆ ในแบบจำลอง และจะทำให้ผู้ใช้ระบบ ผู้วิเคราะห์ระบบ และผู้เขียนโปรแกรม มีความเข้าใจในแบบจำลองที่เขียนขึ้นตรงกัน ลดข้อผิดพลาดในการสื่อสารที่อาจเกิดขึ้นในกระบวนการพัฒนาซอฟท์แวร์ การกำหนดความหมายและความสัมพันธ์ที่ชัดเจนจะนำไปสู่การพัฒนาเครื่องมือทางซอฟท์แวร์ (Software Tools) ที่มีความสามารถในการตรวจสอบความถูกต้องสอดคล้องระหว่างส่วนประกอบต่างๆ ของระบบ ก่อนที่จะเริ่มลงมือเขียนโปรแกรมต่างๆ ของระบบงานประยุกต์ (Application Programs) และจะนำไปสู่การสร้างเครื่องมือทางซอฟท์แวร์สำหรับการสร้างโปรแกรมต่างๆ ในระบบงานโดยอัตโนมัติจากแบบจำลอง ซึ่งจะช่วยลดเวลาและค่าใช้จ่ายในการพัฒนาระบบงานได้เป็นอย่างมาก

งานวิจัยนี้มุ่งเน้นไปที่การสร้างองค์ความรู้ใหม่ในการใช้โปรแกรมเชิงประกาศ (Declarative Program) [4] ที่ใช้ข้อมูลในรูปแบบของ XML Metadata Interchange Format (XMI)[2] เป็นข้อมูลพื้นฐานในการกำหนดความสัมพันธ์ทางความหมายของไดอะแกรมต่างๆ ใน UML เนื่องจาก XMI เป็นรูปแบบมาตรฐานสำหรับแลกเปลี่ยนข้อมูลเกี่ยวกับส่วนประกอบต่างๆ ของแบบจำลองของระบบ และส่วนประกอบต่างๆ ของโปรแกรมบน Internet และ World Wide Web องค์ความรู้นี้จะสอดคล้องกับแนวความคิดในการพัฒนาโปรแกรมแบบเปิด บุคลากรใน

---

1 รายละเอียดขององค์การ OMG สามารถดูได้จาก http://www.omg.org/

2 รายละเอียดของ XMI สามารถดูได้จาก http://www.omg.org/technology/documents

Output จากโครงการวิจัยที่ได้รับทุนจาก สกว.

ผลงานที่เสนอในที่ประชุมทางวิชาการนานาชาติ (รายละเอียดอยู่ในภาคผนวก)

E. Nantajeewarawat, V. Wuwongse, C. Anutariya, K. Akama, and S. Thiemjarus. "Towards Reasoning with UML Diagrams Based-on XML Declarative Description Theory", in V. Kreinovich and J. Daengdej, editors, *Proceedings of the First International Conference on Intelligent Technologies (InTech'2000)*, Bangkok, Thailand, pages 341-350, December 2000. ISBN 974-615-055-3.

E. Nantajeewarawat and R. Sombatsrisomboon. "On the Semantics of UML Diagrams Using Z Notation", in V. Kreinovich and J. Daengdej, editors, *Proceedings of the First International Conference on Intelligent Technologies (InTech'2000)*, Bangkok, Thailand, pages 325-334, December 2000. ISBN 974-615-055-3.

E. Nantajeewarawat, V. Wuwongse, S. Thiemjarus, K. Akama, and C. Anutariya. "Generating Relational Database Schemas from UML Diagrams Through XML Declarative Descriptions", in T. Tanprasert, editor, *Proceedings of the Second International Conference on Intelligent Technologies (InTech'2001)*, Bangkok, Thailand, pages 240-249, November 2001. ISBN 974-615-068-5.

E. Nantajeewarawat, K. Akama, and H. Koike. "Expanding Transformation: A Basis for Verifying the Correctness of Rewriting Rules", in T. Tanprasert, editor, *Proceedings of the Second International Conference on Intelligent Technologies (InTech'2001)*, Bangkok, Thailand, pages 392-401, November 2001. ISBN 974-615-068-5.

K. Akama, E. Nantajeewarawat, and H. Koike. "A Class of Rewriting Rules and Reverse Transformation for Rule-Based Equivalent Transformation", in M. van den Brand and R. Verma, editors, *Proceedings of the Second International Workshop on Rule-Based Programming (RULE-2001)*, Firenze, Italy, pages 4-18, September 2001. [Also published in *Electronic Notes in Theoretical Computer Science*, Vol. 59, No. 4, 16 pages, 2001. Elsevier Science Publishers. ISBN 0444510761.]

H. Unphon and E. Nantajeewarawat. "The Roles of Ontologies in Manipulation of XML Data", in *Proceedings of the Joint International Conference of SNLP-Oriental COCOSDA 2002* (the Fifth Symposium on Natural Language Processing & Oriental COCOSDA Workshop 2002), Hua Hin, Prachuapkirikhan, Thailand, Pages 89-96, May 2002. ISBN 974-572-947-7.

บทความที่ส่งไปเพื่อรับการพิจารณาถึงความเป็นไปได้ในการตีพิมพ์ในวารสารวิชาการนานาชาติ

E. Nantajeewarawat, V. Wuwongse, C. Anutariya, K. Akama, and S. Thiemjarus. "Towards Reasoning with UML Diagrams Based-on XML Declarative Description Theory". Submitted to *International Journal of Intelligent Systems*.

E. Nantajeewarawat and R. Sombatsrisomboon. "On the Semantics of UML Diagrams Using Z Notation". Submitted to *International Journal of Intelligent Systems*.

ทีมพัฒนาซอฟท์แวร์สามารถที่จะทำงานในสถานที่ต่างๆ กัน และใช้เครื่องมือทางซอฟท์แวร์ที่ตนเองถนัดในการทำงานในส่วนที่ตนเองรับผิดชอบ และแลกเปลี่ยนเชื่อมโยงส่วนต่างๆ ของระบบเข้าด้วยกันโดยใช้รูปแบบของ XMI เป็นสื่อกลาง ผ่านระบบเครือข่าย Internet

วัตถุประสงค์ของโครงการมีดังต่อไปนี้

1. เพื่อสร้างองค์ความรู้ใหม่ในการกำหนดความหมายและความสัมพันธ์ของไดอะแกรมต่างๆ ในภาษา UML โดยใช้ทฤษฎีโปรแกรมเชิงประกาศสำหรับ XML [5, 6] เป็นพื้นฐาน

2. ศึกษาค้นคว้าถึงวิธีการนิรนัย (Deduction) ตามเนื้อหาของโปรแกรมเชิงประกาศ โดยใช้วิธีการเปลี่ยนแปลงโดยสมมูล (Equivalent Transformation) [7, 8, 9] เป็นกลไกพื้นฐานในการประมวลผล

3. ศึกษาถึงการนำองค์ความรู้ไปใช้ในการนิรนัยเพื่อสร้างส่วนประกอบของระบบงานประยุกต์ให้สอดคล้องกับรายละเอียดที่กำหนดไว้ในแบบจำลองของระบบงาน

4. ศึกษาเปรียบเทียบแนวทางการกำหนดความสัมพันธ์และความหมายของไดอะแกรมในภาษา UML โดยใช้โปรแกรมเชิงประกาศ กับแนวทางอื่นๆ เช่น การกำหนดความสัมพันธ์โดยใช้ภาษา Z [10]

วิธีการ

โครงการนี้เริ่มต้นด้วยการพัฒนาทฤษฎีโปรแกรมเชิงประกาศสำหรับข้อมูลที่อยู่ในรูปแบบของ XML Metadata Interchange Format (XMI) เพื่อเป็นพื้นฐานในการแสดงความสัมพันธ์โดยนัยระหว่างไดอะแกรมต่างๆ ของแบบจำลองระบบงาน และการแสดงความสัมพันธ์ระหว่างไดอะแกรมกับส่วนประกอบของระบบงานประยุกต์โดยใช้โปรแกรมเชิงประกาศ ต่อด้วยการศึกษาค้นคว้าถึงการนิรนัยตามเนื้อหาของฐานความรู้ เพื่อหาคุณสมบัติโดยนัยของไดอะแกรม และเพื่อสังเคราะห์ส่วนประกอบของงานประยุกต์ (ตัวอย่าง เช่น โครงสร้างฐานข้อมูลแบบสัมพันธ์) จากแบบจำลอง โดยใช้การประมวลผลแบบการเปลี่ยนแปลงโดยสมมูลเป็นกลไกหลัก ลำดับขั้นตอนต่างๆในการวิจัยมีดังต่อไปนี้

1. ออกแบบ Specialization System ที่เหมาะสมเพื่อเป็นโครงสร้างทางคณิตศาสตร์สำหรับการแสดงความสัมพันธ์ระหว่างข้อมูลที่อยู่ในรูปแบบของ XMI โดยจะมีการขยายรูปแบบของ XMI ให้สามารถมีการใช้ตัวแปร (Variables) เพื่อเชื่อมโยงส่วนต่างๆ ของข้อมูล และเพื่อแสดงส่วนของข้อมูลที่ไม่เฉพาะเจาะจงได้ Specialization System ที่ออกแบบขึ้นมานี้จะถูกนำมาใช้เป็นพื้นฐานในการกำหนดโปรแกรมเชิงประกาศสำหรับ XMI/XML

2. ศึกษาถึงความสัมพันธ์โดยนัยทางความหมายของไดอะแกรมต่างๆ ในภาษา UML และบรรยายความสัมพันธ์เหล่านี้โดยใช้โปรแกรมเชิงประกาศบนพื้นฐานของ Specialization System ที่กำหนดขึ้นในข้อ 1

3. ศึกษาถึงวิธีการนิรนัยตามเนื้อหาในโปรแกรมที่สร้างขึ้นในข้อ 2 โดยใช้การประมวลผลแบบการเปลี่ยนแปลงโดยสมมูลเป็นกลไกหลัก

4. ศึกษาการเขียนโปรแกรมเชิงประกาศ เพื่อแสดงความสัมพันธ์ระหว่างไดอะแกรมในภาษา UML กับส่วนประกอบของระบบงานประยุกต์ ที่สอดคล้องกับรายละเอียดที่กำหนดไว้ในแบบจำลองของระบบงาน

5. ศึกษาการนิรนัยเพื่อสังเคราะห์ส่วนประกอบที่สำคัญบางส่วนของระบบงานประยุกต์โดยอัตโนมัติ จากแบบจำลองที่จัดเก็บอยู่ในรูปแบบของ XMI โดยใช้พื้นฐานจากข้อ 1 ข้อ 3 และข้อ 4 และพัฒนาฐานความรู้ต้นแบบ โดยใช้การนิรนัยแบบการเปลี่ยนแปลงโดยสมมูล

6. เปรียบเทียบแนวความคิดที่เสนอขึ้นกับงานวิจัยอื่นๆ ที่เกี่ยวข้อง ตัวอย่างเช่น งานของนักวิจัยนานาชาติในกลุ่ม Precise UML[3]

ผู้สนใจสามารถศึกษาถึงรายละเอียดของวิธีการดังกล่าวได้จากบทความที่รวบรวมไว้ในภาคผนวก

ผลของโครงการ

1. มีการออกแบบ Specialization System ที่เหมาะสมสำหรับการแสดงความสัมพันธ์ระหว่างข้อมูลที่อยู่ในรูปแบบของ XMI

2. วิธีการสำหรับการจัดเก็บความรู้ และการนิรนัย เกี่ยวกับแบบจำลองของระบบงานที่เขียนขึ้นในภาษา UML โดยใช้ทฤษฎีโปรแกรมเชิงประกาศสำหรับ XML เป็นพื้นฐานได้ถูกเสนอขึ้น โดยมีการนำ XML Document Type Definition (DTD) ที่ถูกกำหนดขึ้นโดย XMI มาใช้เป็นมาตรฐานในการจัดเก็บข้อมูลเกี่ยวกับไดอะแกรมต่างๆ

3. มีการศึกษาค้นคว้าถึงวิธีการนิรนัยตามเนื้อหาในโปรแกรมเชิงประกาศสำหรับ XMI โดยใช้การประมวลผลแบบการเปลี่ยนแปลงโดยสมมูลเป็นกลไกหลัก

4. มีการแสดงอย่างชัดเจนถึงวิธีการใช้ Definite Clause สำหรับ XML ในการจัดเก็บความสัมพันธ์และกฎเกณฑ์ทั่วไปในโดเมนของ UML

5. มีการแสดงการนำวิธีการที่เสนอขึ้นไปประยุกต์ใช้ในการจัดเก็บกฎเกณฑ์การสร้างส่วนประกอบหลักส่วนหนึ่งของโปรแกรมประยุกต์ นั่นคือโครงสร้างฐานข้อมูลแบบสัมพันธ์ (Relational Database Schema) จากแบบจำลองระบบงานที่อยู่ในรูปของ UML

---

3 รายละเอียดของกลุ่ม PUML สามารถดูได้จาก http://www.cs.york.ac.uk/puml/

6. ต้นแบบของฐานความรู้ได้ถูกพัฒนาขึ้นเพื่อการทดสอบ

รายละเอียดของผลของโครงการสามารถดูได้จากบทความที่รวบรวมไว้ในภาคผนวก

บทวิจารณ์

เป็นที่คาดหวังว่าวิธีการที่เสนอขึ้นนี้จะสามารถนำไปประยุกต์ใช้ได้กับงานทางด้านการจัดเก็บกฎเกณฑ์การสังเคราะห์ส่วนประกอบของระบบงานประยุกต์โดยอัตโนมัติจากแบบจำลองระบบงาน และงานทางด้านการตรวจสอบความถูกต้องสอดคล้องของแบบจำลองระบบงาน นอกจากนี้ยังมีความเป็นไปได้ที่จะนำไปประยุกต์ใช้ร่วมกับวิธีการอื่นๆ ที่ใช้อยู่ในซอฟท์แวร์สำหรับช่วยการพัฒนาแบบจำลองระบบงานต่างๆ เนื่องจากซอฟท์แวร์เหล่านี้ส่วนใหญ่สามารถอ่าน และบันทึกแบบจำลองระบบงานในรูปแบบ XMI/XML

งานวิจัยที่ควรจะทำในอนาคตต่อเนื่องจากโครงการนี้ คือการปรับปรุงประสิทธิภาพทางด้านความเร็วของกระบวนการนิรนัยโดยใช้ข้อมูลขนาดใหญ่ การทดลองเบื้องต้นโดยใช้ฐานความรู้ต้นแบบที่สร้างขึ้นเพื่อการทดสอบ แสดงให้เห็นว่าข้อมูล XMI/XML ที่แทนไดอะแกรมต่างๆในภาษา UML เป็นข้อมูลที่มีขนาดค่อนข้างใหญ่ ถึงแม้ว่าการนิรนัยตามวิธีการที่เสนอขึ้น จะให้ผลถูกต้องอย่างที่ต้องการ ความเร็วในการประมวลผลยังควรจะได้รับการปรับปรุง ควรมีการค้นคว้าวิจัยเพิ่มเติมเพื่อหาเทคนิคที่จะช่วยในการเพิ่มความเร็วในการประมวลผลข้อมูลขนาดใหญ่โดยใช้กลไกในการประมวลผลแบบสมมูล ซึ่งนอกจากจะทำให้วิธีการที่เสนอขึ้นนี้สามารถถูกนำไปประยุกต์ใช้อย่างมีประสิทธิภาพมากยิ่งขึ้นแล้ว ยังจะเป็นการสร้างองค์ความรู้ใหม่ในการนิรนัยโดยใช้ข้อมูลขนาดใหญ่ในรูปแบบของ XML โดยรวมอีกด้วย เนื่องจากกลไกในการประมวลผลแบบสมมูล เป็นกลไกที่ถูกออกแบบมาให้สามารถรองรับการควบคุมการนิรนัยโดยใช้เนื้อหาของข้อมูลเป็นหลัก โดยไม่ใช้การควบคุมแบบตายตัว การพัฒนาเทคนิคเพื่อช่วยเพิ่มประสิทธิภาพการประมวลผลน่าจะเป็นเรื่องที่เป็นไปได้ โดยควรมีการวิจัยเพิ่มเติมในเรื่องของเทคนิคดังกล่าวทั้งในทางทฤษฎี และทางการทดลองเชิงปฏิบัติ

เอกสารอ้างอิง

[1] Rumbaugh, J., Jacobson, I., and Booch, G., *The Unified Modeling Language Reference Manual*, Addison Wesley, 1999.

[2] Booch, G., Rumbaugh, J., and Jacobson, I., *The Unified Modeling Language User Guide*, Addison Wesley, 1999.

[3] Jacobson, I., Booch, G., and Rumbaugh, J., *The Unified Software Development Process*, Addison Wesley, 1999.

[4] Akama, K., Declarative Semantics of Logic Programs on Parameterized Representation Systems, *Advances in Software Science and Technology*, vol. 5, pp. 45-63, 1993.

[5] Wuwongse, V., Anutariya, C., Akama, K., and Nantajeewarawat, E., XML Declarative Description: A Language for the Semantic Web, *IEEE Intelligent Systems*, May/June 2001, pp. 54-65.

[6] Wuwongse, V., Akama, K., Anutariya, C., and Nantajeewarawat, E., A Data Model for XML Databases, *Lecture Notes in Artificial Intelligence*, vol. 2198, pp. 237-246, 2001.

[7] Akama, K., Shigeta, Y., and Miyamoto, E., Solving Problems by Equivalent Transformation of Logic Programs, Proceedings of the 5th International Conference on Information Systems Analysis and Synthesis, Orlando, Florida, 1999.

[8] Akama, K., Shimizu, T., and Miyamoto, E., Solving Problems by Equivalent Transformation of Declarative Programs, *Journal of the Japanese Society for Artificial Intelligence*, vol. 13, no. 6, pp. 944-952, 1998.

[9] Akama, K., Nantajeewarawat, E., and Koike, H., A Class of Rewriting Rules and Reverse Transformation for Rule-Based Equivalent Transformation, *Electronic Notes in Theoretical Computer Science*, vol. 59, no. 4, Elsevier Science, 2001.

[10] Spivey, J. M., *The Z Notation – a Reference Manual*, Prentice Hall, 2nd Edition, 1992.

ภาคผนวก

Published in Kreinovich, V. and Daengdej, J., editors, Proceedings of the First International Conference on Intelligent Technologies (InTech'2000), Bangkok, Thailand, pages 341-350, December 2000. ISBN 974-615-055-3.

Towards Reasoning with UML Diagrams Based-on XML Declarative Description Theory

Ekawit Nantajeewarawat
IT, Sirindhorn International Inst. of Tech.
Thammasat University, Rangsit Campus
Pathumthani 12121, Thailand
E-mail: ekawit@siit.tu.ac.th

Vilas Wuwongse[1] and Chutiporn Anutariya[2]
CSIM, School of Advanced Technologies
Asian Institute of Technology
Pathumthani 12120, Thailand
E-mail: vw[1],ca[2]@cs.ait.ac.th

Kiyoshi Akama
Center of Information and Multimedia Studies
Hokkaido University, Sapporo 060, Japan
E-mail: akama@cims.hokudai.ac.jp

Surapa Thiemjarus
IT, Sirindhorn International Inst. of Tech.
Thammasat University, Rangsit Campus
Pathumthani 12121, Thailand

Abstract: A practical framework for representing knowledge and reasoning in the domain of UML is proposed. In this framework, graphical diagrams in a UML model are encoded as XML/XMI elements, which are regarded as facts about a specific problem instance in a knowledge base, and the general knowledge on UML, such as inherent interrelationships among diagram components and implicit properties of diagrams, is represented as a set of XML definite clauses. Based on Akama's theory of declarative descriptions, the semantics of such a knowledge base can be precisely determined. Equivalent Transformation is employed as a fundamental computation mechanism for reasoning with the UML diagrams represented in the knowledge base.
Key words: UML, XML/XMI, Declarative description, Knowledge representation, Automated reasoning, Knowledge-based software engineering

1. Introduction

The Unified Modeling Language (UML) [8] is a graphical language, adopted as a standard by the Object Management Group (OMG), for visualizing, specifying, constructing, and documenting the artifacts of a software-intensive system. As reported by recent works on the formal semantics of UML, e.g., [4, 5, 7], there exist inherent interrelationships between components of a UML model. These interrelationships are essentially general knowledge about the domain of UML, which may be used, for example, for deriving implicit properties and verifying the consistency of the model. With this knowledge, a system analyst can make use of the information contained in one diagram to add more components to some other related diagrams, thereby improving the completeness of the model.

This paper proposes a solid practical framework for knowledge representation and reasoning in the domain of UML. The framework is based on the theory of XML declarative descriptions [3, 9], which in turn uses Akama's theory of declarative descriptions (DD theory) [1] as its primary foundation. As outlined in Figure 1, the diagrams in a UML model will be represented as textual structured data in Extensible Markup Language (XML) [6], and the general knowledge about the UML domain as an XML declarative description. Equivalent Transformation (ET) [2] is used as a computation mechanism for inferring the answers to posed queries or for automatic refinement of the encoded UML diagrams according to the represented general knowledge.

One serious question about the feasibility of this approach is how to construct a sufficiently comprehensive XML Document Type Definition (DTD) that can serve as an appropriate schema

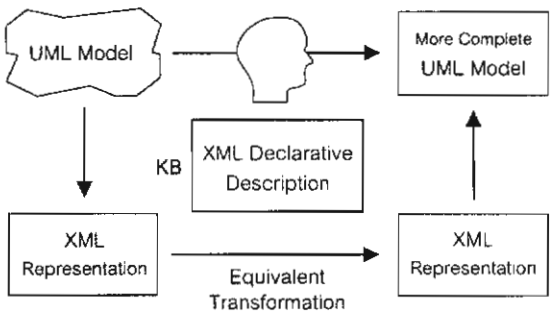

Figure 1: Overview of the Framework

for representing UML diagrams in XML. XML Metadata Interchange Format (XMI) [10], a technology recommended lately by OMG, provides a realistic answer to this. XMI specifies an open information interchange model that facilitates the exchange of programming data over the Internet in a standardized way. It identifies standard XML DTD for UML, and, therefore, provides the presented framework with the ontology of the UML domain. Moreover, the conversion between UML diagrams and XML/XMI representations can be automated by currently available software tools, such as UCI's Argo/UML and IBM's XMI Toolkit.

To start with, DD theory and the concept of XML declarative description are briefly reviewed in Sections 2 and 3, respectively. Section 4 describes, by means of examples, a UML knowledge base represented as an XML declarative description, and Section 5 demonstrates computation with UML diagrams, based on ET paradigm, in the presented framework.

2. Declarative Description Theory

Akama's DD theory [1] is an axiomatic theory which purports to generalize the concept of conventional logic programs to cover a wider variety of data domains. The theory suppresses the differences in the forms of (extended) atomic formulae in various definite-clause knowledge representation languages, and captures the common interrelations between atomic formulae and substitutions by a mathematical abstraction, called a *specialization system*. Despite its simplicity, the specialization system provides a sufficient structure for defining declarative descriptions together with their meanings. DD theory has provided a template for developing declarative semantics for declarative descriptions in various specific data domains.

2.1 Specialization Systems

The concepts of specialization system and declarative description will be reviewed first.

Definition 1 (Specialization System) A *specialization system* is a quadruple $(\mathcal{A}, \mathcal{G}, \mathcal{S}, \mu)$ of three sets $\mathcal{A}, \mathcal{G}$ and $\mathcal{S}$, and a mapping μ from $\mathcal{S}$ to $partial\_map(\mathcal{A})$ (i.e., the set of all partial mappings on $\mathcal{A}$), that satisfies the conditions:

1. $(\forall s', s'' \in \mathcal{S})(\exists s \in \mathcal{S}) : \mu s = (\mu s'') \circ (\mu s')$,
2. $(\exists s \in \mathcal{S})(\forall a \in \mathcal{A}) : (\mu s)a = a$,
3. $\mathcal{G} \subseteq \mathcal{A}$.

The elements of $\mathcal{A}$ are called *atoms*, the set $\mathcal{G}$ *interpretation domain*, the elements of $\mathcal{S}$ *specialization parameters* or simply *specializations*, and the mapping μ *specialization operator*. A specialization $s \in \mathcal{S}$ is said to be *applicable* to $a \in \mathcal{A}$, if and only if $a \in dom(\mu s)$. □

In the sequel, let $\Gamma = (\mathcal{A}, \mathcal{G}, \mathcal{S}, \mu)$ be a specialization system. A specialization in $\mathcal{S}$ will often be denoted by a Greek letter such as θ. For the sake of simplicity, a specialization $\theta \in \mathcal{S}$ will be identified with the partial mapping $\mu\theta$ and used as a postfix unary (partial) operator on $\mathcal{A}$, e.g., $(\mu\theta)a = a\theta$.

2.2 Declarative Descriptions and Their Meanings

A declarative description on Γ will now be defined. Every logic program in the conventional theory can be regarded as a declarative description on some specialization system.

Definition 2 (Definite Clause and Declarative Description) Let X be a subset of $\mathcal{A}$. A *definite clause* C *on* X is a formula of the form:

$$a \leftarrow b_1, \ldots, b_n$$

where $n \geq 0$ and $a, b_1, \ldots, b_n$ are atoms in X. The atom a is denoted by $head(C)$ and the set $\{b_1, \ldots, b_n\}$ by $Body(C)$. A definite clause C such that $Body(C) = \emptyset$ is called *unit clause*. The set of all definite clauses on X is denoted by $Dclause(X)$. A *declarative description* on Γ is a (possibly infinite) subset of $Dclause(\mathcal{A})$. □

Let C be a definite clause $(a \leftarrow b_1, \ldots, b_n)$ on $\mathcal{A}$. A definite clause C' is an *instance* of C, if and only if there exists $\theta \in \mathcal{S}$ such that θ is applicable to $a, b_1, \ldots, b_n$ and $C' = (a\theta \leftarrow b_1\theta, \ldots, b_n\theta)$. Denote by $C\theta$ such an instance C' of C and by $Instance(C)$ the set of all instances of C.

Next, let P be a declarative description on Γ. Denote by $Gclause(P)$ the set

$$\bigcup_{C \in P} (Instance(C) \cap Dclause(\mathcal{G})),$$

i.e., the set of all instances of clauses in P which are constructed solely out of atoms in $\mathcal{G}$. Associated with P is the mapping T_P on $2^{\mathcal{G}}$, defined as follows: for each $X \subseteq \mathcal{G}$, $T_P(X)$ is the set

$$\{head(C) \mid C \in Gclause(P) \ \&\ Body(C) \subseteq X\}.$$

The meaning of P, denoted by $\mathcal{M}(P)$, is then defined by

$$\mathcal{M}(P) = \bigcup_{n=1}^{\infty} T_P^n(\emptyset),$$

where $T_P^1(\emptyset) = T_P(\emptyset)$ and $T_P^n(\emptyset) = T_P(T_P^{n-1}(\emptyset))$ for each $n > 1$.

3. XML Declarative Descriptions

XML is a textual representation of structured or semistructured data, adopted as a standard by the World Wide Web Consortium (W3C). The forms of conventional XML elements will be recalled first, and then extended by incorporation of variables. The concepts of XML specialization system and XML declarative description [3, 9] will next be presented.

3.1 XML Elements

A conventional XML element takes one of the forms:

- $<t\ a_1 = v_1\ \cdots\ a_m = v_m/>$,
- $<t\ a_1 = v_1\ \cdots\ a_m = v_m> v_{m+1} </t>$,
- $<t\ a_1 = v_1\ \cdots\ a_m = v_m> e_1 \cdots e_n </t>$,

where $n, m \geq 0$, t is a tag name (or element type), the a_i are distinct attribute names, the v_i are strings and the e_i are XML elements. An XML element of the first, the second, and the third forms are called *empty*, *simple*, and *nested* elements, respectively.

In the next subsection the concept of an XML element with variables, called an *XML expression*, will be introduced. A variable has two roles. First, it is used as a specialization wild card, i.e., a variable can be specialized into an XML element or a component of an XML element. As its second role, a variable behaves as an equality constraint imposed on components of XML expressions, i.e., all occurrences of the same variable in an expression must be specialized into identical components.

3.2 XML Expressions

Assume that Σ_X is an alphabet comprising the symbols from the following seven sets:

1. A set C of characters.
2. A set N of tag names and attribute names.
3. A set V_N of name-variables, or, for short, N-variables.
4. A set V_S of string-variables, or, for short, S-variables.
5. A set V_P of attribute-value-pair-variables, or, for short, P-variables.
6. A set V_E of XML-expression-variables, or, for short, E-variables.
7. A set V_I of intermediate-expression-variables, or, for short, I-variables.

Also assume that '$' $\notin C$, no element of N begins with '$', and the elements of V_N, V_S, V_P, V_E and V_I begin with "`$N:`", "`$S:`", "`$P:`", "`$E:`" and "`$I:`", respectively.

Definition 3 (XML Expression) An *XML expression* on Σ_X takes one of the following forms:

1. v_E,
2. $<t\ a_1 = v_1\ \cdots\ a_m = v_m\ \ v_{P_1} \cdots v_{P_l}/>$,
3. $<t\ a_1 = v_1\ \cdots\ a_m = v_m\ \ v_{P_1} \cdots v_{P_l}> v_{m+1} </t>$,
4. $<t\ a_1 = v_1\ \cdots\ a_m = v_m\ \ v_{P_1} \cdots v_{P_l}> e_1 \cdots e_n </t>$,
5. $<v_I> e_1 \cdots e_n </v_I>$,

where $l, m, n \geq 0$; $v_E \in V_E$; $t, a_i \in N \cup V_N$; $v_i, v_{m+1} \in C^* \cup V_S$; $a_i \neq a_{i'}$ if $i \neq i'$ $(1 \leq i \leq m; 1 \leq i' \leq m)$; $v_{P_j} \in V_P$ $(1 \leq j \leq l)$; $v_I \in V_I$; and e_k is an XML expression on Σ_X $(1 \leq k \leq n)$. The order of the m pairs $a_1 = v_1 \cdots a_m = v_m$ and the order of the l P-variables $v_{P_1} \cdots v_{P_l}$ are immaterial, but the order of the n expressions $e_1 \cdots e_n$ is important. An XML expression with no occurrence of any variable is called a *ground* XML expression. An XML expression of the second, the third or the fourth form is referred to as a *t-expression*, while that of the fifth form as a *v_I-expression*. A ground t-expression will also be called a *t-element*. When $n = 0$, an XML expression

$$<t\ a_1 = v_1\ \cdots\ a_m = v_m\ \ v_{P_1} \cdots v_{P_l}></t>$$

of the fourth form is assumed to be identical to the XML expression

$$<t\ a_1 = v_1\ \cdots\ a_m = v_m\ \ v_{P_1} \cdots v_{P_l}/>$$

of the second form. The parts enclosed by a pair of < and />, a pair of < and >, or a pair of </ and > are referred to as *tags*. For each i $(1 \leq i \leq m)$, if $a_i \in N$, a_i will be called an *attribute name*, and if $a_i \in V_N$, it will be called an *attribute-name variable*. □

3.3 XML Specialization System and XML Declarative Descriptions

The concept of an XML specialization generation system will be presented first. Based on this structure, the notion of an XML specialization system will then be defined.

Definition 4 (XML Specialization Generation System) Let Δ_X be a quadruple

$$\langle \mathcal{A}_X, \mathcal{G}_X, \mathcal{C}_X, \nu_X \rangle,$$

where $\mathcal{A}_X$ is the set of all XML expressions on Σ_X, $\mathcal{G}_X$ is the set of all ground XML expressions on Σ_X, $\mathcal{C}_X$ is the union of the following sets:

- $V_N \times N$
- $(V_S \times C^*) \cup (V_E \times \mathcal{A}_X)$
- $(V_N \times V_N) \cup (V_S \times V_S) \cup (V_P \times V_P) \cup (V_E \times V_E) \cup (V_I \times V_I)$
- $V_P \times (V_N \times V_S \times V_P)$
- $V_E \times (V_E \times V_E)$
- $(V_P \cup V_E) \times \{\epsilon\}$
- $V_I \times \{\epsilon\}$
- $V_I \times (V_N \times V_P \times V_E \times V_E \times V_I)$;

and $\nu_X : \mathcal{C}_X \rightarrow \mathit{partial\_map}(\mathcal{A}_X)$ is defined as follows: Let $c \in \mathcal{C}_X$ and $a \in \mathcal{A}_X$.

[N-Variable Instantiation]
If $c = (v, b) \in V_N \times N$ and each tag containing v as an attribute-name variable in a does not contain b as an attribute name, then $(\nu_X c)a$ is the XML expression obtained from a by simultaneously replacing each occurrence of v in a with b.

[S- or E-Variable Instantiation]
If $c = (v, b) \in (V_S \times C^*) \cup (V_E \times \mathcal{A}_X)$, then $(\nu_X c)a$ is the XML expression obtained from a by simultaneously replacing each occurrence of v in a with b.

[Variable Renaming]
If $c = (v, u) \in (V_N \times V_N) \cup (V_S \times V_S) \cup (V_P \times V_P) \cup (V_E \times V_E) \cup (V_I \times V_I)$, then $(\nu_X c)a$ is the XML expression obtained from a by simultaneously replacing each occurrence of v in a with u.

[P-Variable Expansion]
If $c = (v, (u, w, v')) \in V_P \times (V_N \times V_S \times V_P)$ and each tag containing v in a does not contain u as an attribute-name variable, then $(\nu_X c)a$ is the XML expression obtained from a by simultaneously replacing each occurrence of v in a with the pair $u = w$ followed by v'.

[E-Variable Expansion]
If $c = (v, (u, w)) \in V_E \times (V_E \times V_E)$, then $(\nu_X c)a$ is the XML expression obtained from a by simultaneously replacing each occurrence of v in a with u followed by w.

[P- or E-Variable Removal]
If $c = (v, \epsilon) \in (V_P \cup V_E) \times \{\epsilon\}$, then $(\nu_X c)a$ is the XML expression obtained from a by removing each occurrence of v in a.

[I-Variable Removal]
If $c = (v, \epsilon) \in V_I \times \{\epsilon\}$, then $(\nu_X c)a$ is the XML expression obtained from a by removing each occurrence of $<v>$ and each occurrence of $</v>$ in a.

[I-Variable Instantiation]
If $c = (v_I, (u_N, u_P, u_E, w_E, v'_I)) \in V_I \times (V_N \times V_P \times V_E \times V_E \times V_I)$, then $(\nu_X c)a$ is the XML expression obtained from a by simultaneously replacing each occurrence in a of each v_I-expression

$$<v_I>\ e_1 \cdots e_n\ </v_I>$$

with the u_N-expression

$$<u_N\ u_P>\ u_E\ <v'_I>\ e_1 \cdots e_n\ </v'_I>\ w_E\ </u_N>.$$

Δ_X will be referred to as the *XML specialization generation system* on Σ_X. □

Next, an XML specialization system is defined.

Definition 5 (XML Specialization System) Based on Δ_X, the *specialization system for XML expressions* on Σ_X, denoted by Γ_X, is defined by

$$\Gamma_X = \langle \mathcal{A}_X, \mathcal{G}_X, \mathcal{S}_X, \mu_X \rangle,$$

where $\mathcal{S}_X = \mathcal{C}_X^*$, i.e., the set of all sequences over $\mathcal{C}_X$, and $\mu_X : \mathcal{S}_X \rightarrow \mathit{partial\_map}(\mathcal{A}_X)$ is given as follows: For each $a \in \mathcal{A}_X$,

- $(\mu_X \lambda)a = a$, where λ denotes the null sequence, and
- for each $c \in \mathcal{C}_X$, $s \in \mathcal{S}_X$, $(\mu_X(c \cdot s))a = (\mu_X s)((\nu_X c)(a))$. □

Obviously, Γ_X satisfies all the three conditions of Definition 1. Examples demonstrating the application of specializations in $\mathcal{S}_X$ to XML expressions will be seen in Section 5.

An *XML declarative description* is then defined as a declarative description on Γ_X, and its declarative meaning follows directly from DD theory.

4. UML Knowledge Base

Subject to the XML DTD for UML specified by XMI, a UML model will be converted into a number of XML elements (ground XML expressions), which are regarded as specific facts about the model. These specific facts will be formalized as ground XML unit clauses, constituting an XML declarative description K_F. By contrast, inherent interrelationships among components of UML diagrams will be represented as another XML declarative description K_R, which basically consists of non-unit XML definite clauses (or rules). The union of K_F and K_R will then be considered as a knowledge base for the model.

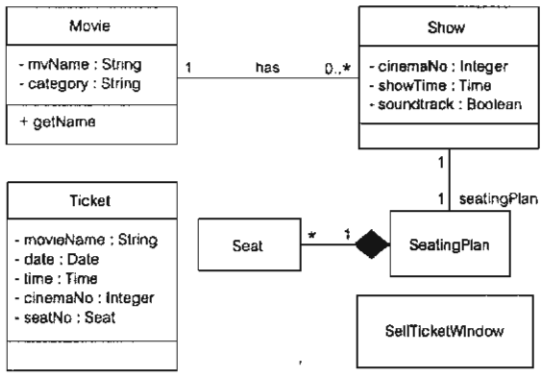

Figure 2: A UML Class Diagram

```
<Class xmi.id="C1.4">
   <name>Movie</name>
   <associationEnd>
      <AssociationEnd xmi.idref="C1.8.2"/>
   </associationEnd>
   <feature>
      <Attribute xmi.id="C1.4.1">
         <name>mvName</name>
         <visibility xmi.value="private"/>
         <type>
            <Primitive xmi.idref="_1.1.21"/>
         </type>
      </Attribute>
      <Attribute xmi.id="C1.4.2">
         <name>category</name>
         <visibility xmi.value="private"/>
         <type>
            <Primitive xmi.idref="_1.1.21"/>
         </type>
      </Attribute>
      <Operation xmi.id="C1.4.11">
         <name>getName</name>
         <visibility xmi.value="public"/>
      </Operation>
   </feature>
</Class>
```

Figure 3: A `Class`-element, C_{mov}

4.1 Encoding UML Diagrams: Examples

Consider the UML class diagram in Figure 2. Each class in this diagram is represented in XML/XMI by a `Class`-element, and each association by an `Association`-element. For example, the class Movie in Figure 2 is represented by the `Class`-element in Figure 3, and the association has in Figure 2 by the `Association`-element in Figure 4. The `associationEnd`-element of the `Class`-element in Figure 3 specifies that Movie is a class at an endpoint of the association has by referring to the second `AssociationEnd`-element of the `connection`-

```
<Association xmi.id="C1.8">
   <name>has</name>
   <connection>
      <AssociationEnd xmi.id="C1.8.1">
         <name/>
         <isNavigable xmi.value="true"/>
         <aggregation xmi.value="none"/>
         <multiplicity>0..*</multiplicity>
         <type>
            <Class xmi.idref="C1.3"/>
         </type>
      </AssociationEnd>
      <AssociationEnd xmi.id="C1.8.2">
         <name/>
         <isNavigable xmi.value="true"/>
         <aggregation xmi.value="none"/>
         <multiplicity>1</multiplicity>
         <type>
            <Class xmi.idref="C1.4"/>
         </type>
      </AssociationEnd>
   </connection>
</Association>
```

Figure 4: An `Association`-element, C_{has}

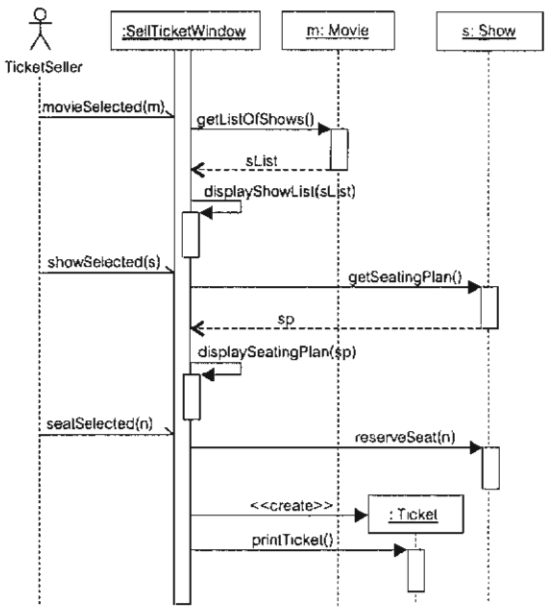

Figure 5: A UML Sequence Diagram

element in Figure 4. The `feature`-element in Figure 3 describes the attributes and operations of Movie. Each `AssociationEnd`-element inside the `connection`-element in Figure 4 details an endpoint, e.g., its multiplicity, connected class and navigability, of the association has. (In the figure, assume that the identifiers of the `Class`-elements representing the classes Show and Movie are `C1.3` and `C1.4`, respectively.)

```
<Collaboration xmi.id="S3">
   <name>SellingMovieTicket</name>
   <ownedElement>
      <ClassifierRole>
         <name>TicketSeller</name>
         <message>
            <Message xmi.idref="S20"/>
            <Message xmi.idref="S21"/>
            <Message xmi.idref="S22"/>
         </message>
      </ClassifierRole>
      <ClassifierRole>
         <name/>
         <message>
            <Message xmi.idref="S23"/>
            <Message xmi.idref="S24"/>
            <Message xmi.idref="S25"/>
            <Message xmi.idref="S26"/>
            <Message xmi.idref="S27"/>
            <Message xmi.idref="S28"/>
            <Message xmi.idref="S29"/>
         </message>
         <message2>
            <Message xmi.idref="S20"/>
            <Message xmi.idref="S20.5"/>
            <Message xmi.idref="S21"/>
            <Message xmi.idref="S21.5"/>
            <Message xmi.idref="S22"/>
         </message2>
         <base>
            <Class xmi.idref="C1.2"/>
         </base>
      </ClassifierRole>
      <ClassifierRole>
         <name>m</name>
         <message>
            <Message xmi.idref="S20.5"/>
         </message>
         <message2>
            <Message xmi.idref="S23"/>
         </message2>
         <base>
            <Class xmi.idref="C1.4"/>
         </base>
      </ClassifierRole>
      <ClassifierRole>
         <name>s</name>
           ...
      </ClassifierRole>
      <ClassifierRole>
         <name/>
           ...
      </ClassifierRole>
   </ownedElement>
   <interaction>
     ...
        ...
     ...
   </interaction>
</Collaboration>
```

Figure 6: A `Collaboration`-element, C_{seq}

```
<SendAction xmi.id="S138">
   <name>movieSelected</name>
   <isAsynchronous xmi.value="true"/>
   <Action.message>
      <Message xmi.idref="S20"/>
   </Action.message>
     ...
   <actualArgument>
      <Argument>
         <name>m</name>
      </Argument>
   </actualArgument>
</SendAction>
```

Figure 7: A `SendAction`-element, C_{mvs}

```
<CallAction xmi.id="S141">
   <name>getListOfShows</name>
   <isAsynchronous xmi.value="false"/>
   <Action.message>
      <Message xmi.idref="S23"/>
   </Action.message>
</CallAction>
```

Figure 8: A `CallAction`-element, C_{get}

```
<CallAction xmi.id="S143">
   <name>displayShowList</name>
   <isAsynchronous xmi.value="false"/>
   <Action.message>
      <Message xmi.idref="S24"/>
   </Action.message>
   <actualArgument>
      <Argument>
         <name>sList</name>
      </Argument>
   </actualArgument>
</CallAction>
```

Figure 9: A `CallAction`-element, C_{dsp}

```
<ReturnAction xmi.id="S142">
   <name/>
   <isAsynchronous xmi.value="false"/>
   <Action.message>
      <Message xmi.idref="S20.5"/>
   </Action.message>
   <actualArgument>
      <Argument>
         <name>sList</name>
      </Argument>
   </actualArgument>
</ReturnAction>
```

Figure 10: A `ReturnAction`-element, C_{ret}

Figure 6 illustrates the XML/XMI representation of the UML sequence diagram in Figure 5, which describes a normal scenario of the use case "Selling Movie Ticket" of a movie-ticketing system. The `ownedElement`-subelement of the `Collaboration`-element in Figure 6 contains five `ClassifierRole`-elements, each of which describes an object or an actor participating in the sequence diagram. A `ClassifierRole`-element typically has one `message`-element and one `message2`-element, referring to the messages sent and received, respectively, by the object or the actor the element describes. For example, the `message`-element in the first `ClassifierRole`-element in Figure 6 indicates that the actor TicketSeller sends three messages, which are described by the `Message`-elements having the identifiers `S20`, `S21` and `S22`, respectively, and the absence of the `message2`-element signifies that this actor receives no message. Likewise, the `message`-element and the `message2`-element in the second `ClassifierRole`-element itemize the messages the anonymous SellTicketWindow object sends and receives, respectively. A `ClassifierRole`-element describing an object, such as the second and the third `ClassifierRole`-elements, normally contains a `base`-element, which refers to the class of the described object. (Assume that the identifier of the `Class`-element for the class SellTicketWindow is `C1.2`.)

The `Message`-elements referred to by the `message`-elements and the `message2`-elements, together with the predecessor relation and the successor relation on their corresponding messages in the sequence diagram, are defined within the `interaction`-subelement (the last subelement) in Figure 6. Due to space limitation, the details of this `interaction`-element and the `ClassifierRole`-elements for the Show object s and the anonymous Ticket object are not shown.

The action of each message is specified by a `SendAction`-element, a `CallAction`-element or a `ReturnAction`-element, depending on the type of the message. For example, the operation of the `Message`-element having the identifier `S20`, i.e., the first message the actor TicketSeller sends, is detailed by the `SendAction`-element in Figure 7. Similarly, the operations of the `Message`-elements having the identifiers `S23` and `S24`, i.e., the first and the second messages the SellTicketWindow object sends, are described by the `CallAction`-elements in Figures 8 and 9, respectively; and the `Message`-element with the identifier `S20.5`, i.e., the first return message the SellTicketWindow object receives, is defined by the `ReturnAction`-element in Figure 10.

```
<Class xmi.id=$S:CID $P:1> $E:1
   <feature> $E:2
      <Operation>
         <name>$S:NM</name>
      </Operation>
   </feature>
</Class>

   ←

   <$I:1>
      <classifierRole> $E:3
         <message2> $E:4
            <Message xmi.idref=$S:MID/> $E:5
         </message2> $E:6
         <base>
            <Class xmi.idref=$S:CID/>
         </base>
      </classifierRole>
   </$I:1>,

   <CallAction $P:2>
      <name>$S:NM</name> $E:7
      <Action.message>
         <Message xmi.idref=$S:MID/>
      </Action.message> $E:8
   </CallAction>,

   <Class xmi.id=$S:CID $P:1> $E:1
      <feature> $E:2
      </feature>
   </Class>
```

Figure 11: A Definition Clause, C_{R1}

4.2 General Knowledge about the Domain

The detailed formal analysis of the semantics of UML in [4, 5, 7] uncovers several inherent interrelationships between UML diagrams as well as implicit properties of diagram components. The descriptions of these interrelationships and properties can be regarded as axioms (or general rules) in the domain of UML, which will be represented as XML definite clauses in the proposed framework.

As an illustration, the axiomatic assertion that "the operation of any message received by an object in a sequence diagram must be an operation provided by the class of that object in a class diagram", given in [7], can be encoded as the XML definite clause in Figure 11. More comprehensively, this definite clause states that if

- the `$I:1`-expression in its body can be specialized into an XML-element that contains a `classifierRole`-element for an object having a `Message`-element identified by `$S:MID` as the representation of one of its received messages and having a `Class`-element identified by `$S:CID` as the representation of its class, and

- there is a `CallAction`-element that has as its name `$S:NM` and refers to the `Message`-element having the identifier `$S:MID`,

then the `feature`-element of the `Class`-element with the identifier `$S:CID` has an `Operation`-element with the name `$S:NM`. Observe that the `Class`-expression in the head and that in the body of this clause are identical except that the expression in the head has an additional `Operation`-expression inside its `feature`-subexpression. Each of the E-variables occurring in the clause, e.g., `$E:1` and `$E:2`, can be instantiated into zero or more XML elements.

Figure 12 provides another example of an encoded general rule. The XML definite clause in the figure represents the axiom "a class inherits from its superclass the associations that the superclass has with other classes along with the information about the multiplicities of the endpoints that connect the associations with those classes", by stating that if there are

- a `Generalization`-element, describing a generalization relationship, of which the `child`-subelement and the `parent`-subelement refer to the `Class`-elements having the identifiers `$S:SubID` and `$S:SupId`, respectively, and
- an `Association`-element with an `AssociationEnd`-element referring to the `Class`-element identified by `$S:SupId`,

then one can construct another `Association`-element that has the same content as the former `Association`-element except that the first `AssociationEnd`-element is replaced with an `AssociationEnd`-element that refers to the `Class`-element identified by `$S:SubId` and contains no `multiplicity`-element, and the second `AssociationEnd`-element is replaced with an `AssociationEnd`-element having the same `multiplicity`-element and the same `type`-element.

```
<Association>
   <$I:1>
      <AssociationEnd>
         <type>
            <Class xmi.idref=$S:SubID/>
         </type>
      </AssociationEnd>
      <AssociationEnd>
         <multiplicity>$S:M2</multiplicity>
         <type>$E:C</type>
      </AssociationEnd>
   </$I:1>
</Association>

   ←

   <Generalization> $E:1
      <child>
         <Class xmi.idref=$S:SubID/>
      </child>
      <parent>
         <Class xmi.idref=$S:SupID/>
      </parent>
   </Generalization>,

   <Association $P:1>
      <$I:1>
         <AssociationEnd $P:2> $E:2
            <multiplicity>$S:M1</multiplicity>
            <type>
               <Class xmi.idref=$S:SupID/>
            </type>
         </AssociationEnd>
         <AssociationEnd $P:3> $E:3
            <multiplicity>$S:M2</multiplicity>
            <type>$E:C</type>
         </AssociationEnd>
      </$I:1>
   </Association>
```

Figure 12: A Definite Clause, C_{R2}

5. Equivalent Transformation

Equivalent Transformation (ET) paradigm [2] is a new computational model for solving problems based on semantics-preserving transformation. In ET framework, the specification of a problem is formalized as a declarative description, and the problem will be solved by transforming this declarative description successively into a simpler but equivalent declarative description, from which the solutions to the problem can be obtained easily and directly.

The correctness of the computation mechanism in ET paradigm relies solely on the equivalence of all declarative descriptions in a transformation process. Two declarative descriptions P and P' are said to be *equivalent* if and only if they have exactly the same meaning, i.e., $\mathcal{M}(P) = \mathcal{M}(P')$. In this paper, only unfolding transformation will be applied. In general, other kinds of semantics-preserving transformation can also be used, especially to improve computation efficiency.

To demonstrate computation with XML/XMI elements under ET framework, assume that $C_{mov}, C_{has}, C_{seq}, C_{mvs}, C_{get}, C_{dsp}$ and C_{ret} are the unit clauses the heads of which are the XML/XMI elements in Figures 3, 4, 6, 7, 8, 9 and 10, respectively; also that C_{R1} and C_{R2} are, respectively, the definite clauses in Figures 11 and 12. Then, let KB be the XML declarative description consisting of these nine definite clauses. Now suppose that one wants to find the names of the operations provided by the class Movie. The

problem can be formulated as the declarative description

$$P_1 = KB \cup \{C_0\},$$

where C_0 is the definite clause

```
<answer>$S:X</answer> ←
      <Class $P:Y1>
         <name>Movie</name> $E:Y2
         <feature> $E:Y3
            <Operation $P:Y4>
               <name>$S:X</name> $E:Y5
            </Operation> $E:Y6
         </feature>
      </Class>.
```

The `class`-expression in the body of C_0 is unifiable with the head of the unit clause C_{mov} (Figure 3) using the specialization

$$\langle(\texttt{\$P:Y1}, (\texttt{\$N:V1}, \texttt{\$S:V2}, \texttt{\$P:V3})),$$
$$(\texttt{\$N:V1}, \texttt{xmi.id}), (\texttt{\$S:V2}, \texttt{"C1.4"}), (\texttt{\$P:V3}, \epsilon),$$
$$(\texttt{\$E:Y2}, E_1), (\texttt{\$E:Y3}, (\texttt{\$E:V4}, \texttt{\$E:V5})),$$
$$(\texttt{\$E:V4}, E_2), (\texttt{\$E:V5}, E_3),$$
$$(\texttt{\$P:Y4}, (\texttt{\$N:V6}, \texttt{\$S:V7}, \texttt{\$P:V8})),$$
$$(\texttt{\$N:V6}, \texttt{xmi.id}), (\texttt{\$S:V7}, \texttt{"C1.4.11"}),$$
$$(\texttt{\$P:V8}, \epsilon), (\texttt{\$S:X}, \texttt{getName}),$$
$$(\texttt{\$E:Y5}, E_4), (\texttt{\$E:Y6}, \epsilon)\rangle$$

in $\mathcal{S}_X$ as a unifier, where E_1, E_2, E_3 and E_4 denote the `associationEnd`-element, the first and the second `Attribute`-elements and the last `visibility`-element, respectively, in C_{mov}. This `class`-expression in C_0 is moreover unifiable with the head of the clause C_{R1} (Figure 11) using the unifier

$$\langle(\texttt{\$P:Y1}, (\texttt{\$N:W1}, \texttt{\$S:W2}, \texttt{\$P:W3})),$$
$$(\texttt{\$N:W1}, \texttt{xmi.id}), (\texttt{\$S:W2}, \texttt{\$S:CID}), (\texttt{\$P:W3}, \texttt{\$P:1}),$$
$$(\texttt{\$E:1}, (\texttt{\$E:W4}, \texttt{\$E:W5})),$$
$$(\texttt{\$E:W4}, \texttt{<name>Movie</name>}),$$
$$(\texttt{\$E:W5}, \texttt{\$E:Y2}), (\texttt{\$E:2}, \texttt{\$E:Y3}), (\texttt{\$P:Y4}, \epsilon),$$
$$(\texttt{\$S:NM}, \texttt{\$S:X}), (\texttt{\$E:Y5}, \epsilon), (\texttt{\$E:Y6}, \epsilon)\rangle.$$

By unfolding C_0, P_1 can thus be transformed into

$$P_2 = KB \cup \{C_1, C_2\},$$

where C_1 is the unit clause

```
<answer>getName</answer> ←
```

and C_2 is the definite clause with the head `<answer>$S:X</answer>` and with the same body as that of C_{R1} except that `$S:NM` is replaced with `$S:X` and the `Class`-expression in the body is changed into

```
<Class xmi.id=$S:CID $P:1>
   <name>Movie</name> $E:Y2
   <feature> $E:Y3
   </feature>
</Class>.
```

At this step, one answer, i.e., `getName`, is directly obtained from C_1. Other answers may be computed by further transforming P_2. The `Class`-expression in the body of C_2 is unifiable with the unit clause C_{mov} (Figure 3) using the unifier

$$\langle(\texttt{\$S:CID}, \texttt{"C1.4"}), (\texttt{\$P:1}, \epsilon), (\texttt{\$E:Y2}, E_1),$$
$$(\texttt{\$E:Y3}, (\texttt{\$E:U1}, \texttt{\$E:U2})), (\texttt{\$E:U2}, (\texttt{\$E:U3}, \texttt{\$E:U4})),$$
$$(\texttt{\$E:U1}, E_2), (\texttt{\$E:U3}, E_3), (\texttt{\$E:U4}, E_5)\rangle,$$

where E_1, E_2, E_3 and E_5 denote the `associationEnd`-element, the first and the second `Attribute`-elements and the `Operation`-element, respectively, in C_{mov}. By resolving C_2 with C_{mov} upon this `Class`-expression, P_2 is rewritten into

$$P_3 = KB \cup \{C_1, C_3\},$$

where the head of the clause C_3 is the same as that of C_2, and the body of C_3 is same as the body of C_{R1} except that `$S:NM` is replaced with `$S:X`, `$S:CID` with `"C1.4"` and the `Class`-expression in the body is removed. Next, the `$I:1`-expression in the body of C_3 can be unified with the unit clause C_{seq} (Figure 6) using the specialization

$$\langle(\texttt{\$I:1}, (\texttt{\$N:Z1}, \texttt{\$P:Z2}, \texttt{\$E:Z3}, \texttt{\$E:Z4}, \texttt{\$I:Z5})),$$
$$(\texttt{\$N:Z1}, \texttt{Collaboration}), (\texttt{\$E:Z4}, E_6),$$
$$(\texttt{\$P:Z2}, (\texttt{\$N:Z6}, \texttt{\$S:Z7}, \texttt{\$P:Z8})),$$
$$(\texttt{\$N:Z6}, \texttt{xmi.id}), (\texttt{\$S:Z7}, \texttt{"S3"}), (\texttt{\$P:Z8}, \epsilon),$$
$$(\texttt{\$E:Z3}, \texttt{<name>SellingMovieTicket</name>}),$$
$$(\texttt{\$I:5}, (\texttt{\$N:Z9}, \texttt{\$P:Z10}, \texttt{\$E:Z11}, \texttt{\$E:Z12}, \texttt{\$I:Z13})),$$
$$(\texttt{\$N:Z9}, \texttt{ownedElement}), (\texttt{\$P:Z10}, \epsilon),$$
$$(\texttt{\$E:Z11}, (\texttt{\$E:Z14}, \texttt{\$E:Z15})),$$
$$(\texttt{\$E:Z12}, (\texttt{\$E:Z16}, \texttt{\$E:Z17})),$$
$$(\texttt{\$E:Z14}, E_7), (\texttt{\$E:Z15}, E_8),$$
$$(\texttt{\$E:Z16}, E_9), (\texttt{\$E:Z17}, E_{10}),$$
$$(\texttt{\$I:13}, \epsilon), (\texttt{\$E:3}, (\texttt{\$E:Z18}, \texttt{\$E:Z19})),$$
$$(\texttt{\$E:Z18}, \texttt{<name>m</name>}), (\texttt{\$E:Z19}, E_{11}),$$
$$(\texttt{\$E:4}, \epsilon), (\texttt{\$E:5}, \epsilon), (\texttt{\$E:6}, \epsilon), (\texttt{\$S:MID}, \texttt{"S23"})\rangle$$

in $\mathcal{S}_X$, where $E_6, E_7, E_8, E_9, E_{10}$ and E_{11} denote the `interaction`-element, the first, the second, the fourth and the fifth `ClassifierRole`-elements, and the `message`-subelement of the third `ClassifierRole`-element, respectively, in C_{seq}. As a result, P_3 can be transformed into

$$P_4 = KB \cup \{C_1, C_4\},$$

where C_4 is the clause

```
<answer>$S:X</answer> ←
      <CallAction $P:2>
         <name>$S:X</name> $E:7
         <Action.message>
            <Message xmi.idref="S23"/>
```

```
        </Action.message> $E:8
    </CallAction>.
```

Obviously, by further resolving the clause C_4 with the unit clause C_{get} (Figure 8), P_4 can be transformed into

$$P_5 = KB \cup \{C_1, C_5\},$$

where C_5 is the unit clause

```
<answer>getListOfShows</answer> ← ,
```

from which the second answer, i.e., **getListOfShows**, can be directly drawn. As neither C_1 nor C_5 can further be transformed, no other answer will be derived. Since only unfolding transformation, which always preserves the equivalence of declarative descriptions, is used in each step,

$$\mathcal{M}(P_1) = \mathcal{M}(P_5),$$

and the two obtained answers are guaranteed to be correct with respect to KB. Providing that KB is augmented with the XML/XMI elements representing all components of the diagrams in Figures 2 and 5, one can derive, for example, the names of the operations offered by the class Show, i.e., getSeatingPlan and reserveSeat, through the clause C_{R1}, in a similar way (although the class Show has no explicitly declared operation in the class diagram of Figure 2).

6. Concluding Remarks

Apart from the general rules illustrated in this paper, encoding other known implicit interrelationships between UML diagrams as XML definite clauses along with discovering additional inherent interrelationships in the UML domain is in progress. The development of a prototype UML knowledge-based system under the proposed framework is now under way at AIT and SIIT. Since program code, e.g., Java code, can also be represented as XML data, the presented framework furthermore has a significant application in forward engineering—the process of transforming a model into code through a mapping to an implementation language. As depicted by Figure 13, general rules specifying the mapping, i.e., forward engineering knowledge, can be expressed as an XML declarative description, and Equivalent Transformation can be used as underlying inference machinery for generating program code from a UML model. Acquisition of forward engineering knowledge in the UML domain is also an ongoing research at SIIT.

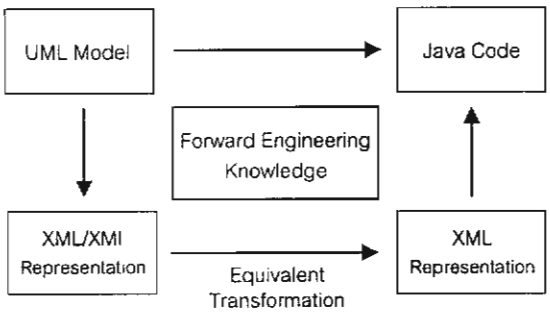

Figure 13: Forward Engineering Framework

Acknowledgement

This work was supported by the Thailand Research Fund, under Grant No. PDF/31/2543.

References

[1] Akama, K., Declarative Semantics of Logic Programs on Parameterized Representation Systems, *Advances in Software Science and Technology*, vol. 5, pp. 45–63, 1993.

[2] Akama, K., Shimitsu, T. and Miyamoto, E., Solving Problems by Equivalent Transformation of Declarative Programs, *J. JSAI*, vol. 13, no. 6, pp. 944–952, 1998.

[3] Anutariya, C., Wuwongse, V., Nantajeewarawat, E. and Akama, K., Towards a Foundation for XML Document Databases, *Proc. 1st International Conference on E-Commerce and Web Technologies*, UK, Lecture Notes in Computer Science, vol. 1875, pp. 324–333, Springer-Verlag, 2000.

[4] Evans A. S., Reasoning with UML Class Diagrams, *Proc. 2nd IEEE workshop on Industrial-Strength Formal Specification Techniques*, Florida, IEEE Press, 1998.

[5] France, R., Evans, A. S., Lano, K. and Rumpe, B., The UML as a Formal Modeling Notation, *Computer Standards and Interfaces*, vol. 19, no. 7, pp. 325–334, 1998.

[6] Goldfarb, C. F. and Prescod, P., *The XML Handbook*, Prentice Hall, 1998.

[7] Nantajeewarawat, E. and Sombatsrisomboon, R., On the Semantics of UML Diagrams Using Z Notaion, *Proc. International Conference on Intelligent Technologies*, Bangkok, Thailand, 2000.

[8] Rumbaugh, J., Jacobson, I. and Booch, G., *The Unified Modeling Language Reference Manual*, Addison Wesley, 1999.

[9] Wuwongse, V., Akama, K., Anutariya, C. and Nantajeewarawat, E., A Foundation for XML Document Databases: Data Model, Technical Report, CSIM, AIT, 1999.

[10] XML Metadata Interchange Format (XMI), IBM Application Development, www-4.ibm.com/software/ad/standards/xmi.html.

Published in Kreinovich, V. and Daengdej, J., editors, Proceedings of the First International Conference on Intelligent Technologies (InTech'2000), Bangkok, Thailand, pages 325-334, December 2000. ISBN 974-615-055-3.

On the Semantics of UML Diagrams Using Z Notation

Ekawit Nantajeewarawat and Ratanachai Sombatsrisomboon
Information Technology Program
Sirindhorn International Institute of Technology
Thammasat University, Rangsit Campus
Pathum Thani 12121, Thailand
E-mail: ekawit@siit.tu.ac.th

Abstract: After the method war in the early 90's, the Unified Modeling Language (UML) has emerged as a de facto standard notation for object-oriented system analysis and design. However, due to the lack of the precise semantics of UML, interrelationships among components of UML models can hardly be analyzed and the consistency of the models cannot be formally verified. As a step towards the precise semantics of UML, this paper employs the Z notation, an expressive mathematical language, to develop formal specifications for two important parts of UML, i.e., class diagrams and sequence diagrams, and to precisely define the well-formedness rules and the model-theoretic semantics of these two kinds of diagrams. Based on this established foundation, a number of sound deductive inference rules, which can be used for rigorously reasoning with UML class diagrams and sequence diagrams, are presented.

Key words: UML, Model-theoretic semantics, Formal deduction, Entailment, Inference rules, Inheritance, Diagrammatical transformation

1. Introduction

In response to the popularity of object-oriented software development, more than thirty different object-oriented modeling methods and languages were proposed during 1889-1994. The differences between these methods and notations were nonetheless often superficial, e.g., the same concept was often realized using subtly different graphical syntax and terminology in different methods. System analysts and software developers had difficulty in choosing a suitable modeling language that met their requirements completely and in understanding software specifications written in various modeling languages. Before long, three leading object-oriented methodologists, Booch, Jacobson and Rumbaugh, were motivated to unify the modeling notations of their methods, i.e., the Booch method, Jacobson's OOSE and Rumbaugh's OMT, and to incorporate ideas from other modeling languages, and began to develop the Unified Modeling Language (UML) [1, 5, 7, 8, 9], which has become a standard modeling language for object-oriented systems.

Although the UML architects have claimed that UML has a well-defined semantics, as defined in the UML Semantics Document [9], its current semantics is only described in a "semi-formal" style that combines graphical notation and formal language with lengthy and loose explanations in natural language (English), and is not sufficiently precise. The lack of the precise semantics is a serious hindrance to the detailed and accurate analysis of the interrelationships between model components as well as their properties, the verification of the consistency and correctness of designs, and, moreover, the construction of rigorous system-modeling and automation tools.

1.1 Related Works

Being well aware of the necessity of the formal and precise semantics and a solid theoretical basis, international researchers and practitioners in the precise UML group (pUML) [10], who share the aim of developing UML as a precise modeling language, have attempted to clarify and make precise the semantics of UML [2, 3, 4]. The Z notation [11, 12, 13], a mature and expressive mathematical language that is well supported by tools, is employed to describe the abstract syntax and constraints on the syntactic structures of graphical object-oriented notation in UML, define the semantics domain and associate meanings with well-formed syntactic structures. The concept of entailment between UML diagrams is formulated, and a set of sound inference rules, called *diagrammatical transformation rules*, each of which transforms a given diagram into some of its logical consequence, is introduced as a tool for proving properties of and reasoning about components of UML models.

As an illustration of reasoning through diagrammatical transformation, consider the UML class diagram in Figure 1, which describes the relationship between students and instructors and that between students and courses. In addition to specifying that each student has exactly one instructor as his/her advisor, the diagram also asserts that each full-time

student takes at least three but at most ten courses, and, on the other hand, at least fifteen students take each course. In order to deduce the relationship between students and courses in general, a few inference rules introduced by [2] can be successively applied to transform the class diagram in Figure 1 into the class diagram in Figure 2, from which the conclusion that some student may take no course can be directly drawn.

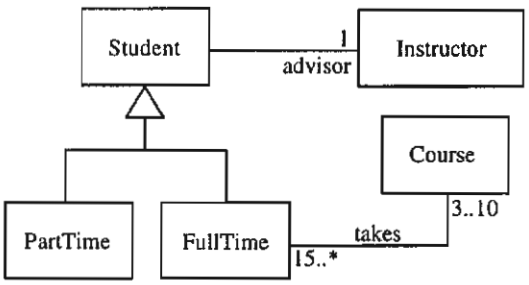

Figure 1: A Class Diagram

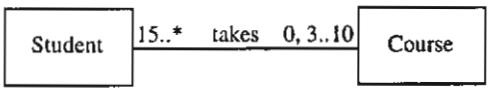

Figure 2: Part of a Derived Class Diagram

However, the works reported by this group [2, 3, 4] presently capture the syntax and semantics of only some components of UML class diagrams, i.e., classes, associations and generalization relationships, and the inference rules presented in these works can only be used for reasoning about the properties of these components. How to deal with internal components of classes, e.g., attributes and operations, how to formulate the concrete semantics of other prominent kinds of UML diagrams, e.g., sequence diagrams, collaboration diagrams, statechart diagrams and activity diagrams, and how to formally analyze and reason about their interrelationships and their properties remain challenging issues.

1.2 The Presented Work

As a step towards the precise semantics of UML, this paper first extends the abstract syntax of and the well-formedness rules for class diagrams in [2, 3, 4] to embrace attribute declarations and operation declarations, which are important internal components of classes, and defines the abstract syntax of UML sequence diagrams as well as the well-formedness rules for them (Section 2). An appropriate semantics domain for assigning meanings to components of UML class diagrams and to those of UML sequence diagrams is then specified, and the model-theoretic semantics of these two kinds of diagrams is developed (Section 3). The proposed semantics enables the precise discussion on inheritance of attributes and operations and, moreover, the analysis of the inherent interrelationships between class diagrams and sequence diagrams. Sound inference rules for deductive reasoning about inheritance and about interconnections between components of the two kinds of UML diagrams are presented (Section 4). Applicability of the proposed inference rules in computer-aided software engineering tools is explained (Section 5).

2. Well-Formed Diagrams

Subsection 2.1 briefly recalls the abstract syntax of some basic concepts, i.e., *AssociationEnd* and *Association*, defined by [2, 4], and, then, defines the abstract syntax of attributes and operations together with the concept of a well-formed class diagram. Subsection 2.2 defines the abstract syntax of the components of UML sequence diagrams along with the notion a well-formed sequence diagram.

Throughout the paper, the sets *ClassName*, *ObjectName*, *Actor* and *Name* are assumed as basic types. These four sets are presumed to contain all class names, object names, actors, and other names (e.g., attribute names, operation names, association names, association-end names, and parameter names), respectively, used in a model.

2.1 Well-Formed Class Diagrams

An association represents a structural relationship among objects. An association typically has two end-points, called *association ends*, each of which connects the association with a class of objects. The schema for association ends is given below.

AssociationEnd

$rolename : Name$
$class : ClassName$
$multiplicity : \mathbb{P}_1 \mathbb{N}$

$multiplicity \neq \{0\}$

A *role name* of an association end specifies the role that an object of its connected class plays in an association. A *multiplicity* specifies the possible number of objects that may be connected across an association instance. A multiplicity is defined as a non-empty subset of the set $\mathbb{N}$ of non-negative integers. Since the multiplicity $\{0\}$ of an association end indicates that the association does not actually exist, the constraint that a multiplicity cannot be the singleton set $\{0\}$ is imposed. An *association* has two association ends with different role names.

Association

$name : Name$
$e_1, e_2 : AssociationEnd$

$e_1.rolename \neq e_2.rolename$

Example 1 Consider the class diagram in Figure 3. The set *ClassName* is assumed to contain *Person*, *Student*, *Instructor*, *Course*, *String*, *Money*, *Year* and *Integer*; and the set *Name* is assumed to contain *advisor*, *takes*, *name*, *addr*, *chngAddr*, *salary*, *spouse*, *getSalary*, *y*, *code* and *credit*. This class diagram contains four association ends, i.e., ae_1, ae_2, ae_3 and ae_4. The value of $ae_2.rolename$ is *advisor*, while the *rolename* of each of the other three association ends is undefined. The values of $ae_1.class$ and $ae_2.class$ are *Student* and *Instructor*, respectively, while those of $ae_3.class$ and $ae_4.class$ are *Student* and *Course*, respectively. The multiplicity of ae_1 is unspecified, the multiplicity of ae_2 is the singleton set $\{1\}$, and the multiplicities of ae_3 and ae_4 are the infinite sets $\{n \in \mathbb{N} \mid n \geq 15\}$ and $\mathbb{N}$, respectively. There are two associations in the figure, i.e., a_1 and a_2, where $a_1.name$ is undefined, $a_1.e_1$ and $a_1.e_2$ are ae_1 and ae_2, respectively, and $a_2.name$, $a_2.e_1$ and $a_2.e_2$ are *takes*, ae_3 and ae_4, respectively. ■

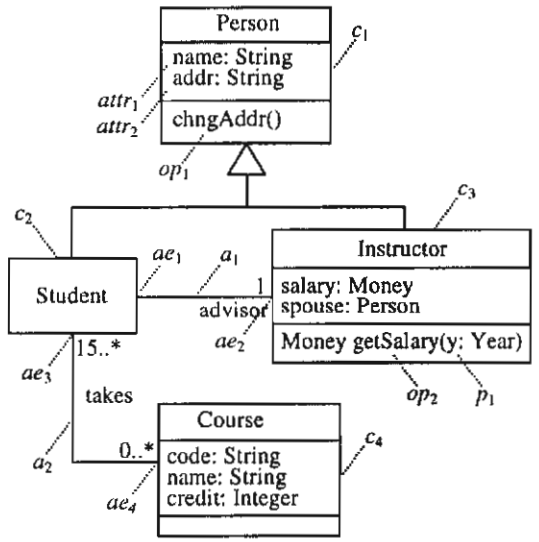

Figure 3: A Class Diagram

A class may have attributes and operations as its components. An *attribute* has a name and a type specifying its possible values.

Attribute
name : *Name*
type : *ClassName*

The signature of an *operation* is a combination of its name, its type and a number of formal parameters (formal arguments). The type of an operation specifies the range of possible values the operation may return when it is invoked. To capture the order of parameters, the component *arguments* of an operation is defined as a finite partial function from $\mathbb{N}$ to *Parameter* (the set of all formal parameters) the domain of which is the set $\{n \in \mathbb{N} \mid 1 \leq n \leq m\}$, for some non-negative integer m. Let the set of all such functions be denoted by seq(*Parameter*).

Parameter
name : *Name*
type : *ClassName*

Operation
name : *Name*
type : *ClassName*
arguments : seq(*Parameter*)
numOfArgs : $\mathbb{N}$

---

numOfArgs = #dom *arguments*

A *class* is then considered as an abstract entity that has as its components a name, a finite number of declared attributes and a finite number of declared operations.

Class
name : *ClassName*
declrAttrs : $\mathbb{F}$ *Attribute*
declrOpers : $\mathbb{F}$ *Operation*

Example 2 The class diagram in Figure 3 has four classes, i.e., c_1, c_2, c_3 and c_4, the names of which are *Person*, *Student*, *Instructor* and *Course*, respectively. The value of $c_1.declrAttrs$ is the set $\{attr_1, attr_2\}$, where the names of $attr_1$ and $attr_2$ are *name* and *addr*, respectively, and their types are *String*. The value of $c_1.declrOpers$ is the singleton set $\{op_1\}$. No attribute and operation is declared in c_2, whence both $c_2.declrAttrs$ and $c_2.declrOpers$ are the empty set. The value of the component *declrOpers* of the class c_3 is the set $\{op_2\}$, where $op_2.name$, $op_2.type$, $op_2.arguments$ and $op_2.numOfArgs$ are *getSalary*, *Money*, the mapping $\{(1, p_1)\}$, and the integer 1, respectively. The values *name* and *type* of the parameter p_1 are *y* and *Year*, respectively. ■

The notion of a well-formed class diagram will now be defined. A *well-formed class diagram* consists of a finite set, *classes*, of classes, a finite set, *associations*, of associations, a partial function, *superclass*, which defines superclass relationships, a partial function, *allsubs*, associating with a class the set of its subclasses, and a set, *top classes*, of the classes that are considered as the highest classes in the ontological classification taxonomy of the system being modeled. The schema for well-formed class diagrams is given below.

WFD_CD
classes : $\mathbb{F}$ *Class*
associations : $\mathbb{F}$ *Association*
superclass : *Class* $\rightarrow\!\!\!\!\!\!+$ *Class*
allsubs : *Class* $\rightarrow\!\!\!\!\!\!+$ $\mathbb{F}$ *Class*

$topclasses : \mathbb{F}\ Class$

$\forall c, c' : classes \bullet c \neq c' \Rightarrow c.name \neq c'.name$
$\forall c : topclasses \bullet$
$\quad (c \in classes \wedge c \notin \text{dom } superclass)$
$\forall c, c' : classes \bullet$
$\quad (c' \in allsubs(c) \Leftrightarrow superclass(c') = c)$
$\forall c : classes \bullet c \notin allsubs(c)$

The constraints of this schema ensure that each class has a unique name, each top class does not have any superclass, the partial functions *superclass* and *allsubs* are consistent with each other, and, furthermore, a class can be neither a superclass nor a subclass of itself (i.e., circular inheritance is not allowed).

Example 3 The class diagram in Figure 3 can be considered as a well-formed class diagram D_1, where the component *classes* of D_1 is the set $\{c_1, c_2, c_3, c_4\}$, the component *topclasses* is the set $\{c_1, c_4\}$, which means c_1 and c_4 are assumed to have no superclass, the component *associations* is the set $\{a_1, a_2\}$, the component *superclass* is the mapping $\{(c_2, c_1), (c_3, c_1)\}$, and the component *allsubs* is the mapping $\{(c_1, \{c_2, c_3\}), (c_2, \varnothing), (c_3, \varnothing), (c_4, \varnothing)\}$. ■

2.2 Well-Formed Sequence Diagrams

A sequence diagram describes an interaction arranged in time sequence. It specifies participating objects, their lifelines and the sequence of messages they exchange, but does not show the structural associations among the objects. Objects and messages are basic components of a sequence diagram. An *object* is an individual instance of some class. It has two components, i.e., *name* and *type*, which refers to its class.

Object
name : *ObjectName*
type : *Class*

The concept of *action* encompasses messages that are exchanged between objects. Two basic types of actions are considered, i.e., *action calls* and *return actions*.

$Action = ActionCall \cup ReturnAction$

ActionCall
source : *Object* ∪ *Actor*
target : *Object*
opName : *Name*
actualArgs : seq(*Object*)
numOfActArgs : ℕ

$numOfActArgs = \#\text{dom } actualArgs$

ReturnAction
source : *Object*
target : *Object* ∪ *Actor*
return : *Object*

An *action call* has *source*, *target*, *opName*, *actualArgs* and *numOfActArgs* as its components. The component *source* refers to the caller, which can either be an actor or an object, of the action, whereas the component *target* refers to the object that receives the call. The components *opName* and *actualArgs* refer to the name and the actual parameters (actual arguments), respectively, of the called operation. The component *actualArgs* of an action call and the component *arguments* of an operation have the same structure (see the schema *Operation*) except that an actual argument is an object rather than a formal parameter. A *return action* also has the components *source* and *target*, but instead of having an operation name and actual arguments, it has a returned object as its part.

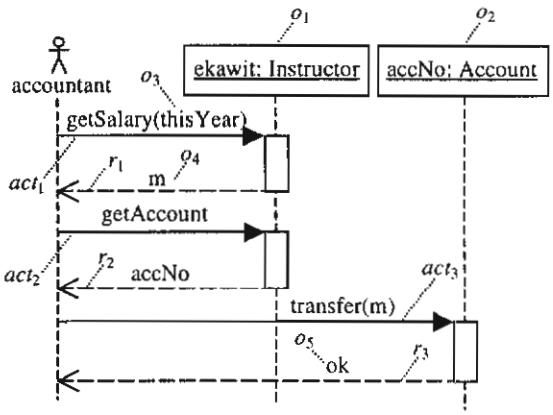

Figure 4: A Sequence Diagram

Example 4 Assume that the set *ObjectName* contains *ekawit*, *thisYear*, *m*, *accNo* and *ok*, the set *Actor* contains *accountant* and the set *Name* contains *getSalary*, *getAccount* and *transfer*. Consider the sequence diagram in Figure 4, in which two objects, i.e., o_1 and o_2, and one actor, i.e., *accountant*, participate. The objects o_1, for instance, has *ekawit* as its name and the class c_3 of Figure 3 as its type. There are three action calls in the diagram, i.e., act_1, act_2 and act_3, where the source of each of them is the actor *accountant*, their targets are o_1, o_1 and o_2, respectively, and their operation names are *getSalary*, *getAccount* and *transfer*, respectively. The values of *actualArgs* of o_1, o_2, and o_3 are the mappings $\{(1, o_3)\}$, $\varnothing$ and $\{(1, o_4)\}$, respectively. The diagram contains three return actions, i.e., r_1, r_2 and r_3, where, for example, $r_1.source$ is o_1, $r_1.target$ is *accountant* and $r_1.return$ is o_4. Observe that $r_1.return$ and $r_2.return$ are also used as the actual argument and the target object, respectively, of act_3. ■

A *well-formed sequence diagram* consists of a finite set of objects, a finite set of actors, a finite set of action calls, a finite set of return actions, a partial injective function that specifies the order of actions, and a partial injective function that associates with an action call its corresponding return action.

WFD_SD

$objects : \mathbb{F}\ Object$
$actors : \mathbb{F}\ Actor$
$calls : \mathbb{F}\ ActionCall$
$returns : \mathbb{F}\ ReturnAction$
$order : Action \rightarrowtail\!\!\!\!\!\mapsto \mathbb{N}$
$matchRet : ActionCall \rightarrowtail\!\!\!\!\!\mapsto ReturnAction$

$\mathrm{dom}\ order = (calls \cup returns)$
$\mathrm{ran}\ order = 1..\#(calls \cup returns)$
$\mathrm{dom}\ matchRet = calls$
$\mathrm{ran}\ matchRet = returns$
$\forall a : calls \bullet$
$\quad (a.source = matchRet(a).target$
$\quad \wedge\ a.target = matchRet(a).source$
$\quad \wedge\ a.order < matchRet(a).order$
$\quad \wedge\ a.source \in objects \cup actors$
$\quad \wedge\ a.target \in objects)$
$\forall r : returns \bullet$
$\quad (r.source \in objects$
$\quad \wedge\ r.target \in objects \cup actors)$

By the constraints of this schema, the sender of an action call must be the receiver of its matching return action, and, conversely, the sender of a return action must be the receiver of its matching action call. Moreover, an action call always occurs in time sequence before its matching return action.

Example 5 The sequence diagram in Figure 4 can be regarded as a well-formed sequence diagram D_2, where $D_2.objects$ is the set $\{o_1, o_2\}$, $D_2.actors$ is the singleton set $\{accountant\}$, the components *calls* and *returns* of D_2 are the sets $\{act_1, act_2, act_3\}$ and $\{r_1, r_2, r_3\}$, respectively, and the components *order* and *matchRet* of D_2 are the partial injections $\{(act_1, 1), (r_1, 2), (act_2, 3), (r_2, 4), (act_3, 5), (r_3, 6)\}$ and $\{(act_1, r_1), (act_2, r_2), (act_3, r_3)\}$, respectively. ■

As only class diagrams and sequence diagrams are considered in this paper, a well-formed diagram is either a well-formed class diagram or a well-formed sequence diagram. In the sequel, let *WFD* be the union of *WFD_CD* and *WFD_SD*.

3. Semantics

In classical logic, a well-formed formula has different interpretations in different possible worlds. Likewise, a well-formed UML diagram has many possible interpretations. The notion of a *set assignment* will be used to capture the concept of an interpretation in the context of UML. Under a set assignment, for example, a class has a set of object identities as its meaning, an association has a binary relation on object identifiers as its meaning. A set assignment also assigns possible meanings to objects, attributes and operations.

3.1 Preliminary

In the rest of the paper, the set *OId* of all object identifiers is assumed, and let the terms *Tuple*, *MapTuple*, *AllMapOIdTuple* and *TupleProjection* be defined using the following abbreviation definitions.

- $Tuple(X, n) == \{x_1, \ldots, x_n : X \bullet (x_1, \ldots, x_n)\}$
- $MapTuple(X, n) == Tuple(X, n) \rightarrow\!\!\!\!\!\mapsto X$
- $AllMapOIdTuple ==$
  $\{n : \mathbb{N} \upharpoonright n \geq 1 \bullet MapTuple(OId, n)\}$
- $TupleProjection(i, n, T) ==$
  $\{(x_1, \ldots, x_n) : T ; k : \mathbb{N} \mid (k = i \wedge k \leq n) \bullet x_k\}$

That is, *Tuple(X, n)* denotes the set of all *n*-tuples of elements of *X*; *MapTuple(X, n)* the set of all partial functions which maps an *n*-tuple of elements of *X* to some element of *X*; *AllMapOIdTuple* the collection comprising the sets *MapTuple(X, n)* for each positive integer *n*; and *TupleProjection(i, n, T)* the set consisting of the *i*th-element of each *n*-tuple in *T*.

3.2 Set Assignment

The schema *S* for set assignments is now defined.

S

$obj : Class \rightarrow\!\!\!\!\!\mapsto \mathbb{P}\ OId$
$links : Name \rightarrow\!\!\!\!\!\mapsto (OId \leftrightarrow OId)$
$attribute : Attribute \rightarrow\!\!\!\!\!\mapsto (OId \rightarrow\!\!\!\!\!\mapsto OId)$
$operation : Operation \rightarrow\!\!\!\!\!\mapsto \bigcup AllMapOIdTuple$
$id : Object \rightarrowtail\!\!\!\!\!\mapsto OId$

$\forall a : \mathrm{dom}\ attribute\ ; \exists c : \mathrm{dom}\ obj \bullet$
$\quad \mathrm{dom}\ attribute(a) = obj(c)$
$\forall op : \mathrm{dom}\ operation \bullet$
$\quad (operation(op) \in$
$\quad\quad MapTuple(OId, op.numOfArgs + 1)) \wedge$
$\quad (\exists c : \mathrm{dom}\ obj \bullet obj(c) =$
$\quad\quad TupleProjection(1, op.numOfArgs + 1,$
$\quad\quad\quad \mathrm{dom}\ operation(op)))$

The components *obj*, *links*, *attribute*, and *operation* of a set assignment provide interpretations of basic abstract components of a class diagram, i.e., classes, associations, attributes and operations, respectively, whereas the component *id* simply assigns a single object identifier to an object. Suppose that a set assignment *s* is given. A class *c* has the set *s.obj(c)* of object identifiers as its extension under *s*, and an association name *a* has as its meaning under *s* the binary relation *s.links(a)* on the set of object identifiers.

An attribute *attr* is interpreted by *s* as the partial function *s.attribute(attr)* associating with the identifier of each object *o* at most one object identifier, which will be regarded as the value of the attribute

attr of *o* under *s*. It is specified as a constraint of the schema that whenever *s.attribute*(*attr*) is defined, its domain must be the extension of some class *c*; and, consequently, the value of the attribute *attr* of each object belonging to such class *c* is defined under *s*.

An operation *op* with *n* parameters is interpreted by *s* as the partial function *s.operation*(*op*) associating at most one object identifier oid_r with each $(n+1)$-tuple $(oid_0, oid_1, \ldots, oid_n)$ of object identifiers, where oid_r is regarded as the value returned by the operation *op* when it is invoked with the actual parameters $oid_1, \ldots, oid_n$ on the host object identified by oid_0. The schema also requires that for every operation *op*, if the partial function *s.operation*(*op*) is defined, then the set of the identifiers of the host objects in its domain must be the extension of some class, and, as a result, every object in this class provides the operation *op*.

3.3 Components of Diagrams and Their Satisfactory Conditions

Intuitively, when the meaning of a diagram component α under a set assignment *s* conforms to some possible consistent instance of a model (of some system) containing α, the set assignment *s* will be considered to *satisfy* the component α, denoted by $s \models \alpha$. Referring to Figure 3, for example, if the meanings of the classes c_1 and c_2 under a set assignment *s* are sets O_1 and O_2, respectively, of object identifiers and O_1 includes O_2, then the set assignment *s* can be considered to satisfy the generalization relationship between c_1 and c_2.

In order to specify the precise satisfactory conditions for diagram components, the (free type) definition, *Component*, of the components of UML diagrams considered in this paper is first given.

$$
\begin{aligned}
&Component ::= \\
&\quad class\langle\!\langle Class \rangle\!\rangle \mid \\
&\quad top\langle\!\langle Class \rangle\!\rangle \mid \\
&\quad gen\langle\!\langle Class \times Class \rangle\!\rangle \mid \\
&\quad association\langle\!\langle Association \rangle\!\rangle \mid \\
&\quad declrAttribute\langle\!\langle Attribute \times Class \rangle\!\rangle \mid \\
&\quad availAttribute\langle\!\langle Attribute \times Class \rangle\!\rangle \mid \\
&\quad declrOperation\langle\!\langle Operation \times Class \rangle\!\rangle \mid \\
&\quad availOperation\langle\!\langle Operation \times Class \rangle\!\rangle \mid \\
&\quad classWfd\langle\!\langle WFD\_CD \rangle\!\rangle \mid \\
&\quad ptcpObj\langle\!\langle Object \rangle\!\rangle \mid \\
&\quad call\langle\!\langle ActionCall \rangle\!\rangle \mid \\
&\quad seqWfd\langle\!\langle WFD\_SD \rangle\!\rangle \mid \\
&\quad wfd\langle\!\langle WFD \rangle\!\rangle
\end{aligned}
$$

The relation $\models$ is then defined as a relation from *S* to *Component*. For any set assignment *s* and any component α, when $s \models \alpha$, *s* can be regarded as a *model* of α (in the sense of a model of a well-formed formula in classical logic). The satisfactory conditions for each element of *Component* will be described in the next two subsections.

3.4 Satisfactory Conditions for Class Diagrams

The satisfactory conditions for the components of a class diagram will now be given.

Class

$$
\begin{aligned}
&\_ \models \_ : S \leftrightarrow Component \\
&\hline \\
&\forall s : S;\ c : Class \bullet \\
&\qquad s \models class(c) \ \Leftrightarrow \ c \in \mathrm{dom}\ s.obj
\end{aligned}
$$

Generalization

$$
\begin{aligned}
&\forall s : S;\ c, c' : Class \bullet \\
&\qquad s \models gen(c, c') \ \Leftrightarrow \ s.obj(c) \subseteq s.obj(c')
\end{aligned}
$$

That is, given any classes *c* and *c'*, a set assignment *s* satisfies the component *class*(*c*) if and only if the extension of *c* under *s* is defined, and satisfies the component *gen*(*c*, *c'*) if and only if the extension under *s* of *c'* includes that of *c*.

Associations

$$
\begin{aligned}
&\forall s : S;\ r : Association \bullet \\
&\qquad s \models association(r) \ \Leftrightarrow \\
&\qquad (\mathrm{dom}(s.links(r.name)) \subseteq s.obj(r.e_1.class) \ \wedge \\
&\qquad\ \ \mathrm{ran}(s.links(r.name)) \subseteq s.obj(r.e_2.class)) \ \wedge \\
&\qquad (\forall o : s.obj(r.e_1.class) \bullet \\
&\qquad\qquad \#\{o' : s.obj(r.e_2.class) \mid \\
&\qquad\qquad\qquad (o, o') \in s.links\ (r.name)\} \\
&\qquad\qquad \in r.e_2.multiplicity) \ \wedge \\
&\qquad (\forall o' : s.obj(r.e_2.class\ ) \bullet \\
&\qquad\qquad \#\{o : s.obj(r.e_1.class) \mid \\
&\qquad\qquad\qquad (o, o') \in s.links(r.name)\} \\
&\qquad\qquad \in r.e_1.multiplicity)
\end{aligned}
$$

Intuitively, for any association (relationship) *r*, a set assignment *s* satisfies the component *association*(*r*) if and only if, under *s*, the association *r* only relates objects belonging to the classes indicated at its association ends, and the number of objects participating in the association conforms to the multiplicity specified at the association ends.

From the abstract syntax of class diagrams defined in Subsection 2.1, a class typically has a number of declared attributes and operations. In addition to these declared components, the class may have some other attributes and operations through inheritance. In order to precisely define the meanings of these internal components of a class, the notions of declared components and available components are introduced. A *declared attribute* of a class is an attribute that is declared explicitly in the class, whereas an *available attribute* of a class is an attribute that is either declared explicitly in the class or inherited from some ancestor of the class.

Available Attribute

$$
\begin{aligned}
&\forall s : S;\ c : Class;\ a : Attribute \bullet \\
&\quad s \models availAttribute(a, c) \Leftrightarrow \\
&\qquad (s.obj(c) \subseteq \mathrm{dom}\ s.attribute(a)) \ \wedge \\
&\qquad (\forall c' : Class \mid c'.name = a.type \bullet \\
&\qquad\qquad \mathrm{ran}\ s.attribute(a) \subseteq s.obj(c'))
\end{aligned}
$$

Declared Attribute

$$\forall s : S;\ c : Class;\ a : Attribute \bullet$$
$$s \models declrAttribute(a, c) \Leftrightarrow$$
$$(s \models availAttribute(a, c)) \wedge$$
$$(s.obj(c) = \text{dom } s.attribute(a))$$

Roughly speaking, for an attribute a and a class c, a set assignment s satisfies the components *availAttribute*(a, c), meaning that a is regarded as an available attribute of c under s, if the value of the attribute a of every object in the extension of c under s belongs to the extension under s of the type of a. Under the same condition except that every object the value of the attribute a of which is defined also belongs to the extension of c under s, the set assignment s satisfies the component *declrAttribute*(a, c). It follows directly that:

Proposition 1

$$\forall s : S;\ c : Class;\ a : Attribute \bullet$$
$$s \models declrAttribute(a, c)$$
$$\Rightarrow s \models availAttribute(a, c) \quad \blacksquare$$

Similarly, while a *declared operation* of a class is an operation that is declared explicitly in the class, an *available attribute* of a class is an operation that the class provides, which may be derived from an ancestor of the class by means of inheritance.

Available Operation

$$\forall s : S;\ c : Class;\ op : Operation \bullet$$
$$s \models availOperation(op, c) \Leftrightarrow$$
$$TupleProjection(1, op.numOfArgs + 1,$$
$$\text{dom } s.operation(op)) \supseteq s.obj(c) \wedge$$
$$(\forall i : \mathbb{N};\ c' : Class \mid$$
$$2 \le i \le op.numOfArgs + 1 \wedge$$
$$c'.name = op.arguments(i - 1).type \bullet$$
$$TupleProjection(i, op.numOfArgs + 1,$$
$$\text{dom } s.operation(op)) \subseteq s.obj(c')) \wedge$$
$$(\forall c'' : Class \mid c''.name = op.type \bullet$$
$$\text{ran } s.operation(op) \subseteq s.obj(c''))$$

Declared Operation

$$\forall s : S;\ c : Class;\ op : Operation \bullet$$
$$s \models declrOperation(op, c) \Leftrightarrow$$
$$(s \models availOperation(op, c)) \wedge$$
$$(TupleProjection(1, op.numOfArgs + 1,$$
$$\text{dom } s.operation(op)) = s.obj(c))$$

Intuitively, given an operation op and a class c, a set assignment s satisfies the component *availOperation*(op, c), if every object in the extension of c under s is a host object of op, and each possible actual argument belongs to the extension under s of the class of its corresponding formal parameter, and, moreover, each possible returned value belongs to the extension under s of the return type of op. Under the same condition except that every possible host object of op also belongs to the extension of c under s, the set assignment s satisfies the component *declrOperation*(op, c). The next proposition directly follows.

Proposition 2

$$\forall s : S;\ c : Class;\ op : Operation \bullet$$
$$s \models declrOperation(op, c)$$
$$\Rightarrow s \models availOperation(op, c) \quad \blacksquare$$

Next, given a class c, a set assignment s satisfies the component *top*(c), meaning that c can be considered as one of the highest classes in a classification taxonomy under s, if and only if there exists no other class the extension under s of which includes the extension of c under s.

Top Class

$$\forall s : S;\ c : Class \bullet\ s \models top(c) \Leftrightarrow$$
$$(\forall c' : \text{dom } s.obj \mid c' \neq c \bullet\ s.obj(c) \not\subset s.obj(c'))$$

Then, for any well-formed class diagram d, a set assignment s satisfies the component *classWfd*(d), if and only if it satisfies every component of d.

Class Diagrams

$$\forall s : S;\ d : WFD\_CD \bullet$$
$$s \models classWfd(d) \Leftrightarrow$$
$$(\forall c : d.classes \bullet s \models class(c)) \wedge$$
$$(\forall c : \text{dom } d.superclass \bullet$$
$$s \models gen(c, d.superclass(c))) \wedge$$
$$(\forall a : d.associations \bullet s \models association(a)) \wedge$$
$$(\forall c : d.classes;\ op : Operation \bullet$$
$$op \in c.declrOpers$$
$$\Rightarrow s \models declrOperation(op, c)) \wedge$$
$$(\forall c : d.topclasses \bullet s \models top(c))$$

3.5 Satisfactory Conditions for Sequence Diagrams

The satisfactory conditions for objects participating in a sequence diagram and for action calls will now be described.

Participating Object

$$\forall s : S;\ o : Object \bullet$$
$$s \models ptcpObj(o) \Leftrightarrow$$
$$(o.type \in \text{dom } s.obj) \wedge (o \in \text{dom } s.id) \wedge$$
$$(s.id(o) \in s.obj(o.type))$$

In plain words, for any object o, a set assignment s satisfies the component *ptcpObj*(o) when the identifier of o is consistent with the extension of its type under s. The next proposition follows directly.

Proposition 3

$$\forall s : S;\ o : Object \bullet$$
$$s \models ptcpObj(o) \Rightarrow s \models class(o.type) \quad \blacksquare$$

Next, consider the satisfactory conditions for action calls. Basically, a set assignment satisfies an action call, if and only if, under that set assignment, the class of the receiver of the call provides the operation of the call and, furthermore, the actual arguments of the call all conform to the signature of the operation. This is formally described as follows.

Call

$$\begin{array}{l}\forall s : S;\ act : ActionCall \bullet \\ s \models call(act) \Leftrightarrow \\ \quad (s \models ptcpObj(act.target)) \ \wedge \\ \quad (\exists\, op : Operation \bullet \\ \quad\quad (s \models availOperation(op, act.target.type)) \\ \quad\quad (op.name = act.opName) \ \wedge \\ \quad\quad (op.numOfArgs = act.numOfActArgs) \ \wedge \\ \quad\quad (\forall i : \mathbb{N} \mid 1 \le i \le op.numOfArgs \bullet \\ \quad\quad\quad (\exists\, c : Class \bullet \\ \quad\quad\quad\quad (op.arguments(i).type = c.name) \ \wedge \\ \quad\quad\quad\quad (s.id(act.actualArgs(i)) \in s.obj(c)))))\end{array}$$

Proposition 4, which will be used in the next section, follows readily.

Proposition 4

$$\begin{array}{l}\forall s : S;\ act : ActionCall \bullet \\ \quad s \models call(act) \ \Rightarrow \ \exists\, op : Operation \bullet \\ \quad\quad (s \models availOperation(op, act.target.type)) \wedge \\ \quad\quad (op.name = act.opName) \wedge \\ \quad\quad (op.numOfArgs = act.numOfActArgs) \quad \blacksquare\end{array}$$

Next, a set assignment s satisfies a well-formed sequence diagram d if and only if it satisfies every component of d; and, finally, a set assignment can satisfy well-formed class diagrams or well-formed sequence diagrams, and other kinds of diagrams are not discussed in this paper.

Sequence Diagrams

$$\begin{array}{l}\forall s : S;\ d : WFD\_SD \bullet \\ \quad s \models seqWfd(d) \Leftrightarrow \\ \quad\quad (\forall o : d.objects \bullet s \models ptcpObj(o)) \ \wedge \\ \quad\quad (\forall a : d.calls \bullet s \models call(a))\end{array}$$

Diagrams

$$\begin{array}{l}\forall s : S;\ d : WFD \bullet \\ \quad s \models wfd(d) \ \Leftrightarrow \ (s \models seqWfd(d) \vee s \models classWfd(d))\end{array}$$

4. Reasoning with UML Diagrams

Inference rules for deducing from a given set of UML diagrams some of their logical consequences and for proving their properties are presented in this section. Before developing such inference rules, what it means for one diagram to entail another diagram is precisely defined in Subsection 4.1. The notion of entailment is then used as a basis for verifying the soundness of the inference rules described in Subsection 4.2.

4.1 Entailment Relationship on UML Diagrams

In [2], the entailment relation, in symbols $\models_d$, between well-formed diagrams is defined as follows.

$$\begin{array}{l}\_ \models_d \_ : WFD \leftrightarrow WFD \\ \hline \forall D, D' : WFD \bullet \\ \quad D \models_d D' \ \Leftrightarrow \ (\forall s : S \bullet s \models wfd(D) \Rightarrow s \models wfd(D'))\end{array}$$

That is, one well-formed diagram entails another well-formed diagram, if and only if every set assignment satisfying the former also satisfies the latter. This definition of $\models_d$ will be used as a basis for proving the soundness of inference rules for deriving from a single diagram some of its implicit properties or components, e.g., the first three inference rules in Subsection 4.2.

By means of overloading, the entailment relation $\models_d$ will additionally be used in this paper as a relation that connects a pair of well-formed diagrams with another well-formed diagram. As formalized below, given two well-formed diagrams D and D', the pair (D, D') is considered to entail another well-formed diagram D'', if and only if every set assignment that satisfies both D and D' always satisfies D''.

$$\begin{array}{l}\_ \models_d \_ : (WFD \times WFD) \leftrightarrow WFD \\ \hline \forall D, D', D'' : WFD \bullet \\ \quad (D, D') \models_d D'' \ \Leftrightarrow \\ \quad (\forall s : S \bullet (s \models wfd(D) \wedge s \models wfd(D')) \\ \quad\quad \Rightarrow s \models wfd(D''))\end{array}$$

This extended relation $\models_d$ will be used as the grounds for justifying the soundness of rules for inferring some new diagram components from two existing diagrams, e.g., Rules 4 and 5 in the next subsection.

4.2 Inference Rules for UML Diagrams

Based on the concept of the entailment relation defined in Subsection 4.1, an inference rule R is said to be *sound* if either of the following two conditions is satisfied.

1) If R infers from a well-formed diagram D a well-formed diagram D', then $D \models_d D'$.
2) If R infers from well-formed diagrams D and D' a well-formed diagram D'', then $(D, D') \models_d D''$.

A number of inference rules will now be presented. The first rule can be considered as the inheritance mechanism for UML class diagrams.

Rule 1: (***Inheritance***) Each available attribute (or operation) of a class c is also an available attribute (or operation, respectively) of every subclass of the class c. ■

The soundness of Rule 1 follows directly from the next proposition.

Proposition 5

$$\begin{array}{l}1)\ \forall s : S;\ attr : Attribute;\ c, c' : Class \bullet \\ \quad\quad (s \models availAttribute(attr, c) \ \wedge \ s \models gen(c', c)) \\ \quad\quad\quad \Rightarrow s \models availAttribute(attr, c') \\ 2)\ \forall s : S;\ op : Operation;\ c, c' : Class \bullet \\ \quad\quad (s \models availOperation(op, c) \ \wedge \ s \models gen(c', c)) \\ \quad\quad\quad \Rightarrow s \models availOperation(op, c')\end{array}$$

Proof Let $s \in S$, $attr \in Attribute$ and $c, c' \in Class$ such that $s \models availAttribute(attr, c)$ and $s \models gen(c', c)$. Since $s \models availAttribute(attr, c)$, dom $s.attribute(attr)$ includes $s.obj(c)$. As $s \models gen(c', c)$, $s.obj(c)$ includes $s.obj(c')$. Thus dom $s.attribute(att) \supseteq s.obj(c')$. As a consequence, $s \models availAttribute(attr, c')$, and the first result holds. The second result of this proposition can be proven in a similar way. ■

The next two rules can be used for reasoning about available and declared attributes/operations of a class.

Rule 2: (*Deriving Available Attributes/Operations from Declared Attributes/Operations*) Each declared attribute (or operation) of a class c is also an available attribute (or operation, respectively) of the class c. ■

Rule 3: (*Deriving Declared Operations from Available Operations*) Each available operation of a class c that is not a subclass of any other class is also a declared operation of the class c. ■

The soundness of Rule 2 follows from Propositions 1 and 2, while that of Rule 3 follows immediately from the next proposition.

Proposition 6

$\forall s : S; op : Operation; c : Class \bullet$
$(s \models availOperation(op, c) \land s \models top(c))$
$\Rightarrow s \models declrOperation\ (op, c)$

Proof Let $s \in S$, $op \in Operation$ and $c \in Class$ such that $s \models availOperation(op, c)$ and $s \models top(c)$. Let O denote the set $TupleProjection(1, op.numOfArgs + 1, \text{dom } operation(op))$. As $s \models availOperation(op, c)$, $s.obj(c) \subseteq O$. By the constraints of the schema for S, there exists $c' \in$ dom obj such that $s.obj(c') = O$. Assume that $s.obj(c) \neq s.obj(c')$. Then $c \neq c'$ and $s.obj(c) \subset s.obj(c')$. But, as $s \models top(c)$, $s.obj(c) \not\subset s.obj(c')$, which is a contradiction. Hence $s.obj(c) = s.obj(c')$. It follows that $s.obj(c) = O$, and, as a result, $s \models declrOperation\ (op, c)$. ■

Some components of a class diagram can be inferred from a sequence diagram by the application of the next two inference rules.

Rule 4: (*Deriving Classes from Sequence Diagrams*) If c is the class of some object participating in a sequence diagram D and c does not exist in a class diagram D', then c can be added into the class diagram D'. ■

Rule 5: (*Deriving Available Operations from Sequence Diagrams*) If an action call of which the operation is *op* is invoked on an object of some class c in a sequence diagram D and c is a class in a class diagram D', then the operation *op* is an available operation of c in the class diagram D'. ■

The soundness of Rules 4 and 5 follow from Propositions 3 and 4, respectively.

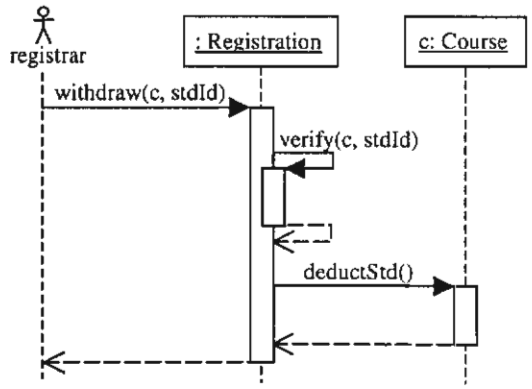

Figure 5: A Sequence Diagram

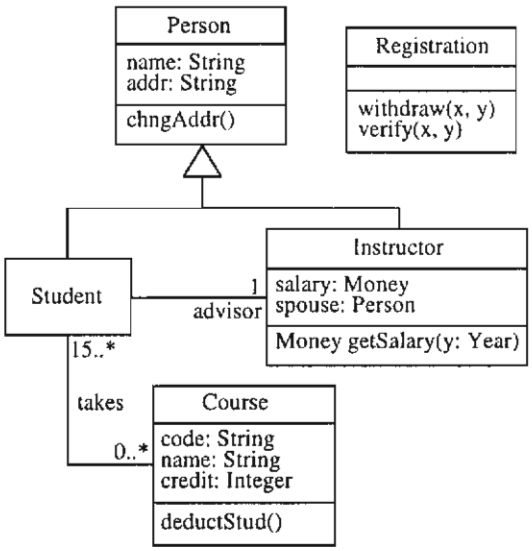

Figure 6: A Class Diagram

4.3 Example

This subsection illustrates the application of the inference rules presented in Subsection 4.2. Consider the class diagram in Figure 3 (in Subsection 1.1) and the sequence diagram in Figure 5. From these two diagrams, one can use Rule 4 to infer that there exists a class the name of which is *Registration*, and use Rule 5 in infer that this class has at least two available operations, i.e., *withdraw* and *verify*, each takes two arguments. Rule 5 can furthermore be applied to infer that *deductStud* is an available operation in the class *Course*. Then, as neither the class *Registration* nor the class *Course* has a superclass, one can infer that *withdraw* and *verify* are declared

operation of the class *Registration*, and *deductStd* is a declared operation of the class *Course*, using Rule 3. As a result, the class diagram in Figure 6 is derivable from the class diagram in Figure 3 and the sequence diagram in Figure 5. Now, from the class diagram in Figure 6, one can use Rule 2 to infer, for example, that the class *Person* has *name* and *addr* as available attributes and *chngAddr* as an available operation, and, then, use Rule 1 to infer that the classes *Student* and *Instructor* also have these available attributes and operation.

5. Concluding Remarks

After a formal semantics of UML class diagrams (including attribute and operation declarations) and sequence diagrams is developed, sound inference rules for reasoning with these two kinds of diagrams are proposed. The proposed inference rules are practically useful, for instance, for implementing computer-aided system modeling tools. In an early step of a model development process, a system analyst commonly uses a class diagram for visualizing the structural aspect of the system being modeled. Such a class diagram typically focuses solely on the static relationships among classes of objects in the problem domain, and the internal components of a class, such as the operations each class provides, are often left unspecified. Thereafter, in order to describe the dynamic behavior of the system, the system analyst usually uses sequence diagrams for specifying how objects in the system collaborate on performing tasks in various scenarios. By using the inference rules, such as Rules 3, 4 and 5, a modeling tool can make use of the information contained the sequence diagrams to automatically refine the class diagram, e.g., to declare necessary operations in classes. Other inference rules, such as Rules 1 and 2, can then be used, for example, for deriving implicit properties of diagram components and for checking the consistency of UML models.

The authors believe that the work reported in this paper provides a solid theoretical basis for the semantics of UML and for the construction of computer-aided software engineering tools. For example, using knowledge-based software engineering approach (e.g., [6]), the presented inferences rules can be encoded as part of the general knowledge on the domain of UML, which can then be used by some inference engine in order to make a UML model more complete and consistent. Furthermore, once the precise semantics of UML is firmly established, the mapping rules for transforming a UML model to some specific implementation language, such as Java or C++, can be accurately identified, and, consequently, forward engineering and reverse engineering UML models can be (at least partly) automated.

Acknowledgement

This work was supported by the Thailand Research Fund, under Grant No. PDF/31/2543.

References

[1] Booch, G., Jacobson, I. and Rumbaugh J., *The Unified Modeling Language User Guide*, Addison-Wesley, 1999.

[2] Evans A. S., *Reasoning with UML Class Diagrams*, Proc. 2nd IEEE Workshop on Industrial-Strength Formal Specification Techniques, Boca Raton, Florida, IEEE Press, 1998.

[3] Evans, A. S. and Kent, S., *Core Meta-Modeling Semantics of UML: The pUML Approach*, Lecture Notes in Computer Science, vol. 1723, pp. 140-155, Springer-Verlag, 1999.

[4] France, R., Evans, A. S., Lano, K. and Rumpe, B., *The UML as a Formal Modeling Notation*, Computer Standards and Interfaces, vol. 19, no. 7, pp. 325-334, Elsevier Science, 1998.

[5] Jacobson, I., Booch, G. and Rumbaugh, J., *The Unified Software Development Process*, Addison Wesley, 1999.

[6] Nantajeewarawat, E., Wuwongse, V., Anutariya, C., Akama, K. and Thiemjarus, S., *Towards Reasoning with UML Diagrams Based-on XML Declarative Description Theory*, Proc. International Conference on Intelligent Technologies, Bangkok, Thailand, 2000.

[7] Rumbaugh, J., Jacobson, I. and Booch, G., *The Unified Modeling Language Reference Manual*, Addison Wesley, 1999.

[8] The UML Group, *The Unified Modeling Language Notation Guide (version 1.1)*, http://www.rational.com/uml.

[9] The UML Group, *The Unified Modeling Language Semantics Document (version 1.1)*, http://www.rational.com/uml.

[10] The precise UML group (pUML), information available at http://www.cs.york.ac.uk/puml.

[11] Spivey, J. M., *The Z Notation – A Reference Manual*, Prentice Hall, 2nd Edition, 1992.

[12] Woodcock, J. and Davies, J., *Using Z Specification, Refinement and Proof*, Prentice Hall, 1996.

[13] Wordsworth, J. B., Software Development with Z - A Practical Approach to Formal Methods in Software Engineering, Addison-wesley, 1992.

Published in Tanprasert, T., editor, Proceedings of the Second International Conference on Intelligent Technologies (InTech'2001), Bangkok, Thailand, pages 240-249, November 2001. ISBN 974-615-068-5.

Generating Relational Database Schemas from UML Diagrams Through XML Declarative Descriptions

Ekawit Nantajeewarawat
IT Program
Sirindhorn Intl. Inst. of Tech.
Thammasat University
Pathumthani 12121, Thailand
E-mail: ekawit@siit.tu.ac.th

Vilas Wuwongse
CSIM Program
School of Advanced Tech.
Asian Institute of Technology
Pathumthani 12120, Thailand
E-mail: vw@cs.ait.ac.th

Surapa Thiemjarus
IT Program
Sirindhorn Intl. Inst. of Tech.
Thammasat University
Pathumthani 12121, Thailand
E-mail: st01@doc.ic.ac.uk

Kiyoshi Akama
Center for Information
and Multimedia Studies
Hokkaido University
Sapporo 060-0811, Japan
E-mail: akama@cims.hokudai.ac.jp

Chutiporn Anutariya
CSIM Program
School of Advanced Tech.
Asian Institute of Technology
Pathumthani 12120, Thailand
E-mail: ca@cs.ait.ac.th

Abstract: With strong support from leading system-modeling methodologists, academics and, most importantly, the Object Management Group (OMG), it comes as no surprise that the Unified Modeling Language (UML) is maturing into a de facto standard object-oriented language for modeling software-intensive systems. For a variety of reasons, e.g., compatibility with existing systems and databases, most object-oriented applications still rely upon a relational database management system despite their original object-centered designs. Integrating relational databases into object-oriented applications necessitates transformations from the structural parts of object-oriented models into relational database schemas. It is demonstrated in this paper that mapping rules for such transformations, which constitute an important part of general knowledge in the domain of UML, can be represented as XML definite clauses. Of central importance to this approach, such definite clauses use XML expressions as their underlying data structure; consequently, not only can they directly describe diagram components that are represented in XML Metadata Interchange format (XMI)—a standard XML-based interchange format for UML diagrams—in addition, they can seamlessly specify information to be extracted from the diagram components as well as new information to be derived.

Key words: UML, XMI, XML declarative descriptions (XDD), Knowledge representation, Knowledge-based systems, Object-oriented models, Relational models, Forward engineering, Database schemas

1 Introduction

The past decade saw rapid growth in the popularity of object-oriented (OO) software development. Notwithstanding some architectural inelegance, most OO applications are still employing relational databases as their persistent data repositories. Such practices arise from several reasons: compatibility with existing legacy systems, reliability and existing user-awareness of the relational database technology, and the simplicity, with sound mathematical foundation, of the relational model [7]. Irrespective of storage technology, the Unified Modeling Language (UML) [6, 13] has undoubtedly become the most widely-used standard notation for specifying, visualizing and documenting the artifacts of large-scale OO-based software systems.

This paper discusses a practical area in which the framework for knowledge representation in the domain of UML proposed in [11] is applicable; that is, automated database schemas generation. The framework is based on the concepts of XML specialization system and XML declarative description (XDD) [5, 14]. UML diagrams are represented in XML Metadata Interchange (XMI) format [16], a standard text-

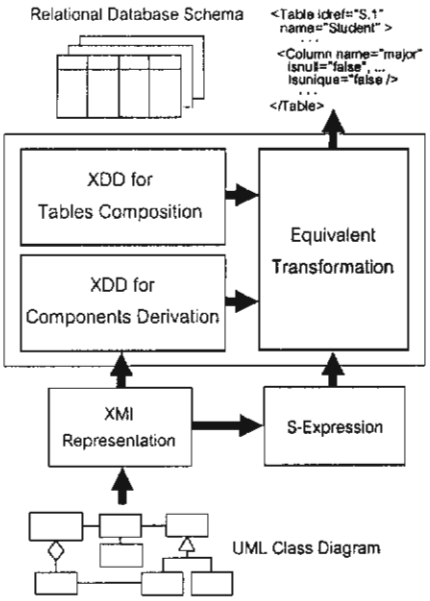

Figure 1: Overview of the framework

based representation for UML, which enhances the interoperability between UML supporting tools. General knowledge in the UML domain is represented as a set of XML definite clauses. Equivalent Transformation (ET) [3, 4] is employed as a computation foundation. Altogether, a knowledge base prototype has been built using Equivalent Transformation Interpreter (ETI), an ET-based reasoning engine recently developed at Hokkaido University, as its computation apparatus.

As outlined in Figure 1, in order to generate table schemas, represented in the XML format [9], from persistent classes and their associations, the components of a UML diagram are first converted into their XMI representations using some currently available software tool, such as Rational Rose, UCI's Argo/UML and IBM's XMI Toolkit. The general knowledge for constructing relational database schemas from UML class diagrams is divided into two layers: components derivation and tables composition. Not only does this two-layer architecture allow derived table components to be rendered in a variety of formats; it also makes the prototype system more amenable to extensions and modifications. The XMI representations are translated into s-expressions, while XML definite clauses representing the general knowledge are implemented as ET rules. The ETI engine operates on these procedural rules and s-expressions to generate table components and combine them in a required form.

The focus of this paper is on the employment of XML definite clauses in representing mapping rules for transforming the structural parts of UML models into relational database schemas. To start with, Section 2 summarizes such mapping rules. Section 3 illustrates XMI representations of UML class diagrams. It is followed by an informal review of XML definite clauses and XDD theory in Section 4. Section 5 shows how to represent the mapping rules and at the same time explains the use of XML definite clauses by means of practical examples. The conversion of XMI representations into s-expressions along with illustrations of ET-rules obtained from XML definite clauses is given in the appendix.

2 Transforming Class Diagrams into Relational Schemas

To bridge the gap between OO constructs and relational schemas, their interconnections have been studied extensively and variations of mapp-

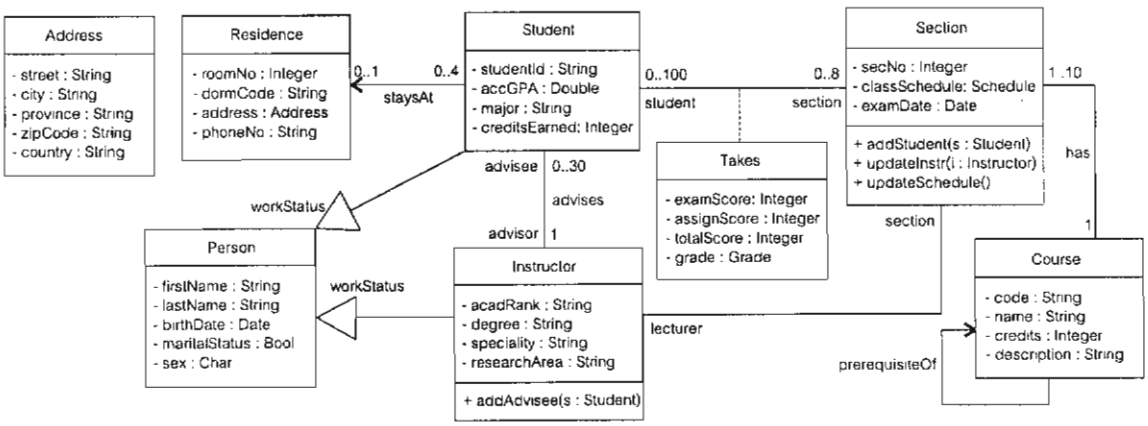

Figure 2: A UML class diagram

ings between UML class diagrams and database tables have been proposed [7, 8, 10]. Although only a set of selected widely used mapping rules is discussed in this paper, the presented approach is directly applicable to other mapping rules.

As a specification of the static design view of a system, a class diagram contains a collection of structural model elements, centering round the concept of class. There are various types of classes, not all of which should be materialized as part of a database schema. In general, entity classes are suitable candidates; regardless of their surroundings and applications, they model information and associated behavior that last long. In practice, classes of objects that will be stored in a database for future retrieval are often marked with the stereotype "persistent".

Classes, Association Classes, and Their Internal Components As a commonly used class-to-table mapping rule, a persistent class will be mapped into a table. However, not only are tables generated from persistent classes, they can also be created, as will be seen later, from associations of several kinds (e.g., ordinary associations, derived associations, and aggregations). For the sake of uniformity, a distinguished column named "ID" of type **Integer** will be used as the primary key of each generated table. With the assumption that primitive types are supported by most relational database systems, an attribute having a primitive type will simply be mapped into a column of that type in the table for its owner class. Association classes (e.g., Takes in Figure 2) will be treated as ordinary classes.

Associations and Aggregations An association will normally be mapped into a separate table; then, in order to refer to the objects connected across an association instance, the primary key of the table for the class at each endpoint of the association will be used as a foreign key in the table for the association. However, instead of generating a separate table, when the multiplicity at its navigable endpoint is not greater than one, a unidirectional association can be buried as a foreign key in an existing table—the table for the class at its non-navigable endpoint. Considering the class diagram in Figure 2, for example, the association staysAt can be buried in the table for Student as a foreign key referring to the table for Residence; likewise, the association has can be buried as a foreign key in the table for Section. An aggregation is regarded as a kind of association; therefore, it follows the same mapping rules.

Derived Associations An attribute having a non-primitive type will be transformed into an

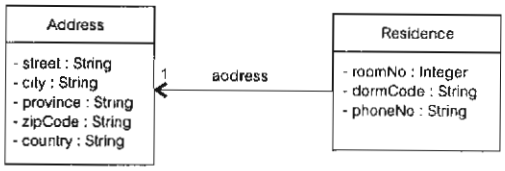

Figure 3: A derived association

```
<UML:Class xmi.id="S.7" name="Student"
  generalization="G.24">
  <UML:Namespace.ownedElement>
    <UML:Generalizaiton xmi.id="G.24"
      name="workingStatus" visibility="public"
      child="S.7" parent="S.1"/>
  </UML:Namespace.ownedElement>
  <UML:Classifier.feature>
    <UML:Attribute xmi.id="S.8"
      name="studentId" type="G.19">
      <UML:StructuralFeature.multiplicity>
        <UML:Multiplicity>
          <UML:Multiplicity.range>
            <UML:MultiplicityRange
              lower="1" upper="1"/>
          </UML:Multiplicity.range>
        </UML:Multiplicity>
      </UML:StructuralFeature.multiplicity>
      <UML:Attribute.initialValue>
        <UML:Expression body="0"/>
      </UML:Attribute.initialValue>
    </UML:Attribute>
    ...
  </UML:Classifier.feature>
</UML:Class>
```

Figure 4: A `Class`-element

association, called *derived association*, connecting its owner class with the non-primitive type. Such an association is always unidirectional; the non-primitive-type endpoint is navigable and the multiplicity at this endpoint follows the multiplicity of the attribute. For example, the attribute **address** of the class **Residence** in the class diagram in Figure 2 will be transformed into the derived association shown in Figure 3, provided that the multiplicity of this attribute is one. The mapping rules for usual associations apply to derived associations.

Generalizations There are three basic mapping approaches for generalization relationships: the *normal* approach, where a class and each of its subclasses are mapped to separate tables; the *many-subclass* approach, which eliminates the table for a superclass and replicates all attributes of the superclass in the table for each of its subclasses; the *one-superclass* approach, which brings all attributes of subclasses up to a superclass level. Adopted in this paper is the normal approach, as it straightforwardly harmonizes with other mapping rules.

3 XMI Representations of Class Diagrams: Examples

Each class in a class diagram is encoded in the XMI format as a Class-element, each association and each aggregation as an Association-element, each association class as an AssociationClass-element, each generalization as a Generalization-element, and each primitive type as a DataType-element. The namespace UML is used. Referring to the class diagram in Figure 2, for example, the class Student, the association advises and the association class Takes are represented by the XML elements in Figures 4, 5 and 6, respectively (using XMI version 1.1). The DataType-element representing the primitive type String is shown in Figure 7. Assuming that the identifier of the Class-element representing the class Person is S.1, the Generalization-element enclosed within the Class-element in Figure 4 indicates that Student is a subclass of Person. Each attribute of Student is represented by an Attribute-element; due to space constraints, only the element representing the attribute studentId is shown in Figure 4. Unless another value is specified in a class diagram, the multiplicity of an attribute, which is encoded as a Multiplicity-element, is assumed to be one.

Each of the two AssociationEnd-elements enclosed in the Association-element in Figure 5 represents one endpoint of the association advises and describes the adornments, e.g., role name,

```
<UML:Association xmi.id="G.7" name="advises"
  <UML:Association.connection>
    <UML:AssociationEnd xmi.id="G.8"
      name="advisor" isNavigable="true"
      aggregation="none" type="S.12">
      <UML:AssociationEnd.multiplicity>
        <UML:Multiplicity>
          <UML:Multiplicity.range>
            <UML:MultiplicityRange
              lower="1" upper="1"/>
          </UML:Multiplicity.range>
        </UML:Multiplicity>
      </UML:AssociationEnd.multiplicity>
    </UML:AssociationEnd>
    <UML:AssociationEnd xmi.id="G.9"
      name="advisee" isNavigable="true"
      aggregation="none" type="S.7">
      <UML:AssociationEnd.multiplicity>
        <UML:Multiplicity>
          <UML:Multiplicity.range>
            <UML:MultiplicityRange
              lower="0" upper="30"/>
          </UML:Multiplicity.range>
        </UML:Multiplicity>
      </UML:AssociationEnd.multiplicity>
    </UML:AssociationEnd>
  </UML:Association.connection>
</UML:Association>
```

Figure 5: An Association-element

```
<UML:AssociationClass xmi.id="S.30" name="Takes">
  <UML:Association.connection>
    <UML:AssociationEnd xmi.id="G.17"
      name="" isNavigable="true"
      aggregation="none" type="S.26">
        ...
    </UML:AssociationEnd>
    <UML:AssociationEnd xmi.id="G.18"
      name="theStudent" isNavigable="true"
      aggregation="none" type="S.7">
        ...
    </UML:AssociationEnd>
  </UML:Association.connection>
  <UML:Classifier.feature>
    <UML:Attribute xmi.id="S.31"
      name="grade" type="G.19">
        ...
    </UML:Attribute>
      ...
  </UML:Classifier.feature>
</UML:AssociationClass>
```

Figure 6: An AssociationClass-element

```
<UML:DataType xmi.id="G.19" name="String"
  visibility="public" isRoot="false"
  isLeaf="false" isAbstract="false"
  isSpecification="false"/>
```

Figure 7: A DataType-element

navigability and multiplicity, of the association at that endpoint. For instance, the second AssociationEnd-element indicates that Student is at one endpoint of advises by referring to the identifier S.7 through the attribute type, and describes the navigability and multiplicity of advises at this endpoint using the attribute isNavigable and a Multiplicity-element, respectively. The attribute aggregation of an AssociationEnd-element specifies whether the endpoint it represents is an aggregate.

Since an association class, e.g., Takes in Figure 2, is regarded as both a class and an association, the structure of the AssociationClass-element representing it, e.g., the element in Figure 6, subsumes the structure of a Class-element and that of an Association-element. More examples of XMI representations of UML diagrams, including interaction diagrams, are provided in [11].

4 XML Declarative Descriptions: An Informal Review

XML declarative description (XDD) theory [5, 14] is developed based on Akama's theory of declarative descriptions [1]—an axiomatic theory that has provided a general template for discussing the semantics of definite-clause-style declarative descriptions in a wide variety of data domains, including typed feature terms [12] and

conceptual graphs [15]. In XDD theory, the ordinary well-formed XML elements [9] are extended by incorporation of variables. Such extended XML elements are called *XML expressions*. A variable has a dual function: it denotes a specialization wildcard (i.e., a variable can be specialized into an XML expression or a part thereof) and, at the same time, behaves as an equality constraint (i.e., any occurrence of a variable within the same scope must be specialized in the same way). Five disjoint classes of variables, with different syntactical usage and specialization characteristics, are employed: N-variables (*name*-variables), S-variables (*string*-variables), P-variables (*attribute-value-pair*-variables), E-variables (*XML-expression*-variables), and I-variables (*intermediate-expression*-variables). An N-variable is assumed to be prefixed with "`$N:`" and can only be instantiated into either a tag name or an attribute name; an S-variable prefixed with "`$S:`" and instantiated into a string; a P-variable prefixed with "`$P:`" and instantiated into zero or more attribute-value pair(s); an E-variable prefixed with "`$E:`" and instantiated into zero or more XML expression(s); finally, an I-variable prefixed with "`$I:`" and instantiated into a part of an XML expression of some specified pattern. Conventional well-formed XML elements are regarded as variable-free XML expressions, called *ground* XML expressions.

An *XML definite clause* C is an expression of the form

$$H \leftarrow B_1, \ldots, B_m, \beta_1, \ldots, \beta_n,$$

where $m, n \geq 0$, H and the B_i are XML-expressions, and each of the β_j is a predefined constraint, whose satisfaction is independent of any XML definite clause and is determined in advance. The XML expression H and the set $\{B_1, \ldots, B_m, \beta_1, \ldots, \beta_n\}$ are called, respectively, the head and the body of C. When its body is the empty set, C will be referred to as an *XML unit clause* and the symbol '$\leftarrow$' will often be omitted. An XML definite clause will also be called a definite clause or simply a clause, provided that no confusion is caused. Figure 8 illustrates a simple XML definite clause, where the *member*-expression in its body is a predefined constraint.

The scope of a variable is a single XML definite clause. For the sake of readability, a variable that specifies an equality constraint, i.e., a variable with more than one occurrence in a clause (such as `$S:CId` and `$N:Tag` in Figure 8), will be underlined. The Prolog notation for anonymous variables is adopted; i.e., a variable suffixed with the symbol '?' (such as `$E:?` and `$P:?` in Figure 8) is regarded as an anonymous variable (different occurrences of which are always considered to be unrelated).

An *XML declarative description (XDD)* is a set of XML definite clauses. By means of examples, the usage of variables and XML definite clauses will be explained from a practical viewpoint in the next section. For theoretical details of XDD theory, including the precise specialization operation on XML expressions and the formal semantics of XDDs, the reader is referred to [5, 11, 14].

5 Representing Transformation Rules as XML Definite Clauses

The XML elements representing diagram components, e.g., those in Figures 4, 5, 6 and 7, will be regarded as XML unit clauses. The use of XML (non-unit) definite clauses in describing the mapping rules discussed in Section 2 will now be demonstrated.

5.1 Deriving Table Components

From Classes and Association Classes The clause C_{TN1} in Figure 8 specifies that a table name can be derived from either a class or an association class. The `$N:Tag`-expression in its body can match a ground `Class`-element or `AssociationClass`-element, say E_C, by instantiating the N-variable `$N:Tag` into the tag name `UML:Class` or the tag name `UML:AssociationClass`; the S-variables `$S:CId` and `$S:Nm` into the identifier and the name, respectively, of E_C; the anonymous P-variable `$P:?` into zero or more attribute-value pair(s) of E_C; and the anonymous E-variable `$E:?` into zero or more immediate subelement(s) of E_C. By specifying the list of the two tag names as its second argument, the *member*-constraint in the body of C_{TN1} disallows any instantiation of `$N:Tag` into any other tag name. Once the body matches the ground element E_C, a `TableName`-element is derived, along with the identifier and the name of E_C as its reference and its name, respectively. For instance, by specializing the body of C_{TN1} into the `Student`-element in Figure 4, the element `<TableName idref="S.7" name="Student"/>` is obtained.

```
<dd:TABLENAME>
  <TableName idref=$S:CId name=$S:Nm/>
</dd:TABLENAME>

← <dd:FACT>
     <$N:Tag xmi.id=$S:CId name=$S:Nm $P:?> $E:?
     </$N:Tag>
  </dd:FACT>,
  member($N:Tag,[UML:Class,UML:AssociationClass])
```

Figure 8: Clause C_{TN1}, Generating table names from classes or association classes

```
<dd:COLUMN>
  <Column idref=$S:CId name=$S:ANm type=$S:TNm/>
</dd:COLUMN>
```

```
← <dd:FACT>
    <$N:Tag xmi.id=$S:CId $P:?>
      <$I:1>
        <UML:Attribute
          name=$S:ANm type=$S:AType $P:?> $E:?
        </UML:Attribute>
      </$I:1> $E:?
    </$N:Tag>
  </dd:FACT>,
  <dd:FACT>
    <UML:DataType xmi.id=$S:AType
      name=$S:TNm $P:?> $E:?
    </UML:DataType>
  </dd:FACT>,
  member($N:Tag, [UML:Class, UML:AssociationClass])
```

Figure 9: Clause C_{CL1}, Generating columns from attributes with primitive types

From Attributes The clause C_{CL1} in Figure 9 maps an attribute with a primitive type into a column of the table for its owner class (or association class). The body of this clause refers to a `Class`-element or an `AssociationClass`-element, say E_C, and a `DataType`-element, say E_D. To specify that E_C contains an `Attribute`-element, say E_A, representing an attribute with a primitive type, an equality constraint between the type of E_A and the identifier of E_D is imposed using the S-variable `$S:AType`. When such elements E_C, E_D and E_A are found, the clause C_{CL1} generates a `Column`-element with the identifier of E_C, the name of E_A and the name of E_D as its reference, its name and its type name, respectively, through the S-variables `$S:CId`, `$S:ANm` and `$S:TNm`.

This clause also illustrates an application of another kind of variable—I-variable. The I-variable `$I:1` is used in the body of C_{CL1} to form a generic expression, i.e., `$I:1`-expression, which can be specialized into any XML element containing as its (not necessarily immediate) subelement an `Attribute`-element of the pattern specified by the enclosed `Attribute`-expression. As an illustration, the `$N:Tag`-expression, enclosing the `$I:1`-expression, can be instantiated into the `Class`-element in Figure 4; then, since the `DataType`-expression in the body matches the `DataType`-element in Figure 7, the clause C_{CL1} yields among others the element `<Column idref="S.7" name="studentId" type="String"/>`.

An attribute with a non-primitive type will not be transformed into a column directly, but into a derived association, which will then be treated virtually as an ordinary association. Transformation of such an attribute is described by the clause C_{DA} in Figure 10. The variable `$S:AType` in its body specifies that this clause is active when the type of an `Attribute`-element, say E_A, is the identifier of some `Class`-element, say E_C, which means the type of E_A is non-primitive. When active, the clause generates a derived binary association with the class (or association class) to which the attribute represented by E_A belongs as one endpoint and the class represented by E_C as the other endpoint. The multiplicity at the former endpoint is unspecified ("-1" denotes an unbounded upper limit), while that at the latter follows the `Multiplicity`-element enclosed within E_A. Moreover, the latter endpoint is navigable, whereas the former is not.

```
<dd:DERIVED:ASSOCIATION>
  <DERIVED:Association idref=$S:AId name=$S:ANm>
    <AssociationEnd type=$S:CId
      name=$S:CNm isNavigable="false" $P:?>
      <Multiplicity lower="0" upper="-1"/>
    </AssociationEnd>
    <AssociationEnd type=$S:AType name=$S:ANm
      attributeType="true" isNavigable="true">
      <Multiplicity
        lower=$S:Lower upper=$S:Upper/>
    </UML:AssociationEnd>
  </DERIVED:Association>
</dd:DERIVED:ASSOCIATION>
```

```
← <dd:FACT>
    <$N:Tag xmi.id=$S:CId name=$S:CNm $P:?>
      <$I:1>
        <UML:Attribute xmi.id=$S:AId
          name=$S:ANm type=$S:AType $P:?>
          <$I:2>
            <UML:Multiplicity.Range
              lower=$S:Lower upper=$S:Upper/>
          </$I:2> $E:?
        </UML:Attribute>
      </$I:1> $E:?
    </$N:Tag>
  </dd:FACT>,
  <dd:FACT>
    <UML:Class xmi.id=$S:AType $P:?> $E:?
    </UML:Class>
  </dd:FACT>,
  member($N:Tag, [UML:Class, UML:AssociationClass])
```

Figure 10: C_{DA}, Deriving associations from attributes with non-primitive types

From Associations As a special case, a unidirectional association with the multiplicity at its navigable endpoint not greater than one will be mapped into a foreign key in the table for the class at its non-navigable endpoint, rather than transformed into a separate table. Derivation of such a foreign key is described by the clause C_{CL2} in Figure 11. The body of C_{CL2} specifies the pattern of an `Association`-element, say E_{Ass}, one `AssociationEnd`-subelement of

```
<dd:COLUMN>
  <Column idref=$S:CId1 name=$S:AssocNm
    type="Integer" reference=$S:CNm2
    isnull=$S:IsNull />
</dd:COLUMN>
← <dd:FACT>
    <UML:Association name=$S:AssocNm $P:?> $E:?
      <UML:Association.connection>
        <UML:AssociationEnd type=$S:CId2
          isNavigable="true" $P:?>
          <$I:1>
            <UML:MultiplicityRange
              lower=$S:Lower1 upper="1"/>
          </$I:1> $E:?
        </UML:AssociationEnd>
        <UML:AssociationEnd type=$S:CId1
          isNavigable="false" $P:?> $E:?
        </UML:AssociationEnd>
      </UML:Association.connection> $E:?
    </UML:Association>
  </dd:FACT>,
  <dd:FACT>
    <UML:Class xmi.id=$S:CId2 name=$S:CNm2 $P:?>
      $E:?
    </UML:Class>
  <dd:FACT>,
  isnull($S:Lower1,$S:IsNull)
```

Figure 11: C_{CL_2}, Burying unidirectional associations with suitable multiplicity as foreign keys

```
<dd:TABLENAME>
  <TableName idref=$S:AssocId name=$S:AssocNm/>
</dd:TABLENAME>
← <dd:FACT>
    <UML:Association xmi.id=$:S:AssocId
      name=$S:AsocNm $P:?> $E:?
      <UML:Association.connection>
        <UML:AssociationEnd
          isNavigable=$S:Nv1 $P:?>
          <$I:1>
            <UML:MultiplicityRange
              upper=$S:Upper1 $P:?/>
          </$I:1> $E:?
        </UML:AssociationEnd>
        <UML:AssociationEnd
          isNavigable=$:Nv2 $P:?>
          <$I:2>
            <UML:MultiplicityRange
              upper=$S:Upper2 $P:?/>
          </$I:2> $E:?
        </UML:AssociationEnd>
      </UML:Association.connection> $E:?
    </UML:Association>
  </dd:FACT>,
  seperateTable($S:Nv1,$S:Nv2,$S:Upper1,$S:Upper2)
```

Figure 12: C_{TN_2}, Generating table names from associations to which C_{CL_2} is inapplicable

which, say E_{E_1}, represents a navigable endpoint with the multiplicity upper limit bounded to one, while the other `AssociationEnd`-subelement of which, say E_{E_2}, represents a non-navigable endpoint. When such an `Association`-element E_{Ass} is found, a `Column`-element, say E_{Col}, is derived.

```
<dd:COLUMN>
  <Column idref=$S:AssocId name=$S:End1Nm
    type="Integer" reference=$S:CNm
    isnull="false"/>
</dd:COLUMN>
← <dd:FACT>
    <UML:Association xmi.id=$S:AssocId $P:?> $E:?
      <UML:Association.connection>
        <UML:AssociationEnd name=$S:End1Nm
          isNavigable=var$S:Nv1 type=$S:CId $P:?>
          <$I:1>
            <UML:MultiplicityRange
              upper=$S:U1 $P:?>
          </$I:1> $E:?
        </UML:AssociationEnd>
        <UML:AssociationEnd
          isNavigable=$S:Nv2 $P:?>
          <$I:2>
            <UML:MultiplicityRange
              upper=$S:U2 $P:?/>
          </$I:2> $E:?
        </UML:AssociationEnd>
      </UML:Association.connection> $E:?
    </UML:Association>
  </dd:FACT>,
  <dd:FACT>
    <UML:Class xmi.id=$S:CId
      name=$S:CNm $P:?> $E:?
    </UML:Class>
  </dd:FACT>,
  seperateTable($S:Nv1,$S:Nv2,$S:U1,$S:U2)
```

Figure 13: C_{CL_3}, Generating columns from associations to which C_{CL_2} is inapplicable

The element E_{Col} adopts the name of E_{Ass}, and refers to the type of E_{E_2}, meaning that it represents a column of the table for the class at the endpoint represented by E_{E_2}. Furthermore, it makes a reference to the name of the class at the endpoint represented by E_{E_1}, which is also the name of the table generated from this class; this means the column represented by E_{Col} is a foreign key referring to this table. Tested by the *isnull*-constraint in the body of C_{CL_2}, if the lower bound of the multiplicity at the endpoint described by E_{E_1} is zero, then the column represented by E_{Col} may have the null value.

Other associations will be directly transformed into separate tables. Their transformations into table names and columns are described by the clauses C_{TN_2} in Figure 12 and C_{CL_3} in Figure 13. The 4-ary constraint predicate *seperateTable* is used for examining whether the navigability and multiplicity of the two endpoints of an association do not satisfy the condition required by the clause C_{CL_2}. That is, given any strings v_1 and u_1 indicating, respectively, the values of the navigability and multiplicity upper bound of one `AssociationEnd`-element and any strings v_2 and u_2 indicating the corresponding values of another `AssociationEnd`-element,

```
<dd:COLUMN>
  <Column idref=$S:ParentId
    name=$S:GenralizationNm type="char"/>
</dd:COLUMN>

← <dd:FACT>
     <$I:1>
       <UML:Generalization
         name=$S:GenralizationNm
         parent=$S:ParentId $P:?/>
     </$I:1> $E:?
   </dd:FACT>
```

Figure 14: C_{CL4}, Generating additional columns for parent classes

```
<dd:COLUMN>
  <Column idref=$S:ChildId name=$S:ColumnNm
    type="Integer" isunique="true"
    reference=$S:ParentNm/>
</dd:COLUMN>

← <dd:FACT>
     <$I:1>
       <UML:Generalization child=$S:ChildId
         parent=$S:ParentId $P:?/>
     </$I:1> $E:?
   </dd:FACT>,
   <dd:FACT>
     <UML:Class xmi.id=$S:ParentId
       name=$S:ParentNm $P:?> $E:?
     </UML:Class>
   </dd:FACT>
   concat($S:ParentNm,"ID",$S:ColumnNm),
```

Figure 15: C_{CL5}, Generating additional columns for child classes

$seperateTable(v_i, v_j, u_i, u_j)$ is not satisfied if and only if v_i = "true", u_i = "1" and v_j = "false" for some $i, j \in \{1, 2\}$ such that $i \neq j$. The identifier and the name of an `Associatation`-element into which the `Associatation`-expression in the body of C_{TN2} is instantiated will be used as the reference and the name, respectively, of the generated `TableName`-element. As detailed by the clause C_{CL3}, from each endpoint of an association that is mapped into a separate table, a column of that table will be generated.

XML definite clauses for generating table components from derived associations obtained from the clause C_{DA} in Figure 10 are similar to those for handling ordinary associations, and are omitted due to space limitations.

From Generalizations Following the normal approach for dealing with generalization relationships, a table will be generated for each class. The components of such a table are generated in a straightforward way using the clauses C_{TN1} in Figure 8 and C_{CL1} in Figure 9. Furthermore, an additional column will be added into the table for a parent class; this column is used for indicating a specific class of an object described by each record in that table. On the other hand, the table for each child class will have an additional column used as the foreign key referring to the table for its parent class. Generation of such a column for a parent class and that for a child class are specified by the clauses C_{CL4} in Figure 14 and C_{CL5} in Figure 15, respectively, where the 3-ary constraint predicate *concat* in the body of C_{CL5} yields as its third argument the concatenation of any two strings given as its first two arguments.

5.2 Combining Table Components

Out of their components, e.g., table names and columns, generated through the clauses presented in the preceding subsection, in order to construct XML representations of derivable tables, XML definite clauses will be extended with the concept of set-aggregate. A set-aggregate used in this paper is an expression of the form

```
<dd:Aggregate>
  <set> Σ </set>
  <pattern> E_P </pattern>
</dd:Aggregate>,
```

where $E_{\mathcal{P}}$ is an XML expression specifying the pattern of XML elements of interest and Σ a collection of all derivable elements of the specified pattern $E_{\mathcal{P}}$. As an illustration, consider the extended XML definite clause C_{TB} in Figure 16. For each derived `TableName`-element E_N, the set-aggregate in the body of C_{TB} collects all derivable `Column`-elements that refers to the element E_N; then, the clause C_{TB} generates a `Table`-element, say E_T, that adopts the name of E_N and contains all the collected `Column`-elements together with another `Column`-element representing the special column "ID", used as the primary

```
<dd:TABLE>
  <Table idref=$S:TabRef name=$S:TabNm>
    <Column name="ID" isprimary="true"
      isunique="true" isnull="false"
      type="Integer"/>
    $E:AllDerivedColumns
  </Table>
</dd:TABLE>

← <dd:TABLENAME>
     <TableName idref=$S:TabRef name=$S:TabNm/>
   <dd:TABLENAME>,
   <dd:Aggregate>
     <set>$E:AllDerivedColumns</set>
     <pattern>
       <dd:COLUMN>
         <Column idref=$S:TabRef $P:1/>
       </dd:COLUMN>
     </pattern>
   </dd:Aggregate>
```

Figure 16: C_{TB}, Constructing a table