ĐẠI HỌC QUỐC GIA HÀ NỘI
TRƯỜNG ĐẠI HỌC CÔNG NGHỆ

LÊ VĂN VINH

NGHIÊN CỨU CÁC KỸ THUẬT BIỂU DIỄN VÀ CHUYỂN ĐỔI MÔ HÌNH CHO THIẾT KẾ HƯỚNG MIỀN

LUẬN ÁN TIẾN SĨ KỸ THUẬT PHẦN MỀM

Hà Nội - 2026

ĐẠI HỌC QUỐC GIA HÀ NỘI
TRƯỜNG ĐẠI HỌC CÔNG NGHỆ

LÊ VĂN VINH

NGHIÊN CỨU CÁC KỸ THUẬT BIỂU DIỄN VÀ CHUYỂN ĐỔI MÔ HÌNH CHO THIẾT KẾ HƯỚNG MIỀN

Chuyên ngành: Kỹ thuật phần mềm

Mã số: 9480103

LUẬN ÁN TIẾN SĨ KỸ THUẬT PHẦN MỀM

NGƯỜI HƯỚNG DẪN KHOA HỌC:

PGS.TS. Đặng Đức Hạnh

Hà Nội - 2026

LỜI CAM ĐOAN

Tôi xin cam đoan luận án **"Nghiên cứu các kỹ thuật biểu diễn và chuyển đổi mô hình cho thiết kế hướng miền"** là công trình nghiên cứu của riêng tôi. Các số liệu, kết quả được trình bày trong luận án là hoàn toàn trung thực và chưa từng được công bố trong bất kỳ một công trình nào khác.

- Tôi đã trích dẫn đầy đủ các tài liệu tham khảo, công trình nghiên cứu liên quan ở trong nước và quốc tế. Ngoại trừ các tài liệu tham khảo này, luận án hoàn toàn là công việc của riêng tôi.
- Trong các công trình khoa học được công bố trong luận án, tôi đã thể hiện rõ ràng và chính xác đóng góp của các đồng tác giả và những gì do tôi đã đóng góp.
- Luận án được hoàn thành trong thời gian tôi làm Nghiên cứu sinh tại Bộ môn Công nghệ phần mềm, Khoa Công nghệ thông tin, Trường Đại học Công nghệ, Đại học Quốc gia Hà Nội.

Tác giả: _____

Hà Nội: _____

LỜI CẢM ƠN

Trước hết, tôi muốn bày tỏ sự biết ơn đến cán bộ hướng dẫn PGS.TS. Đặng Đức Hạnh, người đã hướng dẫn, khuyến khích, truyền cảm hứng, chỉ bảo và tạo cho tôi những điều kiện tốt nhất từ khi bắt đầu làm nghiên cứu sinh đến khi hoàn thành luận án này.

Tôi xin chân thành cảm ơn các thầy cô giáo khoa Công nghệ thông tin, Trường Đại học Công nghệ, Đại học Quốc gia Hà Nội, đặc biệt là các Thầy Cô trong Bộ môn Công nghệ Phần mềm đã tận tình đào tạo, cung cấp cho tôi những kiến thức vô cùng quý giá, đã tạo điều kiện tốt nhất cho tôi về môi trường làm việc trong suốt quá trình học tập và nghiên cứu.

Tôi xin trân trọng cảm ơn Phòng Đào tạo, Phòng Công tác sinh viên và Ban giám hiệu trường Đại học Công nghệ đã tạo điều kiện thuận lợi cho tôi trong suốt quá trình thực hiện luận án.

Tôi cũng bày tỏ sự biết ơn đến Trường Đại học Sư phạm Kỹ thuật Vinh đã tạo điều kiện về thời gian và tài chính cho tôi thực hiện luận án này. Tôi muốn cảm ơn đến tập thể, các cán bộ giảng viên Trường Đại học Sư phạm Kỹ thuật Vinh đã cổ vũ, động viên và tạo điều kiện về thời gian cho tôi trong suốt quá trình nghiên cứu.

Tôi muốn cảm ơn đến tất cả những người bạn, các anh chị em nghiên cứu sinh và những người đồng nghiệp của tôi. Những người đã luôn chia sẻ với tôi những khó khăn và động viên giúp đỡ tôi bất cứ khi nào tôi cần và tôi luôn ghi nhớ điều đó.

Cuối cùng, tôi xin bày tỏ lòng biết ơn vô hạn tới gia đình, đặc biệt là vợ và các con, những người luôn ủng hộ và yêu thương tôi vô điều kiện. Nếu không có sự động viên và hậu thuẫn đó, tôi không thể hoàn thành luận án này.

NCS. Lê Văn Vinh

TÓM TẮT

Thiết kế hướng miền (*Domain-Driven Design – DDD*) đã nổi lên như một phương pháp nổi bật nhằm giải quyết các thách thức liên quan đến sự gia tăng về độ phức tạp, quy mô và tính không đồng nhất của các hệ thống phần mềm hiện đại. Phương pháp DDD thực hiện phát triển lặp lại dựa trên một mô hình miền giàu ngữ nghĩa, trong đó mô hình hóa lô-gic cốt lõi và các quy tắc của miền vấn đề. DDD sử dụng nhất quán ngôn ngữ chung, qua đó tăng cường giao tiếp hiệu quả, sự hiểu biết chung giữa chuyên gia miền và lập trình viên trong suốt vòng đời phần mềm. Đã có nhiều nghiên cứu về các kỹ thuật biểu diễn mô hình miền bằng ngôn ngữ chuyên biệt miền (*Domain-Specific Language – DSL*). Các nghiên cứu này sử dụng DSL để liên kết chặt chẽ mô hình thiết kế với quá trình triển khai, qua đó nâng cao tính chính xác, khả năng bảo trì và mở rộng của phần mềm. Các nghiên cứu gần đây theo nguyên lý của DDD tập trung vào việc sử dụng các ngôn ngữ chuyên biệt miền dựa trên chú thích, kết hợp với các kỹ thuật chuyển đổi mô hình nhằm sinh tự động phần mềm. Tuy nhiên, các nghiên cứu trước đây vẫn chưa mô tả một cách rõ ràng và thống nhất các khía cạnh hành vi trong mô hình miền, đồng thời gặp hạn chế trong việc đặc tả và tích hợp các ràng buộc phức tạp. Bên cạnh đó, các mô hình miền thực thi hiện nay còn thiếu một cơ sở ngữ nghĩa hình thức để kiểm chứng tính đúng đắn của chúng, và chưa có một phương pháp hoàn chỉnh cho bộ chuyển đổi mô hình từ đặc tả yêu cầu sang mã nguồn có thể thực thi.

Nhằm giải quyết các thách thức về kỹ thuật biểu diễn và chuyển đổi mô hình cho thiết kế hướng miền, qua đó thu hẹp khoảng cách giữa mô hình miền và phần mềm thực thi, luận án này đề xuất các kỹ thuật biểu diễn và chuyển đổi mô hình cho thiết kế hướng miền. Các đóng góp chính của luận án như sau.

Thứ nhất, luận án đề xuất các kỹ thuật biểu diễn mô hình miền, hướng tới việc tích hợp đặc tả cấu trúc, các ràng buộc phức tạp, hành vi và bảo mật trong cùng một khuôn khổ biểu diễn, làm cơ sở cho việc đặc tả đầy đủ các thông tin cần thiết phục vụ sinh tự động phần mềm trong bối cảnh thiết kế hướng miền.

Thứ hai, luận án đề xuất một kỹ thuật tích hợp các DSL theo các mối quan tâm không đồng nhất vào một mô hình miền hợp nhất, được gọi là *Unified Domain Model Language (UDML)*. Trong đó, UDML được đặc tả

đầy đủ ở cả cú pháp và ngữ nghĩa thực thi hình thức, đồng thời hỗ trợ kiểm chứng hình thức thông qua cơ chế ánh xạ ngữ nghĩa sang các đặc tả hình thức tương ứng.

Thứ ba, luận án đề xuất các kỹ thuật chuyển đổi mô hình, cải tiến một hoặc một số đặc trưng được lựa chọn, cho phép sinh tự động các bản mẫu phần mềm từ mô hình miền dựa trên các kỹ thuật biểu diễn trên miền đã xây dựng, hướng tới các chuyển đổi mô hình có chất lượng và phù hợp với các kịch bản sử dụng cụ thể trong các miền ứng dụng chuyên biệt.

Cuối cùng, luận án phát triển công cụ thực nghiệm và tiến hành đánh giá tính khả thi của việc áp dụng phương pháp đề xuất trong thực tế.

Các đóng góp này có ý nghĩa trong việc hỗ trợ đặc tả chính xác các mô hình miền, đồng thời hiện thực hóa các thao tác tự động trên mô hình miền, bao gồm sinh tự động mã nguồn và kiểm tra tính tuân thủ giữa thiết kế và cài đặt. Qua đó, luận án góp phần gia tăng mức độ tự động hóa trong phát triển phần mềm.

Từ khóa: Thiết kế hướng miền, kỹ thuật biểu diễn, chuyển đổi mô hình, UDML, DSL, aDSL.

Mục lục

DANH MỤC CÁC TỪ VIẾT TẮT

Từ viết tắt	Tiếng Anh	Tiếng Việt
AI	Artificial Intelligence	Trí tuệ nhân tạo
AGL	Activity Graph Language	Ngôn ngữ đồ thị hoạt động
ASM	Abstract Syntax Meta-Model	Siêu mô hình cú pháp trừu tượng
AST	Abstract Syntax Tree	Cây cú pháp trừu tượng
aDSL	annotation-based Domain-Specific Language	Ngôn ngữ chuyên biệt miền dựa vào chú thích
CAP	Constraint Annotation Pattern	Mẫu chú thích ràng buộc
CSM	Concrete Syntax Meta-Model	Siêu mô hình cú pháp cụ thể dạng văn bản
DDD	Domain-Driven Design	Thiết kế hướng miền
DM	Domain Model	Mô hình miền
DCSL	Domain Class Specification Language	Ngôn ngữ đặc tả lớp miền
DSML	Domain-Specific Modeling	Ngôn ngữ mô hình hóa chuyên biệt miền
DSM	Domain-Specific Modeling	Mô hình hóa chuyên biệt miền
DSL	Domain-Specific Language	Ngôn ngữ chuyên biệt miền
EMF	Eclipse Modeling Framework	Khung mô hình hóa Eclipse
GUI	Graphical User Interface	Giao diện người dùng đồ họa

IDE	Integrated Development Environment	Môi trường phát triển tích hợp
IFML	Interaction Flow Modeling Language	Ngôn ngữ mô hình hóa luồng tương tác
LLM	Large Language Model	Mô hình ngôn ngữ lớn
MDE	Model-Driven Engineering	Kỹ nghệ hướng mô hình
MOF	Meta-Object Facility	Phương tiện siêu mô hình
M2M	Model to Model	Mô hình sang mô hình
M2T	Model to Text	Mô hình sang văn bản
MOSA	Module-based Software Architecture	Kiến trúc phần mềm dựa vào mô-đun
MCC	Module Configuration Class	Lớp cấu hình mô đun phần mềm
MDA	Model-Driven Architecture	Kiến trúc hướng mô hình
MDSE	Model-Driven Software Engineering	Kỹ nghệ phát triển phần mềm hướng mô hình
MVC	Model View Controller	Mô hình thiết kế phần mềm
OMG	Object Management Group	Nhóm quản lý đối tượng
OCL	Object Constraint Language	Ngôn ngữ ràng buộc đối tượng
OOPL	Object-Oriented Programming Language	Ngôn ngữ lập trình hướng đối tượng
RM	Requirement Model	Mô hình yêu cầu
UL	Ubiquitous Language	Ngôn ngữ chung
UDML	Unified Domain Model Language	Ngôn ngữ mô hình miền hợp nhất
UML	Unified Modeling Language	Ngôn ngữ mô hình hóa hợp nhất
USE	UML-based Specification Environment	Môi trường đặc tả dựa trên UML
XML	eXtensible Markup Language	Ngôn ngữ đánh dấu mở rộng

DANH MỤC CÁC BẢNG

DANH MỤC CÁC HÌNH VẼ

Chương 1

GIỚI THIỆU

1.1 Đặt vấn đề

Trong nghiên cứu kỹ nghệ phần mềm hiện đại, thiết kế phần mềm được xem là một quá trình mang tính sáng tạo và lặp lại, trong đó các nguyên tắc, kỹ thuật và công cụ được khám phá và áp dụng xuyên suốt vòng đời phát triển phần mềm [30, 34]. Chất lượng của sản phẩm phần mềm vì vậy phụ thuộc đáng kể vào kinh nghiệm, kỹ năng và cách tiếp cận của các nhà phát triển phần mềm. Trong thực tiễn phát triển phần mềm, việc xây dựng các hệ thống có quy mô lớn và độ phức tạp cao bằng các phương pháp mang tính thủ công vẫn gặp nhiều thách thức, bao gồm khó khăn trong quản lý sự phụ thuộc, gia tăng chi phí và kéo dài thời gian triển khai. Do đó, nhu cầu rút ngắn chu kỳ phát triển, giảm thiểu chi phí, đồng thời đảm bảo khả năng thích ứng với các yêu cầu ngày càng đa dạng của phần mềm trong nhiều lĩnh vực ứng dụng trở nên cấp thiết.

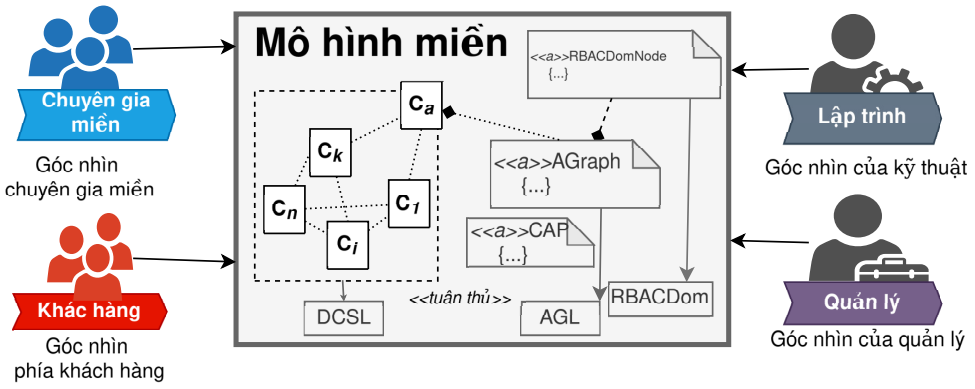
Trong bối cảnh đó, phát triển phần mềm dựa trên mô hình đã nổi lên như một hướng tiếp cận có hệ thống nhằm giảm thiểu độ phức tạp của phát triển phần mềm thông qua việc sử dụng các mô hình trừu tượng làm trung tâm [12, 34]. Đặc biệt, kỹ nghệ phần mềm hướng mô hình (*Model-Driven Software Engineering – MDSE*) [12, 17, 34] được xem là một trong những hướng phát triển rõ nét nhất của phát triển phần mềm dựa trên mô hình, tập trung vào việc khai thác các mô hình ở mức trừu tượng cao để hỗ trợ và tự động hóa quá trình phát triển phần mềm, từ thiết kế đến hiện thực hóa [30, 34]. Song song với MDSE, một phương pháp tiếp cận khác, khiêm

tốn hơn nhưng trực tiếp và thực tiễn hơn, là phương pháp thiết kế hướng miền (*Domain-Driven Design – DDD*) do Evans đề xuất [39, 121, 122].

DDD nhấn mạnh vai trò trung tâm của mô hình miền (*Domain Model - DM*) trong toàn bộ quá trình phát triển phần mềm. Theo Evans [39], DM không chỉ là một biểu diễn trừu tượng của miền nghiệp vụ mà còn đóng vai trò như một ngôn ngữ chung (*Ubiquitous Language – UL*) được chia sẻ giữa các bên liên quan. Trên cơ sở đó, DM trở thành nền tảng để định hình cấu trúc, hành vi và ngữ nghĩa nghiệp vụ của hệ thống phần mềm, đồng thời định hướng việc xây dựng và tiến hóa hệ thống một cách nhất quán. DDD có mối liên hệ chặt chẽ với các ngôn ngữ lập trình hướng đối tượng (*Object-Oriented Programming Languages – OOPL*). Theo Evans [39], các khái niệm cốt lõi của DDD như thực thể, đối tượng giá trị, tập hợp và dịch vụ miền có thể được ánh xạ một cách tự nhiên sang các cấu trúc của lập trình hướng đối tượng. Nhờ đó, DM vừa có khả năng biểu đạt cao, vừa khả thi về mặt kỹ thuật khi được triển khai trong các hệ thống phần mềm hướng đối tượng. Sự phù hợp này đã được chỉ ra trong nghiên cứu [16] cho rằng đối tượng là phương tiện tự nhiên để biểu diễn các thực thể của miền thế giới thực, đồng thời cấu trúc đối tượng cũng là nền tảng của các ngôn ngữ mô hình hóa phân tích và thiết kế ở mức cao. Quan điểm này tiếp tục được kế thừa và mở rộng trong các nghiên cứu gần đây, trong đó lập trình hướng đối tượng được xem như một ngôn ngữ khái niệm, có thể kết hợp với các yếu tố của lập trình hàm nhằm tăng cường khả năng biểu đạt trong việc mô tả các khái niệm miền và DM [110]. Tuy nhiên, mặc dù OOPL tạo điều kiện thuận lợi cho việc biểu diễn và triển khai DM, quá trình chuyển từ DM trừu tượng sang phần mềm có khả năng thực thi vẫn chủ yếu dựa vào thao tác thủ công của nhà phát triển, từ đó làm gia tăng nguy cơ sai lệch ngữ nghĩa giữa DM và hệ thống được xây dựng.

Trong bối cảnh đó, các OOPL không chỉ đóng vai trò là môi trường triển khai mà còn có thể được xem như ngôn ngữ chủ để biểu diễn và mở rộng DM. Thông qua các cơ chế như chú thích, kiểu dữ liệu trừu tượng và cấu trúc ngôn ngữ, OOPL cho phép nhúng trực tiếp các đặc tả miền mở rộng—bao gồm hành vi, ràng buộc và các chính sách bảo mật—vào DM. Tuy nhiên, các cơ chế này thường được sử dụng rời rạc và thiếu một nền tảng ngữ nghĩa thống nhất, dẫn đến hạn chế trong việc tích hợp các mối quan tâm và hỗ trợ thực thi một cách có hệ thống theo DDD.

Trong thực tiễn phát triển phần mềm, các bên liên quan tiếp cận hệ thống từ những góc nhìn khác nhau; chẳng hạn, người dùng chủ yếu tương tác thông qua ngôn ngữ tự nhiên và giao diện, trong khi nhà phát triển làm việc với các cấu trúc kỹ thuật và ngôn ngữ lập trình được trình bày trong Hình 1.1. Theo triết lý của DDD, sự khác biệt về góc nhìn này cần được quy chiếu về một ngôn ngữ chung thống nhất, trong đó DM đóng vai trò là lõi ngữ nghĩa, liên kết tri thức nghiệp vụ với thiết kế và triển khai hệ thống. Tuy nhiên, trong thực tiễn, vẫn tồn tại một khoảng cách đáng kể giữa mô hình miền và khả năng thực thi. DM thường được đặc tả bằng các biểu đồ lớp UML (*Unified Modeling Language*) kết hợp với các ràng buộc OCL (*Object Constraint Language*) [91] nhằm mô tả cấu trúc và các ràng buộc nghiệp vụ trên DM, trong khi việc đưa các đặc tả này đến trạng thái có khả năng thực thi phần lớn vẫn được thực hiện thủ công. Cách tiếp cận này khiến DM chủ yếu đóng vai trò như một đặc tả ở mức thiết kế, khó đạt được ngữ nghĩa của một DM *có khả năng thực thi*. Mặc dù các tiếp cận dựa trên ngôn ngữ chuyên biệt miền (*Domain-Specific Language – DSL*) [43, 103, 132] và MDSE [17] đã được đề xuất nhằm thu hẹp khoảng cách này, việc áp dụng và tích hợp chúng một cách chặt chẽ trong bối cảnh DDD vẫn còn nhiều hạn chế. Khoảng cách này làm suy giảm tính nhất quán ngữ nghĩa, đồng thời gây khó khăn cho việc bảo trì, mở rộng và tiến hóa phần mềm theo đúng tinh thần của DDD.



Hình 1.1: Các góc nhìn khác nhau trên cùng một mô hình miền.

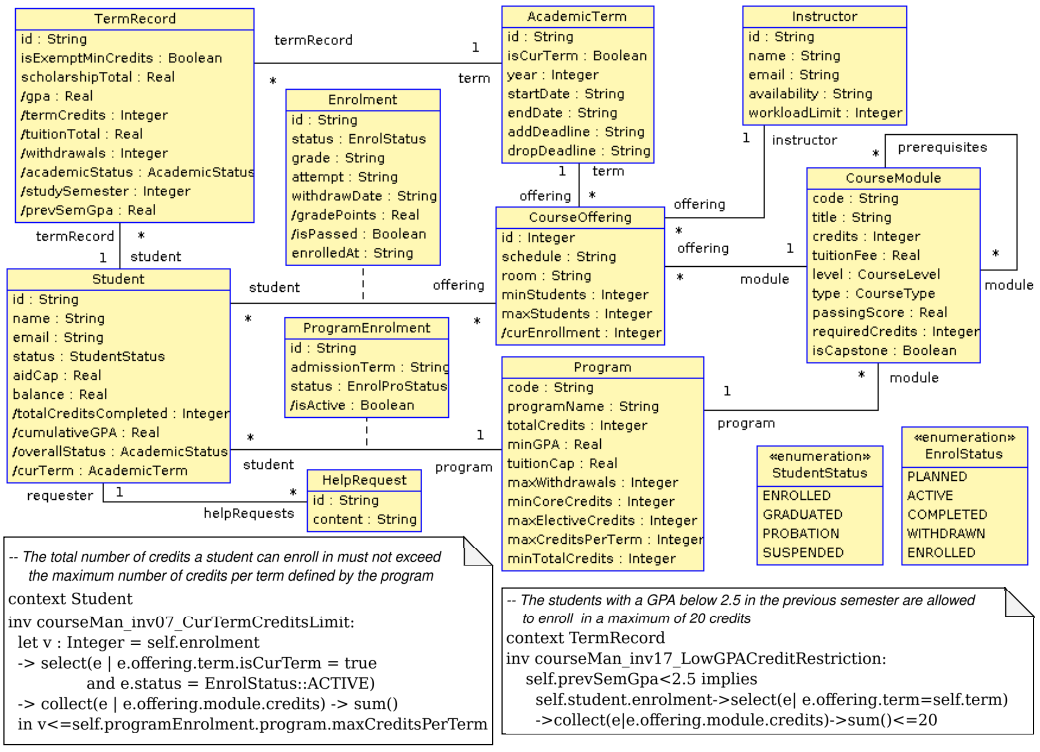
Mặc dù DDD đặt DM làm trung tâm của quá trình phát triển phần mềm, các cách tiếp cận hiện hữu chưa cung cấp được một cơ chế biểu diễn cho phép đặc tả một cách đầy đủ và nhất quán các khía cạnh cốt lõi của miền—bao gồm cấu trúc, hành vi và các ràng buộc nghiệp vụ—cũng như chưa hỗ trợ hiệu quả việc tích hợp các đặc tả này thành một mô hình miền

hợp nhất. Hơn nữa, việc chuyển đổi mô hình để thu được mô hình có khả năng thực thi, làm cơ sở cho sinh tự động phần mềm đúng yêu cầu và tuân thủ các nguyên lý của DDD, vẫn còn thiếu các cơ chế mang tính hệ thống. Các vấn đề này được làm rõ hơn thông qua ví dụ thúc đẩy ở phần tiếp theo.

1.1.1 Ví dụ thúc đẩy: Hệ thống `CourseMan`

Luận án sử dụng Hệ thống quản lý khóa học (CourseMan) như một ví dụ thúc đẩy nhằm minh họa các vấn đề nghiên cứu cốt lõi trong biểu diễn và chuyển đổi mô hình miền. Đây là một hệ thống thông tin điển hình trong miền quản lý đào tạo, bao quát nhiều khía cạnh nghiệp vụ như quản lý sinh viên, học phần, tổ chức giảng dạy, đăng ký học và theo dõi kết quả học tập.

CourseMan được lựa chọn do tính đại diện và mức độ phức tạp vừa phải của miền ứng dụng: hệ thống không chỉ bao gồm các cấu trúc dữ liệu phong phú mà còn liên quan đến nhiều quy tắc nghiệp vụ và quy trình xử lý đặc trưng. Nhờ đó, ví dụ này cho phép làm rõ các vấn đề về biểu diễn mô hình miền một cách cụ thể, đồng thời tạo cơ sở để dẫn dắt và minh họa các thách thức nghiên cứu được xem xét trong luận án.



Hình 1.2: Mô hình miền của Hệ thống quản lý khóa học.

Hình 1.2 trình bày mô hình miền của hệ thống CourseMan, trong đó các khái niệm miền chính được biểu diễn thông qua các lớp như `Student`, `CourseModule`, `CourseOffering`, `Instructor`, `AcademicTerm` và `Program`. Quan hệ đăng ký học phần của sinh viên được mô hình hóa thông qua lớp `Enrolment`, trong khi kết quả học tập theo từng học kỳ được thể hiện bởi `TermRecord`. Mô hình này phản ánh các thực thể và quan hệ cốt lõi của miền ứng dụng. Bên cạnh cấu trúc, hệ thống còn bao gồm các ràng buộc nghiệp vụ và các hành vi miền cơ bản. Các ràng buộc thể hiện các quy tắc toàn vẹn, chẳng hạn điều kiện đăng ký học phần hoặc giới hạn số lượng sinh viên của lớp học phần. Các hành vi miền liên quan đến các quy trình như mở lớp học phần, đăng ký học và cập nhật kết quả học tập. Các yếu tố này cho thấy mô hình miền cần bao quát đồng thời cấu trúc, ràng buộc và hành vi ở mức trừu tượng phù hợp.

Tuy nhiên, từ ví dụ trên có thể nhận thấy rằng các khía cạnh của mô hình miền, bao gồm cấu trúc, ràng buộc và hành vi, thường được biểu diễn bằng các ngôn ngữ và ở các mức trừu tượng khác nhau, và được phát triển tương đối độc lập. Điều này dẫn đến sự không đồng nhất trong biểu diễn và gây khó khăn trong việc duy trì một cách hiểu nhất quán về miền. Hệ quả là việc thiết lập và duy trì các liên kết ngữ nghĩa giữa các mô hình trở nên khó khăn, đặc biệt trong bối cảnh hệ thống tiến hóa và các thay đổi cần được phản ánh nhất quán trên nhiều khía cạnh. Bên cạnh đó, việc thiếu các cơ chế tích hợp và chuyển đổi mô hình một cách có hệ thống làm hạn chế khả năng bảo toàn ngữ nghĩa, đồng thời gây khó khăn cho việc kiểm chứng và tự động hóa các hoạt động phát triển phần mềm. Do đó, bài toán đặt ra không chỉ là xây dựng các mô hình riêng lẻ cho từng khía cạnh, mà là làm thế nào để biểu diễn, liên kết và chuyển đổi các mô hình này một cách nhất quán, nhằm hướng tới một biểu diễn miền hợp nhất có khả năng bảo toàn ngữ nghĩa.

Mặc dù CourseMan được sử dụng như một ví dụ thúc đẩy để làm rõ bài toán nghiên cứu, luận án không giới hạn việc đánh giá các kỹ thuật đề xuất trong một miền ứng dụng duy nhất. Nhằm đánh giá tính tổng quát và khả năng áp dụng của phương pháp, luận án còn xem xét các ca nghiên cứu bổ sung bao gồm: Hệ thống quản lý đặt hàng (OrderMan), Hệ thống quản lý quy trình nghiệp vụ (ProcessMan) và Hệ thống quản lý tạp chí mở (OJS). Cụ thể, OrderMan được sử dụng để khảo sát các đặc trưng liên quan đến

quy trình nghiệp vụ và sự phối hợp giữa các hành vi miền; ProcessMan cho phép đánh giá khả năng biểu diễn và xử lý các mô hình quy trình trong các bối cảnh ứng dụng khác nhau; trong khi đó, OJS là một hệ thống thực tế, được sử dụng để minh họa các yêu cầu về kiểm soát truy cập dựa trên vai trò và khả năng tích hợp các chính sách bảo mật ở mức mô hình. Việc xem xét đồng thời các hệ thống này cho phép đánh giá phương pháp đề xuất trên nhiều miền ứng dụng với các đặc trưng khác nhau, từ đó làm rõ khả năng mở rộng và tính tổng quát của cách tiếp cận.

1.1.2 Các thách thức nghiên cứu trong DDD

Thứ nhất, các nghiên cứu hiện tại về mô hình hóa trong DDD cho thấy vẫn còn hạn chế trong việc biểu diễn đồng thời và nhất quán các khía cạnh cấu trúc, hành vi và ràng buộc của miền trong một khuôn khổ hợp nhất. Phần lớn các tiếp cận sử dụng các biểu đồ UML và các ràng buộc OCL ở các không gian biểu diễn tách biệt, hoặc chỉ khai thác từng lát cắt riêng lẻ của mô hình miền (DM) thông qua các DSL [43, 103]. Một số nghiên cứu đề xuất các DSL nội sinh [132], đặc biệt là các tiếp cận dựa trên chú thích (*annotation-based Domain-Specific Language – aDSL*), trong đó DM được nhúng trực tiếp vào ngôn ngữ chủ [13, 31, 70, 95, 119]. Mặc dù các tiếp cận này cho phép xây dựng các DM có khả năng thực thi ở một mức độ nhất định, mối quan hệ ngữ nghĩa giữa cấu trúc, hành vi và ràng buộc vẫn chưa được đặc tả đầy đủ và nhất quán trong một khuôn khổ DM hợp nhất.

Thứ hai, trong các nghiên cứu hiện có, việc mô hình hóa và tích hợp nhiều mối quan tâm trong bối cảnh DDD hiện nay vẫn thiếu một nền tảng hợp nhất có ngữ nghĩa rõ ràng và khả năng thực thi. Các mối quan tâm như hành vi nghiệp vụ, bảo mật và các yêu cầu phi chức năng thường được đặc tả bằng các DSL riêng biệt theo từng góc nhìn [26, 37]. Mặc dù cách tiếp cận này làm tăng tính mô-đun và khả năng biểu đạt cho từng mối quan tâm, việc thiếu một DM hợp nhất phản ánh đầy đủ ngữ nghĩa tổng thể của hệ thống khiến DM khó đóng vai trò là trung tâm ngữ nghĩa như yêu cầu của DDD.

Thứ ba, trong các nghiên cứu về chuyển đổi mô hình trong kỹ nghệ phần mềm hướng mô hình (*Model-Driven Engineering – MDE*) [17, 29] chủ yếu tập trung vào các phép chuyển đổi riêng lẻ và thường được thực hiện ở mức

cú pháp. Các bộ chuyển đổi từ mô hình yêu cầu, bao gồm các biểu đồ UML và các ràng buộc OCL, sang DM và tiếp tục tới các hệ thống có khả năng thực thi hiện chưa được tổ chức một cách có hệ thống, đặc biệt về cơ chế bảo toàn ngữ nghĩa của hành vi và các ràng buộc miền [17, 69]. Điều này hạn chế khả năng kiểm soát chất lượng và tính đúng đắn trong quá trình sinh phần mềm dựa trên DM.

Từ những hạn chế đã được chỉ ra trong các nghiên cứu hiện có, luận án tập trung nghiên cứu các kỹ thuật biểu diễn và chuyển đổi mô hình nhằm xây dựng một DM làm trung tâm, vừa đóng vai trò là ngôn ngữ chung cho các bên liên quan, vừa cho phép tích hợp các mối quan tâm như hành vi, ràng buộc và bảo mật. Trên cơ sở đó, luận án hướng tới xây dựng các mô hình miền có khả năng thực thi và hỗ trợ sinh tự động phần mềm thông qua các cơ chế chuyển đổi mô hình, phù hợp với triết lý của DDD.

1.1.3 Phát biểu bài toán nghiên cứu

Luận án xác định các bài toán nghiên cứu cốt lõi cần được giải quyết, được cụ thể hóa thành các vấn đề nghiên cứu sau.

Vấn đề 1: Kỹ thuật biểu diễn mô hình miền tích hợp, giàu ngữ nghĩa và có khả năng thực thi. Trong DDD, mô hình miền được xem là trung tâm ngữ nghĩa của hệ thống phần mềm [39]. Tuy nhiên, các tiếp cận mô hình hóa hiện nay chủ yếu tập trung vào khía cạnh cấu trúc, trong khi các khía cạnh quan trọng khác của miền—như hành vi và các ràng buộc nghiệp vụ—thường được đặc tả tách rời bằng các mô hình hoặc ngôn ngữ khác nhau [17, 43].

Mặc dù một số tiếp cận dựa trên DSL nội sinh và chú thích cho phép gắn kết mô hình miền với triển khai [132], chúng vẫn chưa cung cấp được một cơ chế biểu diễn hợp nhất cho phép đặc tả đầy đủ và nhất quán các khía cạnh cốt lõi của miền trong một mô hình miền có khả năng thực thi. Do đó, việc nghiên cứu các kỹ thuật biểu diễn mô hình miền tích hợp, giàu ngữ nghĩa và hướng thực thi đặt ra như một vấn đề nghiên cứu quan trọng trong bối cảnh DDD.

Vấn đề 2: Cơ chế hợp nhất các mối quan tâm trong một mô hình miền thống nhất. Trong thực tiễn phát triển phần mềm, các mối

quan tâm khác nhau của miền—chẳng hạn như cấu trúc, hành vi và bảo mật—thường được đặc tả thông qua các mô hình hoặc DSL riêng biệt [17]. Cách tiếp cận này giúp tăng tính mô-đun, nhưng đồng thời làm suy giảm vai trò trung tâm ngữ nghĩa của mô hình miền theo tinh thần của DDD [39].

Các nghiên cứu hiện nay chủ yếu dừng ở việc tích hợp các mối quan tâm ở mức cú pháp hoặc cấu trúc mô hình, trong khi thiếu một cơ chế hợp nhất dựa trên ngữ nghĩa cho phép diễn giải thống nhất các mối quan tâm này trong một mô hình miền duy nhất [120]. Do đó, việc xây dựng một cơ chế hợp nhất các mối quan tâm dựa trên một nền tảng ngữ nghĩa chung, cho phép mô hình miền được xem như một thực thể thống nhất có khả năng thực thi và kiểm chứng, vẫn là một thách thức nghiên cứu mở.

Vấn đề 3: Bộ chuyển đổi mô hình. Mặc dù nhiều nghiên cứu trong kỹ nghệ phần mềm hướng mô hình đã đề xuất các kỹ thuật chuyển đổi mô hình nhằm tự động hóa quá trình phát triển phần mềm [17], phần lớn các tiếp cận này mới chỉ tập trung vào các phép chuyển đổi riêng lẻ và chưa hình thành được một khung chuyển đổi mô hình hoàn chỉnh đặt mô hình miền làm trung tâm. Bên cạnh đó, vấn đề duy trì tính nhất quán giữa các góc nhìn khác nhau trên cùng một mô hình miền vẫn chưa được xem xét một cách đầy đủ và có hệ thống.

Trong bối cảnh DDD, mô hình miền không chỉ đóng vai trò là công cụ thiết kế, mà còn là nơi hội tụ và chia sẻ tri thức miền giữa nhiều nhóm liên quan, bao gồm nhà phân tích nghiệp vụ, kiến trúc sư, nhà phát triển và các bên liên quan khác. Tuy nhiên, việc chuyển đổi từ mô hình miền sang các hiện thực phần mềm có khả năng thực thi hiện nay vẫn thiếu các cơ chế mang tính hệ thống nhằm bảo toàn ngữ nghĩa nghiệp vụ và đảm bảo tính nhất quán giữa các biểu diễn mô hình ở các mức trừu tượng khác nhau, từ đặc tả yêu cầu, mô hình thiết kế cho đến hiện thực phần mềm [121].

Để giải quyết vấn đề này, luận án đặt ra yêu cầu cần có một tập các bộ chuyển đổi mô hình có vai trò bổ trợ lẫn nhau, nhằm duy trì tính nhất quán giữa các góc nhìn của các bên liên quan trên cùng một mô hình miền. Cụ thể, các bộ chuyển đổi này bao gồm:(1) bộ chuyển đổi sinh tự động các bản mẫu phần mềm từ mô hình miền nhằm cung cấp góc nhìn thực thi; (2) bộ chuyển đổi từ các đặc tả yêu cầu mức cao (UML/OCL) sang mô hình miền nhằm gắn kết góc nhìn phân tích với mô hình nghiệp vụ; (3) bộ chuyển đổi

giữa cú pháp trừu tượng và cú pháp cụ thể nhằm đảm bảo tính nhất quán giữa các biểu diễn của ngôn ngữ miền; và (4) bộ chuyển đổi từ mô hình miền sang không gian ngữ nghĩa hình thức, chẳng hạn Event-B, nhằm hỗ trợ kiểm chứng và xác nhận các thuộc tính ngữ nghĩa quan trọng.

Do đó, nghiên cứu các bộ chuyển đổi mô hình có tính hệ thống, đặt mô hình miền làm trung tâm, bảo toàn ngữ nghĩa và hỗ trợ đồng thời nhiều góc nhìn khác nhau không chỉ là một yêu cầu kỹ thuật, mà còn là một vấn đề nghiên cứu quan trọng nhằm thu hẹp khoảng cách giữa đặc tả nghiệp vụ, mô hình và hiện thực phần mềm trong phát triển phần mềm hướng miền.

Tổng hợp các phân tích trên cho thấy rằng, vẫn còn tồn tại những vấn đề quan trọng chưa được giải quyết một cách hệ thống, cụ thể như sau.

1. Thiếu các kỹ thuật biểu diễn DM tích hợp, giàu ngữ nghĩa và có khả năng thực thi, trong đó các khía cạnh hành vi, bảo mật và ràng buộc được gắn trực tiếp vào DM, đồng thời vẫn bảo đảm vai trò của DM như một ngôn ngữ chung giữa các bên liên quan.

2. Thiếu một cơ chế hợp nhất các mối quan tâm vào DM hợp nhất, cũng như cơ chế kiểm chứng một cách tổng thể và có hệ thống, nhằm đảm bảo tính mô-đun, tính nhất quán ngữ nghĩa và khả năng sinh mã tự động.

3. Thiếu các bộ chuyển đổi mô hình hỗ trợ thao tác tự động trên các mô hình miền hợp nhất, chẳng hạn như sinh các bản mẫu phần mềm từ mô hình yêu cầu theo DDD.

1.2 Mục tiêu và phạm vi nghiên cứu

Luận án nghiên cứu và đề xuất các kỹ thuật biểu diễn và chuyển đổi mô hình trong thiết kế hướng miền (DDD), nhằm thu hẹp khoảng cách giữa mô hình miền và hiện thực phần mềm. Luận án hướng tới việc biểu diễn đầy đủ các khía cạnh của miền nghiệp vụ, bao gồm cấu trúc, hành vi và các chính sách bảo mật, đồng thời nghiên cứu các kỹ thuật sinh tự động phần mềm từ mô hình miền, qua đó đảm bảo ngữ nghĩa của các cấu trúc, hành vi và ràng buộc nghiệp vụ được duy trì nhất quán trong quá trình chuyển đổi từ

mô hình miền đến hiện thực phần mềm. Để đạt được mục tiêu tổng quát này, luận án tập trung vào các mục tiêu nghiên cứu cụ thể sau.

Mục tiêu 1. Nghiên cứu và đề xuất các kỹ thuật biểu diễn mở rộng cho DM, bao gồm: đặc tả hành vi bằng ngôn ngữ đồ thị hoạt động (*Activity Graph Language - AGL*); đặc tả các chính sách bảo mật dựa trên vai trò bằng DSL là RBACDom (*Domain RBAC Specific Language*); và đặc tả các ràng buộc OCL bằng các mẫu chú thích ràng buộc CAP (*Constraint Annotation Patterns*). Qua đó, hình thành một DM biểu diễn đầy đủ và nhất quán các khía cạnh cấu trúc, hành vi và ràng buộc nghiệp vụ của miền, làm cơ sở cho việc phát triển phần mềm đáp ứng các yêu cầu nghiệp vụ.

Mục tiêu 2. Đề xuất ngôn ngữ mô hình miền hợp nhất UDML (*Unified Domain Model Language*) trong bối cảnh DDD, nhằm tích hợp các mối quan tâm của DM—cấu trúc, hành vi và bảo mật—ở mức ngữ nghĩa miền và hành vi. Trên cơ sở đó, luận án đề xuất cơ chế hợp nhất (*Compose*) dựa trên chú thích để kết hợp các DSL chuyên biệt vào một mô hình miền thống nhất, đồng thời xây dựng nền tảng ngữ nghĩa phục vụ kiểm chứng có hệ thống và hỗ trợ sinh tự động các mô hình miền có khả năng thực thi, góp phần thu hẹp khoảng cách giữa đặc tả miền và hiện thực triển khai theo tinh thần của DDD.

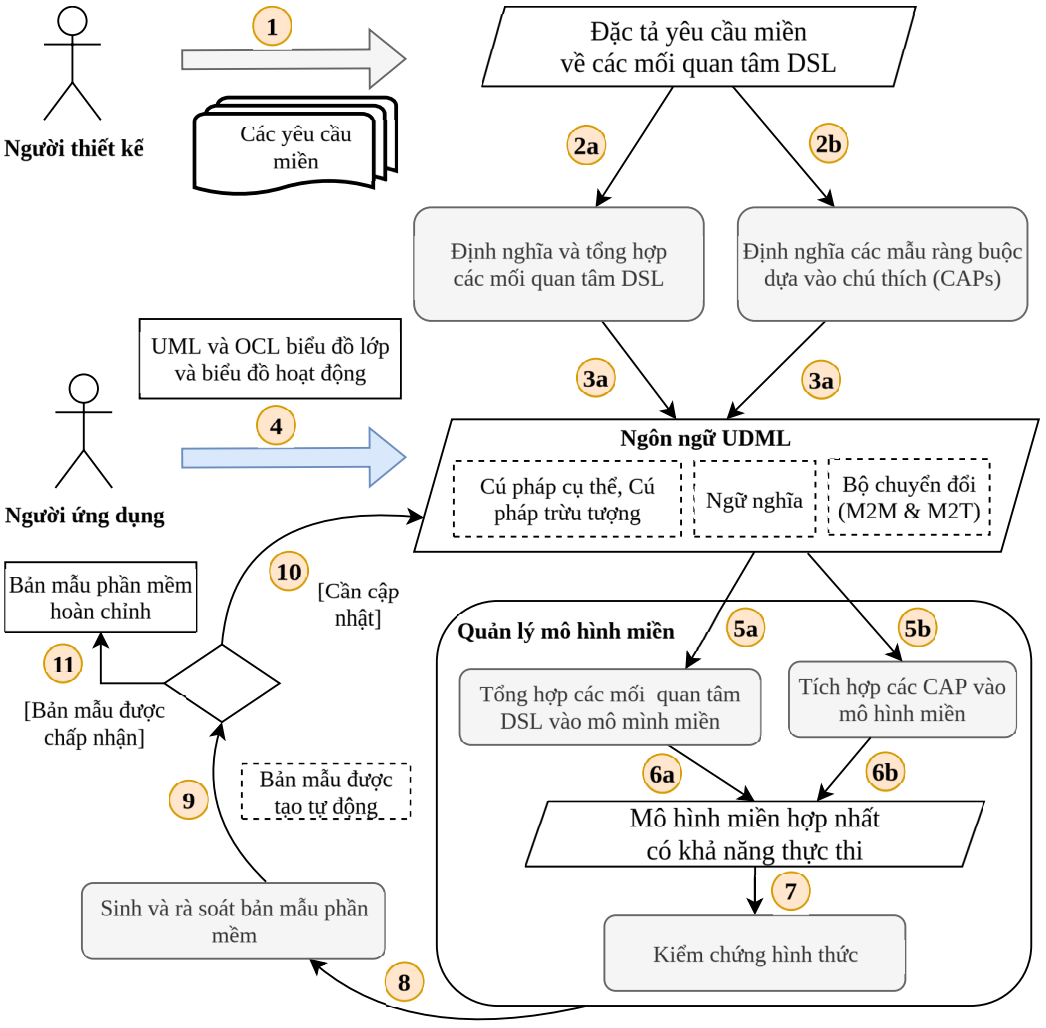
Mục tiêu 3. Nghiên cứu và đề xuất bộ chuyển đổi mô hình nhằm gia tăng mức độ tự động hóa trong phát triển phần mềm, hướng tới việc sinh tự động các bản mẫu phần mềm có khả năng thực thi.

Phạm vi nghiên cứu của luận án tập trung vào các kỹ thuật biểu diễn, hợp nhất và chuyển đổi mô hình dựa trên DSL trong bối cảnh DDD. Luận án kết hợp mô hình hóa bằng DSL với các kỹ thuật thiết kế và kiểm chứng hình thức, đồng thời phát triển các ngôn ngữ và bộ chuyển đổi mô hình theo quy trình lặp lấy mô hình miền làm trung tâm.

Để kiểm chứng tính khả thi và khả năng áp dụng của phương pháp, các đề xuất được đánh giá thông qua thực nghiệm trên một tập các ca nghiên cứu điển hình (CourseMan, OrderMan, ProcessMan và OJS). Các ca nghiên cứu này được trình bày và phân tích trong Chương 6 của luận án.

1.3 Nội dung và phương pháp nghiên cứu

Để đạt được các mục tiêu đã nêu, luận án tập trung nghiên cứu các kỹ thuật biểu diễn, hợp nhất và chuyển đổi DM trong bối cảnh DDD. Cụ thể, luận án xem xét việc đặc tả các mối quan tâm khác nhau của miền—bao gồm cấu trúc, hành vi và bảo mật—thông qua các DSL, đồng thời nghiên cứu cơ chế hợp nhất các đặc tả này vào một ngôn ngữ mô hình miền hợp nhất, gọi là UDML. Trên cơ sở đó, luận án đề xuất các cơ chế chuyển đổi mô hình và nền tảng ngữ nghĩa phục vụ kiểm chứng hình thức, hướng tới việc xây dựng các mô hình miền hợp nhất có khả năng thực thi và hỗ trợ sinh tự động các bản mẫu phần mềm. Các thành phần chính của mô hình miền hợp nhất được minh họa trong Hình 1.3 bao gồm các tác vụ chính, được gán nhãn lần lượt từ *1* đến *11*.



Hình 1.3: Tổng quan phương pháp đề xuất cho kỹ thuật biểu diễn và chuyển đổi mô hình.

Trước hết, người thiết kế ngôn ngữ tiến hành đặc tả các yêu cầu miền xuất phát từ nhiều miền ứng dụng khác nhau, bao gồm các mối quan tâm được đặc tả bằng DSL và các nhóm ràng buộc OCL, như minh họa trong *nhãn 1*. Bước này nhằm xác định và khái quát hóa các mối quan tâm miền liên quan đến cấu trúc, hành vi, bảo mật, cũng như các nhóm ràng buộc OCL tương ứng. Trên cơ sở đó, các mối quan tâm DSL được định nghĩa và hợp nhất, như thể hiện trong *nhãn 2a*, bao gồm các thành phần cú pháp trừu tượng và các dạng cú pháp cụ thể (văn bản hoặc đồ họa) để hỗ trợ xây dựng các mô hình chi tiết cho từng mối quan tâm cụ thể. Song song với đó, một danh mục các mẫu chú thích ràng buộc CAP được xây dựng bằng cách khai thác và khái quát hóa các ràng buộc từ yêu cầu miền, như minh họa trong *nhãn 2b*. Mỗi CAP được định nghĩa bao gồm: (i) một biểu đồ lớp UML mô tả các khái niệm miền và mối quan hệ giữa chúng; (ii) một đặc tả OCL biểu diễn các luật nghiệp vụ và các bất biến; và (iii) tập các chú thích dùng để gắn các ràng buộc OCL vào DM.

Tiếp đó, với ngôn ngữ UDML đã được thiết kế đầy đủ về cú pháp, ngữ nghĩa, cùng với các bộ chuyển đổi M2M và M2T, người dùng tập trung áp dụng ngôn ngữ này để biểu diễn mô hình miền cho hệ thống cụ thể. Các đặc tả DSL theo mối quan tâm chuyên biệt (*nhãn 3a*) và các mẫu CAP (*nhãn 3b*), được xây dựng trong giai đoạn thiết kế bởi chuyên gia miền, được sử dụng kết hợp với các DM cụ thể dưới dạng biểu đồ lớp và biểu đồ hoạt động UML/OCL, như minh họa trong *nhãn 4*. Trên cơ sở đó, các mối quan tâm DSL được tổng hợp và các CAP được tích hợp vào DM, như thể hiện trong *nhãn 5a* và *nhãn 5b*, nhằm suy dẫn ra một mô hình miền hợp nhất làm nền tảng cho việc sinh bản mẫu phần mềm. UDML được thiết kế với một mô hình lõi cho phép tích hợp một cách thống nhất các mối quan tâm DSL vào DM, như minh họa trong *nhãn 6a* và *nhãn 6b*. Kết quả thu được là một mô hình miền hợp nhất có khả năng thực thi, làm cơ sở cho bước kiểm chứng hình thức mô hình miền hợp nhất trước khi sinh bản mẫu phần mềm, như thể hiện trong *nhãn 7*.

Cuối cùng, mô hình UDML đã được tích hợp đầy đủ đóng vai trò làm nền tảng để sinh tự động bản mẫu phần mềm, như thể hiện trong *nhãn 8*. Bản mẫu phần mềm này được tạo dựa trên khung JDA [71] và sau đó được các chuyên gia miền đánh giá để thu thập phản hồi, như minh họa trong *nhãn 9*. Trên cơ sở phản hồi thu được, cả mô hình UDML và các đặc tả

mối quan tâm liên quan sẽ được cập nhật tương ứng, như thể hiện trong *nhãn 10*. Quy trình này được lặp lại một cách có hệ thống, hỗ trợ chu trình cải tiến liên tục cho đến khi hệ thống phần mềm đáp ứng đầy đủ các yêu cầu đã đặt ra, như minh họa trong *nhãn 11*.

1.4 Các đóng góp chính của luận án

Luận án đề xuất các kỹ thuật biểu diễn và chuyển đổi mô hình cho DDD, tập trung vào việc biểu diễn và tích hợp các mối quan tâm vào mô hình miền hợp nhất, hỗ trợ kiểm chứng và hướng thực thi, đồng thời sinh tự động các bản mẫu phần mềm thông qua chuyển đổi mô hình, cụ thể như sau.

1. Đề xuất các kỹ thuật biểu diễn mô hình miền (DM) trong bối cảnh DDD, cho phép đặc tả một cách tường minh và nhất quán các khía cạnh cấu trúc, hành vi và ràng buộc nghiệp vụ của các miền nghiệp vụ. Các kỹ thuật này mở rộng khả năng biểu đạt của DM, làm cơ sở cho các bước hợp nhất, kiểm chứng và chuyển đổi mô hình tiếp theo.

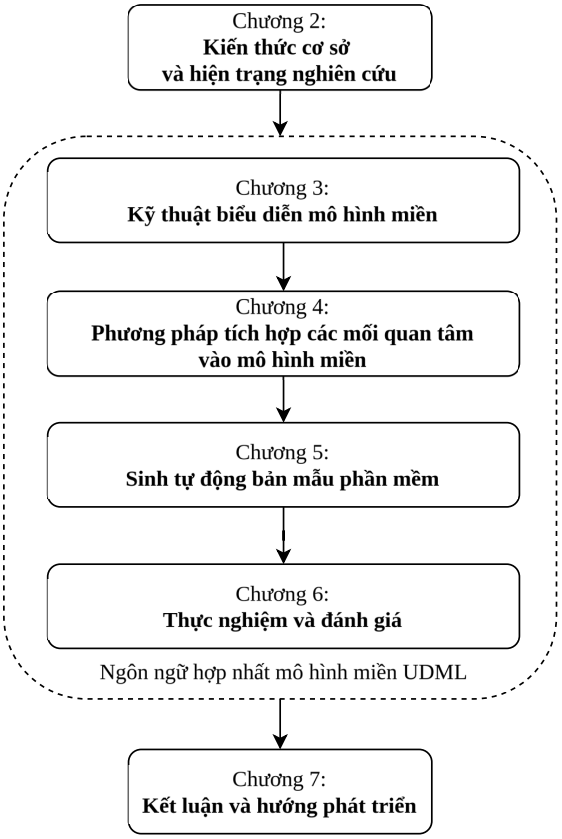
2. Đề xuất một kỹ thuật hợp nhất các DSL theo các mối quan tâm chuyên biệt vào một mô hình miền hợp nhất có khả năng thực thi. Trên cơ sở đó, luận án xây dựng nền tảng ngữ nghĩa phục vụ kiểm chứng hình thức, nhằm đảm bảo tính nhất quán ngữ nghĩa của mô hình miền hợp nhất ở giai đoạn thiết kế.

3. Đề xuất các kỹ thuật chuyển đổi mô hình dựa trên DM, cho phép sinh tự động các bản mẫu phần mềm có khả năng thực thi, đồng thời bảo toàn ngữ nghĩa miền và chất lượng chuyển đổi, phù hợp với các kịch bản sử dụng cụ thể trong các miền ứng dụng chuyên biệt.

4. Phát triển một công cụ thực nghiệm nhằm hiện thực hóa các kỹ thuật biểu diễn, hợp nhất và chuyển đổi mô hình được đề xuất, đồng thời đánh giá tính khả thi và khả năng áp dụng của các kỹ thuật này trong các bối cảnh ứng dụng thực tế.

1.5 Cấu trúc luận án

Luận án *"Nghiên cứu các kỹ thuật biểu diễn và chuyển đổi mô hình cho thiết kế hướng miền"* được tổ chức thành bảy chương. Trong đó, *Chương 1 – Giới thiệu* trình bày lý do chọn đề tài, mục tiêu, nội dung, đối tượng nghiên cứu và các đóng góp chính của luận án. Cấu trúc tổng thể của luận án được minh họa trong Hình 1.4, với nội dung các chương được tổ chức như sau.



Hình 1.4: Cấu trúc luận án.

Chương 2 trình bày cơ sở lý thuyết và tổng quan nghiên cứu liên quan đến đề tài. Chương giới thiệu các kỹ thuật biểu diễn mô hình miền trong DDD dựa trên DSL (nội sinh và ngoại sinh), đồng thời phân tích vai trò của các mối quan tâm như hành vi, ràng buộc và bảo mật trong mô hình miền. Bên cạnh đó, chương trình bày các kỹ thuật chuyển đổi mô hình và sinh tự động phần mềm trong kỹ nghệ phần mềm hướng mô hình. Trên cơ sở đó, chương đánh giá các hạn chế của các tiếp cận hiện tại, đặc biệt trong việc tích hợp các mối quan tâm và bảo toàn ngữ nghĩa, qua đó xác định khoảng trống nghiên cứu cho luận án.

Chương 3 trình bày nghiên cứu về kỹ thuật tích hợp khía cạnh hành vi vào DM nhằm hỗ trợ sinh tự động phần mềm theo DDD. Chương này đề xuất một cơ chế tích hợp hành vi thông qua DSL dựa trên chú thích, gọi là *Activity Graph Language* (AGL), để biểu diễn các hành vi miền. Đồng thời, phương pháp tích hợp các ràng buộc phức tạp vào DM thông qua mẫu *Constraint Annotation Pattern* (CAP) cũng được trình bày và đánh giá.

Chương 4 trình bày phương pháp tích hợp các mối quan tâm vào mô

hình miền hợp nhất có khả năng thực thi trong DDD. Chương đề xuất cơ chế hợp nhất các DSL theo mối quan tâm thông qua tiếp cận dựa trên chú thích ở mức cây cú pháp trừu tượng, cho phép tích hợp các đặc tả hành vi, ràng buộc và bảo mật vào mô hình miền một cách nhất quán. Trên cơ sở đó, chương giới thiệu DSL bảo mật RBACDom như một mối quan tâm cụ thể và trình bày cách tích hợp với mô hình hành vi. Đồng thời, chương đề xuất nền tảng ngữ nghĩa và phương pháp kiểm chứng hình thức cho mô hình miền hợp nhất, nhằm đảm bảo tính nhất quán ngữ nghĩa giữa các mối quan tâm.

Chương 5 trình bày kỹ thuật sinh tự động bản mẫu phần mềm dựa trên các bộ chuyển đổi mô hình. Chương này tập trung xây dựng bộ chuyển đổi từ mô hình yêu cầu (*Requirement Model – RM*) sang mô hình miền hợp nhất (*Unified Domain Model – UDM*), gọi là RM2UDM, nhằm hỗ trợ tự động hóa quá trình tạo bản mẫu phần mềm.

Chương 6 trình bày công cụ hỗ trợ và đánh giá các phương pháp đề xuất thông qua các ca nghiên cứu điển hình, bao gồm CourseMan, OrderMan, ProcessMan và OJS. Các kết quả thực nghiệm và thảo luận trong chương này được sử dụng để đánh giá tính khả thi và hiệu quả của phương pháp đề xuất trong các bối cảnh ứng dụng thực tế.

Cuối cùng, *Chương 7* tổng kết các đóng góp chính của luận án và thảo luận các hướng nghiên cứu tiếp theo dựa trên các kết quả đã đạt được.

Chương 2

CƠ SỞ LÝ THUYẾT VÀ TỔNG QUAN TÌNH HÌNH NGHIÊN CỨU

Trong chương này, luận án trình bày các kiến thức nền tảng liên quan đến biểu diễn mô hình miền, tích hợp các mối quan tâm và chuyển đổi mô hình trong thiết kế hướng miền. Nội dung chương nhằm làm rõ hiện trạng nghiên cứu của các tiếp cận hiện có trong việc đặc tả đầy đủ các khía cạnh cấu trúc, hành vi và ràng buộc miền, cũng như trong việc hợp nhất các mối quan tâm miền và bảo toàn ngữ nghĩa khi sinh phần mềm. Trên cơ sở đó, chương thiết lập nền tảng khái niệm và ngữ nghĩa cho việc đề xuất các kỹ thuật biểu diễn mô hình miền giàu ngữ nghĩa, hợp nhất các mối quan tâm miền vào một mô hình miền hợp nhất có khả năng thực thi, và hỗ trợ chuyển đổi mô hình để sinh tự động các bản mẫu phần mềm trong các chương tiếp theo.

2.1 Tổng quan về thiết kế hướng miền

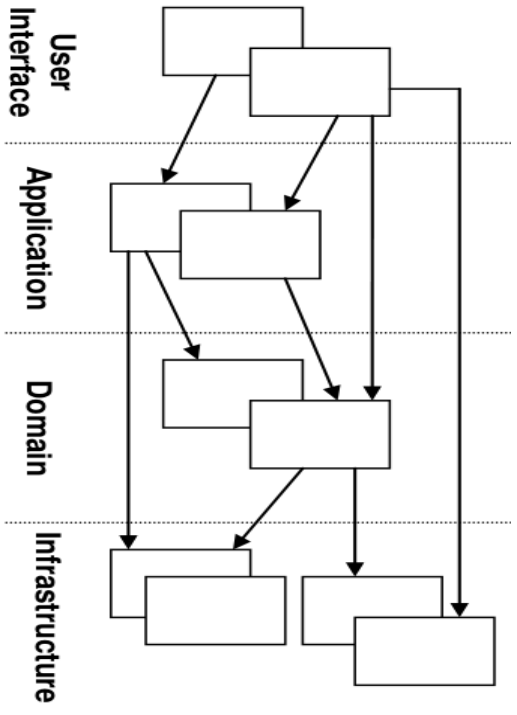
Trong mục này, luận án trình bày các khái niệm nền tảng của thiết kế hướng miền làm cơ sở lý thuyết cho các kỹ thuật biểu diễn được đề xuất trong các chương sau, bao gồm AGL, CAP, RBACDom và UDML.

2.1.1 Nền tảng thiết kế hướng miền

Thiết kế hướng miền (*Domain-Driven Design - DDD*) [39] hướng tới việc phát triển phần mềm theo cách lặp và tiến hóa, dựa trên mô hình miền

(*Domain Model - DM*) phản ánh trung thực miền ứng dụng; DM này vừa phải nắm bắt đầy đủ các yêu cầu nghiệp vụ, vừa phải khả thi về mặt kỹ thuật để hiện thực hóa trong triển khai. Theo Evans [39], các ngôn ngữ lập trình hướng đối tượng (*Object-Oriented Programming Language - OOPL*) có sự phù hợp tự nhiên khi áp dụng DDD. Trước đó, các nghiên cứu gần đây về DM đã chỉ ra rằng DM cần đảm bảo cả tính biểu đạt và khả thi thực thi [15, 34]. Thứ nhất, các đối tượng trong ngôn ngữ hướng đối tượng phản ánh trực tiếp các thực thể tồn tại trong miền thực tế. Thứ hai, cấu trúc đối tượng cũng là cơ sở cho các ngôn ngữ mô hình hóa trừu tượng được sử dụng trong phân tích và thiết kế, qua đó hỗ trợ khái niệm hóa và hiện thực hóa miền ứng dụng [32]. Trong bối cảnh này, DDD được xem như một phương pháp cụ thể hóa việc phát triển phần mềm hướng đối tượng dựa trên DM.

Trong phát triển phần mềm hiện đại, đặc biệt với các hệ thống nghiệp vụ phức tạp, người ta ngày càng nhận ra rằng vấn đề chính không nằm ở công nghệ, mà nằm ở chỗ hiểu và nắm bắt đúng miền nghiệp vụ. Trong phương pháp DDD, DM đóng vai trò trung tâm của phần mềm – nơi chứa đựng phần lớn độ phức tạp. Nơi các khái niệm nghiệp vụ, hành vi và quy tắc được mô hình hóa thống nhất, và dùng một ngôn ngữ chung để tất cả các bên tham gia cùng trao đổi. Evans nhấn mạnh rằng DM không chỉ là tài liệu phân tích mà phải gắn chặt với cài đặt, chi phối việc phát triển phần mềm trong suốt vòng đời. Điều này bao hàm hai yêu cầu: (1) DM phải giàu ngữ nghĩa để phản ánh đúng bản chất nghiệp vụ, và (2) DM phải đủ cấu trúc để định hướng mã nguồn triển khai.



Hình 2.1: Kiến trúc phân tầng chung cho DDD (Nguồn [39]).

Mô hình miền (DM) là mô hình mô tả cấu trúc các khái niệm, mối quan hệ giữa chúng trong một miền vấn đề nào đó. Mỗi miền nghiệp vụ đều có

thể được biểu diễn bởi một DM; ví dụ DM cho quản lý đào tạo, giao hàng, ngân hàng, v.v.

Hình 2.1 mô tả mô hình kiến trúc chung cho DDD dạng phân tầng: DM nằm ở tầng miền (*Domain*), đóng vai trò trung tâm, được trích từ sách của Evans, khẳng định vai trò trung tâm của DM trong một kiến trúc phần mềm tổng quát dạng phân tầng. Mô hình này bao gồm bốn tầng, trong đó DM nằm ở tầng miền. Tầng trên cùng là tầng giao diện (*User Interface - UI*), nơi đặt các phần giao diện dành cho người sử dụng tương tác với phần mềm. Tầng tiếp theo là tầng ứng dụng (*Application*), nơi đặt các thành phần định nghĩa các dịch vụ mà phần mềm cung cấp cho người sử dụng hay các hệ thống khác. Để thực hiện các dịch vụ, các thành phần này tìm và điều phối hoạt động của các thành phần ở tầng miền. Tầng miền là nơi đặt DM, biểu diễn lô-gic miền vấn đề chuyên biệt của mỗi phần mềm. Để thiết kế được DM này đòi hỏi sự hiểu biết cặn kẽ về nghiệp vụ của miền đó. Tầng dưới cùng là tầng cơ sở hạ tầng (*Infrastructure*), nơi đặt các thành phần chịu trách nhiệm giao tiếp với nền tảng công nghệ được sử dụng để chạy phần mềm. Tầng này giúp các tầng trên hoạt động mà không phụ thuộc vào nền tảng công nghệ, qua đó nâng cao khả năng tái sử dụng phần mềm cho nhiều nền tảng khác nhau.

Trong DDD, hai đặc trưng quan trọng được nhấn mạnh gồm: (1) tính khả thi, tức DM phải có khả năng trở thành mã nguồn và ngược lại; và (2) tính thỏa mãn, tức DM phải phản ánh đúng các yêu cầu nghiệp vụ thông qua một ngôn ngữ chung được phát triển và tinh chỉnh trong quá trình làm việc với các bên liên quan [39]. Ngôn ngữ này cho phép các yêu cầu miền được diễn đạt chính xác và nhất quán, đồng thời đảm bảo mô hình luôn bám sát nhu cầu nghiệp vụ. Việc đạt được hai đặc trưng cốt lõi này hiện đang là một trong những trọng tâm của các nghiên cứu về DDD.

Bên cạnh đó, DDD còn dựa trên hai nguyên lý nền tảng giúp duy trì sự tương thích giữa mô hình và hiện thực là ngữ cảnh giới hạn và ngôn ngữ chung (UL). Ngữ cảnh giới hạn xác định phạm vi mà các khái niệm miền được diễn giải, ngăn ngừa sự pha tạp ngữ nghĩa giữa các vùng nghiệp vụ, đồng thời bảo đảm mọi kỹ thuật mô hình hóa được triển khai trong một không gian ngữ nghĩa thống nhất. Mặt khác, UL đóng vai trò là phương tiện ngôn ngữ xuyên suốt giữa DM, ngôn ngữ chuyên biệt miền (*Domain-Specific Language - DSL*) [43] và mã nguồn, giúp loại bỏ sự chênh lệch giữa tài liệu

phân tích và mã chương trình, vốn là nguyên nhân sâu xa của khoảng cách giữa mô hình–cài đặt. DSL giúp liên kết chặt chẽ giữa mô hình thiết kế và hiện thực, từ đó cải thiện tính đúng đắn, khả năng bảo trì, và tính mở rộng của phần mềm [22, 42, 94, 117].

Evans nhấn mạnh rằng DM cần được hiện thực hóa trực tiếp trong mã nguồn, thay vì chỉ tồn tại ở dạng tài liệu. Tuy nhiên, trong thực tiễn, DM thường chỉ được mô tả bằng UML/OCL [91], tài liệu văn bản hoặc thông qua trao đổi bằng lời nói, dẫn đến khoảng cách đáng kể giữa mô hình và hiện thực triển khai. UL không được chuyển xuống mã nguồn một cách hệ thống. Mặc dù, đã chỉ ra UL phải được dùng trong cả mô hình và mã nguồn, nhưng trong phương pháp DDD chưa đưa ra cơ chế kỹ thuật để UL trở thành một ngôn ngữ hình thức thay vì chỉ là ngôn ngữ tự nhiên. Kết quả là UL dễ bị sai lệch giữa phân tích – thiết kế – cài đặt [59].

2.1.2 Mô hình miền hướng thực thi

Mục này trình bày các tiếp cận biểu diễn phổ biến của DM, nhấn mạnh ưu điểm về khả năng biểu đạt và chuẩn hóa, đồng thời chỉ ra hạn chế về tính hợp nhất và khả năng hướng thực thi của các biểu diễn hiện hành trong bối cảnh DDD.

Trong nhiều hệ thống phần mềm hiện nay, DM chủ yếu đóng vai trò mô tả, chưa có khả năng thực thi trực tiếp. Tuy nhiên, nghiên cứu [36] đã nhấn mạnh rằng DM cần được **kích hoạt** nhằm hỗ trợ sinh tự động bản mẫu phần mềm hoặc thậm chí có thể được thực thi trực tiếp.

Các hướng tiếp cận nghiên cứu liên quan đến biểu diễn và thực thi hành vi miền hiện nay tập trung giải quyết một số thách thức chính sau:

- Các nghiên cứu về DDD chủ yếu tập trung vào khía cạnh cấu trúc của DM–bao gồm thực thể, tổng hợp, kho lưu trữ và các khối xây dựng tương tự–trong khi hành vi miền thường bị tách ra khỏi mô hình. Trong nhiều công trình, hành vi được hiện thực trong mã nguồn thông qua các dịch vụ miền hoặc các cơ chế sự kiện, hoặc được mô tả bằng các biểu đồ UML tách rời [59, 122, 134, 135].
- Một số hướng nghiên cứu trong kỹ nghệ hướng mô hình (*Model-Driven Engineering – MDE*) cho phép gắn các mô hình hành vi như máy trạng

thái hoặc biểu đồ hoạt động vào mô hình, tuy nhiên các tiếp cận này thường nằm ngoài ngữ cảnh của DDD và chưa đồng nhất hóa hành vi với cấu trúc trong một DM duy nhất [14, 17, 77, 120, 131].

- Nhiều công trình trong nhóm này tập trung vào việc định nghĩa các phép ánh xạ từ biểu đồ hoạt động, trạng thái hoặc tuần tự của UML sang mã thực thi, chủ yếu dựa trên các quy tắc kỹ thuật ở mức điều khiển luồng [75, 124].

- Bài toán *thu hẹp khoảng cách giữa yêu cầu nghiệp vụ và mô hình hóa hướng đối tượng* luôn được xem là trọng tâm. Tuy nhiên, phần lớn các tiếp cận hiện nay mới dừng lại ở mức khái niệm nhằm hỗ trợ mô hình hóa đúng tư duy miền, trong khi việc ánh xạ có hệ thống hành vi miền sang mã nguồn thường được giao cho các khung làm việc hoặc các quy ước lập trình, thiếu một ngữ nghĩa hình thức làm cơ sở [135].

- Song song đó, một số nghiên cứu trong MDE và lĩnh vực sinh mã nguồn từ mô hình hành vi đã đề xuất các bộ chuyển đổi từ mô hình hành vi sang mã nguồn [108]. Tuy nhiên, các tiếp cận này thường vận hành độc lập với kiến trúc hướng miền và chưa được tích hợp với một DM cụ thể hay một ngôn ngữ miền thống nhất.

- Nhiều công trình khác tập trung vào sinh mã tự động từ các mô hình hành vi UML với mục tiêu đảm bảo tính nhất quán giữa thiết kế hành vi và mã nguồn thông qua các phép ánh xạ mang tính cơ học. Mã sinh ra chủ yếu là mã điều khiển tổng quát, chưa được cố kết trong một khung làm việc hướng miền rõ ràng [124].

- Ngoài ra, một số nghiên cứu ưu tiên các tiêu chí như an toàn, độ tin cậy hoặc hiệu năng. Mặc dù có giá trị trong lĩnh vực sinh mã nguồn, các tiếp cận này không đặt trọng tâm vào các nguyên lý cốt lõi của DDD, chẳng hạn như UL hay khả năng để chuyên gia miền trực tiếp hiểu và xác nhận mô hình hành vi.

Trong bối cảnh các hệ thống được phát triển theo tinh thần DDD, một số công trình thực tiễn như Apache Isis [119] và OpenXava [95] đã khai thác các ngôn ngữ chuyên biệt miền dựa trên chú thích (*annotation-based Domain-Specific Languages – aDSL*). Cách tiếp cận này có thể được xem

là một dạng DSL nội sinh, được nhúng trong OOPL (như Java), tận dụng cú pháp và ngữ nghĩa sẵn có của ngôn ngữ chủ để biểu đạt và hiện thực hóa DM. Từ cách biểu diễn này, có thể thu được DM thực thi theo hai hướng tiếp cận chính. Cách tiếp cận trực tiếp, nhúng thẳng phần hiện thực và các mối quan tâm của chương trình vào chính OOPL như Java [47] và C# [55], giúp tạo ra phần mềm chạy được nhưng dễ làm DM bị pha tạp và khó duy trì. Cách tiếp cận gián tiếp, dựa trên phương pháp phát triển hướng mô hình, trong đó DM được hợp nhất với các mối quan tâm khác (như cấu trúc, hành vi và bảo mật) thông qua các mô hình trừu tượng hơn bằng UML/OCL hoặc các DSL bổ trợ. Chương trình cuối cùng có thể được tạo ra bằng các chuyển đổi mô hình, bao gồm chuyển đổi mô hình sang mô hình (*Model to Model - M2M*) hoặc mô hình sang văn bản (*Model to Text - M2T*). Trong đó điển hình là kỹ thuật biểu diễn DM gắn trực tiếp vào các lớp miền, các ràng buộc thiết yếu được biểu diễn ngay trong ngữ cảnh của DM. Nhờ đó, DM vẫn giữ được tính biểu đạt cao trong UL mà không tách rời khỏi cấu trúc và ngữ nghĩa thực thi [70, 71].

Do đó, việc kế thừa và mở rộng hướng tiếp cận này bằng một phiên bản tinh chỉnh của phương pháp phát triển phần mềm dựa trên aDSL của DDD [70], theo hướng mở rộng biểu diễn mô hình miền cho hành vi nghiệp vụ, bảo mật và các ràng buộc nghiệp vụ ngoài những nghiệp vụ thiết yếu 2.1, nhằm tăng cường khả năng mô hình hóa và hỗ trợ sinh mã nhất quán, vẫn còn là một khoảng trống trong nghiên cứu.

2.1.3 Ngôn ngữ chuyên biệt miền DSL

Ngôn ngữ chuyên biệt miền (*Domain-Specific Language - DSL*) [43] là một ngôn ngữ được thiết kế để mô tả và thao tác trong một miền bài toán cụ thể, DSL được xây dựng với mục tiêu làm cho các khái niệm, quy tắc và thao tác của miền trở nên *tự nhiên* và *gần gũi* nhất với cách mà chuyên gia nghiệp vụ suy nghĩ và trao đổi, từ đó tăng khả năng biểu đạt, giảm khoảng cách giữa đặc tả và hiện thực.

Trong DDD, DSL đóng vai trò như một ngôn ngữ hình thức, hỗ trợ mô hình hóa miền, sinh mã tự động và kiểm soát sự tiến hóa của hệ thống.

DSL ngoại sinh là các ngôn ngữ chuyên biệt miền tồn tại độc lập với ngôn ngữ lập trình chủ, có cú pháp và ngữ nghĩa được xác định tường minh

cùng hệ sinh thái công cụ riêng. Nhờ đó, DSL ngoại sinh phù hợp cho phân tích mô hình ở mức trừu tượng cao, chuyển đổi mô hình giữa nhiều DSL và kiểm chứng ngữ nghĩa một cách có hệ thống [17].

DSL nội sinh được nhúng trong ngôn ngữ lập trình chủ và khai thác trực tiếp cú pháp, ngữ nghĩa của ngôn ngữ này để biểu đạt các khái niệm miền. Cách tiếp cận này giúp DSL nội sinh dễ tích hợp với mã nguồn và không yêu cầu xây dựng một ngữ nghĩa hình thức độc lập [13, 31, 132]. Các DSL nội sinh, đặc biệt là các tiếp cận dựa trên chú thích (aDSL), cho phép nhúng trực tiếp các đặc tả miền vào ngôn ngữ lập trình chủ như Java hoặc C# [119, 132]. Trong bối cảnh DDD, aDSL có vai trò quan trọng nhờ các đặc điểm sau:

- Gắn ngữ nghĩa miền trực tiếp vào lớp miền, trong đó mỗi chú thích vừa thể hiện ý nghĩa nghiệp vụ, vừa cung cấp thông tin cho các công cụ chuyển đổi mô hình và sinh mã.
- Thu hẹp khoảng cách giữa mô hình và mã nguồn, khi DM được thể hiện trực tiếp trong mã thay vì tồn tại tách rời dưới dạng UML/OCL, phù hợp với nguyên lý gắn kết mô hình và hiện thực của Evans.
- Hỗ trợ sinh tự động bản mẫu phần mềm, cho phép các công cụ như JDA (*Java Domain Application framework*) [71] khai thác trực tiếp chú thích để tạo các thành phần phần mềm mà không cần lớp mô hình trung gian.

Các tiếp cận này giúp rút ngắn khoảng cách giữa mô hình và triển khai, đồng thời hỗ trợ xây dựng các DM có khả năng thực thi ở một mức độ nhất định. Tuy nhiên, việc tích hợp đồng thời hành vi, ràng buộc và các chính sách miền trong một mô hình thống nhất vẫn còn nhiều hạn chế [70].

Trong DDD, UL không chỉ là ngôn ngữ giao tiếp mà còn phải được nhúng trực tiếp vào mô hình và mã nguồn, và DSL chính là phương tiện hiện thực hóa yêu cầu này: từ vựng của DSL–tên chú thích, thuộc tính, hành vi–được thiết kế trùng khớp với thuật ngữ nghiệp vụ, trong khi siêu mô hình và các quy tắc DSL mô tả chính xác cấu trúc và lô-gic của miền. Khi các bộ chuyển đổi và sinh mã nguồn được áp dụng, mọi tạo tác phần mềm sinh ra đều tuân thủ cùng một UL, giúp chuyên gia nghiệp vụ có thể đọc và xác nhận DM ngay trong DSL mà không cần phụ thuộc vào UML/OCL.

Ví dụ. Một aDSL của DCSL được sử dụng cho lớp `Student`. Chú thích `@DAttr(id = true, auto = true)` xác định `id` là khóa định danh của thực thể, với giá trị được tự động sinh. Chú thích `@DAttr(optional = false, length = 50)` quy định `name` là thuộc tính bắt buộc và áp đặt ràng buộc độ dài tối đa của giá trị.

```
1    public class Student {
2        @DAttr(id = true, auto = true)
3        private int id;
4        @DAttr(optional = false, length = 50)
5        private String name;
6        ...
7    }
```

2.1.4 DCSL và kiến trúc JDA

DCSL (*Domain-Class Specification Language - DCSL*) là một ngôn ngữ miền chuyên biệt miền dựa trên chú thích, được giới thiệu trong [70], dùng để đặc tả DM. Ngôn ngữ DCSL này cho phép mô tả ngắn gọn và dễ đọc các khái niệm miền, ràng buộc và quy tắc nghiệp vụ, với cú pháp gần giống cú pháp của OOPL. DCSL hỗ trợ biểu diễn các ràng buộc OCL thiết yếu [90]. Cụ thể, các chú thích được sử dụng để ghi nhận và mô tả các ràng buộc cốt lõi trong DM, như minh họa ở Bảng 2.1.

Trong ngôn ngữ DCSL được định nghĩa bởi các siêu khái niệm của nó bao gồm các cấu trúc cốt lõi của OOPL cùng các thành phần liên quan đến ràng buộc. Các siêu khái niệm chính bao gồm: lớp miền (`DClass`), thuộc tính miền (`DAttr`), thuộc tính liên kết (`DAssoc`), và phương thức miền (`DOpt`). Các siêu khái niệm còn lại, cùng với các thuộc tính của tất cả các khái niệm, được định nghĩa nhằm hỗ trợ biểu diễn các ràng buộc OCL thiết yếu được tóm tắt trong Bảng 2.1.

Việc áp dụng kiến trúc mô hình (*Model–View–Controller - MVC*) trong phát triển phần mềm thực tiễn là điều phổ biến, đặc biệt đối với các hệ thống cần giao diện người dùng đồ họa để hỗ trợ nhóm phát triển. Nguyên nhân là bởi người ta tin rằng quá trình xây dựng phần mềm không thể được tự động hóa hoàn toàn do sự tham gia của yếu tố con người trong vòng đời phát triển [44]. Kiến trúc MVC bao gồm ba thành phần: *model*, *view* và *controller*, trong đó mỗi thành phần có thiết kế nội tại độc lập và ảnh

Bảng 2.1: Các ràng buộc cấu trúc thiết yếu cho DM

Ràng buộc	**Loại**	**Mô tả**
Tính bất biến của đối tượng	Boolean	Đối tượng của một lớp có thể thay đổi hay không [74].
Tính bất biến của trường	Boolean	Một trường có thể thay đổi hay không (tức là giá trị của nó có thể bị thay đổi) [57].
Tính tùy chọn của trường	Boolean	Một trường có tùy chọn hay không (tức là giá trị của nó không cần được khởi tạo khi một đối tượng được tạo) [57].
Tính duy nhất của trường	Boolean	Các giá trị của một trường có duy nhất hay không [57].
Trường Id	Boolean	Một trường có phải là trường định danh của đối tượng hay không [57].
Trường tự động	Boolean	Giá trị của một trường có được tạo tự động bởi hệ thống hay không [57].
Độ dài của trường	Non-Boolean	Độ dài tối đa (nếu có) của một trường (tức là các giá trị của trường không được vượt quá độ dài này) [57].
Giá trị tối thiểu của trường	Non-Boolean	Giá trị tối thiểu (nếu có) của một trường (tức là các giá trị của trường không được thấp hơn giá trị này) [57].
Giá trị tối đa của trường	Non-Boolean	Giá trị tối đa (nếu có) của một trường (tức là các giá trị của trường không được cao hơn giá trị này) [57].
Số lượng đối tượng liên kết tối thiểu	Non-Boolean	Số lượng đối tượng tối thiểu mà mỗi đối tượng của một lớp có thể được liên kết [91].
Số lượng đối tượng liên kết tối đa	Non-Boolean	Số lượng đối tượng tối đa mà mỗi đối tượng của một lớp có thể được liên kết [91].

hưởng tối thiểu đến hai thành phần còn lại. Tính mô-đun có thể được tăng cường hơn nữa bằng cách áp dụng kiến trúc này ở cấp mô-đun, hình thành nên một kiến trúc thiết kế phân cấp trong đó phần mềm được cấu thành từ nhiều mô-đun theo cấu trúc cây.

Để xây dựng phần mềm theo DDD từ DM, cần một mô hình kiến trúc tuân theo kiến trúc phân lớp tổng quát [39, 121], trong đó DM được đặt

ở lớp lõi và tách biệt với giao diện người dùng cũng như các lớp còn lại, kiến trúc JDA [71] được đề xuất để giải quyết vấn đề này. Ngoài ra, các khung làm việc DDD hiện có [95, 119] cũng sử dụng kiến trúc MVC. Giao diện người dùng đóng vai trò quan trọng trong việc trình bày DM cho các bên liên quan và hỗ trợ họ xây dựng mô hình hiệu quả. Do đó, các công cụ DDD tuân theo kiến trúc phân lớp thường lựa chọn kiến trúc MVC như một khuôn mẫu triển khai phù hợp, nhằm hỗ trợ trình bày và tương tác hiệu quả với DM.

2.1.5 Phương pháp biểu diễn bằng siêu mô hình hóa

Trong mô hình hóa hướng miền, DSL đóng vai trò là phương tiện cốt lõi để biểu diễn DM một cách chính xác, nhất quán và có thể thực thi. Mỗi DSL được xây dựng dựa trên một siêu mô hình (*Metamodel*) nhằm mô tả rõ ràng các khía cạnh cấu trúc, ràng buộc và ngữ nghĩa của ngôn ngữ. Việc lựa chọn MOF/Ecore [29, 79] làm chuẩn khung siêu mô hình hóa bảo đảm rằng DSL có nền tảng hình thức đủ mạnh để kiểm chứng, phân tích và tham gia vào chuyển đổi mô hình theo chuẩn MDE [17]. Siêu mô hình xác định các phần tử ngôn ngữ, quan hệ giữa chúng và các ràng buộc cần tuân thủ, từ đó cung cấp một cấu trúc thống nhất cho việc định nghĩa, mở rộng và tạo nền tảng thống nhất để định nghĩa, mở rộng và tích hợp các DSL trong kiến trúc mô hình.

Từ góc nhìn của DDD, MOF/Ecore cung cấp cơ chế cho phép các khái niệm miền—đối tượng, quan hệ, hành vi, ràng buộc—được định nghĩa một cách chặt chẽ và ánh xạ trực tiếp vào DSL tương ứng, mang lại ba lợi ích sau:

- Thứ nhất, hỗ trợ bảo toàn ngữ nghĩa trong quá trình chuyển đổi mô hình, vì mọi chuyển đổi mô hình đều vận hành trên cấu trúc được đặc tả rõ ràng, tránh sai lệch khi ánh xạ từ mô hình này sang mô hình khác.
- Thứ hai, siêu mô hình cho phép kiểm tra tính đúng đắn và tính nhất quán của DM ở mức cú pháp và ngữ nghĩa
- Cuối cùng, siêu mô hình chính là cầu nối giúp DM chuyển hóa thành mô hình thực thi thông qua các bộ chuyển đổi mô hình và trở thành một phần của hệ thống chạy được.

Vì vậy, DSL và siêu mô hình hóa không chỉ là công nghệ hỗ trợ mô hình hóa mà còn là nền tảng phương pháp luận để bảo đảm DM giữ vai trò trung tâm, có thể diễn đạt chính xác kiến thức miền và được chuyển hóa một cách tin cậy thành phần mềm theo đúng tinh thần của DDD.

2.2 Các hướng tiếp cận biểu diễn mô hình miền

Trong thực tiễn, mô hình miền (DM) có thể được biểu diễn bằng các ngôn ngữ mô hình hóa tổng quát như UML/OCL, hoặc bằng các DSL theo các mức độ gắn kết khác nhau với mã nguồn. Mục này trình bày các tiếp cận biểu diễn phổ biến, nhấn mạnh ưu điểm về khả năng biểu đạt và chuẩn hóa, đồng thời chỉ ra hạn chế về tính hợp nhất và khả năng hướng thực thi của các biểu diễn hiện hành trong bối cảnh DDD.

2.2.1 Biểu diễn các ràng buộc trên mô hình miền

UML và OCL là các công cụ phổ biến để biểu diễn cấu trúc và ràng buộc của DM trong DDD. UML cho phép mô tả các khái niệm miền và mối quan hệ giữa chúng, trong khi OCL cung cấp cơ chế đặc tả chính xác các bất biến và điều kiện nghiệp vụ [91]. Tuy nhiên, các đặc tả này thường tồn tại tách rời khỏi mã nguồn và thiếu khả năng hỗ trợ trực tiếp cho việc thực thi DM.

Trong bối cảnh DDD, biểu đồ lớp UML thường được dùng để phác thảo DM ở mức khái niệm, giúp các bên liên quan thảo luận và thống nhất về các thực thể miền, thuộc tính, liên kết và các cấu trúc tổng hợp. Tuy nhiên, UML chủ yếu mạnh ở việc mô tả cấu trúc; các quy tắc nghiệp vụ và các điều kiện hợp lệ của miền thường không thể hiện đầy đủ chỉ bằng quan hệ và bội số.

Để bổ sung khả năng đặc tả chính xác các quy tắc và ràng buộc nghiệp vụ, OCL thường được sử dụng kèm theo UML [90]. OCL cho phép mô tả một cách tường minh các bất biến, điều kiện trước/sau của trạng thái, và các ràng buộc dẫn xuất dựa trên ngữ nghĩa của mô hình UML [91]. Nhờ đặc tính khai báo và có cú pháp hình thức, OCL giúp giảm tính mơ hồ so với mô tả bằng ngôn ngữ tự nhiên, đồng thời hỗ trợ phân tích tĩnh, kiểm tra tính nhất quán và phát hiện vi phạm ràng buộc ở giai đoạn thiết kế.

Mặc dù UML/OCL cung cấp một nền tảng chuẩn hóa và giàu khả năng mô tả cho DM, cách tiếp cận này bộc lộ một số hạn chế khi đặt trong yêu cầu "mô hình miền hướng thực thi" của DDD. Thứ nhất, UML phân tách cấu trúc, hành vi và ràng buộc thành các không gian biểu diễn khác nhau (biểu đồ lớp, biểu đồ hành vi và đặc tả OCL), khiến tri thức miền bị phân mảnh và khó duy trì tính nhất quán khi mô hình tiến hóa. Thứ hai, các đặc tả UML/OCL thường tồn tại tách rời khỏi mã nguồn; do đó, việc hiện thực hóa DM từ UML/OCL vẫn chủ yếu phụ thuộc vào thao tác thủ công của nhà phát triển, làm gia tăng nguy cơ sai lệch ngữ nghĩa giữa đặc tả và hệ thống triển khai. Thứ ba, mặc dù OCL có thể được dùng để kiểm tra ràng buộc ở mức mô hình, nhưng bản thân UML/OCL không cung cấp một cơ chế thống nhất để đưa DM đến trạng thái có khả năng thực thi hoặc phục vụ sinh mã một cách trực tiếp trong quy trình DDD. Các công cụ MDE có thể hỗ trợ sinh mã từ UML và dịch một phần ràng buộc OCL sang các đoạn kiểm tra, tuy nhiên việc chuyển hóa này thường phụ thuộc vào công cụ, quy ước triển khai và phạm vi ràng buộc được hỗ trợ; đồng thời khó đảm bảo rằng ngữ nghĩa nghiệp vụ được bảo toàn xuyên suốt từ DM đến hệ thống chạy được. Vì vậy, UML/OCL là tiếp cận nền tảng để biểu diễn cấu trúc và ràng buộc của DM nhờ tính chuẩn hóa và khả năng đặc tả hình thức. Tuy nhiên, trong bối cảnh DDD, việc biểu diễn phân tách và tách rời khỏi mã nguồn khiến UML/OCL chưa đáp ứng đầy đủ yêu cầu về một DM hợp nhất, hướng thực thi và có khả năng làm trung tâm cho tự động hóa phát triển phần mềm.

Nhiều nghiên cứu đã tập trung vào việc chuyển đổi các biểu đồ lớp và các ràng buộc nghiệp vụ được đặc tả bằng OCL thành mã thực thi hoặc các cấu trúc tương ứng trong OOPL. Công trình của Cabot và Teniente [24] đề xuất sinh mã Java từ các ràng buộc OCL bằng cách dịch chúng thành các truy vấn SQL; cách tiếp cận này chủ yếu hướng đến việc kiểm tra tính hợp lệ ở mức cơ sở dữ liệu, thay vì tích hợp các ràng buộc trực tiếp vào DM trong bộ nhớ. Tương tự, Rackov và cộng sự [96] khảo sát việc sinh mã Java dựa trên các quy tắc OCL, trong đó các ràng buộc được hiện thực hóa thành các phương thức kiểm tra độc lập, dẫn đến sự tách rời giữa lô-gic ràng buộc và cấu trúc DM.

Một số công trình khác [27, 52] đề xuất các khung làm việc dịch OCL sang các chú thích Java nhằm hỗ trợ tích hợp ràng buộc vào mã nguồn. Tuy

nhiên, các giải pháp này thường yêu cầu sự can thiệp thủ công đáng kể khi xử lý các ràng buộc phức tạp. Công cụ `OCL2Java`, mặc dù hỗ trợ bán tự động, chủ yếu chỉ xử lý được các mẫu OCL đơn giản và thiếu hỗ trợ cho các cấu trúc như `exists`, `forAll` và các phép toán trên tập hợp. Gần đây hơn, một số nghiên cứu [7, 18, 31, 87] đã kết hợp OCL với cơ chế chú thích để biểu diễn DM theo tinh thần DDD. Ngoài ra, phương pháp `OCL2MSFOL` [33] chuyển các ràng buộc OCL sang lô-gic vị từ bậc nhất nhằm phục vụ kiểm chứng hình thức, nhưng không xem xét việc hiện thực và tích hợp các ràng buộc này trong DM thực thi.

Tổng hợp các công trình trên cho thấy các giải pháp chuyển đổi và hiện thực hóa ràng buộc OCL vẫn còn nhiều thách thức. Nhiều phương pháp mới chỉ đạt mức bán tự động và thường phải can thiệp thủ công khi gặp các ràng buộc phức tạp. Bên cạnh đó, phạm vi hỗ trợ OCL trong thực tế còn hạn chế, đặc biệt với các ràng buộc nghiệp vụ giàu ngữ nghĩa trên lớp, thuộc tính, quan hệ và phương thức [27]. Ngoài ra, nhiều giải pháp chủ yếu tập trung vào cơ chế kiểm tra ràng buộc, trong khi ít xem xét dẫn xuất giá trị hay cơ chế lan truyền thay đổi theo sự tiến hóa trạng thái của mô hình, làm suy giảm mức độ gắn kết giữa lô-gic ràng buộc và thực thi miền. Cuối cùng, việc phụ thuộc vào các công cụ và bước sinh mã chuyên biệt làm tăng độ phức tạp của quy trình, đồng thời gây khó khăn cho bảo trì khi mô hình tiếp tục tiến hóa.

Trên cơ sở các hạn chế đó, kỹ thuật biểu diễn ràng buộc dựa vào mẫu chú thích (gọi là CAP) trong luận án được đề xuất như một cách tiếp cận dựa trên chú thích nhằm đặc tả cấu trúc và ngữ nghĩa của các ràng buộc OCL trực tiếp trong ngữ cảnh của DM thực thi được trình bày trong Mục 3.2. Cách tiếp cận này hướng đến việc tăng cường tính rõ ràng và khả năng hiểu theo ngữ cảnh của các ràng buộc, đồng thời cải thiện khả năng cộng tác, truy vết và mức độ gắn kết giữa các ràng buộc nghiệp vụ và DM trong các hệ thống phần mềm phức tạp.

2.2.2 Biểu diễn khía cạnh hành vi miền

Hành vi miền là thành phần cốt lõi phản ánh cách các khái niệm miền tương tác, tiến hóa trạng thái và hiện thực hóa các quy trình nghiệp vụ [15]. Trong DDD, hành vi không chỉ là các thao tác kỹ thuật mà còn mang ý nghĩa

nghiệp vụ rõ ràng, gắn chặt với ngôn ngữ chung và tri thức miền [39, 121]. Do đó, việc biểu diễn và tích hợp hành vi vào DM là điều kiện cần để DM có thể đóng vai trò trung tâm ngữ nghĩa và hướng thực thi.

(i) Biểu diễn hành vi bằng UML tách rời cấu trúc. Cách tiếp cận phổ biến nhất hiện nay là sử dụng các biểu đồ hành vi của UML, chẳng hạn biểu đồ hoạt động, biểu đồ trạng thái hoặc biểu đồ tuần tự, để mô tả hành vi nghiệp vụ [91, p. 373]. Các biểu đồ này cho phép thể hiện luồng điều khiển, sự thay đổi trạng thái và tương tác giữa các đối tượng ở mức trực quan [102]. Tuy nhiên, trong thực tiễn, các biểu đồ hành vi thường tồn tại như các tạo tác tách rời khỏi biểu đồ lớp mô tả cấu trúc miền. Hành vi vì vậy không được gắn kết trực tiếp với các thực thể miền cụ thể, mà chủ yếu đóng vai trò tài liệu thiết kế hoặc hỗ trợ trao đổi. Sự tách rời này dẫn đến khó khăn trong việc duy trì tính nhất quán giữa cấu trúc và hành vi, đặc biệt khi mô hình tiến hóa, đồng thời hạn chế khả năng sử dụng trực tiếp các biểu đồ hành vi như một phần của DM có khả năng thực thi.

(ii) Biểu diễn hành vi bằng DSL. Một hướng tiếp cận khác là định nghĩa các DSL cho hành vi miền, cho phép mô tả các quy trình nghiệp vụ, luồng hoạt động hoặc máy trạng thái bằng cú pháp và thuật ngữ gần với ngôn ngữ nghiệp vụ. Các DSL này giúp tăng khả năng biểu đạt và tạo điều kiện cho chuyên gia miền tham gia trực tiếp vào việc đặc tả hành vi [83]. Trong bối cảnh MDE, hành vi được đặc tả bằng DSL có thể được chuyển đổi sang mã thực thi thông qua các bộ chuyển đổi M2M hoặc M2T. Tuy nhiên, các DSL hành vi thường được thiết kế độc lập với DSL cấu trúc hoặc mô hình lớp UML, dẫn đến việc hành vi và cấu trúc được biểu diễn trong các không gian ngữ nghĩa khác nhau. Việc hợp nhất các DSL này đòi hỏi các cơ chế ánh xạ và phối hợp phức tạp, và trong nhiều trường hợp chưa có một nền tảng ngữ nghĩa thống nhất để đảm bảo rằng hành vi được diễn giải đúng trên cấu trúc miền tương ứng.

(iii) Tích hợp hành vi thông qua DSL nội sinh. Gần hơn với tinh thần của DDD, một số tiếp cận dựa trên DSL nội sinh, đặc biệt là các DSL dựa trên chú thích (aDSL), cho phép gắn hành vi trực tiếp vào các lớp miền trong ngôn ngữ lập trình chủ [43, 89]. Theo hướng này, hành vi được biểu diễn ngay trong ngữ cảnh của thực thể miền, giúp rút ngắn khoảng cách giữa mô hình và hiện thực, đồng thời tạo ra các DM có khả năng thực thi ở một mức độ nhất định. Tuy nhiên, phần lớn các tiếp cận aDSL hiện nay mới

chỉ hỗ trợ hành vi ở mức thao tác cục bộ hoặc các quy ước lập trình, chưa cung cấp một cơ chế hình thức để mô tả hành vi nghiệp vụ phức hợp, có ngữ nghĩa rõ ràng và có thể kiểm chứng, cũng như chưa giải quyết triệt để việc hợp nhất hành vi với các mối quan tâm khác như ràng buộc hay bảo mật trong một DM hợp nhất.

Vấn đề ngữ nghĩa chung khi hợp nhất hành vi và cấu trúc. Từ các tiếp cận trên có thể thấy rằng thách thức cốt lõi không chỉ nằm ở việc biểu diễn hành vi, mà ở chỗ thiếu một nền tảng ngữ nghĩa chung để hợp nhất hành vi với cấu trúc miền. Khi hành vi được mô tả tách rời hoặc bằng các ngôn ngữ khác nhau, việc diễn giải hành vi trên trạng thái của DM trở nên không rõ ràng và khó kiểm chứng. Điều này khiến DM khó được xem như một hệ thống chuyển trạng thái thống nhất, trong đó cấu trúc xác định không gian trạng thái và hành vi xác định các phép chuyển trạng thái hợp lệ.

Do đó, trong bối cảnh DDD, cần có các kỹ thuật biểu diễn và hợp nhất hành vi cho phép: (i) gắn hành vi trực tiếp với các khái niệm miền; (ii) diễn giải hành vi trên cùng một không gian ngữ nghĩa với cấu trúc và các mối quan tâm khác; và (iii) tạo nền tảng cho việc kiểm chứng và chuyển đổi mô hình hướng tới các DM có khả năng thực thi.

Trong lĩnh vực kỹ nghệ DSL, nhiều nghiên cứu đã đề xuất các cách phân loại DSL theo phạm vi áp dụng và mối quan hệ với ngôn ngữ chủ [8, 43, 126]. Trên cơ sở đó, kỹ thuật tích hợp hành vi vào mô hình miền (gọi là AGL) mà luận án đề xuất, được trình bày trong Mục 3.3 có định vị như một DSL nội sinh và ngang, tập trung vào một miền con cụ thể là các đồ thị hoạt động. Điểm khác biệt của AGL không nằm ở việc mở rộng hay tinh chỉnh các tiêu chí phân loại DSL nói chung, mà ở cách khai thác trực tiếp các đặc trưng hình thành nên miền đồ thị hoạt động để phục vụ cho việc mô hình hóa hành vi miền trong bối cảnh DDD.

Ý tưởng kết hợp DDD với DSL nhằm nâng cao mức trừu tượng của mô hình phần mềm đã được gợi mở trong [43]. Tuy nhiên, công trình này không đề xuất một cơ chế cụ thể để tích hợp hành vi miền như một phần của DM thực thi. Trong luận án này, AGL được đề xuất nhằm lấp đầy khoảng trống đó bằng cách kết hợp mô hình lớp hợp nhất với mô hình đồ thị hoạt động trong cùng một DM thống nhất, trong đó cấu trúc và hành vi được biểu diễn bằng hai aDSL bổ trợ cho nhau là DCSL [70] và AGL.

Mặc dù Evans [39] không đề cập tường minh việc mô hình hóa hành vi như một thành phần độc lập của phương pháp DDD, tác giả nhấn mạnh vai trò thiết yếu của hành vi đối tượng trong DM và sử dụng các biểu đồ tương tác UML làm ví dụ minh họa. Trong các hiện thực cụ thể của DDD như Apache Isis [119], một ngôn ngữ hành động đơn giản được sử dụng để đặc tả hành vi của đối tượng thông qua các chú thích trong ngôn ngữ lập trình hướng đối tượng. Cách tiếp cận này có thể được xem là một hiện thực cụ thể của ngôn ngữ con hành động trong biểu đồ hoạt động UML [91, p. 441], nhưng không cung cấp một phương pháp mô hình hóa hành vi có cấu trúc ở mức mô hình. Trái lại, AGL cung cấp một biểu diễn hành vi miền ở mức mô hình, cho phép tích hợp trực tiếp các đặc tả hành vi vào mô hình miền hợp nhất.

Trong thiết kế hướng đối tượng truyền thống, các khía cạnh hành vi thường được xử lý thông qua các mẫu thiết kế hành vi [16], vốn chủ yếu hoạt động ở mức giải pháp và hiện thực nhằm tổ chức sự tương tác giữa các đối tượng. Do đó, hành vi miền thường bị phân tán trong các lớp và không được xem như một thực thể mô hình hóa hạng nhất. Một số nghiên cứu gần đây đã nhấn mạnh nhu cầu về các trừu tượng hành vi có khả năng thực thi và được tích hợp trực tiếp với DM; tuy nhiên, các ngôn ngữ tiếp cận theo hướng này, chẳng hạn như `ReMoDeL` [110], có xu hướng tách rời hành vi khỏi cấu trúc miền hướng đối tượng. AGL khác biệt ở chỗ định vị hành vi như một mối quan tâm miền hạng nhất, được biểu diễn dưới dạng đồ thị hoạt động có khả năng thực thi và được tích hợp trực tiếp với các khía cạnh cấu trúc được đặc tả bằng DCSL.

Về mặt ngữ nghĩa, một số công trình đã nghiên cứu việc kết hợp các biểu đồ cấu trúc và hành vi của UML nhằm xây dựng mô hình hệ thống hoàn chỉnh, chẳng hạn thông qua sự kết hợp giữa biểu đồ lớp và biểu đồ máy trạng thái [6, 85]. AGL có điểm tương đồng với các tiếp cận này ở mục tiêu hợp nhất cấu trúc và hành vi, nhưng khác biệt ở cách tiếp cận: thay vì dựa trên máy trạng thái, AGL sử dụng các đồ thị hoạt động để nhấn mạnh khía cạnh hành vi thực thi, trong đó trạng thái được xem xét thông qua mối quan hệ trước–sau của các mô-đun hoạt động [91, p. 305]. Ngoài ra, luận án cũng khai thác cách tiếp cận MVC ở mức *vi mô* để tổ chức các mô-đun phần mềm, sau đó kết hợp các mô-đun này thông qua các đồ thị hoạt động, qua đó tạo ra một cơ chế tích hợp hành vi phù hợp với DM hợp nhất.

Cuối cùng, trong khi nhiều công trình trong lĩnh vực kỹ nghệ phần mềm hướng mô hình sử dụng nhiều DSL khác nhau để xây dựng các mô hình phần mềm hoàn chỉnh [17, 66], các DSL này thường không phân định rõ vai trò của DM so với các mô hình khác trong hệ thống. Khác với các tiếp cận đó, AGL trong luận án này được thiết kế như một aDSL nội sinh, gắn chặt với ranh giới của DM theo tinh thần DDD và chỉ được sử dụng cùng với DCSL để xây dựng DM hợp nhất, thay vì mở rộng sang các DSL phục vụ các mối quan tâm ngoài miền.

2.2.3 Biểu diễn khía cạnh bảo mật với RBAC

Kiểm soát truy cập dựa trên vai trò (*Role-Based Access Control - RBAC*) [9, 58] gắn các quyền với các vai trò phản ánh trách nhiệm trong tổ chức, và người dùng được gán vào các vai trò này. Cách phân tách này đơn giản hóa quản trị, hỗ trợ nguyên lý đặc quyền tối thiểu và cải thiện khả năng bảo trì. Trong bối cảnh các hệ thống có yêu cầu phân quyền phức tạp, đặc biệt là các hệ thống với vai trò người dùng động, kiểm soát truy cập trở thành một vấn đề trung tâm [82]. Việc tách rời các chính sách bảo mật khỏi lô-gic nghiệp vụ lõi dẫn đến sự không nhất quán, khiến các chính sách này khó bảo trì và khó kiểm chứng [91, 93]. Trong bối cảnh DDD, RBAC không chỉ được xem là một mối quan tâm bảo mật độc lập, mà còn ảnh hưởng trực tiếp đến hành vi của các thực thể miền; do đó, RBAC cần được đặc tả tường minh trong DM [46, 73]. Theo đó, việc tích hợp RBAC vào DM trở thành một yêu cầu thiết yếu đối với các hệ thống DDD.

Theo tiêu chuẩn RBAC [9], mô hình RBAC lõi được định nghĩa dựa trên các khái niệm cơ bản sau:

- **Người dùng.** Biểu diễn một thực thể hoạt động (con người hoặc hệ thống) tương tác với hệ thống và yêu cầu truy cập tới các tài nguyên được bảo vệ.
- **Vai trò.** Biểu diễn một chức năng công việc hoặc trách nhiệm trong tổ chức (ví dụ: *Author*, *Reviewer*, *Editor*, *Subscriber*). **Vai trò** là một lớp trừu tượng trung gian giữa người dùng và quyền.

- **Quyền.** Biểu diễn một sự cho phép thực hiện một thao tác cụ thể trên một tài nguyên được bảo vệ. Phù hợp với mô hình RBAC, **Quyền** được định nghĩa dưới dạng các cặp ⟨*Hành động*, *Tài nguyên được bảo vệ*⟩.

- **Phiên.** Biểu diễn một ngữ cảnh thời gian chạy trong đó một người dùng kích hoạt một tập con các vai trò đã được gán cho người dùng đó. Một người dùng có thể có nhiều phiên đồng thời, mỗi phiên có thể có một tập vai trò đang hoạt động khác nhau.

- **Ràng buộc phân tách nhiệm vụ.** Các ràng buộc phân tách nhiệm vụ (*Separation of Duty - SoD*) này được sử dụng để ngăn ngừa xung đột lợi ích và cưỡng chế các chính sách được tổ chức:

  – *Phân tách nhiệm vụ tĩnh (Static Separation of Duty - SSD).* Hạn chế các gán vai trò xung đột bằng cách ngăn người dùng được gán vào các vai trò loại trừ lẫn nhau.

  – *Phân tách nhiệm vụ động (Dynamic Separation of Duty - DSD).* Hạn chế việc kích hoạt các vai trò xung đột bằng cách ngăn các vai trò loại trừ lẫn nhau được kích hoạt trong cùng một phiên.

- **Quy tắc truy cập.** Xác định điều kiện chấp nhận một yêu cầu truy cập, theo đó yêu cầu được chấp nhận nếu tồn tại một vai trò đang hoạt động trong phiên hiện tại cung cấp quyền cần thiết, đồng thời tất cả các điều kiện ngữ cảnh hoặc chính sách liên quan đều được thỏa mãn.

Một vấn đề thường gặp là RBAC thường được tích hợp vào các DM theo cách không chính thức hoặc thủ công. Li et al. [73] chỉ ra sự thiếu hụt hỗ trợ hình thức cho việc tích hợp RBAC trong các hệ thống đa miền, đồng thời cho rằng các cách tiếp cận thủ công thường dẫn đến sự không nhất quán. Tương tự, Oruc et al. [93] nghiên cứu việc tích hợp RBAC thủ công trong các hệ thống phức tạp như hệ đa tác tử kết hợp với chuỗi khối, và đề xuất một giải pháp dựa trên DSL.

Việc biểu diễn các chính sách như SoD [4] hoặc các trạng thái truy cập động thường chỉ giới hạn trong mã nguồn hoặc các ghi chú tài liệu, thiếu cơ chế kiểm chứng hình thức. Điều này có thể dẫn đến các vi phạm chính sách không được phát hiện trong các giai đoạn thiết kế hoặc kiểm thử.

Do đó, việc tăng cường khả năng biểu đạt của DM nhằm tự động tích hợp RBAC cùng với một cơ chế kiểm chứng đầy đủ và có thể chứng minh được có thể được xem là một trọng tâm nghiên cứu quan trọng hiện nay.

2.2.4 Mô tả ngữ nghĩa thực thi của DM với Event-B

Event-B là một phương pháp mô hình hóa hình thức dựa trên trạng thái, được phát triển từ phương pháp B [2, 80], dựa trên lô-gic bậc nhất và lý thuyết tập có kiểu để đặc tả và kiểm chứng các hệ thống rời rạc. Phương pháp này tách biệt rõ ràng các khía cạnh tĩnh trong ngữ cảnh và các khía cạnh động trong máy trạng thái, nơi hành vi hệ thống được mô hình hóa thông qua các sự kiện có điều kiện bảo vệ và được ràng buộc bởi các bất biến. Tính đúng đắn của mô hình được đảm bảo chặt chẽ thông qua cơ chế tinh chỉnh và tập các nghĩa vụ chứng minh (*Proof Obligations - POs*) [123].

Trên cơ sở Event-B, UML-B kết hợp UML với Event-B nhằm hỗ trợ kiểm chứng hình thức các mô hình phần mềm bằng cách ánh xạ các mô hình UML sang Event-B [111]. Tuy nhiên, UML-B chủ yếu tập trung vào tính đúng đắn ở mức hệ thống và chưa đặt DM làm trung tâm ngữ nghĩa theo tinh thần của DDD, cũng như chưa hỗ trợ tích hợp các mối quan tâm miền như hành vi và bảo mật trong một DM hợp nhất.

Trong bối cảnh đó, các chính sách kiểm soát truy cập như SSD, DSD và các ràng buộc động có thể được biểu diễn một cách tự nhiên trong Event-B dưới dạng các bất biến hoặc các điều kiện bảo vệ trên các quan hệ phân quyền. Ví dụ, quan hệ gán vai trò cho người dùng (UA) được ràng buộc để chỉ cho phép các cặp hợp lệ giữa người dùng (USR) và vai trò (R): $inv1 : UA \subseteq USR \times R$; Giả sử $r_1, r_2 \in R$ là hai vai trò xung đột tĩnh. $inv2 : \forall u \in USR \cdot \neg\big((u, r_1) \in UA \wedge (u, r_2) \in UA\big)$

Khả năng tự động sinh và kiểm chứng các PO bằng công cụ Rodin [3] giúp phát hiện các lỗi lô-gic hoặc xung đột trong DM ngay từ giai đoạn thiết kế, mà không cần chờ tới hiện thực. Hơn nữa, Event-B có thể chứng minh rằng các chính sách phân quyền động được duy trì trong suốt quá trình tinh chỉnh, từ đặc tả trừu tượng đến hiện thực.

Trong bối cảnh tích hợp với DDD và các DSL, Event-B đảm nhiệm vai trò kiểm chứng hình thức ở tầng xử lý phía sau bằng cách chuyển các khái

niệm chính sách từ DSL thành các biến và các điều kiện bất biến tương ứng [5, 65, 78, 98, 123]. Liên kết này cho phép phân tích và kiểm tra tự động các mâu thuẫn trong chính sách RBAC đã được định nghĩa. Quan trọng hơn, nó đảm bảo rằng mã nguồn được sinh ra từ DSL luôn tuân thủ các quy tắc bảo mật đã được chứng minh, đồng thời duy trì sự đồng bộ chặt chẽ giữa DM và các bản mẫu phần mềm được kiểm chứng hình thức.

2.3 Các hướng tiếp cận tích hợp mối quan tâm trong mô hình miền

Một mối quan tâm được hiểu là một khía cạnh tương đối độc lập của miền, biểu diễn một tập yêu cầu hoặc thuộc tính cần được mô hình hóa riêng biệt với ngữ nghĩa đầy đủ. Theo đó, mỗi mối quan tâm thường được đặc tả bằng một DSL riêng–được thiết kế với cú pháp và ngữ nghĩa phù hợp cho khía cạnh đó. Các DSL này, chẳng hạn cho cấu trúc hoặc cho hành vi, cần được hợp nhất một cách có hệ thống để hình thành một mô hình miền (DM) hợp nhất có khả năng thực thi, bảo đảm tính mô-đun, khả năng tích hợp và hỗ trợ tiến hóa hệ thống.

Trong kỹ nghệ phần mềm, việc đặc tả và tích hợp các mối quan tâm nhằm quản lý các khía cạnh xuyên suốt của hệ thống như cấu trúc, hành vi, bảo mật hay hiệu năng. Mặc dù phân tách mối quan tâm là một nguyên tắc thiết kế quan trọng, thách thức cốt lõi nằm ở việc hợp nhất các mối quan tâm trở lại thành một DM thống nhất, có khả năng thực thi và bảo toàn ngữ nghĩa nghiệp vụ. Nhiều nghiên cứu đã đề xuất các kỹ thuật hợp nhất DSL và siêu mô hình [89, 99, 125]; tuy nhiên, phần lớn các tiếp cận này tập trung vào cú pháp hoặc cấu trúc ngôn ngữ, chưa giải quyết triệt để bài toán hợp nhất mối quan tâm trong phạm vi một DM có khả năng thực thi.

Việc hiện thực hóa các DM hợp nhất thành các tạo tác phần mềm có khả năng thực thi trong thực tiễn có thể được thực hiện theo nhiều cách tiếp cận kiến trúc khác nhau, phụ thuộc vào mục tiêu phát triển cũng như góc nhìn của các bên liên quan. Trong luận án này, các DM được hiện thực hóa dựa trên kiến trúc khung phần mềm dựa trên mô-đun (Module-Based Software Architecture – MOSA) [72], một kiến trúc tuân theo phong cách MVC. MOSA cung cấp một khung kiến trúc thực tiễn cho việc tổ chức,

hiện thực và thực thi các DM trong các hệ thống phần mềm phát triển theo phương pháp DDD, qua đó thu hẹp khoảng cách giữa DM và hiện thực triển khai. Tuy nhiên, MOSA chủ yếu tập trung vào khía cạnh hiện thực và thực thi DM, mà chưa cung cấp một cơ chế hình thức để kiểm chứng việc các mối quan tâm bảo mật, chẳng hạn như RBAC, có thỏa mãn đầy đủ các ràng buộc an toàn đã được đặc tả hay không.

2.3.1 Tích hợp các mối quan tâm trong mô hình miền

Trong các hệ thống phần mềm thực tế, DM không chỉ mô tả lô-gic nghiệp vụ mà còn phải bao quát nhiều mối quan tâm khác như hành vi, bảo mật, ghi nhật ký hay hiệu năng. DM vì vậy thường được làm giàu và chính xác hóa thông qua các DSL [43, 63], bao gồm DSL ngoại sinh với cú pháp trừu tượng [8, 17] và DSL nội sinh với cú pháp cụ thể dạng đồ họa hoặc văn bản [13, 31, 132].

Việc định nghĩa và tích hợp nhiều DSL chuyên biệt theo mối quan tâm–như DSL cho lô-gic nghiệp vụ, bảo mật, giao diện người dùng hoặc hiệu năng–là thiết yếu để nắm bắt đầy đủ các yêu cầu đa dạng của các hệ thống phần mềm phức tạp [17, 51, 81]. Tuy nhiên, các nghiên cứu hiện có thường tập trung vào: (i) định nghĩa DSL để mô tả chính xác DM nhưng vẫn phải dịch ngữ nghĩa hoạt động sang ngôn ngữ lập trình; (ii) sử dụng DSL nội sinh, đặc biệt là các cơ chế dựa trên chú thích, để biểu diễn DM có khả năng thực thi; hoặc (iii) nghiên cứu việc phối hợp nhiều DSL theo mối quan tâm. Các tiếp cận này hoặc thiên về khả năng thực thi, hoặc tập trung vào tách/ghép mối quan tâm, nhưng chưa giải quyết triệt để bài toán hợp nhất các quan điểm mối quan tâm vào một DM hợp nhất thực thi theo đúng nguyên lý DDD [120].

Các mối quan tâm xuyên suốt như bảo mật và hiệu năng thường lan tỏa qua nhiều thành phần của DM và không thể được biểu diễn cục bộ trong một cấu trúc đơn lẻ. Đây là các mối quan tâm phi chức năng hoặc bổ trợ, không thuộc lô-gic cốt lõi của miền nhưng có ảnh hưởng đáng kể đến nhiều phần của mô hình. Do đó, chúng cần được mô hình hóa và tích hợp nhất quán thông qua các cơ chế hợp thành phù hợp nhằm đảm bảo tính khả thi, tính nhất quán ngữ nghĩa và khả năng thực thi của DM hợp nhất.

Các nỗ lực gần đây dựa trên phương pháp siêu mô hình để tổng hợp các DSL chuyên biệt theo mối quan tâm đã giúp hình thành các siêu mô hình cố kết và ánh xạ một–một sang các aDSL nội sinh nhúng trong OOPL [6]. Tuy nhiên, các tiếp cận này vẫn gặp nhiều hạn chế như quá trình tích hợp mang tính bán tự động, dễ phát sinh lỗi, phụ thuộc mạnh vào chất lượng ánh xạ và chưa hỗ trợ hiệu quả các mối quan tâm xuyên suốt trong các hệ thống phức tạp.

Một hướng tiếp cận khác là tổng hợp các DSL chuyên biệt theo mối quan tâm dựa trên cây cú pháp trừu tượng (*Abstract Syntax Tree - AST*) [19], khai thác tính linh hoạt và khả năng thao tác trực tiếp trên cấu trúc ngôn ngữ để hợp nhất nhiều góc nhìn miền. Tuy nhiên, việc định nghĩa đầy đủ cú pháp, ngữ nghĩa cho từng DSL và xây dựng cơ chế hợp thành dựa trên chú thích ở mức AST nhằm tạo ra một DM hợp nhất có khả năng thực thi vẫn là một thách thức nghiên cứu mở.

2.3.2 Tích hợp hành vi miền

Trong DDD, hành vi nghiệp vụ là một thành phần cốt lõi phản ánh động lực và sự tiến hóa trạng thái của DM [39, 127]. Các hành vi này thường gắn liền với các quy trình nghiệp vụ, các quy tắc miền và các điều kiện ngữ cảnh, đồng thời chi phối cách các thực thể miền tương tác và thay đổi theo thời gian. Do đó, việc biểu diễn và tích hợp hành vi miền một cách nhất quán vào DM là yêu cầu thiết yếu để đảm bảo rằng mô hình không chỉ mô tả cấu trúc tĩnh mà còn phản ánh đầy đủ lô-gic nghiệp vụ của hệ thống.

Tuy nhiên, trong nhiều tiếp cận DDD hiện nay [49, 59, 107, 122, 135], DM chủ yếu tập trung vào khía cạnh cấu trúc, trong khi hành vi miền thường được xử lý theo các cách tách rời. Một hướng tiếp cận phổ biến là mô tả hành vi bằng các biểu đồ UML như biểu đồ hoạt động, trạng thái hoặc tuần tự [8, 116, 124]. Các biểu đồ này cung cấp cái nhìn trực quan về luồng xử lý, nhưng thường tồn tại như các tạo tác độc lập, không được tích hợp chặt chẽ với cấu trúc miền và thiếu một ngữ nghĩa thống nhất để hỗ trợ khả năng thực thi hoặc kiểm chứng.

Một hướng khác là sử dụng các DSL cho hành vi nhằm mô tả các quy trình hoặc luồng nghiệp vụ ở mức trừu tượng cao [106]. Mặc dù các DSL này giúp tăng khả năng biểu đạt hành vi và hỗ trợ tự động hóa ở một mức

độ nhất định, chúng thường được thiết kế và vận hành như các ngôn ngữ độc lập. Việc hợp nhất ngữ nghĩa hành vi với cấu trúc miền vì vậy phải dựa vào các cơ chế ánh xạ hoặc chuyển đổi bổ sung, làm gia tăng độ phức tạp và nguy cơ sai lệch ngữ nghĩa.

Trong các tiếp cận dựa trên DSL nội sinh là aDSL, hành vi miền có thể được gắn trực tiếp vào các lớp miền thông qua các cấu trúc ngôn ngữ của ngôn ngữ lập trình chủ. Cách tiếp cận này giúp rút ngắn khoảng cách giữa mô hình và hiện thực, đồng thời hỗ trợ khả năng thực thi [95, 119]. Tuy nhiên, hành vi miền trong trường hợp này thường bị pha trộn với chi tiết kỹ thuật hoặc bị phân tán trong mã nguồn, khiến DM khó giữ vai trò là một đặc tả nghiệp vụ rõ ràng và nhất quán.

Nhìn chung, các tiếp cận hiện có cho thấy một vấn đề chung: hành vi miền chưa được tích hợp một cách hệ thống vào DM với một nền tảng ngữ nghĩa thống nhất. Sự thiếu vắng một cơ chế hợp nhất hành vi với cấu trúc miền khiến mô hình khó được xem như một thực thể ngữ nghĩa hoàn chỉnh, vừa có khả năng diễn đạt nghiệp vụ, vừa có khả năng thực thi và kiểm chứng. Khoảng trống này đặt ra nhu cầu nghiên cứu các kỹ thuật biểu diễn hành vi miền gắn chặt với DM, đồng thời hỗ trợ hợp nhất ngữ nghĩa hành vi và cấu trúc trong một DM hướng thực thi theo tinh thần của DDD.

2.3.3 Tích hợp ràng buộc và chính sách bảo mật

UML/OCL [90, 91] mạnh về mô tả, nhưng chúng chưa đáp ứng yêu cầu của DM có thể thực thi. Các DM thường được biểu diễn bằng biểu đồ lớp UML/OCL [91] nhằm mô tả các khái niệm miền và mối quan hệ giữa chúng. Nhiều nghiên cứu về khả năng sử dụng OCL cho thấy cú pháp và cách diễn đạt của ngôn ngữ này gây khó khăn đáng kể đối với những người không chuyên mô hình hóa, ngay cả khi họ am hiểu nghiệp vụ. Nhiều công trình nghiên cứu đã đề xuất các cải tiến cho OCL, các nỗ lực này chủ yếu tập trung vào việc nâng cấp môi trường thao tác—bao gồm trình soạn thảo chuyên dụng, làm nổi bật cú pháp, hướng dẫn tái cấu trúc, hay các tiện ích hỗ trợ khác. Tuy nhiên, những cải tiến đó vẫn duy trì quan điểm xem OCL như một ngôn ngữ hình thức độc lập, được gắn kèm vào mô hình UML/Ecore [21, 79, 129] và tách rời khỏi ngôn ngữ nghiệp vụ mà chuyên gia miền sử dụng. Do đó, khoảng cách ngữ nghĩa giữa mô hình kỹ thuật

và cách biểu đạt tri thức miền vẫn còn tồn tại, thay vì nhúng OCL vào một ngôn ngữ miền nhằm tăng khả năng diễn đạt trực tiếp các khái niệm nghiệp vụ.

Một số thư viện hiện đại như `OCL.js` hoặc các hướng tiếp cận như `JjOM` đưa OCL vào hệ sinh thái `JavaScript` và ứng dụng web, song cách dùng vẫn xoay quanh việc viết và quản lý các biểu thức OCL rời rạc, buộc chuyên gia miền phải học cú pháp OCL thay vì tương tác với một cơ chế ràng buộc được thiết kế theo ngôn ngữ miền [20]. Nhiều nỗ lực cải thiện công cụ và trải nghiệm thao tác gắn kết trực tiếp với mã nguồn [20, 25, 28, 35, 115]. Các nghiên cứu trong MDE cho phép gắn OCL vào siêu mô hình (như Ecore hoặc UML biểu đồ lớp) nhằm tạo ra các ràng buộc tích hợp, tuy nhiên cách tích hợp này vẫn để ràng buộc tồn tại dưới dạng OCL độc lập, không đồng nhất với cú pháp của lớp miền và không thân thiện với chuyên gia miền. Tương tự, một số khung làm việc dựa trên chú thích cho phép mô tả các ràng buộc mức cơ bản như phạm vi giá trị, tính tùy chọn hoặc quan hệ, nhưng hoàn toàn không hỗ trợ ràng buộc phức tạp tương đương OCL. Vì vậy, dù có khả năng gắn siêu dữ liệu lên lớp miền, các khung làm việc này chưa thể hiện thực hóa việc gắn các ràng buộc nghiệp vụ như `forAll`, `exists` một cách tự nhiên, dễ đọc, và tích hợp liền mạch vào DM như một phần của DSL đặc tả miền. Các nghiên cứu liên quan đến OCL và DSL cho cấu hình sản phẩm, cũng như các nghiên cứu về sinh kiểm thử từ OCL hoặc suy luận dựa trên ràng buộc cũng coi OCL như một tập ràng buộc tổng quát mà không tổ chức các ràng buộc này thành các nhóm miền có ngữ nghĩa ổn định và tên gọi dễ nhớ. Từ đó, có thể thấy rằng các công trình hiện tại mới chỉ chạm tới vấn đề phân loại ràng buộc ở mức lô-gic, chưa phát triển thành một danh mục ràng buộc theo miền nghiệp vụ với cú pháp và ngữ nghĩa được chuẩn hóa để chuyên gia miền có thể nhận diện và sử dụng trực tiếp.

Trong các công trình sinh mã từ OCL, việc truy vết thường được hiểu như một hệ quả tự nhiên của luật chuyển đổi, chứ không phải một cơ chế được thiết kế có chủ đích. Các khung như OCL2Java [130], USE Tool [56] hay những bộ sinh mã nguồn xác thực trong EMF cung cấp khả năng dịch ràng buộc OCL thành mã kiểm tra. Các môi trường chạy như Eclipse OCL có thể lưu trữ ràng buộc như một phần của mô hình, nhưng cũng không đưa ra hệ thống truy vết hai chiều mang tính hệ thống, nơi mỗi ràng buộc

có thể được theo dõi xuyên suốt qua mô hình, chuyển đổi và mã thực thi. Trong khi đó, các nỗ lực sử dụng aDSL như DCSL [70] đã cho phép biểu diễn và chạy mô hình miền hiệu quả, nhưng lại chỉ hỗ trợ một tập ràng buộc cơ bản, không đủ khả năng mô tả và tích hợp các ràng buộc OCL phức tạp, đa lớp và phụ thuộc ngữ cảnh. Sự tách rời giữa năng lực biểu đạt mạnh của OCL và khả năng thực thi của aDSL đã trở thành nút thắt ngăn cản việc phát triển một DM thực thi thống nhất.

Trong phân tích và thiết kế hướng đối tượng, DM thường được biểu diễn bằng các biểu đồ lớp UML kết hợp với OCL [68, 91] nhằm đặc tả cấu trúc và các ràng buộc nghiệp vụ. Trong DDD, Evans [39] mở rộng vai trò của DM, coi đây vừa là trung tâm của tri thức và luật lệ miền, vừa là phương tiện giao tiếp giữa chuyên gia miền và lập trình viên.

Các nghiên cứu trong MDE [8, 17] tiếp tục hướng tiếp cận này bằng cách phát triển các DSL để mô tả DM một cách chính xác hơn, dựa trên siêu mô hình và cú pháp chuyên biệt. Tuy nhiên, các mô hình theo hướng mô tả này chủ yếu tập trung vào tính biểu đạt cú pháp, trong khi chưa trực tiếp đáp ứng yêu cầu về khả năng thực thi và ngữ nghĩa hành vi trong bối cảnh DDD. Phương pháp tích hợp các mối quan tâm vào mô hình miền hợp nhất (gọi là, UDML) được đề xuất nhằm khắc phục hạn chế này bằng cách đặc tả một DM hợp nhất có khả năng thực thi.

Trong các hệ thống có yêu cầu phân quyền phức tạp, kiểm soát truy cập trở thành một mối quan tâm trung tâm thay vì thứ yếu [112]. Nhiều nghiên cứu cho thấy việc tách rời chính sách bảo mật khỏi lô-gic nghiệp vụ lõi có thể dẫn đến sự không nhất quán giữa hành vi miền và các quy tắc phân quyền [11, 91, 93]. Dưới góc nhìn DDD, các chính sách RBAC vì thế cần được biểu diễn tường minh như một phần của hành vi miền [46, 73]. Tuy nhiên, phần lớn các cách tiếp cận hiện nay tích hợp RBAC theo cách thủ công hoặc phi hình thức, thiếu một nền tảng ngữ nghĩa hợp nhất cho toàn bộ DM.

Các nghiên cứu về DSL nội sinh [43, 132] cho thấy đây là một hướng tiếp cận hiệu quả để xây dựng các DM có khả năng thực thi, nhờ tận dụng cú pháp và ngữ nghĩa của ngôn ngữ chủ. Nhiều công trình đã sử dụng chú thích để mô hình hóa ràng buộc miền và xây dựng các DSL nhúng trong OOPL [31, 95, 119], hoặc duy trì ánh xạ giữa DM ở mức mã nguồn và

UML [13]. Tiêu biểu trong số đó là DCSL [70], cho phép đặc tả cấu trúc miền có khả năng thực thi.

Tuy nhiên, các tiếp cận này chủ yếu nhấn mạnh tính thực thi ở mức hiện thực hóa, trong khi chưa giải quyết thỏa đáng bài toán hợp nhất nhiều mối quan tâm và kiểm chứng hình thức các ràng buộc động, đặc biệt là các ràng buộc phân quyền. OCL hoặc mã dựa trên chú thích chủ yếu hỗ trợ phân tích tĩnh, chưa cung cấp các bảo đảm hình thức về tính an toàn và không xung đột. Phương pháp của luận án hướng đến xây dựng một DM hợp nhất vừa có thể thực thi vừa được kiểm chứng hình thức, trong đó các chính sách RBAC được đảm bảo an toàn và nhất quán trong toàn bộ vòng đời phần mềm.

Tách biệt và hợp nhất mối quan tâm là nguyên tắc cốt lõi trong quản lý độ phức tạp hệ thống [120, 128]. Nhiều nghiên cứu đã sử dụng DSL để mô tả các mối quan tâm riêng biệt và chỉ ra khó khăn trong việc kết hợp chúng thành một mô hình thống nhất [89, 99, 126]. Các cơ chế hợp nhất phổ biến bao gồm tham chiếu, mở rộng, nhúng và tái sử dụng, hoặc điều phối ở mức siêu mô hình [105].

Khác với các cách tiếp cận tập trung vào hợp nhất cú pháp hoặc cấu trúc, luận án đề xuất cơ chế hợp nhất mối quan tâm trong một DM hợp nhất có khả năng thực thi, dựa trên chú thích và hợp nhất ở mức cây cú pháp trừu tượng (AST). Cách tiếp cận này cho phép hợp nhất đồng thời cú pháp, ngữ nghĩa và khả năng thực thi của các mối quan tâm, đồng thời duy trì sự gắn kết với DM theo tinh thần DDD.

Các phương pháp hình thức, đặc biệt là Event-B, đã được sử dụng rộng rãi để kiểm chứng các hệ thống an toàn và đảm bảo về bảo mật [65, 76]. Nhiều nghiên cứu đã mã hóa RBAC dưới dạng bất biến và điều kiện bảo vệ để kiểm chứng các thuộc tính bảo mật [5, 78]. Một số công trình cũng khảo sát việc kết nối DSL với Event-B nhằm cung cấp kiểm tra tính đúng đắn dữ liệu và lô-gic nghiệp vụ ở phía máy chủ [98, 123].

Tuy nhiên, các cách tiếp cận này thường xem phân quyền như một bài toán chính sách độc lập, tách rời khỏi hành vi miền. Ngược lại, phương pháp được đề xuất trong luận án sử dụng Event-B như một nền tảng ngữ nghĩa hình thức cho DM hợp nhất được đặc tả bằng UDML được trình bày chi tiết trong Chương 4 của luận án, trong đó các ràng buộc RBAC trực tiếp

giới hạn các chuyển tiếp hành vi. Cách tiếp cận này cho phép kiểm chứng đồng thời tính đúng đắn miền và các thuộc tính bảo mật trong một khuôn khổ hình thức thống nhất.

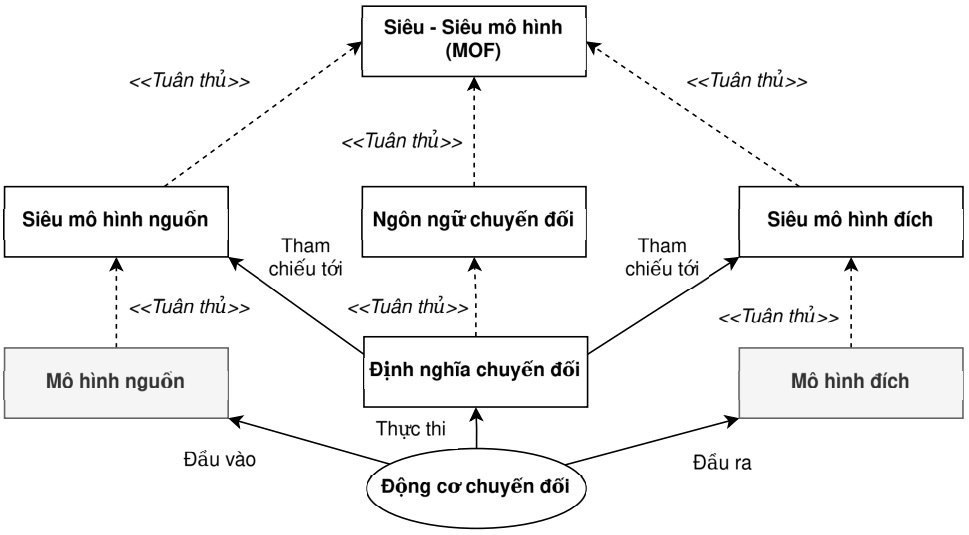
2.4 Các thao tác chuyển đổi mô hình miền

Phần này trình bày các thao tác chuyển đổi mô hình miền trong khuôn khổ kỹ nghệ phần mềm hướng mô hình, với mục tiêu làm rõ vai trò của chuyển đổi mô hình trong việc liên kết các mức trừu tượng khác nhau của mô hình miền, đồng thời tạo cơ sở cho quá trình sinh các chế tác phần mềm từ mô hình miền trong bối cảnh thiết kế hướng miền. Trong cách tiếp cận này, mô hình miền không chỉ được xem như một biểu diễn khái niệm của tri thức nghiệp vụ, mà còn là đầu vào trung tâm cho các bộ chuyển đổi phục vụ sinh các chế tác phần mềm.

2.4.1 Kỹ thuật chuyển đổi mô hình

Trong kỹ nghệ phần mềm hướng mô hình [17], chuyển đổi mô hình là cơ chế trung tâm để tự động hóa quá trình phát triển phần mềm. Một bộ chuyển đổi điển hình tham gia vào các giai đoạn phát triển từ mô hình yêu cầu, mô hình phân tích, mô hình thiết kế đến mã nguồn, trong đó mỗi giai đoạn đều cần duy trì tính nhất quán giữa mô hình nguồn và mô hình đích. Xét về bản chất, một chuyển đổi mô hình không chỉ được thực hiện giữa hai mô hình cụ thể, mà được xác lập ở cấp độ siêu mô hình, với điều kiện các mô hình nguồn và mô hình đích phải tuân thủ các siêu mô hình tương ứng. Đây cũng là quan điểm chung của kỹ nghệ hướng mô hình (MDE) và kiến trúc hướng mô hình (MDA), trong đó các phép chuyển đổi được sử dụng để liên kết các mô hình ở các mức trừu tượng khác nhau, từ mô hình độc lập tính toán (CIM), mô hình độc lập nền tảng (PIM) đến mô hình chuyên biệt nền tảng (PSM).

Hình 2.2 mô tả khung công việc chung của một bộ chuyển đổi mô hình điển hình [17]. Theo khung này, bộ chuyển đổi được xác định bởi cặp siêu mô hình nguồn và siêu mô hình đích, cùng với định nghĩa chuyển đổi và môi trường thực thi tương ứng. Việc phát biểu chuyển đổi ở mức siêu mô hình cho phép tách biệt giữa đặc tả chuyển đổi và các mô hình thể hiện cụ thể,



Hình 2.2: Kiến trúc chung của bộ chuyển đổi mô hình.

đồng thời tạo điều kiện cho việc tái sử dụng, kiểm chứng và hiện thực hóa các phép chuyển đổi trên các công cụ khác nhau.

Xét theo cách tiếp cận cài đặt, các bộ chuyển đổi mô hình thường được phân thành bốn nhóm chính, bao gồm: cách tiếp cận khai báo/quan hệ, cách tiếp cận mệnh lệnh/thực thi, cách tiếp cận dựa trên chuyển đổi đồ thị, và cách tiếp cận lai. Cách tiếp cận khai báo tập trung mô tả quan hệ giữa các phần tử của mô hình nguồn và mô hình đích; cách tiếp cận mệnh lệnh tập trung mô tả trình tự thực hiện các bước chuyển đổi; cách tiếp cận dựa trên chuyển đổi đồ thị biểu diễn mô hình bằng đồ thị định kiểu và đặc tả chuyển đổi bằng các luật viết lại đồ thị; trong khi cách tiếp cận lai cố gắng kết hợp ưu điểm của các hướng trên. Trong thực tiễn, các ngôn ngữ và công cụ như ATL [50], QVT [88] và Acceleo [17] thường được sử dụng để hiện thực các bộ chuyển đổi mô hình và sinh mã nguồn.

Trong bối cảnh mô hình miền giàu thông tin, đặc biệt khi mô hình miền đồng thời chứa cấu trúc, hành vi và các ràng buộc nghiệp vụ, thách thức cốt lõi của chuyển đổi mô hình không chỉ nằm ở việc ánh xạ cấu trúc, mà còn ở việc bảo toàn ngữ nghĩa của mô hình trong toàn bộ quá trình chuyển đổi. Theo quan điểm chất lượng của các bộ chuyển đổi mô hình, các thuộc tính cần được quan tâm bao gồm: tính đúng đắn về ngữ pháp của mô hình đích, tính đúng đắn về ngữ nghĩa, tính đầy đủ, và các thuộc tính hành vi chức năng như kết thúc hay hội tụ. Đối với mô hình miền trong DDD, yêu

cầu này càng trở nên quan trọng vì mô hình đích không chỉ cần hợp lệ về cú pháp, mà còn phải phản ánh nhất quán tri thức miền và logic nghiệp vụ của mô hình nguồn. Do đó, các phép chuyển đổi cần dựa trên một biểu diễn trung gian có ngữ nghĩa rõ ràng và nhất quán, chẳng hạn như trong khung JDA [71].

Trên phương diện thao tác, chuyển đổi mô hình thường được thực hiện dưới hai dạng chính. Thứ nhất là chuyển đổi mô hình–sang–mô hình (M2M), trong đó mô hình nguồn được ánh xạ sang một mô hình trung gian hoặc mô hình đích ở cùng hay khác mức trừu tượng. Thứ hai là chuyển đổi mô hình–sang–văn bản (M2T), trong đó mô hình được chuyển thành các biểu diễn văn bản như mã nguồn, tệp cấu hình hoặc các dạng đặc tả có thể thực thi. Trong nhiều trường hợp, đặc biệt đối với sinh phần mềm dựa trên mô hình, M2M và M2T không tồn tại tách biệt mà được tổ chức thành một bộ chuyển đổi: trước hết mô hình nguồn được chuyển thành một biểu diễn trung gian có ngữ nghĩa rõ ràng, sau đó mới tiếp tục sinh ra các tạo tác văn bản hoặc mã nguồn tương ứng. Cách tổ chức này giúp giảm độ phụ thuộc trực tiếp giữa mô hình yêu cầu và mã nguồn, đồng thời tạo điều kiện thuận lợi cho việc kiểm soát, kiểm chứng và tiến hóa mô hình.

Một hướng tiếp cận phù hợp cho mục tiêu đó là kỹ thuật sinh mã nguồn dựa trên siêu mô hình và khuôn mẫu văn bản [127]. Thay vì chuyển trực tiếp từ mô hình sang mã nguồn, mô hình đầu vào trước hết được ánh xạ sang một mô hình trung gian tuân thủ một siêu mô hình xác định, từ đó các thông tin cần thiết được trích xuất và kết hợp với các khuôn mẫu để sinh ra mã nguồn hoặc các biểu diễn thực thi tương ứng. Trong bối cảnh luận án này, cách tiếp cận như vậy đặc biệt phù hợp với yêu cầu chuyển đổi mô hình miền giàu hành vi và ràng buộc, vì nó cho phép xác lập một biểu diễn trung gian có ngữ nghĩa rõ ràng trước khi sinh các chế tác phần mềm.

2.4.2 Sinh chế tác phần mềm từ mô hình miền

Trên cơ sở các kỹ thuật chuyển đổi mô hình nêu trên, một hướng ứng dụng quan trọng là sinh các chế tác phần mềm từ mô hình miền. Trong kỹ nghệ phần mềm hướng mô hình, các chế tác sinh ra có thể bao gồm mô hình trung gian, mô hình thực thi, mã nguồn, giao diện người dùng, các mô-đun cấu hình, hoặc bản mẫu phần mềm. Đối với DDD, mục tiêu của các thao

tác này không chỉ là tự động hóa phát triển phần mềm, mà còn là duy trì mối liên kết chặt chẽ giữa tri thức miền, mô hình miền và hiện thực phần mềm.

Một lớp thao tác quan trọng là sinh bản mẫu phần mềm từ mô hình miền. Nhiều nghiên cứu đã đề xuất các phương pháp sinh tự động các tạo tác phần mềm từ đặc tả yêu cầu, chủ yếu tập trung vào các thành phần giao diện và các chức năng cơ bản. Chẳng hạn, IFML được sử dụng để mô hình hóa luồng tương tác và sinh tự động giao diện [101]; các nghiên cứu khác khai thác mô hình ca sử dụng hoặc các biểu đồ UML như biểu đồ lớp, biểu đồ hoạt động và biểu đồ tuần tự để sinh giao diện phục vụ phát triển nhanh ứng dụng [10, 14, 67, 84, 131]. Tuy nhiên, phần lớn các tiếp cận này mới dừng ở việc sinh giao diện hoặc một phần cấu trúc chức năng, trong khi chưa thiết lập được một chuỗi chuyển đổi đủ chặt chẽ từ đặc tả yêu cầu đến mô hình miền có khả năng thực thi và bản mẫu phần mềm hoàn chỉnh.

Sinh giao diện người dùng là một trường hợp tiêu biểu của sinh chế tác phần mềm từ mô hình. Đây là hướng được quan tâm nhiều vì giao diện là tạo tác trực quan, dễ đánh giá và có giá trị cao trong việc xác minh yêu cầu với người dùng cuối. Tuy vậy, nếu chỉ dừng ở sinh giao diện thì khoảng cách giữa mô hình yêu cầu và logic nghiệp vụ bên trong hệ thống vẫn chưa được giải quyết triệt để. Đối với các hệ thống phát triển theo DDD, giao diện người dùng chỉ là một phần của bản mẫu phần mềm; điều quan trọng hơn là phải bảo đảm rằng giao diện, hành vi và cấu trúc dữ liệu đều được dẫn xuất nhất quán từ cùng một mô hình miền hoặc từ cùng một đặc tả yêu cầu hướng miền.

Việc đặc tả yêu cầu chức năng phần mềm nhằm xác định rõ các chức năng cần triển khai và các yêu cầu mà hệ thống phải đáp ứng, bao gồm mô tả chức năng, giao diện người dùng, các yêu cầu kỹ thuật và các ràng buộc hoạt động [40]. Trong thực tiễn, nhiều phương pháp và công cụ đã được đề xuất để đặc tả yêu cầu chức năng, sử dụng các hình thức như ngôn ngữ tự nhiên, UML hoặc các DSL [48]. Trong đó, UML/OCL là các công cụ được chuẩn hóa và sử dụng rộng rãi, trong khi DSL mang lại lợi thế về tính chuyên biệt, khả năng đơn giản hóa và hỗ trợ tự động hóa [17, 43, 60].

Một lớp thao tác khác có ý nghĩa nền tảng là sinh mô hình miền từ mô hình yêu cầu. Nhiều nghiên cứu đã tập trung vào các phương pháp chuyển

đổi mô hình hành vi sang mã triển khai [116], tiêu biểu là các kỹ thuật sinh mã từ biểu đồ hoạt động và biểu đồ tuần tự UML [124]. Một hướng tiếp cận khác đề xuất tự động hóa chuyển đổi từ mô hình độc lập tính toán dựa trên nghiệp vụ sang mô hình độc lập nền tảng cho các ứng dụng web [133]. Trong nghiên cứu [72], tác giả đề xuất phương pháp phát triển phần mềm sinh theo DDD dựa trên kiến trúc phần mềm hướng mô-đun và kỹ thuật sinh cấu hình mô-đun.

Từ khảo sát trên có thể thấy nhu cầu cấp thiết về các bộ chuyển đổi giúp sinh nhanh các tạo tác phần mềm từ mô hình miền, đồng thời cung cấp nhiều góc nhìn phù hợp cho các bên liên quan. Các bộ chuyển đổi này đóng vai trò then chốt trong việc thu hẹp khoảng trống giữa các mô hình miền ở mức trừu tượng cao và các hiện thực phần mềm có khả năng thực thi, qua đó duy trì tính nhất quán ngữ nghĩa giữa các đặc tả, mô hình và các tạo tác phần mềm sinh ra.

Phương pháp thiết kế hướng miền mang lại nhiều lợi ích trong phát triển các hệ thống phần mềm phức tạp [59]. Tuy nhiên, khi yêu cầu nghiệp vụ thay đổi, mô hình miền cần được cập nhật tương ứng, đòi hỏi tính linh hoạt cao trong thiết kế và hiện thực, điều này có thể gây khó khăn cho cả nhà thiết kế và nhà phát triển. Trong nghiên cứu về DSL [43], mô hình được xem như một đặc tả trong ngôn ngữ chuyên biệt miền, và quá trình sinh mã nguồn thường trải qua nhiều bước chuyển đổi mô hình–sang–mô hình trước khi sinh mã và thực thi. Các nghiên cứu gần đây đã đề xuất DCSL [70] và MCCL [72] nhằm hỗ trợ phát triển phần mềm theo DDD thông qua các ngôn ngữ chuyên biệt miền dựa trên chú thích (aDSL) được nhúng trong OOPL.

Bên cạnh đó, nhiều công trình đã đề xuất các kỹ thuật sinh tự động giao diện người dùng và bản mẫu phần mềm từ các biểu đồ UML và các đặc tả ca sử dụng [67, 84, 101], tập trung vào phát triển công cụ chuyển đổi và đảm bảo tính nhất quán của mã nguồn sinh ra. Tuy nhiên, vẫn chưa có một phương pháp hoàn chỉnh nào thu hẹp hiệu quả khoảng cách ngữ nghĩa giữa đặc tả yêu cầu ở dạng ngôn ngữ tự nhiên hoặc bán hình thức như UML/OCL [42, 91] và mã nguồn cài đặt bản mẫu phần mềm.

Từ khảo sát trên có thể thấy rằng bài toán sinh chế tác phần mềm từ mô hình miền thực chất bao gồm một chuỗi các thao tác chuyển đổi liên tiếp: từ

đặc tả yêu cầu sang mô hình miền, từ mô hình miền sang mô hình thực thi, và từ đó sang mã nguồn hoặc bản mẫu phần mềm. Trong chuỗi này, vai trò của biểu diễn trung gian là đặc biệt quan trọng, vì nó giúp thu hẹp khoảng cách giữa mô hình khái niệm và hiện thực phần mềm, đồng thời tạo điều kiện bảo toàn ngữ nghĩa của miền nghiệp vụ trong suốt quá trình chuyển đổi. Trên cơ sở đó, luận án tập trung nghiên cứu các thao tác chuyển đổi mô hình miền theo hướng tích hợp đồng thời khía cạnh cấu trúc và hành vi, từ đó tạo nền tảng cho việc sinh tự động bản mẫu phần mềm được trình bày trong Chương 5.

2.5 Hướng tiếp cận sử dụng AI/LLM

Sự phát triển của các mô hình ngôn ngữ lớn (*Large Language Models - LLMs*) như GPT-4 đã thúc đẩy việc ứng dụng trí tuệ nhân tạo (*Artificial Intelligence - AI*) trong phân tích và thiết kế phần mềm. Các nghiên cứu gần đây cho thấy LLMs có khả năng xử lý đặc tả ngôn ngữ tự nhiên và hỗ trợ sinh các biểu diễn mô hình như UML hoặc đặc tả có cấu trúc từ mô tả nghiệp vụ [61, 86, 92]. Một số công trình tiêu biểu như Ferrari et al. [41] và Giannouris et al. [45] cho thấy LLMs có thể trích xuất các khái niệm miền và quan hệ để sinh biểu đồ UML từ tài liệu yêu cầu, trong khi Eisenreich et al. [38] bước đầu áp dụng LLMs trong bối cảnh DDD.

Trong biểu diễn mô hình miền, các tiếp cận dựa trên LLMs chủ yếu hỗ trợ suy diễn cấu trúc mô hình từ các đặc tả không hình thức, thông qua việc nhận diện các thực thể, thuộc tính và quan hệ. Cách tiếp cận này giúp tự động hóa bước hình thành mô hình ban đầu và hỗ trợ khai phá tri thức miền. Tuy nhiên, các biểu diễn được sinh ra mang tính xác suất và không dựa trên một khuôn khổ ngữ nghĩa hình thức, do đó chưa đảm bảo tính nhất quán, khả năng tích hợp nhiều mối quan tâm và vai trò lấy một mô hình miền làm trung tâm theo tinh thần của DDD.

Trong chuyển đổi mô hình, các tiếp cận AI/LLM chủ yếu thực hiện sinh trực tiếp các tạo tác đích, chẳng hạn mã nguồn hoặc mô hình ở mức thấp, từ mô tả đầu vào [86, 92]. Một số nghiên cứu cũng đề cập đến khả năng sinh ánh xạ giữa các biểu diễn khác nhau [61]. Tuy nhiên, các phép chuyển đổi này không được đặc tả tường minh mà phụ thuộc vào quá trình suy diễn

của mô hình, do đó thiếu cơ chế kiểm soát, truy vết và không đảm bảo bảo toàn ngữ nghĩa giữa mô hình nguồn và mô hình đích.

Nhìn chung, các tiếp cận AI/LLM hiện nay chủ yếu hỗ trợ ở giai đoạn sinh và khám phá mô hình ban đầu, nhưng còn hạn chế trong việc đặc tả ngữ nghĩa, tích hợp nhiều mối quan tâm và đảm bảo tính đúng đắn trong chuyển đổi mô hình. Điều này cho thấy vẫn còn khoảng trống nghiên cứu đối với các phương pháp biểu diễn và chuyển đổi mô hình có ngữ nghĩa rõ ràng, hỗ trợ tích hợp và kiểm chứng một cách có hệ thống, trong bối cảnh thiết kế hướng miền.

2.6 Tổng kết chương

Chương này đã trình bày cơ sở lý thuyết và tổng quan tình hình nghiên cứu làm cơ sở lý luận cho luận án, tập trung vào DDD và vai trò trung tâm của DM trong toàn bộ quá trình phát triển phần mềm. Các khái niệm cốt lõi của DDD, bao gồm DM, UL và yêu cầu gắn kết chặt chẽ giữa mô hình và hiện thực triển khai, đã được phân tích nhằm làm rõ bối cảnh phương pháp luận của nghiên cứu.

Trên cơ sở đó, chương đã khảo sát và phân tích các tiếp cận hiện có trong việc biểu diễn DM, tích hợp cấu trúc, hành vi, ràng buộc và bảo mật thông qua UML/OCL và các DSL, cũng như các kỹ thuật hợp nhất mối quan tâm và chuyển đổi mô hình trong kỹ nghệ phần mềm hướng mô hình. Phân tích này cho thấy rằng mặc dù đã tồn tại nhiều tiếp cận nhằm hỗ trợ mô hình hóa, thực thi và kiểm chứng DM, các giải pháp hiện nay vẫn còn phân tán, thiếu một khuôn khổ thống nhất đặt DM làm trung tâm ngữ nghĩa, cho phép tích hợp đồng thời các mối quan tâm cấu trúc, hành vi và bảo mật, đồng thời hỗ trợ khả năng thực thi và chuyển đổi mô hình theo đúng tinh thần của DDD.

Khoảng trống nghiên cứu này cho thấy nhu cầu cần có các kỹ thuật biểu diễn và hợp nhất DM giàu ngữ nghĩa hơn, cùng với các cơ chế kiểm chứng và chuyển đổi mô hình bảo toàn ngữ nghĩa, nhằm thu hẹp khoảng cách giữa đặc tả miền và hiện thực phần mềm. Đây chính là động lực và nền tảng cho các kỹ thuật và phương pháp được đề xuất trong các chương tiếp theo của luận án.

Chương 3

KỸ THUẬT BIỂU DIỄN MÔ HÌNH MIỀN

Trong chương này, luận án đề xuất các kỹ thuật mở rộng và tích hợp mô hình miền (DM) nhằm đưa các khía cạnh hành vi và ràng buộc nghiệp vụ vào một mô hình miền hợp nhất có khả năng thực thi. Cụ thể, chương giới thiệu ngôn ngữ đồ thị hoạt động *Activity Graph Language* (AGL) như một DSL dựa trên chú thích để biểu diễn và gắn kết trực tiếp hành vi nghiệp vụ với các khái niệm miền, đồng thời trình bày mẫu chú thích ràng buộc *Constraint Annotation Pattern* (CAP) nhằm biểu diễn và tích hợp các ràng buộc OCL vào DM thực thi. Thông qua đó, các kỹ thuật đề xuất góp phần thu hẹp khoảng cách giữa miền nghiệp vụ và không gian kỹ thuật, đồng thời nâng cao khả năng biểu đạt, thực thi và kiểm chứng của DM trong DDD.

3.1 Giới thiệu

Trong các tiếp cận DDD hiện nay, các DSL dựa trên chú thích (aDSL) được nhúng trong OOPL đã được đề xuất nhằm hỗ trợ xây dựng DM thực thi [95, 119]. Các tiếp cận này cho phép mã hóa trực tiếp các khái niệm miền trong mã nguồn của hệ thống, hoặc kết hợp DM với các đặc tả ở mức trừu tượng cao hơn thông qua UML và các DSL, làm cơ sở cho các phép biến đổi mô hình và sinh tạo tác phần mềm.

Tuy nhiên, các aDSL hiện có [8, 31, 43, 126] chủ yếu tập trung vào biểu diễn cấu trúc của DM. Trong khi đó, các khía cạnh hành vi, thường được mô

tả bằng biểu đồ hoạt động hoặc biểu đồ máy trạng thái UML, cùng với các ràng buộc OCL nghiệp vụ phức tạp, như các ràng buộc sử dụng `forAll`, `exists` hoặc các phép toán trên tập hợp, vẫn chưa được tích hợp một cách tường minh, có cấu trúc và hợp nhất trong DM thực thi. Hạn chế này làm suy giảm khả năng biểu đạt đầy đủ các quy tắc nghiệp vụ, đồng thời gây khó khăn cho việc hợp nhất các khía cạnh miền trong một mô hình hợp nhất nhất.

Luận án hướng đến việc mở rộng DM theo hướng tích hợp các khía cạnh hành vi, từ đó hình thành một DM hợp nhất có khả năng biểu đạt và hỗ trợ tốt hơn cho quá trình xây dựng phần mềm.

Trong bối cảnh đó, luận án đề xuất một ngôn ngữ hỗ trợ chuyên biệt cho việc tích hợp hành vi miền nhằm thu hẹp khoảng cách giữa DM và hiện thực của nó, đồng thời tạo điều kiện thuận lợi cho việc xây dựng phần mềm thông qua các phép biến đổi mô hình từ DM có tích hợp các đặc tả hành vi được biểu đạt bằng UML và DSL. Ngôn ngữ này được gọi là AGL (*Activity Graph Language*), được thiết kế như một aDSL tập trung vào việc biểu diễn các khía cạnh hành vi miền dưới dạng đồ thị hoạt động, cũng như tích hợp trực tiếp các đặc tả hành vi này vào DM hợp nhất. AGL được giới hạn trong một miền con của biểu đồ hoạt động của UML, dựa trên các mẫu mô hình hóa hoạt động cốt lõi [91, p. 373]. Cách tiếp cận hướng mô hình cho DSL [66] được áp dụng, với UML/OCL [90, 91] dùng để đặc tả mô hình cú pháp trừu tượng và cú pháp cụ thể của AGL.

Để tích hợp AGL vào DM hợp nhất, aDSL đã được phát triển trước đó là DCSL [70] được sử dụng nhằm biểu diễn mô hình lớp hợp nhất. Mô hình lớp hợp nhất này được xem như một DM mở rộng kiến trúc phần mềm dạng mô-đun (MOSA) [72], trong đó các lớp miền, bao gồm các lớp hoạt động gắn với đồ thị AGL, được ánh xạ trực tiếp sang hiện thực kỹ thuật thực thi bằng JDA [72].

Mặc dù DCSL cung cấp một biểu diễn ngắn gọn và gắn chặt với hiện thực triển khai cho các khái niệm cấu trúc của miền, các aDSL dạng này vẫn gặp hạn chế trong việc biểu diễn và tích hợp các ràng buộc OCL nghiệp vụ phức tạp, vốn là thành phần thiết yếu để đặc tả chính xác các quy tắc nghiệp vụ 2.1 trong DDD [39, 90].

Các DSL gần đây như B-OCL [53] hỗ trợ mô hình hóa cấu trúc và một

phần các ràng buộc nghiệp vụ; tuy nhiên, các tiếp cận này vẫn gặp nhiều thách thức trong việc chuyển đổi UML/OCL sang mã thực thi, tích hợp cơ chế kiểm tra OCL tại giai đoạn chương trình thực thi, cũng như bảo toàn tính đúng đắn của mô hình xuyên suốt vòng đời phát triển. Song song đó, các DSL nội sinh [7, 18, 59, 87, 135] nhúng trực tiếp lô-gic miền vào ngôn ngữ chủ như Java [114], qua đó hỗ trợ sinh kiểm thử tự động và xây dựng các hiện thực có thể kiểm chứng. Tuy nhiên, các tiếp cận này thường chỉ xử lý được một tập con nhỏ của OCL, chưa giải quyết đầy đủ việc biểu diễn các ràng buộc OCL nghiệp vụ phức tạp, cũng như chưa hỗ trợ sinh mã tự động ở mức toàn diện. Hạn chế cốt lõi nằm ở tính thỏa mãn của cách tiếp cận, do cú pháp của ngôn ngữ lập trình chủ thường là OOPL có giới hạn trong việc biểu diễn tự nhiên các khái niệm miền phức tạp.

Trước những hạn chế đó, luận án đề xuất một mở rộng aDSL nhằm biểu diễn và tích hợp các ràng buộc OCL nghiệp vụ phức tạp trực tiếp trong DM thực thi. Theo đó, mỗi ràng buộc OCL được ánh xạ sang một mẫu chú thích, gọi là CAP (*Constraint Annotation Pattern*), cho phép biểu diễn ngữ nghĩa ràng buộc bằng các cấu trúc chú thích gắn với các phần tử miền tương ứng.

CAP cung cấp một cơ chế tổ chức ràng buộc có tính hệ thống, giúp tích hợp chặt chẽ các ràng buộc vào DM thống nhất thực thi, qua đó tạo nền tảng cho việc hợp thành các khía cạnh cấu trúc, hành vi và ràng buộc trong cùng một DM. Cách tiếp cận này hỗ trợ cho các bước kiểm chứng và sinh tạo tác phần mềm.

Các mục còn lại của chương này được cấu trúc như sau. Mục 3.2 trình bày kỹ thuật biểu diễn và tích hợp các ràng buộc OCL dựa vào mẫu chú thích (CAP) trình bày trong . Mục 3.3 trình bày kỹ thuật biểu diễn, tích hợp hành vi vào mô hình miền (AGL). Cuối cùng, tổng kết các kết quả đạt được trình bày trong Mục 3.4 của luận án.

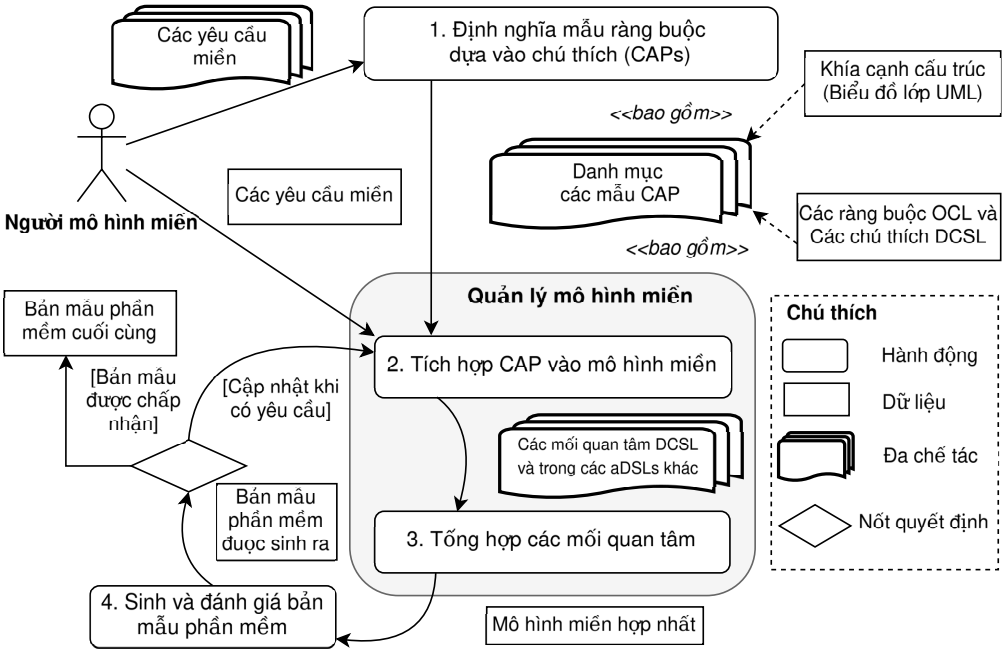
3.2 Kỹ thuật tích hợp ràng buộc OCL vào mô hình miền

Phần này trình bày một kỹ thuật tích hợp các ràng buộc OCL nghiệp vụ phức tạp vào DM, gọi là CAP, nhằm thu hẹp khoảng cách giữa không gian

bài toán và không gian kỹ thuật, đồng thời vẫn bảo toàn khả năng biểu đạt của DDD. Kỹ thuật này hỗ trợ tự động hóa việc kiểm soát ràng buộc và sinh mã trong quá trình phát triển phần mềm.

3.2.1 Tổng quan về phương pháp đề xuất

Đề xuất kỹ thuật tích hợp ràng buộc phức tạp vào DM theo DDD được mô tả trong Hình 3.1, gồm hai giai đoạn chính:



Hình 3.1: Kỹ thuật tích hợp ràng buộc phức tạp vào DM theo DDD.

Giai đoạn 1 tập trung vào việc xác định một tập hợp các mẫu dựa trên chú thích ràng buộc (*Constraint Annotation Patterns* – viết tắt là CAPs) thông qua việc trích xuất và khái quát hóa chúng từ các yêu cầu miền hiện có (Bước 1 trong Hình 3.1). *Giai đoạn 2*, bao gồm ba bước còn lại trong Hình 3.1, nhằm mục tiêu tạo ra một bản mẫu phần mềm cuối cùng thỏa mãn các yêu cầu đầu vào của hệ thống đích. Các bước chính của phương pháp được tóm lược như sau.

Bước 1 – Xác định CAP: Dựa trên tập hợp các yêu cầu miền thu thập từ nhiều lĩnh vực khác nhau, bước này nhằm nhận diện các nhóm ràng buộc OCL, khái quát hóa từng nhóm và xây dựng một mẫu chung – gọi là CAP – để biểu diễn các ràng buộc OCL thuộc nhóm đó. Mỗi CAP bao

gồm: (i) một biểu đồ lớp UML để đặc tả các khái niệm miền và quan hệ giữa chúng, (ii) một đặc tả OCL mô tả các quy tắc nghiệp vụ và bất biến, và (iii) các chú thích biểu diễn các ràng buộc OCL.

Bước 2 – Tích hợp CAPs vào DM: Đầu vào của bước này gồm tập các ràng buộc miền OCL và danh mục CAPs đã có. Mục tiêu là biểu diễn các ràng buộc OCL dưới dạng các CAP, qua đó ràng buộc DM. Mỗi ràng buộc dựa trên CAP được thể hiện thông qua phần mở rộng của DCSL.

Bước 3 – Hợp thành các mối quan tâm để tạo DM hợp nhất: Tất cả các mối quan tâm liên quan đến cấu trúc và hành vi của DM – được biểu diễn bởi các aDSL như DCSL, AGL – sẽ được hợp thành để tạo nên một DM hợp nhất.

Bước 4 – Sinh và đánh giá bản mẫu phần mềm: DM hợp nhất đóng vai trò làm bản thiết kế cho quá trình sinh tự động các bản mẫu phần mềm. Khung phần mềm JDA có thể được sử dụng để thực hiện việc sinh mã này. Bản mẫu thu được sẽ được bàn giao cho các bên liên quan để đánh giá. Nếu có phản hồi, DM được quản lý sẽ được cập nhật tương ứng. Quy trình sau đó được lặp lại, hỗ trợ chu trình cải tiến liên tục cho đến khi bản mẫu phần mềm đáp ứng đầy đủ các yêu cầu đã chỉ định.

3.2.2 Tích hợp mẫu CAP vào mô hình miền

Phần này trình bày phương pháp cho việc đặc tả và quản lý các CAP. Là phương pháp mở rộng DCSL nhằm hỗ trợ biểu diễn các ràng buộc dựa trên CAP. Ngoài ra, thực hiện xây dựng một danh mục CAP ban đầu, cho phép biểu diễn và tích hợp các ràng buộc OCL vào DM, từ đó tạo ra DM hợp nhất và khả thi để thực thi.

3.2.2.1 Khía cạnh cú pháp

Ý tưởng cốt lõi của một CAP là sử dụng một mẫu để biểu diễn các ràng buộc OCL trong DM. Về bản chất, một CAP là một mẫu được *"tham số hóa"*. Mỗi lần áp dụng CAP nhằm biểu diễn một ràng buộc OCL cụ thể tương ứng với việc gán giá trị cụ thể cho các tham số của mẫu.

Các tham số đầu vào của mỗi mẫu được định nghĩa thông qua một cơ chế chú thích. Thực hiện mở rộng DCSL để biểu diễn CAP, cho phép đặc tả và quản lý chính quy các CAP trong cùng một khung mô hình hóa.

Xét trên phương diện cú pháp, mỗi mẫu CAP được mô tả với các thành phần chính sau:

Tên mẫu: Mỗi CAP được gán một định danh duy nhất để hỗ trợ quản lý và truy xuất trong danh mục CAP hiện có.

Mô tả: Cung cấp giải thích ngắn gọn về mục đích và ngữ nghĩa của nhóm ràng buộc OCL mà CAP biểu diễn.

Mẫu tham số hóa: Mô tả biểu thức OCL mang tính tham số, tuân thủ cú pháp OCL và tương ứng với một nhóm ràng buộc OCL có cùng cấu trúc. Cụ thể, mẫu bao gồm:

- *Cấu trúc biểu đồ lớp* — mô tả phần liên quan của biểu đồ lớp dùng để xác lập ngữ cảnh OCL. Cấu trúc này có các tham số biểu diễn tên lớp, thuộc tính và quan hệ.
- *Mẫu OCL* — biểu diễn dạng tham số hóa của biểu thức OCL.
- *Đặc tả chú thích* — định nghĩa các tham số thông qua cơ chế chú thích có cấu trúc, được sử dụng trong biểu thức OCL của mẫu.

Ví dụ: Cung cấp minh họa cụ thể về cách một CAP được áp dụng trong thực tế.

Tóm lại, CAP cung cấp một cơ chế tái sử dụng và có tham số hóa để biểu diễn các ràng buộc OCL lặp lại trong các mô hình miền. Theo đó, Định nghĩa 1 tóm tắt khái niệm CAP.

Định nghĩa 1 (Mẫu chú thích ràng buộc)**.** *Một Mẫu chú thích ràng buộc – CAP được định nghĩa là một khuôn mẫu có tham số, dùng để nắm bắt cấu trúc cú pháp và ngữ nghĩa chung của một lớp các ràng buộc OCL. Một cách hình thức, CAP được biểu diễn như một bộ: CAP* $= (N, D, \mathcal{T})$ *trong đó:*

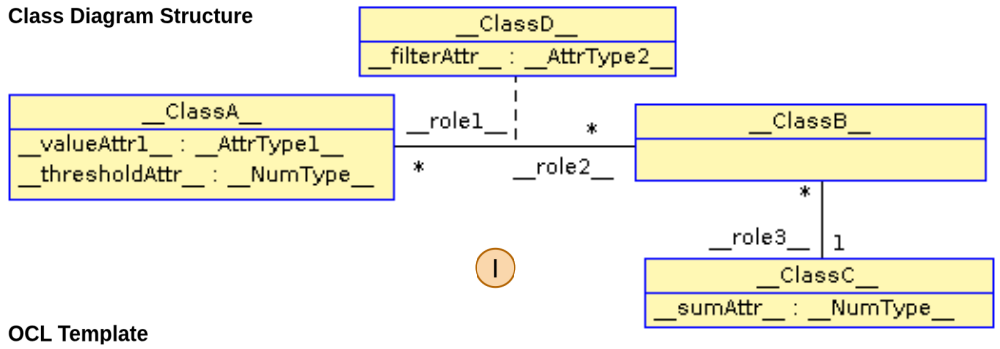
- *N là tên của mẫu;*
- *D là mô tả bằng văn bản nhằm giải thích mục đích và ngữ nghĩa của các ràng buộc OCL được biểu diễn bởi mẫu;*

- $\mathcal{T}$ *là một tập các bộ* $\langle P, T \rangle$*, trong đó:* P *là tập các tham số biểu diễn các thành phần biến đổi của khuôn mẫu (ví dụ: thuộc tính, liên kết hoặc giá trị), được đặc tả thông qua các chú thích;* T *là một biểu thức OCL có tham số mô tả cấu trúc của ràng buộc OCL.*

Một thể hiện của CAP được tạo ra bằng cách thay thế các tham số P *bằng các phần tử cụ thể của mô hình miền, từ đó sinh ra một ràng buộc OCL hợp lệ trong ngữ cảnh tương ứng.*□

3.2.2.2 Ví dụ và danh mục CAP đầu tiên

Trong phần này, giới thiệu một mẫu CAP có tên SumConstraint nhằm minh họa cho phương pháp đề xuất. Sau đó, mô tả danh mục CAP đầu tiên được thu thập và đặc tả từ nhiều nguồn yêu cầu miền khác nhau.



OCL Template

```
context ClassA inv __sumContraint01_invID__:
  let v:__NumType__=self.__ClassD__
      -> select(e | e.__filterAttr__=__value2__)
      -> collect(o | o.__role2__.__role3__.__sumAttr__)
      -> sum() in (self.__valueAttr1__=__value1__
      implies v<=self.__thresholdAttr__)
```

II

Annotation Specification

```
@SumConstraint(
  name=__sumConstraint01_invID__,
  assocCls=@AssocCls(assoc={__ClassD__}),
  rolePath=@RolePath(role={__role2__,__role3__}),
  collect={@AttrCond(attr=__filterAttr__,matchVal=__value2__)},
  sumAttr=__sumAttr__,
  fixAttr={ @AttrRef(maxLim=__thresholdAttr__)},
  ifPart={@AttrCond(attr=__valueAttr1__,matchVal=__value1__)}
)
```

III

Hình 3.2: Đặc tả cho mẫu CAP SumConstraint.

Tên mẫu: SumConstraint

Mô tả: Mẫu này định nghĩa các ràng buộc OCL nhằm kiểm soát và quản lý số lượng, giá trị hoặc tài nguyên thông qua các phép tổng hợp. Cụ thể, mẫu này đảm bảo rằng giá trị tổng hợp được lưu trữ trong một thực thể luôn bằng tổng các thành phần liên quan của nó. Mẫu được sử dụng để ngăn chặn sự sai lệch giữa dữ liệu dẫn xuất và dữ liệu nguồn, từ đó hỗ trợ đối soát và kiểm toán.

Mẫu tham số hóa: Biểu thức OCL tham số hóa của mẫu này được định nghĩa dựa trên các thành phần chính sau.

- *Cấu trúc biểu đồ lớp:* Hình 3.2 (nhãn I) minh họa biểu đồ lớp chứa ba lớp, các thuộc tính và các liên kết giữa chúng. Biểu đồ này xác lập ngữ cảnh cho mẫu ràng buộc OCL tương ứng.
- *Mẫu OCL:* Hình 3.2 (nhãn II) mô tả mẫu OCL dùng để sinh các ràng buộc thể hiện hạn chế sau: Tổng giá trị của thuộc tính `__sumAttr__` trên tất cả các đối tượng `o` trong một tập con được lọc từ tập các đối tượng `__ClassD__` liên kết với thể hiện `__ClassA__` hiện tại (`self`) được tính toán dựa trên tiêu chí lọc được xác định bởi biến `__filterAttr__`. Giá trị tổng được tính này sau đó được kiểm tra đối với `__thresholdAttr__` để đảm bảo điều kiện ngưỡng được thỏa mãn, và để bảo đảm rằng các đối tượng không vượt quá giới hạn cho phép. Lưu ý rằng tên các biến trong mẫu bắt đầu và kết thúc bằng "__". Tên biến `__sumConstraint01_invID__` cho biết đây là mẫu OCL loại 01 thuộc mẫu SumConstraint.
- *Đặc tả chú thích:* Mở rộng DCSL với các chú thích mới như minh họa trong Hình 3.2 (nhãn III), nhằm biểu diễn các tham số của mẫu OCL. Đặc tả chú thích này cho phép sinh ra ràng buộc OCL khi các tham số được gán giá trị cụ thể.

Ví dụ: Trên mô hình CourseMan, như thể hiện trong Hình 1.2, để minh họa mẫu này. Ràng buộc liên quan đảm bảo rằng số tín chỉ tối đa một sinh viên được phép đăng ký là một ngưỡng có giá trị xác định (12 tín chỉ) tín chỉ mỗi học kỳ khi sinh viên đang trong diện cảnh báo học vụ. Điều này giúp sinh viên tập trung cải thiện điểm trung bình (GPA). Ràng buộc được biểu diễn bằng OCL như sau:

```
context Student
    inv courseMan_inv06_ProbationRules:
    let v:Integer=self.enrolment
    ->select(e|e.status=EnrolStatus::ACTIVE)
    ->collect(e|e.offering.module.credits)->sum()
    in self.overallStatus=AcademicStatus::PROBATION
    implies v<=12
```

Thực hiện định nghĩa đặc tả trong DCSL để sinh ra ràng buộc như sau:

```
@SumConstraint(
    name='courseMan_inv06_ProbationRules',
    assocCls=@AssocCls(assoc={'Enrolment'}),
    rolePath=@RolePath(role={'offering','module'}),
    collect={@AttrCond(attr='status',
      matchVal='EnrolStatus::ACTIVE')},
    sumAttr='credits',
    fixAttr={@AttrRef(maxLim = 12)},
    ifPart={@AttrCond(attr='overallStatus',
    matchVal='AcademicStatus::PROBATION')}
)
    class Student {...}
```

Một mẫu CAP biểu diễn một họ ngữ nghĩa của các ràng buộc OCL có cấu trúc liên quan với nhau. Mỗi CAP có thể bao gồm nhiều biến thể khuôn mẫu OCL có tham số, tương ứng với các dạng cú pháp khác nhau của các ràng buộc trong cùng một lớp ngữ nghĩa.

Ví dụ, mẫu SumConstraint bao gồm một số biến thể khuôn mẫu OCL tương ứng với các nhóm ràng buộc dựa trên phép tổng hợp khác nhau. Khuôn mẫu OCL được minh họa trong Hình 3.2 là một trong các biến thể như vậy. Các biến thể khác có thể biểu diễn các ràng buộc tương tự nhưng có bổ sung các điều kiện lọc hoặc các vị từ ngữ cảnh.

Mặc dù các biến thể này cùng chia sẻ một tên mẫu, chúng khác nhau về cấu trúc biểu diễn. Trong quá trình mô hình hóa, biến thể khuôn mẫu phù hợp sẽ được lựa chọn và các tham số của nó được khởi tạo thông qua chú thích DCSL. Người dùng xây dựng các ràng buộc bằng cách khởi tạo chú thích CAP tương ứng với các giá trị tham số phù hợp. Từ đó, ràng buộc OCL cụ thể sẽ được sinh tự động từ biến thể khuôn mẫu đã chọn. Mẫu SumConstraint bao gồm nhiều biến thể mẫu OCL tương ứng với các nhóm ràng buộc OCL khác nhau. Mẫu OCL minh họa trong Hình 3.2 là một trong

những biến thể đó. Chẳng hạn, hãy xét một ràng buộc OCL khác trong DM được thể hiện ở Hình 1.2, như mô tả dưới đây. Ràng buộc này đảm bảo rằng sinh viên có GPA dưới 2.5 trong học kỳ trước chỉ được phép đăng ký tối đa 20 tín chỉ.

```
context TermRecord
    inv courseMan_inv17_LowGPACreditRestriction:
    self.prevSemGpa<2.5 implies
    self.student.enrolment
    ->select(e| e.offering.term=self.term)
    ->collect(e|e.offering.module.credits)->sum()<=20
```

Mẫu OCL tương ứng của SumConstraint để sinh ra ràng buộc này được định nghĩa như sau:

```
@SumConstraint(
    name='courseMan_inv17_LowGPACreditRestriction',
    assocCls=@AssocCls(assoc={'Enrollments'}),
    rolePath=@RolePath(role={'student','offering','module'}),
    rolePath2=@RolePath(role={'term','termRecord'}),
    collect={@AttrCond(attr='curTerm',matchVal='true')},
    sumAttr='credits',
    fixAttr={@AttrRef(maxLim=20)},
    ifPart={@AttrCond(attr='prevSemGpa',maxLim=2.5)}
)
    class TermRecord { ... }
```

Tiếp cận này đã xác định một danh mục CAP gồm mười mẫu, mỗi mẫu được đặc tả chính quy và được quản lý trong một khung mô hình hóa thống nhất. Mỗi CAP biểu diễn một họ ràng buộc, trong đó các ràng buộc có cùng mục đích ngữ nghĩa được gom nhóm và khái quát hóa thành một cấu trúc mẫu chung. Mỗi CAP bao gồm một tập hợp các kiểu nhằm phân biệt giữa các biến thể mẫu OCL khác nhau của cùng một mẫu CAP. Hai bất biến OCL được trình bày trong phần này được sinh bởi hai mẫu OCL tương ứng của mẫu SumConstraint.

3.2.2.3 Khía cạnh ngữ nghĩa

Ngữ nghĩa của CAP tương ứng trực tiếp với ngữ nghĩa của OCL, bởi mỗi thể hiện CAP có quan hệ một-một với một bất biến OCL trong DM. Sự tương ứng trực tiếp này bảo đảm rằng việc đánh giá bất kỳ ràng buộc CAP nào

cũng cho kết quả lô-gic giống hệt với việc đánh giá bất biến OCL tương ứng, qua đó duy trì ngữ nghĩa OCL trong DCSL.

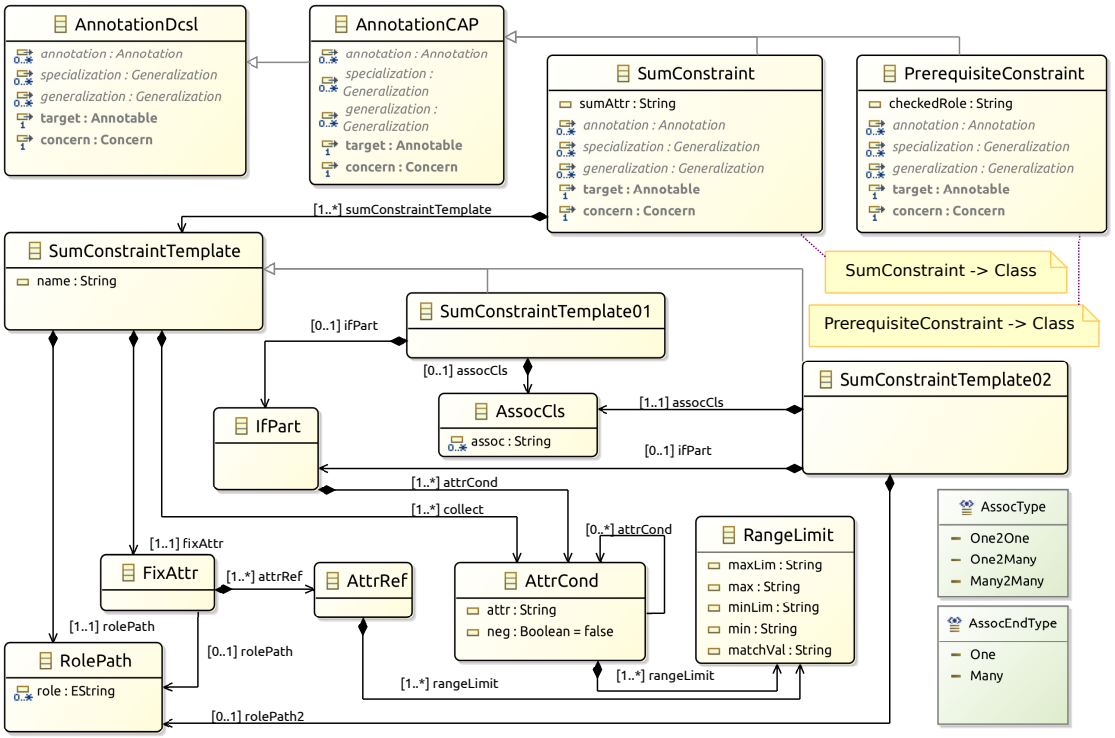
Định nghĩa 2 (Ứng dụng CAP)**.** *Cho CD là một biểu đồ lớp biểu diễn mô hình miền; Cho OCL_{CD} là tập tất cả các bất biến OCL hợp lệ có thể biểu diễn trên CD; Cho CAP_i là một mẫu* CAP *được định nghĩa bởi mẫu OCL tham số hóa T_i với không gian tham số $\mathcal{D}_i$. Một ứng dụng CAP trên CD là quá trình gán một bộ tham số $d_i \in \mathcal{D}_i$ cho T_i sao cho mẫu OCL của CAP trở thành một bất biến OCL hợp lệ trên CD. Một cách hình thức, quá trình này được biểu diễn bằng ánh xạ sinh: $\mathcal{G} : (CAP_i, d_i) \longmapsto oclInv \in OCL_{CD}$, trong đó $\mathcal{G}(CAP_i, d_i)$ biểu diễn bất biến OCL cụ thể được sinh bằng cách thể hiện mẫu CAP là T_i với tập tham số d_i*□

Ví dụ: Xét ràng buộc `courseMan_inv06_ProbationRules`. Ràng buộc này được sinh bằng cách áp dụng mẫu SumConstraint, như đã giải thích trong Mục 3.2.2.2. Ánh xạ sinh tương ứng được xác định như sau:

- CD là mô hình miền CourseMan, được mô tả trong Hình 1.2.
- CAP_i là mẫu `SumConstraint`, trong đó:
  - + T_i là mẫu OCL của CAP được minh họa trong Hình 3.2.
  - + $\mathcal{D}_i$ là không gian tham số được xác định trên CD;
  - + $d_i \in \mathcal{D}_i$ là bộ tham số tương ứng với đặc tả chú thích của bất biến `courseMan_inv06_ProbationRules`, như mô tả trong Mục 3.2.2.2.

Định nghĩa 3 (Tính đầy đủ của danh mục CAP)**.** *Cho $\mathcal{C}_{CAP}$ là tập các CAP. Danh mục $\mathcal{C}_{CAP}$ được gọi là đầy đủ đối với mô hình miền CD nếu, với mọi bất biến $oclInv \in OCL_{CD}$, tồn tại một $CAP_i \in \mathcal{C}_{CAP}$ và một bộ tham số $d_i \in \mathcal{D}_i$ sao cho: $\mathcal{G}(CAP_i, d_i) = oclInv$*□

Hình 3.3 minh họa phần mở rộng của siêu mô hình DCSL, như được trình bày trong Hình 4.4, nhằm biểu diễn CAPs. Mỗi CAP được mô hình hóa bởi một siêu lớp (*Meta-class*) tương ứng, chẳng hạn như `SumConstraint` và `PrerequisiteConstraint`. Siêu lớp này kế thừa từ `AnnotationDcsl`, đóng vai trò là điểm mở rộng của DCSL dành cho CAPs. Mỗi siêu lớp CAP bao gồm một tập các siêu lớp biểu diễn các mẫu CAP tương ứng. Mỗi mẫu



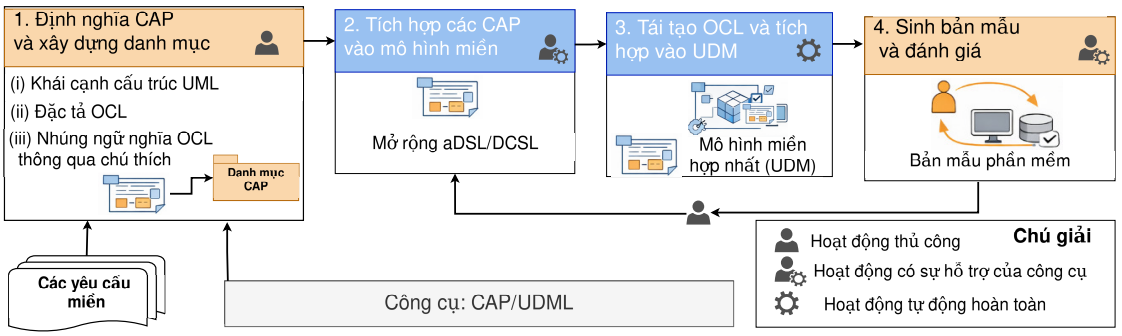
Hình 3.3: Mở rộng siêu mô hình DCSL để biểu diễn các CAP.

CAP gồm các tham số được đặc tả bằng các siêu lớp liên quan. Ví dụ, CAP `SumConstraint` bao gồm hai mẫu: `SumConstraintTemplate01` và `SumConstraintTemplate02`, cả hai cùng kế thừa một lõi chung được định nghĩa bởi `SumConstraintTemplate`. Các tham số của mỗi mẫu ánh xạ một-một với các thuộc tính của chú thích `SumConstraint` trong Hình 3.2.

3.2.3 Áp dụng mẫu CAP và sinh bản mẫu phần mềm

Mục này trình bày quy trình phương pháp áp dụng các mẫu chú thích ràng buộc (CAP) trong quá trình mô hình hóa hướng miền, cũng như việc sinh các bản mẫu phần mềm thực thi từ mô hình miền hợp nhất (UDM). Hình 3.4 minh họa tổng quan quy trình dựa trên CAP từ định nghĩa mẫu ràng buộc đến sinh bản mẫu phần mềm, kết hợp các giai đoạn thủ công, bán tự động và tự động.

Quy trình gồm bốn bước có liên kết lô-gic, trong đó các ràng buộc miền được chuyển đổi dần thành các bản mẫu phần mềm có thể thực thi. Tùy theo từng bước, các hoạt động có thể bao gồm các thao tác mô hình hóa



Hình 3.4: Quy trình sinh bản mẫu phần mềm dựa trên CAP.

thủ công, cấu hình bán tự động hoặc chuyển đổi hoàn toàn tự động, như được tóm tắt trong Hình 3.4.

Bước 1 – Định nghĩa CAP và xây dựng danh mục. Giai đoạn đầu tập trung vào việc xác định các cấu trúc ràng buộc có thể tái sử dụng và tổ chức chúng thành một danh mục CAP. Đây chủ yếu là hoạt động mô hình hóa thủ công do các chuyên gia miền hoặc người thiết kế ngôn ngữ thực hiện. Các CAP được xây dựng bằng cách phân tích các ràng buộc OCL lặp lại trong nhiều mô hình miền khác nhau và khái quát hóa chúng thành các họ ràng buộc có thể tái sử dụng. Mỗi CAP được đặc tả hình thức thông qua ba thành phần bổ trợ:

- một khía cạnh cấu trúc UML mô tả ngữ cảnh mô hình liên quan;
- một khuôn mẫu OCL có tham số đặc tả cấu trúc ngữ nghĩa của ràng buộc;
- một định nghĩa chú thích xác định không gian tham số cho việc khởi tạo mẫu.

Sau khi được định nghĩa, các khuôn mẫu CAP có thể được kiểm tra dựa trên ngữ cảnh cấu trúc để đảm bảo tính nhất quán ngữ nghĩa. Danh mục CAP thu được đóng vai trò như một kho mẫu ràng buộc có thể tái sử dụng cho nhiều mô hình miền khác nhau.

Bước 2 – Khởi tạo CAP trong mô hình miền. Ở giai đoạn thứ hai, các CAP được khởi tạo trong mô hình miền được biểu diễn bằng ngôn ngữ

đặc tả miền dựa trên chú thích. Đây là một hoạt động bán tự động, kết hợp giữa việc đặc tả thủ công của người mô hình hóa và việc diễn giải tự động của môi trường mô hình hóa. Thay vì đặc tả trực tiếp các ràng buộc OCL, người mô hình hóa biểu diễn các ràng buộc miền dưới dạng các chú thích CAP (ví dụ: `@SumConstraint`). Mỗi chú thích tương ứng với một phép gán tham số cụ thể $d_i \in \mathcal{D}_i$ cho khuôn mẫu CAP T_i, theo ánh xạ áp dụng CAP được định nghĩa trong Định nghĩa 2.

Các khía cạnh cấu trúc của miền được biểu diễn thông qua các lớp UML, trong khi ngữ nghĩa ràng buộc được đặc tả thông qua các chú thích CAP, liên kết các phần tử miền với các tham số của khuôn mẫu tương ứng. Cách tiếp cận này cho phép biểu diễn ràng buộc ở mức khai báo mà không cần thao tác trực tiếp với cú pháp OCL.

Bước 3 – Tái tạo OCL và tích hợp vào UDM. Sau khi các chú thích CAP được đặc tả, các ràng buộc OCL tương ứng được tái tạo tự động thông qua ánh xạ sinh (được trình bày trong Định nghĩa 3). Đây là một bước hoàn toàn tự động. Đối với mỗi khởi tạo CAP (CAP_i, d_i), khuôn mẫu OCL có tham số T_i được khởi tạo để tạo ra một ràng buộc cụ thể: $G(CAP_i, d_i) \longmapsto oclInv$.

Các ràng buộc sinh ra được tích hợp vào mô hình miền cùng với các đặc tả cấu trúc, tạo thành UDM. Do đó, UDM kết hợp hai thành phần bổ sung:

- đặc tả cấu trúc của các thực thể và quan hệ miền;
- các ràng buộc OCL hình thức được tái tạo từ các khuôn mẫu CAP.

Bước tái tạo tự động này bảo toàn ngữ nghĩa hình thức của OCL, đồng thời cho phép đặc tả ràng buộc thông qua các trừu tượng mức cao hơn. Mô hình kết quả có thể được kiểm chứng bằng các cơ chế kiểm tra OCL chuẩn nhằm đảm bảo tính đúng đắn cú pháp và sự nhất quán với cấu trúc miền.

Bước 4 – Sinh bản mẫu và đánh giá. UDM sau khi được kiểm chứng đóng vai trò đầu vào cho quá trình sinh bản mẫu phần mềm theo hướng mô hình. Từ mô hình này, một khung sinh mã có thể tự động tạo ra các bản mẫu phần mềm thực thi, bao gồm lớp miền, lô-gic kiểm tra ràng buộc, cấu hình lưu trữ và các thành phần giao diện người dùng.

Bản mẫu sinh ra cung cấp một biểu diễn vận hành của mô hình miền, cho phép đánh giá sớm các quy tắc miền và hành vi hệ thống. Việc đánh giá bản mẫu thường được thực hiện ở hai mức bổ trợ. Thứ nhất, tính đúng đắn ở mức cấu trúc và ràng buộc được kiểm tra thông qua các cơ chế kiểm chứng tự động. Thứ hai, các chuyên gia miền và các bên liên quan đánh giá bản mẫu để xác nhận rằng hành vi hệ thống phù hợp với các quy tắc nghiệp vụ mong muốn. Phản hồi từ quá trình đánh giá có thể dẫn đến việc điều chỉnh mô hình miền hoặc các khởi tạo CAP. Quy trình lặp này hỗ trợ việc đồng bộ liên tục giữa tri thức miền, đặc tả ràng buộc và các bản mẫu phần mềm thực thi.

3.3 Kỹ thuật tích hợp hành vi vào mô hình miền

Trong mục này, trình bày nghiên cứu về kỹ thuật tích hợp khía cạnh hành vi vào DM nhằm hỗ trợ sinh tự động các bản mẫu phần mềm theo tiếp cận DDD. Cụ thể, nội dung tập trung vào ba vấn đề chính: (1) đề xuất một cơ chế tích hợp hành vi miền vào DM hợp nhất thông qua một aDSL, được định nghĩa để biểu diễn hành vi miền phục vụ quá trình tích hợp; (2) trình bày phương pháp mô hình hóa hợp nhất cho phát triển phần mềm theo DDD, trong đó hành vi được gắn kết trực tiếp với DM; và (3) làm rõ vai trò của kỹ thuật tích hợp hành vi trong việc nâng cao khả năng biểu đạt của mô hình miền.

3.3.1 Tổng quan về phương pháp đề xuất

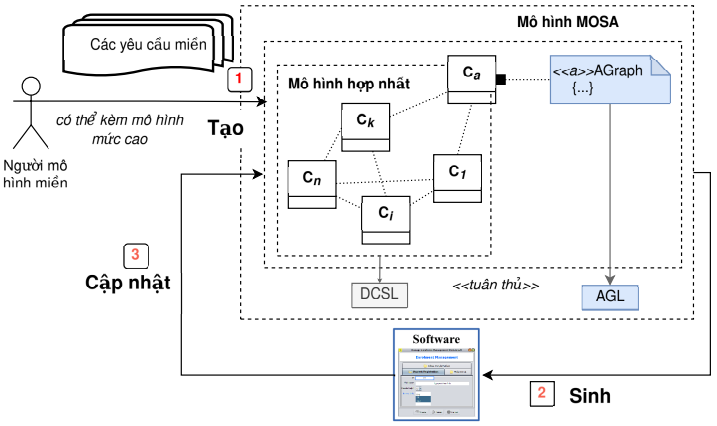
Đề xuất của luận án về kỹ thuật biểu diễn tích hợp hành vi vào DM theo DDD được mô tả trong hình 3.5, bao gồm ba bước lặp lại:

Trước hết, quá trình bắt đầu từ các yêu cầu miền, được biểu diễn bởi một DM thiết yếu, mô tả góc nhìn cấu trúc thông qua các khái niệm miền, và các biểu đồ hoạt động UML, mô tả hành vi miền. Mục tiêu là kết hợp các yêu cầu miền đầu vào này thành một DM hợp nhất, được cấu thành từ một mô hình DCSL và một mô hình AGL. Mô hình DCSL mở rộng DM thiết yếu nhằm liên kết góc nhìn cấu trúc với góc nhìn hành vi. Mô hình AGL biểu diễn các hành vi miền. Định nghĩa chi tiết hơn của mô hình DCSL

được trình bày ở Mục 3.3.1.2. Các Mục 3.3.1.1 và 3.3.2 giải thích chi tiết cách định nghĩa mô hình AGL.

Thứ hai, DM hợp nhất được sử dụng làm đầu vào để tự động sinh phần mềm dựa trên giao diện đồ họa và mô-đun. Phần mềm sinh ra sẽ được trình bày cho chuyên gia miền để thu thập phản hồi.

Thứ ba, nếu có phản hồi, mô hình đầu vào sẽ được cập nhật và chu trình được lặp lại. Nếu chuyên gia miền hài lòng với các mô hình, chu trình kết thúc.



Hình 3.5: Kỹ thuật biểu diễn tích hợp hành vi vào DM theo DDD.

3.3.1.1 Tích hợp hành vi miền

Cơ chế được đề xuất để tích hợp hành vi miền vào DM dựa trên cấu trúc và ngữ nghĩa hành vi tại hai điểm. Thứ nhất, xác định một tập hành động thiết yếu cho mỗi lớp mô-đun sở hữu một lớp miền tương ứng. Các hành động thiết yếu này là hành động nguyên tử (trình bày chi tiết hơn trong Mục 3.3.2), cho phép thao tác trên các thể hiện của lớp miền. Thứ hai, hành vi miền được xem như sự hợp tác giữa các mô-đun trong MOSA [72]. Mỗi sự hợp tác mô-đun được điều phối bởi một mô-đun hợp thành và được mô tả bởi một mô hình hoạt động tương ứng. Mỗi mô hình hoạt động được ánh xạ đến một lớp miền mới, gọi là lớp hoạt động, lớp này trực thuộc mô-đun hoạt động tương ứng.

Cơ chế trên cho phép sử dụng biểu đồ hoạt động UML để biểu diễn hành vi miền, nhưng có những giới hạn nhằm đảm bảo rằng các biểu đồ này phù hợp với ngữ nghĩa hành vi của mô-đun hợp thành điều phối sự hợp tác giữa các mô-đun. Để đạt được điều này, áp dụng phương pháp dựa trên mẫu,

trong đó hành vi miền được đặc tả bằng biểu đồ hoạt động UML sử dụng các cấu trúc cơ bản tương ứng với năm mẫu mô hình hoạt động thiết yếu được trình bày trong [70]. Các mẫu này được đặt tên theo năm luồng hoạt động cơ bản: tuần tự, rẽ nhánh quyết định, phân nhánh song song, hợp nhất song song và gộp nhánh. Mô tả chi tiết hơn được trình bày trong Mục 3.3.3.

3.3.1.2 Mô hình lớp hợp nhất

Mô hình lớp hợp nhất là phần mở rộng của DM nhằm cho phép tích hợp các hành vi miền. Trong cách tiếp cận này, hành vi miền được mô tả bằng các mô hình hoạt động sử dụng các biểu đồ hoạt động UML.

Các lớp hoạt động—chẳng hạn lớp $\texttt{C}_\texttt{a}$ trong Hình 3.5—được thêm vào để biểu diễn từng hoạt động trong hành vi miền. Các lớp hoạt động này liên kết với mô hình hoạt động tương ứng, đóng vai trò như đồ thị hoạt động và đồng bộ hóa lô-gic hành vi của hoạt động với trạng thái hiện tại của DM. Mô hình lớp hợp nhất thu được có thể hiện thực bằng DCSL, và được gọi là mô hình hợp nhất DCSL.

Định nghĩa 4. (Mô hình lớp hợp nhất) *Cho trước một mô hình hoạt động được đặc tả bằng biểu đồ hoạt động UML để biểu diễn hành vi miền. Một* ***mô hình lớp hợp nhất*** *liên quan đến mô hình hoạt động là DM được mở rộng với các thành phần sau:*

- ***lớp hoạt động:*** *một lớp miền đại diện cho hoạt động.*
- ***lớp thành phần dữ liệu (hay gọi tắt là lớp dữ liệu):*** *một lớp miền đại diện cho mỗi kho dữ liệu.*
- ***lớp thành phần điều khiển (hoặc lớp điều khiển):*** *nắm bắt trạng thái miền cụ thể của nút điều khiển. Một lớp điều khiển đại diện (không đại diện) một nút điều khiển được đặt tên theo (phủ định) ví dụ, lớp quyết định (không quyết định), lớp tham gia (không tham gia), v.v.*
- ***liên kết dành riêng cho hoạt động:*** *liên kết giữa mỗi cặp lớp sau:*
  - *lớp hoạt động và lớp hợp nhất.*
  - *lớp hoạt động và lớp rẽ nhánh.*
  - *một lớp hợp nhất (rẽ nhánh) và một lớp dữ liệu đại diện cho kho lưu trữ dữ liệu của một nút hành động được kết nối với nút hợp nhất (rẽ nhánh).*

+ lớp hoạt động và một lớp dữ liệu không đại diện cho kho dữ liệu của nút hành động được kết nối với nút hợp nhất hoặc nút rẽ nhánh.

Gọi chung các lớp dữ liệu và điều khiển của mô hình lớp hoạt động là các lớp thành phần. □

Lưu ý rằng biểu đồ biểu diễn trong định nghĩa trên không bao gồm tất cả các mối liên kết có thể có giữa các lớp thành phần. Nó chỉ tập trung vào các hoạt động cụ thể, tức là nói chung, trong phạm vi của luận án chỉ tập trung vào miền ngữ nghĩa hạn chế của biểu đồ hoạt động UML cho AGL.

Các liên kết đóng hai vai trò quan trọng: Đầu tiên, mô hình hóa rõ ràng các liên kết giữa các trạng thái theo miền cụ thể của các nút hành động. Thứ hai, chúng được sử dụng để kết hợp các mô-đun của các lớp dữ liệu và điều khiển vào cây chứa mô-đun hoạt động, do đó thúc đẩy mô-đun này trở thành mô-đun chính để quản lý toàn bộ hoạt động.

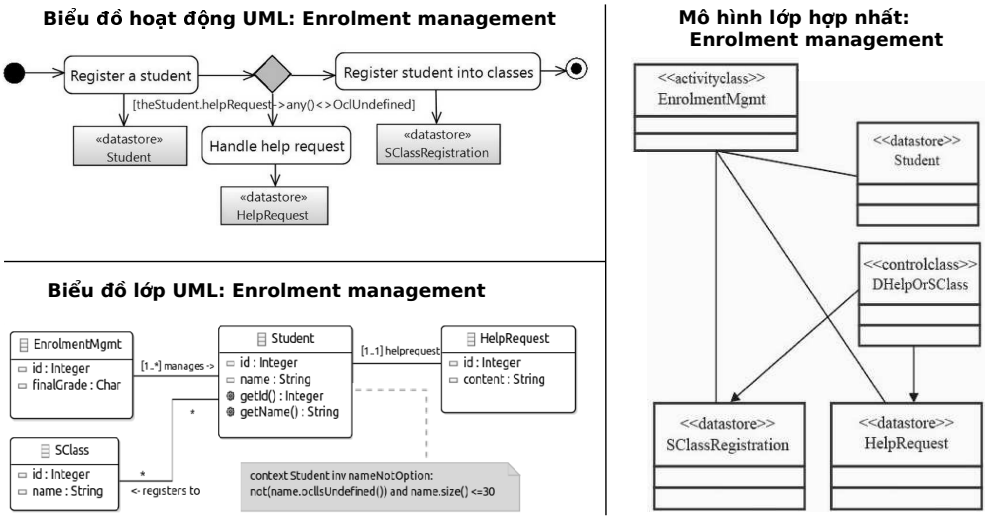
Điều kiện áp đặt cho cặp liên kết dành riêng cho hoạt động của lớp thứ tư xuất phát từ thực tế là không cần xác định rõ ràng mối liên kết giữa một lớp hoạt động và một lớp dữ liệu đại diện cho kho lưu trữ dữ liệu của một nút hành động được kết nối với hợp nhất hoặc nút rẽ nhánh. Một lớp dữ liệu như vậy được liên kết "gián tiếp" với lớp hoạt động, thông qua hai liên kết: một là giữa nó và lớp hợp nhất hoặc rẽ nhánh (cặp lớp thứ ba), và lớp khác là giữa lớp hoạt động và lớp điều khiển này (cặp lớp đầu tiên hoặc cặp lớp thứ hai).

Định nghĩa 5. (Mô hình hợp nhất DCSL) *Mô hình hợp nhất DCSL là một mô hình DCSL hiện thực hóa mô hình lớp hợp nhất như sau:*

- *một lớp miền C_a (được gọi là lớp miền hoạt động) để hiện thực hóa lớp hoạt động.*
- *các lớp miền $C_1, ..., C_n$ để hiện thực hóa các lớp thành phần.*
- *để $C_{i1}, ..., C_{ik} \in \{C_1, ..., C_n\}$ hiện thực hóa các lớp thành phần không quyết định và không tham gia, sau đó $C_a, C_{i1}, ..., C_{ik}$ chứa các trường kết hợp nhận ra các đầu liên kết tương ứng của các liên kết dành riêng cho hoạt động có liên quan.* □

Để đơn giản ký pháp, lớp hoạt động được dùng để chỉ lớp miền C_a và lớp thành phần để chỉ các lớp $C_1, ..., C_n$.

Ví dụ: Hình 3.6 (A) mô tả biểu đồ hoạt động và biểu đồ lớp UML của quản lý việc đăng ký (tên là CourseMan), trong khi Hình 3.6(B) minh họa mô hình lớp hợp nhất kết quả của hoạt động. Biểu đồ lớp hợp nhất bao gồm năm lớp miền và việc hiện thực hóa năm mối liên kết dành riêng cho hoạt động. Lớp `EnrolmentMgmt` đóng vai trò lớp hoạt động; `DHelpOrSClass` là lớp quyết định ghi nhận lô-gic quyết định miền chuyên biệt. Ba lớp còn lại là các lớp dữ liệu tương ứng với ba kho dữ liệu, đồng thời cũng là các lớp miền trong biểu đồ lớp UML. Ba lớp còn lại là các lớp dữ liệu tương ứng với ba kho dữ liệu và chúng cũng tương ứng với ba lớp miền trong biểu đồ lớp UML. Mô hình lớp hợp nhất bỏ qua các liên kết theo miền cụ thể.



Hình 3.6: (A: Bên trái) biểu đồ hoạt động và biểu đồ lớp UML của CourseMan xử lý việc hoạt động đăng ký; (B: Phải) Kết quả là mô hình lớp hợp nhất.

Ba trong số các liên kết này liên kết lớp hoạt động `EnrolmentMgmt` với các lớp dữ liệu, liên kết các mô-đun của chúng với cây chứa mô-đun hoạt động `ModuleEnrolmentMgmt`. Hai liên kết còn lại liên kết lớp quyết định `DHelpOrSClass` với hai lớp dữ liệu `SClassRegistration` và `HelpRequest`, tương ứng với các kho dữ liệu được kết nối với hai nút hành động phân nhánh từ nút quyết định. Các liên kết này được coi là liên kết phụ thuộc yếu và được đưa vào trường hợp này để cho phép lô-gic quyết định được gói gọn bởi `DHelpOrSClass` để tham chiếu hai lớp dữ liệu.

3.3.2 Ngữ nghĩa hành động mô-đun

Trong mục này trình bày định nghĩa hình thức của *hành động mô-đun* dựa trên biểu đồ hoạt động của UML [91]. Định nghĩa này tập trung mô tả cấu trúc của hành động mô-đun cùng các tiền và hậu trạng thái của nó.

Hành động mô-đun được định nghĩa đệ quy, bắt đầu từ kiểu hành động nguyên thủy nhất gọi là hành động nguyên tử. Các hành động nguyên tử này sau đó được kết hợp để tạo thành chuỗi hành động nguyên tử và rộng hơn là hành động nguyên tử có cấu trúc, dẫn đến một đặc tả chính xác về ngữ nghĩa hành vi của các mô-đun trong MOSA.

3.3.2.1 Hành động nguyên tử

Mặc dù mỗi mô-đun có thể khác nhau, có thể quan sát rằng tồn tại một tập các hành vi nguyên thủy làm nền tảng cho mọi mô-đun. Các hành vi nguyên thủy này được khái quát hóa thành hành động nguyên tử.

Định nghĩa 6. (Hành động nguyên tử) *Một* ***hành động nguyên tử*** *là hành vi mô-đun nhỏ nhất nhưng có ý nghĩa, được cung cấp cho một tác nhân (con người hoặc mô-đun/hệ thống khác) thông qua khung nhìn nhằm thao tác trên các đối tượng miền của lớp miền tương ứng. Hành động nguyên tử được đặc trưng bởi:*

- `name`*: tên của hành động.*
- `preStates` *(tương ứng* `localPrecondition` *[91]): tập các trạng thái mà mô-đun hiện tại phải đang ở để hành động này có thể được thực hiện.*
- `postStates` *(tương ứng* `localPostcondition` *[91]): tập các trạng thái mà hành động đạt tới khi hoàn tất quá trình thực thi trên mô-đun hiện tại.*
- `fieldValSet` *(tương ứng* `input` *[91]): biểu diễn dữ liệu đầu vào của hành động. Đây là một tập các cặp (f, v), trong đó f là tên của một trường miền (domain field) của lớp miền, và v là giá trị được hành động gán cho trường đó.*
- `output`*: lớp miền được sử dụng cho các hành động thao tác trên đối tượng, và rỗng đối với tất cả các hành động còn lại.*

  Mặc dù thuộc tính `name` *là duy nhất, để thuận tiện trình bày, thường liệt kê thêm* `postStates` *và* `fieldValSet` *cùng với* `name`*. Vì vậy, một*

hành động nguyên tử a được ký hiệu $a = (o, s, i)$ *có thể được ký hiệu, trong đó o là* `name`*, s là* `postStates`*, và i là* `fieldValSet`*. Dấu chấm được dùng để truy cập thành phần, ví dụ:* a.`postStates` $= s$□

Các lưu ý về định nghĩa trên: Thứ nhất, các trạng thái mô-đun được dùng để trừu tượng hóa các tiền điều kiện-trạng thái và hậu-trạng thái của mỗi hành động, giúp tạo ra sự linh hoạt trong việc kết hợp các hành động dựa trên trạng thái để hình thành hành vi phức tạp hơn. Một trạng thái mô-đun biểu diễn trạng thái của MVC của mô-đun khi xử lý một hành động mô-đun. Một số trạng thái có thể diễn ra đồng thời, được gọi là trạng thái đồng thời và được biểu diễn bằng toán tử "+". Hành động nguyên tử có đúng một hậu trạng thái `postStates`; các hành động phức tạp hơn có thể có nhiều hậu trạng thái `postStates`. Thứ hai, một khía cạnh quan trọng của hành động là giá trị đầu vào. Do các hành động thao tác trên các trường miền, đầu vào được định nghĩa như một tập các cặp trường–giá trị (có thể rỗng), trong đó giá trị có thể được cung cấp bởi người dùng hoặc được sinh ra từ các hành động trước đó trong hành vi hợp thành. Thứ ba, mỗi hành động có tối đa một kiểu đầu ra, tương ứng với lớp miền của mô-đun hiện tại. Chỉ các hành động thao tác đối tượng mới sinh ra đầu ra này; các hành động khác không tạo ra giá trị kết quả.

Bảng 3.1 liệt kê các hành động nguyên tử cốt lõi. Nghiên cứu này chia các hành động thành hai nhóm. Nhóm thứ nhất bao gồm các hành động liên quan đến ngữ cảnh hoạt động tổng thể của mô-đun. Các hành động trong nhóm này gồm: `open`, `newObject`, `setDataFieldValues`, `reset`, và `cancel`. Các `postStates` của các hành động này bao gồm các trạng thái: `Opened`, `NewObject`, `Editing`, `Reset`, và `Cancelled` (tương ứng). Nhóm thứ hai bao gồm ba hành động thao tác đối tượng miền thiết yếu: `createObject`, `updateObject`, và `deleteObject`. Các `postStates` của những hành động này bao gồm các trạng thái: `Created`, `Updated`, và `Deleted` (tương ứng).

Lưu ý từ Bảng 3.1 rằng chỉ có hành động `setDataFieldValues` yêu cầu `fieldValSet` được chỉ định làm đầu vào. Các hành động khác không yêu cầu đầu vào, do đó tập này là rỗng đối với chúng. Cũng lưu ý rằng hai trạng thái của mô-đun `ObjIsPresent` và `ObjIsNotPresent` có thể xảy ra đồng thời với bất kỳ một trong các trạng thái sau: `Editing`, `Reset`, và `Cancelled`. Ví dụ, trạng thái đồng thời `Editing` + `ObjIsPresent` có nghĩa là mô-đun hiện

Bảng 3.1: Các hành động nguyên tử cốt lõi

name	preStates	postStates	Mô tả
`open`	{`Init`}	{`Opened`}	Mở giao diện của mô-đun để hiển thị lớp miền.
`newObject`	{`Opened`, `Created`, `Updated`, `Reset`, `Cancelled`}	{`NewObject`}	Xóa khỏi giao diện bất kỳ đối tượng nào đang được hiển thị và chuẩn bị giao diện để tạo một đối tượng mới.
`setDataField−Values`	{`NewObject`, `Editing`, `Created`, `Updated`, `Reset`, `Cancelled`}	{`Editing`}	Gán giá trị cho một tập con các trường dữ liệu của giao diện.
`createObject`	{`NewObject`, `Editing` + `ObjIsNotPresent`}	{`Created`}	Tạo một đối tượng mới từ dữ liệu được nhập trên giao diện. Đối tượng vừa tạo sẽ được hiển thị trên giao diện.
`updateObject`	{`Editing` + `ObjIsPresent`}	{`Updated`}	Cập nhật đối tượng hiện tại từ dữ liệu được nhập trên giao diện. Đối tượng đã được cập nhật vẫn được hiển thị trên giao diện.
`deleteObject`	{`Created`, `Updated`, `Reset` + `ObjIsPresent`, `Cancelled` + `ObjIsPresent`}	{`Deleted`}	Xóa đối tượng hiện tại. Đối tượng bị xóa sẽ được loại khỏi giao diện.
`reset`	{`Editing`}	{`Reset`}	Khởi tạo lại giao diện để hiển thị lại đối tượng hiện tại (hủy bỏ toàn bộ dữ liệu người dùng đã nhập).
`cancel`	{`NewObject`, `Editing` + `ObjIsNotPresent`}	{`Cancelled`}	Hủy thao tác tạo đối tượng mới (hủy bỏ toàn bộ dữ liệu người dùng đã nhập, nếu có).

đang hiển thị một đối tượng trên giao diện và đối tượng này đang được người dùng chỉnh sửa. Ngược lại, `Editing` + `ObjIsNotPresent` có nghĩa là mô-đun hiện đang yêu cầu người dùng nhập dữ liệu cho một đối tượng mới; đối tượng này vẫn chưa được tạo ra.

3.3.2.2 Chuỗi hành động nguyên tử

Trong thực tế, các hành động nguyên tử cốt lõi được kết hợp theo trình tự để tạo thành những hành vi hữu ích hơn. Hành vi này, được gọi là chuỗi hành động nguyên tử (*Atomic Action Sequence - ASE*), tương ứng với một kịch bản tương tác. Mô hình hóa chuỗi này bằng các hành động có cấu trúc trong biểu đồ hoạt động UML [91]. Ký hiệu `first` và `last` là hai hàm trả về phần tử đầu tiên và phần tử cuối cùng (tương ứng) trong một chuỗi.

Định nghĩa 7. (Chuỗi hành động nguyên tử và trạng thái có thể đạt tới) *Một* ***chuỗi hành động nguyên tử*** $S = (a_1, \ldots, a_n)$ *được gọi là*

một hành động của mô-đun khi và chỉ khi $a_i.\texttt{postStates} \subseteq a_{i+1}.\texttt{preStates}$ $(\forall\, a_i, a_{i+1} \in S)$. *Chuỗi* S *có các thuộc tính sau:*

- $S.\texttt{preStates} = \texttt{first}(S).\texttt{preStates}$
- $S.\texttt{postStates} = \texttt{last}(S).\texttt{postStates}$
- $S.\texttt{fieldValSet} = \texttt{first}(S).\texttt{fieldValSet}$
- $S.\texttt{output} = \texttt{last}(S).\texttt{output}$

Một trạng thái của mô-đun s' *được gọi là* ***có thể đạt tới*** *từ một hành động nguyên tử* a *nếu tồn tại một chuỗi hành động nguyên tử* S *sao cho* $a = \texttt{first}(S)$ *và* $S.\texttt{postStates} = s'$. *Khi đó,* a *được gọi là hành động nguồn của* s'. □

Rõ ràng, một hành động nguyên tử luôn có thể đạt đến hậu trạng thái của chính nó. Việc xác định các trạng thái có thể đạt tới cho các hành động nguyên tử được trình bày trong Bảng 3.1. Trước hết, các trạng thái có thể đạt tới của hành động `open` bao gồm `Opened`, `NewObject`, `Editing`, `Created`, `Updated`, `Deleted`, `Reset`, và `Cancelled`. Thứ hai, các trạng thái có thể đạt tới của `newObject` bao gồm `NewObject`, `Editing`, `Created`, `Reset`, và `Cancelled`. Ngoài ra, `newObject` không thể đạt tới `Updated` và `Deleted` vì hành động này được dành riêng cho việc tạo một đối tượng mới và không thể dẫn đến việc cập nhật hoặc xóa một đối tượng đã tồn tại. Thứ ba, các trạng thái có thể đạt tới của `setDataFieldValues` bao gồm `Editing`, `Created`, `Updated`, và `Reset`. Hành động này không thể đạt tới `NewObject`, `Deleted` và `Cancelled` vì nó chỉ liên quan đến dữ liệu nhập vào và không thể khởi tạo hoặc hủy việc tạo đối tượng, cũng như không thể dẫn đến việc xóa đối tượng. Cuối cùng, năm hành động còn lại mỗi hành động chỉ có một trạng thái có thể đạt tới, đó chính là trạng thái của riêng hành động đó. Các hành động này là "`stubs`" theo nghĩa chúng kết thúc mọi chuỗi ASE dẫn đến chúng.

3.3.2.3 Hành động nguyên tử có cấu trúc

Ở mức tổng quát hơn, trong phần này trình bày xác định một tập các ASE liên quan có thể hình thành một hành động nguyên tử có cấu trúc. Về bản chất, hành động này định nghĩa một hành vi tổng quát bao gồm nhiều kịch bản tương tác thay thế nhau (mỗi kịch bản được đặc tả bởi một ASE trong

tập), và những kịch bản này thường được người dùng thực hiện (có thể đồng thời).

Định nghĩa 8. (Hành động nguyên tử có cấu trúc) *Một **hành động nguyên tử có cấu trúc - SAA**, đối với một hành động nguyên tử nguồn a và một tập các hậu trạng thái $E = \{s_1, \ldots, s_n\}$ có thể đạt tới từ a, là tập $A = \{S : ASE \mid \texttt{first}(S) = a,\ S.\texttt{postStates} \subseteq E\}$, trong đó:*

- $A.\texttt{preStates} = a.\texttt{preStates}$
- $A.\texttt{postStates} = E$
- $A.\texttt{fieldValSet} = a.\texttt{fieldValSet}$
- $A.\texttt{output} = \bigcup_{S \in A}(S.\texttt{output})$

Ở dạng trừu tượng, viết $A = (a, \{s_1, \ldots, s_n\}, i)$ với $i = a.\texttt{fieldValSet}$. Nếu $i = \varnothing$ thì được lược bỏ thành phần này và chỉ viết $A = (a, \{s_1, \ldots, s_n\})$.□

Rõ ràng, SAA khái quát hóa cả hành động nguyên tử lẫn ASE: một ASE là một SAA chỉ có một phần tử, trong khi một hành động nguyên tử a chính là SAA $(a, \{a.\texttt{postState}\})$. Hơn nữa, SAA ngắn gọn hơn đáng kể so với việc xây dựng một tập ASE: tất cả những gì cần làm là chỉ định hành động nguyên tử khởi đầu và các hậu trạng thái mong muốn.

3.3.3 Các mẫu hành vi miền

Phần này trình bày cách tiếp cận dựa trên mẫu nhằm tích hợp các hành vi miền vào DM. Mỗi hành vi miền, được mô tả ở mức trừu tượng cao bằng biểu đồ hoạt động UML và các câu lệnh dựa trên DM, được chuyển đổi thành một đặc tả gồm hai thành phần: (1) một phần của mô hình lớp hợp nhất, trong đó bổ sung các lớp hoạt động mới; và (2) lô-gic đồ thị hoạt động tương ứng với hành vi đầu vào, cùng với các ánh xạ kết nối đồ thị này với mô hình lớp hợp nhất. Quá trình chuyển đổi được thực hiện thông qua việc áp dụng các mẫu hành vi miền, được định nghĩa tương ứng với năm mẫu mô hình hóa hoạt động UML thiết yếu [70], đóng vai trò như các ràng buộc nhằm tinh chỉnh miền ngữ nghĩa của AGL.

3.3.3.1 Đặc tả các mẫu hành vi miền

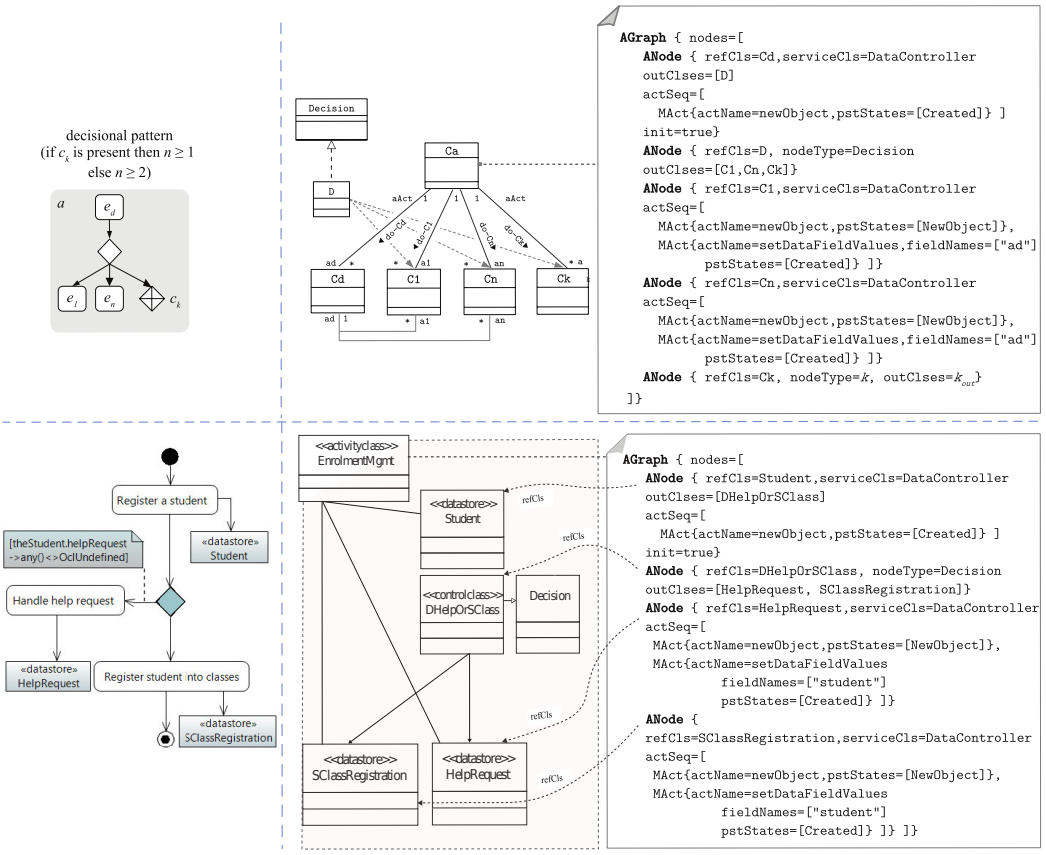
Nghiên cứu này đặc biệt quan tâm đến thiết kế các dạng mẫu [54, 64]. Để bảo đảm tính tổng quát của các mẫu, với mỗi dạng mẫu, luận án trình bày một biểu đồ hoạt động UML và một mô hình lớp hợp nhất cấu hình, hiện thực hóa mẫu tương ứng. Mô hình mẫu này là một mô hình DCSL hợp nhất đã được *"tham số hóa"*, trong đó các phần tử của các siêu khái niệm (không phải chú thích) được đặt tên theo vai trò tổng quát mà chúng đảm nhiệm trong mẫu.

Để đơn giản hóa việc trình bày, các trường kết hợp và các phương thức miền cơ bản được lược bỏ khỏi biểu đồ của mô hình mẫu. Gắn với mỗi mô hình mẫu là một đặc tả đồ thị hoạt động trong AGL (được trình bày chi tiết trong Mục 3.3.4), nhằm mô tả lô-gic của đồ thị hoạt động tương ứng với hành vi đầu vào, đồng thời duy trì sự đồng bộ giữa trạng thái thực thi của hoạt động và trạng thái hiện tại của miền trong mô hình lớp hợp nhất.

Phương pháp đặc tả các mẫu hành vi miền được xây dựng dựa trên năm mẫu mô hình hóa hoạt động UML thiết yếu, bao gồm: *Sequential*, *Decisional*, *Forked*, *Joined* và *Merged*. Mỗi mẫu tương ứng với một biến thể của DM hợp nhất dành cho các hoạt động trong hệ thống CourseMan. Đối với mỗi mẫu, luận án cung cấp một ví dụ minh họa bao gồm DM hợp nhất đã được cấu hình tương ứng và một hoặc nhiều giao diện người dùng, được trình bày trong Mục 6.2.2. Tuy nhiên, để làm rõ cơ chế đặc tả và tích hợp hành vi miền, luận án tập trung phân tích chi tiết phương pháp đặc tả mẫu hành vi miền cho mẫu quyết định (*Decisional pattern*) như một trường hợp đại diện.

Mẫu quyết định (Decisional pattern)

Phần trên bên trái của Hình 3.7 minh họa biểu đồ hoạt động UML, trong khi phần trên bên phải thể hiện mô hình lớp hợp nhất được cấu hình mẫu. Ngoài lớp hoạt động `C_a`, mô hình này bao gồm năm lớp miền khác: `C_d`, `D`, `C_1`, `C_n`, và `C_k`, tương ứng với năm nút hành động. Đặc biệt, lớp `C_k` là một lớp điều khiển được tham chiếu bởi nút điều khiển c_k của biểu đồ hoạt động. Lớp `D` là lớp quyết định triển khai giao diện `Decision`. Vì lô-gic quyết định có thể cần biết về các lớp miền liên quan (cụ thể là `C_1`, `C_n`, và `C_k`), nên tồn tại các liên kết phụ thuộc yếu tùy chọn giữa `D` và các lớp này. Tùy thuộc vào yêu



Hình 3.7: Đặc tả mẫu hành vi miền cho mẫu quyết định.

cầu miền, một số liên kết này có thể cần hoặc không cần được sử dụng. Lớp `Ca` được liên kết với bốn lớp miền còn lại theo mối quan hệ một–nhiều. Cũng cần lưu ý rằng liên kết với lớp `Ck` có thể đóng vai trò như một cầu nối đến các khối luồng hoạt động khác trong một biểu đồ hoạt động lớn hơn. Tuy nhiên, nếu c_k là một nút quyết định, thì `Ck` không có liên kết nào, và liên kết đến `Ck` sẽ được thay thế hoặc "mở rộng" thành một tập các liên kết nối trực tiếp `Ca` với các lớp miền của mô hình có chứa `Ck`. Trong mô hình mẫu, hai liên kết giữa `Cd` và `C1`, `Cn` cho thấy rằng cả `C1` và `Cn` đều biết đến `Cd` do việc truyền các thẻ đối tượng từ e_d sang e_1 và e_n thông qua nút quyết định.

Đặc tả AGL cho mẫu này bao gồm năm `ANode`. `ANode` đầu tiên tạo một đối tượng `Cd` mới, trong khi `ANode` thứ hai thực thi lô-gic quyết định. `ANode` thứ ba và thứ tư biểu diễn hai trường hợp quyết định: trường hợp thứ nhất tạo một đối tượng `C1` mới cho đối tượng `Cd` đã cho, và trường hợp thứ hai (được lặp lại cho tất cả n) tạo một đối tượng `Cn` mới cho cùng một `Cd`. `ANode` thứ năm được sử dụng khi `Ck` được chỉ định và bao gồm hai biến, k và k_{out}, cả hai đều phụ thuộc vào `Ck`. Biến k đặc tả kiểu của nút điều khiển, trong khi biến k_{out} đặc tả mảng các lớp miền đầu ra của `Ck`.

3.3.3.2 Áp dụng các mẫu hành vi miền

Để thu được một đặc tả AGL cụ thể khi áp dụng một mẫu hành vi miền, về cơ bản thực hiện theo ba bước chính:

1. Biểu diễn biểu đồ hoạt động UML đầu vào và mô hình lớp hợp nhất được cấu hình mẫu dưới dạng các đồ thị hoạt động;

2. Xác định việc ghép nối giữa mô hình mẫu và hoạt động nhằm xác định các `ANode`, thứ tự của chúng, và các lớp miền được tham chiếu bởi chúng. Việc tham chiếu từ các `ANode` đến các lớp miền (được thể hiện bằng các từ khóa `refCls` và `outCls`) cung cấp một ánh xạ giữa activity và mô hình lớp hợp nhất;

3. Biểu diễn hành vi tổng quát của mô-đun miền đối với lớp miền được tham chiếu bởi mỗi `ANode` thông qua một SAA bằng từ khóa `actSeq`. Điều này đảm bảo rằng trạng thái thực thi của hoạt động được đồng bộ với trạng thái hiện tại của mô hình lớp hợp nhất.

Định nghĩa 9. (DM hợp nhất dựa trên AGL) *Cho một mô hình lớp hợp nhất chứa một tập không rỗng các lớp hoạt động, mỗi lớp trong số đó được gắn với một đặc tả AGL mô tả lô-gic đồ thị hoạt động của một hoạt động miền. Một DM hợp nhất D là sự kết hợp giữa mô hình lớp hợp nhất và các đặc tả AGL. Một phần mềm được sinh ra trong MOSA đối với D bao gồm một tập các mô-đun, mỗi mô-đun sở hữu một lớp miền trong D, và hành vi của hành động* `newObject` *của mỗi mô-đun sở hữu một lớp hoạt động sẽ bao gồm lô-gic được mô tả bởi đồ thị hoạt động được cấu hình bởi đặc tả AGL gắn với lớp đó.* □

3.3.4 Ngôn ngữ hành vi miền dựa trên mô-đun

Phần này cung cấp một tổng quan ngắn gọn về AGL như một aDSL dùng để tích hợp các hành vi miền vào DM. Ngôn ngữ này cho phép tạo các đồ thị hoạt động bằng cách cấu hình chúng trực tiếp trên DM thông qua các chú thích. Theo cách tiếp cận siêu mô hình hóa cho DSLs [66], một siêu mô hình cú pháp trừu tượng (*Abstract Syntax Meta-Model - ASM*) và một siêu mô hình cú pháp cụ thể dạng văn bản dựa trên chú thích (*Concrete Syntax*

Meta-Model - CSM) cho AGL được định nghĩa. Chỉ các khía cạnh cú pháp của AGL được trình bày ở đây, vì ngữ nghĩa hành vi của ngôn ngữ được mô tả trong các Mục 3.3.2 và 3.3.3.

3.3.4.1 Cú pháp trừu tượng

Thực hiện định nghĩa các yêu cầu miền của AGL bằng cách áp dụng các điều khoản bao gồm (I), loại trừ (X), và ràng buộc (R) lên các yêu cầu của biểu đồ hoạt động trong UML như được mô tả chi tiết trong [91, p. 373]. Cụ thể, các điều khoản sau được áp dụng:

- (I1) một hành động của mô-đun, như đã thảo luận trong Mục 3.3.2, là một dạng đặc biệt của hành động [91, p. 441];
- (R1) mỗi nút thực thi [91, p. 403] thực hiện một chuỗi các hành động của mô-đun;
- (X1) không sử dụng biến trong hoạt động [91, p. 377];
- (X2) không sử dụng các hành động thay đổi [91, p. 469].

I1 và R1 là cần thiết để tích hợp đồ thị hoạt động vào MOSA. X1 và X2 loại bỏ việc sử dụng biến, vốn là một lựa chọn thay thế cho luồng đối tượng—phương tiện chính để truyền dữ liệu trong các hoạt động của UML [91, p. 377]. Mô hình hoạt động của AGL nắm bắt trạng thái hiện tại của hệ thống bằng cách tham chiếu trực tiếp đến mô hình lớp hợp nhất, thay vì sử dụng luồng đối tượng.

Hình 3.8 mô tả một mô hình siêu đơn giản hóa cho cú pháp trừu tượng của AGL. Cụ thể, siêu mô hình của AGL bao gồm các siêu khái niệm được điều chỉnh từ siêu mô hình UML dành cho mô hình đồ thị hoạt động: `ActivityGraph`, `Node`, `ControlNode`, và `Edge`. Một số ràng buộc được định nghĩa trên `Node` để hình thành miền đồ thị hoạt động của AGL, vốn hẹp hơn (như một tập con của) miền đồ thị hoạt động của UML.

Siêu khái niệm `Node` biểu diễn các nút hành động và có bốn thuộc tính. Thuộc tính thứ nhất là `label`, biểu diễn nhãn của nút. Thuộc tính thứ hai là `out`, được dẫn xuất từ liên kết `hasSrc(Edge, Node)` và ghi nhận các cạnh đi ra từ một `Node`. Hai thuộc tính tiếp theo đặc tả mô-đun được tham chiếu và

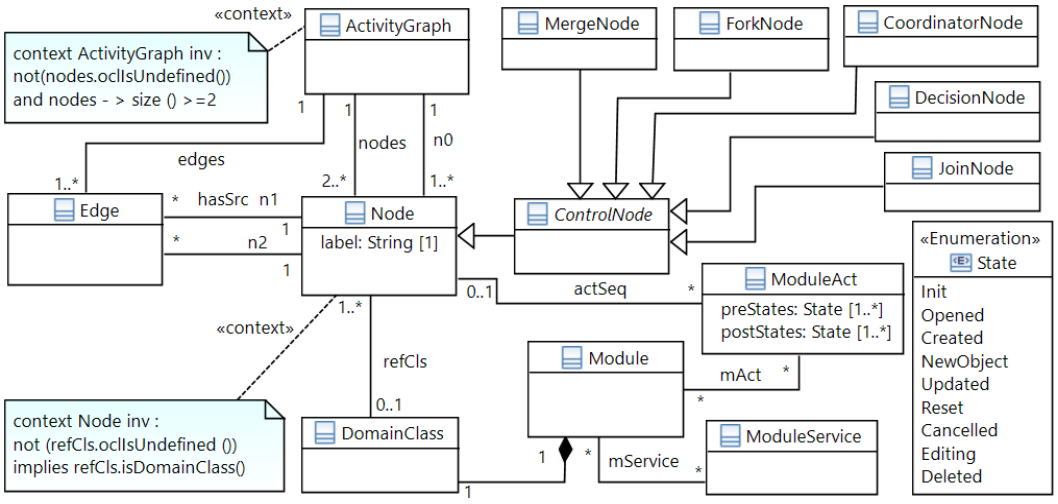
gắn với nút này. Thuộc tính `refCls` tham chiếu đến lớp miền của mô-đun được tham chiếu, trong khi thuộc tính `serviceCls` đặc tả lớp `ModuleService` thực tế của mô-đun. Thuộc tính `serviceCls` tương ứng với vai trò đích của liên kết từ `Node` đến `ModuleMAct` và cho phép `Node` hiện tại thực thi các `ModuleMAct` được chỉ định bởi thuộc tính `actSeq`.

Siêu khái niệm `CoordinatorNode` được dùng để biểu diễn một nhóm tác vụ và không truyền dữ liệu của nó ra các cạnh đi ra. Thay vào đó, nó truyền dữ liệu đầu vào ban đầu ra ngoài, và các tác vụ thành viên tương tác với nhau để thực hiện lô-gic của nhóm. Giao diện người dùng của điều phối đóng vai trò là vùng chứa giao diện người dùng của các tác vụ thành viên, cho phép người dùng quan sát tổng thể nhóm.

Siêu khái niệm `ModuleMAct` biểu diễn các hành động mô-đun kiểu SAA. Lưu ý rằng việc sử dụng một kiểu liệt kê gọi là `ActName` và một kiểu liệt kê gọi là `State` để biểu diễn tên hành động và hợp của các tiền và hậu trạng thái (tương ứng). Đặc biệt, `State` biểu diễn cả các trạng thái bình thường và trạng thái đồng thời (xem Mục 3.3.2.1).

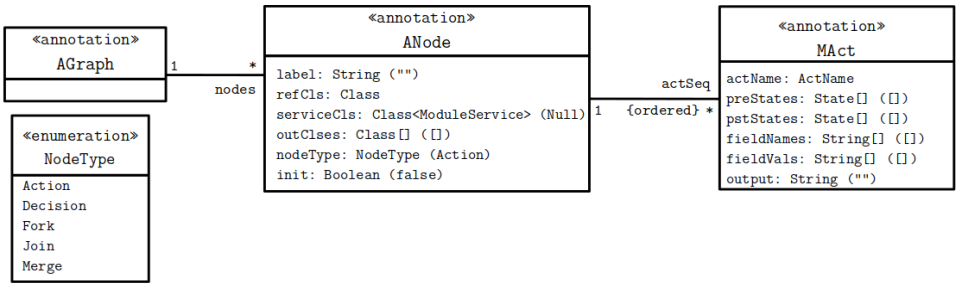
3.3.4.2 Cú pháp cụ thể dạng văn bản dựa trên chú thích

Cú pháp cụ thể của AGL được đề xuất theo cách tiếp cận xây dựng một siêu mô hình cú pháp cụ thể thông qua việc biến đổi từ siêu mô hình cú



Hình 3.8: Một siêu mô hình giản lược cho cú pháp trừu tượng của AGL.

pháp trừu tượng. Hình 3.9 minh họa siêu mô hình CSM dựa trên chú thích, phù hợp để nhúng trực tiếp vào OOPL.



Hình 3.9: Cú pháp văn bản dựa trên chú thích của AGL, được hiện thực bằng Java.

Ngoài ra, nghiên cứu hướng đến việc biểu diễn một CSM súc tích, sử dụng một tập nhỏ các chú thích. Từ góc độ thực tiễn, một mô hình như vậy là mong muốn vì nó tạo ra cú pháp cụ thể gọn nhẹ, đòi hỏi ít nỗ lực hơn từ ngôn ngữ lập trình được sử dụng để xây dựng DM hợp nhất.

Để thực hiện việc chuyển đổi từ siêu mô hình ASM sang một siêu mô hình phù hợp cho biểu diễn dựa trên chú thích, quá trình được thực hiện qua hai bước. Ở bước thứ nhất, ASM được chuyển đổi thành siêu mô hình trung gian CSM_T, là một mô hình gọn nhẹ và thích hợp cho biểu diễn dựa trên chú thích. CSM_T bao gồm ba siêu khái niệm chính: đồ thị hoạt động (`AGraph`), nút hành động (`ANode`) và hành động mô-đun (`MAct`).

Ở bước thứ hai, CSM_T được chuyển đổi thành siêu mô hình cú pháp cụ thể dựa trên chú thích, được *"nhúng"* trực tiếp vào OOPL. Siêu mô hình này được xây dựng dựa trên ba siêu khái niệm cốt lõi của OOPL: `class`, `annotation` và `property`. Do được nhúng trực tiếp vào OOPL, cấu trúc của CSM xác định cấu trúc lõi của cú pháp văn bản AGL.

Ví dụ. Phần phía dưới bên phải của Hình 3.7 minh họa mô hình lớp hợp nhất của hoạt động quản lý đăng ký học trong hệ thống CourseMan. Đặc tả AGL cho hoạt động này được biểu diễn bởi một phần tử `AGraph`, được viết bên trong một hộp ghi chú và gắn với lớp hoạt động `EnrolmentMgmt` trong mô hình lớp hợp nhất.

3.4 Tổng kết chương

Trong chương này, luận án đã đề xuất các kỹ thuật tích hợp hành vi và ràng buộc OCL vào DM nhằm mở rộng mô hình DCSL, cho phép biểu diễn DM đầy đủ thông tin bao gồm cấu trúc, hành vi và các ràng buộc nghiệp vụ phức tạp, qua đó nâng cao khả năng biểu đạt theo phương pháp DDD. Các kỹ thuật được đề xuất dựa trên aDSL để xây dựng DM hợp nhất, tạo nền tảng cho việc thiết kế và phát triển phần mềm trong các bối cảnh ứng dụng thực tế.

Kỹ thuật tích hợp hành vi vào DM được giới thiệu lần đầu trong bài báo đăng trên tạp chí *Information and Software Technology [V1]*. Ngôn ngữ AGL được xây dựng nhằm biểu diễn các hành vi miền thông qua hệ thống các mẫu điều khiển tương ứng với các cấu trúc cốt lõi của biểu đồ hoạt động UML. AGL cho phép gắn trực tiếp hành vi vào DM thông qua cơ chế aDSL và được hiện thực trong khung làm việc JDA, qua đó đảm bảo cả tính biểu đạt và khả năng thực thi của DM trong quá trình sinh phần mềm.

Kỹ thuật tích hợp ràng buộc OCL vào DM được giới thiệu lần đầu trong bài báo đăng trên tạp chí *e-Informatica Software Engineering Journal [V6]*. Phương pháp CAP mở rộng năng lực biểu diễn của DCSL bằng cách cung cấp các mẫu ràng buộc OCL được tham số hóa, cho phép khái quát hóa và tái sử dụng các ràng buộc miền có cấu trúc tương tự. Các CAP được ánh xạ một–một sang các bất biến OCL và được tích hợp trực tiếp vào DM hợp nhất, hỗ trợ sinh bản mẫu phần mềm có khả năng kiểm tra tính đúng đắn trên dữ liệu thực, qua đó bảo đảm cả tính biểu đạt và tính khả thi trong thiết kế phần mềm theo DDD.

Chương 4

PHƯƠNG PHÁP TÍCH HỢP CÁC MỐI QUAN TÂM VÀO MÔ HÌNH MIỀN

Trong chương này, luận án trình bày phương pháp tích hợp các mối quan tâm vào mô hình miền có khả năng thực thi theo tiếp cận thiết kế hướng miền, hình thành một mô hình hợp nhất. Phương pháp nhằm giải quyết bài toán phân mảnh mô hình khi các mối quan tâm (cấu trúc, hành vi, bảo mật) được đặc tả bằng nhiều DSL chuyên biệt theo mối quan tâm, đồng thời bảo đảm khả năng thực thi và kiểm chứng của mô hình miền. Luận án đề xuất hai cơ chế tích hợp: (i) tích hợp dựa trên siêu mô hình và (ii) tích hợp dựa trên cây cú pháp trừu tượng. Trên cơ sở đó, mô hình miền hợp nhất được đưa vào khung kiểm chứng ngữ nghĩa hình thức, nâng mô hình từ đặc tả mô tả lên một tạo tác có thể thực thi và được bảo đảm đúng đắn bằng lập luận toán học.

4.1 Giới thiệu

Thiết kế hướng miền (DDD) [39] nhấn mạnh việc đặt mô hình miền vào vị trí trung tâm của quá trình phát triển phần mềm, với kỳ vọng rằng các mô hình này có thể phản ánh chính xác tri thức miền và định hướng cho việc triển khai hệ thống. Trong thực tế, các mô hình miền hiếm khi chỉ tập trung vào một khía cạnh duy nhất. Thay vào đó, chúng thường bao hàm nhiều

mối quan tâm khác nhau, bao gồm khía cạnh cấu trúc, lô-gic hành vi, cũng như các mối quan tâm xuyên suốt như bảo mật [97, 100].

Trong bối cảnh đó, DM không chỉ là một đặc tả cấu trúc, mà còn là một tạo tác ngữ nghĩa nắm bắt các ràng buộc và quy tắc chi phối các trạng thái và hành vi hợp lệ của hệ thống. Khi DM đồng thời bao hàm nhiều mối quan tâm, thách thức đặt ra là xây dựng các DM vừa có khả năng thực thi vừa nhất quán về mặt ngữ nghĩa. Mặc dù các mối quan tâm cấu trúc và hành vi thường được mô hình hóa tách rời, ngữ nghĩa thực thi của chúng vốn phụ thuộc lẫn nhau; sự hiện diện của các mối quan tâm bổ sung như kiểm soát truy cập càng ràng buộc khi nào và bằng cách nào hành vi miền được phép thực thi. Khi thiếu một nền tảng ngữ nghĩa hợp nhất, việc tích hợp nhiều mối quan tâm có nguy cơ tạo ra bất nhất, mơ hồ hoặc các hành vi không mong muốn tại thời gian chạy.

Các biểu diễn truyền thống của DM, chẳng hạn sơ đồ lớp UML kết hợp với OCL [91], cho phép đặc tả cấu trúc và các ràng buộc tĩnh nhưng không cung cấp một ngữ nghĩa chuyển trạng thái tường minh để diễn giải sự tương tác giữa các mối quan tâm khác nhau. Tương tự, các DSL—đặc biệt là DSL ngoại sinh—có thể nâng cao khả năng biểu đạt ở mức cú pháp, song vẫn thiếu một nền tảng ngữ nghĩa thống nhất cho phép phân tích hành vi và kiểm chứng khi nhiều mối quan tâm cùng tồn tại trong mô hình.

Một số hướng nghiên cứu gần đây [43, 132] đề xuất sử dụng DSL nội sinh và các aDSL nhúng trong OOPL [13, 31, 70, 95, 119] nhằm xây dựng các DM có khả năng thực thi. Mặc dù các tiếp cận này thu hẹp khoảng cách giữa mô hình và hiện thực, ngữ nghĩa thực thi chủ yếu vẫn được suy diễn từ ngôn ngữ lập trình chủ, dẫn đến khó khăn trong việc diễn giải và hợp thành ngữ nghĩa khi nhiều mối quan tâm được tích hợp đồng thời thông qua cơ chế chú thích.

Bài toán tích hợp nhiều mối quan tâm đã được thảo luận trong nhiều công trình về kết hợp DSL và siêu mô hình [89, 99, 120, 126, 128]. Tuy nhiên, các tiếp cận này chưa cung cấp được một nền tảng ngữ nghĩa thống nhất cho phép biểu diễn mô hình miền, và cũng chưa xác lập một nền tảng ngữ nghĩa hình thức cho các mối quan tâm khác nhau tương tác trong quá trình thực thi. Đặc biệt, ngữ nghĩa của hành vi miền dưới tác động của các ràng buộc bổ sung—chẳng hạn như các chính sách bảo mật hoặc phân

quyền—vẫn còn mang tính ngầm định và phụ thuộc vào công cụ.

Trong các hệ thống hướng miền, các cơ chế RBAC [9, 58] không chỉ giới hạn quyền truy cập mà còn ràng buộc trực tiếp các chuyển trạng thái hợp lệ của DM; do đó, bảo mật trở thành một phần của ngữ nghĩa hành vi. Tuy nhiên, các phương pháp kiểm chứng hiện nay chủ yếu dựa trên kiểm thử hoặc cưỡng chế tại thời gian chạy, chưa cung cấp được các đảm bảo hình thức về tính đúng đắn của DM tích hợp bảo mật.

Mặc dù các kỹ thuật hình thức như Alloy, Petri nets hay ngữ nghĩa hoạt động dựa vào cấu trúc [65, 109, 113] cho phép kiểm chứng mạnh, chúng hiếm khi được tích hợp một cách có hệ thống vào quy trình mô hình hóa theo DDD hoặc DSL, đặc biệt là trong bối cảnh tích hợp mối quan tâm dựa trên chú thích.

Xuất phát từ các hạn chế trên, luận án đề xuất phương pháp tích hợp các mối quan tâm vào mô hình miền thông qua ngôn ngữ UDML (*Unified Domain Model Language*). Thực hiện định nghĩa một nền tảng ngữ nghĩa hình thức thống nhất cho UDML, cho phép tích hợp một cách có hệ thống các mối quan tâm miền dựa trên DSL dạng chú thích. Event-B [2, 3, 80, 98] được lựa chọn làm nền tảng ngữ nghĩa cho UDML, cung cấp một khung ngữ nghĩa cho các mô hình miền hợp nhất trong UDML. Các khía cạnh hành vi được đặc tả bằng AGL (trình bày trong Mục 3.3) xác định các chuyển trạng thái hợp lệ của miền và đóng vai trò là trụ cột ngữ nghĩa của mô hình miền hợp nhất. Trên cơ sở đó, luận án định nghĩa RBACDom, một mối quan tâm bảo mật được biểu diễn dưới dạng DSL dựa trên chú thích cho mô hình phân quyền theo vai trò (RBAC), trong đó các quy tắc ủy quyền được diễn giải như các điều kiện bảo vệ ràng buộc các chuyển trạng thái đã được xác định về mặt hành vi. Như vậy, các mối quan tâm về cấu trúc, hành vi và bảo mật—tương ứng được đặc tả bởi DCSL, AGL và RBACDom—được đồng bộ hóa trên một trạng thái hệ thống mối quan tâm chung, tạo thành một hệ chuyển trạng thái hợp nhất.

Luận án biểu diễn UDML theo hai phương pháp: phương pháp siêu mô hình, trong đó mỗi mối quan tâm được đặc tả bằng một DSL tương ứng và được hợp nhất thông qua cơ chế chú thích; và phương pháp cây cú pháp trừu tượng (AST), nhằm xây dựng một DM hợp nhất có khả năng thực thi. Cụ thể, chương giới thiệu (i) một khung ngữ nghĩa cho phép giải thích

ngữ nghĩa thao tác của mô hình miền hợp nhất; (ii) một DSL mang tên RBACDom để biểu diễn phân quyền theo vai trò như một mối quan tâm được tích hợp vào mô hình miền hợp nhất; và (iii) một phép chuyển đổi từ các mô hình UDML sang Event-B, cho phép kiểm chứng dựa trên chứng minh hình thức bằng nền tảng Rodin [2, 3, 80, 98], nhằm phục vụ cho mục đích mô phỏng và kiểm chứng các mô hình miền hợp nhất.

Các mục còn lại của chương này được cấu trúc như sau. Mục 4.2 trình bày phương pháp tích hợp các mối quan tâm vào UDML dựa trên siêu mô hình và khung ngữ nghĩa và hỗ trợ công cụ cho các mô hình miền hợp nhất có thể thực thi trong UDML. Phương pháp tích hợp các mối quan tâm vào UDML dựa trên AST được trình bày trong Mục 4.3. Cuối cùng, tổng kết các kết quả đạt được được trình bày trong Mục 4.5 của luận án.

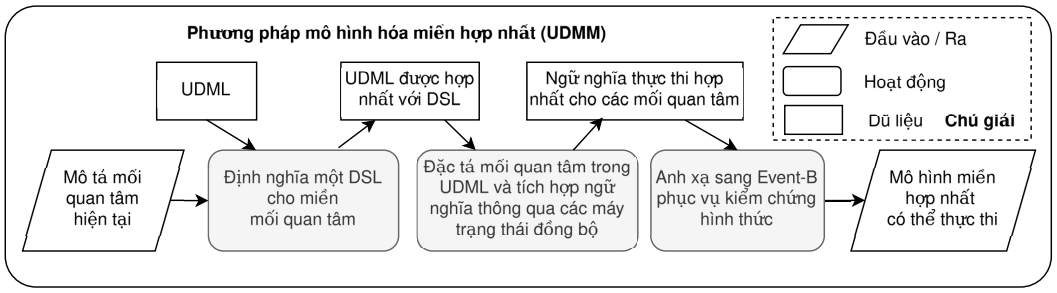
4.2 Phương pháp biểu diễn mô hình miền hợp nhất dựa vào siêu mô hình

Mục này, luận án giới thiệu một khung ngữ nghĩa cho mô hình miền, trong đó nhiều mối quan tâm khác nhau, bao gồm mối quan tâm về cấu trúc, hành vi và bảo mật, có thể được tích hợp.

4.2.1 Tổng quan về phương pháp đề xuất

Hình 4.1 minh họa phương pháp được đề xuất, mang tên UDMM (Unified Domain Modeling Method), được tổ chức như một quy trình gồm ba bước. Phương pháp này nhận đầu vào là các đặc tả của các miền mối quan tâm và tạo ra đầu ra là các mô hình miền hợp nhất có khả năng thực thi và đã được kiểm chứng, được biểu diễn bằng UDML.

Bước 1: Siêu mô hình hóa và hợp thành mối quan tâm. Với mỗi mối quan tâm, nhà thiết kế định nghĩa một DSL chuyên biệt theo mối quan tâm (*concern-specific DSL* – DSL_c), để biểu diễn miền của mối quan tâm đó. Quá trình này bao gồm việc định nghĩa các siêu khái niệm trong miền dựa trên mô tả của nó, từ đó tạo ra một siêu mô hình cho DSL_c sau đó được hợp nhất vào UDML (*Unified Domain Modeling Language*) thông qua cơ chế chú thích.



Hình 4.1: Tổng quan phương pháp đề xuất nhằm mô hình hóa miền hợp nhất.

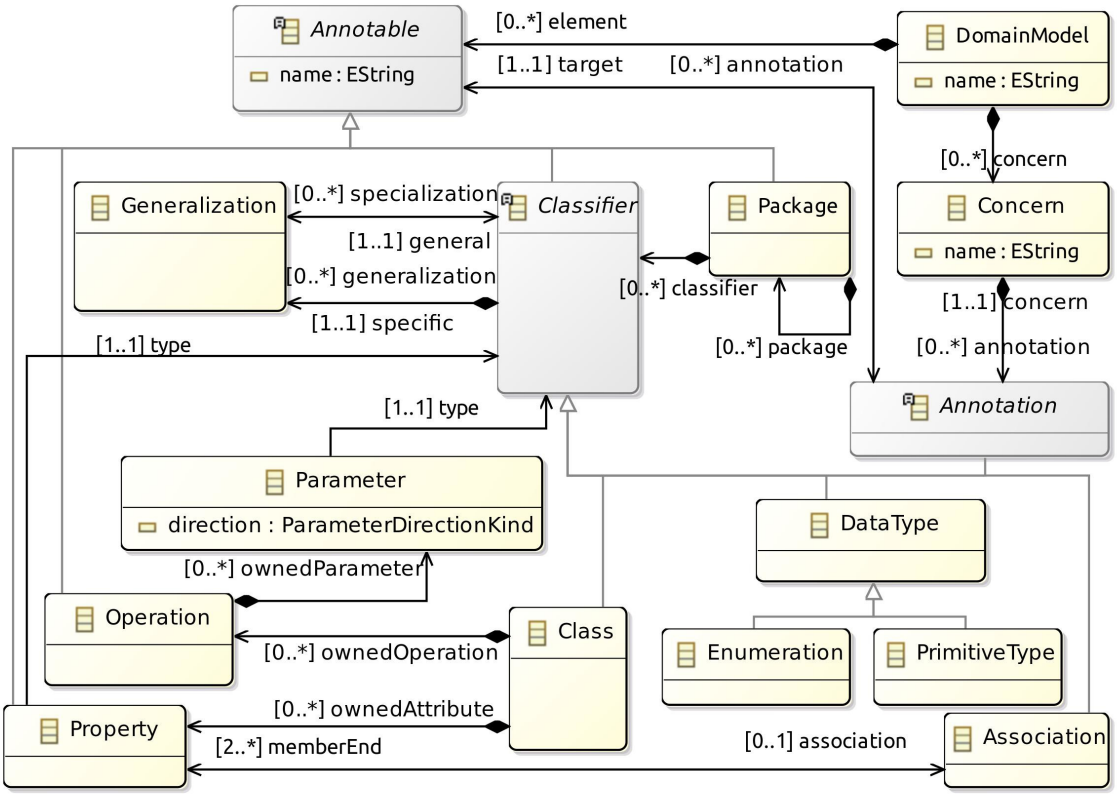
Bước 2: Định nghĩa ngữ nghĩa thông qua chuyển đổi hình thức. Bước này xác lập ngữ nghĩa hình thức cho mô hình miền hợp nhất. Một phép chuyển đổi được định nghĩa từ mô hình UDML được làm giàu bởi các DSL_c (bao gồm DCSL, AGL và RBACDom) sang Event-B, qua đó cung cấp một diễn giải toán học chính xác cho mô hình hợp nhất.

Trong phép chuyển đổi này, ngữ nghĩa hình thức của từng DSL_c được xác định dựa trên mô hình hệ chuyển trạng thái. Các đặc tả hành vi được biểu diễn bằng AGL quyết định các chuyển trạng thái hợp lệ, trong khi các ràng buộc cấu trúc từ DCSL và các ràng buộc ủy quyền từ RBACDom được diễn giải dưới dạng các bất biến và điều kiện bảo vệ nhằm giới hạn các chuyển trạng thái đó.

Bước 3: Kiểm chứng hình thức. Bước này thực hiện kiểm chứng hình thức bằng nền tảng Rodin [3]. Rodin tự động sinh ra các nghĩa vụ chứng minh tương ứng với ngữ nghĩa thực thi hợp nhất được làm giàu bởi các mối quan tâm đã được hợp thành. Việc giải quyết thành công các nghĩa vụ chứng minh này xác lập tính đúng đắn của mô hình miền hợp thành, bao gồm tính nhất quán xuyên mối quan tâm và việc bảo toàn các thuộc tính an toàn và bảo mật quan trọng.

Luận án đã định nghĩa siêu mô hình của UDML lõi dựa trên các siêu khái niệm của UML [91], như được minh họa trong Hình 4.2. Bốn siêu khái niệm mới—`DomainModel`, `Concern`, `Annotation` và `Annotable`—được bổ sung nhằm hỗ trợ việc hợp thành các mối quan tâm với DM và hợp nhất DSL_c trong UDML.

Cụ thể, một `DomainModel` bao gồm các phần tử `Annotable` và các `Concern` là các thể hiện của siêu lớp `Concern`. Một `Concern` bao gồm các `Annotation`.



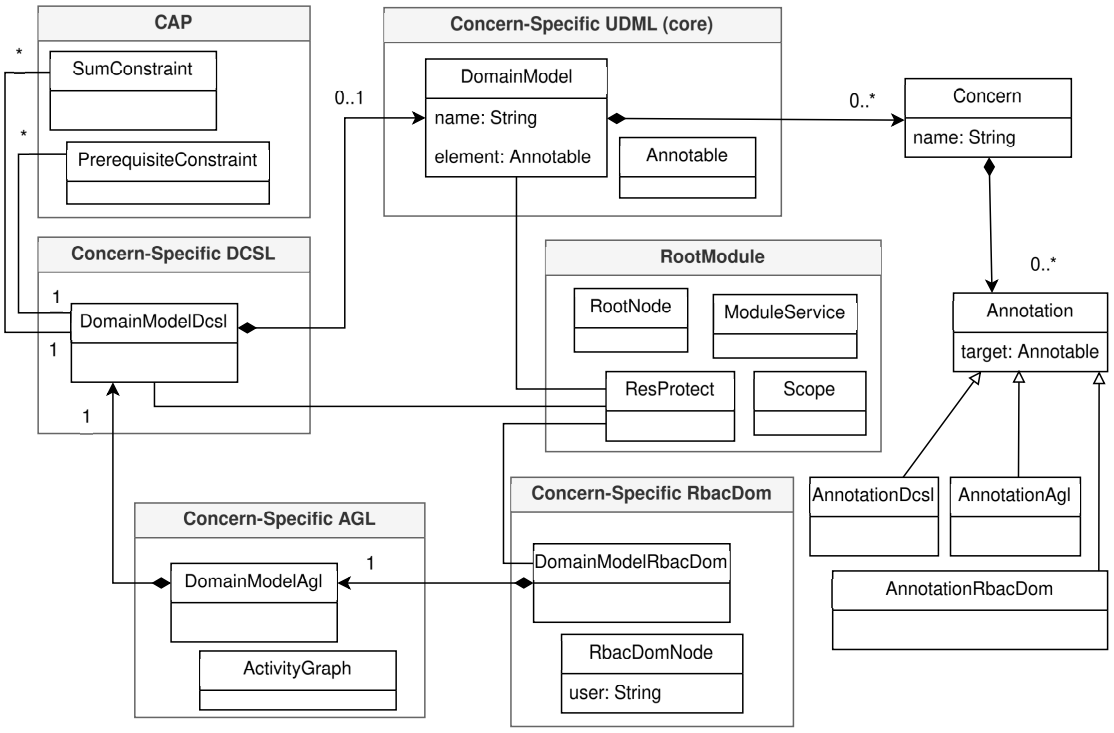
Hình 4.2: Siêu mô hình UDML lõi cho mô hình miền hợp nhất.

Mỗi phần tử `Annotable` là một thể hiện của một siêu lớp trong siêu mô hình UML dành cho biểu đồ lớp, bao gồm `Package`, `Class`, `Property`, `Association` và `Operation`.

4.2.2 Biểu diễn mô hình miền hợp nhất

Hình 4.3 trình bày siêu mô hình rút gọn của UDML dùng để biểu diễn các mô hình miền hợp nhất. Siêu mô hình này được thiết kế có chủ đích nhằm hỗ trợ một cách diễn giải chính xác và được xác định rõ ràng cho các mô hình miền hợp nhất, trong đó các mối quan tâm về cấu trúc, hành vi và bảo mật có thể được tích hợp.

Siêu mô hình rút gọn của UDML cung cấp một lõi tối giản nhưng giàu khả năng biểu đạt của siêu khái niệm của UDML lõi đóng vai trò như một lớp hợp thành trung gian cho nhiều mối quan tâm khác. Cách biểu diễn này hỗ trợ tường minh việc hợp thành có hệ thống các mối quan tâm miền vào một mô hình miền hợp nhất.



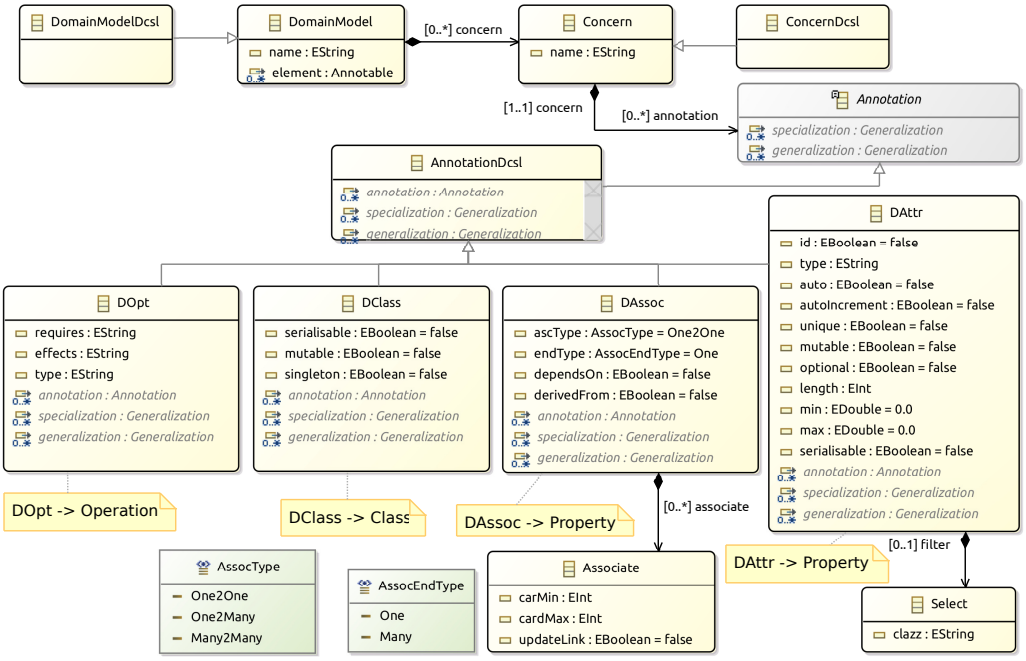
Hình 4.3: Siêu mô hình rút gọn của UDML.

Trong cách tiếp cận hợp thành dựa trên ngôn ngữ này, mỗi DSL_c, chẳng hạn như DCSL [70] cho các ràng buộc cấu trúc, hay AGL trình bày trong Mục 3.3 cho mô hình hóa hành vi, đều được định nghĩa bởi siêu mô hình riêng của nó và vẫn giữ tính độc lập về mặt khái niệm. UDML áp dụng cách tiếp cận hợp thành dẫn dắt bởi siêu mô hình, trong đó các mối quan tâm được tích hợp thông qua các tương ứng tường minh, được trung gian hóa bởi lõi UDML. Cách tiếp cận này làm cho các tương tác nhiều mối quan tâm trở nên tường minh, có thể phân tích và phù hợp cho việc diễn giải ngữ nghĩa hình thức. Cụ thể, các mô hình miền chuyên biệt theo mối quan tâm (ví dụ `DomainModelDcsl`, `DomainModelAgl` và `DomainModelRbacDom`) được liên kết với một thực thể `DomainModel` chung, qua đó bảo đảm một ngữ cảnh mô hình hóa thống nhất.

4.2.2.1 Tích hợp các ràng buộc cấu trúc

Để biểu diễn các ràng buộc cấu trúc và tích hợp chúng vào mô hình miền hợp nhất, luận án dựa trên siêu mô hình của DCSL trong [70], để xác định ngữ nghĩa hình thức của ngôn ngữ này nhằm cho phép việc tích hợp DCSL vào mô hình miền hợp nhất.

Cú pháp trừu tượng. Hình 4.4 trình bày siêu mô hình của DCSL, được giới thiệu trong công trình [70], đồng thời minh họa mối quan hệ ánh xạ giữa DCSL và UDML.



Hình 4.4: Siêu mô hình DCSL cho mô hình miền hợp nhất.

Siêu mô hình DCSL giới thiệu các siêu khái niệm lõi `DomainModelDcsl`, `ConcernDcsl` và `AnnotationDcsl`, cho phép các ràng buộc cấu trúc được đặc tả và hợp thành cùng với lõi UDML. Các ràng buộc cấu trúc cụ thể được biểu diễn thông qua các kiểu chú thích `DClass`, `DAttr`, `DAssoc` và `DOpt`, tương ứng được ánh xạ tới các khái niệm `Class`, `Property`, `Property` và `Operation`. Để bảo đảm tính đúng đắn của các ánh xạ này, các quy tắc hợp lệ hình thức được đặc tả dưới dạng các ràng buộc OCL gắn với siêu mô hình DCSL:

```
1     context DClass inv mapToCls:
2     self.target.oclIsKindOf(Class)
```

Các khái niệm khác, cùng với thuộc tính của tất cả các khái niệm được định nghĩa để biểu diễn các ràng buộc OCL thiết yếu, được tóm tắt trong Bảng 2.1.

Ngữ nghĩa. DCSL được xem như một DSL chuyên biệt theo mối quan tâm, dựa trên chú thích, nhằm nắm bắt khía cạnh cấu trúc của mô hình miền. Thay vì giới thiệu một siêu mô hình cấu trúc độc lập, DCSL đóng góp một

tập các chú thích cấu trúc cùng với các quy tắc hợp lệ hình thức, được gắn kết trực tiếp với các phần tử lõi của UDML.

Thực hiện xác định ngữ nghĩa hình thức cho DCSL như sau.

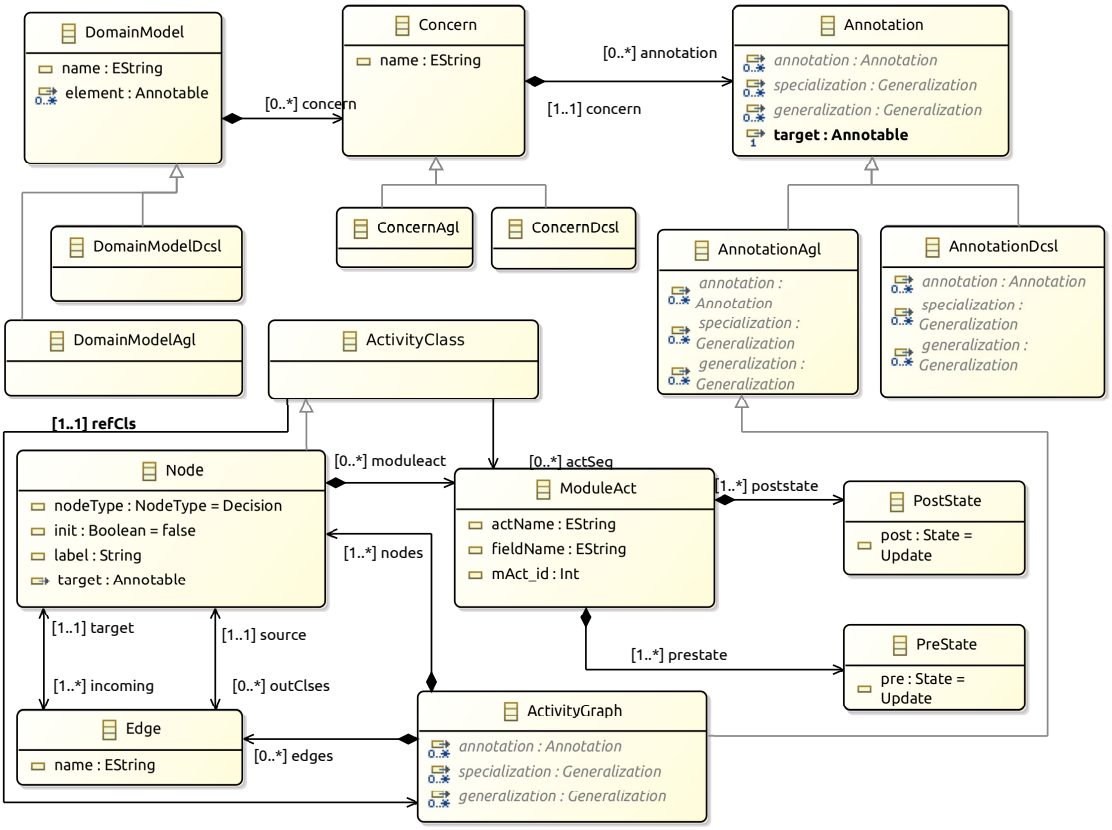
- *Liên kết từ chú thích đến UDML lõi:* Giả sử các tập mang được cho như sau.
  $AN_{dcsl} \subseteq AN$; DCL, DAT, DAS, $DOP \subseteq AN_{dcsl}$, trong đó DCL biểu thị các thể hiện của `DClass`; DAT biểu thị các thể hiện của `DAttr`; DAS biểu thị các thể hiện của `DAssoc`; DOP biểu thị các thể hiện của `DOpt`.
  Các chú thích của mối quan tâm về cấu trúc được liên kết với các phần tử lõi của UDML thông qua hàm UDML $target : AN \to AE$, với các ràng buộc kiểu sau đây:
  (D1) $\forall\, a \in DCL \cdot target(a) \in Class$;
  (D2) $\forall\, a \in DAT \cdot target(a) \in Property$;
  (D3) $\forall\, a \in DAS \cdot target(a) \in Property$;
  (D4) $\forall\, a \in DOP \cdot target(a) \in Operation$.
  Các ràng buộc này hình thức hóa ánh xạ dự định giữa các siêu khái niệm của DCSL và các siêu khái niệm lõi của UDML, đảm bảo rằng thông tin về mối quan tâm cấu trúc được gắn vào đúng các phần tử cấu trúc tương ứng.

- *Các ràng buộc tính đúng đắn của cấu trúc:* Các ràng buộc tính đúng đắn cấu trúc được định nghĩa nhằm đảm bảo tính nhất quán của miền cấu trúc.
  Gọi R_{dcsl} là tập các ràng buộc cấu trúc do DCSL đóng góp (ví dụ: các ràng buộc không gian trạng thái trên các trường miền và các đầu mút liên kết).
  Mô hình mối quan tâm cấu trúc được xem là đúng đắn khi và chỉ khi tất cả các ràng buộc trong R_{dcsl} đều thỏa mãn đối với các mục tiêu tương ứng trong mô hình UDML: $\bigwedge_{\varphi \in R_{dcsl}} \varphi$.
  Các ràng buộc này tạo thành các điều kiện đúng đắn về mặt cấu trúc, những điều kiện phải được bảo toàn trong quá trình thực thi của mô hình hợp nhất.

4.2.2.2 Tích hợp hành vi miền

Để biểu diễn các hành vi miền và tích hợp chúng vào mô hình miền hợp nhất, phần này tiếp tục tinh chỉnh AGL bằng cách xây dựng siêu mô hình của nó và định nghĩa ngữ nghĩa hình thức tương ứng.

Cú pháp trừu tượng. Hình 4.5 minh họa siêu mô hình AGL, mô hình này nắm bắt hành vi miền có khả năng thực thi bằng cách đặc tả việc thực thi hành vi dưới dạng các đồ thị hoạt động, các nút và các quan hệ luồng điều khiển. Định nghĩa siêu mô hình AGL được trình bày trong Mục 3.3 trong bối cảnh các mô hình miền hợp nhất có khả năng thực thi; một định nghĩa tường minh như vậy là cần thiết nhằm hỗ trợ việc tích hợp ngữ nghĩa một cách chính xác với các mối quan tâm về cấu trúc và bảo mật.



Hình 4.5: Siêu mô hình AGL dùng để nắm bắt hành vi miền có khả năng thực thi.

Các siêu khái niệm lõi của AGL bao gồm: `ActivityGraph` biểu diễn một đơn vị hành vi miền có khả năng thực thi và tổ chức hành vi dưới dạng một đồ thị có hướng. Mỗi `ActivityGraph` được liên kết với một `RootNode`, xác định điểm vào dùng để kích hoạt luồng hành vi tương ứng tại thời điểm

thực thi. `Node` biểu diễn một đơn vị hành vi nguyên tử và xác định một ranh giới thực thi tường minh. Mỗi nút tham chiếu đến một lớp miền thông qua thuộc tính `refCls`, qua đó liên kết việc thực thi hành vi với mô hình cấu trúc miền. `Edge` xác định các quan hệ luồng điều khiển hợp lệ giữa các nút và ràng buộc sự tiến triển có thể của quá trình thực thi. `ModuleAct` đặc tả các hành động cụ thể được thực thi tại một nút. Mỗi `ModuleAct` được liên kết với một `ModuleService` tương ứng, được định nghĩa trong `RootModule`, nhằm đảm bảo rằng các hành động hành vi được hiện thực một cách nhất quán trong kiến trúc thực thi theo mô-đun. `PreState` và `PostState` lần lượt đặc trưng cho trạng thái miền mong đợi trước và sau khi thực thi một hành động mô-đun.

Trong siêu mô hình này, mỗi `ActivityGraph` sở hữu các nút và các cạnh của nó, hình thành một ngữ cảnh thực thi hành vi có phạm vi xác định rõ ràng. Việc thực thi hành vi được khởi tạo tại `RootNode`, tiến triển theo các cạnh hợp lệ và gọi các dịch vụ mô-đun thông qua các phần tử `ModuleAct`. Bằng cách tham chiếu tới các lớp miền ở mức nút, AGL cung cấp một liên kết tường minh giữa hành vi và mô hình cấu trúc miền, tạo nền tảng cho ngữ nghĩa thực thi trong UDML.

Ngữ nghĩa. Khía cạnh hành vi của các mô hình UDML được định nghĩa trong AGL thông qua các đồ thị hoạt động được tích hợp với mô hình cấu trúc miền. Tiến hành định nghĩa ngữ nghĩa hình thức của AGL như sau.

Gọi: AG, ND, ED, MA, AC lần lượt là các tập đồ thị hoạt động, các nút hoạt động, các cạnh, các hành động mô-đun và các lớp hoạt động.

- *Đồ thị hoạt động:* Một đồ thị hoạt động $g \in AG$ được định nghĩa là một bộ:
  $g = \langle N_g,\ E_g,\ src_g,\ tgt_g,\ init_g,\ actSeq,\ refActCls \rangle$, trong đó $N_g \subseteq ND$ là tập hữu hạn các nút hoạt động; $E_g \subseteq ED$ là tập hữu hạn các cạnh; $src_g, tgt_g : E_g \to N_g$ xác định nút nguồn và nút đích của mỗi cạnh; $init_g \in N_g$ là nút khởi đầu của đồ thị hoạt động; $actSeq : ND \to \mathcal{P}(MA)$ ánh xạ mỗi nút tới một tập (có thể rỗng) các hành động mô-đun được thực thi tại nút đó; $refActCls : ND \to AC$ ánh xạ mỗi nút tới lớp hoạt động mà nó tham chiếu.

- *Các lớp hoạt động:* Các lớp hoạt động đóng vai trò là phần mở rộng hành vi của các lớp miền, cung cấp ngữ cảnh cấu trúc trong đó các đồ

thị hoạt động được diễn giải. Mỗi đồ thị hoạt động được gắn với đúng một lớp hoạt động, và các nút hoạt động tham chiếu tới lớp này khi thực thi các hành động mô-đun.

- *Các nút có thể gắn chú thích:* Một quyết định thiết kế then chốt trong UDML là các nút hoạt động là các phần tử có thể được gắn chú thích. $ND \subseteq AE$. Điều này cho phép các ngữ nghĩa đặc thù theo mối quan tâm, bao gồm các ràng buộc bảo mật được đặc tả trong RBACDom (sẽ trình bày trong phần tiếp theo), được gắn trực tiếp vào các ranh giới thực thi hành vi.

- *Các ràng buộc tính đúng đắn của AGL:* Các ràng buộc tính đúng đắn sau đây phải được thỏa mãn đối với mọi đồ thị hoạt động $g \in AG$:
  (A1) $N_g \neq \varnothing$;
  (A2) $\forall\, e \in E_g \cdot src_g(e) \in N_g \wedge tgt_g(e) \in N_g$;
  (A3) $init_g \in N_g$;
  (A4) $\forall\, n \in N_g \cdot refActCls(n)$ được xác định.

Trong Mục 3.3 trình bày một phép thực thi của mô hình AGL được định nghĩa là một chuỗi hợp lệ các hành động nguyên tử được kích hoạt, chuỗi này gây ra một dãy tương ứng các trạng thái hệ thống có thể đạt được, tuân theo luồng điều khiển của đồ thị hoạt động và tôn trọng các điều kiện trạng thái trước và sau của mỗi hành động.

Định nghĩa 10 (Thực thi của một mô hình AGL)**.** *Cho $\mathcal{G} = (N, E)$ là một đồ thị hoạt động AGL, trong đó mỗi nút $n \in N$ được gắn với một hành động nguyên tử có cấu trúc $SAA(n)$, biểu diễn một tập các chuỗi hành động nguyên tử cho phép (ASEs). Một phép thực thi của $\mathcal{G}$ là một chuỗi (hữu hạn hoặc vô hạn) các nút $\pi = (n_0, n_1, \ldots, n_k)$ sao cho: (i) với mọi $i < k$, $(n_i, n_{i+1}) \in E$; và (ii) tồn tại một dãy các chuỗi hành động nguyên tử $S_0, S_1, \ldots, S_k$, với $S_i \in SAA(n_i)$, mà các phép biến đổi trạng thái do chúng gây ra là tương thích, tức là trạng thái hậu của S_i thỏa mãn trạng thái tiền của S_{i+1}. Tương đương, một phép thực thi tương ứng với một chuỗi hành động nguyên tử hợp lệ tạo ra một dãy các trạng thái hệ thống có thể đạt được.*□

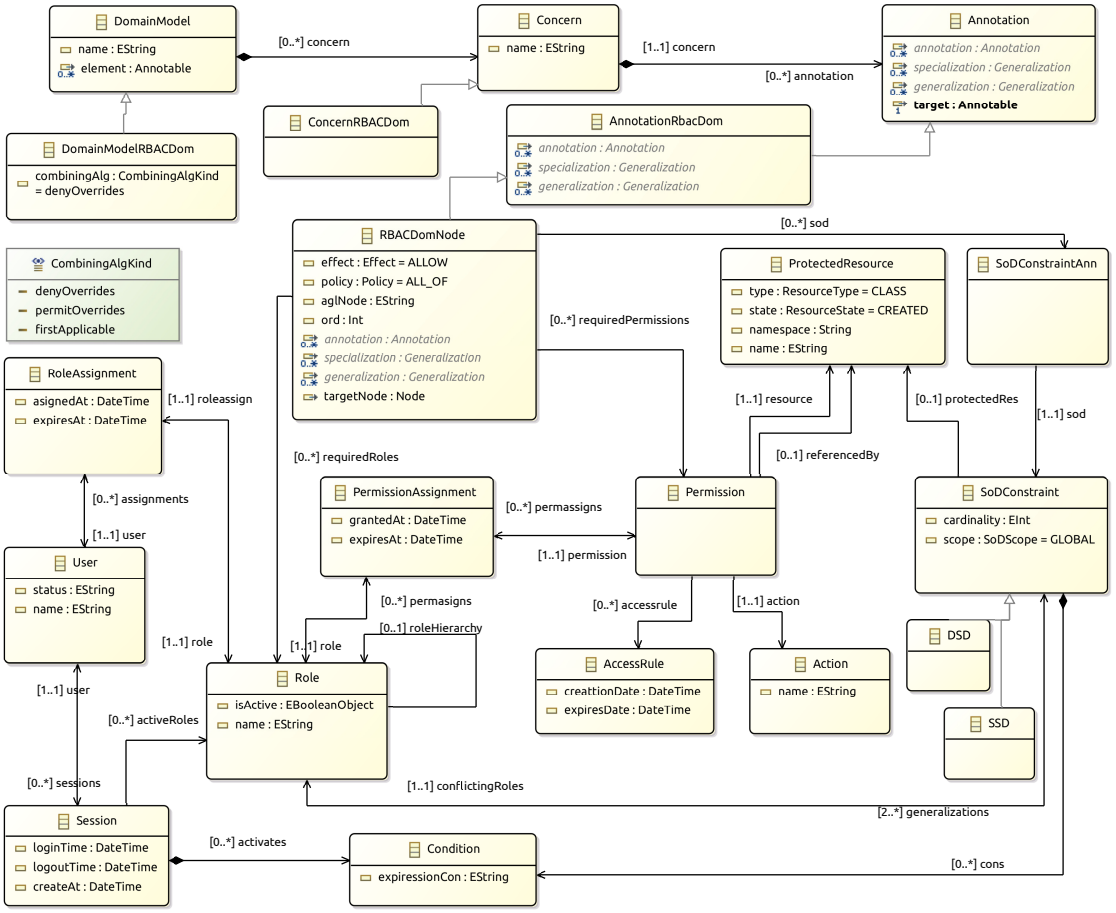
4.2.2.3 Tích hợp mối quan tâm bảo mật

Để hỗ trợ việc mô hình hóa tường minh cơ chế kiểm soát truy cập, luận án giới thiệu RBACDom như một DSL chuyên biệt theo mối quan tâm, chuyên biệt cho RBAC trong các mô hình miền hợp nhất. RBACDom được thiết kế phù hợp với các nguyên lý hướng mối quan tâm của UDML, cho phép các chính sách bảo mật được đặc tả một cách độc lập trong khi vẫn có khả năng kết hợp với các mối quan tâm về cấu trúc và hành vi. Phù hợp với định hướng thiết kế ngôn ngữ của UDML, RBACDom được định nghĩa thông qua cú pháp trừu tượng, cú pháp cụ thể và ngữ nghĩa hình thức, qua đó cho phép các ràng buộc ủy quyền được tích hợp một cách có hệ thống vào các mô hình miền hợp nhất.

Cú pháp trừu tượng. Hình 4.6 minh họa cú pháp trừu tượng của RBACDom dùng để nắm bắt mối quan tâm bảo mật. RBACDom được định nghĩa bởi một siêu mô hình chuyên biệt dựa trên các siêu khái niệm UML [91]. Siêu mô hình này nắm bắt các khái niệm lõi của RBAC và các quan hệ cấu trúc giữa chúng, đồng thời đặc tả cách thức thông tin kiểm soát truy cập được biểu diễn và liên kết với các mô hình UDML.

RBACDom được đặc tả như một DSL độc lập theo mối quan tâm, trong đó mô hình miền của nó, `DomainModelRbacDom`, được kết hợp với các mô hình miền theo mối quan tâm khác thông qua lõi UDML. Cụ thể, `DomainModelRbacDom` được liên kết với `DomainModelAgl` ở mức UDML, qua đó duy trì sự phân tách rõ ràng giữa mô hình hóa hành vi và đặc tả kiểm soát truy cập.

Siêu khái niệm `DomainModelRbacDom` định nghĩa một thuộc tính *combiningAlg* nhằm xác định chiến lược kết hợp luật gắn với mô hình RBACDom. Thuộc tính này quy định cách thức đánh giá nhiều chú thích áp dụng đồng thời trên cùng một nút hành vi (ví dụ: denyOverrides, permitOverrides, firstApplicable). Việc tích hợp ở mức nút được thực hiện thông qua siêu khái niệm `RbacDomNode`. Một thể hiện của `RbacDomNode` biểu diễn một đặc tả chính sách RBAC và được khai báo trong một `ActivityGraph` như một phần của `DomainModelRbacDom`. Mỗi `RbacDomNode` thiết lập một ánh xạ tường minh tới một nút hành vi được định nghĩa trong AGL thông qua một thuộc tính tham chiếu (ví dụ: `aglNode`), dùng để xác định nút đích theo nhãn của nó.



Hình 4.6: Siêu mô hình RBACDom nắm bắt mối quan tâm bảo mật.

Ánh xạ này mang tính cấu trúc và không làm thay đổi cú pháp trừu tượng của `AGL` :: `Node`.

Siêu khái niệm `RbacDomNode` đặc tả các thuộc tính đặc thù của RBAC, bao gồm các vai trò và/hoặc quyền yêu cầu, chế độ đánh giá chính sách (`ALL_OF` hoặc `ANY_OF`), hiệu lực (`ALLOW` hoặc `DENY`), và các ràng buộc phân tách nhiệm vụ (separation-of-duty) tùy chọn. Một vị từ phạm vi tùy chọn có thể được sử dụng để giới hạn phạm vi áp dụng của chính sách đối với một tập con các đối tượng miền, qua đó hỗ trợ kiểm soát truy cập theo ngữ cảnh ở mức mô hình.

Ngoài ra, `RbacDomNode` định nghĩa một thuộc tính tùy chọn *ord*, biểu diễn thứ tự đánh giá của các chú thích. Thuộc tính này chỉ được sử dụng khi chiến lược kết hợp là firstApplicable, trong đó các chú thích được đánh giá theo thứ tự đã gán.

Các tham số kết hợp luật có thể cấu hình. Trong các hệ thống kiểm soát truy cập thực tế, nhiều luật phân quyền có thể đồng thời áp dụng cho cùng một

thao tác được bảo vệ. Do đó, các hệ thống chính sách khác nhau sử dụng các chiến lược kết hợp luật khác nhau để xác định quyết định phân quyền cuối cùng. Để phản ánh tính linh hoạt này ở mức mô hình, RBACDom cung cấp một tham số tường minh nhằm xác định cách kết hợp các chú thích áp dụng đồng thời.

Ký hiệu $CA = \{\mathsf{denyOverrides}, \mathsf{permitOverrides}, \mathsf{firstApplicable}\}$ là tập các chiến lược kết hợp luật hợp lệ.
Thuộc tính $combiningAlg : DomainModelRbacDom \to CA$ gán một chiến lược kết hợp cho mỗi mô hình RBACDom. Nếu không được chỉ định tường minh, chiến lược mặc định là denyOverrides.

Gọi $R \in DomainModelRbacDom$ là một mô hình RBACDom. Đối với việc đánh giá phụ thuộc thứ tự theo chiến lược firstApplicable, ta xét thêm một hàm riêng phần: $ord : RbacDomNode \rightharpoonup \mathbb{N}$, trong đó $ord(rna)$ biểu diễn thứ tự đánh giá của chú thích rna.

Gọi $Ann(n) \subseteq RbacDomNode$ là tập các chú thích RBACDom gắn với nút $n \in ND$. Điều kiện well-formedness sau đảm bảo tính xác định trong việc đánh giá các chính sách cục bộ tại nút: $(RC1)$ $combiningAlg(R) = \mathsf{firstApplicable} \Rightarrow \forall\, n \in ND \cdot \forall\, rna_1, rna_2 \in Ann(n) \cdot (ord(rna_1) = ord(rna_2) \Rightarrow rna_1 = rna_2)$.

RBACDom còn định nghĩa các khái niệm RBAC chuẩn `User`, `Role`, `Permission` và `Session` một cách độc lập với các phần tử miền của UDML. Các quan hệ gán người dùng–vai trò và vai trò–quyền được biểu diễn tường minh dưới dạng các quan hệ, trong khi các quyền được mô hình hóa như các cặp ⟨hành động, tài nguyên được bảo vệ⟩. Các tài nguyên được bảo vệ có thể tùy chọn được liên kết với các phần tử có thể gắn chú thích của UDML nhằm hỗ trợ việc tích hợp với các mô hình miền.

Để biểu diễn các ràng buộc cho RBACDom, mô hình này giới thiệu siêu khái niệm trừu tượng ràng buộc phân tách nhiệm vụ `SoDConstraint`, với các chuyên biệt hóa cho phân tách nhiệm vụ tĩnh `SSD` và phân tách nhiệm vụ động `DSD`. Việc mô hình hóa tường minh các ràng buộc SoD giúp cải thiện tính mô-đun và khả năng phân tích cấu trúc của các đặc tả kiểm soát truy cập.

Siêu mô hình RBACDom được trang bị một tập các quy tắc tính đúng đắn cấu trúc, được tóm tắt trong Bảng 4.1. Các quy tắc này đảm bảo tính

nhất quán nội tại của các đặc tả RBAC và tính hợp lệ của việc gắn chúng vào các mô hình UDML trước khi tiến hành diễn giải ngữ nghĩa hoặc kiểm chứng hình thức.

Bảng 4.1: Các quy tắc hợp lệ cấu trúc của siêu mô hình RBACDom

Ràng buộc	**Mô tả**
WF-S1: Định danh RBAC duy nhất	Mọi thực thể RBAC (người dùng, vai trò, quyền, hành động và tài nguyên được bảo vệ) phải có định danh duy nhất trong một mô hình RBACDom.
WF-S2: Gán kết vai trò hợp lệ	Mỗi gán kết người dùng–vai trò phải tham chiếu đến một người dùng và một vai trò tồn tại. Không cho phép các gán kết vai trò không xác định hoặc bị treo.
WF-S3: Gán kết quyền hợp lệ	Mỗi gán kết vai trò–quyền phải tham chiếu đến một vai trò tồn tại và một quyền hợp lệ được định nghĩa trong mô hình.
WF-S4: Định nghĩa quyền hợp lệ	Mỗi quyền phải được định nghĩa chính xác bởi một hành động áp dụng lên đúng một tài nguyên được bảo vệ, theo diễn giải RBAC coi quyền như các cặp ⟨thao tác, đối tượng⟩.
WF-S5: Bộ đôi quyền duy nhất	Không có hai quyền nào được phép định nghĩa cùng một cặp hành động–tài nguyên.
WF-S6: Gắn kết tài nguyên tùy chọn	Một tài nguyên được bảo vệ có thể được gắn hoặc không gắn với một phần tử miền cụ thể của UDML. Việc không gắn được cho phép nhằm hỗ trợ gắn kết muộn.
WF-S7: Tương thích kiểu tài nguyên–miền	Nếu một tài nguyên được bảo vệ được gắn với một phần tử miền, kiểu của tài nguyên phải tương thích với loại phần tử UDML mà nó được gắn vào.
WF-S8: Ràng buộc phân tách nhiệm vụ hợp lệ	Mỗi ràng buộc SoD phải tham chiếu đến ít nhất hai vai trò khác nhau và xác định một lực lượng hợp lệ ($n \geq 2$).
WF-S9: Nhất quán phân tách nhiệm vụ tĩnh	Các ràng buộc SoD tĩnh không được bị vi phạm bởi các gán kết người dùng–vai trò.
WF-S10: Gắn kết chú thích và phạm vi	Mọi chú thích RBAC phải được gắn vào các phần tử UDML được xác định rõ ràng.

Cú pháp cụ thể. Để hỗ trợ hiện thực hóa thực tiễn và tích hợp công cụ, phần này định nghĩa một cú pháp cụ thể dạng văn bản dựa trên chú thích cho RBACDom, phù hợp với các nguyên lý kết hợp hướng mối quan tâm và điều khiển bởi siêu mô hình của UDML. Cú pháp cụ thể này được dẫn xuất từ siêu mô hình cú pháp trừu tượng của RBACDom bằng cách định nghĩa một siêu mô hình cú pháp cụ thể tương ứng, thích hợp để nhúng vào một ngôn ngữ lập trình hướng đối tượng chủ (ví dụ: Java).

Phù hợp với phong cách tích hợp dựa trên chú thích của UDML, RBACDom được hiện thực như một DSL bên ngoài dựa trên chú thích. Các chính sách RBAC được đặc tả dưới dạng văn bản thông qua các chú thích, các chú thích này khởi tạo các phần tử `RbacDomNode` của siêu mô hình RBACDom, trong khi mối liên kết ngữ nghĩa của chúng với hành vi có khả năng thực thi được thiết lập thông qua sự tương ứng tường minh ở mức nút với các nút AGL. Thiết kế này bảo toàn sự phân tách giữa các mối quan tâm về hành

vi và bảo mật, đồng thời cho phép kết hợp chúng một cách có hệ thống ở mức mô hình hóa.

Cú pháp cụ thể được xây dựng xoay quanh chú thích `@RBACDomNode`, chú thích này biểu diễn một đặc tả RBACDom ở mức nút. Mỗi thể hiện của `@RBACDomNode` tương ứng với một phần tử của cú pháp trừu tượng `RbacDomNode` và xác định các ràng buộc ủy quyền được gắn với một ranh giới thực thi hành vi. Phần thân của chú thích mã hóa các thuộc tính đặc thù của RBAC như các vai trò hoặc quyền yêu cầu, chế độ đánh giá chính sách, hiệu lực, và các ràng buộc SoD tùy chọn.

Đoạn liệt kê sau minh họa cú pháp văn bản dựa trên chú thích được sử dụng để đặc tả các chính sách RBACDom:

```
    @RBACDomNode(
    id     = "U.ID",
    roles  = {<RoleName>},
    perms  = {(<Op>, <Res>), (<...>)},
    policy = ALL_OF | ANY_OF,
    effect = ALLOW | DENY,
    scope  = {<ScopePredicate>,<...>},
    sod    = { SSD | DSD | ... },
    ord    = <Integer>
    )
```

Ký pháp này nắm bắt các yêu cầu ủy quyền ở mức hành vi có khả năng thực thi. Các yêu cầu về vai trò và quyền xác định ai được phép thực thi một nút hành vi, các toán tử chính sách xác định cách kết hợp nhiều yêu cầu, và thuộc tính hiệu lực quyết định việc cho phép hay từ chối thực thi. Các vị từ phạm vi và các ràng buộc SoD tùy chọn tiếp tục giới hạn phạm vi áp dụng mà không đưa thêm các cấu trúc hành vi mới.

Ví dụ. Ví dụ sau minh họa một đặc tả RBACDom cho vai trò **Author** trong miền OJS:

```
    @RBACDomNode(
    id     = "U1.Author",
    roles  = {"Author"},
    perms  = {
        ("CreateSubmission", "Submission"),
        ("UploadManuscriptFile","ManuscriptFile")
    },
    policy = ANY_OF,
    effect = ALLOW,
```

```
10      sod     = {DSD}1,
11      ord     = 1
12      )
```

Chú thích này được khai báo như một phần của mô hình RBACDom và được liên kết với một nút hành vi thông qua một sự tương ứng tường minh tới nhãn nút được định nghĩa trong đồ thị hoạt động AGL.

4.2.2.4 Ánh xạ giữa AGL và RBACDom

Các mối quan tâm về bảo mật được nắm bắt một cách độc lập trong RBACDom thông qua siêu khái niệm `RbacDomNode`, siêu khái niệm này đặc tả các ràng buộc ủy quyền, yêu cầu về vai trò và các thuộc tính bảo mật liên quan. Thay vì nhúng trực tiếp bảo mật vào các nút hành vi, các nút RBACDom được kết hợp với các nút AGL thông qua một quan hệ tương ứng tường minh—thường được hiện thực bằng một tham chiếu hoặc nhãn (ví dụ: `aglNode`)—nhằm xác định nút hành vi mà việc thực thi của nó bị ràng buộc.

Cách ánh xạ này bảo toàn tính mô-đun của các DSL chuyên biệt theo mối quan tâm, đồng thời căn chỉnh ngữ nghĩa của chúng trong khuôn khổ UDML. AGL xác định các chuyển tiếp hành vi hợp lệ, trong khi RBACDom ràng buộc khả năng thực thi bằng cách hạn chế việc kích hoạt các chuyển tiếp dưới các điều kiện ủy quyền. Các mối quan tâm này được hợp nhất thông qua lõi UDML trên một diễn giải hành vi chung, mà không trộn lẫn các cú pháp trừu tượng của chúng. Việc hiện thực khả thi được hỗ trợ bởi *RootModule*, thành phần này đóng gói các đặc tả DCSL, AGL và RBACDom đã được kết hợp thành một đơn vị nhất quán. Được điều phối bởi `ModuleService` và trung gian bởi `ResProtect` cùng các định nghĩa phạm vi, cấu trúc này đảm bảo việc thực thi nhất quán cả hành vi lẫn kiểm soát truy cập, từ đó tạo ra một cú pháp trừu tượng hợp nhất làm nền tảng cho ngữ nghĩa thực thi và kiểm chứng hình thức.

Đoạn trích sau minh họa cách các đặc tả RBACDom cùng tồn tại với các định nghĩa hành vi trong một ngôn ngữ chủ dựa trên chú thích. Các nút hành vi được định nghĩa bằng các chú thích của AGL, trong khi các nút RBACDom được khai báo độc lập và được liên kết với các nút hành vi thông qua các quan hệ tương ứng thay vì nhúng cú pháp trực tiếp. Khi chiến lược kết hợp là `firstApplicable`, các chú thích được đánh giá theo

thuộc tính `ord`. Trong ví dụ này, luật đầu tiên thỏa mãn sẽ quyết định kết quả cuối cùng. Nếu người dùng thỏa mãn luật thứ nhất (ví dụ: có vai trò `Guest`), quyền truy cập vào hoạt động *Submission* sẽ bị từ chối. Ngược lại, luật thứ hai sẽ được đánh giá, cho phép thực thi đối với những người dùng có vai trò `Author` và thỏa mãn các quyền yêu cầu.

```
@AGraph(nodes = {
    @ANode(label = "Submission",
    nodeType = NodeType.Action,
    init = true,
    refCls = Submission.class,
    outNodes = {"SubmissionQueue"},
    moduleact = {@MAct(
        actName = ActNames.newObject,
        poststate = {State.Created})}
    )
    @RBACDomNode(
    id = "U1.GuestDeny",
    aglNode = "Submission",
    effect  = Effect.DENY,
    roles   = {Role.Guest},
    ord     = 1
    ),
    @RBACDomNode(
    id = "U1.AuthorAllow",
    aglNode = "Submission",
    effect  = Effect.ALLOW,
    roles   = {Role.Author},
    perms   = {
        ("CreateSubmission","Submission"),
        ("UploadManuscriptFile","ManuscriptFile")
    },
    policy  = Policy.ANY_OF,
    sod     = {DSD},
    ord     = 2
    ), ...
}
)
public class SubmissionMng{
    ...
}
```

Ngữ nghĩa cho RBACDom. Các khía cạnh bảo mật được nắm bắt bởi RBACDom, một DSL chuyên biệt theo mối quan tâm dựa trên chú thích, hình thức hóa RBAC cùng với các ràng buộc SoD và liên kết chúng với các

phần tử UDML có thể gắn chú thích, đặc biệt là các nút hoạt động AGL. Thực hiện định nghĩa ngữ nghĩa hình thức cho RBACDom như sau.

Giả sử các tập mang sau đây được cho:
USR, R, P, SES, ACT, RES, SC, $RbacDomNode$ lần lượt biểu diễn người dùng, vai trò, quyền, phiên, hành động, tài nguyên được bảo vệ, các ràng buộc SoD và các chú thích RBAC ở mức nút.

- *Gán vai trò và quyền:* RBACDom nắm bắt các phép gán dưới dạng các quan hệ. $UA \subseteq USR \times R$; $PA \subseteq R \times P$, trong đó $(u, r) \in UA$ có nghĩa là người dùng u được gán vai trò r, và $(r, p) \in PA$ có nghĩa là vai trò r được cấp quyền p.

- *Phiên và các vai trò đang kích hoạt:* Các phiên được liên kết với người dùng và các vai trò đang kích hoạt bởi: $userOf : SES \to USR$; $activeRoles \subseteq SES \times R$, trong đó $(s, r) \in activeRoles$ có nghĩa là vai trò r đang hoạt động trong phiên s.

- *Quyền và tài nguyên được bảo vệ:* Mỗi quyền được đặc trưng bởi một hành động và một tài nguyên được bảo vệ. $permAction : P \to ACT$; $permRes : P \to RES$. Các tài nguyên được bảo vệ có thể được liên kết với các phần tử UDML có thể gắn chú thích nhằm hỗ trợ tham chiếu ở mức miền: $bindsTo : RES \rightharpoonup AE$.

- *Phân tách nhiệm vụ (SoD):* Một ràng buộc SoD $c \in SC$ được định nghĩa là một bộ:
  $c = \langle conflictSet(c),\ card(c),\ kind(c),\ scope(c) \rangle$,
  trong đó $conflictSet(c) \subseteq R$, $card(c) \in \mathbb{N}$, $kind(c) \in \{SSD, DSD\}$, và $scope(c)$ biểu thị phạm vi của ràng buộc (ví dụ: $GLOBAL$).

- *Các chú thích RBACDom gắn với nút:* Các chính sách RBACDom được gắn với các ranh giới thực thi hành vi thông qua các chú thích ở mức nút. Gọi $ND \subseteq AE$ là tập các nút hoạt động có thể gắn chú thích được định nghĩa bởi AGL. Mỗi chú thích RBACDom $rna \in RbacDomNode$ được gắn với đúng một nút hoạt động thông qua hàm:
  $targetNode : RbacDomNode \to ND$. Mỗi chú thích cung cấp các thành phần cú pháp trừu tượng sau:
  $policy : RbacDomNode \to \{ALL\_OF, ANY\_OF\}$;

$effect : RbacDomNode \to \{ALLOW, DENY\}$;
$ReqRoles : RbacDomNode \to \mathcal{P}(R)$;
$ReqPerms : RbacDomNode \to \mathcal{P}(P)$;
$SoD : RbacDomNode \to \mathcal{P}(SC)$.

Ánh xạ cho RBACDom. Trong UDML, RBACDom được diễn giải như một mối quan tâm về bảo mật, có vai trò ràng buộc khả năng thực thi hành vi thông qua các tương ứng ở mức nút với các nút hoạt động trong AGL. Mỗi `RbacDomNode` tạo ra một điều kiện phân quyền trên nút đích của nó, qua đó giới hạn việc kích hoạt nút mà không làm thay đổi cấu trúc hành vi hay luồng điều khiển. Do đó, một nút hoạt động chỉ có thể thực thi khi nó được kích hoạt theo AGL và đồng thời thỏa mãn tất cả các ràng buộc RBACDom liên quan trong ngữ cảnh phân quyền hiện tại.

Nói chung, nhiều chú thích RBACDom có thể được gắn với cùng một nút hành vi. Vì vậy, quyết định phân quyền cuối cùng phụ thuộc vào chiến lược kết hợp luật được lựa chọn trong mô hình RBACDom. Hàm $combiningAlg : DomainModelRbacDom \to CA$ xác định cách đánh giá các chú thích áp dụng đồng thời. Do đó, ngữ nghĩa dưới đây được tham số hóa theo chiến lược kết hợp luật này. Trong ngữ nghĩa này, AGL xác định tiến hóa hành vi, trong khi RBACDom loại bỏ các chuyển trạng thái không hợp lệ, qua đó cung cấp cơ sở hợp thành cho chuyển đổi bảo toàn ngữ nghĩa và kiểm chứng hình thức.

Định nghĩa 11 (Tương ứng nút RBACDom)**.** *Giả sử $G = (N, E)$ là một đồ thị hoạt động AGL với $N \subseteq ND$. Việc tích hợp RBACDom ở mức nút vào G được định nghĩa thông qua ánh xạ toàn phần $targetNode : RbacDomNode \to ND$ ánh xạ mỗi $rna \in RbacDomNode$ tới một nút đích duy nhất. Một nút $n \in N$ được kích hoạt trong một ngữ cảnh thực thi khi và chỉ khi tất cả các ràng buộc RBAC và ràng buộc phân tách nhiệm vụ được đặc tả bởi mọi rna sao cho $targetNode(rna) = n$ đều được thỏa mãn.*□

Ngữ nghĩa kích hoạt nút cho các đồ thị được gắn RBACDom. Phần này định nghĩa ngữ nghĩa hình thức của việc kích hoạt nút đối với các đồ thị hoạt động được gắn RBACDom. Việc ủy quyền được diễn giải như một ràng buộc ngữ nghĩa trên thực thi hành vi: một nút hoạt động chỉ có thể được kích hoạt và thực thi trong một phiên cho trước nếu các chính sách

RBAC liên kết với nó được thỏa mãn. Ngữ nghĩa này được định nghĩa ở mức các nút của đồ thị hoạt động, vốn đóng vai trò là các ranh giới thực thi của hành vi miền.

Một nút n được kích hoạt trong một hình chụp trạng thái hệ thống $\sigma \in \Sigma$ khi và chỉ khi nó được kích hoạt bởi ngữ nghĩa luồng điều khiển của AGL và tất cả các ràng buộc RBACDom được thỏa mãn đối với ngữ cảnh ủy quyền chứa trong $S_{DSL_c}(\sigma)$.

Các hàm phụ trợ và kiểu. Các hàm và vị từ phụ trợ được sử dụng trong ngữ nghĩa kích hoạt được định nghĩa và gán kiểu như sau:
$Ann : ND \to \mathcal{P}(RbacDomNode)$
ánh xạ mỗi nút hoạt động tới tập các chú thích RBACDom của nó.

Với mỗi chú thích $rna \in RbacDomNode$, các hàm ánh xạ nút RbacDom được định nghĩa như sau:

$$
\begin{aligned}
&policy : RbacDomNode \to \{ALL\_OF, ANY\_OF\}\\
&effect : RbacDomNode \to \{ALLOW, DENY\}\\
&ReqRoles : RbacDomNode \to \mathcal{P}(R)\\
&ReqPerms : RbacDomNode \to \mathcal{P}(P)\\
&SoD : RbacDomNode \to \mathcal{P}(SC)
\end{aligned}
$$

Các quyền được kích hoạt bởi một tập vai trò được suy ra bởi:
$Perms : \mathcal{P}(R) \to \mathcal{P}(P)$, trong đó $Perms(R') = \{p \in P \mid \exists\, r \in R' \cdot (r, p) \in PA\}$.

Vị từ $Violates : SC \times SES \to \mathbb{B}$ chỉ ra liệu một ràng buộc phân tách nhiệm vụ có bị vi phạm trong một phiên cho trước hay không.

Một chú thích RBACDom gắn với nút $rna \in Ann(n)$ được xem là *áp dụng* trong một phiên $s \in SES$ khi và chỉ khi tất cả các điều kiện về vai trò, quyền và SoD của nó đều được thỏa mãn: $Applies(rna, s) \quad s \in SES \triangleq RolesSat(rna, s) \wedge PermsSat(rna, s) \wedge SoDSat(rna, s)$.

Các vị từ thành phần được định nghĩa như sau:
$RolesSat(rna, s) \triangleq \big(policy(rna) = ALL\_OF \Rightarrow ReqRoles(rna) \subseteq activeRoles(s)\big)$
$\wedge\ \big(policy(rna) = ANY\_OF \Rightarrow ReqRoles(rna) \cap activeRoles(s) \neq \varnothing\big)$
$PermsSat(rna, s) \triangleq ReqPerms(rna) \subseteq Perms(activeRoles(s))$
$SoDSat(rna, s) \triangleq \forall\, c \in SoD(rna) \cdot \neg\, Violates(c, s)$

Ngữ nghĩa kết hợp luật. Gọi $s \in SES$ là một trạng thái phiên, trong đó SES là tập các trạng thái phiên. Đối với một nút hành vi $n \in ND$, một chú thích RBACDom $rna \in RbacDomNode$ được gọi là áp dụng trong trạng thái s nếu các ràng buộc về vai trò, quyền và ngữ cảnh của nó được thỏa mãn. Ta ký hiệu vị từ này là $Applies(rna, s)$. Tập các chú thích RBACDom áp dụng cho nút n trong trạng thái s được định nghĩa như sau: $App(n, s) = \{\, rna \in Ann(n) \mid Applies(rna, s) \,\}$. Gọi $effect : RbacDomNode \to \{ALLOW, DENY\}$ là hàm biểu diễn hiệu lực phân quyền gắn với một chú thích RBACDom. Quyết định phân quyền cho nút n trong trạng thái s được xác định bởi một hàm quyết định phụ thuộc chiến lược
$Dec : ND \times SES \to \{ALLOW, DENY\}$. Định nghĩa của $Dec(n, s)$ phụ thuộc vào chiến lược kết hợp luật được chọn bởi $combiningAlg(R)$.

Deny-Overrides. Nếu $combiningAlg(R) =$ denyOverrides, quyết định được xác định như sau:

$$Dec(n,s) = \begin{cases} DENY & \text{nếu } \exists\, rna \in App(n,s) \cdot effect(rna) = DENY \\ ALLOW & \text{nếu } \exists\, rna \in App(n,s) \cdot effect(rna) = ALLOW \\ DENY & \text{trong các trường hợp còn lại.} \end{cases}$$

Permit-Overrides. Nếu $combiningAlg(R) =$ permitOverrides, quyết định được xác định như sau:

$$Dec(n,s) = \begin{cases} ALLOW & \text{nếu } \exists\, rna \in App(n,s) \cdot effect(rna) = ALLOW \\ DENY & \text{nếu } \neg \exists\, rna \in App(n,s) \cdot effect(rna) = ALLOW \\ & \qquad \wedge \exists\, rna \in App(n,s) \cdot effect(rna) = DENY \\ DENY & \text{trong các trường hợp còn lại.} \end{cases}$$

First-Applicable. Nếu $combiningAlg(R) =$ firstApplicable, các luật được đánh giá theo thứ tự được xác định bởi quan hệ ord.

$$first(n,s) = \arg\, min_{rna \in App(n,s)}\, ord(rna)$$

Khi đó, quyết định được xác định như sau:

$$Dec(n,s) = \begin{cases} effect(first(n,s)) & \text{nếu } App(n,s) \neq \varnothing \\ DENY & \text{trong các trường hợp còn lại.} \end{cases}$$

Cuối cùng, một nút hành vi được xem là được cấp quyền thực thi nếu $Enable_{RBAC}(n, s) \triangleq Dec(n, s) = ALLOW$.

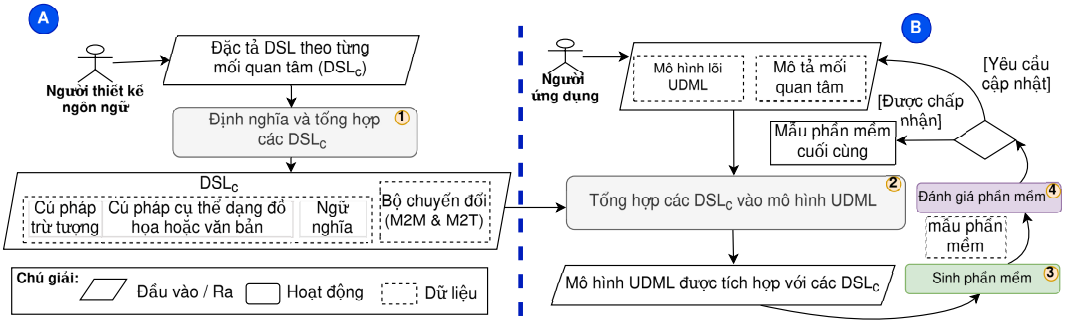
Do đó, thành phần RBACDom trong việc kích hoạt nút được biểu diễn bởi $Enable_{RBAC}(n, s)$. Một nút hành vi n chỉ có thể thực thi khi nó được kích hoạt theo ngữ nghĩa hành vi của AGL và đồng thời điều kiện phân quyền $Enable_{RBAC}(n, s)$ được thỏa mãn.

4.3 Phương pháp biểu diễn mô hình miền hợp nhất dựa vào cây cú pháp

Phần này trình bày một cách tiếp cận khác để hợp nhất các DSL được thiết kế chuyên biệt cho từng mối quan tâm vào một mô hình miền hợp nhất, thông qua cơ chế tích hợp dựa trên cây cú pháp trừu tượng (AST). Cách tiếp cận này hướng tới việc xây dựng một miền kỹ thuật có khả năng thực thi, trong đó các mối quan tâm được kết hợp trực tiếp ở mức cú pháp trừu tượng và được gắn với ngữ nghĩa hình thức.

4.3.1 Tổng quan về phương pháp đề xuất

Phương pháp tổng hợp các DSL theo mối quan tâm vào một DM hợp nhất, như minh họa trong Hình 4.7, được tổ chức thành một quy trình lặp gồm bốn bước, nhằm tích hợp và tinh chỉnh các DSL hướng tới việc sinh tự động bản mẫu phần mềm.



Hình 4.7: Tổng quan phương pháp đề xuất, được tổ chức thành hai giai đoạn: (A) thiết kế ngôn ngữ và (B) ứng dụng ngôn ngữ.

Thứ nhất, ở giai đoạn thiết kế ngôn ngữ (nhãn A), nhà thiết kế đặc tả miền mối quan tâm bằng một DSL chuyên biệt cho mối quan tâm (DSL_c). Đối với mỗi DSL_c, cú pháp trừu tượng (AS), cú pháp cụ thể (CS), và ngữ nghĩa được định nghĩa một cách hình thức, tạo thành một DSL_c mô tả chính xác mối quan tâm mục tiêu. *Thứ hai,* ở giai đoạn ứng dụng (nhãn B), nhà thiết kế sử dụng đầu ra của bước một cùng với mô hình lõi UDML hiện có. Kết hợp với mô tả về các yêu cầu mới hoặc thay đổi của mối quan tâm, các đầu vào này cho phép tổng hợp các DSL_c vào mô hình UDML theo cách thống nhất và nhất quán. Cú pháp cụ thể dạng văn bản hoặc đồ họa có thể được sử dụng để xây dựng mô hình chi tiết biểu diễn thể hiện cụ thể của mối quan tâm. Kết quả đầu ra của bước này là một mô hình UDML đã được tích hợp đầy đủ với mối quan tâm tương ứng. *Thứ ba,* mô hình UDML đã tích hợp đóng vai trò làm nền tảng cho việc sinh phần mềm. Tạo tác phần mềm thu được sau đó được nhà thiết kế đánh giá để thu thập phản hồi. *Cuối cùng,* nếu có phản hồi, mô hình UDML và đặc tả mối quan tâm tương ứng sẽ được cập nhật. Quy trình sau đó được lặp lại, hỗ trợ chu trình cải tiến liên tục cho đến khi hệ thống phần mềm thỏa mãn đầy đủ các yêu cầu đã đề ra.

4.3.2 Biểu diễn và tích hợp các mối quan tâm

Phần này định nghĩa cú pháp và ngữ nghĩa của UDML và giới thiệu thuật toán tổng hợp các DSL vào một DM hợp nhất.

Cây cú pháp trừu tượng của UDML

Phương pháp được trình bày trong mục này, thực hiện định nghĩa UDML tuân theo một khuôn khổ ba lớp có cấu trúc, bao gồm cú pháp trừu tượng, cú pháp cụ thể và ngữ nghĩa hình thức, cùng với một chiến lược tích hợp.

Ngôn ngữ mô hình miền hợp nhất (UDML) được xây dựng từ một lõi chung (UDML core) và một tập các DSL theo từng mối quan tâm $DSL_{i=1}^{n}$, trong đó mỗi DSL_i đặc tả một khía cạnh riêng biệt của hệ thống. Mỗi DSL được đặc trưng bởi ba thành phần: cú pháp trừu tượng (AS_i), cú pháp cụ thể (CS_i) và ngữ nghĩa (Sem_i), qua đó cho phép biểu diễn và tích hợp các mối quan tâm không đồng nhất vào một mô hình miền hợp nhất.

Trên cây cú pháp trừu tượng của UDML được xác định đầy đủ về AS, CS và ngữ nghĩa. AS của UDML được hình thức hóa dưới dạng một cấu trúc cây phân cấp, trong đó mỗi nút biểu diễn một thể hiện của khái niệm ngôn ngữ; các cạnh mô tả quan hệ tổng hợp hoặc tham chiếu; và các thuộc tính ghi nhận đặc điểm cục bộ của từng nút. Các concern DSL mở rộng cây này một cách tăng dần bằng cách bổ sung các khái niệm và quan hệ mới mà vẫn duy trì khung xương chung được xác định trong UDML lõi.

Định nghĩa 12 (Cây cú pháp trừu tượng của UDML)**.** *Cho C_{UDML} là tập hợp các siêu khái niệm của UDML và A_{UDML} là tập thuộc tính tương ứng. Với mỗi khái niệm $c \in C_{UDML}$, $A(c) \subseteq A_{UDML}$ là tập thuộc tính của c. Cú pháp trừu tượng **AS** của UDML được định nghĩa bởi bộ $AS_{UDML} = (N, root, child, refCons, label, attr, P_{UDML})$, trong đó:*

- *N: tập hữu hạn các nút.*
- *$root \in N$: nút gốc biểu diễn DM hợp nhất, được thể hiện trong UDML dưới dạng* `DomainModel`*.*
- *$child \subseteq N \times N$: quan hệ cây xác định các nút con có thứ tự.*
- *$refCons \subseteq N \times N$: tập các cạnh tham chiếu liên kết các nút giữa các concern.*
- *$label : N \to C_{UDML}$: hàm gán nhãn xác định siêu khái niệm của mỗi nút.*
- *$attr : N \to (A(label(n)) \to V)$: hàm gán giá trị thuộc tính cho từng nút.*
- *P_{UDML}: tập các quy tắc và ràng buộc bảo đảm tính đúng đắn của cây.* □

CS được định nghĩa như một phép chiếu từ cây AST sang các ký hiệu hướng người dùng. Mỗi góc nhìn cú pháp cung cấp cách biểu diễn phù hợp với miền, trong khi tất cả các góc nhìn đều thao tác trên cùng một AST nhằm bảo đảm tính nhất quán và đồng bộ.

Định nghĩa 13 (Cú pháp cụ thể của UDML)**.** *Cú pháp cụ thể **CS** của UDML được định nghĩa bởi bộ $CS_{UDML} = (V, M, \pi, H, P_{CS})$, trong đó:*

- *V: tập các ký hiệu trực quan (hộp, mũi tên, bảng, nhãn...).*
- *M: tập quy tắc ánh xạ liên kết đến nút AST hoặc cạnh được biểu diễn trực quan.*
- *$\pi : N \cup (N \times N) \to V$: hàm chiếu ánh xạ nút/cạnh sang ký hiệu cụ thể.*
- *H: tập các bộ xử lý thao tác (tạo, chỉnh sửa, kéo/thả, gán chú thích,...).*

- P_{CS}*: tập ràng buộc bảo đảm tính hợp lệ của phép biểu diễn.*□

Ngoài ra, hàm chiếu có thể được chỉ số hóa theo từng góc nhìn, $\pi_v : N \cup (N \times N) \to V$, với $v \in Views$, cho phép nhiều ký pháp khác nhau mô tả cùng một AST.

Ngữ nghĩa của UDML được xác định bằng cách gán ý nghĩa trực tiếp cho từng nút và cạnh trong AST. Mỗi concern DSL đóng góp một ánh xạ ngữ nghĩa độc lập, và các ánh xạ này được tổng hợp dựa theo cấu trúc AST.

Định nghĩa 14 (Ngữ nghĩa của UDML)**.** *Ngữ nghĩa* ***Sem*** *của UDML được định nghĩa bởi hàm* $\langle Sem \rangle_{UDML} : N \cup (N \times N) \to D$*, trong đó:*

- N*: tập các nút AST.*
- $N \times N$*: tập các cạnh (cây hoặc tham chiếu).*
- D*: miền ngữ nghĩa, là tích của các miền mối quan tâm:* $D = D_{dcsl} \times D_{agl} \times D_{rbacDom}$
- $\langle Sem(n) \rangle$*: ngữ nghĩa tại nút* n*, do concern DSL tương ứng xác định.*
- $\langle Sem(n_i, n_j) \rangle$*: ngữ nghĩa của cạnh, cho biết cách ngữ nghĩa được lan truyền theo quan hệ cha–con hoặc tham chiếu.*□

Ba định nghĩa (Định nghĩa 12, Định nghĩa 13 và Định nghĩa 14) thiết lập cơ sở hình thức cho ngôn ngữ UDML, bao gồm cú pháp trừu tượng, cú pháp cụ thể và ngữ nghĩa. Cụ thể, cú pháp trừu tượng xác định cấu trúc của mô hình miền hợp nhất, cú pháp cụ thể hỗ trợ biểu diễn và thao tác mô hình, trong khi ngữ nghĩa quy định cách diễn giải các phần tử mô hình trong miền ngữ nghĩa. Trên cơ sở đó, các định nghĩa này đóng vai trò nền tảng cho việc xây dựng biểu diễn miền hợp nhất, thiết lập các liên kết ngữ nghĩa giữa các DSL theo mối quan tâm, cũng như hỗ trợ các phép chuyển đổi mô hình và kiểm chứng hình thức trong các phần tiếp theo của luận án.

Ngữ nghĩa toàn cục của UDML được xác định thông qua việc tổng hợp ngữ nghĩa địa phương của các nút và cạnh trong mô hình. Mỗi DSL bảo toàn quy tắc diễn giải riêng trong cấu trúc hợp nhất, đồng thời bảo đảm tính nhất quán ngữ nghĩa xuyên mối quan tâm. Quá trình hợp nhất được thực hiện theo cách tiếp cận tăng dần thông qua thuật toán hợp cây (Thuật toán 4.1), trong đó các khái niệm được tích hợp dựa trên cơ chế kế thừa và liên kết ngữ nghĩa.

Cơ chế tích hợp các DSL theo mối quan tâm vào UDML

Thuật toán 4.1 Thuật toán hợp nhất cây tích hợp các DSL theo mối quan tâm vào UDML

Đầu vào: $D = \{DSL_1, DSL_2, \ldots, DSL_n\}$: Tập các concern DSLs, mỗi DSL được đặc tả bởi bộ ba (AS_i, CS_i, Sem_i)

Đầu ra : $UDML_{unified}$: UDML hợp nhất bao gồm cú pháp trừu tượng AS, cú pháp cụ thể CS và ngữ nghĩa Sem

1 $AS_{UDML} \leftarrow AS_{core}$, $CS_{UDML} \leftarrow CS_{core}$, $Sem_{UDML} \leftarrow Sem_{core}$
2 **foreach** *mỗi* $DSL_i = (AS_i, CS_i, Sem_i) \in D$ **do**
  // Xử lý các khái niệm trong AS
3 **foreach** *mỗi khái niệm* $c \in AS_i$ *sao cho* $c \notin AS_{UDML}$ **do**
4 **if** *c mở rộng một khái niệm lõi* **then**
5 Tích hợp quan hệ kế thừa vào AS_{UDML}
6 **end if**
7 Thêm các thuộc tính, quan hệ và ràng buộc của c vào AS_{UDML}
8 **end foreach**
9 **foreach** *mỗi cạnh* $(n_i, n_j) \in child_i \cup refCons_i$ **do**
10 **if** $(n_i, n_j) \notin (child_{UDML} \cup refCons_{UDML})$ **then**
11 Thêm cạnh (n_i, n_j) vào AS_{UDML}
12 **end if**
13 **else**
14 Giải quyết xung đột bằng cách áp dụng quy tắc ánh xạ hoặc quy tắc ưu tiên
15 **end if**
16 **end foreach**
  // Xử lý cú pháp cụ thể
17 **foreach** *mỗi khái niệm mới hoặc khái niệm được mở rộng* c **do**
18 Thêm các ký hiệu đồ họa hoặc văn bản tương ứng vào CS_{UDML} Mở rộng các bộ xử lý UI (ví dụ: kéo-thả, chú thích, thao tác chỉnh sửa)
19 **end foreach**
  // Tích hợp ngữ nghĩa
20 Tích hợp Sem_i vào Sem_{UDML} theo kiểu mô-đun hợp nhất các ràng buộc mới vào P_{UDML} Kiểm tra tính nhất quán giữa các ngữ nghĩa xuyên mối quan tâm của Sem_i và Sem_{UDML}
21 **end foreach**
22 **return** $UDML_{unified} = (AS_{UDML}, CS_{UDML}, Sem_{UDML})$

Một AST hợp nhất được xây dựng từ nhiều DSL theo mối quan tâm dựa trên UDML lõi và các phần mở rộng của nó (như DCSL, AGL, CAP). Mỗi nút biểu diễn một thể hiện siêu khái niệm (ví dụ: `Student:DClass`) và các cạnh biểu diễn quan hệ tổng hợp hoặc tham chiếu (ví dụ: `AGraph` → `DClass`).

Thuật toán 4.1 mô tả quy trình hợp nhất tăng dần các DSL theo mối quan tâm vào UDML. Thuật toán khởi tạo UDML với các thành phần lõi (dòng 1), sau đó lần lượt hợp nhất từng DSL trong tập đầu vào (dòng 2). Đầu tiên, cú pháp trừu tượng được tích hợp bằng cách thêm các khái niệm mới và quan hệ mới (dòng 3–16). Tiếp theo là hợp nhất cú pháp cụ thể

(dòng 17–19), rồi hợp nhất ngữ nghĩa gửi từ từng DSL theo từng mối quan tâm vào ngữ nghĩa tổng thể của UDML (dòng 20–21). Sau khi tất cả các DSL được xử lý, thuật toán trả về UDML hợp nhất (dòng 22), tạo thành mô hình thực thi thống nhất, bảo tồn tính mô-đun và truy vết xuyên suốt các mối quan tâm.NguNghiaHinhThucUDML

4.4 Ngữ nghĩa của mô hình miền hợp nhất

Mục này trình bày khung ngữ nghĩa hình thức cho các mô hình miền hợp nhất trong UDML. Một mô hình UDML được xem như sự tích hợp có hệ thống giữa mô hình miền lõi và các DSL chuyên biệt theo mối quan tâm, trong đó mỗi mối quan tâm đóng góp các khía cạnh ngữ nghĩa riêng nhưng có liên kết chặt chẽ. Ngữ nghĩa của mô hình được đặc trưng theo quan điểm thực thi, dựa trên các dãy hình chụp trạng thái phản ánh sự tiến hóa của hệ thống theo thời gian. Mỗi hình chụp biểu diễn một trạng thái hợp nhất, bao gồm trạng thái của mô hình lõi và các trạng thái do các DSL theo mối quan tâm đóng góp $(S_{\text{core}}, S_{DSL_c})$, qua đó thiết lập một ngữ nghĩa thao tác chung mà không phụ thuộc vào cách hiện thực cụ thể của từng DSL. Trên cơ sở đó, mục này xây dựng các định nghĩa hình thức cho mô hình UDML, điều kiện hợp lệ của trạng thái, và ngữ nghĩa thực thi, làm nền tảng cho việc phân tích, kiểm chứng và chuyển đổi mô hình bảo toàn ngữ nghĩa.

4.4.1 Định nghĩa hình thức các mô hình UDML.

Định nghĩa hình thức các mô hình UDML như các mô hình miền có khả năng thực thi hợp nhất, thu được bằng cách kết hợp nhiều mối quan tâm trực giao nhưng có liên hệ ngữ nghĩa với nhau.

Định nghĩa 15 (Mô hình UDML)**.** *Một mô hình UDML được định nghĩa là một bộ:* $\mathcal{M} = \langle \mathcal{U}, \mathcal{S}, \mathcal{B}, \mathcal{R}, \mathcal{C} \rangle$ *trong đó:*

- $\mathcal{U}$ *là siêu mô hình kết hợp lõi của UDML, cung cấp các trừu tượng cho mô-đun hóa mối quan tâm và liên kết chú thích, đồng thời độc lập với ngữ nghĩa thao tác;*

- $\mathcal{S}$ *là mô hình mối quan tâm cấu trúc DCSL, bao gồm các chú thích và các ràng buộc của DCSL được gắn với các phần tử cấu trúc của mô hình miền;*
- $\mathcal{B}$ *là mô hình mối quan tâm hành vi AGL, bao gồm các đồ thị hoạt động mà các nút và cạnh của chúng đặc trưng cho sự tiến hóa hành vi cho phép và các ranh giới thực thi cho các hành động miền;*
- $\mathcal{R}$ *là mô hình mối quan tâm bảo mật RBACDom, đặc tả các ràng buộc ủy quyền và SoD nhằm hạn chế khả năng thực thi của các phần tử hành vi (đặc biệt là việc thực thi nút);*
- $\mathcal{C}$ *là tập các ràng buộc toàn cục, bao gồm các điều kiện hợp lệ và các ràng buộc nhất quán xuyên mối quan tâm phải được bảo toàn bởi mọi phép thực thi mô hình.*□

Sự phân rã này làm rõ vai trò của từng mối quan tâm mô hình hóa, đồng thời cho phép tích hợp chúng một cách có hệ thống trong một mô hình miền có khả năng thực thi duy nhất. Cụ thể, mô hình hành vi $\mathcal{B}$ quyết định không gian các chuyển tiếp trạng thái cho phép, mô hình bảo mật $\mathcal{R}$ ràng buộc những chuyển tiếp nào có thể thực thi dưới các điều kiện ủy quyền cho trước, và tập ràng buộc $\mathcal{C}$ xác định các thuộc tính đúng đắn toàn cục phải được duy trì trong suốt quá trình thực thi.

Hình thức hóa lõi UDML. Lõi UDML cung cấp các trừu tượng nền tảng để kết hợp nhiều DSL chuyên biệt theo mối quan tâm theo cách thống nhất và không xâm lấn. Lõi này định nghĩa "kết dính" cấu trúc cho phép tích hợp các mối quan tâm cấu trúc, hành vi và bảo mật mà bản thân nó không áp đặt ngữ nghĩa thực thi. Việc định nghĩa ngữ nghĩa hình thức của UDML như sau.

- *Các phần tử miền lõi:* Lõi UDML được hình thức hóa như một bộ: $\mathcal{U} = \langle DM,\ CN,\ AN,\ AE,\ elements,\ concerns,\ annotations,\ target \rangle$. Giả sử các tập mang đôi một rời nhau sau đây được cho: $DM,\ CN,\ AN,\ AE$, trong đó DM là tập các mô hình miền; CN là tập các mối quan tâm được gắn với các mô hình miền; AN là tập các chú thích được dùng để biểu diễn thông tin đặc thù theo mối quan tâm; AE là tập các phần tử có thể gắn chú thích, tức là các phần tử của mô hình miền có thể được mở rộng bởi chú thích.

Cho các phần tử miền lõi sau đây:

$$\mathit{Class},\ \mathit{Property},\ \mathit{Operation},\ \mathit{Association} \subseteq AE$$

Các phần tử này cấu thành cú pháp trừu tượng của mô hình miền trong UDML. Chúng biểu diễn các khái niệm mô hình hóa cấu trúc cơ bản—tương tự các lớp, thuộc tính, thao tác và liên kết trong UML—mà trên đó hành vi miền và các ngữ nghĩa đặc thù theo mối quan tâm được định nghĩa.

Vì vậy, các phần tử có thể gắn chú thích tạo thành một không gian đích chung cho việc tích hợp mối quan tâm: mọi DSL chuyên biệt theo mối quan tâm như DCSL, AGL và RBACDom đóng góp ngữ nghĩa bằng cách gắn chú thích lên các phần tử trong AE thay vì tái định nghĩa chính cấu trúc mô hình miền.

- *Các quan hệ lõi:* Các quan hệ lõi của $\mathcal{U}$ được định nghĩa như sau: $\mathit{elements} \subseteq DM \times AE$ liên kết mỗi mô hình miền với các phần tử có thể gắn chú thích của nó; $\mathit{concerns} \subseteq DM \times CN$ liên kết mỗi mô hình miền với các mối quan tâm được khai báo; $\mathit{annotations} \subseteq CN \times AN$ liên kết mỗi mối quan tâm với các chú thích của nó; $\mathit{target} : AN \to AE$ ánh xạ mỗi chú thích tới phần tử có thể gắn chú thích đích.

- *Các ràng buộc hợp lệ:* Các ràng buộc sau đây phải được thoả mãn.
  (U1) $\forall\, a \in AN \cdot \mathit{target}(a) \in AE$;
  (U2) $\forall\, a \in AN \cdot \exists! c \in CN : (c, a) \in \mathit{annotations}$;
  (U3) $\forall\, c \in CN \cdot \exists! d \in DM : (d, c) \in \mathit{concerns}$.
  (U4) $\forall\, a \in AN \cdot \exists\, d \in DM : (d, \mathit{target}(a)) \in \mathit{elements} \wedge \exists\, c \in CN : (d, c) \in \mathit{concerns} \wedge (c, a) \in \mathit{annotations}$.
  Các ràng buộc này đảm bảo cấu trúc sở hữu và gắn kết của các mối quan tâm và chú thích trong một mô hình miền được xác định rõ ràng.

Định nghĩa 16 (Hình chụp hệ thống (snapshot))**.** *Cho $\mathcal{M}$ là một mô hình UDML. Ký hiệu Σ là tập các hình chụp do $\mathcal{M}$ sinh ra. Mỗi hình chụp $\sigma \in \Sigma$ được định nghĩa như một trạng thái có cấu trúc: $\sigma \triangleq \langle S_{core}(\sigma),\ S_{DSL_c}(\sigma)\rangle$, trong đó $S_{core}(\sigma)$ biểu diễn trạng thái của mô hình miền lõi, còn $S_{DSL_c}(\sigma)$ biểu diễn trạng thái của các DSL theo mối quan tâm (ví dụ: hành vi từ AGL và ngữ cảnh ủy quyền từ RBACDom). Sự tương thích giữa các thành phần này được xác định thông qua các liên kết ngữ nghĩa trong lõi UDML.*□

Dựa trên khái niệm hình chụp hệ thống, ta xác định điều kiện hợp lệ của một hình chụp như sau.

Định nghĩa 17 (Hình chụp hợp lệ)**.** *Cho $\mathcal{M}$ là một mô hình UDML và $\sigma \in \Sigma$. Hình chụp σ được gọi là* hợp lệ*, ký hiệu (σ), khi và chỉ khi σ thỏa mãn đồng thời: (i) các ràng buộc cấu trúc suy ra từ DCSL; (ii) các bất biến bảo mật suy ra từ RBACDom; (iii) các bất biến do các DSL theo mối quan tâm khác đóng góp; và (iv) các điều kiện liên kết ngữ nghĩa trong lõi UDML nhằm đảm bảo sự nhất quán giữa $S_{core}(\sigma)$ và $S_{DSL_c}(\sigma)$.*□

Điều kiện hợp lệ này xác định các trạng thái có ý nghĩa về ngữ nghĩa, và sẽ được sử dụng để ràng buộc quá trình thực thi của mô hình.

Định nghĩa 18 (Ngữ nghĩa thực thi)**.** *Ngữ nghĩa của một mô hình UDML $\mathcal{M}$ được định nghĩa như một hệ chuyển: $\mathcal{M} = \langle \Sigma, \Sigma_0, \rightarrow \rangle$, trong đó Σ là tập các hình chụp, $\Sigma_0 \subseteq \Sigma$ là tập các hình chụp khởi tạo, và $\rightarrow \subseteq \Sigma \times ND \times \Sigma$ là quan hệ chuyển trạng thái. Một phép thực thi (hay một đường đi) của $\mathcal{M}$ là một dãy (hữu hạn hoặc vô hạn) $\sigma_0, \sigma_1, \ldots$ sao cho $\sigma_0 \in \Sigma_0$, (σ_0), và với mỗi $i \geq 0$, $\sigma_i \xrightarrow{n_i} \sigma_{i+1}$ với (σ_{i+1}).*□

Một phép thực thi biểu diễn một lần chạy khả dĩ của mô hình $\mathcal{M}$, bắt đầu từ một hình chụp khởi tạo và tiến hóa qua các lần thực thi nút liên tiếp. Mỗi chuyển trạng thái tương ứng với việc thực thi một nút làm biến đổi trạng thái hệ thống từ hình chụp này sang hình chụp kế tiếp. Ở mỗi bước, điều kiện kích hoạt nút $Enable(n_i, s)$ được đánh giá trên hình chụp hiện tại σ_i, sử dụng thành phần phiên $s \in SES$ chứa trong S_{DSL_c}. Việc chuyển sang σ_{i+1} chỉ được phép nếu nó bảo toàn tất cả các ràng buộc cấu trúc do DCSL suy ra và tất cả các bất biến do các DSL_c đã được kết hợp đóng góp trong hình chụp kết quả.

Trong phần tiếp theo, luận án làm rõ một cách tường minh phép ánh xạ từ các phép thực thi của AGL sang ngữ nghĩa thực thi của mô hình UDML thông qua các hình chụp trạng thái hệ thống.

4.4.2 Ánh xạ thực thi trong AGL sang UDML

Định nghĩa 19 (Ánh xạ các hình chụp)**.** *Cho $\mathcal{M}$ là một mô hình UDML và Σ_{AGL} là tập các hình chụp do đặc tả AGL của nó sinh ra. Một hình chụp*

của AGL $q \in \Sigma_{AGL}$ tương ứng với một hình chụp của UDML $\sigma \in \Sigma$, ký hiệu $q \sim \sigma$, khi và chỉ khi: $q = \pi_{AGL}(\sigma)$, trong đó $\pi_{AGL} : \Sigma \to \Sigma_{AGL}$ là phép chiếu trích xuất thành phần hành vi AGL từ $S_{DSL_c}(\sigma)$. Ánh xạ này cho phép liên kết ngữ nghĩa giữa trạng thái hành vi của AGL và trạng thái hợp nhất của UDML.□

Ví dụ. Ví dụ sau minh họa một phép thực thi của hệ thống OJS như một dãy các hình chụp do đặc tả AGL của nó sinh ra. $q_0 \xrightarrow{\text{submit}} q_1 \xrightarrow{\text{approve}} q_2$ tương ứng với phép thực thi UDML $\sigma_0 \xrightarrow{\text{submit}} \sigma_1 \xrightarrow{\text{approve}} \sigma_2$. Ta có các ánh xạ hình chụp $q_i \sim \sigma_i$, (i=0,2). Nếu RBACDom yêu cầu vai trò Manager cho thao tác approve, thì bước thứ hai chỉ được kích hoạt khi điều kiện ủy quyền được thỏa mãn; nếu không, phép thực thi sẽ dừng tại σ_1. Trong cả hai trường hợp, sự tương ứng về hành vi vẫn được bảo toàn.

Theorem 4.1 (Đồng thực thi trên UDML và AGL)**.** *Cho $\mathcal{M}$ là một mô hình UDML kết hợp AGL với một tập các DSL chuyên biệt theo mối quan tâm. Giả sử rằng mỗi DSL mối quan tâm được kết hợp chỉ ràng buộc hành vi bằng cách hạn chế việc kích hoạt nút (tức là bằng cách bổ sung các điều kiện bảo vệ và/hoặc các bất biến trạng thái), và không làm thay đổi hình chụp AGL. Khi đó: (i) mọi phép thực thi UDML khi được chiếu qua π_{AGL} sẽ cho ra một phép thực thi AGL hợp lệ; và (ii) mọi phép thực thi AGL thỏa mãn các ràng buộc kích hoạt và các bất biến do các mối quan tâm được kết hợp sinh ra đều có thể được "nâng" lên thành một phép thực thi UDML tương ứng.*□

Chứng minh *(Ánh xạ một-một).*

> Cho $\mathcal{M}$ là một mô hình UDML đảm báo tính hợp lệ.
> Theo cấu trúc xây dựng, mỗi hình chụp của UDML $\sigma \in \Sigma$ chứa một thành phần hành vi AGL, và phép chiếu π_{AGL} thiết lập một ánh xạ từ các hình chụp của UDML sang các hình chụp của AGL.

Tính đúng. Mỗi bước UDML $\sigma_i \xrightarrow{n_i} \sigma_{i+1}$ thực thi một nút hành vi n_i được định nghĩa bởi AGL. Vì các DSL mối quan tâm được kết hợp không làm thay đổi trạng thái hành vi AGL và chỉ hạn chế khi nào một nút được kích hoạt (thông qua các điều kiện bảo vệ) và những trạng thái nào là cho phép (thông qua các bất biến), mỗi bước UDML tương ứng với một bước AGL

cho phép trên các trạng thái đã được chiếu. Do đó, chiếu một phép thực thi UDML qua π_{AGL} sẽ thu được một phép thực thi AGL hợp lệ.

Tính đầy đủ. Ngược lại, xét một phép thực thi AGL thỏa mãn tất cả các hạn chế kích hoạt và các bất biến do các mối quan tâm được kết hợp sinh ra. Bắt đầu từ một hình chụp UDML khởi tạo σ_0 với $\pi_{AGL}(\sigma_0) = q_0$, mỗi bước AGL có thể được hiện thực như một chuyển tiếp UDML vì cùng một nút được kích hoạt và hình chụp kết quả vẫn là cho phép. Lặp lại lập luận này theo từng bước sẽ xây dựng được một phép thực thi UDML mà phép chiếu của nó khớp với phép thực thi AGL đã cho.

Vì vậy, dọc theo các phép thực thi, các hình chụp của UDML và các hình chụp của AGL tương ứng một-một theo từng bước thông qua π_{AGL}.

4.4.3 Ánh xạ định nghĩa ngữ nghĩa sang Event-B

Mục này tập trung vào việc chuyển đổi mô hình UDML sang môi trường Event-B nhằm phục vụ cho mục đích mô phỏng và kiểm chứng các mô hình miền hợp nhất. Công việc tiếp theo là định nghĩa một phép biến đổi bảo toàn ngữ nghĩa, trong đó các mô hình UDML đã được tích hợp được chuyển dịch thành các "ngữ cảnh" và "máy trạng thái" của Event-B. Trên cơ sở các Định nghĩa 16, Định nghĩa 17, Định nghĩa 18 và Định nghĩa 19, không gian trạng thái của UDML được đặc trưng bởi tập các hình chụp Σ, trong đó mỗi trạng thái của máy Event-B tương ứng với một hình chụp hợp lệ của mô hình UDML. Thông qua phép chiếu π_{AGL}, mỗi hình chụp $\sigma \in \Sigma$ xác định một hình chụp hành vi $q \in \Sigma_{AGL}$, qua đó bảo toàn ngữ nghĩa thực thi của các phép chuyển trạng thái được đặc tả trong AGL. Ngược lại, mỗi sự kiện Event-B biểu diễn một chuyển trạng thái khả dĩ trong quan hệ $\rightarrow$, đồng thời phản ánh sự tiến hóa của thành phần hành vi được ánh xạ từ AGL. Do đó, ánh xạ từ UDML sang Event-B không chỉ bảo toàn cấu trúc trạng thái và các điều kiện bất biến, mà còn bảo toàn liên kết ngữ nghĩa giữa hành vi AGL và trạng thái hợp nhất của UDML, qua đó tạo cơ sở cho kiểm chứng hình thức và chuyển đổi bảo toàn ngữ nghĩa.

Thuật toán chuyển đổi mô hình UDML2Event-B 4.2 về cơ bản được xác định bởi ánh xạ, các phần tử cấu trúc và các ràng buộc toàn cục được chuyển dịch thành các "ngữ cảnh" trong Event-B, trong khi hành vi có khả năng thực thi và cơ chế cưỡng chế bảo mật được chuyển dịch thành các "máy

trạng thái" của Event-B. Sự phân tách này phù hợp với kỹ thuật mô hình hóa của Event-B và hỗ trợ suy luận mô-đun về cấu trúc miền, hành vi và kiểm soát truy cập.

Ánh xạ các phần tử cấu trúc. Các thành phần cấu trúc được đặc tả bằng DCSL—bao gồm các lớp miền, thuộc tính, liên kết và các bất biến cấu trúc—được ánh xạ sang các *ngữ cảnh - Context* của Event-B. Các lớp miền được biểu diễn dưới dạng các tập mang, các thuộc tính và liên kết được biểu diễn dưới dạng các biến hoặc hằng với các bất biến kiểu tương ứng, và các ràng buộc cấu trúc được chuyển dịch thành các bất biến của Event-B. Phép ánh xạ này xác lập không gian trạng thái miền tĩnh, trên đó hành vi được thực thi.

Ánh xạ các mô hình hành vi. Hành vi được đặc tả bằng AGL được ánh xạ sang các *máy trạng thái - machine* của Event-B. Mỗi đồ thị hoạt động được chuyển thành một *máy trạng thái*, trong đó các biến trạng thái mã hóa trạng thái điều khiển hiện tại của đồ thị. Các nút của đồ thị hoạt động được ánh xạ thành các sự kiện của Event-B, còn các cạnh xác định quan hệ luồng điều khiển giữa các sự kiện. Việc kích hoạt một sự kiện tương ứng với việc thực thi nút hoạt động liên quan, dẫn đến một chuyển trạng thái phản ánh cả sự tiến triển của luồng điều khiển và các cập nhật trạng thái miền do các hành động mô-đun tương ứng gây ra.

Ánh xạ các ràng buộc RBACDom. Các mối quan tâm bảo mật đặc tả bằng RBACDom được tích hợp trực tiếp vào ngữ nghĩa hành vi thông qua các điều kiện bảo vệ của sự kiện và các bất biến. Mỗi `RbacDomNode` tương ứng với một nút AGL (thông qua thuộc tính `aglNode`) được chuyển dịch thành các vị từ gác trên sự kiện Event-B tương ứng. Các vị từ gác này mã hóa các điều kiện kích hoạt nút, được suy ra từ yêu cầu về vai trò, kiểm tra quyền và các ràng buộc phân tách nhiệm vụ (SoD). Theo cách này, ủy quyền được cưỡng chế như một điều kiện tiên quyết cho việc thực thi sự kiện, thay vì là một kiểm tra bên ngoài áp đặt sau cùng.

Các ràng buộc SoD trải rộng trên nhiều nút hoặc nhiều phiên được chuyển dịch thành các bất biến của Event-B, bảo đảm rằng mọi trạng thái có thể đạt được của machine đều thỏa mãn các thuộc tính bảo mật đã được đặc tả. Điều này bảo đảm rằng các ràng buộc RBACDom được bảo toàn trong toàn bộ các lần thực thi và các bước tinh chỉnh của mô hình. Bảng 4.2 tóm

tắt các quy tắc ánh xạ giữa các cấu trúc của UDML và các tạo tác Event-B tương ứng. Từ góc nhìn này, ngữ nghĩa thực thi của một mô hình UDML

Bảng 4.2: UDML2Event-B: Ánh xạ từ UDML sang Event-B

Cấu trúc UDML	**Hiện vật Event-B**	**Mô tả**
Mô hình miền (DomainModel)	Ngữ cảnh	Định nghĩa miền tĩnh của hệ thống, bao gồm các tập kiểu cơ sở và các hằng số toàn cục.
Mối quan tâm (Concern)	Ngữ cảnh	Dùng để tổ chức các hằng, tiên đề và bất biến liên quan đến một mối quan tâm mô hình hóa cụ thể.
Lớp miền (DClass)	Tập mang / Hằng	Mỗi lớp miền được ánh xạ thành một tập mang hoặc một tập trừu tượng biểu diễn các thể hiện của nó.
Thuộc tính (DAttr)	Biến / Hàm	Thuộc tính được ánh xạ thành các biến trạng thái hoặc các hàm, trong đó miền xác định và miền giá trị được ràng buộc bởi các bất biến.
Liên kết (DAssoc)	Quan hệ / Bất biến	Liên kết được ánh xạ thành các quan hệ giữa các tập mang, với các ràng buộc cơ chế thực thi dưới dạng bất biến.
Đồ thị hoạt động (AGraph)	Máy trạng thái	Mỗi đồ thị hoạt động được ánh xạ thành một machine Event-B nắm bắt ngữ nghĩa thực thi của nó.
Nút khởi tạo	Khởi tạo sự kiện	Nút khởi tạo của đồ thị hoạt động xác định giá trị ban đầu của biến trạng thái điều khiển.
Nút (ANode)	Sự kiện	Mỗi nút được ánh xạ thành một sự kiện Event-B biểu diễn một bước thực thi nguyên tử.
Cạnh	Điều kiện bảo vệ	Các cạnh được ánh xạ thành các điều kiện ràng buộc bảo vệ sự kiện-nút nào có thể bấn dựa trên trạng thái điều khiển hiện tại.
Trạng thái điều khiển (nút hiện tại)	Biến	Nút đang hoạt động được biểu diễn bởi một biến trạng thái trong máy trạng thái.
Mô-đun hoạt động (ModuleAct)	Hành động sự kiện	Các hành động mô-đun được chuyển dịch thành các phép thay thế Event-B cập nhật trạng thái của máy trạng thái.
Nút bảo mật (RbacDomNode)	Điều kiện bảo vệ sự kiện	Các ràng buộc RBAC được đặc tả bởi `RbacDomNode` (tương ứng với các nút AGL) được ánh xạ thành các điều kiện bảo vệ nhằm cưỡng chế các yêu cầu về vai trò và quyền hạn.
Các vai trò yêu cầu	Vị từ bảo vệ	Yêu cầu vai trò được mã hóa dưới dạng các vị từ trên tập vai trò đang kích hoạt của phiên hiện tại.
Các quyền yêu cầu	Vị từ bảo vệ	Yêu cầu quyền hạn được mã hóa dưới dạng các vị từ suy ra từ các quan hệ gán vai trò–quyền hạn.
Phiên	Biến	Ngữ cảnh thực thi hiện tại được biểu diễn như một biến phiên.
Ràng buộc SoD	Bất biến	Các ràng buộc SoD được mã hóa thành các bất biến hạn chế việc gán vai trò hoặc kích hoạt vai trò.
Chính sách (ALL_OF / ANY_OF)	Lô-gic bảo vệ	Ngữ nghĩa của chính sách chi phối cách kết hợp các điều kiện bảo vệ RBAC theo phép AND hoặc OR.
Tác động (ALLOW / DENY)	Ngữ nghĩa bảo vệ	Hiệu lực xác định liệu điều kiện bảo vệ cho phép hay chặn việc thực thi sự kiện.
Chiến lược kết hợp các quy tắc (DENY_OVERRIDES)	Xây dựng điều kiện bảo vệ	Các điều kiện bảo vệ phân quyền được xây dựng dưới dạng $AllowApplies \wedge \neg DenyApplies$, đảm bảo rằng các chính sách từ chối có thể áp dụng sẽ chặn thực thi sự kiện ngay cả khi tồn tại các chính sách cho phép có thể áp dụng.
Các quy tắc hợp lệ	Bất biến / Tiên đề	Các ràng buộc hợp lệ được chuyển dịch thành các bất biến hoặc tiên đề để đảm bảo tính nhất quán của mô hình.

được xác định bởi hành vi của các *ngữ cảnh* và *máy trạng thái* Event-B được sinh ra, cũng như bởi các nghĩa vụ chứng minh mà chúng tạo ra.

Thuật toán 4.2 hiện thực hóa phép chuyển bảo toàn của UDML được định nghĩa hình thức ở trên, đồng thời xem UDML như một tầng tích hợp thống nhất cho các mối quan tâm cấu trúc, hành vi và bảo mật. Quá trình biên dịch được tổ chức thành các pha liên tiếp, mỗi pha tương ứng với một khối dòng được xác định rõ trong thuật toán.

Biên dịch cấu trúc (Dòng 1–4). Dòng 1–4 kiểm tra tính hợp lệ của mô hình đầu vào và khởi tạo ngữ cảnh và máy trạng thái Event-B. Sau đó, đặc tả cấu trúc được biểu diễn trong DCSL được chuyển đổi.

Biên dịch hành vi (Dòng 5–6). Dòng 5–6 chuyển dịch đặc tả hành vi trong AGL. Các tập mang và các hằng được đưa vào để biểu diễn các nút, cạnh và các quan hệ luồng điều khiển của đồ thị hoạt động. Một biến trạng thái điều khiển *pc* được thêm vào để ghi nhận nút đang hoạt động.

Biên dịch bảo mật (Dòng 7–8). Dòng 7–8 biên dịch đặc tả bảo mật RBACDom. Các khái niệm RBAC cốt lõi, bao gồm người dùng, vai trò, quyền hạn và phiên, được chuyển dịch thành các tập mang và các hằng. Các quan hệ gán và kích hoạt vai trò được biên dịch thành các biến của máy trạng thái, trong khi các ràng buộc phân tách nhiệm vụ và các ràng buộc RBAC khác được chuyển dịch thành các bất biến phải được bảo toàn bởi mọi sự kiện.

Tích hợp liên-mối quan tâm (Dòng 9). Dòng 9 tích hợp các mối quan tâm hành vi và bảo mật. Với mỗi sự kiện Event-B tương ứng với một nút đồ thị hoạt động được bảo vệ, điều kiện bảo vệ của sự kiện được tăng cường bằng một vị từ phân quyền được suy ra từ `RbacDomNode` liên kết.

Tính áp dụng của luật (Dòng 10–16). Dòng 10–16 định nghĩa tính áp dụng của các luật phân quyền và việc xây dựng các điều kiện bảo vệ phân quyền cho các chính sách từ chối. Để mã hóa đúng ngữ nghĩa kết hợp luật, chúng tôi áp dụng chiến lược `DENY_OVERRIDES`, theo đó việc thực thi một nút chỉ được cho phép khi tồn tại ít nhất một chính sách cho phép có thể áp dụng và không có chính sách từ chối có thể áp dụng nào.

Khởi tạo và kiểm chứng (Dòng 17–18). Dòng 17–18 sinh sự kiện khởi tạo, thiết lập một trạng thái ban đầu nhất quán cho mọi biến của máy trạng

Thuật toán 4.2 UDML2Event-B: Chuyển đổi UDML sang Event-B và kiểm chứng.

Đầu vào: Một mô hình UDML hợp lệ $M = \langle \mathcal{U}, \mathcal{S}, \mathcal{B}, \mathcal{R}, \mathcal{C} \rangle$.
Đầu ra : Một phát triển Event-B $\mathcal{D}(M) = \langle C, \mathsf{Mach} \rangle$ và thống kê chứng minh.

$\mathcal{T}_D$ — các hàm chuyển cho cấu trúc ; $\mathcal{T}_A$ — các hàm chuyển cho hành vi ; $\mathcal{T}_S$ — các hàm chuyển cho bảo mật ; $\mathsf{WF}(\cdot)$ — bộ kiểm tra hợp lệ DM hợp nhất ; $\mathsf{AnnRbac}(n)$ — `RbacDomNode` (tùy chọn) gắn với nút n

assert $\mathsf{WF}(M)$ `// Khởi tạo phát triển Event-B`

Khởi tạo một *context* Event-B rỗng $C \leftarrow \langle \mathsf{SETS}, \mathsf{CONSTS}, \mathsf{AXMS} \rangle$
Khởi tạo một *machine* Event-B rỗng $\mathsf{Mach} \leftarrow \langle \mathsf{VARS}, \mathsf{INVS}, \mathsf{EVTS} \rangle$
Liên kết Mach với C (tức là Mach **sees** C)

`// Dịch phần cấu trúc`

$(\mathsf{SETS}_D, \mathsf{CONSTS}_D, \mathsf{AXMS}_D) \leftarrow \mathcal{T}_D^{ctx}(M.\mathcal{S})$
$C.\mathsf{SETS} \leftarrow C.\mathsf{SETS} \cup \mathsf{SETS}_D$
$C.\mathsf{CONSTS} \leftarrow C.\mathsf{CONSTS} \cup \mathsf{CONSTS}_D$
$C.\mathsf{AXMS} \leftarrow C.\mathsf{AXMS} \cup \mathsf{AXMS}_D$
$\mathsf{Mach.INVS} \leftarrow \mathsf{Mach.INVS} \cup \mathcal{T}_D^{inv}(M.\mathcal{S})$

`// Dịch phần hành vi`

$C.\mathsf{SETS} \leftarrow C.\mathsf{SETS} \cup \{\texttt{NODES}, \texttt{EDGES}, \texttt{GRAPHS}\}$
$C.\mathsf{CONSTS} \leftarrow C.\mathsf{CONSTS} \cup \{src, tgt, nodesOf, edgesOf, initOf\}$
$C.\mathsf{AXMS} \leftarrow C.\mathsf{AXMS} \cup \mathcal{T}_A^{ctx}(M.\mathcal{B})$

Thêm các biến máy trạng thái cho thực thi: $\mathsf{Mach.VARS} \leftarrow \mathsf{Mach.VARS} \cup \{curG, pc\}$; Thêm các bất biến hành vi: $\mathsf{Mach.INVS} \leftarrow \mathsf{Mach.INVS} \cup \mathcal{T}_A^{inv}(M.\mathcal{B}, curG, pc)$; Sinh các sự kiện hành vi: $\mathsf{Mach.EVTS} \leftarrow \mathsf{Mach.EVTS} \cup \mathcal{T}_A^{evt}(M.\mathcal{B}, curG, pc)$

`// Dịch phần bảo mật (RBAC/SoD)`

$(\mathsf{SETS}_S, \mathsf{CONSTS}_S, \mathsf{AXMS}_S) \leftarrow \mathcal{T}_S^{ctx}(M.\mathcal{R})$
$C.\mathsf{SETS} \leftarrow C.\mathsf{SETS} \cup \mathsf{SETS}_S$
$C.\mathsf{CONSTS} \leftarrow C.\mathsf{CONSTS} \cup \mathsf{CONSTS}_S$
$C.\mathsf{AXMS} \leftarrow C.\mathsf{AXMS} \cup \mathsf{AXMS}_S$

Thêm các hằng $permAction \in \texttt{PERMISSIONS} \to \texttt{ACTIONS}$ và $permRes \in \texttt{PERMISSIONS} \to \texttt{RESOURCES}$; Thêm các tiên đề gán kiểu cho $permAction, permRes$; Thêm các biến trạng thái RBAC: $\mathsf{Mach.VARS} \leftarrow \mathsf{Mach.VARS} \cup \mathcal{T}_S^{vars}(M.\mathcal{R})$; Thêm các bất biến RBAC (bao gồm SoD): $\mathsf{Mach.INVS} \leftarrow \mathsf{Mach.INVS} \cup \mathcal{T}_S^{inv}(M.\mathcal{R})$; Thêm hằng (hoặc biến) $mapRes \in \texttt{RESOURCES} \rightharpoonup \texttt{AE}$; Thêm tiên đề/bất biến $\mathsf{Typing}(mapRes)$

`// Vị từ suy diễn dùng trong guard`

$EnabledRoles(sess) \triangleq active(sess)$
$EnabledPerms(sess) \triangleq \{p \mid \exists\, r \in active(sess) \cdot (r, p) \in pa\}$

foreach *event thực thi nút* $ev_n \in \mathsf{Mach.EVTS}$ *tương ứng với nút* $n \in ND$ **do**
  Đặt $Allow \leftarrow \varnothing$ và $Deny \leftarrow \varnothing$ **foreach** $rna \in Ann(n)$ **do**
    Tính $Applies(rna, sess) \equiv RolesSat(rna, sess) \wedge PermsSat(rna, sess) \wedge SoDSat(rna, sess)$ **if** $effect(rna) = ALLOW$ **then** $Allow \leftarrow Allow \cup \{Applies(rna, sess)\}$;
    else $Deny \leftarrow Deny \cup \{Applies(rna, sess)\}$;
  end foreach
  Định nghĩa $AllowApplies(n, sess) \equiv \bigvee Allow$; Định nghĩa $DenyApplies(n, sess) \equiv \bigvee Deny$; Tăng cường guard(ev_n) với $AllowApplies(n, sess) \wedge \neg DenyApplies(n, sess)$
end foreach

Sinh *INITIALISATION* để khởi tạo nhất quán mọi biến máy theo các bất biến: đặt $curG$ bằng thể hiện đồ thị được chọn (hoặc đồ thị mặc định) ; đặt $pc \leftarrow initOf(curG)$; khởi tạo $ua, pa, sess, active$ (và mọi biến trạng thái miền) sao cho mọi bất biến đều thỏa

`// Sinh và giải nghĩa vụ chứng minh`

Sinh các nghĩa vụ chứng minh $PO(\mathcal{D}(M))$; Giải $PO(\mathcal{D}(M))$ bằng các bộ chứng minh Rodin và ghi nhận thống kê chứng minh

return $(\mathcal{D}(M), PO\text{-statistics})$

thái. Cuối cùng, các nghĩa vụ chứng minh được sinh ra và được giải bằng nền tảng Rodin.

Để cho phép các DM có khả năng thực thi và hỗ trợ sinh tự động các tạo tác phần mềm, các mối quan tâm bảo mật cần được được ánh xạ 1-1 từ DSL ngoại sinh sang DSL nội sinh nhúng vào ngôn ngữ lập trình. Cụ thể, RBACDom đòi hỏi một cú pháp cụ thể cho phép các chính sách kiểm soát truy cập được biểu diễn một cách tường minh, tích hợp liền mạch với các DM và hành vi, đồng thời có thể được xử lý bởi các chuỗi công cụ hướng mô hình.

4.5 Tổng kết chương

Trong chương này, luận án đã trình bày phương pháp tổng hợp các ngôn ngữ chuyên biệt miền theo mối quan tâm vào một DM hợp nhất có khả năng thực thi theo DDD, cùng với một khung kiểm chứng ngữ nghĩa hình thức cho UDML. Phương pháp đề xuất cho phép tích hợp có hệ thống các mối quan tâm cấu trúc, hành vi và bảo mật trong một DM thống nhất. Trên cơ sở đó, chương đã xây dựng và tích hợp DSL bảo mật RBACDom vào UDML, đồng thời thực hiện kiểm chứng hình thức DM hợp nhất ở giai đoạn thiết kế nhằm đảm bảo tính nhất quán và tính đúng đắn ngữ nghĩa của các đặc tả bảo mật động trước khi sinh bản mẫu phần mềm.

Các kết quả nghiên cứu cốt lõi của chương này đã được công bố trong bài báo hội thảo quốc tế RIVF 2024 *[V3]*, được mở rộng trong SOICT 2025 *[V5]*, và tiếp tục phát triển theo hướng tích hợp khía cạnh bảo mật và kiểm chứng hình thức cho UDML trong bài báo tạp chí JCSCE 2026 *[V7]*.

Chương 5

SINH TỰ ĐỘNG BẢN MẪU PHẦN MỀM DỰA VÀO MÔ HÌNH MIỀN HỢP NHẤT

Chương này trình bày các thao tác và kỹ thuật sinh tự động bản mẫu phần mềm dựa vào mô hình miền hợp nhất trong khuôn khổ thiết kế hướng miền. Trọng tâm của chương là xây dựng các phép chuyển đổi có hệ thống nhằm thu hẹp khoảng cách giữa đặc tả yêu cầu và hiện thực phần mềm, đồng thời bảo đảm tính nhất quán ngữ nghĩa trong suốt quá trình chuyển đổi. Cụ thể, chương đề xuất phương pháp RM2UDM để chuyển đổi mô hình yêu cầu, được biểu diễn bằng các biểu đồ lớp và biểu đồ hoạt động UML/OCL, sang mô hình miền hợp nhất tuân thủ siêu mô hình UDML. Trên cơ sở đó, các kỹ thuật chuyển đổi tiếp theo được phát triển nhằm sinh đặc tả mô hình miền thực thi AGL^{+} và mã nguồn tương ứng thông qua tiếp cận chuyển đổi mô hình–sang–văn bản. Bên cạnh đó, chương cũng trình bày các thuật toán và luật chuyển đổi tiêu biểu, như AD2AGL, cho phép tích hợp hành vi vào mô hình miền và hỗ trợ sinh tự động bản mẫu phần mềm. Các kỹ thuật này được tổ chức trong một quy trình lặp, cho phép cập nhật và tinh chỉnh mô hình trên cơ sở phản hồi của chuyên gia miền. Thông qua đó, chương làm rõ bộ chuyển đổi từ mô hình yêu cầu đến bản mẫu phần mềm trong bối cảnh thiết kế hướng miền, đồng thời xác lập cơ sở phương pháp cho việc hiện thực, thực nghiệm và đánh giá ở các chương tiếp theo.

5.1 Giới thiệu

Trong quá trình phát triển phần mềm, việc xây dựng bản mẫu phần mềm đóng vai trò quan trọng trong việc diễn đạt ý tưởng, xác minh yêu cầu và cải thiện giao tiếp giữa các bên liên quan. Bản mẫu phần mềm cho phép kiểm tra và tinh chỉnh các tính năng trước khi triển khai, từ đó giảm thiểu rủi ro, tối ưu hóa quy trình phát triển và góp phần bảo đảm sản phẩm cuối cùng đáp ứng đúng nhu cầu người dùng với chất lượng mong muốn [117]. Vì vậy, việc tạo nhanh bản mẫu phần mềm từ đặc tả yêu cầu có ý nghĩa thiết thực đối với phát triển nhanh và kiểm soát chất lượng. Tuy nhiên, việc tự động hóa hoạt động này vẫn là một thách thức do khoảng cách ngữ nghĩa giữa các đặc tả yêu cầu bằng ngôn ngữ tự nhiên hoặc ngôn ngữ bán hình thức như UML/OCL [91] và mã nguồn dùng để hiện thực bản mẫu.

Nhiều nghiên cứu đã tập trung vào các phương pháp và công cụ sinh tự động bản mẫu phần mềm từ đặc tả yêu cầu. Trong [101], nhóm tác giả đề xuất sử dụng IFML để biểu diễn mô hình luồng tương tác, từ đó sinh tự động giao diện người dùng (*Graphical User Interface - GUI*). Trong [10], kỹ thuật sinh tự động GUI từ mô hình ca sử dụng được đề xuất. Các nghiên cứu [14, 131] sử dụng tiếp cận theo mô hình để sinh GUI phục vụ phát triển nhanh các ứng dụng Internet và các hệ thống điều khiển. Một số công trình khác đề xuất sinh GUI từ mô hình miền (Apache Isis [119] và OpenXava [95]), hoặc từ các biểu đồ UML như biểu đồ lớp, biểu đồ hoạt động, biểu đồ tuần tự và các đặc tả ca sử dụng [67, 84]. Gần đây, thiết kế hướng miền [39, 121] đã góp phần thu hẹp khoảng cách ngữ nghĩa giữa mô hình yêu cầu và hiện thực bản mẫu phần mềm. Tuy nhiên, để hướng tới sinh tự động bản mẫu phần mềm vẫn còn hai thách thức chính: (1) đặc tả chính xác mô hình yêu cầu hướng miền; và (2) xây dựng các bộ chuyển đổi để sinh tự động mã nguồn cài đặt bản mẫu từ mô hình yêu cầu.

Trong nghiên cứu trình bày tại Mục 3.3, hành vi miền đã được tích hợp vào mô hình miền nhằm hợp nhất các khía cạnh cấu trúc và hành vi. Tuy nhiên, việc xác định đặc tả ở mức trừu tượng cao dựa vào biểu đồ lớp và biểu đồ hoạt động UML, rồi chuyển đổi đặc tả này sang DCSL [70] và đặc tả AGL (gọi chung là đặc tả AGL^{+}) để sử dụng trong DDD nhằm sinh các bản mẫu phần mềm, vẫn là một thách thức đáng kể.

Đặc tả mức cao dựa vào biểu đồ lớp và biểu đồ hoạt động UML có thể được xem như một mô hình mô tả hành vi tổng thể của hệ thống, bao gồm luồng dữ liệu và luồng điều khiển. Trong kỹ thuật siêu mô hình hóa cho DSL [66], mô hình khái niệm của miền thường được xây dựng dưới dạng biểu đồ lớp UML/OCL, tương ứng với mô hình cú pháp trừu tượng và có thể được nhúng vào OOPL. UML [91] hỗ trợ xây dựng mô hình hướng đối tượng cho cú pháp trừu tượng, cũng như ký pháp cụ thể và ngữ nghĩa của ngôn ngữ mục tiêu [28]. Trong bối cảnh đó, mục tiêu trực tiếp của chương này là hỗ trợ sinh tự động đặc tả AGL^+ và bản mẫu phần mềm từ đặc tả yêu cầu mức cao.

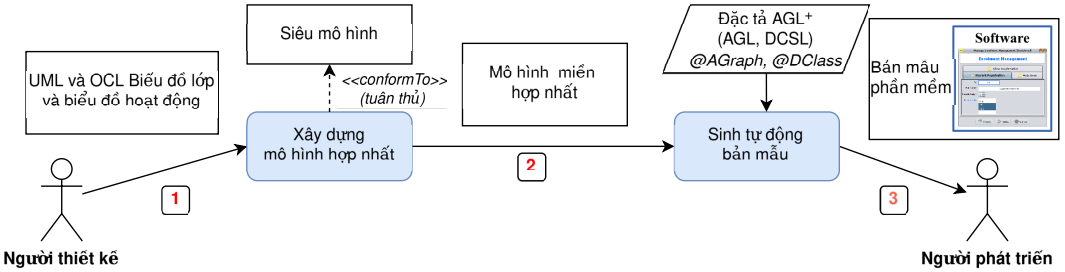
Trong chương này, luận án trình bày hai nội dung chính. Thứ nhất, phương pháp sinh tự động bản mẫu phần mềm dựa vào bộ chuyển đổi mô hình RM2UDM, trong đó mô hình yêu cầu đầu vào được chuyển thành mô hình miền hợp nhất UDM nhằm hỗ trợ tự động hóa quá trình tạo bản mẫu phần mềm. Thứ hai, kỹ thuật sinh đặc tả mô hình miền thực thi dựa vào bộ chuyển đổi AD2AGL kết hợp với chuyển đổi mô hình–sang–văn bản (M2T), sử dụng mô hình mức cao gồm biểu đồ lớp và biểu đồ hoạt động UML để sinh mã nguồn chứa đặc tả AGL^+, phục vụ hiện thực theo DDD trong ngôn ngữ lập trình chủ như Java.

Các mục còn lại của chương này được cấu trúc như sau. Mục 5.3 trình bày phương pháp sinh tự động bản mẫu phần mềm từ mô hình yêu cầu theo hướng RM2UDM. Mục 5.2 trình bày tổng quan phương pháp. Mục 5.4 trình bày kỹ thuật chuyển đổi từ đặc tả yêu cầu sang mô hình miền thực thi theo hướng AD2AGL/M2T. Cuối cùng, tổng kết chương được trình bày trong Mục 5.5.

5.2 Tổng quan phương pháp

Phần này trình bày tổng quan phương pháp sinh tự động bản mẫu phần mềm từ mô hình yêu cầu dựa vào các kỹ thuật chuyển đổi mô hình trong bối cảnh thiết kế hướng miền. Như minh họa trong Hình 5.1, phương pháp được tổ chức thành ba giai đoạn chính và vận hành theo quy trình lặp, bao gồm: xây dựng mô hình miền hợp nhất từ mô hình yêu cầu, sinh mô hình

miền thực thi và mã nguồn từ các đặc tả mức cao, và sinh bản mẫu phần mềm để tiếp nhận phản hồi từ chuyên gia miền.



Hình 5.1: Tổng quan phương pháp sinh tự động bản mẫu phần mềm dựa vào mô hình miền hợp nhất.

Giai đoạn 1 - Chuyển đổi mô hình yêu cầu đầu vào, gồm biểu đồ lớp và biểu đồ hoạt động UML/OCL, thành thể hiện mô hình miền hợp nhất UDM theo UDML. Kết quả của giai đoạn này là một biểu diễn trung gian hợp nhất, trong đó khía cạnh cấu trúc được đặc tả bằng DCSL, khía cạnh hành vi được đặc tả bằng AGL, đồng thời tích hợp các cấu hình mô-đun cần thiết phục vụ các bước sinh bản mẫu phần mềm tiếp theo.

Giai đoạn 2 - Sinh mô hình miền thực thi từ đặc tả mức cao thông qua bộ chuyển đổi AD2AGL kết hợp với kỹ thuật chuyển đổi mô hình–sang–văn bản. Với cùng đầu vào là biểu đồ lớp và biểu đồ hoạt động UML/OCL, giai đoạn này dẫn xuất đặc tả AGL^+, trong đó AGL biểu diễn khía cạnh hành vi, còn DCSL biểu diễn khía cạnh cấu trúc của mô hình miền. Trên cơ sở đặc tả AGL^+ thu được, kỹ thuật chuyển đổi mô hình–sang–văn bản tiếp tục được áp dụng để sinh mã nguồn và hiện thực các tạo tác phần mềm ở các bước tiếp theo.

Giai đoạn 3 - Sinh bản mẫu phần mềm từ các mô hình trung gian và đặc tả thực thi đã thu được ở các giai đoạn trước. Trên cơ sở mô hình miền thực thi, các cấu hình mô-đun và các đặc tả liên quan, hệ thống tiến hành sinh tự động bản mẫu phần mềm. Bản mẫu sinh ra được chuyển cho chuyên gia miền để đánh giá; nếu có phản hồi, mô hình sẽ được cập nhật và quy trình được lặp lại, ngược lại, khi bản mẫu đáp ứng yêu cầu, quá trình kết thúc.

5.3 Phương pháp sinh tự động bản mẫu phần mềm từ mô hình yêu cầu

Phần này trình bày phương pháp RM2UDM nhằm sinh tự động bản mẫu phần mềm từ mô hình yêu cầu. Ý tưởng cốt lõi của phương pháp là chuyển đổi một cách có hệ thống mô hình yêu cầu đầu vào sang một mô hình miền hợp nhất, từ đó tạo cơ sở để sinh mô hình miền thực thi tương ứng. Cách tiếp cận này góp phần thu hẹp khoảng cách ngữ nghĩa giữa đặc tả yêu cầu và hiện thực phần mềm trong bối cảnh thiết kế hướng miền. UDML được hiểu là ngôn ngữ/siêu mô hình dùng để biểu diễn mô hình miền hợp nhất, trong khi UDM (*Unified Domain Model*) được dùng để chỉ một mô hình miền hợp nhất cụ thể được xây dựng từ mô hình yêu cầu và được biểu diễn theo UDML. Theo đó, phương pháp RM2UDM nhận đầu vào là mô hình yêu cầu $RM = \langle CD, AD \rangle$, trong đó CD là biểu đồ lớp UML/OCL đặc tả khía cạnh cấu trúc và AD là biểu đồ hoạt động UML đặc tả khía cạnh hành vi; đầu ra là mô hình miền hợp nhất $UDM = \langle DM, AG, MC \rangle$, tuân thủ siêu mô hình *UDML*, trong đó DM là phần mô hình miền cấu trúc, AG là phần đồ thị hoạt động biểu diễn hành vi miền, và MC là phần cấu hình mô-đun phục vụ sinh bản mẫu phần mềm.

5.3.1 Xây dựng mô hình miền hợp nhất từ mô hình yêu cầu

Việc xây dựng thể hiện UDM từ mô hình yêu cầu theo UDML gồm hai thành phần: (1) Biểu diễn khía cạnh cấu trúc, tương ứng với DCSL; (2) Biểu diễn khía cạnh hành vi, tương ứng với AGL.

Biểu diễn khía cạnh cấu trúc. Khía cạnh cấu trúc của UDM được xây dựng từ biểu đồ lớp đầu vào và được biểu diễn dưới dạng mô hình DCSL kết hợp với cấu hình mô-đun. Thành phần này bảo toàn các lớp miền, các thuộc tính, các quan hệ kết hợp và các ràng buộc cấu trúc được đặc tả trong mô hình yêu cầu. Cụ thể:

- Một lớp miền hoạt động c_a được xác định tương ứng với mỗi biểu đồ hoạt động đầu vào nhằm biểu diễn ngữ cảnh thực thi của hành vi miền.
- Các lớp miền $c_1, \ldots, c_n$ được xác định tương ứng với các lớp trong biểu đồ lớp đầu vào.

- Với mỗi tập lớp thành phần $c_{i1}, \ldots, c_{ik} \in \{c_1, \ldots, c_n\}$ tham gia thực hiện các hành động miền, các liên kết tương ứng được bổ sung vào lớp hoạt động và các lớp liên quan để phản ánh các quan hệ cấu trúc cần thiết cho quá trình thực thi.
- Mỗi lớp miền sau khi được ánh xạ vào mô hình DCSL tiếp tục được gắn với một mô-đun cấu hình tương ứng nhằm hỗ trợ tổ chức phần mềm và sinh mã bản mẫu.

Biểu diễn khía cạnh hành vi. Khía cạnh hành vi của UDM được xây dựng bằng cách ánh xạ biểu đồ hoạt động đầu vào sang đồ thị hoạt động AGL. Trong đó, hành vi được tích hợp vào mô hình miền thông qua lớp hoạt động (*ActivityClass*), đóng vai trò trung tâm để liên kết các lớp dữ liệu, lớp điều khiển và các quan hệ thực thi. Các thành phần chính bao gồm:

- Lớp hoạt động: là lớp miền đại diện cho một hoạt động nghiệp vụ.
- Lớp dữ liệu: là lớp miền đại diện cho các kho dữ liệu hoặc thực thể dữ liệu tham gia vào hoạt động.
- Lớp điều khiển: là lớp nắm bắt trạng thái miền gắn với các nút điều khiển của biểu đồ hoạt động, chẳng hạn như nút quyết định, nút rẽ nhánh, nút hợp nhất hoặc nút đồng bộ.
- Lớp điều phối: là lớp đại diện cho một nhóm nhiệm vụ điều phối trong luồng thực thi.
- Các liên kết dành riêng cho hoạt động: là các liên kết được bổ sung giữa lớp hoạt động và các lớp dữ liệu, lớp điều khiển hoặc các lớp liên quan khác nhằm phản ánh đúng luồng điều khiển và luồng dữ liệu của biểu đồ hoạt động đầu vào.

Như vậy, UDM thu được là một mô hình miền hợp nhất cụ thể, trong đó cấu trúc và hành vi được tích hợp trong cùng một biểu diễn tuân thủ UDML. Đây là cơ sở trực tiếp để thực hiện bộ chuyển đổi RM2UDM.

5.3.2 Đặc tả bộ chuyển đổi mô hình RM2UDM

Bộ chuyển đổi RM2UDM được xây dựng nhằm tự động chuyển mô hình yêu cầu RM sang mô hình miền hợp nhất UDM. Về bản chất, đây là một phép

chuyển đổi mô hình–sang–mô hình (M2M), trong đó mô hình nguồn là RM và mô hình đích là một thể hiện UDM tuân thủ siêu mô hình UDML.

Thuật toán 5.1 Thuật toán RM2UDM: Sinh UDM từ RM

Đầu vào:

- CD: Biểu đồ lớp UML/OCL đặc tả khía cạnh cấu trúc của mô hình yêu cầu;
- AD: Biểu đồ hoạt động UML đặc tả khía cạnh hành vi của mô hình yêu cầu.

Đầu ra : UDM = ⟨DM, AG, MC⟩: mô hình miền hợp nhất tuân thủ UDML.

```
1  Khởi tạo mô hình hợp nhất rỗng UDM
   Khởi tạo ba thành phần DM, AG, MC của UDM
2  // Pha 1: Ánh xạ khía cạnh cấu trúc từ CD sang DM và MC
   foreach class ∈ CD do
3  |  genDCSL(class, DM) ; genModule(class, MC)
4  end foreach
5  // Pha 2: Ánh xạ khía cạnh hành vi từ AD sang AG
   genActivityClass(AD, DM); initActivityGraph(AD, AG)
6  foreach node ∈ AD do
7  |  if node là nút hành động then
8  |  |  genActionNode(node, AG)
   |  |   bindRefCls(node, DM) bindOutCls(node, DM)
9  |  else if node là nút điều khiển kiểu Decision then
10 |  |  genDecisionNode(node, AG)
11 |  else if node là nút điều khiển kiểu Sequential then
12 |  |  genSequentialNode(node, AG)
13 |  else if node là nút điều khiển kiểu Fork then
14 |  |  genForkNode(node, AG)
15 |  else if node là nút điều khiển kiểu Merge then
16 |  |  genMergeNode(node, AG)
17 |  else if node là nút điều khiển kiểu Join then
18 |  |  genJoinNode(node, AG)
19 |  else
20 |  |  genCoordinateNode(node, AG)
21 end foreach
22 foreach edge ∈ AD do
23 |  genEdge(edge, AG) ;              // Sinh các cạnh của đồ thị hoạt động
24 end foreach
25 // Pha 3: Tích hợp hành vi với mô hình miền
   foreach node là nút hành động trong AG do
26 |  bindActSeq(node, DM)
27 end foreach
28 genActivityLinks(DM, AG)
29 Kiểm tra tính phù hợp của UDM với siêu mô hình UDML
   return UDM
```

Thuật toán 5.1 được tổ chức theo ba pha. Pha thứ nhất xây dựng phần cấu trúc của mô hình đích từ biểu đồ lớp đầu vào, bao gồm các lớp miền, thuộc tính, quan hệ, ràng buộc và các mô-đun cấu hình tương ứng. Pha thứ hai xây dựng phần hành vi từ biểu đồ hoạt động, trong đó các nút hành

động và nút điều khiển được ánh xạ sang các thành phần tương ứng của đồ thị hoạt động AGL. Pha thứ ba tích hợp hai phần này thông qua các tham chiếu như `refCls`, `outCls` và `actSeq`, đồng thời bổ sung các liên kết hoạt động cần thiết để thu được một mô hình miền hợp nhất hoàn chỉnh. Trên cơ sở ba pha nêu trên, RM2UDM được đặc tả bằng một tập các luật chuyển đổi chính như sau.

Luật 1. Mỗi biểu đồ hoạt động trong mô hình yêu cầu được ánh xạ sang một lớp hoạt động (`ActivityClass`) trong UDM. Lớp này là phần tử trung tâm để biểu diễn đồ thị hoạt động tương ứng.

Luật 2. Mỗi nút hành động trong biểu đồ hoạt động được ánh xạ sang một thể hiện `Node` trong UDM. Các tham chiếu `refCls` và `outCls` được xác định trên cơ sở lớp miền tương ứng trong biểu đồ lớp.

Luật 3. Mỗi nút điều khiển trong mô hình yêu cầu được ánh xạ sang một mẫu miền (`DomainPattern`) tương ứng trong UDM, chẳng hạn như `Decision`, `Fork`, `Merge`, `Join`, `Sequential` hoặc `Coordinate`.

Luật 4. Với mỗi `Node` được tạo từ một nút hành động, chuỗi hành động thực thi được biểu diễn thông qua thuộc tính `actSeq` và được gắn với lớp miền tương ứng nhằm đồng bộ trạng thái hành vi với trạng thái miền.

Luật 5. Mỗi lớp, thuộc tính và ràng buộc OCL trong biểu đồ lớp được ánh xạ sang các thành phần DCSL tương ứng trong UDM, qua đó bảo toàn khía cạnh cấu trúc của mô hình đầu vào.

Luật 6. Mỗi lớp miền trong mô hình yêu cầu được ánh xạ sang một mô-đun cấu hình tương ứng trong UDM nhằm hỗ trợ tổ chức phần mềm và sinh bản mẫu.

Như vậy, các luật chuyển đổi trên là cơ sở hình thức cho thuật toán RM2UDM. Trên phương diện phương pháp, chúng xác định mối tương ứng giữa các phần tử của mô hình nguồn và mô hình đích; trên phương diện hiện thực, chúng làm nền tảng cho việc xây dựng bộ chuyển đổi và công cụ hỗ trợ ở Chương 6.2.4.

5.4 Chuyển đổi từ đặc tả yêu cầu sang mô hình miền hợp nhất

Phần này trình bày một kỹ thuật chuyển đổi từ đặc tả yêu cầu mức cao sang mô hình miền thực thi trong bối cảnh thiết kế hướng miền. Khác với Mục 5.3, vốn tập trung vào chuyển đổi mô hình yêu cầu sang thể hiện UDM theo UDML, mục này xem xét một hướng chuyển đổi chuyên biệt nhằm sinh đặc tả AGL$^+$ của mô hình miền. Cách tiếp cận này hiện thực hóa mô hình miền theo một dạng biểu diễn có thể thực thi, trong đó khía cạnh hành vi được biểu diễn bằng AGL, còn khía cạnh cấu trúc được biểu diễn bằng DCSL. Mục này tập trung vào việc xây dựng một phép chuyển đổi có hệ thống để dẫn xuất đồng thời cấu hình hành vi miền và cấu trúc miền từ đặc tả yêu cầu đầu vào, sau đó tích hợp chúng trong cùng một biểu diễn thực thi. Kết quả của phép chuyển đổi này đóng vai trò là bước trung gian giữa mô hình yêu cầu và các bản mẫu phần mềm được sinh ra.

5.4.1 Bộ chuyển đổi AD2AGL

Bộ chuyển đổi AD2AGL kế thừa cùng mô hình đầu vào đã được xác định ở Mục 5.3, tức mô hình yêu cầu $RM = \langle CD, AD \rangle$, trong đó AD là biểu đồ hoạt động UML dùng để nắm bắt hành vi miền, còn CD là biểu đồ lớp UML/OCL dùng để nắm bắt các lớp miền, thuộc tính, quan hệ và các ràng buộc cấu trúc. Từ đặc tả mức cao này, AD2AGL sinh ra đặc tả mô hình miền thực thi $DM = AGL^+$, trong đó phần AGL biểu diễn cấu hình đồ thị hoạt động của hành vi miền, còn phần DCSL biểu diễn khía cạnh cấu trúc của mô hình miền. Theo đó, AD2AGL có thể được xem là một phép chuyển đổi chuyên biệt, sử dụng cùng đầu vào như RM2UDM nhưng hướng tới một biểu diễn thực thi của mô hình miền. Trong bối cảnh này, AGL được dùng để biểu diễn hành vi miền thông qua các đồ thị hoạt động, các nút hoạt động và các hành động mô-đun, trong khi AGL$^+$ là đặc tả hoàn chỉnh hình thành từ việc kết hợp phần AGL với phần DCSL tương ứng. Kết quả chuyển đổi này đóng vai trò là cầu nối giữa mô hình yêu cầu và biểu diễn thực thi của phần mềm, đồng thời có thể được chú thích và hiện thực trong Java thông qua các phần tử như `@AGraph`, `@ANode`, `@MAct`, `@DClass`, `@DAttr`, v.v.

Thuật toán 5.2 mô tả quy trình tổng quát để sinh đặc tả AGL$^+$ từ đặc tả mức cao. Thuật toán được tổ chức theo hai pha chính. Pha thứ nhất duyệt biểu đồ hoạt động để sinh phần AGL của mô hình miền. Pha thứ hai duyệt biểu đồ lớp UML/OCL để sinh phần DCSL và hợp nhất vào cùng mô hình đích. Thuật toán được thực hiện theo chiều sâu (DFS) trên đồ thị hoạt động, nhằm thăm toàn bộ các nút và sinh ra các phần tử AGL tương ứng.

Thuật toán 5.2 Thuật toán AD2AGL: Sinh đặc tả AGL$^+$ từ đặc tả mức cao

Đầu vào:

- AD: Biểu đồ hoạt động UML mô tả hành vi hệ thống;
- CD: Biểu đồ lớp UML/OCL đặc tả mô hình miền.

Đầu ra : DM: Đặc tả AGL$^+$ của mô hình miền thực thi.

```
1  Khởi tạo DM rỗng. AG ← đồ thị hoạt động được suy ra từ AD

2  // Pha 1: Sinh phần AGL từ AD bằng duyệt DFS
   foreach node ∈ AG theo thứ tự duyệt DFS do
3      if node là nút Decision then
4          genDecisionNode(node, DM)
5      else if node là nút Sequential then
6          genSequentialNode(node, DM)
7      else if node là nút Fork then
8          genForkNode(node, DM)
9      else if node là nút Merge then
10         genMergeNode(node, DM)
11     else if node là nút Join then
12         genJoinNode(node, DM)
13     else
14         genActionNode(node, DM)
15     end if
16 end foreach
17 // Pha 2: Sinh phần DCSL từ CD
   foreach class ∈ CD do
18     genDCSL(class, DM)
19 end foreach
20 return DM
```

Trước hết, thực hiện khởi tạo một *DM* rỗng để chứa đặc tả AGL$^+$ của mô hình miền và trỏ tới nút khởi đầu của *AD*.

Thứ hai, thuật toán sử dụng phương pháp duyệt theo chiều sâu trên *AD* để thăm tất cả các nút trong đồ thị hoạt động. Một đặc tả AGL sẽ được bổ sung mỗi khi một nút hành động mới được thăm. Theo chuẩn UML [91], biểu đồ hoạt động có thể được xem như một đồ thị. Cụ thể, đồ thị hoạt động *AG* [124] được biểu diễn hình thức dưới dạng một bộ sáu phần tử $AG = \langle N, E, eve, gua, var, obj \rangle$, trong đó N là tập các nút, E là tập các cạnh,

eve là tập các sự kiện, *gua* là tập các điều kiện gác, *var* là tập các biến cục bộ, và *obj* là tập các đối tượng. Một nút có thể thuộc hai loại: `ActionNode` (bao gồm các nút: `action`, `acceptEvent` và `sendSignal`) và `ControlNode` (bao gồm các nút: `initial`, `final`, `sequential`, `decision`, `merge`, `fork` và `join`).

Thứ ba, thuật toán kiểm tra từng nút trong *AD*, bao gồm toàn bộ các nút hành động và nút điều khiển, và gọi các hàm con tương ứng để sinh đặc tả AGL. Khi duyệt đến một nút hiện tại và xác định nút kế tiếp của nó, thuật toán kiểm tra loại nút kế tiếp và gọi hàm sinh AGL tương ứng với tham số đầu vào là nút hiện tại (*curNode*), đồng thời cập nhật đặc tả AGL của mô hình miền. Mỗi hàm này bổ sung một đặc tả AGL theo mô hình cú pháp cụ thể dạng văn bản rên chú thích dành cho AGL được định nghĩa trong Mục 3.3. Các hàm được sử dụng bao gồm: `genDecisionNode(curNode, DM)`, `genForkNode(curNode, DM)`, `genMergeNode(curNode, DM)`, `genJoinNode(curNode, DM)`, `genSequentialNode(curNode, DM)`.

Thứ tư, thuật toán tiếp tục duyệt tới nút kế tiếp và lặp lại bước thứ ba cho đến khi tất cả các nút được thăm (tức là đạt tới *finalNode*).

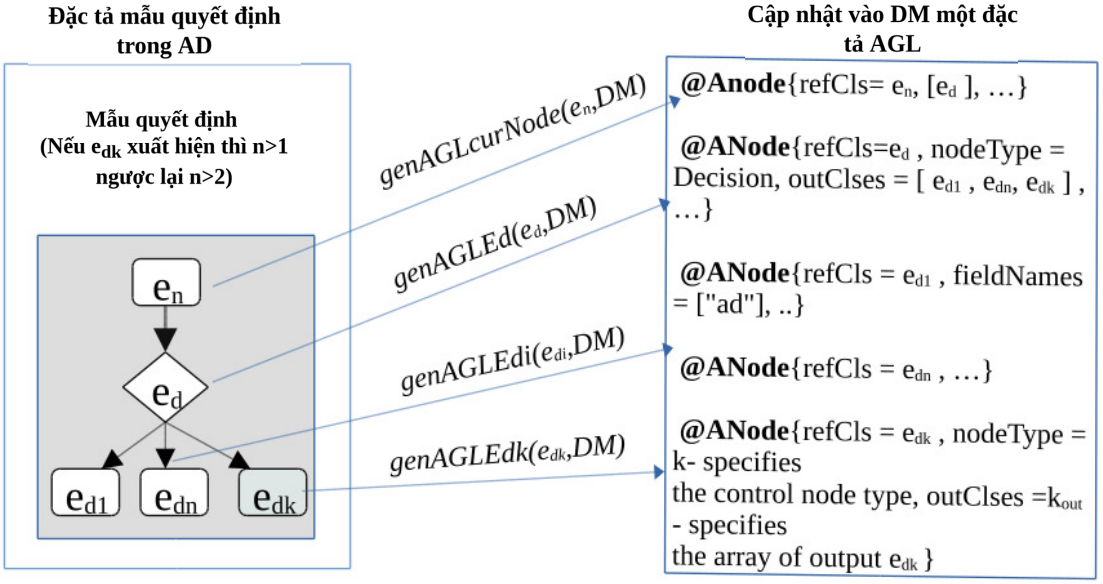
Thứ năm, thuật toán sinh đặc tả DCSL từ *CD* bằng cách duyệt toàn bộ lớp trong biểu đồ lớp và cập nhật phần đặc tả DCSL tương ứng của mô hình miền.

Để làm rõ cơ sở hình thức của thuật toán AD2AGL, Bảng 5.1 trình bày ánh xạ giữa các phần tử hình thức hóa của đồ thị hoạt động và các thành phần trong siêu mô hình biểu diễn tương ứng. Bảng này làm rõ mối liên hệ giữa mô hình hành vi hình thức và đặc tả AGL$^+$. Bảng 5.1 cho thấy phép chuyển đổi AD2AGL không chỉ là một ánh xạ cú pháp đơn thuần, mà dựa vào việc chuyển các thành phần của đồ thị hoạt động hình thức sang các phần tử tương ứng của mô hình miền thực thi, giúp bảo đảm cấu hình AGL thu được vẫn phản ánh đúng cấu trúc hành vi của mô hình đầu vào.

Ví dụ hàm xử lý nút quyết định. Trong hàm `genDecisionNode(curNode, DM)`, đầu vào là e_n—nút hiện tại, có một nút quyết định kế tiếp là e_d trong *AD*, như được minh họa trong Hình 5.2. Hàm `genDCSL(class, DM)` sinh ra đặc tả DCSL từ *CD* (đặc tả DCSL theo cách tiếp cận dựa vào chú thích được định nghĩa trong [70]) bằng cách duyệt tất cả các lớp trong *CD* và cập nhật đặc tả DCSL vào *DM*.

Bảng 5.1: Ánh xạ các phần tử hình thức hóa sang siêu mô hình

Phần tử trong siêu mô hình	**Thành phần trong đặc tả hình thức**
activity graph	AG
activityNode	N
activityEdge	E
events	eve
guard condition	gua
variable	var
objects	obj
nút hiện tại	*curNode*
duyệt nút kế tiếp	*curNode.Next*
mô hình miền nắm bắt AGL	*DM*
nút thực thi	*genDecisionNode(curNode,DM)* *genSequentialNode(curNode,DM)* *genForkNode(curNode,DM)* *genMergeNode(curNode,DM)* *genJoinNode(curNode,DM)*



Hình 5.2: Đặc tả nút quyết định trong AD và ký hiệu của nó trong thuật toán.

5.4.2 Sinh đặc tả mô hình miền thực thi (AGL^+)

Sau khi xác lập phép chuyển đổi AD2AGL ở mức khái niệm, bước tiếp theo là sinh biểu diễn AGL^+ dưới dạng có thể thực thi thông qua cách tiếp cận

chuyển đổi mô hình–sang–văn bản (M2T). Trong bối cảnh này, M2T có vai trò chuyển các phần tử của mô hình đầu vào thành các đặc tả văn bản tương ứng của AGL^+, qua đó tạo cơ sở cho việc sinh mã và tích hợp vào môi trường thực thi phần mềm. Về đầu vào, phép sinh M2T tiếp nhận đặc tả mức cao gồm *AD* và *CD*, hoặc mô hình trung gian thu được sau bước AD2AGL, trong đó đã xác định các nút hoạt động, các nút điều khiển, các cạnh, các lớp miền và các ràng buộc liên quan. Về đầu ra, phép sinh tạo ra đặc tả AGL^+ ở dạng văn bản, bao gồm:

- phần AGL biểu diễn cấu hình đồ thị hoạt động thông qua các phần tử như `@AGraph`, `@ANode`, `@MAct`;
- phần DCSL biểu diễn mô hình lớp miền thông qua các phần tử như `@DClass`, `@DAttr`, `@DOpt`, `@DAssoc`.

Trên cơ sở đó, các quy tắc chuyển đổi có thể được tổ chức thành ba nhóm chính:

- Nhóm quy tắc sinh AGL từ biểu đồ hoạt động: nhóm này ánh xạ các nút hoạt động và nút điều khiển trong *AD* sang các phần tử AGL tương ứng. Các quy tắc trong nhóm này hiện thực trực tiếp các thủ tục như `genDecisionNode`, `genSequentialNode`, `genForkNode`, `genMergeNode` và `genJoinNode`.
- Nhóm quy tắc sinh DCSL từ biểu đồ lớp UML/OCL: nhóm này ánh xạ các lớp, thuộc tính, phương thức, quan hệ và ràng buộc trong *CD* sang các thành phần DCSL tương ứng, thực hiện vai trò bổ sung phần cấu trúc cho AGL^+.
- Nhóm quy tắc hợp nhất thành AGL^+: nhóm này kết hợp phần AGL và phần DCSL vào cùng một mô hình miền thực thi, bảo đảm rằng các hành động nghiệp vụ được gắn với các lớp miền tương ứng và có thể tiếp tục được ánh xạ sang các tạo tác thực thi.

Như vậy, M2T đóng vai trò là bước hiện thực hóa của bộ chuyển đổi AD2AGL, biến các kết quả chuyển đổi ở mức mô hình thành đặc tả AGL^+ ở dạng văn bản có thể được sử dụng trong chuỗi sinh tạo tác phần mềm. Chi tiết hiện thực các mẫu bằng công cụ được trình bày ở Chương 6.2.4.

5.5 Tổng kết chương

Chương này đã trình bày các kỹ thuật chuyển đổi mô hình phục vụ sinh tự động bản mẫu phần mềm trong bối cảnh thiết kế hướng miền. Cụ thể, luận án đề xuất phương pháp RM2UDM để chuyển mô hình yêu cầu UML/OCL sang mô hình miền hợp nhất UDM tuân thủ UDML, đồng thời trình bày kỹ thuật AD2AGL kết hợp với đặc tả AGL^+ nhằm sinh mô hình miền thực thi và các tạo tác phần mềm tương ứng. Các kết quả này thiết lập một bộ chuyển đổi từ mô hình yêu cầu đến bản mẫu phần mềm, qua đó tăng cường sự liên kết giữa yêu cầu nghiệp vụ, mô hình miền và hiện thực phần mềm.

Cách tiếp cận chuyển đổi đặc tả mức cao sang mô hình miền thực thi trong DDD của luận án được công bố lần đầu trong bài báo hội thảo quốc tế KSE 2023 *[V2]*; phương pháp sinh tự động bản mẫu phần mềm từ mô hình yêu cầu được công bố lần đầu trong bài báo hội thảo quốc gia FAIR 2024 *[V4]*.

Chương 6

THỰC NGHIỆM VÀ ĐÁNH GIÁ

Chương này trình bày việc áp dụng và đánh giá các phương pháp đề xuất của luận án thông qua các ca nghiên cứu điển hình, bao gồm CourseMan, OrderMan, ProcessMan và hệ thống OJS. Các hệ thống này đại diện cho các miền ứng dụng khác nhau với sự đa dạng về cấu trúc, hành vi và các mối quan tâm liên quan. Mục tiêu của chương là kiểm chứng tính khả thi và hiệu quả của cách tiếp cận thông qua việc xây dựng biểu diễn miền hợp nhất, thiết lập liên kết ngữ nghĩa và thực hiện các chuyển đổi mô hình phục vụ kiểm chứng và tự động hóa phát triển phần mềm.

Các ca nghiên cứu được sử dụng với vai trò khác nhau tùy theo nội dung đánh giá. Cụ thể, CourseMan, OrderMan và ProcessMan được dùng để đánh giá các kỹ thuật biểu diễn và chuyển đổi mô hình, trong khi OJS được sử dụng để minh họa và đánh giá tích hợp bảo mật dựa trên vai trò và kiểm chứng hình thức. Trên cơ sở đó, các kết quả thực nghiệm được phân tích nhằm làm rõ tính khả thi, tính nhất quán ngữ nghĩa và khả năng áp dụng của phương pháp đề xuất.

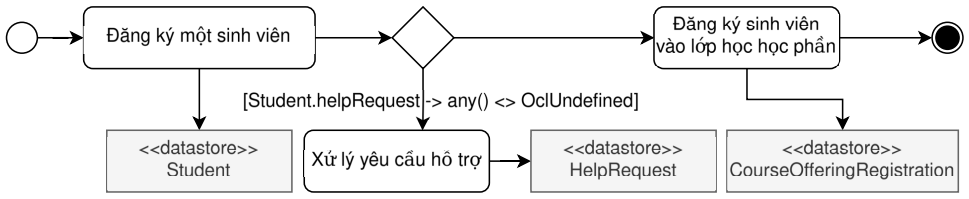
6.1 Giới thiệu

Luận án sử dụng hệ thống quản lý khóa học (CourseMan) làm ca nghiên cứu xuyên suốt cho thực nghiệm và đánh giá. Miền vấn đề của hệ thống được thể hiện thông qua mô hình miền trong Hình 1.2. Biểu đồ lớp UML trong hình này mô tả khía cạnh cấu trúc của mô hình miền CourseMan. Để đặc tả các quy tắc nghiệp vụ mà UML không thể biểu đạt đầy đủ, các ràng buộc OCL

được sử dụng; ràng buộc minh họa trong hình đảm bảo rằng tổng số tín chỉ sinh viên đăng ký không vượt quá mức tối đa theo chương trình đào tạo.

Mục tiêu của luận án là tích hợp mô hình miền UML/OCL này vào bối cảnh DDD. Tuy nhiên, điều này đặt ra thách thức vì DDD không chỉ yêu cầu mô hình hóa cấu trúc mà còn đòi hỏi mô hình miền phải có ngữ nghĩa vận hành rõ ràng, tức là có khả năng thực thi hoặc diễn giải như một chương trình. Như đã phân tích trong Mục 3.3 và Mục 3.2, mô hình miền trong DDD cần cung cấp các góc nhìn phù hợp cho các bên liên quan, từ chuyên gia miền đến lập trình viên, thông qua các hiện vật có thể kiểm chứng và khai thác trực tiếp.

Hình 6.1 trình bày biểu đồ hoạt động UML cho quy trình quản lý đăng ký, bao gồm đăng ký sinh viên, đăng ký học phần (`CourseOffering`) và xử lý các yêu cầu hỗ trợ trong quá trình đăng ký.



Hình 6.1: Biểu đồ hoạt động UML mô tả hoạt động quản lý đăng ký lớp học phần.

Để tạo ra một phiên bản có thể thực thi của hệ thống, các hành vi miền cần được tích hợp chặt chẽ với mô hình miền cấu trúc trong Hình 1.2. Trong cách tiếp cận truyền thống, hành vi được mô tả bằng UML và tính nhất quán giữa mô hình hành vi và mô hình miền thường chỉ được đảm bảo ở tầng cài đặt. Như một hướng tiếp cận thay thế, luận án đề xuất xem hành vi miền như một phần mở rộng của mô hình miền thiết yếu, từ đó hình thành một mô hình miền hợp nhất. Cách tiếp cận này tuân theo phương pháp DDD được đề xuất trong [70], trong đó mô hình miền hợp nhất phải đáp ứng các yêu cầu về tính khả thi kỹ thuật, hiệu năng và khả năng hiểu được bởi chuyên gia miền.

Công cụ được trình bày trong chương này nhằm minh họa cách áp dụng các kỹ thuật AGL và CAP trong thực tiễn, đồng thời khảo sát khả năng sử dụng của phương pháp đề xuất trong việc phát triển phần mềm cho các miền vấn đề thực tế. Bên cạnh đó, chương này cũng mô tả công cụ hỗ trợ

đã được xây dựng, làm rõ các quyết định thiết kế chính và các công nghệ được sử dụng.

Mục tiêu của thực nghiệm là chỉ ra rằng AGL và CAP vừa có khả năng biểu đạt cần thiết, vừa có tính khả dụng trong thực tiễn. Việc đánh giá được tiến hành theo các hướng dẫn nghiên cứu tình huống được đề xuất trong [104] nhằm đảm bảo tính chặt chẽ về phương pháp luận. Ngoài ra, các tiêu chí đánh giá DSL được đề xuất trong [118] cũng được sử dụng để phân tích AGL và CAP, tập trung vào các câu hỏi nghiên cứu sau:

- RQ1. Làm thế nào có thể định nghĩa ngữ nghĩa vận hành hình thức cho các mô hình miền hợp nhất trong UDML?
- RQ2. Làm thế nào các mối quan tâm xuyên suốt động, tiêu biểu là RBAC, có thể được đặc tả dưới dạng các DSL dựa trên chú thích và được tích hợp vào các mô hình miền hợp nhất?
- RQ3. Mức độ biểu đạt của CAPs trong việc mô hình hóa các khái niệm miền so với các phương pháp DDD hiện có là như thế nào?
- RQ4 CAP có thể được áp dụng ở mức độ nào trong thực tiễn phát triển phần mềm?
- RQ5. Phương pháp tích hợp khía cạnh hành vi vào mô hình miền hiệu quả như thế nào so với các phương pháp DDD hiện có?
- RQ6. Mức độ nỗ lực cần thiết để định nghĩa một mô hình miền hợp nhất bằng aDSL (bao gồm DCSL và AGL) nhằm sinh phần mềm theo DDD là bao nhiêu?
- RQ7. Làm thế nào có thể tích hợp một mối quan tâm nền tảng vào mô hình miền?
- RQ8. Làm thế nào có thể biểu diễn miền của mối quan tâm, vốn có thể được xem như một ngôn ngữ chuyên biệt miền dựa trên chú thích (aDSL)?
- RQ9. Làm thế nào các DSL theo mối quan tâm có thể được hợp nhất một cách có hệ thống vào một cây cú pháp trừu tượng hợp nhất?

- RQ10. Những cơ chế hợp thành ngôn ngữ và các phương pháp có sự hỗ trợ của công cụ nào là phù hợp để tạo điều kiện cho việc tích hợp và tiến hóa các mối quan tâm không đồng nhất trong các mô hình miền mô-đun và có khả năng mở rộng?

Phần tiếp theo trình bày cấu trúc của công cụ hỗ trợ trong Mục 6.2. Các kết quả thực nghiệm và thảo luận đánh giá hiệu quả của các phương pháp đề xuất được trình bày trong Mục 6.3.

6.2 Công cụ hỗ trợ

Xây dựng công cụ hỗ trợ cho kỹ thuật tích hợp ràng buộc và hành vi vào mô hình miền.

6.2.1 Công cụ hỗ trợ kỹ thuật tích hợp ràng buộc vào mô hình miền

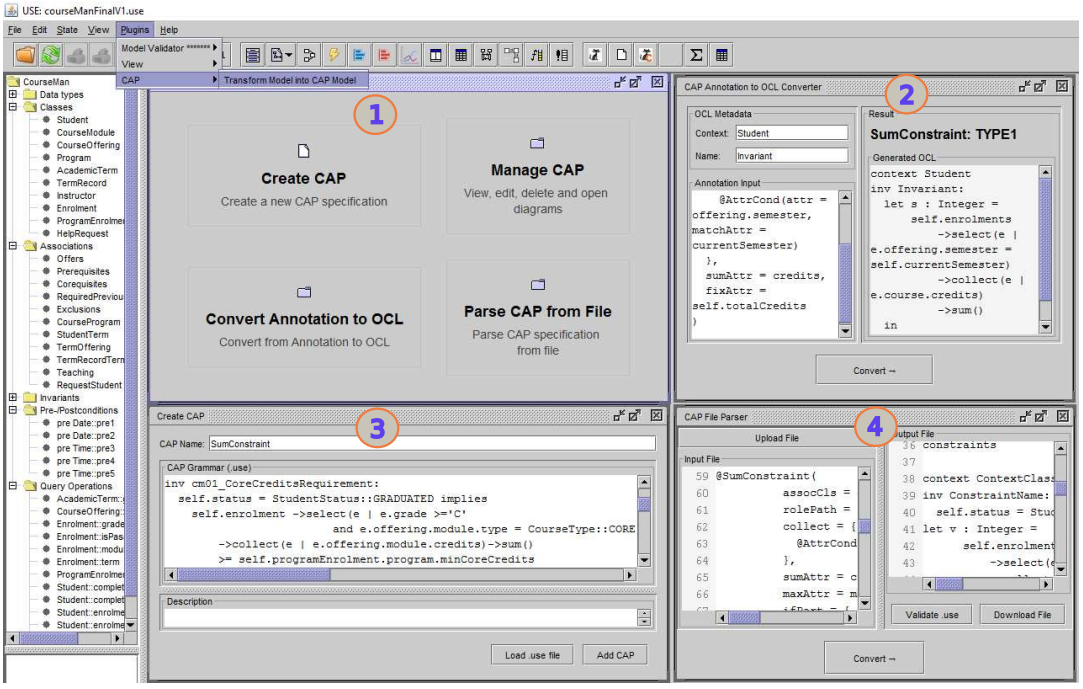
Kiến trúc công cụ: Mở rộng CAP/UDML của USE

Trong tiểu mục này, luận án trình bày phương pháp xây dựng công cụ hỗ trợ, được gọi là CAP/UDML. Công cụ này được hiện thực như một phần mở rộng của môi trường đặc tả dựa trên UML USE (UML-based Specification Environment) [23], mở rộng khả năng hỗ trợ sẵn có của USE về mô hình hóa cấu trúc UML và kiểm chứng ràng buộc OCL bằng cách bổ sung cơ chế đặc tả CAP dựa trên chú thích và tái tạo tự động các bất biến.

Để hỗ trợ việc tích hợp các CAPs vào mô hình miền hợp nhất (UDM), CAP/UDML mở rộng ngữ pháp và pipeline xử lý của USE nhằm: (i) nhận diện các chú thích CAP, (ii) liên kết chúng với các khuôn mẫu CAP trong danh mục, và (iii) tự động sinh các bất biến OCL tương ứng theo ánh xạ sinh hình thức được định nghĩa trong Mục 3.2.2.3. Các bất biến được sinh ra sau đó được kiểm chứng bằng bộ máy OCL của USE nhằm đảm bảo tính đúng đắn cú pháp và tính nhất quán cấu trúc với mô hình miền.

Cài đặt của công cụ được công bố dưới dạng mã nguồn mở trên GitHub[1].

[1] https://github.com/vinhskv/Tool-ThesisVinh/tree/main/CAP_UDML-main/CAP_UDML



Hình 6.2: Công cụ CAP/UDML hỗ trợ quản lý CAP và tái tạo OCL

Hình 6.2 cung cấp cái nhìn tổng quan về kiến trúc của công cụ CAP/UDML, minh họa các lớp mở rộng được xây dựng trên nền USE cũng như quy trình từ phân tích chú thích đến tái tạo OCL và tích hợp vào UDM.

Ở phía trên bên trái (nhãn (1)) của Hình 6.2, giao diện hiển thị bốn chức năng chính của công cụ CAP/UDML, bao gồm: (i) Tạo CAP, (ii) Chuyển đổi chú thích độc lập thành CAP, (iii) Phân tích các chú thích từ tệp của USE (`.use`), và (iv) Quản lý CAP: mỗi CAP có thể bao gồm nhiều kiểu khác nhau; chức năng này cho phép người dùng định nghĩa và quản lý các kiểu này trong một CAP đã được xác định.

Ở phía trên bên phải (nhãn (2)) của Hình 6.2, giao diện hiển thị chức năng **Chuyển đổi chú thích độc lập thành CAP**. Chức năng này tập trung vào việc phân tích các chú thích độc lập, tức là các chú thích không gắn với một ngữ cảnh mô hình cụ thể. Để chuyển đổi các chú thích này thành ràng buộc OCL, người dùng cần xác định thủ công ngữ cảnh (ví dụ: lớp áp dụng như `Student` hoặc `CourseModule`) và cung cấp tên bất biến, đóng vai trò là định danh cho ràng buộc OCL được sinh ra.

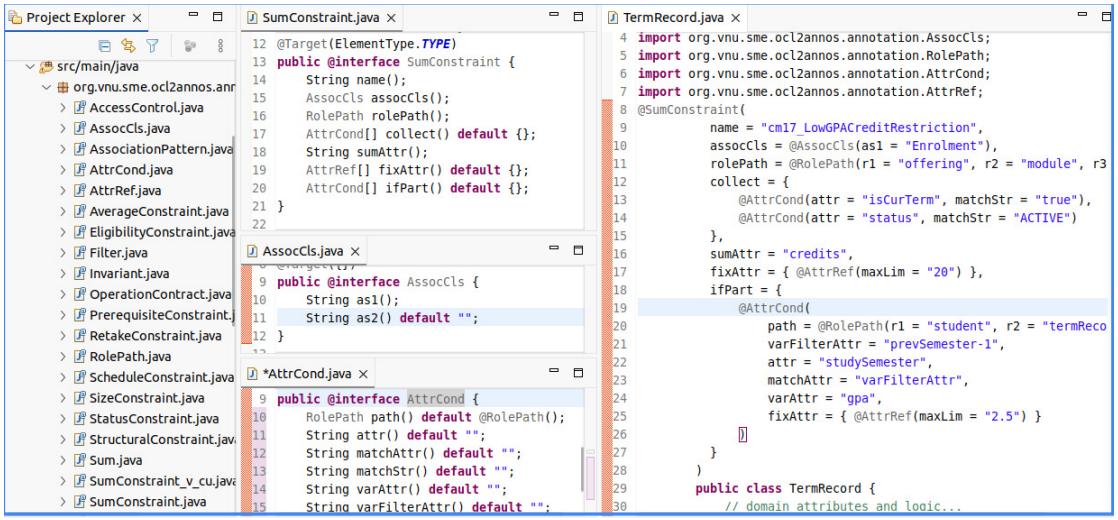
Ở phía dưới bên trái (nhãn (3)) của Hình 6.2, giao diện hiển thị chức năng **Tạo CAP**. Chức năng này cho phép người dùng định nghĩa một kiểu CAP

mới. Khi tạo CAP, người dùng cần chỉ định tên CAP (ví dụ: SumConstraint) và cung cấp một tệp `".use"` tương ứng mô tả cấu trúc sơ đồ lớp liên quan đến CAP đó. Hệ thống cho phép kiểm tra tệp `".use"` nhằm đảm bảo ngữ pháp và cú pháp tuân thủ đặc tả của USE.

Ở phía dưới bên phải (nhãn (4)) của Hình 6.2, giao diện hiển thị chức năng **Phân tích các chú thích từ tệp `".use"` File**. Chức năng này xử lý các chú thích đã tồn tại trong một tệp mô hình `".use"`. Khi tệp được nạp, plugin sẽ phân tích các chú thích cùng với ngữ cảnh mô hình tương ứng, tái tạo các ràng buộc OCL và sinh ra một tệp `".use"` mới, trong đó các chú thích được tự động chuyển thành các ràng buộc này. Các ràng buộc được tái tạo sau đó được tích hợp vào UDM. Sau khi sinh, tệp `".use"` kết quả có thể được kiểm chứng để đảm bảo tính đúng đắn về ngữ pháp và cú pháp.

Sinh bản mẫu phần mềm

Sinh bản mẫu phần mềm được minh họa trong Hình 6.3, thông qua việc xây dựng bằng Spring Boot và khung JDA [71]. Khung này được phát hành dưới dạng mã nguồn mở trên GitHub[2] Dựa trên UDM, JDA tự động sinh các phần mềm thực thi, bao gồm các lớp miền, lô-gic kiểm tra ràng buộc được suy ra từ các bất biến OCL đã được tái tạo, cấu hình lưu trữ dữ liệu, và các thành phần giao diện người dùng.

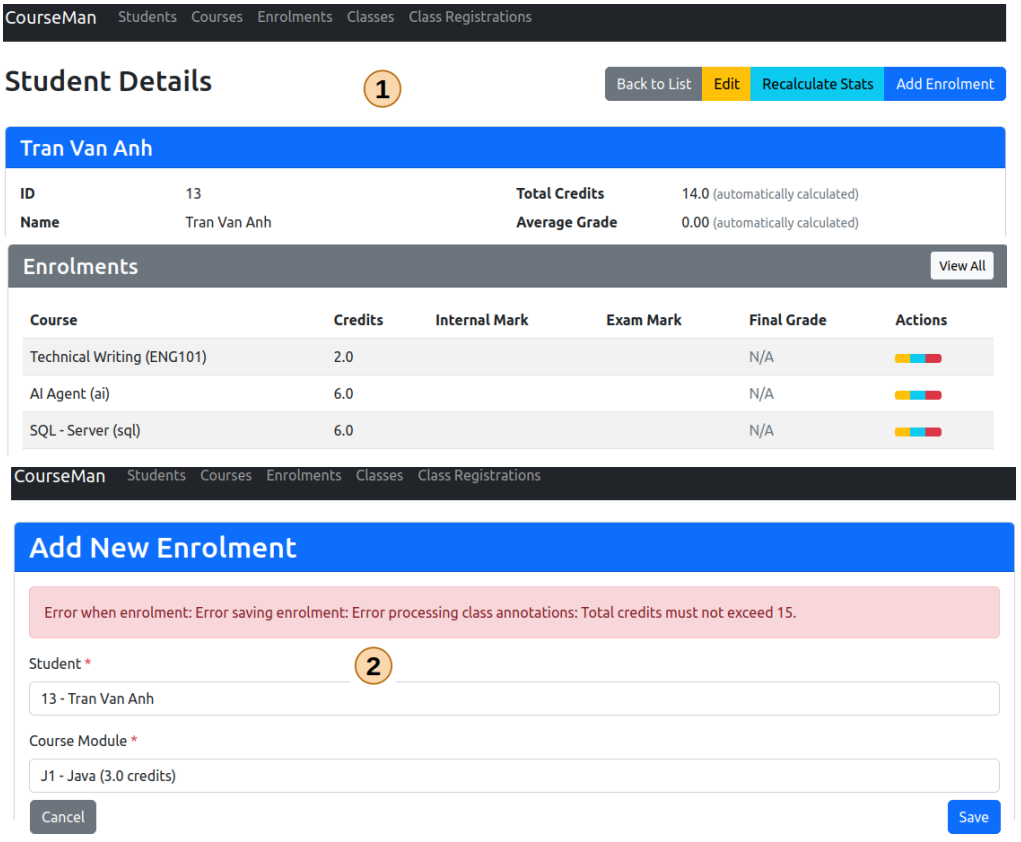


Hình 6.3: Hiện thực công cụ và khả năng sử dụng của khung làm việc CAP.

[2] https://github.com/vinhskv/Tool-ThesisVinh/tree/main/CAP_UDML-main/frameWorkGenCode

Giao diện bên trái của Hình 6.3 hiển thị danh sách các lớp mô-đun và các lớp miền tương ứng trong mô hình cấu trúc, mỗi lớp được gắn với `@SumConstraint` và `@PrerequisiteConstraint` theo đặc tả của CAP (như minh họa ở vùng giao diện giữa).

Ứng dụng CourseMan được sinh tự động với giao diện đồ họa (GUI) được trình bày trong Hình 6.4. Ví dụ này minh họa cơ chế kiểm tra ràng buộc tại thời gian chạy nhằm đảm bảo quy tắc giới hạn tín chỉ: tổng số tín chỉ mà một sinh viên đăng ký không được vượt quá 15. Ở phần trên của hình, mục (1) cho thấy một trường hợp hợp lệ, trong đó sinh viên "Trần Văn Anh" đã đăng ký 14 tín chỉ; trong khi mục (2) minh họa lỗi kiểm tra ràng buộc khi hệ thống phát hiện việc đăng ký vượt quá giới hạn cho phép. Khi thực hiện thao tác lưu để ghi nhận đăng ký môn học mới, hệ thống sẽ tự động kích hoạt cơ chế kiểm tra ràng buộc theo phương pháp mà đã đề xuất.



Hình 6.4: GUI của phần mềm CourseMan được sinh tự động bởi công cụ.

6.2.2 Công cụ hỗ trợ kỹ thuật tích hợp khía cạnh hành vi vào mô hình miền

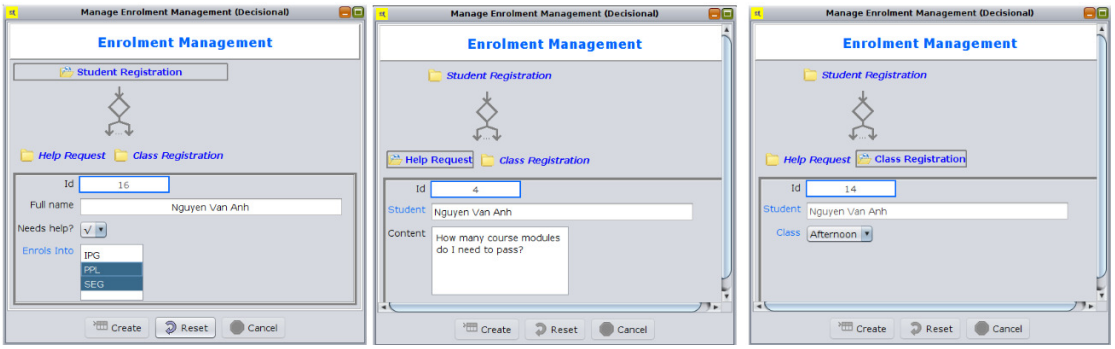
Trong phần này, trình bày minh họa phương pháp bằng mẫu quyết định (*Decisional*). Mẫu là một biến thể của mô hình hợp nhất dành cho hoạt động quản lý ghi danh của CourseMan. Một ví dụ về mẫu bao gồm mô hình hợp nhất đã được cấu hình và một hoặc nhiều giao diện phần mềm (GUI).

Mẫu quyết định

Phần dưới của Hình 3.7 minh họa cách mẫu này được áp dụng cho hoạt động quản lý ghi danh của CourseMan. Thực hiện cài đặt cụ thể như sau: $\mathtt{C_a}$ = `EnrolmentMgmt`, $\mathtt{C_d}$ = `Student`, `D` = `DHelpOrSClass`, $n = 2$, $\mathtt{C_1}$ = `HelpRequest`, và $\mathtt{C_2}$ = `CourseOffering`. Nút điều khiển c_k không được chỉ định.

Hình 6.5 trình bày ba ảnh chụp GUI của ví dụ: GUI thứ nhất dành cho việc đăng ký sinh viên; GUI thứ hai và thứ ba lần lượt dành cho hai trường hợp sinh viên có yêu cầu hỗ trợ và không có yêu cầu hỗ trợ. Giao diện của hoạt động chứa giao diện của cả ba hành động trong các "thẻ" riêng biệt.

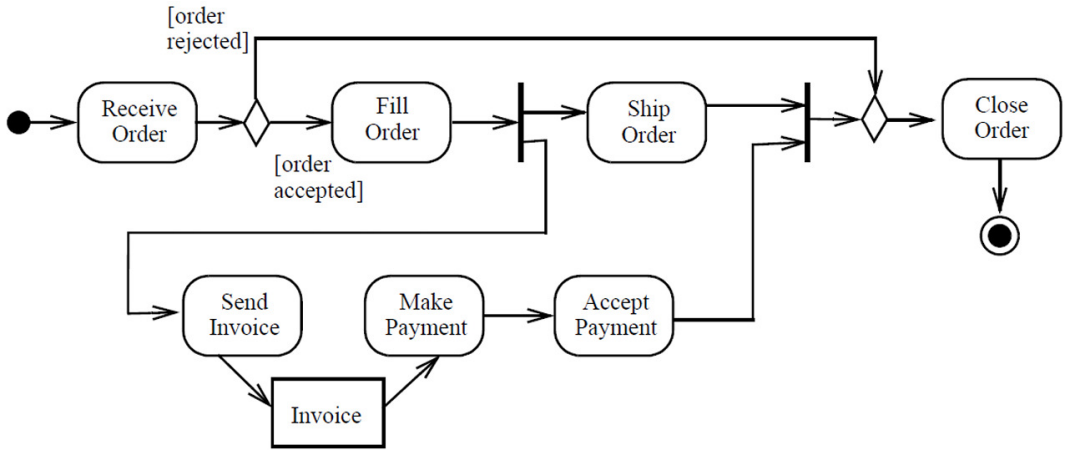
Ở cả hai nhánh quyết định, đối tượng `Student` được tạo từ hành động đầu tiên được truyền sang hành động kế tiếp và được hiển thị trong trường dữ liệu của thuộc tính `student` của lớp miền tương ứng.



Hình 6.5: Biểu diễn theo mẫu quyết định của hoạt động quản lý ghi danh.

Một nghiên cứu tình huống phức tạp khác cũng được trình bày—miền quản lý đơn hàng (OrderMan)—được điều chỉnh từ đặc tả OMG/UML [91, p. 396]. Mục đích là minh họa khả năng áp dụng AGL trong thực tiễn và khảo sát cách phương pháp đã đề xuất có thể được sử dụng để phát triển

phần mềm cho các miền vấn đề trong thế giới thực. Ngoài ra, phần này còn trình bày công cụ hỗ trợ được sử dụng, tập trung giải thích các quyết định thiết kế chính và công nghệ được áp dụng.

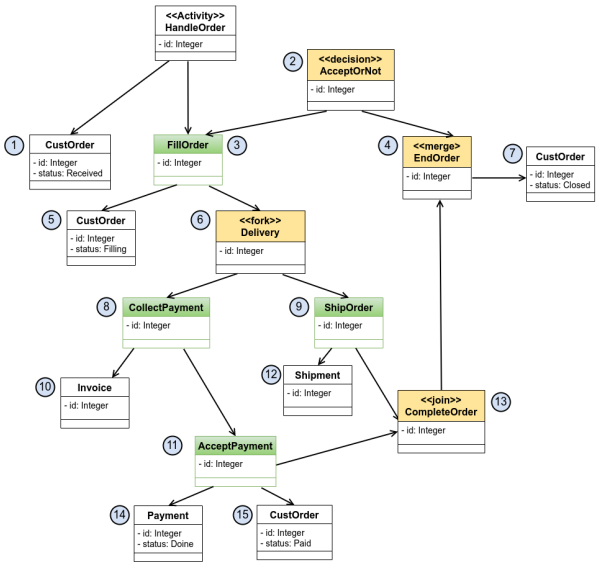


Hình 6.6: Biểu đồ hoạt động UML cho quy trình xử lý đơn hàng, được trích từ [91, p. 369].

Hình 6.6 minh họa biểu đồ hoạt động UML mô tả quy trình xử lý đơn hàng trong OrderMan. Để tạo ra mô hình miền hợp nhất cho OrderMan, như được trình bày trong Hình 6.7, thực hiện áp dụng toàn bộ các mẫu hành vi miền. Hình 6.7(A) trình bày một đồ thị hoạt động, trong đó các nút được gán số thứ tự tương ứng với hoạt động `HandleOrder`. Hình cũng mô tả một phần của mô hình lớp hợp nhất, bao gồm một lớp hoạt động (`HandleOrder`), bốn lớp dữ liệu chính (`CustOrder`, `Invoice`, `Shipment`, `Payment`), bốn lớp điều khiển (`AcceptOrNot`, `Delivery`, `CompleteOrder`, `EndOrder`), và bốn lớp còn lại liên quan đến các coordinator nodes (`FillOrder`, `CollectPayment`, `ShipOrder`, `AcceptPayment`).

Các nút điều phối được sử dụng để cung cấp một cái nhìn đầy đủ về một nhóm nhiệm vụ. Chẳng hạn, `FillOrder` (nút 3) điều phối hai nhiệm vụ: `UpdateOrder` (nút 5) và `DeliveryOrder` (nút 6). `FillOrder` chỉ làm nhiệm vụ điều phối để đảm bảo rằng `UpdateOrder` được thực hiện trước `DeliveryOrder`. Nó không đóng góp dữ liệu vào luồng xử lý, nhưng cho phép người dùng quan sát và thực thi luồng nhiệm vụ trên giao diện người dùng.

Hình 6.7(B) minh họa cách mỗi nút trong đồ thị hoạt động được ánh xạ đến một mô-đun tương ứng (so với lớp miền được tham chiếu bởi `refCls`), các nút kết quả (cũng dựa trên `refCls`), và các đối tượng `ModuleAct` đặc tả



1:AG (HandleOrder)

Node	label	nodeType	refCls	serviceCls	outNodes	actSeq
1	MOrder	-	CustOrder	default	[2:Node]	[1:MAct]
2	MAcceptOrNot	Decision	AcceptOrNot	-	[3:Node, 4:Node]	-
3	MFillOrder	Coordinator	FillOrder	default	[5:]	[2:, 3:, 4:]
4	MEndOrder	Merge	EndOrder	-	[7:]	-
5	MOrder	-	CustOrder	default	[6:]	[5:]
6	MDelivery	Fork	Delivery	default	[8:, 9:]	[2:, 4:]
7	MOrder	-	CustOrder	default		[6:, 7:]
8	MCollectPayment	Coordinator	CollectPayment	default	[10:]	[2:, 3:, 4:]
9	MShipOrder	Coordinator	ShipOrder	default	[12:]	[2:, 3:]
10	MInvoice	-	Invoice	default	[11:]	[2:, 3:]
11	MAccept Payment	Coordinator	AcceptPayment	default	[14:]	[2:, 4:, 8:]
12	MShipment	-	Shipment	default	[13:]	[2:, 3:]
13	MComplete Order	Join	CompleteOrder	-	[4:]	-
14	MPayment	-	Payment	default	[15:]	[2:, 8:]
15	MOrder	-	CustOrder	default	[13:]	[9:, 5:]

MAct

Id	actName	postStates	filterType	fieldNames
1	newObject	{Created}		
2	newObject	{NewObject}		
3	setDataFieldValues	{Created}		{"order"}
4	createObject	{Created}		
5	showObject	{Updated}		
6	showObject			
7	updateObject	{Updated}		
8	setDataFieldValues	{Created}		{"invoice"}
9	filterInput		FilterCustOrder FromPayment	

Hình 6.7: (A: Trái) Đồ thị hoạt động với các nút được gán nhãn bằng các lớp hoạt động và lớp thành phần; (B: Trên–phải) Các đối tượng `Node`; (C: Dưới–phải) Các đối tượng `ModuleAct` được tham chiếu bởi các `Node`.

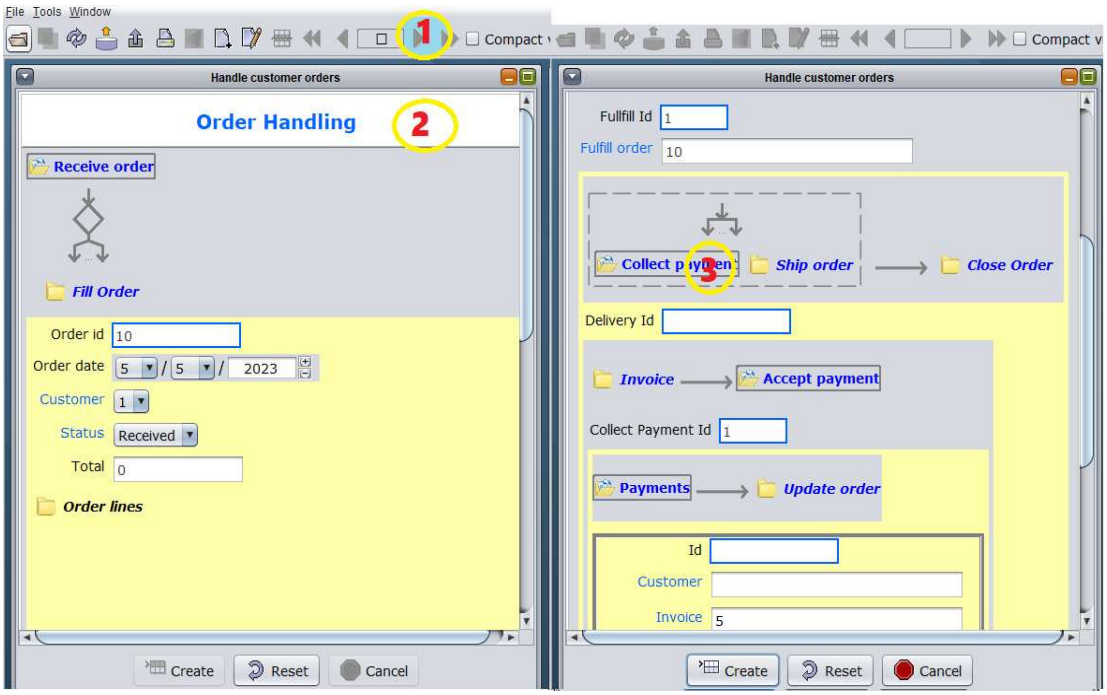
các SAA cho hành vi của mô-đun. Các đối tượng `ModuleAct` này, cùng với các SAA tương ứng, được liệt kê trong Hình 6.7(C).

Để phát triển phần mềm OrderMan với giao diện GUI, như thể hiện trong Hình 6.8, mô hình lớp hợp nhất — tích hợp đặc tả AGL cho mô hình hoạt động — cần được hiện thực hóa bằng mã Java. Việc hiện thực cho OrderMan có thể được tìm thấy trong kho mã nguồn[3]. Việc hiện thực phương pháp được minh họa trong Hình 6.9, sử dụng một công cụ hỗ trợ được xây dựng trên JDA, một khung phần mềm Java mà đã trình bày trong [70]. Công cụ này có thể truy cập tại kho mã nguồn[4]

Để sinh phần mềm từ mô hình miền hợp nhất đầu vào, lập trình viên cần mã hóa mô hình miền hợp nhất bằng DCSL và AGL, tạo ra một chương trình Java bao gồm hai thành phần chính: (1) Mô hình hợp nhất DCSL, bao gồm các lớp thành phần và lớp hoạt động; (2) Đặc tả AGL cho các đồ thị hoạt động gắn với các lớp hoạt động.

[3] https://github.com/vinhskv/Tool-ThesisVinh/tree/main/examples/orderman

[4] https://github.com/vinhskv/Tool-ThesisVinh

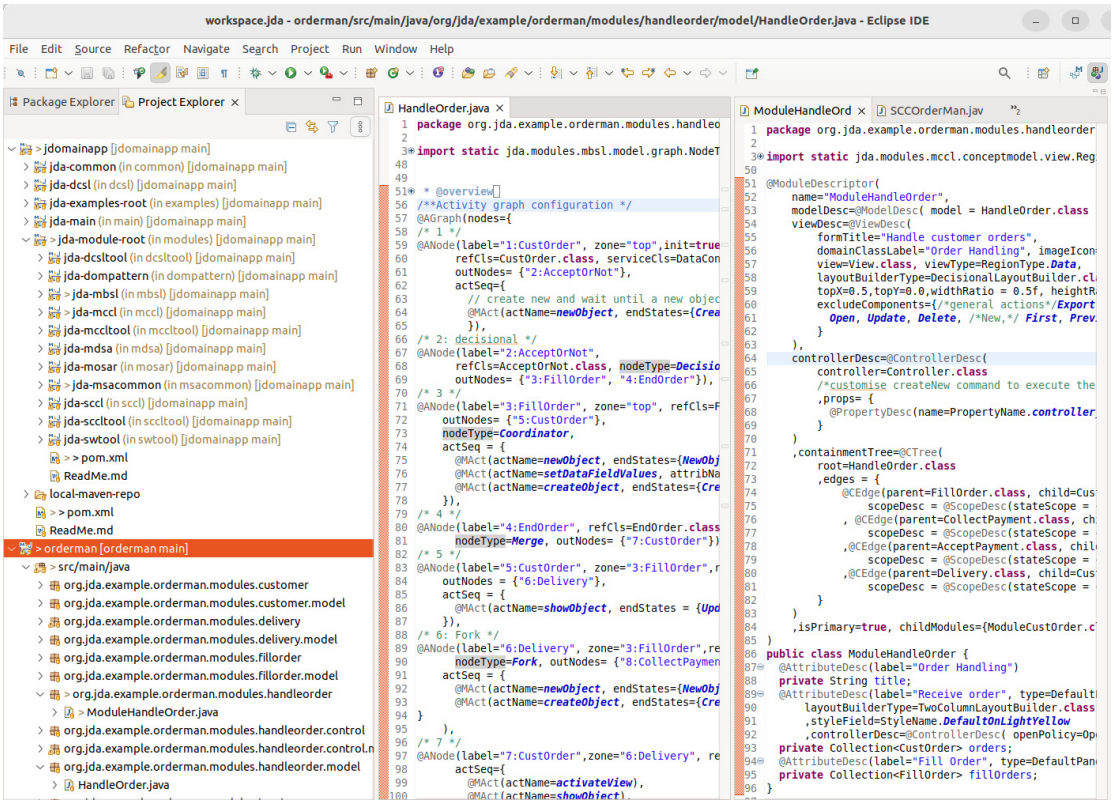


Hình 6.8: Giao diện người dùng của OrderMan được tạo ra bởi công cụ.

Cần lưu ý rằng cả đặc tả DCSL và AGL đều được mã hóa trong Java, như được minh họa ở hai khung giữa và phải của Hình 6.9.

Công cụ bao gồm ba thành phần chính: bộ quản lý mô hình, bộ quản lý giao diện và bộ quản lý đối tượng. Quản lý mô hình chịu trách nhiệm đăng ký và cung cấp mô hình lớp hợp nhất cho các thành phần khác. Cửa sổ bên trái của Hình 6.9 hiển thị danh sách các lớp mô-đun và các lớp miền tương ứng, được gắn chú thích `AGraph` nhằm biểu diễn đặc tả AGL, như được trình bày trong cửa sổ ở giữa. Quản lý giao diện tự động sinh giao diện người dùng của phần mềm và xử lý tương tác của người dùng dựa trên mô tả cấu hình của các mô-đun được đặc tả trong MOSA [72]. Ví dụ, lớp mô-đun `ModuleHandleOrder` chứa lớp hoạt động `HandleOrder` được đặc tả bằng mô tả mô-đun trong MCCL, như thể hiện trong cửa sổ bên phải của hình. Cuối cùng, quản lý bộ đối tượng thời gian chạy cho mỗi lớp miền và cung cấp một thành phần lưu trữ đối tượng tổng quát hỗ trợ cả lưu trữ dựa trên tệp và cơ sở dữ liệu quan hệ. Mô hình dữ liệu quan hệ được sinh tự động từ mô hình lớp hợp nhất khi phần mềm chạy lần đầu tiên.

Xét tình huống cụ thể với mô-đun `ModuleHandleOrder` và lớp hoạt động `HandleOrder`, được hiện thực như trong Listing 6.1. Khi phần mềm chạy, một thể hiện của mô-đun `ModuleHandleOrder` được gọi.



Hình 6.9: Minh họa việc hiện thực và khả năng sử dụng AGL dựa trên nền tảng JDA.

```
    /**Activity graph configuration in AGL /
    @AGraph(nodes={...
        / 14 */
        @ANode(label="14:Payment", zone="11:AcceptPayment",
        refCls=Payment.class, serviceCls=DataController.class,
        outNodes={"15:CustOrder"},
        actSeq={
            @MAct(actName=newObject, endStates={NewObject}),
            @MAct(actName=setDataFieldValues, attribNames={"
  invoice"},
            endStates={Created})
        }), ...
    })
    /**END: activity graph configuration */
    public class HandleOrder {...}
```

Đặc tả 6.1: Lớp hoạt động `HandleOrder` trong Java

Dựa trên mô tả cấu hình của mô-đun (trình bày trong cửa sổ bên phải của Hình 6.9), một đối tượng `ActivityModel` được tạo dưới dạng tổ hợp giữa đối

tượng `HandleOrder` và đồ thị hoạt động được mô tả bằng AGL tương ứng với lớp hoạt động này.

Nhờ cơ chế chú thích trong Java, đối tượng `HandleOrder` có thể được xử lý như một đối tượng `AGraph`, cho phép nó biểu diễn và quản lý đồ thị hoạt động. Đối tượng `AGraph` này cho phép mô-đun `ModuleHandleOrder` xử lý từng `ANode` của nó — chẳng hạn `ANode` tương ứng với nút 14 trong đoạn mã 6.1 — cũng như lớp miền được tham chiếu bởi `ANode` đó, trong ví dụ này là lớp `Payment`.

Cách AGL và các chú thích đi kèm được hiện thực trong Java. Đặc tả AGL của một lớp hoạt động được biểu diễn dưới dạng chú thích ngay trên lớp đó trong mã Java. Ví dụ, đoạn mã 6.1 minh họa cách chú thích `AGraph` được sử dụng để biểu diễn đặc tả AGL của lớp hoạt động `HandleOrder`. Tất cả các chú thích của AGL, như tóm tắt trong Hình 3.9, đều phải được định nghĩa trong Java. Đoạn mã 6.2 cung cấp ví dụ về định nghĩa chú thích `AGraph`, và các chú thích khác trong AGL cũng được định nghĩa tương tự.

```
1  package jda.modules.mbsl.model.graph.meta;
2  import java.lang.annotation.*;
3  import jda.modules.mbsl.model.graph.ActivityGraph;
4  @Retention(RetentionPolicy.RUNTIME)
5  @Target(value=java.lang.annotation.ElementType.TYPE)
6  @Documented
7  public @interface AGraph {
8      ANode[] nodes();
9  }
```

Đặc tả 6.2: Hiện thực Java của chú thích `AGraph`

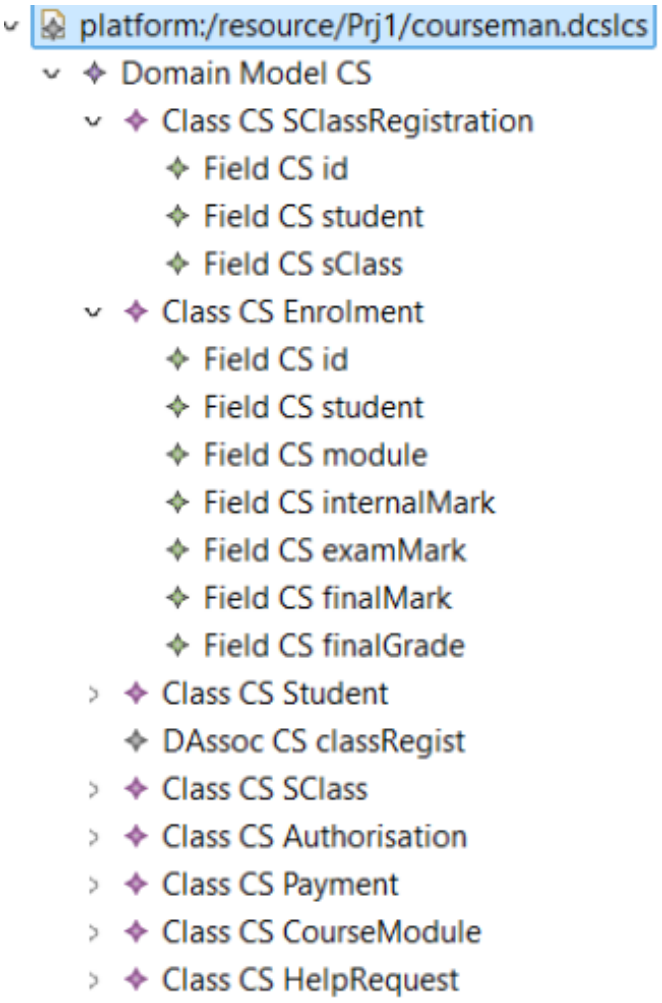
6.2.3 Công cụ hỗ trợ phương pháp tích hợp các mối quan tâm vào mô hình miền

Trong phần này trình bày công cụ hỗ trợ có tên UDMM và đánh giá kết quả thực nghiệm phương pháp đề xuất trên nghiên cứu điển hình CourseMan

Công cụ hỗ trợ tích hợp theo phương pháp siêu mô hình

Công cụ hỗ trợ UDMM bao gồm ba hoạt động chính: (1) định nghĩa và kết hợp các DSL theo mối quan tâm để tích hợp vào mô hình UDML; (2) sinh mã nguồn; và (3) tạo phần mềm hoặc bản mẫu, theo phương pháp được đề xuất trong Hình 4.1.

Việc xây dựng công cụ tập trung vào xác định các thành phần DSL cần biểu diễn đồ họa (lớp, thuộc tính, quan hệ), định nghĩa siêu mô hình UDML bằng EMF nhằm đảm bảo tính chặt chẽ và khả năng mở rộng, và thiết kế giao diện kéo–thả trực quan bằng Sirius. Trên cơ sở đó, cây cú pháp đồ họa được xây dựng để tổ chức và biểu diễn trực quan các thành phần của DSL.



Hình 6.10: Tạo giao diện kéo thả tuân thủ cú pháp trừu tượng của UDML.

Tích hợp với quy trình phát triển phần mềm: Mô hình đồ họa do Sirius tạo ra có thể được chuyển đổi thành mã thực thi, hỗ trợ trực tiếp quá trình phát triển ứng dụng.

Quá trình ánh xạ từ mô hình sang mã nguồn Java được định nghĩa thông qua các luật chuyển đổi trong các tệp **.mtl** thuộc Acceleo. Các luật này quy định cách các thành phần trong mô hình dcslcs được chuyển thành mã nguồn Java tương ứng. **Lớp (ClassCS)** trong mô hình được ánh xạ thành các lớp Java với khai báo thuộc tính, phương thức và các mối quan hệ. **Thuộc tính (FieldCS)** được ánh xạ thành các biến thành viên trong lớp Java với các phương thức getter và setter tương ứng. **Phương thức (MethodCS)** được ánh xạ thành các phương thức, hàm tương ứng. **Liên kết (DAssocCS)** được chuyển đổi thành quan hệ giữa các đối tượng, sử dụng kiểu dữ liệu phù hợp như danh sách (List) hoặc tham chiếu (Reference). **Các kiểu dữ liệu (PrimitiveType, ReferenceTypeCS)** được ánh xạ sang các kiểu dữ liệu tương ứng trong Java

Sau khi xây dựng, nhà thiết kế có thể thực hiện việc kéo thả để tạo mô hình. Công cụ tiến hành thực nghiệm trên CourseMan Hình 1.2, được biểu diễn bằng biểu đồ lớp UML cùng với các ràng buộc OCL. Sử dụng giao diện kéo thả tạo mô hình và thực hiện chuyển đổi mô hình sang mô hình mô hình miền hợp nhất dạng cụ thể (gần với ngôn ngữ OOPL) có thể tạo mã bằng công cụ ATL Hình 6.10. Giúp nâng cao khả năng sử dụng và hiểu biết về hệ thống là cú pháp cụ thể dạng đồ họa. Giúp các bên liên quan, bao gồm nhà phát triển, kiến trúc sư phần mềm và chuyên gia miền, dễ dàng tiếp cận và phân tích mô hình, hỗ trợ trực quan hóa các thành phần của hệ thống một cách rõ ràng.

Mô hình miền hợp nhất dạng cụ thể được tạo ra chuyển sang bước tiếp theo là sinh mã nguồn tự động, sử dụng Acceleo để ánh xạ các thành phần trong mô hình thành mã nguồn. Một phần của các luật chuyển thể hiện trong Hình 6.11, giúp đảm bảo rằng mô hình có thể được sử dụng trực tiếp trong môi trường lập trình.

```
[template public generateElement(aDomainModelCS : DomainModelCS)]
[comment @main/]
    [for (element : ClassCS | aDomainModelCS.element->filter(ClassCS))]
        [generateClass(element)/]
    [/for]
[/template]

[template public generateClass(c : ClassCS)]
    [file (c.name + '.java', false, 'UTF-8')]

       public class [c.name/] {

            [for (field : FieldCS | c.fieldcs)]
                [generateFieldAnnotations(field)/]
                [field.visibility/] [generateFieldType(field)/] [field.name/];
            [/for]

            public [c.name/]() {
                // Constructor body
            }

            [for (field : FieldCS | c.fieldcs)]
                public void set[field.name.toUpperFirst()/]([generateFieldType(field)/] [field.name/]) {
                    this.[field.name/] = [field.name/];
                }
            [/for]

            [for (field : FieldCS | c.fieldcs)]
                public [generateFieldType(field)/] get[field.name.toUpperFirst()/]() {
                    return this.[field.name/];
                }
            [/for]
```

Hình 6.11: Giao diện công cụ Acceleo định nghĩa các luật chuyển để sinh mã nguồn.

Công cụ hỗ trợ phương pháp tích hợp mối quan tâm dựa vào cây cú pháp

Công cụ được được xây dựng bằng MPS [19, 125] có cấu trúcsau bao gồm bốn lớp như sau:

Lớp lõi (core layer): Định nghĩa cú pháp của DSL lõi (`UDML.core`), tạo nền tảng cho việc tích hợp các mối quan tâm khác vào UDML. Lớp này bao gồm các khái niệm cơ bản như `Annotable`, `Annotation`, `Class` và `Association`.

Lớp mối quan tâm (concern layer): Định nghĩa cú pháp trừu tượng cho từng mối quan tâm tùy theo mục tiêu phát triển. Ví dụ: `UDM.rbac`, `UDM.gui`, `UDM.dcsl`, và `UDM.conc` cho các khái niệm tổng quát khác.

Lớp hợp nhất (composition layer): Hỗ trợ việc hợp nhất các DSL theo mối quan tâm vào mô hình UDML thông qua cơ chế hợp nhất được mở rộng, tạo thành một cây cú pháp trừu tượng (AST) thống nhất và nhất quán.

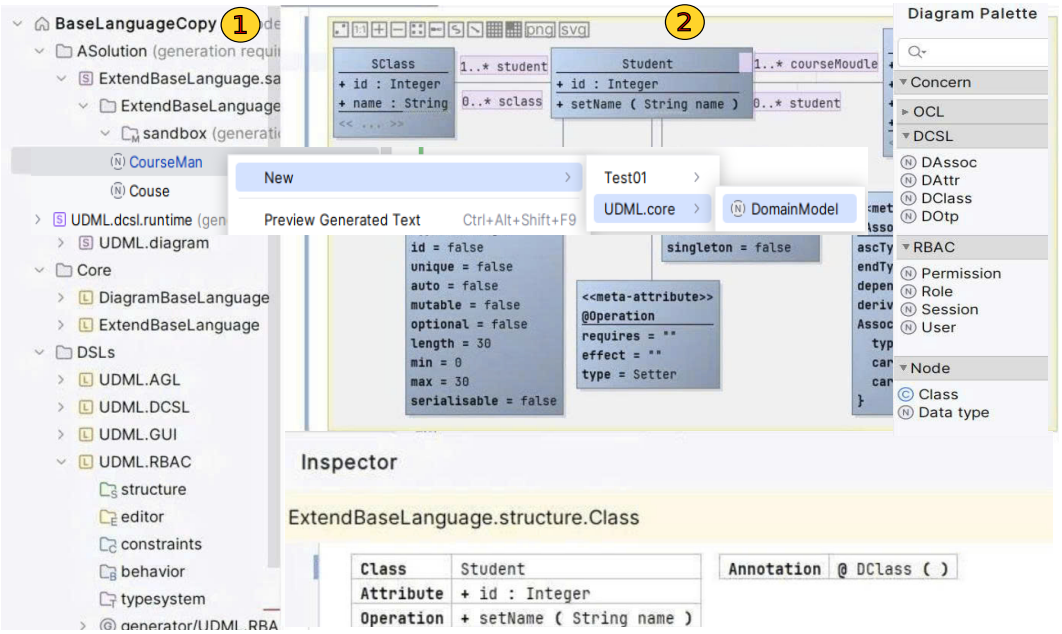
Lớp dịch vụ (services layer): Cung cấp các dịch vụ giao diện đồ hoạ cho phép nhà thiết kế thực hiện các tác vụ như chỉnh sửa trình bày, kiểm tra ràng buộc, sinh phần mềm/nguyên mẫu, cùng với các dịch vụ hỗ trợ như kiểm tra bảo mật.

Phương pháp được hiện thực bằng JetBrains MPS [19, 125], một nền tảng linh hoạt cho việc định nghĩa và hợp nhất DSL dựa trên AST. Mỗi DSL theo mối quan tâm định nghĩa một tập các khái niệm (tương ứng với các kiểu nút trong AST) có thể được tái sử dụng và kết hợp trong mô hình. Các trình chỉnh sửa dạng chiếu (projectional editors) cho phép mỗi DSL có giao diện chuyên biệt riêng, trong khi tất cả cùng đóng góp vào một mô hình thống nhất. MPS cũng hỗ trợ tích hợp vào quy trình phát triển thông qua sinh mã và công cụ build bên ngoài, dù thường yêu cầu điều chỉnh thay vì tích hợp hoàn toàn tự động.

Mỗi concern DSL được định nghĩa dựa trên bốn thành phần chính: (1) Cú pháp trừu tượng (AS): Xác định cấu trúc ngôn ngữ thông qua các Concept—tương ứng với các nút trong AST. (2) Cú pháp cụ thể (Editor): Xác định cách biểu diễn các khái niệm (dạng bảng, văn bản, ký hiệu...). (3) Hệ thống kiểu và ràng buộc: Đảm bảo tính đúng đắn của mô hình qua kiểm tra và ràng buộc ngữ nghĩa. (4) Bộ sinh mã (Generators): Chuyển đổi

mô hình DSL thành mã đích, thường nhúng vào một ngôn ngữ lập trình hướng đối tượng (OOPL).

Công cụ hỗ trợ việc thiết kế, mở rộng và tích hợp các concern DSL, đồng thời kiểm chứng tính biểu đạt, khả thi và mức độ thỏa mãn của phương pháp.



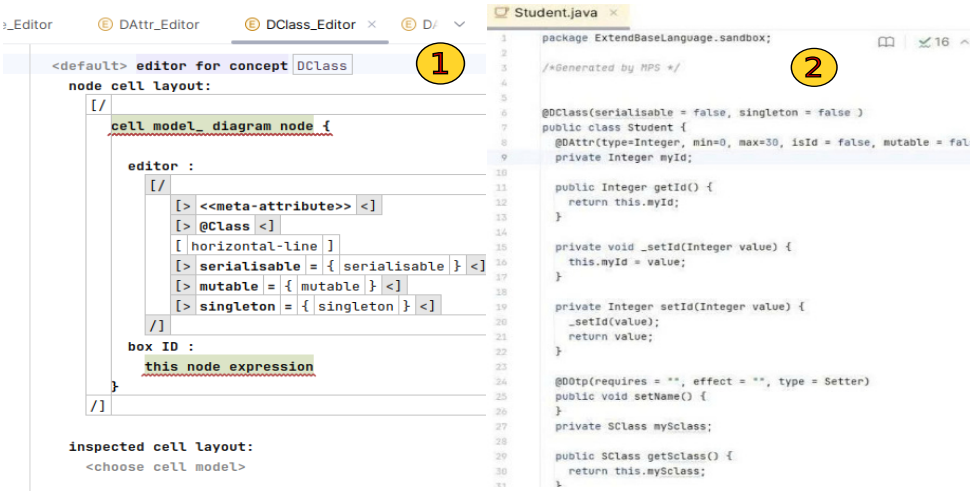
Hình 6.12: Hiện thực hóa dựa trên MPS và tích hợp thực tiễn các DSL theo mối quan tâm trong UDML.

Tính khả thi của phương pháp được minh họa trong Hình 6.12, và mã nguồn được công bố tại GitHub[5] Trong Hình 6.12, phía bên trái thể hiện ba thành phần lõi: (i) Instructor mô tả các siêu khái niệm (metaconcepts) cho UDML và các concern DSL; (ii) Editor cung cấp giao diện cú pháp cụ thể với hỗ trợ kéo–thả; (iii) Generator biến đổi AST thành mã thực thi dựa trên framework JDA [72]. Ngôn ngữ UDML được thiết kế thành các gói mô-đun: `UDML.core` đóng vai trò nền tảng tích hợp; `UDML.dcsl`, `UDML.agl`, và `UDML.rbac` lần lượt mô tả các mối quan tâm về cấu trúc, hành vi và bảo mật. Khu vực trung tâm cung cấp giao diện mô hình hóa miền với hỗ trợ dựng mô hình trực quan.

Luận án áp dụng công cụ vào nghiên cứu tình huống CourseMan. Trong Hình 6.12, nhãn (2) minh hoạ biểu đồ lớp UML cùng các DSL theo mối

[5] https://github.com/vinhskv/Tool-ThesisVinh/tree/main/AST-UDML-main

quan tâm được tích hợp: DCSL, AGL và RBAC. Công cụ sau đó sinh mã nguồn và tạo ra nguyên mẫu phần mềm bằng khung làm việc JDA, với các chi tiết kỹ thuật trình bày trong Hình 6.13.



Hình 6.13: Hiện thực hóa dựa trên MPS và sinh mã thông qua khuôn khổ JDA.

Trong Hình 6.13, phía trái (nhãn (1)) cho thấy định nghĩa template file trong MPS, trong khi phía phải (nhãn (2)) thể hiện mô hình miền `Student` của hệ thống quản lý khóa học. Mô hình này sau đó được nhúng vào JDA để sinh các tạo tác phần mềm.

6.2.4 Công cụ hỗ trợ chuyển đổi mô hình

RM2UDM: Chuyển đổi mô hình yêu cầu sang mô hình miền hợp nhất

Công cụ được hiện thực trong nền tảng JDA nhằm hỗ trợ quá trình chuyển đổi mô hình từ RM sang UDM. Để minh họa cách thức hoạt động của công cụ, luận án sử dụng hệ thống CourseMan như một ví dụ điển hình. Kết quả thực nghiệm cho thấy công cụ hỗ trợ hiệu quả việc xây dựng biểu diễn miền hợp nhất và thực hiện các phép chuyển đổi mô hình. Trên cơ sở đó, các kết quả được phân tích trong bối cảnh các ca nghiên cứu của chương, đồng thời làm rõ các yếu tố ảnh hưởng đến tính đúng đắn và hiệu quả của phương pháp.

Để hiện thực các chuyển đổi mô hình, các luật chuyển được đặc tả bằng ngôn ngữ chuyển đổi mô hình–sang–mô hình (M2M), trong đó ATL [62]

được sử dụng để xây dựng bộ chuyển đổi RM2UDM như đã trình bày trong Mục 5.3. Hình 6.14 (A) minh họa một phần các luật chuyển được cài đặt bằng ATL, trong khi Hình 6.14 (B) thể hiện kết quả ánh xạ từ mô hình yêu cầu sang mô hình miền hợp nhất UDM.

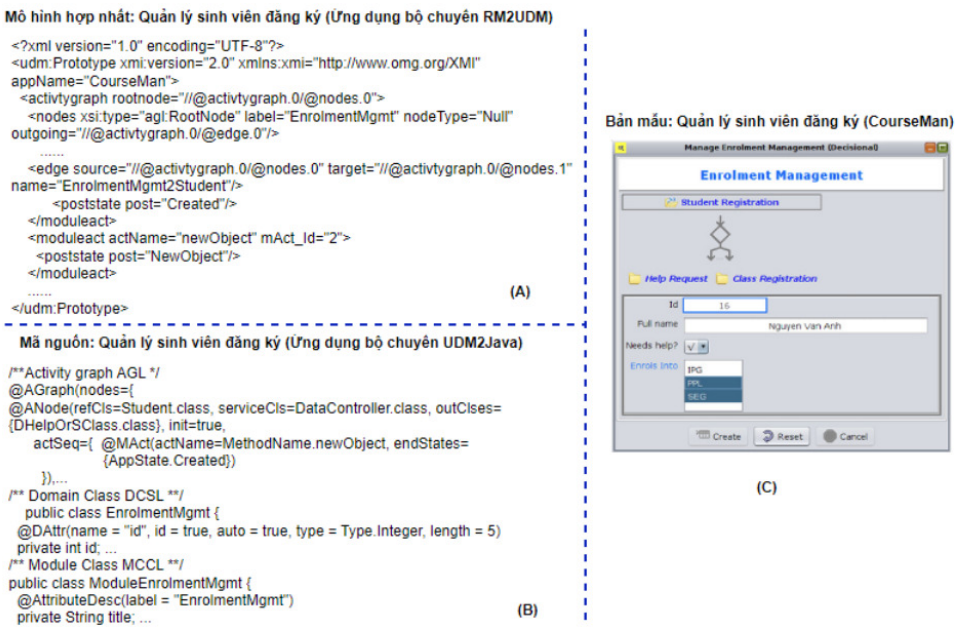
Hình 6.14: Các luật chuyển đổi của ATL và kết quả mô hình UDM.

Thực nghiệm. Công cụ hỗ trợ phương pháp đề xuất được phát triển nhằm sinh tự động bản mẫu phần mềm từ mô hình yêu cầu. Người thiết kế sử dụng các công cụ mô hình hóa trực quan để xây dựng các biểu đồ lớp và biểu đồ hoạt động UML/OCL. Bộ chuyển đổi RM2UDM được hiện thực bằng ATL trên nền tảng Eclipse để chuyển đổi mô hình yêu cầu sang mô hình miền hợp nhất UDM. Sau đó, mô hình UDM được ánh xạ sang mã nguồn Java thông qua Acceleo nhằm sinh tự động bản mẫu phần mềm theo thiết kế hướng miền và thực thi với khung phần mềm JDA.

Hình 6.15 (A) hiển thị một phần UDM dạng XMI là kết quả của bộ chuyển đổi RM2UDM, Hình 6.15 (B) hiển thị một phần mã nguồn Java là kết quả của ánh xạ mô hình UDM bằng Acceleo trên Eclipse và Hình 6.15 (C) biểu diễn kết quả GUI bản mẫu phần mềm cho CourseMan.

Để đánh giá hiệu quả của công cụ trên hệ thống CourseMan, các kết quả thực nghiệm cho thấy: đối với bộ chuyển đổi RM2UDM, mô hình miền hợp nhất UDM đạt mức độ bao phủ và khả năng diễn đạt trên 95% so với mô hình yêu cầu đầu vào.

Đối với bước ánh xạ tự động từ mô hình UDM sang mã nguồn Java nhằm cài đặt bản mẫu phần mềm, toàn bộ các lớp miền trong mô hình yêu cầu đều được chuyển đổi tương ứng thành các lớp trong mã nguồn và các mô-đun phần mềm. Cụ thể, để xây dựng bản mẫu phần mềm của CourseMan



Hình 6.15: Công cụ hỗ trợ sinh tự động mô hình miền hợp nhất có thể thực thi từ đặc tả yêu cầu.

theo cách thủ công cần 549 dòng mã lệnh chính, trong khi phương pháp đề xuất sinh tự động được 477 dòng mã. Tỷ lệ mã nguồn được sinh tự động đạt khoảng 87%, trong khi phần chỉnh sửa và bổ sung thủ công chiếm dưới 13%. Nguyên nhân của các chỉnh sửa này được phân tích trong phần thảo luận tiếp theo.

Thảo luận. Phương pháp đề xuất đã được áp dụng thành công cho hệ thống CourseMan, cho thấy tiềm năng ứng dụng của phương pháp trong thực tiễn. Tuy nhiên, vẫn tồn tại một số mối đe dọa đối với tính hợp lệ của phương pháp.

Thứ nhất, phương pháp tiếp nhận đặc tả yêu cầu đầu vào dưới dạng các biểu đồ UML/OCL, do đó tính đúng đắn của kết quả phụ thuộc vào khả năng diễn đạt và mức độ chính xác của ngôn ngữ mô hình hóa này.

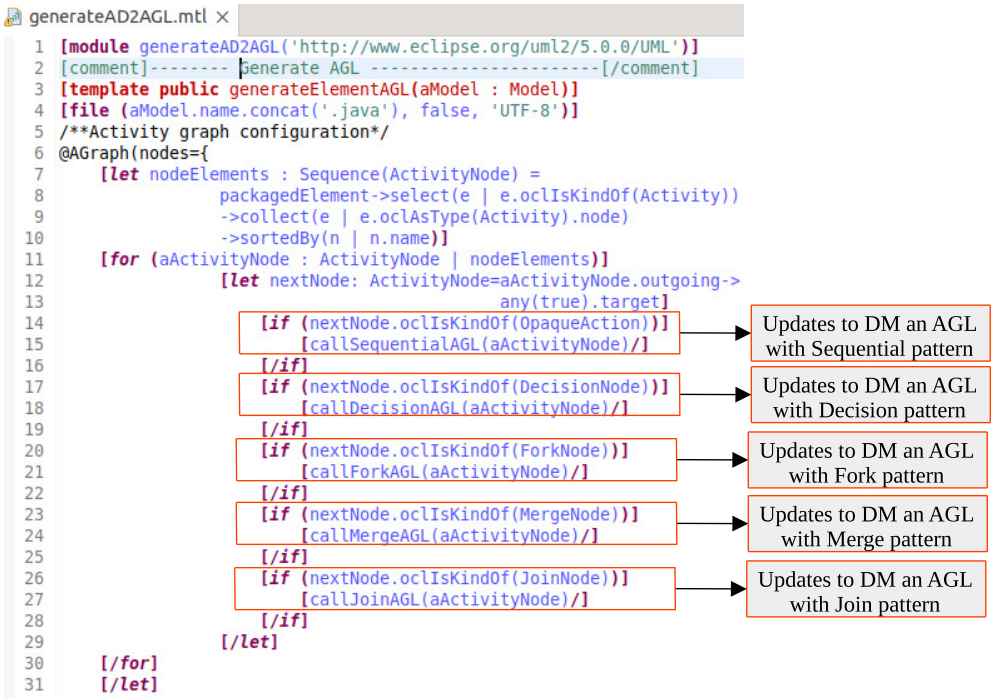
Thứ hai, phương pháp hiện tại chưa đảm bảo rằng mọi mô hình UML/OCL đầu vào đều có thể được chuyển đổi tự động sang một mô hình hợp nhất UDM. Bên cạnh đó, các ràng buộc OCL trong mô hình yêu cầu cần được diễn đạt tương ứng bằng DCSL trong mô hình UDM thông qua các thao tác thủ công. Những hạn chế này liên quan trực tiếp đến phạm vi áp dụng và tính đúng đắn của bộ chuyển đổi RM2UDM. Trong phạm vi phương pháp

đề xuất, tính đúng đắn ở khía cạnh này được đảm bảo thông qua việc rà soát các luật chuyển đổi và thực hiện kiểm thử cho bộ chuyển đổi.

Thứ ba, bản mẫu thực thi sinh ra trong Java có thể chưa hoàn toàn tương thích với đặc tả yêu cầu đầu vào nếu tồn tại sai sót trong các ánh xạ tự động từ mô hình UDM sang mã nguồn. Tương tự như bộ chuyển đổi RM2UDM, độ chính xác của bước sinh mã được đảm bảo bằng cách kiểm tra, rà soát các luật ánh xạ và tiến hành kiểm thử đối với bộ chuyển đổi này.

Chuyển đổi đặc tả mức cao sang mô hình miền thực thi

Trong phần này, luận án trình bày một nghiên cứu tình huống dựa trên Hệ thống quản lý đơn hàng ORDERMAN, nhằm minh họa việc áp dụng kỹ thuật chuyển đổi đặc tả mức cao dưới dạng biểu đồ hoạt động sang đặc tả AGL. Thông qua ca nghiên cứu này, quá trình chuyển đổi từ mô hình hành vi sang biểu diễn miền có khả năng thực thi được làm rõ trong bối cảnh cụ thể.



Hình 6.16: Các luật (mẫu trong Acceleo) để chuyển đổi từ đặc tả mức cao *AD* sang AGL.

Hình 6.16 minh họa quy trình chuyển đổi M2T từ biểu đồ hoạt động (*AD*) sang đặc tả AGL$^+$ sử dụng Acceleo. Quá trình này dựa trên các mẫu chuyển đổi được định nghĩa trên siêu mô hình, trong đó các nút hành vi

trong *AD* được duyệt, phân loại theo kiểu và ánh xạ tương ứng sang các cấu trúc trong AGL, qua đó hình thành biểu diễn miền có khả năng thực thi.

Mẫu trong Acceleo để chuyển đổi một nút *Decisional* trong *AD*, với hàm `genDecisionNode(curNode)`, bao gồm các bước sau:

Bước 1, biểu diễn cấu hình mẫu của đồ thị hoạt động (Activity Graph) cho mẫu quyết định (decision pattern) của đặc tả AGL$^+$;

Bước 2, đọc tất cả các cạnh đi ra từ *DecisionNode*, ánh xạ chúng vào template ở bước thứ nhất và sinh ra các `ANode` tương ứng với cấu trúc của đặc tả decision trong AD. Dựa trên điều kiện cảnh báo, mỗi `ANode` được liên kết đến nút kế tiếp thích hợp;

Bước 3, sử dụng các mẫu truy vấn để trích xuất thông tin từ mô hình một cách thuận tiện nhằm ánh xạ `MAct` và cập nhật đặc tả AGL của mô hình miền.

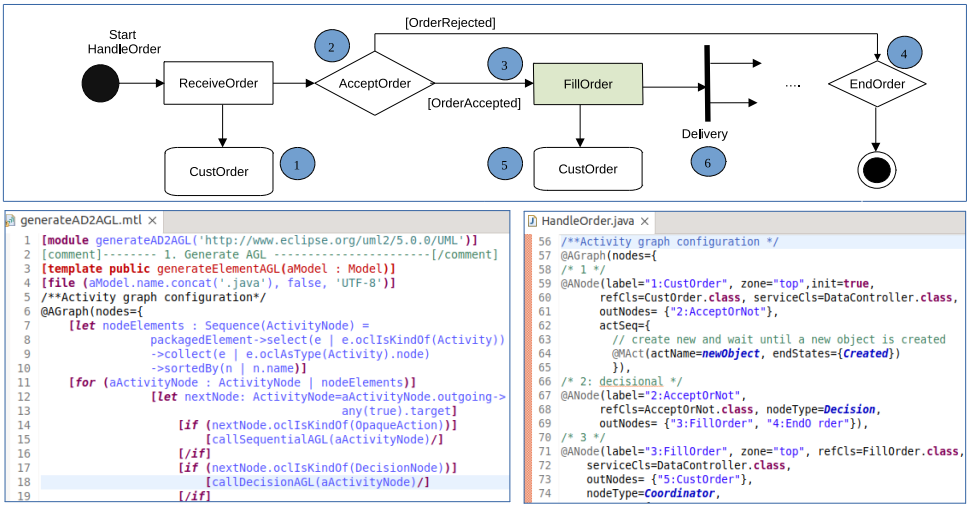
Mẫu trong Acceleo để chuyển đổi từ biểu đồ lớp UML/OCL sang đặc tả DCSL được thực hiện thông qua hàm *genDCSL(model:Model)*. Mô hình đầu vào bao gồm các lớp, thuộc tính, quan hệ kết hợp và các biểu thức OCL trong *CD*. Hàm này thực thi các bước sau: Bước 1, biểu diễn mẫu DCSL của đặc tả AGL$^+$; Bước 2, đọc tất cả các `Class` trong đặc tả *CD* và ánh xạ chúng vào mẫu tương ứng với phương thức của `Class`. Dựa trên các ràng buộc về lớp, thuộc tính và phương thức trong biểu đồ lớp, tiến hành ánh xạ sang mẫu DCSL của đặc tả AGL$^+$; Bước 3, sử dụng các mẫu truy vấn để trích xuất thông tin từ mô hình nhằm ánh xạ *DClass, DAttr, DOpt* và *DAssoc* để cập nhật đặc tả AGL$^+$ của mô hình miền.

```
1     /**actName=newObject,pstStates=\{Created\}*/
2     [query public
3     genMActcreateObject():
4     Set(String)='@MAct(actName=newObject,pstStates=[Created])'
5     /]
6     /** @DClass(serialisable=true, singleton = true)*/
7     [query public
8     genDClass(serialisable: String,singleton: String):
9     Set(String)='@DClass(serialisable
10    ='.concat(serialisable).concat(',singleton=')
11    .concat(singleton).concat(')')/]
```

Đặc tả 6.3: The template generate the *@MAct* and *@DClass*.

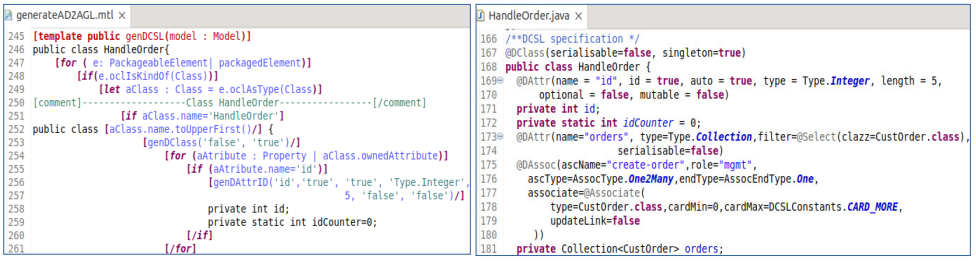
Đoạn mã 6.3 minh họa các mẫu truy vấn mẫu dành cho **@MAct** và **@DClass**. Mã được sinh ra dựa trên các mẫu truy vấn này.

Hình 6.17 minh họa biểu đồ hoạt động của OrderMan ở phía trên, đóng vai trò là mô hình đầu vào của phương pháp đề xuất. Phần bên trái phía dưới hiển thị khuôn mẫu bằng mã Acceleo thực hiện phép chuyển đổi sang mã **HandleOrder**, trong khi kết quả đầu ra của phương pháp—đặc tả AGL—được trình bày ở phía dưới bên phải.



Hình 6.17: Sử dụng Acceleo để chuyển đổi biểu đồ hoạt động của `OrderMan` sang AGL, lớp `HandleOrder` được hiện thực trong Java.

Ngoài ra, Acceleo cũng được sử dụng để chuyển đổi biểu đồ lớp UML/OCL ở mức trừu tượng cao sang đặc tả DCSL. Trong Hình 6.18, khuôn mẫu bằng mã Acceleo dùng để sinh đặc tả DCSL được trình bày ở bên trái, trong khi kết quả đặc tả DCSL thu được được hiển thị ở bên phải.



Hình 6.18: Sử dụng Acceleo để chuyển đổi biểu đồ lớp của `OrderMan` sang đặc tả DCSL.

Tiếp theo, đặc tả AGL được hợp thành với DCSL để tạo thành đặc tả đầy đủ AGL$^+$. Đặc tả AGL$^+$ sau đó được sử dụng làm đầu vào để tự động tạo và hiện thực hệ thống theo phương pháp đề xuất, với sự hỗ trợ của công cụ được xây dựng trên khung phần mềm JDA.

Thảo luận. Nghiên cứu tình huống với hệ thống OrderMan cho thấy kỹ thuật chuyển đổi đặc tả mức cao sang mô hình miền thực thi trong thiết kế hướng miền là khả thi và có tính ứng dụng thực tiễn. Việc sử dụng Acceleo để hiện thực hóa các luật chuyển đổi từ biểu đồ hoạt động UML sang đặc tả AGL, cũng như từ biểu đồ lớp UML/OCL sang đặc tả DCSL, chứng minh rằng các đặc tả mức cao có thể được tự động hóa chuyển hóa thành một mô hình miền mở rộng có khả năng thực thi trong bối cảnh DDD.

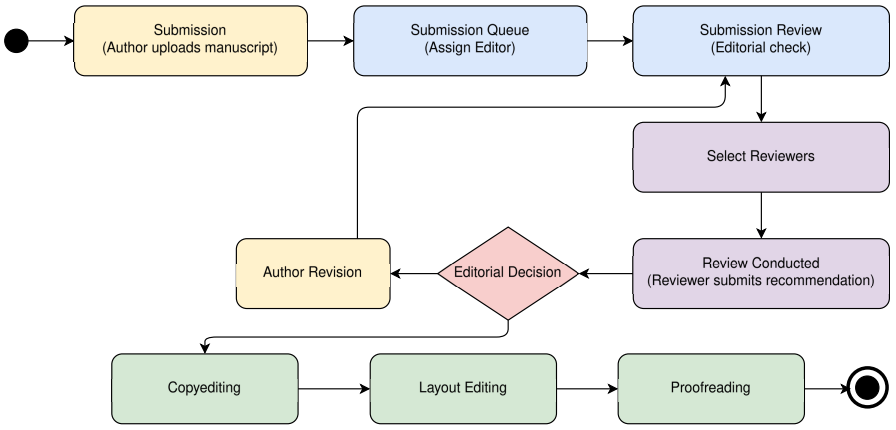
Tuy nhiên, phương pháp đề xuất cũng chịu ảnh hưởng bởi chất lượng và mức độ chính xác của đặc tả đầu vào. Các biểu đồ hoạt động UML chưa được chuẩn hóa hoặc thiếu thông tin ngữ nghĩa (chẳng hạn điều kiện gác hoặc liên kết dữ liệu) có thể dẫn đến các đặc tả AGL chưa đầy đủ và cần được hiệu chỉnh thủ công. Điều này phản ánh hạn chế chung của các tiếp cận sinh mã dựa trên mô hình khi mô hình nguồn không mang đủ thông tin cần thiết cho thực thi.

Việc hợp thành đặc tả AGL và DCSL thành AGL$^+$ khẳng định vai trò trung tâm của mô hình miền trong quá trình sinh phần mềm theo thiết kế hướng miền. Thay vì sinh mã trực tiếp từ UML, phương pháp đề xuất sử dụng mô hình miền hợp nhất như một lớp trung gian mang ngữ nghĩa miền, phù hợp với triết lý DDD khi mô hình miền giữ vai trò xuyên suốt vòng đời phát triển phần mềm.

Công cụ hỗ trợ và đánh giá cho UDML-to-Event-B

Phần này trình bày quy trình làm việc có sự hỗ trợ của công cụ nhằm hiện thực hóa và đánh giá khung ngữ nghĩa được đề xuất cho UDML. Rodin được sử dụng như một hậu-end kiểm chứng để giải quyết các nghĩa vụ chứng minh (POs) của các mô hình Event-B được sinh ra, qua đó xác nhận tính đúng đắn của ngữ nghĩa hình thức đối với các mô hình UDML được hợp thành cùng RBACDom.

Cấu hình thực nghiệm. Tiến hành mô tả cấu hình thực nghiệm được sử dụng trong quá trình đánh giá. Tính khả thi và hiệu quả của phương pháp được đánh giá thông qua một nghiên cứu tình huống dựa trên miền Open Journal Systems (OJS). Hình 6.19 minh họa quy trình quản lý bài nộp của OJS, được mô hình hóa như một đồ thị hoạt động AGL trong UDML và được hợp thành với các chính sách RBACDom thông qua các tương ứng ở mức nút. Việc thực nghiệm xem xét một tập cố định gồm bảy vai trò:



Hình 6.19: Quy trình quản lý bài nộp của OJS.

Roles={Author, Reviewer, JournalManager, Editor, Copyeditor, LayoutEditor, Proofreader}.

Tổng cộng, chín chính sách RBACDom ở mức nút $\langle P_{N1}, \ldots, P_{N9} \rangle$ được đặc tả dưới dạng các thể hiện của `RbacDomNode` trong mô hình RBACDom và được liên kết với các bước của quy trình quản lý bài nộp OJS thông qua sự tương ứng với các nhãn nút AGL (thông qua thuộc tính `aglNode`).

Ánh xạ các chính sách RBAC sang AGL. Mỗi chính sách P_{Ni} được đặc tả như một quy tắc kiểm soát truy cập ở mức nút trong RBACDom và được biểu diễn bởi một `RbacDomNode`. Một chính sách ràng buộc việc thực thi của một bước cụ thể trong quy trình quản lý bài nộp bằng cách áp đặt một điều kiện ủy quyền lên nút AGL tương ứng, được xác định bởi `aglNode`. Cụ thể, một chính sách xác định: (i) các vai trò được phép, (ii) các vị từ phạm vi gắn với đối tượng miền, (iii) chế độ và hiệu lực của chính sách, và (iv) các ràng buộc SoD và tuân thủ tùy chọn. Các chính sách thu được cho quy trình quản lý bài nộp OJS được tổng hợp trong Bảng 6.1.

Bảng 6.1: Chính sách RBACDom dựa trên tương ứng nút cho OJS

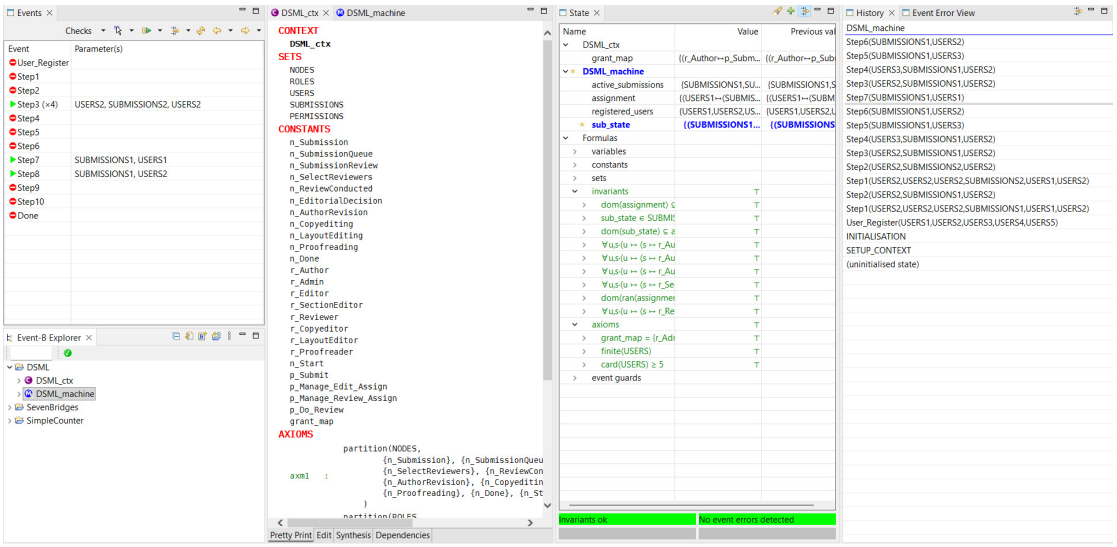
Nút hoạt động	**Chính sách**	**Ràng buộc RBACDom (vai trò, phạm vi, SoD, tuân thủ)**
SubmitPaper	«rbac:P_N1»	{roles={Author}, policy=ALL_OF, effect=ALLOW, scope=owns(u, sub) theo xây dựng (trường tác giả được gán bằng u), SoD=SSD/DSD(Author≠Reviewer trên *sub*, Author≠Editor trên *sub*), compliance=log(createSubmission), hideReviewerIdentities}
AssignEditor	«rbac:P_N2»	{roles={JournalManager}, policy=ALL_OF, effect=ALLOW, scope=state(sub) = SUBMITTED, SoD=DSD(JournalManager≠Author trên *sub*) [tùy chọn], compliance=log(assignEditor)}
EditorialCheck	«rbac:P_N3»	{roles={Editor}, policy=ALL_OF, effect=ALLOW, scope=assignedEditor(u, sub) ∧ state(sub) = SUBMITTED, SoD=DSD(Editor≠Author trên *sub*), compliance=log(editorialCheck)}
SelectReviewer	«rbac:P_N4»	{roles={Editor}, policy=ALL_OF, effect=ALLOW, scope=assignedEditor(u, sub) ∧ state(sub) ∈ {EDITORIAL_CHECKED, UNDER_REVIEW}, SoD=DSD(Editor≠Author trên *sub*)∧DSD(Editor≠Reviewer trên *sub*), compliance=log(assignReviewer), enforceBlindReview}
DoReview	«rbac:P_N5»	{roles={Reviewer}, policy=ALL_OF, effect=ALLOW, scope=assignedReviewer(u, sub) ∧ state(sub) = UNDER_REVIEW ∧ ¬coi(u, sub), SoD=DSD(Reviewer≠Author trên *sub*)∧DSD(Reviewer≠Editor trên *sub*), compliance=log(submitReview), deadlineLogged}
EditorialDecision (Nút quyết định)	«rbac:P_N6»	{roles={Editor}, policy=ALL_OF, effect=ALLOW, scope=assignedEditor(u, sub) ∧ state(sub) = UNDER_REVIEW ∧ allReviewsSubmitted(sub), SoD=DSD(Editor≠Author trên *sub*)∧DSD(Editor≠Reviewer trên *sub*), compliance=log(editorDecision), rationaleRequired [tùy chọn]}
CopyEditing	«rbac:P_N7»	{roles={Copyeditor}, policy=ALL_OF, effect=ALLOW, scope=assignedCopyeditor(u, sub) ∧ state(sub) = ACCEPTED, SoD=DSD(Copyeditor≠Author trên *sub*) [tùy chọn], compliance=log(copyedit), noAccessToReviewData}
LayoutEditing	«rbac:P_N8»	{roles={LayoutEditor}, policy=ALL_OF, effect=ALLOW, scope=assignedLayoutEditor(u, sub) ∧ state(sub) = COPYEDITED, SoD=DSD(LayoutEditor≠Copyeditor trên *sub*) [tùy chọn], compliance=log(layout), checksumVersioning [tùy chọn]}
Proofreading	«rbac:P_N9»	{roles={Proofreader}, policy=ALL_OF, effect=ALLOW, scope=assignedProofreader(u, sub) ∧ state(sub) = LAYOUT_DONE, SoD=DSD(Proofreader≠LayoutEditor trên *sub*) [tùy chọn], compliance=log(proofApproval), readOnlyContent}

Đặc tả ở mức nút này cho phép cục bộ hóa kiểm soát truy cập tại các ranh giới thực thi hành vi, đồng thời vẫn bảo toàn sự tách biệt giữa DCSL, AGL và RBACDom.

Kết quả thực nghiệm. Để ngăn chặn các trạng thái không hợp lệ, hệ thống xác định một tập các bất biến đóng vai trò như một "lưới an toàn" toán học, bao trùm từ các ràng buộc kiểu dữ liệu cơ bản đến các chính sách DSD. Như một ví dụ tiêu biểu, quy tắc nghiệp vụ "Tác giả không được đảm nhận vai trò phản biện cho chính bài nộp của mình" được hình thức hóa bằng vị từ bắt buộc sau:

$$\begin{aligned} &\forall\, u, s \cdot (u \mapsto (s \mapsto r\_Author)) \in assignment \\ &\quad \Rightarrow (u \mapsto (s \mapsto r\_Reviewer)) \notin assignment \end{aligned} \tag{6.1}$$

Tính đúng đắn của mô hình Event-B thu được được kiểm chứng bằng bộ kiểm tra mô hình ProB trong môi trường Rodin; kết quả kiểm chứng được minh họa trong Hình 6.20.



Hình 6.20: Kiểm chứng thực nghiệm bằng Rodin/ProB sử dụng hệ thống OJS làm ca nghiên cứu.

Như thể hiện trong Hình 6.20, quá trình kiểm chứng được trực quan hóa thông qua ba vùng giao diện. Bảng điều khiển bên trái trình bày mối quan hệ cấu trúc giữa ngữ cảnh và máy Event-B, trong khi bảng bên phải ghi lại một vết thực thi cụ thể của một bài báo từ khởi tạo đến hoàn tất, cho thấy quy trình không rơi vào bế tắc. Bảng trung tâm cung cấp bằng chứng rõ ràng về tính an toàn: các chỉ báo "T" (True) màu xanh xác nhận rằng, trong suốt kịch bản thực thi, tất cả các ràng buộc bảo mật — bao gồm cả quy tắc DSD trong Công thức (6.1) — đều được bảo toàn. Mô hình Event-B được sinh ra, bao gồm các thành phần ngữ cảnh và máy, được nhập vào Rodin, nơi các nghĩa vụ chứng minh được tự động sinh và giải quyết nhằm xác nhận tính khả thi thực thi, các ràng buộc bảo mật và tính nhất quán xuyên mối quan tâm.

RQ1: *Làm thế nào có thể định nghĩa ngữ nghĩa vận hành hình thức cho các mô hình miền hợp nhất trong UDML?*

Kết quả cho thấy ngữ nghĩa vận hành của UDML có thể được xác định hình thức bằng cách diễn giải mô hình miền hợp nhất như một *hệ chuyển trạng thái thống nhất*, trong đó các mối quan tâm chia sẻ trạng thái hệ thống

chung nhưng đóng góp các ràng buộc ngữ nghĩa riêng. AGL xác định các chuyển trạng thái hành vi, trong khi DCSL và RBACDom được diễn giải như các bất biến và điều kiện kích hoạt.

Bảng 6.2: Suy diễn thống kê kiểm chứng từ các tạo tác hình thức của OJS và RBACDom

Thành phần mô hình	**Cơ sở hình thức**	**# Bất biến**	**# Sự kiện**	**# Nghĩa vụ chứng minh**
Mô hình RBAC lõi	Các ràng buộc nhất quán vai trò–quyền hạn, loại trừ lẫn nhau và hợp dạng RBAC được suy ra từ RBACDom, độc lập với luồng công việc OJS.	14	8	96
Gán vai trò & ràng buộc SoD	Các ràng buộc phân tách nhiệm vụ tĩnh và động được suy ra từ đặc tả RBACDom cho các vai trò Tác giả, Phản biện, Biên tập viên và các vai trò sản xuất.	11	6	78
Ủy quyền thời gian chạy (Phiên)	Ngữ nghĩa phiên và kích hoạt vai trò, bảo đảm chỉ các vai trò đã được gán và đang hoạt động mới có thể thực thi các thao tác được bảo vệ.	9	5	64
Liên kết xuyên mối quan tâm (RBAC × OJS)	Các bất biến liên kết các ràng buộc ủy quyền RBAC với các bước trong luồng công việc OJS và các trạng thái miền thông qua các chú thích RBACDom.	12	7	102
Ràng buộc theo ngữ cảnh / trạng thái	Các ràng buộc về thứ tự luồng công việc và tiến hóa trạng thái, giới hạn các chuyển trạng thái miền hợp lệ trong vòng đời OJS.	6	4	41
Tổng cộng	—	**52**	**30**	**381**

Ánh xạ UDML sang Event-B hiện thực hóa ngữ nghĩa này: các nút AGL được biên dịch thành các sự kiện Event-B, còn các ràng buộc cấu trúc và bảo mật thành bất biến và điều kiện cảnh báo. Do đó, mỗi phép thực thi UDML tương ứng với một chuỗi sự kiện Event-B bảo toàn các bất biến. Các thống kê trong Bảng 6.2 cung cấp bằng chứng rằng ngữ nghĩa vận hành này được xác định rõ ràng và có thể kiểm chứng bằng các cơ chế hình thức tiêu chuẩn.

Để đánh giá mức độ nỗ lực kiểm chứng, luận án phân tích phân bố các nghĩa vụ chứng minh theo trạng thái hoàn thành. Bảng 6.3 tóm tắt các kết quả được sinh ra bởi nền tảng Rodin.

Mức độ tự động hóa kiểm chứng. Tổng cộng có 381 nghĩa vụ chứng minh được sinh ra, trong đó 369 nghĩa vụ (96,8%) được tự động hoàn tất bởi các bộ chứng minh tích hợp trong Rodin, trong khi 12 nghĩa vụ còn lại (3,2%) yêu cầu một số bước chứng minh tương tác đơn giản.

Bảng 6.3: Phân bố các nghĩa vụ chứng minh và trạng thái hoàn thành

Loại PO	**Tổng**	**Tự động**	**Thủ công**
WD	120	120	0
INV	140	132	8
GRD	80	78	2
THM	41	39	2
Tổng	**381**	**369**	**12**

Lưu ý rằng không phải tất cả các loại nghĩa vụ chứng minh trong Event-B đều xuất hiện trong ca nghiên cứu này. Cụ thể, các nghĩa vụ liên quan đến tinh chỉnh (FIS, SIM, WIT) và các nghĩa vụ dựa trên biến thể (VAR, NAT) không được sinh ra, do mô hình hiện tại không bao gồm các bước tinh chỉnh hoặc các sự kiện hội tụ. Do đó, Bảng 6.3 chỉ báo cáo các loại nghĩa vụ chứng minh thực sự được sinh ra bởi nền tảng Rodin.

RQ2: *Làm thế nào các mối quan tâm xuyên suốt động, tiêu biểu là RBAC, có thể được đặc tả dưới dạng các DSL dựa trên chú thích và được tích hợp vào các mô hình miền hợp nhất?*

Trong đó các chính sách ủy quyền, phân tách nhiệm vụ và ngữ nghĩa phiên được gắn lên các biên thực thi hành vi thay vì mã hóa trực tiếp trong mô hình hành vi. RBACDom được định nghĩa độc lập với AGL nhưng được tích hợp thông qua các liên kết hình thức ở mức cú pháp trừu tượng, qua đó bảo toàn tính mô-đun của các mối quan tâm.

Các bất biến và sự kiện liên quan đến liên kết xuyên mối quan tâm (RBAC $\times$ OJS) trong Bảng 6.2 cho thấy phần lớn nỗ lực kiểm chứng phát sinh từ việc ràng buộc ngữ nghĩa ủy quyền với hành vi và trạng thái miền, phản ánh bản chất động của RBAC trong các hệ thống hướng quy trình. Việc diễn giải RBACDom như các điều kiện bảo vệ trên sự kiện Event-B cho phép áp dụng nhất quán và kiểm chứng được các ràng buộc bảo mật mà không làm thay đổi cấu trúc hành vi.

Để đánh giá mức độ nỗ lực kiểm chứng, luận án phân tích phân bố các nghĩa vụ chứng minh theo trạng thái hoàn thành. Bảng 6.3 tóm tắt các kết quả được sinh ra bởi nền tảng Rodin.

Mức độ tự động hóa kiểm chứng. Tổng cộng có 381 nghĩa vụ chứng

minh được sinh ra, trong đó 369 nghĩa vụ (96,8%) được tự động chứng minh bởi các bộ chứng minh tích hợp trong Rodin, trong khi 12 nghĩa vụ còn lại (3,2%) yêu cầu các bước chứng minh tương tác ở mức đơn giản.

Phần lớn các nghĩa vụ được chứng minh tự động tương ứng với các điều kiện xác định tốt (WD) và tăng cường điều kiện bảo vệ (GRD), vốn được các bộ chứng minh tự động xử lý hiệu quả. Các nghĩa vụ cần chứng minh tương tác chủ yếu liên quan đến việc bảo toàn bất biến và chỉ yêu cầu các bước suy luận đơn giản, chẳng hạn như lựa chọn giả thuyết phù hợp hoặc khởi tạo các biến lượng hóa. Không cần sử dụng các chiến lược chứng minh phức tạp hay thay đổi cấu trúc mô hình; tất cả các nghĩa vụ còn lại đều được hoàn tất bằng bộ chứng minh mệnh đề tích hợp.

Lưu ý rằng không phải tất cả các loại nghĩa vụ chứng minh trong Event-B được liệt kê đều xuất hiện trong ca nghiên cứu này. Cụ thể, các nghĩa vụ liên quan đến tinh chỉnh (FIS, SIM, WIT) và các nghĩa vụ dựa trên biến thể (VAR, NAT) không được sinh ra, do mô hình hiện tại không bao gồm các bước tinh chỉnh hoặc các sự kiện hội tụ. Do đó, Bảng 6.3 chỉ báo cáo các loại nghĩa vụ chứng minh thực sự được sinh ra bởi nền tảng Rodin.

Các kết quả trên OJS xác nhận rằng UDML, khi hợp thành với AGL và RBACDom, hỗ trợ đặc tả các mô hình miền hợp nhất có khả năng thực thi và ngữ nghĩa hình thức rõ ràng, trong đó AGL xác định sự tiến hóa hành vi còn RBACDom giới hạn khả năng thực thi thông qua các điều kiện ủy quyền dựa trên tương ứng nút. Khối lượng nghĩa vụ chứng minh tuân theo các dạng chuẩn của Event-B, cho thấy phương pháp đề xuất có thể tái sử dụng trực tiếp các cơ chế kiểm chứng hiện có mà không cần mở rộng đặc thù, qua đó khẳng định UDML như một nền tảng ngữ nghĩa hình thức cho các mô hình miền hợp nhất có thể phân tích và kiểm chứng.

Thảo luận. Rodin được sử dụng trong bài báo này nhằm xác nhận khung ngữ nghĩa đề xuất, thay vì để đánh giá hiệu năng của công cụ. Cơ chế sinh và chứng minh tự động các nghĩa vụ chứng minh (PO) cho phép phát hiện sớm các bất nhất ngữ nghĩa và các vi phạm chính sách, những vấn đề vốn có thể bị ẩn trong các mô hình miền có khả năng thực thi.

Kết quả trên hệ thống OJS cho thấy các mô hình UDML được hợp thành với RBACDom có thể được diễn giải như các hệ chuyển trạng thái hợp nhất với khả năng thực thi và các thuộc tính bảo mật có thể phân tích hình thức.

Trong ngữ cảnh này, AGL xác định sự tiến hóa hành vi của mô hình miền bằng cách đặc tả khi nào một nút có thể được thực thi theo ngữ nghĩa luồng điều khiển, trong khi RBACDom quyết định liệu nút đó có được phép thực thi hay không. Các thống kê kiểm chứng cho thấy nỗ lực kiểm chứng phần lớn được xử lý bởi các cơ chế chuẩn của Event-B, đạt được mức độ tự động hóa cao mà không cần các mở rộng kiểm chứng đặc thù.

Một khía cạnh thiết kế quan trọng khác là chiến lược kết hợp luật được sử dụng để đánh giá nhiều chú thích RBACDom gắn với cùng một nút hành vi. Trong khung được mở rộng, việc kết hợp luật được mô hình hóa như một tham số cấu hình ở mức mô hình thông qua hàm $combiningAlg : DomainModelRbacDom \rightarrow CA$. Điều này cho phép đặc tả trực tiếp các ngữ nghĩa phân quyền khác nhau, bao gồm denyOverrides, permitOverrides và firstApplicable, ngay trong mô hình RBACDom, đồng thời vẫn bảo toàn cơ chế chuyển đổi UDML sang Event-B. Mở rộng này nâng cao tính tổng quát của khung phương pháp mà không ảnh hưởng đến ngữ nghĩa cấu trúc của DCSL hay ngữ nghĩa hành vi của AGL.

Hạn chế và khả năng sử dụng. Đánh giá hiện tại tập trung vào một quy trình có tính thực tế nhưng quy mô trung bình với tập vai trò cố định; việc khái quát rộng hơn đòi hỏi thêm các ca nghiên cứu bổ sung. Bên cạnh đó, các lựa chọn mô hình hóa như mức độ phân rã của các nút và độ biểu đạt của các vị từ phạm vi có ảnh hưởng trực tiếp đến kích thước của mô hình Event-B được sinh ra cũng như số lượng nghĩa vụ chứng minh.

Hơn nữa, tỷ lệ cao các nghĩa vụ chứng minh được tự động hoàn tất trong ca nghiên cứu OJS (96,8%) cho thấy phần lớn các nghĩa vụ tương ứng với các dạng nghĩa vụ chuẩn của Event-B, vốn được các bộ chứng minh Rodin xử lý hiệu quả. Điều này cung cấp bằng chứng ban đầu cho thấy phương pháp có khả năng mở rộng mà không làm tăng tương ứng nỗ lực kiểm chứng thủ công. Tuy nhiên, việc đánh giá toàn diện trên các hệ thống công nghiệp quy mô lớn vẫn là hướng nghiên cứu trong tương lai.

Nguyên mẫu hiện tại sử dụng các đặc tả dựa trên chú thích để liên kết các chính sách RBACDom với các nút hành vi. Mặc dù cách biểu diễn này cho phép tích hợp trực tiếp với cơ sở mã hiện có, việc viết chú thích thủ công có thể trở nên dễ sai sót đối với các quy trình lớn như OJS. Do đó, các nghiên cứu tiếp theo sẽ tập trung phát triển các công cụ hỗ trợ theo hướng

đồ họa và hướng mô hình cho RBACDom, cho phép đặc tả các chính sách bảo mật trực tiếp ở mức mô hình miền và tự động chuyển đổi sang các chú thích và hiện vật hình thức tương ứng. Các công cụ này sẽ giúp giảm công sức đặc tả thủ công và nâng cao khả năng sử dụng cho các hệ thống quy mô lớn.

6.3 Thực nghiệm và đánh giá

Phần này trình bày, thực nghiệm và đánh giá về các kỹ thuật biểu diễn và chuyển đổi mô hình cho thiết kế hướng miền.

6.3.1 Kỹ thuật biểu diễn mô hình miền tích hợp ràng buộc

Trong phần này, luận án tiến hành đánh giá hai khía cạnh chính: (i) mức độ biểu đạt của CAP trong việc mô hình hóa các khái niệm miền, và (ii) khả năng áp dụng CAP trong thực tiễn phát triển phần mềm. Để đảm bảo tính khách quan và độ tin cậy của kết quả, luận án trình bày chi tiết phương pháp đánh giá cùng với các ca nghiên cứu được sử dụng trong quá trình thực nghiệm.

Phương pháp đánh giá. Để củng cố các kết quả và hạn chế tính chủ quan trong diễn giải, luận án tiến hành các thực nghiệm tái xây dựng có kiểm soát nhằm so sánh CAP với bốn nền tảng: DCSL [70], Apache Causeway [119], OpenXava [95], và Actifsource [1].

Các ca nghiên cứu. Đối với mỗi nền tảng, luận án tái xây dựng các mô hình miền tương đương về chức năng cho ba hệ thống nghiên cứu—CourseMan, ProcessMan, và OrderMan—được lựa chọn nhằm đảm bảo tính đa dạng về miền, sự phong phú của các ràng buộc, cũng như mức độ trưởng thành khác nhau của mô hình hóa. CourseMan là một miền học thuật chuẩn, có nhiều ràng buộc cấu trúc và hành vi phong phú. ProcessMan cho phép so sánh có kiểm soát với các mô hình dựa trên DCSL trước đó nhằm đánh giá lợi ích gia tăng của CAP. OrderMan, được xây dựng dựa trên các kịch bản quy trình nghiệp vụ trong doanh nghiệp, đại diện cho một miền không đồng nhất, giàu luồng công việc, nhằm đánh giá khả năng mở rộng và tính tổng quát của phương pháp.

Bảng 6.4: Tổng hợp các ca nghiên cứu: CourseMan, ProcessMan, và OrderMan

Ca nghiên cứu	**Loại miền**	**Số lượng ràng buộc OCL**	**Loại ràng buộc OCL theo ngữ nghĩa**	**Vai trò trong đánh giá**
CourseMan	Học thuật	184	14	Mốc chuẩn về khả năng biểu đạt
ProcessMan	Quản lý quy trình	140	12	So sánh có kiểm soát với DCSL
OrderMan	Quy trình nghiệp vụ	76	11	Khả năng mở rộng và tổng quát

Bảng 6.4 tóm tắt các đặc trưng định lượng của ba hệ thống, bao gồm số lượng ràng buộc và các loại ràng buộc OCL theo ngữ nghĩa được sử dụng trong đánh giá.

Quy trình tái xây dựng có kiểm soát. Đối với mỗi ca nghiên cứu và mỗi nền tảng, luận án thực hiện các bước sau:

1. Tái xây dựng mô hình miền (biểu đồ lớp) bằng các cơ chế đặc tả gốc của từng nền tảng;
2. Đặc tả các ràng buộc sao cho phù hợp nhất với đặc tả tham chiếu;
3. Ghi nhận các chỉ số có thể đo lường: số lượng ràng buộc được hỗ trợ trực tiếp bởi chú thích hoặc các cấu trúc dựng sẵn; số lượng ràng buộc cần hiện thực thủ công; kích thước mã kiểm tra thủ công (LOC) cho mỗi ràng buộc; thời gian hiện thực; và mức độ hỗ trợ sinh tự động (sinh mã từ mô hình và sinh lô-gic kiểm tra).
4. Kiểm chứng độc lập kết quả tái xây dựng bởi hai nhà nghiên cứu; các sai khác được xử lý thông qua việc rà soát lại các bản mẫu phần mềm và đạt được sự đồng thuận.

RQ3: Mức độ biểu đạt của CAPs trong việc mô hình hóa các khái niệm miền so với các phương pháp DDD hiện có là như thế nào?

Tính biểu đạt. Phân tích khả năng sử dụng chú thích để đặc tả và kiểm tra các khía cạnh của mô hình miền, từ đó xác định các cấu trúc ngôn ngữ và lượng hóa mức độ tương ứng. Mười hai tiêu chí đánh giá được rút ra từ khả năng biểu đạt của hệ thống chú thích và được áp dụng để đánh giá

sử dụng ba mức: (i) Hỗ trợ đầy đủ, (ii) hỗ trợ một phần hoặc gián tiếp, (iii) Không hỗ trợ. Kết quả tổng hợp thể hiện trong Bảng 6.5. Kết quả cho thấy: CAP đạt mức hài lòng 100%, tức toàn bộ 12/12 tiêu chí đều được hỗ trợ đầy đủ hoặc gián tiếp. Điều này chứng minh rằng các mô hình miền xây dựng bằng CAP có khả năng biểu đạt toàn bộ các ràng buộc nghiệp vụ trong ubiquitous language. Ngược lại, DCSL, OpenXAVA và Apache Causeway còn 25% tiêu chí không được hỗ trợ (3/12). Trong khi đó, Actifsource, do được thiết kế chủ yếu cho mô hình hóa cấu trúc và sinh mã, cũng đạt mức 100% hỗ trợ (đầy đủ hoặc gián tiếp) đối với các tiêu chí liên quan đến mô hình miền.

Bảng 6.5: So sánh các công cụ dựa trên mức độ biểu đạt

Mức độ tính biểu đạt\ Các tiêu chí so sánh	Hỗ trợ đầy đủ	Hỗ trợ một phần hoặc gián tiếp	Không hỗ trợ
CAP	10/12	2/12	0
DCSL	4/12	5/12	3/12
OPenXAVA	2/12	7/12	3/12
Apache Causeway	2/12	7/12	3/12
Actifsource	9/12	3/12	0

Mức độ mã hóa thủ công. Các tiêu chí này đánh giá mức độ lập trình thủ công cần thiết để hiện thực và kiểm tra các quy tắc trong mô hình miền, đặc biệt đối với các ràng buộc phức tạp trên năm tiêu chí như sau: (1) *Tự động hóa sinh mã:* Mức độ mà công cụ có thể tự động sinh mã, từ đó giảm thiểu nỗ lực lập trình thủ công. (2) *Hỗ trợ ràng buộc phức tạp:* Khả năng xử lý các ràng buộc nâng cao (ví dụ: biểu thức OCL) mà không cần viết mã tùy chỉnh. (3) *Tích hợp mô hình:* Mức độ dễ dàng khi tích hợp ràng buộc vào mô hình miền với ít điều chỉnh thủ công nhất. (4) *Độ khó học:* Nỗ lực cần thiết để sử dụng công cụ hiệu quả; đường cong học tập càng cao thì nhu cầu lập trình thủ công càng tăng. (5) *Tính linh hoạt:* Khả năng tùy biến hoặc mở rộng lô-gic miền mà không cần nhiều mã bổ sung. Sử dụng ba mức đánh giá là:

Mức độ mã hóa thủ công. Đánh giá này tập trung vào các *ràng buộc OCL cấu trúc thiết yếu* được định nghĩa trong Bảng 2.1 (Mục 2.1.4,

trang 24), trong đó tổng hợp tập các ràng buộc miền cốt lõi được xem xét trong nghiên cứu. Mục tiêu là định lượng mức độ công sức hiện thực thủ công cần thiết để thực thi các ràng buộc này khi sử dụng DCSL mở rộng với CAP so với các nền tảng khác.

Phạm vi đánh giá. Đối với mỗi trong ba ca nghiên cứu (CourseMan, ProcessMan, và OrderMan), luận án tái xây dựng các mô hình miền tương đương và hiện thực cùng một tập các loại ràng buộc OCL thiết yếu, bao gồm: (i) các ràng buộc giá trị thuộc tính; (ii) các ràng buộc bội số và lực lượng; (iii) các ràng buộc phụ thuộc; (iv) các ràng buộc dựa trên phép tổng hợp (ví dụ: tổng, đếm); và (v) các bất biến cấu trúc liên kết chéo.

Chỉ các ràng buộc thiết yếu được đưa vào đo lường, vì chúng đại diện cho các quy tắc toàn vẹn cốt lõi của miền mà CAP hướng tới. Các ràng buộc phụ trợ và ở mức khung làm việc được loại trừ do thường được xử lý tự động và không phản ánh chính xác mức giảm công sức mô hình hóa thủ công.

Quy trình đo lường. Đối với mỗi nền tảng và mỗi thể hiện ràng buộc thiết yếu, luận án ghi nhận: (i) liệu ràng buộc có thể được thực thi theo cách khai báo mà không cần mã kiểm tra thủ công hay không; (ii) số lượng phương thức kiểm tra thủ công cần thiết; (iii) số dòng mã kiểm tra thủ công (LOC), không bao gồm chú thích và dòng trống; và (iv) số lượng lớp validator/service bổ sung được tạo ra chỉ để thực thi ràng buộc.

Lưu ý rằng các khai báo lớp cấu trúc không được tính là mã kiểm tra, vì chúng thuộc đặc tả mô hình miền chứ không phải lô-gic kiểm tra ràng buộc – là trọng tâm của CAP.

Bảng 6.6: Mức độ mã hóa thủ công các ràng buộc OCL thiết yếu

Nền tảng	**LOC thủ công trung bình/ràng buộc**	**% ràng buộc thiết yếu được tự động thực thi**	**Số lớp validator bổ sung**
CAP-extended DCSL	6.2	92%	0
Apache Causeway	54.3	22%	6
OpenXava	37.5	45%	4
Actifsource	31.8	52%	3

Bảng 6.6 tóm tắt các kết quả tổng hợp trung bình trên ba ca nghiên cứu. DCSL mở rộng với CAP đạt tỷ lệ tự động hóa cao nhất (92%) đối với các ràng buộc OCL thiết yếu và yêu cầu số dòng mã kiểm tra thủ công trung

bình thấp nhất (6.2 LOC trên mỗi ràng buộc), đồng thời không cần bổ sung các lớp kiểm tra ràng buộc. Kết quả này cho thấy các bất biến thiết yếu được đặc tả theo cách khai báo thông qua chú thích CAP và được tái tạo tự động thành lô-gic kiểm tra OCL có thể thực thi. DCSL gốc cung cấp hỗ trợ một phần cho các ràng buộc thiết yếu nhưng vẫn yêu cầu các phương thức thủ công đối với các ràng buộc dựa trên tổng hợp và phụ thuộc, dẫn đến gia tăng LOC. Trong Apache Causeway và OpenXava, các ràng buộc cấu trúc phức tạp thường được hiện thực theo kiểu mệnh lệnh, dẫn đến số LOC thủ công cao hơn đáng kể và cần thêm các lớp kiểm tra ràng buộc/dịch vụ. Actifsource hỗ trợ cấu hình luật ở mức mô hình; tuy nhiên, các ràng buộc liên kết chéo và dựa trên tổng hợp thường đòi hỏi thêm các tảo tác phần mềm hoặc cấu hình bổ sung.

Diễn giải. Các kết quả cho thấy việc tích hợp CAP vào DCSL giúp giảm đáng kể công sức lập trình thủ công trong việc thực thi các ràng buộc OCL thiết yếu ở mức mô hình miền. Mức giảm này phản ánh công sức thực thi trên mỗi ràng buộc, thay vì việc dịch chuyển lô-gic tương đương, do ngữ nghĩa OCL được định nghĩa một lần trong các khuôn mẫu CAP có thể tái sử dụng, thay vì phải hiện thực lại cho từng trường hợp cụ thể. Tất cả các phép đo được thực hiện dựa trên các thực nghiệm tái xây dựng có kiểm soát, sử dụng cùng một tập ràng buộc thiết yếu trên các nền tảng.

RQ4: CAP có thể được áp dụng ở mức độ nào trong thực tiễn phát triển phần mềm?

Để đánh giá khả năng áp dụng của CAP trong thực tế, thực hiện triển khai và kiểm thử các mẫu ràng buộc trên ba hệ thống CourseMan, ProcessMan và OrderMan. Kết quả được tổng hợp trong Bảng 6.7, bao gồm mười bốn nhóm ràng buộc OCL. Được đề xuất một danh mục CAPs, bao gồm nhiều mẫu cung cấp một khuôn khổ cú pháp và ngữ nghĩa hình thức để đặc tả các ràng buộc nghiệp vụ trong OCL, bao gồm: tính hợp lệ cấu trúc, giới hạn định lượng, quan hệ phụ thuộc, v.v.

Tuy nhiên, ba nhóm ràng buộc mang tính hành vi sau đây được loại khỏi danh mục CAPs: (1) tiền điều kiện và hậu điều kiện, vốn đòi hỏi ngữ nghĩa tác vụ và suy luận trên các trạng thái trước–sau; (2) kiểm soát truy cập, phụ thuộc vai trò và chính sách nền tảng; (3) phản ứng tự động, liên quan đến ràng buộc sự kiện–tác vụ và thứ tự thực thi.

Bảng 6.7: Các nhóm ràng buộc và các mẫu CAP

Nhóm ràng buộc	**Danh mục CAP**	**Khả năng áp dụng CAP**		
		CourseMan	OrderMan	ProcessMan
Ràng buộc tính hợp lệ (well-formedness)	DCSL trong Bảng 2.1	11/11	11/11	11/11
Giới hạn định lượng	SumConstraint	19/19	4/4	7/7
Ràng buộc phụ thuộc và tiên quyết	PrerequisiteConstraint	10/10	12/12	13/13
Ràng buộc lập lịch và xung đột	ScheduleConstraint	17/17	6/6	10/10
Ràng buộc điều kiện đủ của thực thể	EligibilityConstraint	20/20	5/5	12/12
Quy tắc thi lại / thực hiện lại	RetakeConstraint	8/8	2/2	8/8
Ràng buộc sức chứa	SizeConstraint	17/17	3/3	15/15
Ràng buộc thời gian và hạn chót	TimeConstraint	10/10	6/6	13/13
Quy tắc tính toán và dẫn xuất	SumProduct	15/15	8/8	8/8
Ràng buộc dựa trên trạng thái	StatusConstraint	12/12	6/6	15/15
Ràng buộc cấu trúc tổ chức	StructuralConstraint	29/29	6/6	16/16
Ràng buộc tiền điều kiện / hậu điều kiện	×	0/4	0/4	0/4
Ràng buộc kiểm soát truy cập	×	0/4	0/2	0/4
Ràng buộc phản ứng tự động	×	0/4	0/2	0/4
	Tổng	**172/184**	**66/76**	**128/140**

Ba nhóm này mang tính ràng buộc theo ngữ cảnh hơn là các bất biến cấu trúc, ít có khả năng tái sử dụng như các mẫu bất biến và phụ thuộc mạnh vào kiến trúc triển khai; vì vậy chúng được loại bỏ để giữ cho danh mục CAPs súc tích và mang tính đại diện.

Các thí nghiệm trên ba miền bài toán CourseMan, OrderMan và ProcessMan cho thấy rằng các CAPs được đề xuất có thể được biểu diễn bằng DSL dựa trên chú thích như DCSL và được tích hợp vào mô hình miền thống nhất nhằm hỗ trợ sinh tự động và tạo nguyên mẫu phần mềm.

Trong nghiên cứu tình huống CourseMan, 184 ràng buộc được khảo sát và phân loại vào 14 nhóm. CAPs biểu diễn thành công 172 ràng buộc thuộc 11 nhóm; 12 ràng buộc còn lại nằm ngoài phạm vi hiện tại do phụ thuộc vào trạng thái tại thời gian chạy, sự kiện hoặc điều kiện kích hoạt theo ngữ

cảnh. Các ràng buộc này chủ yếu thuộc nhóm tiền điều kiện/hậu điều kiện, kiểm soát truy cập và phản ứng tự động. Tổng tỷ lệ hỗ trợ đạt 93.5%.

Đối với OrderMan, 76 ràng buộc được phân tích, trải rộng từ ràng buộc cấu trúc đến các quy tắc nghiệp vụ và hành vi phức tạp. Trong số này, 66 ràng buộc được hỗ trợ bởi CAPs; các ràng buộc còn lại có đặc điểm động tương tự CourseMan, dẫn đến tỷ lệ áp dụng 86.8%.

Đánh giá trên ProcessMan cũng cho kết quả tương đồng: trong số 140 ràng buộc được khảo sát, 128 được biểu diễn hiệu quả bằng CAPs, số còn lại liên quan tới hành vi động hoặc cơ chế kích hoạt theo sự kiện vốn nằm ngoài khả năng hiện tại của khuôn khổ, đạt tỷ lệ áp dụng 91.4%.

Bên cạnh tính biểu đạt và khả năng áp dụng, tính khả thi của phương pháp được khẳng định thông qua việc sinh và thực thi các nguyên mẫu phần mềm trực tiếp từ mô hình miền thống nhất có tích hợp CAPs. Điều này chứng minh rằng phương pháp được đề xuất có thể được hiện thực một cách trơn tru dưới dạng các hiện vật phần mềm vận hành thực tế.

Thảo luận. Các mối đe dọa đối với tính hợp lệ của phương pháp đề xuất và quá trình đánh giá, theo phân loại của Runeson et al. [104].

Nghiên cứu giả định rằng các yêu cầu miền được thu thập và diễn giải đầy đủ, chính xác. Nguy cơ diễn giải sai được giảm thiểu thông qua việc biểu diễn tường minh mô hình lớp miền, các ràng buộc OCL và các chú thích tham số hóa, sau đó đóng gói chúng trong các mẫu CAP để hợp nhất vào mô hình miền hợp nhất có khả năng thực thi theo DDD.

Các mối đe dọa chính xuất phát từ việc chuyển đổi đặc tả UML/OCL sang các mẫu CAP và từ khả năng bao phủ chưa đầy đủ của các ràng buộc OCL. Luận án giảm thiểu các rủi ro này bằng cách cung cấp hướng dẫn áp dụng CAP, đánh giá trên nhiều hệ thống có độ phức tạp khác nhau (CourseMan, OrderMan, ProcessMan), và kiểm thử, rà soát kỹ lưỡng công cụ hiện thực trong khung phần mềm JDA.

6.3.2 Kỹ thuật biểu diễn mô hình miền tích hợp hành vi

Luận án sử dụng ba ứng dụng phản ánh yêu cầu thực tế: CourseMan, ProcessMan và OrderMan để đánh giá tính biểu đạt và khả năng áp dụng của kỹ thuật biểu diễn mô hình miền tích hợp hành vi.

Tính biểu đạt được đánh giá bằng cách đối chiếu AGL tích hợp DCSL với các mẫu DDD [39] gọi là AGL$^+$ và so sánh với các khung làm việc DDD dựa trên chú thích: AL [119] và XL [95]. Việc xem xét mô hình hóa cấu trúc (`DomainClass`, `DomainField`, `AssociativeField`, `DomainMethod`), mô hình hoá hành vi, và mức độ mã hoá cần thiết (RCL).

RQ5: *Hiệu quả biểu diễn mô hình miền so với DDD hiện có?*

Bảng 6.8: (A–trái) Tiêu chí 1: Các tiêu chí về tính biểu đạt dựa trên các mẫu DDD; (B–phải) Tiêu chí 2: Các tiêu chí về tính biểu đạt dựa trên các siêu khái niệm của miền

	Tiêu chí 1					**Tiêu chí 2**			
	AGL+	AL	XL	AD		AGL+	AL	XL	AD
DomainClass (DC)	√	√	√	×	DC	**1/1**	1/1	0/1	×
DomainField (DF)	√	√	√	×	DF	**8/8**	4/8	5/8	×
AssociativeField (AF)	√	√	√	×	AF	**7/7**	0/7	1/7	×
DomainMethod (DM)	√	√	√	×	DM	√	×	×	×
ActivityDomainClass (ADC)	*o*	×	×	√	ADC	√	×	×	√

Bảng 6.8(A) cho thấy AGL$^+$ và các phương pháp DDD khác đều hiện thực một phần các mẫu DDD. Bảng 6.8(B) cho thấy AGL$^+$ có tính biểu đạt cao nhất trong cả cấu trúc lẫn hành vi. AL và XL chỉ hỗ trợ một phần cấu trúc và không hỗ trợ hành vi. AD mô tả hành vi tốt nhưng thiếu gắn kết với mô hình cấu trúc.

RQ6: *Nỗ lực cần thiết để định nghĩa mô hình miền thống nhất?*

Từ Figure 3.8 và 3.9, ta có: Bảng 6.9 trình bày số lượng các mẫu hành vi miền được sử dụng (được định nghĩa) để xây dựng các ứng dụng CourseMan, ProcessMan và OrderMan. Trong đó, số lượng mô-đun phản ánh mức độ phức tạp của phần mềm, còn số lượng các mẫu hành vi miền thể hiện mức độ áp dụng của bộ mẫu đầu tiên cũng như lượng công

Bảng 6.9: Thống kê các mẫu hành vi và mô-đun cho CourseMan, ProcessMan và OrderMan

	Mẫu	Mô-đun
CourseMan	3	6
ProcessMan	5	37
OrderMan	5	13

sức được giảm bớt cho lập trình viên trong việc hiện thực phần mềm thủ công.

Thảo luận. Kết quả đánh giá cho thấy phương pháp dựa trên AGLDCSL có khả năng tích hợp hiệu quả vào các quy trình phát triển lặp và linh hoạt, trong đó chuyên gia miền và lập trình viên có thể cộng tác xây dựng mô hình miền hợp nhất và sinh phần mềm trực tiếp từ mô hình. Việc kết hợp với các kỹ thuật kỹ nghệ phần mềm hướng mô hình, đặc biệt là ánh xạ mô hình yêu cầu và mô hình miền trừu tượng sang mô hình miền hợp nhất phụ thuộc nền tảng (AGL$^+$/DCSL trên Java), qua đó thu hẹp khoảng cách từ mô hình đến hiện thực thực thi.

Tính khả dụng của phương pháp chịu ảnh hưởng đáng kể bởi giao diện GUI, vốn được thiết kế đơn giản và nhất quán, phản ánh trực tiếp cấu trúc mô hình miền và hỗ trợ hiệu quả tương tác với chuyên gia miền. Bên cạnh đó, việc AGL được nhúng trong ngôn ngữ lập trình hướng đối tượng cho phép tận dụng các cơ chế kiểm tra tĩnh, tái cấu trúc và hỗ trợ IDE, qua đó nâng cao khả năng xây dựng và bảo trì đặc tả.

Cuối cùng, dựa trên các mẫu hành vi miền và kiến trúc mô-đun độc lập ngôn ngữ, phương pháp có tính tổng quát cao và có thể được triển khai trên các nền tảng khác ngoài Java hoặc thông qua các DSL ngoại sinh sử dụng các khung ngôn ngữ hiện có. Điều này cho thấy tiềm năng mở rộng và khả năng áp dụng của phương pháp trong các bối cảnh phát triển phần mềm đa dạng.

6.3.3 Kỹ thuật tích hợp các DSL theo mối quan tâm vào mô hình miền

Trong phần này, thực hiện đánh giá hiệu quả của phương pháp tích hợp các mối quan tâm vào mô hình miền hợp nhất UDML thông qua các ca nghiên cứu đã được cài đặt và thực nghiệm thành công. Việc đánh giá lần lượt trả lời các câu hỏi nghiên cứu RQ7–RQ10.

RQ7: *Làm thế nào có thể tích hợp một mối quan tâm nền tảng vào mô hình miền?*

UDML trả lời câu hỏi nghiên cứu **RQ7** bằng cơ chế tích hợp dựa trên siêu mô hình, trong đó mỗi mối quan tâm được mô hình hóa như một *concern*

hạng nhất và được gắn vào mô hình miền thông qua bốn siêu khái niệm mở rộng: `DomainModel`, `Concern`, `Annotation` và `Annotable`. Cách tổ chức này cho phép (i) gom nhóm các chú thích của một mối quan tâm trong một `Concern` riêng; (ii) gắn các chú thích đó lên các phần tử miền lõi (gói, lớp, thuộc tính, kết hợp, thao tác) thông qua `Annotable`; và (iii) duy trì ranh giới rõ ràng giữa miền lõi và miền mối quan tâm nhưng vẫn đảm bảo khả năng tích hợp mối quan tâm vào mô hình miền ở đúng vị trí cần thiết. Qua các ca nghiên cứu đã triển khai, việc tích hợp mối quan tâm theo cơ chế này cho thấy mô hình UDML sau hợp nhất vẫn giữ được cấu trúc miền, đồng thời bổ sung được thông tin theo mối quan tâm dưới dạng chú thích mà không làm biến dạng mô hình miền lõi.

RQ8. Làm thế nào có thể biểu diễn miền của mối quan tâm như một aDSL?

Để biểu diễn miền của mối quan tâm như một aDSL bằng cách cách thiết kế miền mối quan tâm theo hai mức: (i) một DSL ngoại sinh DSL_c có siêu mô hình riêng để biểu diễn cú pháp trừu tượng và ràng buộc miền của mối quan tâm; và (ii) một aDSL tương ứng (DSL nội sinh dựa trên chú thích) được xây dựng bằng ánh xạ một–một từ DSL_c. Cách tiếp cận này bảo đảm đồng thời hai yêu cầu: *tính hình thức* ở mức siêu mô hình (dễ kiểm tra hợp dạng, dễ phân tích và chuyển đổi trong MDE) và *tính khả thi* ở mức hiện thực (aDSL nhúng trực tiếp trong OOPL để tham gia sinh mã/thi hành). Các ca thực nghiệm cho thấy cơ chế ánh xạ song ánh giữa mô hình DSL ngoại sinh và mô hình aDSL cho phép duy trì nhất quán giữa mô hình hóa và hiện thực, đồng thời tạo tuyến chuyển đổi M2M/M2T rõ ràng để đưa mối quan tâm vào chuỗi sinh phần mềm.

RQ9: *Làm thế nào các DSL theo mối quan tâm có thể được hợp nhất có hệ thống vào một AST hợp nhất?*

Bằng cơ chế hợp nhất dựa trên AST, trong đó mỗi DSL theo mối quan tâm được chuẩn hóa dưới bộ ba (AS, CS, Sem) và được tổng hợp tăng dần vào UDML lõi bằng thuật toán hợp nhất cây. Ở mức cú pháp trừu tượng, việc hợp nhất được thực hiện bằng cách (i) bổ sung các khái niệm mới hoặc mở rộng khái niệm lõi bằng quan hệ kế thừa; (ii) thêm các thuộc tính và ràng buộc hợp dạng; và (iii) gắn các liên kết xuyên mối quan tâm thông qua các cạnh tham chiếu (`refCons`). Ở mức cú pháp cụ thể, các ký pháp

và thao tác chỉnh sửa được ánh xạ về cùng một AST để bảo đảm đồng bộ đa góc nhìn. Ở mức ngữ nghĩa, mỗi DSL đóng góp một ánh xạ ngữ nghĩa mô-đun và được tổng hợp theo cấu trúc AST hợp nhất. Thực nghiệm cho thấy cách tổ chức "một AST – nhiều góc nhìn" giúp giảm rủi ro lệch mô hình giữa các DSL, đồng thời tạo nền tảng thống nhất cho kiểm tra nhất quán và sinh mã.

R10: *Những cơ chế hợp thành ngôn ngữ và phương pháp có hỗ trợ công cụ nào phù hợp để tích hợp và tiến hóa các mối quan tâm không đồng nhất trong mô hình miền mô-đun, mở rộng?*

Kết quả thực nghiệm cho thấy hai hướng tiếp cận bổ trợ lẫn nhau là phù hợp cho UDML. Thứ nhất, hợp thành dựa trên siêu mô hình và chuyển đổi mô hình (MDE) phù hợp cho việc định nghĩa chính xác miền mối quan tâm, kiểm tra hợp dạng và xây dựng bộ chuyển đổi M2M/M2T nhằm đưa mô hình về dạng thực thi; các công cụ như ATL và Acceleo hỗ trợ hiệu quả việc duy trì ánh xạ giữa DSL ngoại sinh và aDSL, cũng như tiến hóa mô hình thông qua cập nhật luật chuyển đổi. Thứ hai, hợp thành dựa trên AST và chú thích phù hợp khi cần tích hợp các DSL không đồng nhất và mở rộng theo vòng đời, trong đó AST đóng vai trò cấu trúc trung tâm của mô hình hợp nhất.

Đối với các mối quan tâm nhạy cảm như bảo mật, việc sử dụng Event-B như một hậu-end kiểm chứng cung cấp cơ chế kiểm chứng hình thức cho mô hình UDML hợp nhất, cho phép phát hiện xung đột và sai lệch chính sách ở mức ngữ nghĩa.

Thảo luận. Việc áp dụng phương pháp đề xuất trong Mục 4.2 cho các miền bài toán thực tế cho thấy tính khả thi và hiệu quả, song vẫn tồn tại một số mối đe dọa đến tính hợp lệ.

Thứ nhất, phương pháp phụ thuộc vào năng lực biểu đạt của các miền mối quan tâm và khả năng tích hợp chúng vào một mô hình miền hợp nhất có thể thực thi; do đó, UDMM cần tiếp tục được mở rộng để bao quát thêm các mối quan tâm khác.

Thứ hai, tính đúng đắn của mô hình UDM phụ thuộc vào độ chính xác của các phép chuyển đổi từ cú pháp trừu tượng sang cú pháp cụ thể, hiện được đảm bảo thông qua rà soát và kiểm thử các luật chuyển đổi ATL và Acceleo.

Cuối cùng, mã nguồn sinh tự động bằng Java có thể chịu ảnh hưởng từ sai sót trong khung sinh mã; vì vậy, việc rà soát và kiểm thử mã nguồn tiếp tục được thực hiện nhằm nâng cao độ tin cậy của kết quả sinh tự động.

Bên cạnh các kết quả đạt được, một hướng mở rộng tiềm năng được gợi mở từ phân tích ở Mục 2.5 là khai thác các kỹ thuật AI/LLM nhằm hỗ trợ giai đoạn khởi tạo mô hình miền, đặc biệt trong việc sinh nháp đặc tả hoặc đề xuất các thành phần mô hình từ mô tả nghiệp vụ. Cách tiếp cận này có thể góp phần gia tăng mức độ tự động hóa ở bước đầu của quá trình mô hình hóa và giảm chi phí xây dựng mô hình ban đầu. Tuy nhiên, khác với các phép biểu diễn và chuyển đổi có đặc tả tường minh được đề xuất trong luận án, các kết quả do AI/LLM sinh ra mang tính suy diễn và chưa được bảo đảm về ngữ nghĩa. Vì vậy, các đặc tả được sinh ra cần được chuẩn hóa và biểu diễn lại trong các DSL của luận án, sau đó tích hợp vào mô hình miền hợp nhất và kiểm chứng thông qua các cơ chế hình thức đã đề xuất, nhằm bảo đảm tính nhất quán và đúng đắn của mô hình. Hướng nghiên cứu này sẽ được tiếp tục thảo luận trong Chương 7.

6.4 Tổng kết chương

Trong chương này, luận án đã xây dựng và triển khai các công cụ hỗ trợ cho các phương pháp đề xuất, bao gồm kỹ thuật biểu diễn mô hình miền có khả năng thực thi, kỹ thuật tích hợp hành vi và ràng buộc vào mô hình miền hợp nhất, cũng như kỹ thuật sinh tự động bản mẫu phần mềm theo thiết kế hướng miền. Các công cụ này hiện thực hóa chuỗi xử lý hoàn chỉnh từ đặc tả mô hình đầu vào đến các tạo tác phần mềm đầu ra, qua đó minh chứng tính khả thi của các phương pháp đề xuất. Toàn bộ quá trình mô hình hóa, chuyển đổi và sinh mã được đánh giá thông qua các ca nghiên cứu và thực nghiệm nhằm đảm bảo tính đúng đắn, tính nhất quán và khả năng ứng dụng trong thực tiễn phát triển phần mềm.

Chương 7

KẾT LUẬN VÀ HƯỚNG PHÁT TRIỂN

Kỹ thuật biểu diễn và chuyển đổi mô hình cho thiết kế hướng miền. Mặc dù đã tồn tại một số cách tiếp cận nhằm biểu diễn mô hình miền có khả năng thực thi, các cách tiếp cận này thường chỉ tập trung vào từng khía cạnh riêng lẻ. Luận án đề xuất một kỹ thuật biểu diễn mô hình miền hợp nhất, trong đó tích hợp đồng thời khía cạnh hành vi và các ràng buộc OCL. Trong UDML, các khía cạnh này được xem như các DSL theo mối quan tâm và được hợp nhất có hệ thống thành một mô hình miền có khả năng thực thi và được kiểm chứng hình thức. Để hiện thực hóa mô hình miền thực thi, luận án trình bày các kỹ thuật chuyển đổi từ đặc tả mức cao sang các DSL theo mối quan tâm và tiếp tục chuyển đổi sang mã nguồn, được nhúng trực tiếp vào ngôn ngữ lập trình chủ.

7.1 Các kết quả đạt được

Sau quá trình nghiên cứu và giải quyết bài toán này, luận án đã góp phần nâng cao đáng kể trạng thái hiện tại của thiết kế hướng miền bằng cách thu hẹp khoảng cách giữa mô hình và mã nguồn, đồng thời xây dựng được một mô hình miền thống nhất. Trên cơ sở đó, luận án đạt được các kết quả chính sau đây.

1. Thứ nhất, luận án đề xuất phương pháp và công cụ hỗ trợ biểu diễn mô hình miền trong thiết kế hướng miền, cho phép đặc tả một cách

tường minh và nhất quán các khía cạnh cấu trúc, hành vi và ràng buộc nghiệp vụ. Phương pháp này cho phép tăng khả năng diễn đạt của mô hình miền, đảm bảo sự thống nhất của các bên liên quan về yêu cầu phần mềm;

2. Thứ hai, luận án đề xuất kỹ thuật hợp nhất các ngôn ngữ chuyên biệt miền theo các mối quan tâm chuyên biệt vào một mô hình miền hợp nhất có khả năng thực thi. Trên cơ sở đó, luận án xây dựng nền tảng ngữ nghĩa phục vụ kiểm chứng hình thức, nhằm đảm bảo tính nhất quán ngữ nghĩa của mô hình miền hợp nhất;

3. Thứ ba, luận án đề xuất kỹ thuật và công cụ hỗ trợ chuyển đổi mô hình để thao tác trên mô hình miền hợp nhất, cho phép sinh tự động các bản mẫu phần mềm có khả năng thực thi, đồng thời bảo toàn ngữ nghĩa miền và chất lượng chuyển đổi, phù hợp với các kịch bản ca sử dụng cụ thể trong các miền ứng dụng chuyên biệt.

Các kỹ thuật biểu diễn mô hình miền và hợp nhất bằng phương pháp mô hình hóa hợp nhất được kỳ vọng sinh tự động phần mềm, qua đó giúp tạo nhanh bản mẫu phần mềm làm giảm đáng kể chi phí sản xuất và giá thành phần mềm. Thực nghiệm và tiến hành đánh giá tính khả thi của ứng dụng phương pháp trong thực tế bằng các công cụ được phát triển. Ngoài ra, luận án sử dụng các công cụ chuyển đổi mô hình và khung phần mềm JDA thúc đẩy tái sử dụng, nâng cao chất lượng phần mềm làm ra và giúp giảm mức độ sử dụng tài nguyên, nguồn lực của xã hội nói chung.

Các kết quả của luận án đã được công bố trong các công trình khoa học đăng tải tại các kỷ yếu hội thảo quốc tế. Có ý nghĩa hỗ trợ đặc tả chính xác các mô hình miền, đồng thời hiện thực hóa các thao tác tự động trên mô hình miền, bao gồm, sinh tự động mã nguồn, kiểm tra tính tuân thủ với thiết kế và cài đặt. Kết quả này có ý nghĩa gia tăng tự động hóa trong phát triển phần mềm.

7.2 Hướng phát triển tiếp theo

Trong mục này, luận án thảo luận một số hạn chế của nghiên cứu và định hướng phát triển trong tương lai.

- Hoàn thiện công cụ UDML dưới dạng plugin, đồng thời xây dựng các cú pháp biểu diễn đồ họa cho đặc tả hành vi, bảo mật và các mẫu chú thích ràng buộc (CAP), qua đó nâng cao khả năng sử dụng và hỗ trợ người dùng.

- Tiếp tục hoàn thiện kỹ thuật biểu diễn mô hình miền bằng cách mở rộng CAP cho các miền ứng dụng thực tế, thông qua việc xây dựng thư viện các mẫu ràng buộc hỗ trợ biểu thức OCL phức tạp, góp phần nâng cao khả năng áp dụng, tính tích hợp, hiệu năng thực thi và khả năng bảo trì của mô hình miền.

- Mở rộng khung phương pháp cho các DSL chuyên biệt theo từng mối quan tâm, đồng thời hỗ trợ các cơ chế phân quyền động và phụ thuộc ngữ cảnh đa dạng hơn.

- Tích hợp các kỹ thuật AI/LLM vào quy trình mô hình hóa miền nhằm hỗ trợ giai đoạn khởi tạo mô hình, đặc biệt trong việc sinh nháp đặc tả và gợi ý các thành phần từ mô tả nghiệp vụ; các đặc tả này cần được chuẩn hóa trong các DSL và kiểm chứng bằng các cơ chế hình thức để đảm bảo tính đúng đắn và nhất quán.

DANH MỤC CÁC CÔNG TRÌNH KHOA HỌC CỦA TÁC GIẢ LIÊN QUAN TỚI LUẬN ÁN

1. [V1] Duc-Hanh Dang, Duc Minh Le, Van-Vinh Le. AGL: Incorporating behavioral aspects into domain-driven design. *Information and Software Technology*, 163, 107284, 2023.
DOI: 10.1016/j.infsof.2023.107284. (WoS/Scopus, Q1)

2. [V2] Van-Vinh Le, Nghia-Trong Be, Duc-Hanh Dang. On Automatic Generation of Executable Domain Models for Domain-Driven Design. *Proc. 15th Int. Conf. Knowledge and Systems Engineering (KSE)*, IEEE, 2023. *DOI: 10.1109/KSE59128.2023.10299453*. (WoS/Scopus)

3. [V3] Van-Vinh Le, Duc-Hanh Dang. An Approach to Composing Concerns for an Executable Unified Domain Model. *Proc. 18th Int. Conf. Research, Innovation and Vision for the Future (RIVF)*, IEEE, pp. 415–419, 2024. *DOI: 10.1109/RIVF64335.2024.11009109*.
(WoS/Scopus)

4. [V4] Le Van Vinh, Dang Duc Hanh. RM2UDM: A method for automatically generating functional prototypes from requirement models. *Proc. 17th National Conf. Fundamental and Applied Information Technology Research (FAIR)*, pp. 734–741, 2024.
ISBN: 978-604-357-304-6, DOI: 10.15625/vap.2024.0271.

5. [V5] Van-Vinh Le, Nhat-Hoang Nguyen, Duc-Quyen Nguyen, Duc-Hanh Dang. A Method for Composing Concerns into a Unified Domain Model in Domain-Driven Design. *Proc. 14th Int. Symposium on Information and Communication Technology (SOICT)*, Springer, 2025. (WoS/Scopus, accepted)

6. [V6] Van-Vinh Le, Duc-Hanh Dang. Constraint Annotation Patterns for Prototype Generation within Domain-Driven Design.
e-Informatica Software Engineering Journal, 2025. *ISSN: 2084-4840*.
(WoS/Scopus, Q3, submitted)

7. [V7] Van-Vinh Le, Nhat-Hoang Nguyen, Duc-Quyen Nguyen, Duc-Hanh Dang. A Semantic Framework and Tool Support for Unified Executable

Domain Models in UDML: A Case Study on the RBAC Concern. *VNU Journal of Science: Computer Science and Communication Engineering. Vol 42 No 1, pages 79-114, 2026. ISSN: 2615-9260, DOI: 10.25073/2588-1086/vnucsce.6743.*

TÀI LIỆU THAM KHẢO

[1] *Actifsource*. Actifsource AG, Switzerland - all rights reserved., 2017. URL https://www.actifsource.com/_downloads/ActifsourceManual_ActifsourceUserManual.pdf.

[2] Jean-Raymond Abrial. *Modeling in Event-B: System and Software Engineering*. Cambridge University Press, Cambridge, 2010. ISBN 978-0-521-89556-9.

[3] Jean-Raymond Abrial, Michael Butler, Stefan Hallerstede, Thai Son Hoang, Farhad Mehta, and Laurent Voisin. Rodin: an open toolset for modelling and reasoning in Event-B. *International Journal on Software Tools for Technology Transfer*, 12(6):447–466, November 2010. ISSN 1433-2779, 1433-2787. doi: 10.1007/s10009-010-0145-y.

[4] Muhammad Umar Aftab, Zhiguang Qin, Negalign Wake Hundera, Oluwasanmi Ariyo, Zakria, Ngo Tung Son, and Tran Van Dinh. Permission-based separation of duty in dynamic role-based access control model. *Symmetry*, 11(5):669, 2019.

[5] Fatimah Akeel, Asieh Salehi Fathabadi, Federica Paci, Andrew Gravell, and Gary Wills. Formal Modelling of Data Integration Systems Security Policies. *Data Science and Engineering*, 1(3):139–148, September 2016. ISSN 2364-1541. doi: 10.1007/s41019-016-0016-y.

[6] Hessa Ali Alhamad and Mohammad Mahdi Hassan. Aspect-Oriented Models-Based Framework to Secure Intelligent Systems. In *Proc. 2022 8th Int. Conf. Computer Technology Applications*, ICCTA '22, pages 249–262, New York, NY, USA, September 2022. Association for Computing Machinery. ISBN 978-1-4503-9622-6. doi: 10.1145/3543712.3543726.

[7] Shaukat Ali, Muhammad Zohaib Iqbal, Andrea Arcuri, and Lionel C. Briand. Generating Test Data from OCL Constraints with Search Techniques. *IEEE Transactions on Software Engineering*, 39(10): 1376–1402, October 2013. ISSN 1939-3520. doi: 10.1109/TSE.2013.17. Conference Name: IEEE Transactions on Software Engineering.

[8] Andrzej Wasowski and Thorsten Berger. *Domain-Specific Languages Effective Modeling, Automation, and Reuse*. Springer Cham, 2023. ISBN 978-3-031-23668-6.

[9] ANSI/INCITS. Information technology – role-based access control. Technical Report INCITS 359-2012, American National Standards Institute (ANSI), May 2012. URL https://webstore.ansi.org/standards/incits/incits3592012. Reaffirmed as INCITS 359-2012 (R2022).

[10] I. Antović, S. Vlajić, M. Milić, D. Savić, and V. Stanojević. Model and software tool for automatic generation of user interface based on use case and data model. *Institution of Engineering and Technology*, 6(6): 559 – 573, 2012. ISSN Print ISSN 1751-8806. Online ISSN 1751-8814. doi: 10.1049/iet-sen.2011.0060.

[11] David Basin, Jurgen Doser, and Torsten Lodderstedt. Model driven security: From UML models to access control infrastructures. *ACM Trans. Softw. Eng. Methodol.*, 15(1):39–91, January 2006. ISSN 1049-331X. doi: 10.1145/1125808.1125810.

[12] Nelly Bencomo, Jordi Cabot, et al. Abstraction engineering. *arXiv preprint arXiv:2408.14074*, August 2024.

[13] Alexander Bergmayr, Michael Grossniklaus, Manuel Wimmer, and Gerti Kappel. Leveraging annotation-based modeling with Jump. *Software & Systems Modeling*, 17(1):65–89, February 2018. ISSN 1619-1366, 1619-1374.

[14] Mario Luca Bernardi, Giuseppe Antonio Di Lucca, and Damiano Distante. Model-driven fast prototyping of RIAs: From conceptual models to running applications. In *Proc. 2014 Int. Conf. Advances in Computing, Communications and Informatics (ICACCI)*, pages 250–258, 2014. doi: 10.1109/ICACCI.2014.6968522.

[15] Dines Bjørner. Domain modelling: A foundation for software development. In *Theories of Programming and Formal Methods: Essays Dedicated to He Jifeng on the Occasion of His 80th Birthday*, pages 165–210. Springer, 2023. doi: 10.1007/978-3-031-40436-8_7.

[16] Grady Booch, Robert A. Maksimchuk, Michael W. Engle, Bobbi J. Young, Jim Connallen, and Kelli A. Houston. Object-oriented analysis and design with applications, third edition. *ACM SIGSOFT Software Engineering Notes*, 33(5):29–29, August 2008. ISSN 0163-5948. doi: 10.1145/1402521.1413138.

[17] Marco Brambilla, Jordi Cabot, and Manuel Wimmer. *Model-Driven Software Engineering in Practice*. Morgan&Claypool, 2nd edition, 2017. ISBN 978-1-62705-708-0.

[18] Achim D. Brucker, Matthias P. Krieger, Delphine Longuet, and Burkhart Wolff. A Specification-Based Test Case Generation Method for UML/OCL. In *Models in Software Engineering*, pages 334–348. Springer, Berlin, Heidelberg, 2011. ISBN 978-3-642-21210-9. doi: 10.1007/978-3-642-21210-9_33. ISSN: 1611-3349.

[19] Antonio Bucchiarone, Antonio Cicchetti, Federico Ciccozzi, and Alfonso Pierantonio. *Domain-specific languages in practice: with JetBrains MPS*. Springer Nature, 2021.

[20] Antonio Bucchiarone, Juri Di Rocco, Damiano Di Vincenzo, and Alfonso Pierantonio. From OCL to JSX: declarative constraint modeling in modern SaaS tools, September 2025.

[21] Frank Budinsky. *Eclipse Modeling Framework: A Developer's Guide*. Addison-Wesley, 2004. ISBN 978-0-13-142542-2.

[22] Dennis M. Buede and William D. Miller. *The Engineering Design of Systems: Models and Methods*. Wiley, 4th edition, 2024. ISBN 978-1-119-98401-6.

[23] Fabian Büttner and Martin Gogolla. On ocl-based imperative model transformation. *Electronic Communications of the EASST*, 29, 2010.

[24] J. Cabot and E. Teniente. Transformation techniques for OCL constraints. *Science of Computer Programming*, 68(3):179–195, October 2007. ISSN 0167-6423. doi: 10.1016/j.scico.2007.05.001.

[25] Jordi Cabot and Martin Gogolla. Object Constraint Language (OCL): A Definitive Guide. In *Formal Methods for Model-Driven Engineering*, volume 7320, pages 58–90. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012. ISBN 978-3-642-30981-6 978-3-642-30982-3.

[26] Edmilson Campos, Uirá Kulesza, Marília Freire, and Eduardo Aranha. A Generative Development Method with Multiple Domain-Specific Languages. In *M. (eds) Product-Focused Software Process Improvement. PROFES 2014. Lecture Notes in Computer Science*, volume 8892, pages 178–193. Springer, Cham, 2014. ISBN 978-3-319-13834-3.

[27] Joanna Chimiak-Opoka, Birgit Demuth, Andreas Awenius, Dan Chiorean, Sebastien Gabel, Lars Hamann, and Edward Willink. OCL tools report based on the ide4OCL feature model. *Electronic Communications of the EASST*, 44, 2011.

[28] Tony Clark and Jos B. Warmer, editors. *Object modeling with the OCL: the rationale behind the Object Constraint Language*. Number 2263 in Lecture notes in computer science. Springer, Berlin ; New York, 2002. ISBN 978-3-540-43169-5.

[29] Benoit Combemale, Robert France, Jean-Marc Jézéquel, Bernhard Rumpe, James Steel, and Didier Vojtisek. *Engineering Modeling Languages: Turning Domain Knowledge into Tools*. CRC Press, November 2016. ISBN 978-1-315-38793-2.

[30] Krzysztof Czarnecki. Overview of Generative Software Development. In Jean-Pierre Banâtre, Pascal Fradet, Jean-Louis Giavitto, and Olivier Michel, editors, *Unconventional Programming Paradigms*,

pages 326–341, Berlin, Heidelberg, 2005. Springer. ISBN 978-3-540-31482-0. doi: 10.1007/11527800_25.

[31] Irene Córdoba-Sánchez and Juan De Lara. Ann: A domain-specific language for the effective design and validation of Java annotations. *Computer Languages, Systems & Structures*, 45:164–190, April 2016. ISSN 14778424.

[32] Alberto Rodrigues da Silva. Model-driven engineering: A survey supported by the unified conceptual model. *Computer Languages, Systems & Structures*, 43:139–155, October 2015. doi: 10.1016/j.cl.2015.06.001.

[33] Carolina Dania and Manuel Clavel. OCL2MSFOL: a mapping to many-sorted first-order logic for efficiently checking the satisfiability of OCL constraints. In *Proceedings of the ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems*, pages 65–75, Saint-malo France, October 2016. ACM. ISBN 978-1-4503-4321-3. doi: 10.1145/2976767.2976774.

[34] Istvan David, Kousar Aslam, Ivano Malavolta, and Patricia Lago. Collaborative Model-Driven Software Engineering — A systematic survey of practices and needs in industry. *Journal of Systems and Software*, 199:111626, May 2023. ISSN 0164-1212. doi: 10.1016/j.jss.2023.111626.

[35] Dr Birgit Demuth. OCL (Object Constraint Language) by Example. page 60, 2009.

[36] Qing Duan, Junhui Liu, Zhihong Liang, Hongwei Kang, and Xingping Sun. An Analysis of the Keys to the Executable Domain-Specific Model. In Srikanta Patnaik and Xiaolong Li, editors, *Proc. Int. Conf. Soft Computing Techniques and Engineering Application*, pages 567–574, New Delhi, 2014. Springer India. ISBN 978-81-322-1695-7. doi: 10.1007/978-81-322-1695-7_67.

[37] Edmilson Campos, Neto, Marília Aranha, Freire, Uirá, Kulesza, Adorilson, Bezerra, and Eduardo, Aranha. Composition of Domain Specific Modeling Languages-An Exploratory Study.. In *Proc. 1st Int. Conf. Model-Driven Engineering*, pages 149–156, 2013.

[38] Tobias Eisenreich, Husein Jusic, and Stefan Wagner. Automating domain-driven design: Experience with a prompting framework. In *Proceedings of the IEEE/ACM International Conference on Software Engineering Workshops (ICSE Workshops)*. IEEE, 2023.

[39] Eric Evans. *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Addison-Wesley Professional, 2004.

[40] Agung Fatwanto and Clive Boughton. Analysis, Specification and Modeling of Functional Requirements for Translative Model-Driven Development. In *Proc. 2008 Int. Symposium Conf. Knowledge Acquisition and Modeling*, pages 859–863. IEEE, October 2008. ISBN 978-0-7695-3488-6. doi: 10.1109/KAM.2008.185.

[41] Alessio Ferrari, Sallam Abualhaija, and Chetan Arora. Model Generation with LLMs: From Requirements to UML Sequence Diagrams. In *2024 IEEE 32nd International Requirements Engineering Conference Workshops (REW)*, pages 291–300, June 2024. doi: 10.1109/REW61692.2024.00044. URL https://ieeexplore.ieee.org/document/10628665. ISSN: 2770-6834.

[42] Elvis Foster and Bradford Towle Jr. *Software Engineering: A Methodical Approach, 2nd Edition*. Auerbach Publications, New York, 2 edition, July 2021. ISBN 978-0-367-74602-5. doi: 10.1201/9780367746025.

[43] Martin Fowler. *Domain-Specific Languages*. Addison-Wesley Professional, 1st edition, September 2010. ISBN 978-0-13-139280-9.

[44] Alfonso Fuggetta and Elisabetta Di Nitto. Software process. In *Future of Software Engineering Proceedings*, FOSE 2014, pages 1–12, New York, NY, USA, May 2014. Association for Computing Machinery. ISBN 978-1-4503-2865-4. doi: 10.1145/2593882.2593883.

[45] Polydoros Giannouris and Sophia Ananiadou. NOMAD: A Multi-Agent LLM System for UML Class Diagram Generation from Natural Language Requirements, November 2025. URL https://arxiv.org/abs/2511.22409v1.

[46] Fernanda Gonzalez-Lopez, Guillermo Bustos, Jorge Munoz-Gama, and Marcos Sepulveda. Domain model based design of business process architectures. *Applied Sciences*, 12(5):2563, 2022.

[47] James Gosling, Bill Joy, Guy L. Steele, Gilad Bracha, and Alex Buckley. *The Java Language Specification, Java SE 8 Edition*. Addison-Wesley Professional, 1st edition, April 2014. ISBN 978-0-13-390069-9.

[48] Olga Goubali, Patrick Girard, Laurent Guittet, Alain Bignon, Djamal Kesraoui, Pascal Berruet, and Jean-Frédéric Bouillon. Designing Functional Specifications for Complex Systems. In *Human-Computer Interaction. Theory, Design, Development and Practice . HCI 2016. Lecture Notes in Computer Science()*, volume 9731, pages 166–177. Springer, Cham, 2016. ISBN 978-3-319-39509-8. doi: 10.1007/978-3-319-39510-4_16.

[49] Jesse Griffin. Domain-Driven Laravel, Learn to Implement Domain-Driven Design Using Laravel. January 2021. ISSN 978-1-4842-6022-7. doi: 10.1007/978-1-4842-6023-4.

[50] Raffaela Groner, Peter Bellmann, Stefan Höppner, Patrick Thiam, Friedhelm Schwenker, Hans A. Kestler, and Matthias Tichy. Enhanced performance prediction of ATL model transformations. *Performance Evaluation*, 164:102413, May 2024. ISSN 0166-5316. doi: 10.1016/j.peva.2024.102413. URL https://www.sciencedirect.com/science/article/pii/S016653162400018X.

[51] Arne Haber, Markus Look, Antonio Navarro Perez, Pedram Mir Seyed Nazari, Bernhard Rumpe, Steven Völkel, and Andreas Wortmann. Integration of heterogeneous modeling languages via extensible and composable language components. In *2015 3rd Int. Conf. on Model-Driven Engineering and Software Development (MODELSWARD)*, pages 19–31. IEEE, 2015.

[52] Lars Hamann, Oliver Hofrichter, and Martin Gogolla. OCL-Based Runtime Monitoring of Applications with Protocol State Machines. In *Modelling Foundations and Applications*, volume 7349, pages 384–399. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012. ISBN 978-3-642-31490-2 978-3-642-31491-9. doi: 10.1007/978-3-642-31491-9_29. Series Title: Lecture Notes in Computer Science.

[53] Fitash Ul Haq and Jordi Cabot. B-OCL: An Object Constraint Language Interpreter in Python, March 2025. arXiv:2503.00944 [cs].

[54] Lukas Heiland, Marius Hauser, and Justus Bogner. Design Patterns for AI-based Systems: A Multivocal Literature Review and Pattern Repository. In *2023 IEEE/ACM 2nd Int. Conf. AI Engineering – Software Engineering for AI (CAIN)*, pages 184–196, May 2023. doi: 10.1109/CAIN58948.2023.00035.

[55] Anders Hejlsberg, Mads Torgersen, Scott Wiltamuth, and Peter Golde. *The C# Programming Language (Covering C# 4.0)*. Addison-Wesley Professional, October 2010. ISBN 978-0-13-248172-4.

[56] Frank Hilken and Lars Hamann. History of the USE Tool 20 Years of UML/OCL Modeling Made in Germany. *The Journal of Object Technology*, 19(3):3:1, 2020. ISSN 1660-1769. doi: 10.5381/jot.2020.19.3.a20.

[57] Jeffrey A. Hoffer, Joey George, and Joseph A. Valacich. *Modern Systems Analysis and Design*. Pearson, 7a edition, 2013. ISBN 978-0-13-299130-8.

[58] Vincent C Hu, Rick Kuhn, and Dylan Yaga. Verification and test methods for access control policiesmodels. Technical Report NIST SP 800-192, National Institute of Standards and Technology, Gaithersburg, MD, June 2017.

[59] Sandeep Kumar Jaiswal and Rohit Agrawal. Domain-Driven Design (DDD)- Bridging the Gap between Business Requirements and

Object-Oriented Modeling. *Int. Innovative Research in Engineering and Management*, 11(2):79–83, 2024. ISSN 2350-0557.

[60] Manfred Jeusfeld, Matthias Jarke, and John Mylopoulos. *Metamodeling for Method Engineering*. MIT Press, 2009. ISBN 978-0-262-10108-0. Google-Books-ID: i1kRtAEACAAJ.

[61] Zihan et al. Jiang. A survey on large language models for software engineering. *arXiv preprint arXiv:2308.10620*, 2023.

[62] Frédéric Jouault, Freddy Allilaire, Jean Bézivin, and Ivan Kurtev. ATL: A model transformation tool. *Science of computer programming*, 72(1-2):31–39, 2008.

[63] Stefan Kapferer and Olaf Zimmermann. Domain-Driven Service Design. In Schahram Dustdar, editor, *Service-Oriented Computing*, Communications in Computer and Information Science, pages 189–208, Cham, 2020. Springer International Publishing. ISBN 978-3-030-64846-6. doi: 10.1007/978-3-030-64846-6_11.

[64] Elavarasi Kesavan. The Evolution of Software Design Patterns: An In-Depth Review. *International Journal of Innovations in Science, Engineering And Management*, pages 163–167, 2025.

[65] Meriem Kherbouche and Balint Molnar. Formal Model Checking and Transformations of Models Represented in UML with Alloy. In *Modelling to Program*, pages 127–136, Cham, 2021. Springer International Publishing. ISBN 978-3-030-72696-6.

[66] Anneke Kleppe. *Software Language Engineering: Creating Domain-Specific Languages Using Metamodels*. Addison-Wesley Professional, 1 edition, December 2008. ISBN 978-0-321-55345-4.

[67] Preyanoot Kluisritrakul and Yachai Limpiyakorn. Generation of Java Code from UML Sequence and Class Diagrams. In *Proc. 7th Int. Conf. Information Science and Applications (ICISA 2016)*, volume 376, pages 1117–1125. Springer. K.J. Kim and N. Joukov (eds.), 2016. doi: DOI:10.1007/978-981-10-0557-2_106.

[68] Craig Larman. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development*. Pearson, Upper Saddle River, NJ, 2010. ISBN 978-0-13-148906-6.

[69] Michael Lawley and Jim Steel. Practical declarative model transformation with tefkat. In *Proc. int. conf. Satellite Events at the MoDELS*, MoDELS'05, pages 139–150, Berlin, Heidelberg, October 2005. Springer-Verlag. ISBN 978-3-540-31780-7. doi: 10.1007/11663430_15.

[70] Duc Minh Le, Duc-Hanh Dang, and Viet-Ha Nguyen. On design using annotation-based domain specific language. *Computer Languages, Systems & Structures*, 54:199–235, December 2018. ISSN 14778424.

[71] Duc Minh Le, Duc-Hanh Dang, and Ha Thanh Vu. jDomainApp: A Module-Based Domain-Driven Software Framework. In *Proc. 10th Int. Symposium on Information and Communication Technology (SoICT)*, pages 399–406, December 2019. ISBN 978-1-4503-7245-9.

[72] Duc Minh Le, Duc-Hanh Dang, and Viet-Ha Nguyen. Generative software module development for domain-driven design with annotation-based domain specific language. *Information and Software Technology*, 120:106–239, 2020.

[73] Yunliang Li, Zhiqiang Du, Yanfang Fu, and Liangxin Liu. Role-based access control model for inter-system cross-domain in multi-domain environment. *Applied Sciences*, 12(24):13036, 2022.

[74] Barbara Liskov and John Guttag. *Program development in Java: abstraction, specification, and object-oriented design*. Addison-Wesley Professional, 2000. ISBN 978-0-201-65768-5.

[75] Shugang Liu, Jinfeng Chen, Yifei Liu, and Pengrui Lv. Mapping of UML Diagrams to Executable Code. pages 9–16. Atlantis Press, April 2017. ISBN 978-94-6252-327-2. doi: 10.2991/icmse-17.2017.2.

[76] Wissam Mallouli, Jean-Marie Orset, Ana Cavalli, Nora Cuppens, and Frederic Cuppens. A formal approach for testing security rules. In *Proc. 12th ACM symposium on Access control models and technologies - SACMAT '07*, page 127, Sophia Antipolis, France, 2007. ACM Press. ISBN 978-1-59593-745-2. doi: 10.1145/1266840.1266860.

[77] Neel Mani, Markus Helfert, and Claus Pahl. A Domain-specific Rule Generation Using Model-Driven Architecture in Controlled Variability Model. *Procedia Computer Science*, 112:2354–2362, 2017. ISSN 1877-0509. doi: https://doi.org/10.1016/j.procs.2017.08.206.

[78] Ismail Mendil, Yamine Ait-Ameur, Neeraj Kumar Singh, Guillaume Dupont, Dominique Mery, and Philippe Palanque. Formal domain-driven system development in Event-B: Application to interactive critical systems. *Journal of Systems Architecture*, 135:102798, February 2023. ISSN 1383-7621. doi: 10.1016/j.sysarc.2022.102798.

[79] Marcelo Paternostro Ed Dave Steinberg Merks, Frank Budinsky. *EMF: Eclipse Modeling Framework Second Edition*. ISBN 978-0-321-33188-5.

[80] Dominique Mery. The Event B Modelling Method. *Erasmus*, 2011.

[81] Judith Michael, Loek Cleophas, and et al. Zschaler. Model-Driven Engineering for Digital Twins: Opportunities and Challenges. *Systems Engineering*, 28(5):659–670, 2025. ISSN 1520-6858. doi: 10.1002/sys.21815. _eprint: https://incose.onlinelibrary.wiley.com/doi/pdf/10.1002/sys.21815.

[82] Aya Khaled Youssef Sayed Mohamed, Dagmar Auer, Daniel Hofer, and Josef Küng. A systematic literature review for authorization and access control: definitions, strategies and models. *International Journal of Web Information Systems*, 18(2-3):156–180, August 2022. ISSN 1744-0084. doi: 10.1108/IJWIS-04-2022-0077.

[83] Lionel Montrieux, Yijun Yu, Michel Wermelinger, and Zhenjiang Hu. Issues in representing domain-specific concerns in model-driven engineering. In *2013 5th International Workshop on Modeling in Software Engineering (MiSE)*, pages 1–6. IEEE, 2013.

[84] Maryam I. Mukhtar and Bashir S. Galadanci. Automatic code generation from UML diagrams: the state-of-the-art. *Science World Journal*, 13(4):47–60, 2019. ISSN 1597-6343.

[85] Iftikhar Azim Niaz and Jiro Tanaka. An Object-Oriented Approach to Generate Java Code from UML Statecharts. *International Journal of Computer & Information Science*, 6(2):83–98, 2005.

[86] Erik et al. Nijkamp. Codegen: An open large language model for code generation. *arXiv preprint arXiv:2203.13474*, 2022.

[87] Carlos Noguera and Laurence Duchien. Annotation Framework Validation Using Domain Models. In *Model Driven Architecture – Foundations and Applications*, pages 48–62. Springer, Berlin, Heidelberg, 2008. ISBN 978-3-540-69100-6. doi: 10.1007/978-3-540-69100-6_4. ISSN: 1611-3349.

[88] Siegfried Nolte. *QVT-Relations Language*. Springer Science & Business Media, 2009.

[89] Milan Nosál', Matús Sulír, and Ján Juhár. Language composition using source code annotations. *Computer Science and Information Systems*, 13(3):707–729, 2016. ISSN 1820-0214, 2406-1018.

[90] OMG. *Object Constraint Language 4*. 2014.

[91] OMG. *Unified Modeling Language 2.5.1*. Object Management Group, 2017.

[92] OpenAI. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*, 2023.

[93] Orcun Oruc. Role-based embedded domain-specific language for collaborative multi-agent systems through blockchain technology. In *9th Int. Conf. Security, Privacy and Trust Management (SPTM)*, 2021.

[94] Samir Ouchani and Mourad Debbabi. Specification, verification, and quantification of security in model-based systems. *Computing*, 97(7):691–711, July 2015. ISSN 0010-485X, 1436-5057. doi: 10.1007/s00607-015-0445-x.

[95] Javier Paniza. Openxava: Automatic frontend engine for java. https://openxava.org/, 2021. URL https://openxava.org/.

[96] Marijana Rackov, Sebastijan Kaplar, Milorad Filipović, and Gordana Milosavljević. Java code generation based on ocl rules. In *6th International Conference on Information Society and Technology*, pages 191–196, 2016.

[97] Qusai Ramadan, Mattia Salnitriy, Daniel Struber, Jan Jurjens, and Paolo Giorgini. From Secure Business Process Modeling to Design-Level Security Verification. In *2017 ACM/IEEE 20th Int. Conf. Model Driven Engineering Languages and Systems (MODELS)*, pages 123–133, Austin, TX, September 2017. IEEE. ISBN 978-1-5386-3492-9. doi: 10.1109/MODELS.2017.10.

[98] Abhishek Rawat, Raghuraj Singh Suryavanshi, Vishal Nagar, and Diwakar Yadav. Formal Development of Blockchain Enabled E Healthcare System Using Event-B. *Available at SSRN 5167558*, 2025.

[99] Carlos Rodríguez, Mario Sánchez, and Jorge Villalobos. Executable model composition: a multilevel approach. In *Proc. 2011 ACM Symposium on Applied Computing (SAC)*, 11, pages 877–884. Association for Computing Machinery, March 2011. ISBN 978-1-4503-0113-8.

[100] Markus Roggenbach, Antonio Cerone, Bernd-Holger Schlingloff, Gerardo Schneider, and Siraj Ahmed Shaikh. Correction to: Formal Methods for Software Engineering. In Markus Roggenbach, Antonio Cerone, Bernd-Holger Schlingloff, Gerardo Schneider, and Siraj Ahmed Shaikh, editors, *Formal Methods for Software Engineering: Languages, Methods, Application Domains*, pages C1–C1. Springer International Publishing, Cham, 2022. ISBN 978-3-030-38800-3.

[101] António Miguel Rosado da Cruz and João Faria. Automatic Generation of User Interface Models and Prototypes from Domain and Use Case Models. Intech, user interfaces, rita matrai (ed.) edition, 2010. ISBN 978-953-307-084-1. doi: 10.5772/9498.

[102] Djaber Rouabhia and Ismail Hadjadj. Behavioral Augmentation of UML Class Diagrams: An Empirical Study of Large Language Models for Method Generation, June 2025.

[103] Subhrojyoti Roy Chaudhuri, Swaminathan Natarajan, Amar Banerjee, and Venkatesh Choppella. Methodology to develop domain specific modeling languages. In *Proc. 17th ACM SIGPLAN Int. Workshop on Domain-Specific Modeling*, DSM 2019, pages 1–10, New York, NY, USA, October 2019. Association for Computing Machinery. ISBN 978-1-4503-6984-8. doi: 10.1145/3358501.3361235.

[104] Per Runeson and Martin Höst. Guidelines for conducting and reporting case study research in software engineering. *Empirical Software Engineering*, 14(2):131–164, April 2009. ISSN 1573-7616. doi: 10.1007/s10664-008-9102-8.

[105] Mark Sagar. Creating Models for Simulating the Face. In Jon Whittle, Tony Clark, and Thomas Kühne, editors, *Model Driven Engineering Languages and Systems*, pages 394–394, Berlin, Heidelberg, 2011. Springer. ISBN 978-3-642-24485-8. doi: 10.1007/978-3-642-24485-8_28.

[106] Rubén Salado-Cid, Antonio Vallecillo, Kamram Munir, and José Raúl Romero. SWEL: A Domain-Specific Language for Modeling Data-Intensive Workflows. *Business & Information Systems Engineering*, 66(2):137–160, April 2024. ISSN 1867-0202. doi: 10.1007/s12599-023-00826-7.

[107] Josue Sangabriel-Alarcón, Jorge Octavio Ocharán-Hernández, Karen Cortés-Verdín, and Xavier Limón. Domain-Driven Design for Microservices Architecture Systems Development: A Systematic Mapping Study. pages 25–34. IEEE Computer Society, November 2023. ISBN 979-8-3503-2883-7. doi: 10.1109/CONISOFT58849.2023.00014.

[108] Shane Sendall and Wojtek Kozaczynski. Model Transformation: The Heart and Soul of Model-Driven Software Development. *IEEE Software*, 20(05):42–45, September 2003. ISSN 0740-7459. doi: 10.1109/MS.2003.1231150.

[109] B. Shafiq, A. Masood, J. Joshi, and A. Ghafoor. A Role-Based Access Control Policy Verification Framework for Real-Time Systems. In *10th IEEE Inte. Work. Object-Oriented Real-Time Dependable Systems*, pages 13–20, Sedona, AZ, USA, 2005. IEEE. ISBN 978-0-7695-2347-7. doi: 10.1109/WORDS.2005.11.

[110] Anthony Simons. ReMoDeL: A Pure Functional Object-Oriented Concept Language for Models, Metamodels and Model Transformation:. In *Proc. 13th Int. Conf. Model-Based Software and Systems Engineering*, pages 242–249, Porto, Portugal, 2025. SCITEPRESS - Science and Technology Publications. ISBN 978-989-758-729-0. doi: 10.5220/0013184700003896.

[111] Colin Snook, Michael Butler, Thai Son Hoang, Asieh Salehi Fathabadi, and Dana Dghaym. Developing the UML-B Modelling Tools. In Paolo Masci, Cinzia Bernardeschi, Pierluigi Graziani, Mario Koddenbrock, and Maurizio Palmieri, editors, *Software Engineering and Formal Methods. SEFM 2022 Collocated Workshops*, pages 181–188, Cham, 2023. Springer International Publishing. ISBN 978-3-031-26236-4. doi: 10.1007/978-3-031-26236-4_16.

[112] K. Sohr, M. Drouineaud, G.-J. Ahn, and M. Gogolla. Analyzing and Managing Role-Based Access Control Policies. *IEEE Transactions on Knowledge and Data Engineering*, 20(7):924–939, July 2008. ISSN 1041-4347. doi: 10.1109/TKDE.2008.28.

[113] Frank P. M. Stappers, Michel A. Reniers, and Sven Weber. Transforming SOS Specifications to Linear Processes. In *Formal Methods for Industrial Critical Systems*, volume 6959, pages 196–211. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011. ISBN 978-3-642-24430-8 978-3-642-24431-5.

[114] James Gosling Bill Joy Guy Steele and Gilad Bracha Alex Buckley. *The Java language Specificition*. Bhaskarjyoti Saikia, 2020.

[115] Harald Störrle. Improving the Usability of OCL as an Ad-hoc Model Querying Language: 13th International Workshop on OCL, Model Constraint and Query Languages (OCL 2013). *Proc. MODELS 2013 OCL Workshop*, pages 83–92, 2013.

[116] E. V. Sunitha and Philip Samuel. Object constraint language for code generation from activity models. *Information and Software Technology*, 103:92–111, November 2018. ISSN 0950-5849. doi: 10.1016/j.infsof.2018.06.010.

[117] Witold Suryn. *Software Quality Engineering: A Practitioner's Approach*. John Wiley & Sons, December 2013. ISBN 978-1-118-83018-5.

[118] Thakur and U.S. Pandey. The Role of Model-View Controller in Object Oriented Software Development. *Nepal Journal of Multidisciplinary Research*, 2:1–6, November 2019. doi: 10.3126/njmr.v2i2.26279.

[119] The Apache Software Foundation. Apache Causeway: Framework for rapidly developing domain-driven apps in Java. https://causeway.apache.org/, 2023.

[120] Arie Van Deursen, Eelco Visser, and Jos Warmer. Model-driven software evolution: A research agenda. In *Proc. 1st int.l workshop on model-driven software evolution*, pages 41–49. MoDSE Nantes, France, 2007.

[121] Vaughn Vernon. *Implementing domain-driven design*. Addison-Wesley Professional, 1 er edition, 2013. ISBN 978-0-321-83457-7.

[122] Vaughn Vernon. *Domain-Driven Design Distilled*. Addison-Wesley Professional, 1st edition edition, May 2016. ISBN 978-0-13-443442-1.

[123] Inna Vistbakka and Elena Troubitsyna. Towards Integrated Modelling of Dynamic Access Control with UML and Event-B. *Electronic Proceedings in Theoretical Computer Science*, 271:105–116, May 2018. ISSN 2075-2180. doi: 10.4204/EPTCS.271.8.

[124] Sunitha Edacheril Viswanathan and Philip Samuel. Automatic code generation using unified modeling language activity and sequence models. *IET Software*, 10(6):164–172, 2016. ISSN 1751-8814. doi: 10.1049/iet-sen.2015.0138.

[125] Markus Voelter. Language and IDE Modularization and Composition with MPS. volume 7680 of *Lecture Notes in Computer Science (LNPSE)*, pages 383–430. Springer, Berlin, Heidelberg, 2013. ISBN 978-3-642-35992-7.

[126] M. Völter, S Benz, C Dietrich, B Engelmann, M Helander, LCL Kats, E Visser, and GH Wachsmuth. *DSL Engineering - Designing, implementing and using domain-specific languages*. M Volter / DSL-Book.org, Stuttgart, Germany, 2013.

[127] Markus Völter, Thomas Stahl, Jorn Bettin, Arno Haase, and Simon Helsen. *Model-driven software development: technology, engineering, management*. John Wiley & Sons, 2013.

[128] J. B. Warmer and A. G. Kleppe. Building a Flexible Software Factory Using Partial Domain Specific Models. In *6th OOPSLA Workshop on Domain-Specific Modeling, DSM*, pages 15–22. University of Jyvaskyla, 2006. ISBN 951-39-2631-1.

[129] E. D. Willink. An extensible OCL virtual machine and code generator. In *Proceedings of the 12th Workshop on OCL and Textual Modelling*, OCL '12, pages 13–18, New York, NY, USA, September 2012. Association for Computing Machinery. ISBN 978-1-4503-1799-3. doi: 10.1145/2428516.2428519.

[130] Edward D. Willink. Challenges for code generated OCL execution. In *Proc. 25th Int. Conf. Model Driven Engineering Languages and Systems: Companion Proceedings*, MODELS '22, pages 872–881, New York, NY, USA, November 2022. Association for Computing Machinery. ISBN 978-1-4503-9467-3. doi: 10.1145/3550356.3561537.

[131] Willow. Model-Driven Rapid Prototyping for Control Algorithms with the GIPS Framework. In *Proc. 14th. Int. Workshop Graph Computation Models (GCM)*. STAFF/GCM Session 1, 2023.

[132] Andrzej Wasowski and Thorsten Berger. Internal Domain-Specific Languages. In *Domain-Specific Languages: Effective Modeling, Automation, and Reuse*, pages 357–394. Springer, Cham, 2023. ISBN 978-3-031-23669-3.

[133] Idriss Chana Mohammed Lahmer and Abdallah Rhattoy Yassine Rhazali, Youssef Hadi. A model transformation in model driven architecture from business model to web model. 2018.

[134] Chenxing Zhong, Shanshan Li, Huang Huang, Xiaodong Liu, Zhikun Chen, Yi Zhang, and He Zhang. Domain-Driven Design for Microservices: An Evidence-Based Investigation. *IEEE Transactions on Software Engineering*, 50(6):1425–1449, June 2024. ISSN 1939-3520.

[135] Ozan Özkan, Önder Babur, and Mark van den Brand. Domain-Driven Design in software development: A systematic literature review on implementation, challenges, and effectiveness. *Journal of Systems and Software*, 230:112537, 2025. ISSN 0164-1212. doi: https://doi.org/10.1016/j.jss.2025.112537.