arXiv:2301.00624v1 [cs.SE] 2 Jan 2023

基于扩展广义判别网络的模块化与增量式全局模型管理

MatthiasBarkowsky Holger Giese
matthias.barkowsky@hpi.de holger.giese@hpi.de

2023年1月3日

摘要

如今，基于模型驱动工程范式开发的复杂项目通常涉及多个相互关联的模型，这些模型通过大量模型操作进行自动处理。为了支持对可能规模庞大的模型进行高效开发，需要模块化、增量式地构建和执行此类模型网络及模型操作。这一底层问题也称为全局模型管理。

在本报告中，我们提出一种方法，通过扩展现有的广义判别网络（GDN），实现模块化、增量式的全局模型管理。除了进一步推广 GDN 中所采用的查询操作概念外，我们还将此前仅适用于查询的机制调整为可用于具有副作用的操作，从而集成模型转换和模型同步。我们为生成的扩展广义判别网络（eGDN）提供了增量执行算法，并针对若干示例 eGDN 操作给出了原型实现。

基于这一原型实现，我们针对软件开发领域中的一个应用场景开展实验，从可扩展性的角度对我们的方法进行实证评估，并在概念层面展示其在典型场景中的适用性。初步结果证实，所提出的方法确实可用于在所考察的场景中实现高效的全局模型管理。

1 引言

如今，在模型驱动工程范式下开发的复杂项目通常涉及多个相互关联的模型，这些模型通过大量模型操作 [**69**][1] 进行检查、分析、转换和同步。

[1]请注意，粗体标注的参考文献指的是我们自己的出版物。

有效且高效地管理由此产生的复杂模型操作网络，既是开发项目成功的关键前提，也是一个具有挑战性的研究问题，即全局模型管理 [10]。

一方面，为适应不断演进的项目环境、动态的开发流程和变化的需求，需要对模型操作网络进行模块化和增量构建。另一方面，为适应当今可能规模庞大的模型，并支持团队协作开发，需要对这些网络进行模块化和增量执行，因为在发生变化时重新执行整个网络，可能导致无法接受的执行时间和信息丢失 [**39**]。

在此背景下，模型查询发挥着核心作用，因为模型属性和模型一致性检查分别显式或隐式地需要模型查询，模型转换和模型同步也同样如此。因此，相关解决方案必须提供专门支持，以处理可能复杂的模型查询，并便于对其进行模块化组合和复用。

此外，具有副作用的模型操作（如模型转换和模型同步）及其与其他模型操作的交互，也给整体目标带来了独特挑战：确保分布在多个模型中的系统描述保持一致。

在本报告中，我们提出一种全局模型管理方法，旨在同时实现所需的模块化和增量性。我们的解决方案基于对广义判别网络的一种扩展概念 [45], 。这一机制此前已在模型驱动工程领域实现 [**7**] ，用于以嵌套图条件的形式对模型查询进行模块化、增量式的规约和执行 [44]。

因此，我们引入了一种更通用的形式化方法，称为扩展广义判别网络（*eGDN*）。它（i）支持更灵活的模型查询概念，从而提升表达能力；（ii）允许将具有副作用的模型操作集成到统一框架中。此外，我们还提供了用于增量执行 eGDN 的算法。

此外，我们将若干典型的模型操作集成到该方法的原型实现中，并使用这一实现，基于软件开发领域的一个应用场景，对该技术的可扩展性进行初步评估。我们还通过概念评估，考察 eGDN 在典型场景中的适用性，作为这项实证评估的补充。

本报告其余部分的结构如下：第 2 章简要重述模型的基本概念，即类型图和判别网络。介绍所需概念后，我们将在第 3 章讨论全局模型管理解决方案的需求及相关工作，进一步说明设计新解决方案的动机。第 4、5 和 6 章将介绍我们提出的扩展广义判别网络。具体而言，第 4 章定义 eGDN，并介绍其图形表示法。第 5 章描述 eGDN 的增量执行。随后，第 6 章列举若干 eGDN 操作示例，这些操作均已纳入我们的原型实

现。我们使用该原型实现对所提出的概念进行初步实证评估，相关内容将在第7章中介绍，同时还将从概念层面评估 eGDN 在一个示例用例中的适用性。最后，第 8 章总结本报告，并概述未来工作的可能方向。

2 预备知识

本章将重述以类型化图和判别网络形式表示的模型基本概念。

2.1 图与模型

图 $G = (V^G,\ E^G,\ s^G,\ t^G)$ 由顶点集合 V^G、边集合 E^G 以及两个函数 s^G, t^G: $E^G \to V^G$ 组成，这两个函数分别将每条边映射到其源顶点和目标顶点 [24]。图 m: $G \to H$ 之间的图态射 G 和 H 是由一对函数 m^V: $V^G \to V^H$, m^E: $E^G \to E^H$ 构成，并满足 $s^H \circ m^E = m^V \circ s^G$和 $t^H \circ m^E = m^V \circ t^G$。

图 G 可以通过态射 $type^G$: $G \to TG$ 映射到类型图 TG 上，该态射将元素分配给 TG 中定义的类型 G 。由此得到一个类型化图 $G^T = (G,\ type^G)$。两个类型化图 $G^T = (G,\ type^G)$ 和 $H^T = (H,\ type^H)$ 之间的类型化图态射 m^T: $G^T \to H^T$，其中它们都映射到同一个类型图 TG，由满足 $type^G = type^H \circ m^T$的图态射 m: $G \to H$给出。

在本报告的语境下，模型可被定义为一个类型图，其中类型图实际上充当元模型。重要的是，模型元素的属性可以在类型化图的框架中实现：只需将属性值建模为专用节点，即可形成类型化属性图的概念 [47]。建模语言 ML 由一个图 TG 定义，表示所有在 TG上进行类型化的可能图的集合。

图1展示了软件开发领域中的一个示例模型，该模型以类型图 G的形式表示；同时还展示了其相关元模型，即类型图 TG。在节点的情况下，类型映射通过节点标签给出；在边的情况下，则是隐式给出的。示例模型表示一个使用面向对象编程语言编写的程序的抽象语法图（抽象语法图）。模型中的节点表示包、类型和方法。边表示不同概念之间的包含关系，包括方法包含于类型中、类型包含于包中，以及方法与类型之间的返回类型关系。

2.2 判别网络

判别网络是由表示计算单元的节点和表示这些单元之间依赖关系的边构成的图。判别网络是实现模型查询增量执行的一种常用方案，例如计算模型属性或检查模型一致性

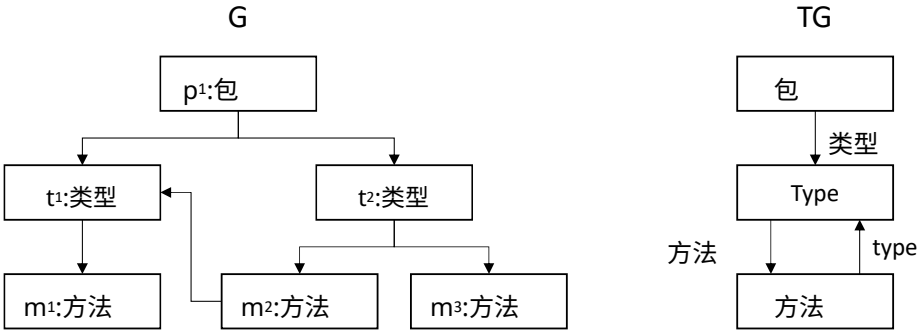


图1：软件开发领域中以类型图和类型图形式表示的示例模型和元模型

条件。因此，模型查询被分解为子查询，这些子查询构成判别网络的节点。

一个子查询的执行可以利用另一个子查询计算出的结果，这由两个子查询之间的依赖关系表示。最终判别网络节点的执行会产生整体查询结果。通过在查询执行结束后继续存储判别网络节点的结果，便可实现增量执行，在后续执行中重用先前计算出的结果。

迄今为止，判别网络主要用于模型查询，因此现有方法对集成带有副作用的模型操作仅提供有限支持，甚至完全不支持，因而至多只能算是全局模型管理的部分解决方案。然而，由于其固有地支持模块化和增量性，它们是一个很有前景的起点。

在模型驱动工程领域，判别网络这一概念有多种实现方式，下面的小节将简要介绍其中两种。

2.2.1 RETE 网络

RETE 网络最初由 Forgy [34] 提出，其特点是节点最多只能依赖另外两个节点。RETE 节点的一些示例如下：

- 输入节点，对应于提取模型中单个元素（即节点或边）的原始模型查询，因此没有任何依赖关系
- 过滤节点，根据条件筛选其他某个子查询的结果，因此只有一个依赖关系
- 连接节点，它们将另外两个子查询的结果合并为一个更复杂子查询的结果，因此具有两个依赖关系

虽然所列节点类型构成了增量模型查询解决方案的核心，例如成熟的 VIATRA [78], RETE 网络，但它们是一种灵活的机制，支持多种其他查询相关节点类型。VIATRA 支持用于指定模型查询的各种高级结构，便体现了这一点，其中包括否定模式和某些聚合操作。

在 RETE 实现中，RETE 网络节点计算出的结果通常存储在内存中所谓的索引器里，这些索引器充当计算节点之间的隐式接口。也可以通过将索引器建模为 RETE 网络的一部分，使其显式化：具体做法是使用另一类 RETE 节点，这类节点不具备任何计算功能，只用于存储其他节点的结果。

2.2.2 广义判别网络

广义判别网络（GDN）是限制比 RETE 网络更少的判别网络形式，由 Hanson 等人开发[45]。本质上，GDN 取消了 RETE 网络对节点依赖关系数量的限制，从而能更灵活地控制哪些中间查询结果需要存储在内存中。

文献 [7] 介绍了模型查询领域中的一种实现。该实现将 GDN 节点实现为模型转换规则，直接在被查询模型中为子查询结果创建标记元素。节点之间的依赖关系通过以下方式实现：在与依赖节点关联的转换规则中，考虑所需节点创建的标记元素。然而，尽管文献 [7] 中的方法基于一种表达能力较强的查询概念（即嵌套图条件），但底层形式体系不支持聚合等与查询相关的操作。

3 全局模型管理的需求

如今，基于模型开发复杂系统需要全局模型管理 （*GMM*） [9, 31] ，以确保不同子系统、不同视图和不同领域的模型能够恰当地组合在一起，即使这些模型可能处于不同的抽象层次。事实上，由于信息物理系统（CPS）等系统具有异质性和复杂性，将系统表示为单一底层模型（SUM）已不再可行。这是因为领域专家在协作构建系统时，已经独立使用了大量语言和工具。重新开发这些工具，进而要求业界改变其惯常做法，不仅需要投入大量开发工作，而且还会遭遇业界对改变开发流程的强烈抵制，因此并不可行。这对于必须经过复杂认证流程的安全关键系统尤为重要。因此，必须使用多个模型来表示系统，并需要恰当的 GMM，以确保针对这些模型开展的开发活动得到妥善协调，从而使这些模型共同构成一个恰当的系统

整体，并确保不同模型所涵盖的各类元素和方面得到正确整合，且彼此保持一致。

文献 [40] 介绍了模型集成问题和基本集成技术的分类。该分类突出了分解和丰富这两种技术，它们刻画了开发过程中的两个正交维度：系统被分解为子系统和定义域（水平维度），同时又被分解为一组详细程度逐步提高的模型（垂直维度）。这要求协调跨越这些维度、作用于模型的所有活动，以确保其一致性。

如今复杂系统的开发活动分布在多个领域和团队之间，每个团队都使用自己的一组建模语言，因此需要对这些语言进行适当的集成。事实上，已有研究表明，使用单一语言覆盖所有领域会导致非常庞大的单体式语言，难以针对开发组织所需的开发环境和工具进行定制。这些考虑促成了多范式建模（*MPM*）[77], ，其倡导集成可复用的模块化建模语言，而不是采用大型单体式语言。因此，GMM 不仅必须支持以适当的模块化方式集成模型，还必须支持集成它们的建模语言（下文称为建模语言集成），此外还要协调所有作用于模型、并被指定为模型操作/转换的活动。这些模型操作的执行必须具备可扩展性，才能处理大型模型。这要求具备增量性：模型发生变化时，仅重新执行受影响的操作，从而避免像增量代码编译器那样重新计算整个模型所需的开销。

GMM 也称为大规模建模，其内容是在宏观实体（模型和元模型）之间建立全局关系（例如，从其他模型生成某个模型的模型操作），同时忽略这些实体的内部细节 [9]。巨模型建模 [8, 31] 旨在描述这些宏观实体及其关系。

因此，我们需要能够以模块化和增量方式，全局管理模型，以便模块化、增量地构建和执行以下内容：I）模型与建模语言集成；II）模型操作；以及 III）巨模型。下文将概述，目前仅存在一些初步方法，它们针对上述问题的部分内容提供临时解决方案，而我们目前对其底层需求和挑战还缺乏扎实的认识。具体而言，现有方法至多只针对某一方面（例如建模语言集成或模型操作）提供一定程度的模块化和／或增量性。然而，迄今为止，还没有方法能够按照实际大型问题的要求，以模块化和增量的方式整体支持复杂建模景观的处理。

接下来，我们将更详细地讨论这些需求，并评述现有解决方案在以下方面的表现：1）模型与建模语言集成的构建与执行，2）模型操作，以及 3）巨模型。现有方法在这些维度上的表现如图所示

表1，其中空白单元格表示未满足的需求，˜表示部分满足该需求，而 +表示该需求已得到充分满足[2]。这一评估将在后续章节中进一步详细讨论。

方法	建模语言集成					模型操作				巨模型			
	Const.			*Exec.*		*Const.*		*Exec.*		*Const.*		*Exec.*	
	Links	*Int.*	*MMI*	*Batch*	*Inc.*	*Flow*	*Ctx.*	*Batch*	*Inc.*	*Mon.*	*Mod.*	*Batch*	*Inc.*
Modeling Languages Integration													
Blanc 等人 [11]				+	+								
EMF IncQuery [25, 76]				+	+								
Egyed 等人 [43, 23]				+	+								
Cabot 等人 [19]				+	+								
ACOL [57]		∼		+									
SmartEMF [58, 50, 73]	+	+											
复合 EMF Models [52, 21]	+	+											
EMF Views [27, 17]	+		∼	+									
Kompren [12, 55] / Kompose [33, 54]			∼										
Reuseware ModelSoc [51, 60]		∼	∼										
Ratiu 等人 [65]	+			∼	∼								
K¨onig 等人 [56]	+			+									
Model Operations													
Wires* [66, 80]						+		+					
ATL Flow [4]						+		+					
Epsilon [64, 28]	+					+		+					
Gaspard2 [29, 37]			∼			+	∼	+	∼				
Debreceni 等人 [22]						+	∼	+	+				
MoTCoF [71]						+	+	+	∼				
MoTE [38][61]	+							+	+				
Integration Languages and Others													
CyPhy [72, 42]	+					+		+					
FUSED [15, 35]	+	∼				+		+					
CONSYSTENT [48, 49]	+					+		+					
Megamodels													
AM3 [79, 1]	+					+		+		+		∼	
FTG+PM [59, 5]						+		+		+		+	
Megal Exp. [32]	+					+				+			
GMM* [13]	+		∼			+		+	∼	+		+	∼
Seibel 等 [70, 68][6]	+									+		+	∼
Stevens [75, 74]						+		+	+	+		+	+
Gleitze 等 [41]						+		+	+	+		+	+
Vitruvius [53]	+			+	+	+		+	+	+		+	+
eGDNs	+	+	+	+	+	+	+	+	+	+	+	+	+

表 1：现有及计划中的全局模型管理方法比较

3.1 模型与建模语言集成：构建与执行

3.1.1 构建

当前方法主要通过三种方式来实现模型与建模语言集成的构建：（1）链接模型和模型元素，（2）模型接口，以及（3）元模型组合。

（1）链接：

所有方法都会利用某种模型与其模型元素之间的追踪链接 来集成模型。在本报告中，我们采用软件可追溯性卓越中心（CoEST） [20]提出的可追溯性定义。跟踪链接 是“.....一对制品之间的一种指定关联，其中一个制品是源制品，另一个是目标制品.....”。同样遵循 CoEST 的定义，追踪链接可细分为垂直 维度和水平 维度上的追踪。因此，垂直 追踪“.....将制品链接到

[2]为方便起见，如果某种方法有对应的工具或项目，我们就用该工具或项目的名称来标识它；否则，使用描述该方法的出版物中第一作者的姓名。

不同的抽象层级，以实现覆盖整个生命周期或端到端的可追溯性，例如从需求到代码.....”。水平追踪链接“.....同一抽象层级上的制品，例如：*(i) Mary* 创建的所有需求之间的追踪，*(ii)* 与系统性能有关的需求之间的追踪，或 *(iii)* 特定需求在不同时点的不同版本之间的追踪”。

有大量利用追踪链接来集成模型的方法（例如，[2, 28, 32, 58, 48, 72, 15] [**61**]）。Atlas模型编织（AMW）语言 [2] 是最早用于捕获模型及模型元素之间分层可追溯性链接的方法之一。其目的是支持模型元素之间的自动导航等活动。该方法提供了一种通用核心可追溯性语言，并可选择扩展该语言，以提供针对待关联模型之元模型的特定语义。类似地，Epsilon框架 [28] 提供了一种用于建立模型间对应关系的工具（ModeLink）。MegaL Explorer [32] 支持使用预定义的关系类型关联异构软件开发制品，这些制品不一定是模型或模型元素。SmartEMF [58]则是另一种用于关联模型的工具，它基于Ecore元模型上的注解，通过对应规则指定模型元素之间由属性值构成的简单关系。复杂关系则通过本体来指定，这些本体关联了相关语言中的概念。所有组合模型都会转换为Prolog事实，以支持模型编辑时的导航、一致性检查和用户指导等活动。CONSYSTENT工具及其方法 [48] 也采用了类似思路。不过，它们使用图结构和模式匹配，以统一的形式化表示组合模型，并识别和管理其中的不一致，而非像SmartEMF那样使用Prolog事实。

此外，还有一些方法，例如 [72] 和 [15] ，通过使用为特定集成建模语言开发的集成语言，在模型之间建立链接。这类集成语言嵌入了与所链接语言相关的专用构造。扩展核心 AMW语言的模型编织语言也采用了这种方式。不过，AMW 的优势在于通过一种通用的核心语言来表示链接领域。链接和集成模型的其他方式还包括三重图语法（TGG），例如模型转换引擎（**MoTE**）工具 [**61**], ；与前述方式类似，这类工具也要求针对所集成的语言指定某种集成语言（对应元模型）。不过，这种方法的一项重要优势在于，它能够以可扩展、增量式 的方式自动建立和管理可追溯性链接，并维护关联模型之间的一致性（模型同步）。最后， [**70, 68**][**6**], 提出了一种可扩展的方法，用于自动创建和维护模型之间的可追溯性链接。该方法侧重于可追溯性管理，而非模型集成；与集成语言相比，它依赖于在模型层级（而非元模型／语言层级）定义的链接类型，因此无需在每次集成新语言时更新集成语言。更近

期， [65], 提出了反应式链接这一概念，它基本上支持在不同语言的模型之间增量传播属性值变化。不过，增量执行仅适用于一种有限的一致性概念。

对这些方法的比较表明，除了我们自己较早提出的方法 [**70, 68**][**6**], 之外，所有方法都存在依赖已集成语言集合的问题，因此需要更好地支持模块化。此外，只有我们自己的工作 [**61**][**70, 68**][**6**] 支持对可追溯性链接的自动化管理。

(2) 接口：除了链接之外，还有一些更复杂的方法（例如 [57, 50, 52, 51]）引入了模型接口（表 1 中的接口列）概念，用于指定模型之间如何建立链接。在 [57], 中，提出了分析约束优化语言（ACOL），该语言旨在可插接到架构描述语言（ADL）中。该方法纳入了接口这一 ACOL 特有的概念，使约束能够引用这些接口，从而关联到预期由 ADL 提供的模型元素。SmartEMF [50,73] 提出了更通用的模型接口概念，用于追踪模型与元模型之间的依赖关系，并提供自动兼容性检查。复合 EMF 模型 [52, 21] 引入了导出和导入接口，用于指定主模型（主体）中的哪些模型元素应公开给其他模型（即属于公共 API），以及主体模型中的哪些元素需要由导出接口提供。在 [51], 中，介绍了一种用于组合语法的方法，该方法具有显式变体点（钩子），构成了隐式侵入式组合接口。

然而，尽管这些方法提供了一些有趣的初步思路，但仍需进一步完善，以涵盖更多非侵入式模型集成用例，例如，指定链接模型元素的修改策略，以确保模型能够保持一致。它们也尚未集成到 GMM 中。

（3）元模型集成：一些方法（例如 [55, 54, 29, 27] [13]）会基于其他元模型或语言片段构建视图元模型（表 1 中的*MMI* 列）。 [29], 介绍了一种在 Gaspard2 工具 [37] 中实现的方法：为组合独立模型转换，人为扩展元模型，从而为扩展后的元模型生成扩展转换。 [12], 提出了一种语言和工具（Kompren） [55], 通过选择输入元模型中的类和属性来指定并生成元模型切片。随后会生成一个简化元模型；一旦输入元模型发生变化，就必须重新完整生成该简化元模型。Kompose 方法 [54], 也是如此；但与 Kompren 不同的是，它提出创建复合元模型，从每个组合元模型中选取一组可见模型元素，并可选择将它们关联起来。EMF Views [27, 17] 提供了类似的方法，但与 Kompose 和 Kompren 不同，它无需复制元模型元素。实际上，EMF Views 允许指定虚拟元模型，这些元模型仅引用现有的元模型元素，而不复制它们。对于虚拟元模型所对应的模型，也采用同样的原则：它们仅引用现有集成模型中的元素，而不复制这些元素。

这些虚拟元模型仅引用现有集成模型中的元素，而不复制这些元素。定义的虚拟视图元模型可供工具透明地使用。此外，由于该方法具有非侵入性，同一模型可以同时供遗留工具和使用虚拟元模型的新工具使用。最后，全局模型管理语言（**GMM***）[3] [13] 提供了指定和解释可复用语言子集的方法，这些子集以约束集合的形式组合起来，构成子集化元模型。与 EMF Views 类似，这些简化元模型在一定程度上也可以供工具透明地使用。面向方面的元模型组合是另一种广为人知的元模型组合技术。不过，它要求元模型采用特定的面向方面格式来表示，因此不符合我们的非侵入性要求。

尽管这些方法都为模块化建模语言集成提供了有趣的支持，但除已集成到我们的**GMM***语言中的子集化元模型外，它们尚未实现统一到共同形式体系、采用显式的模型接口概念以及集成到 GMM 中。我们注意到，在这些方法中，EMF Views 为这项工作提供了一个合适的起点，因为其非侵入性 特性对于重用遗留模型和工具至关重要。然而，在当前实现中，只有虚拟复合模型的属性变更会传播到底层真实模型 [17]。其他变更的传播以及元模型约束组合仍有待解决。未来还需为该方法集成显式的元模型接口构造，以规范元模型的组合方式，并借鉴面向对象编程中发展的 $Traits$ / $Mixins$ 概念，解决合并类的属性和操作冲突。

集成模型的执行不仅涉及对每个组合模型自身的良构性约束进行评估，也涉及对组合模型整体进行评估。据我们所知，尚无方法能够针对复合模型中不同语言片段的良构性条件进行增量检查。不过，目前已经存在一些关于增量约束求值的方法。在 [11]，中，模型变更被表示为一系列原子模型操作，以确定哪些约束受到变更影响，从而只需重新求值这些约束。在 [25, 76]，中，还提出了一种基于图的查询语言（EMF-IncQuery），该语言依赖增量模式匹配来提高性能。在 [23], 中，提出了一种基于查询所引用的模型元素范围进行约束增量求值的方法，该范围在首次查询求值期间确定。该范围被存储到缓存中，并用于根据某些模型变更确定哪些查询需要重新求值。在 [43], 中，该方法扩展到约束本身可能发生变更、同时受约束模型也可能发生变更的情形。最后，在 [19]，中提出了一种增量 OCL 检查器，它根据 OCL 约

[3]我们使用 * 来区分这一现有语言和工具与通用的全局模型管理（GMM）缩写。

束和给定的结构变更事件，计算出更简单的 OCL 表达式以及缩减后的上下文元素集。对缩减后的上下文求值这一更简单的约束，即足以断定初始约束的有效性，并且所需的计算资源显著减少。

在 [56], K¨onig 等人的研究中，他们提出了一种用于检查关联模型之间一致性约束的技术，无需将这些模型合并为单一底层模型，从而获得更好的可扩展性。然而，尽管该方法经过形式化定义并被证明正确，文献 [56] 中的方法并未考虑增量一致性检查。

我们确定了以下与建模语言集成的模块化和增量性相关的主要需求：

R1.1 通过集成链接集成建模语言，并结合良构性条件与一致性约束

R1.2 用于嵌入建模语言的接口

需要注意的是，表 1 中的需求涵盖了链接和接口，它们共同模拟了模块化程度较低的直接元模型集成；而所采用的良构性条件和一致性条件将在下一节讨论模型操作时加以介绍。因此，如表 1 所示，目前尚无任何方法能够同时满足我们所提出的所有这些需求。

3.2 模型操作：构建与执行

文献中主要以两种方式处理模型操作的构建。大多数方法将模型操作组合为模型转换链（（1）流组合），其中每个链式转换都以完整模型为粒度进行操作。为了支持由较简单的建模语言组合而成的复杂建模语言的复用性和可扩展性，少数方法考虑将模型转换指定为白盒。这些操作由针对特定上下文处理模型元素的显式细粒度操作组成，可在多种模型转换中复用（（2）上下文组合）。

（1）流组合方法：

FUSED（形式化统一系统工程开发） [15] 是一种集成语言，用于描述不同语言模型之间的复杂关系。它支持模型转换链，但仅通过工具执行隐式实现，没有显式表示所涉及的转换和处理的数据。与此相反，有大量方法可以显式指定和构建采用数据流范式的模型转换链。例如，AtlanMod 巨模型管理（AM3）工具 [1], 使用 Atlas Transformation Language（ATL） [3]来指定模型转换。此外，人们还开发了一种类型系统 [79], ，可对相关制品进行类型检查和类型推断

这些制品通过模型转换生成。另一个类似但不够成熟的工具是 Epsilon Framework [28], ，它通过 ANT 任务提供模型转换链式编排。Wires [66]和 ATL Flow [4] 是为 ATL 模型转换编排提供图形化语言的工具。在 ATo MPM（多范式建模工具） [5] 中实现的形式化转换图 + 过程模型（FTG+ PM）形式体系 [59]提供了类似功能。不过，它的优势在于，除了所涉及的模型转换之外，还能指定完整的建模过程。它通过将活动图与模型转换规范相结合来实现这一点，并自动执行这些转换以支持开发过程。最后，**GMM***[13]也支持模型转换链式编排，但它通过指定特定元模型的模型之间可串联的关系来实现。该方法的一个优势是，在收到模型变更事件后，能够自动增量式地（重新）执行所指定的模型间关系。将 **MoTE** [**61**] 增量模型转换工具集成到 **GMM***中，也使转换的增量执行成为可能。

然而，尽管串联模型转换在一定程度上提高了模型转换规范的模块化程度，但除 **GMM*** 之外，大多数方法在处理大型模型时都存在可扩展性问题，因为所用的转换工具不支持增量执行。此外，如果手动修改生成模型，以添加无法用原始模型语言表达的信息，这些方法也难以处理，因为重新生成经过手动修改的模型会破坏用户特定的信息。上下文组合方法对这一需求的支持更好。

（2）上下文组合方法 es:

少数方法支持模型操作的上下文组合（表 1 中的列 *Ctx.*）。如上所述， [29] 介绍了一种组合独立模型转换的方法，从而为相应的扩展元模型生成扩展转换。 [22], 介绍了一种方法，通过将派生规则编码为 EMF Inc-Query [25] 语言中查询的注解，利用模型操作的上下文组合来指定视图模型的构建过程。视图模型元素与源模型元素之间的可追溯性链接会自动建立并维护。由于原生采用了 EMF IncQuery，派生规则可以增量执行，从而使视图模型与源模型保持同步。有些视图可以从其他视图派生，因此能够将视图模型串联起来实现流组合。这种方法取得了与支持增量性的 TGG 类似的结果，但存在单向性的缺点。类似地，**MoTCoF** 语言 [**71**] 支持模型转换的流组合和细粒度上下文组合，并具备双向性。不过，与 [29] 相比，其优势在于将模型转换作为黑盒使用，无需根据上下文调整转换。

可以看出，大多数方法仅支持模型

操作的流类型模块化，并且采用批量执行；只有我们的**GMM*** 语言例外，因为它集成了**MoTE** ，从而支持增量执行。当部分模型信息重叠时，这种方式无法扩展，并会导致信息损失。只有少数方法支持上下文模块化，这更有利于增量应用：发生变更后，只需重新应用受影响的操作，从而避免重新计算完整转换的开销。**MoTCoF**便是其中之一，它在理论上支持增量执-行，但目前仍缺乏具体的技术方案。

为解决模型操作的模块化和增量性问题，我们确定了以下主要需求：

R2.1 模型操作的组合

R2.2 针对集成模型的模型操作

R2.3 模型操作的执行方案

请注意，表 1 中的需求涵盖了基于流程和基于上下文的组合，以及批处理和增量执行；首先针对模型操作的所有特殊情况，然后也针对一般情况。因此，如表 1 所示，目前还没有任何方法能够全面满足我们所期望的所有这些需求的组合。

3.3 巨模型及其他全局模型管理方法

GMM 可分为两种思路。第一种采用（*1*）模型集成语言，这类语言是为特定的一组集成建模语言和工具定义的，这意味着每当使用新的语言或工具时，都必须更新集成语言。第二种思路则试图通过使用（*2*）巨模型来解决这一问题，从而提供可配置的全局模型管理。

（1）集成语言及其他方法：

CyPhy [72] （用于 GME 建模工具 [42] ）和 FUSED [15, 35] 都是模型集成语言的例子。但如上所述，一旦需要使用不同的一组集成语言和工具，就必须调整这些语言，因此需要开发者具备高超的技能。因此，集成语言并不实用。

生命周期协作开放服务 (OSLC) [63] 通过 Web 提供工具集成标准。针对变更管理、资源预览、关联数据等，已有许多规范。它基于 W3C关联数据 标准，该标准旨在基于 W3C 资源描述框架 (RDF)，为在 Web 上发布结构化数据提供最佳实践。RDF 是一种用于 Web 上数据交换的模型，其中数据以图的形式表示。然而，OSLC 更侧重于服务（和工具），并继承了

关联数据的缺陷：它专用于 Web，因此无法将数据表示与持久化分开；而在模型驱动工程 (MDE) 中，抽象语法的使用独立于数据存储方式。

另一种利用这些标准的方法是 [48]，并通过一个名为 CONSYSTENT 的工具实现，该工具用于识别并解决因信息重叠而在不同视点之间产生的不一致。开发过程中涉及的所有模型信息都被记录在一个共同的 RDF 图中。该方法依赖人工[4] 指定用于表示图模型之间语义等价链接（语义连接）的模式。系统会持续在 RDF 模型中检查基于这些语义连接的不一致模式，以寻找可能表明存在不一致的匹配项。自动解决不一致的方法仍在开发中。然而，这种方法需要将所有模型转换为 RDF 图，因此不具备增量性，也无法扩展到大型模型。

（2）巨模型：

在这第二条研究脉络中，巨模型用于捕获和管理模型驱动工程资源，例如建模语言、模型转换、模型对应关系，以及建模环境中使用的工具。如前所述，已有若干大型模型建模方法。AM3 [1] 是最早的方法之一，其中巨模型基本上是模型驱动工程资源的注册表。模型转换使用 ATL [3] 指定，模型对应关系则使用 Atlas 模型编织（AMW）语言 [2]指定。同样，如上所述，FTG+PM [59] 也是一种巨模型建模语言，MegaL Explorer [32] 也不例外；它们允许从语言学角度对软件开发环境中使用的制品及其关系进行建模。相关软件语言、技术及技术空间可通过它们之间的语言学关系来捕获，例如成员关系、子集关系、符合关系、输入关系、依赖关系、定义关系等。实体之间的操作也可以被捕获。这些制品不必表示为模型，但巨模型中的每个实体都可以链接到可供浏览和检查的 Web 资源。不过，这种语言似乎主要用于可视化，有助于更好地理解开发制品，但无法执行模型管理。前述 **GMM*** 基础设施 [13]包含一种受 [**46**]启发的巨模型建模语言。用户可以声明元模型，以及这些元模型的模型之间的关系。特别是，同步关系可以关联两个不同元模型的模型，并借助**MoTE** TGG 引擎 [**61**] 转换或同步这些模型。如前所述，模型转换链可以被指定，并在模型变更事件发生时增量执行；还可以声明建模语言的子集。**GMM*** 已在 Kaolin 工具 [14]中进行实验，并使用了 AADL 和 VHDL 等复杂且丰富的工业语言

[4]一种利用贝叶斯信念网络的自动化方法也正在研究中 [49]。

从而对 GMM 应对现实规范的能力提出了挑战。

Stevens 提出了一种利用模型双向转换进行大规模建模的新方法 [75]。 [75] 以超图的形式提出了巨模型的形式化概念，其中模型表示为节点，并可通过表示双向转换的超边相连。该形式体系通常支持增量执行；不过，对于具有特定受限结构、能够保证某种正确性的巨模型，文中才给出了具体算法。作者在 [74] 中拓展了这项工作，将先前的研究与构建系统领域的研究相结合，并引入了一种所谓的定向模型，用以引导巨模型执行，从而放宽对巨模型结构的限制，同时保持正确性的形式保证。然而，对于复杂的模型操作网络，构建定向模型是一个手动且可能颇具挑战性的过程。此外， [75, 74] 的研究将模型操作的技术实现抽象掉，因此没有明确考虑如何以模块化方式组合模型属性计算等操作。

在 [41], Gleitze 等提出了一种针对模型转换网络的增量执行策略，旨在提供一种解决方案，用于解释该策略未能产生一致结果的情况。尽管其策略适用于任意结构的网络，但仅考虑模型对之间的双向转换，因此限制了所支持的模型操作范围。

近年来，模型视图领域也取得了显著进展[16], 。该领域研究如何从由多个相互关联的模型构成的系统描述中推导出一致的视图模型，因此也与全局模型管理相关。最全面、最先进的模型视图技术可能是 Vitruvius 方法 [53], ，它依赖所谓的虚拟单一底层模型（V-SUM）来描述正在开发的整体系统。V-SUM 用于整合描述系统各部分的单个模型，并通过一致性关系推导出新的视图模型。因此，Vitruvius 采用专用的增量算法来执行复杂的一致性保持操作网络。然而， [53] 中的一致性概念仅限于模型元素元组对之间的关系，因此不支持某些模型操作，例如使用聚合计算模型属性。此外，该方法有意不涵盖模型内良构性，也未考虑在元模型层级的复用。

然而，这些巨模型建模方法大多只在一定程度上涵盖了通过元模型和模型操作来描述模型驱动工程资源所需的核心要素，以及适当的模块化和增量性。它们只解决了问题的部分内容。此外，所有这些巨模型建模语言都是单体式的（表 1 中的列单体式 ），因此无法组合和重用预定义的巨模型片段，避免为新项目从头开始重新构建完整的巨模型规范。不过，我们注意到，面向方面的元模型组合可以作为一个启发性的切入点

并将其改造以应用于巨模型建模，从而描述分布式巨模型片段，为集成巨模型提供横切信息。至于巨模型执行，FTG+PM、 [75, 41, 53], GMM*和 [70, 68] 考虑根据模型变更或来自工具用户界面的建模事件进行自动或半自动执行。

相关工作表明，对于全局模型管理，我们需要一种能够在巨模型中整合其所有方面的视图。为解决巨模型的模块化和增量性问题，我们可以得出以下主要需求：

R3.1 一种巨模型建模语言，具备

R3.1.1 对元模型、良构性、模型操作、集成视图和可追溯性链接的支持

R3.1.2 一种巨模型操作模块概念

R3.2 一种稳健的增量式巨模型执行方案

R3.3 巨模型接口

R3.4 一种异步增量式巨模型执行方案

请注意，表 1 中的需求涵盖了模块化构建和增量执行。如表 1 所示，确实有三种方法不支持模块化，但满足了我们目标中的其他所有需求。不过，这些方法都无法提供所需的稳健增量式巨模型操作执行方案。 [75, 74], 中的技术虽然在正确性和终止性方面提供了形式化保证，但仅适用于模型操作网络：其结构为模型对之间同步构成的树，或者需要手动构建定向模型。Vitruvius 方法 [53], 采用不动点迭代，因此不限制网络结构，但也因此无法保证终止。 [41] 中提出的执行方案适用于结构任意的模型对之间的模型同步网络，并且能够保证终止。然而，除在具体实例上执行实际操作外，该方案无法判定某个网络最终是否会以正确结果终止。

3.4 最新研究进展总结

对最新研究进展的这项综述表明，有若干方法能够满足本报告提出的模块化和增量性需求。然而，这些方法都无法同时满足我们认定必不可少的三个层面的需求：模型操作、建模语言集成和巨模型。此外，对于全局模型管理的某些具体方面，甚至还没有单独提出具备足够模块化和增量性的解决方案。本研究专门针对这些尚未得到充分解决的关键需求。

4 扩展广义判别网络

本章将介绍扩展广义判别网络（eGDN）的概念，并说明如何将这一新形式体系用作巨模型语言。

4.1 eGDN 的定义

为了解决现有方案的不足，并支持模型属性、模型一致性操作、模型转换和模型同步等复杂模型操作网络的模块化增量构建与执行，我们进一步将广义判别网络的思想推广为扩展广义判别网络。因此，我们引入 GDN 节点及其接口的广义概念。这使得可以集成具有副作用的模型操作，并且相较于 [7], ，能够更灵活地定义查询，从而提高表达能力。

eGDN $G = （O, S, E, s, t）$本质上是一个二部图，包含两种节点：槽节点和操作节点，其中 O 是操作节点集合，S 是槽节点集合。操作节点可以通过边集 E中的边与槽节点相连，反之亦然。边的源函数和目标函数由 s 给出，分别为 $E \to O \cup S$ 和 $t: E \to O \cup S$。

操作节点表示模型操作或其构成部分，即子操作。槽节点在 eGDN 中存储模型操作及其子操作所使用的信息。边表示操作节点与槽节点之间的依赖关系，其中边的源表示所需节点，边的目标表示依赖节点。操作节点依赖于槽节点，表示相应的模型操作使用存储在该槽中的信息。槽节点依赖于操作节点，表示该操作节点对应的模型操作会修改该槽中的内容。

我们将槽节点或操作节点 n in $O \cup S$ by$in(n) = \{d \in O \cup S | \exists e \in E: s(e) = d \wedge t(e) = n\}$ 的依赖节点集合记为。类似地，我们将 n by $out(n) = \{d \in O \cup S | \exists e \in E: s(e) = n \wedge t(e) = d\}$ 的被依赖节点集合记为。G 是二分的，因为 $\forall o \in O: in(o) \subseteq S \wedge out(o) \subseteq S$ 且 $\forall s \in S : in(s) \subseteq O \wedge out(s) \subseteq O$。对于操作节点 $o \in O$，我们还将槽节点集合 $in(o)$称为 o 的输入槽，并将槽节点集合 $out(o)$ 称为 o 的输出槽。

槽节点 s 始终与建模语言 ML 或有序变量集 $var = \{v_1, v_2, ..., v_k\}$ 相关联，并分别包含 ML 的模型（类型图），或 var的变量赋值集。有序变量集 $var = \{v_1, v_2, ..., v_k\}$ 的变量赋值由一个元组给出，该元组属于 $dom_V(v_1) \times ... \times dom_V(v_k)$，其中 $dom(v_i)$ 表示变量 v_i的定义域，该定义域可以是来自一个或多个模型的节点或边的集合，

也可以是原语的集合，例如 $\mathbb{N}$。我们将 s 可能包含的赋值集或模型的集合称为槽的域：当 s 与建模语言 ML 相关联时，该域由 $dom(s) = ML$ 给出；当 s 与有序变量集 $var = \{v_1, v_2, ..., v_k\}$相关联时，则由 $dom(s) = \mathcal{P}(dom_V(v_1) \times ... \times dom_V(v_k))$ 给出。随后，通过赋值函数 val：$S \to \bigcup_{s \in S} dom(s)$ 将内容赋予 eGDN 的槽，使得$\forall s \in S: val(s) \in dom(s)$。

除了常规模型外，我们还允许模型槽包含链接模型。常规模型与链接模型之间的唯一差异在于，链接模型的顶点集合可以将其他常规模型和链接模型中的顶点引用为边目标，从而建立模型间连接。因此，与链接模型类似，链接模型的元模型，即链接模型的类型图，也可以将其他类型图中的顶点引用为边目标。

对于操作节点，我们进一步区分查询节点、转换节点和混合节点。

查询节点从模型和/或其他查询的结果中提取信息。因此，查询节点 q 可以具有任意数量的输入槽和恰好一个输出槽。 q的输入槽可以包含模型或变量赋值集合，而 q'的输出槽只能包含变量赋值集合。

转换节点基于模型和/或查询结果创建或修改模型。因此，转换节点 t 可以具有任意数量的输入槽和输出槽。 t的输入槽可以包含模型或变量赋值集合，而 t的输出槽只能包含模型。

混合节点 x 由查询节点和转换节点组合而成，可以具有任意数量的输入槽和输出槽，其中可包含模型，也可包含变量赋值集。

每个具有输入槽 $in(o) = \{s_{i_1}, ..., s_{i_k}\}$ 的操作节点 o 都关联一个语义函数 $\gamma_S: dom(s_{i_1}) \times ... \times dom(s_{i_k}) \to \mathcal{P}(\mathbb{F})$，其中 $\mathbb{F}$ 表示函数集合 $f: out(o) \to \bigcup_{s_o \in out(o)} dom(s_o)$，满足 $\forall s_o \in out(o): f(s_o) \in dom(s_o)$。本质上，操作节点的语义函数描述了该操作的输入槽与输出槽之间的一致性关系。

为了表示与 o 相邻的槽位内容与 o 针对赋值函数 val 的语义函数一致，我们写作 $o.valid(val)$。形式上，$o.valid(val) \leftrightarrow \exists f \in \gamma_S(val(s_{i_1}), ..., val(s_{i_k})): \forall s_o \in out(o): f(s_o) = val(s_o)$。

eGDN $G = (O, S, E, s, t)$ 的赋值函数 val 整体上与G 一致，当且仅当满足 $\forall o \in O: o.valid(val)$。

4.2 作为巨模型的 eGDN

由于 eGDN 编码了一个连接一组可能集成模型的模型操作网络，因此它代表一个**巨模型**。因此，eGDN 的定义构成了一种巨模型语言。

重要的是，eGDN 允许组合由实现子操作的节点构成的模型操作。此外，它们还支持层次组合：eGDN（因此，基本 GDN 或 RETE 网络也一样）可以被解释为一个 eGDN 操作节点。输入槽和输出槽分别由其节点中与父 eGDN 的另一个操作节点相连的输入槽和输出槽决定。子 eGDN 中任何未与父 eGDN 的另一个节点相连的槽位都可以作为其内部槽，而无需向父 eGDN 暴露。不过，如果某些此类潜在内部槽的内容与人类用户相关，也可以将它们视为输入槽或输出槽。代表子 eGDN 的操作节点的语义函数，随后由该 eGDN 自身操作节点的语义函数隐式定义。

除了（层次化的）可组合性之外，eGDN 还支持模块化：就输出槽位而言，操作节点的语义仅直接取决于其直接输入槽位的内容。因此，将其他操作节点（以及其他槽节点）集成到 eGDN 中，只需通过连线将其与该节点的输入槽和输出槽适当连接即可，完全不受其他操作节点的影响。由此，槽位实际上充当了模型操作之间的接口。

eGDN 还可以通过将查询节点的结果（必要时结合转换节点，将查询结果中的更改传播回基础模型）用作模型接口或视图，在模型层级实现模块化。例如，可以结合访问限制使用简单的投影查询，为开发过程中的不同角色实现不同的可见范围。或者，也可以结合专用视图模型与双向模型同步操作，在 eGDN 中实现可编辑模型视图。

图 2 展示了我们用于可视化 eGDN 的图形表示法。槽节点以矩形表示；对于赋值槽，右上角标注“A”，对于模型槽，则标注“M”。为使图示更加清晰，包含链接模型的模型槽通过虚线箭头连接到包含关联模型的模型槽。操作节点以圆角矩形表示；查询节点（如模型属性或模型一致性检查）右上角标注“Q”，转换节点（如模型转换和模型同步）右上角标注“T”，混合节点（如子 eGDN）右上角标注“X”。此外，所有节点均按照 ¡名称¿:¡类型¿ 的模式标注。

图 3 展示了一个 eGDN 示例，它由三个槽节点和两个操作节点组成，实现了一个简单的模型操作链。存储在最左侧槽位中的类图通过转换节点转换为抽象语法图。随后，查询节点从生成的抽象语法图中提取信息，并通过赋值槽使查询结果可供访问。

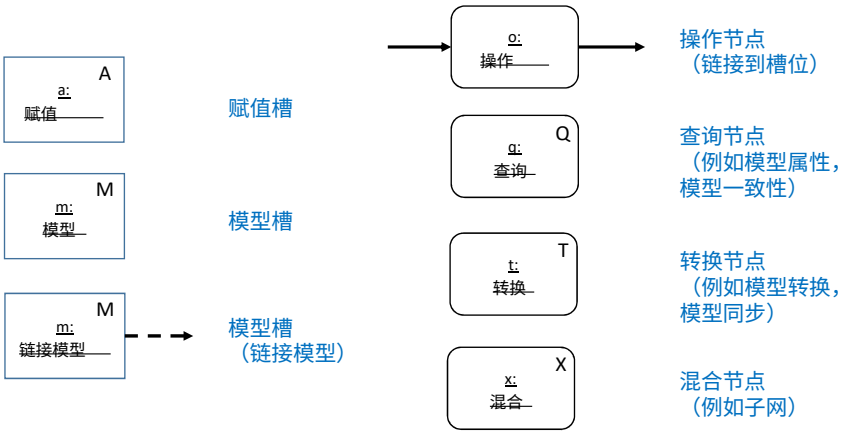


图 2：eGDN 的图形表示法

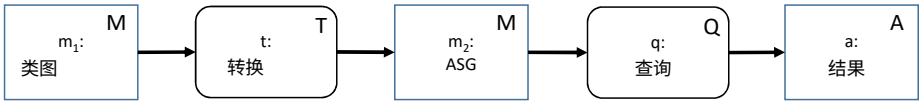


图 3：简单的 eGDN 示例

5 扩展广义判别网络的增量执行

本章将介绍如何执行 eGDN，以响应外部变化，恢复模型网络及模型操作的一致性。

5.1 关于增量执行的定义

用户执行编辑操作后，eGDN 模型槽中的模型 M 可能会发生变化。在此语境下，变化是指创建或删除顶点或边，并以四种类型之一的原子模型差分来表征：

- δ_+^V 是一个单元素元组（v），其中 v 是一个顶点；将 δ_+^V 应用于 M 会把 M 修改为 $M' = (V^M \cup \{v\},\ E^M,\ s^M,\ t^M)$
- δ_-^V 是一个单元素元组（v），其中 $v \in V^M$；将 δ_-^V 应用于 M 会把 M 修改为 $M' = (V^M \setminus \{v\},\ E^M,\ s^M,\ t^M)$
- δ_+^E 是一个元组（$e,\ s,\ t$），其中 e 是一条边，$s,\ t \in V^M$；将 δ_+^E 应用于 M 会把 M 修改为 $M' = (V^M,\ E^M \cup \{e\},\ s^M \cup \{(e,\ s)\},\ t^M \cup \{(e,\ t)\})$
- δ_-^E 是一个单元素元组（e），包含 $e \in E^M$；将 δ_-^E 应用于 M 会将 M 修改为 $M' = (V^M,\ E^M \setminus \{e\},\ s^M \setminus \{(e,\ s^M(e))\},\ t^M \setminus \{(e,\ t^M(e))\})$

重要的是，这种原子增量的概念也可以涵盖类型化属性图模型中属性值发生变化的情况 [47]。在此语境下，可以通过专门的顶点来表示属性值，并通过边来表示这些值赋给常规顶点属性的关系，从而对属性进行建模。

请注意，我们不允许隐式删除边。如果删除一个顶点，则它不得有任何相邻边，也就是说，必须先删除所有相邻边。同样，如果创建一条边，则相邻顶点必须已存在于模型中。

对槽位 s 中的赋值集 A 所做的更改，同样可以用原子槽位增量来描述:

- δ_+^A 是一个单元素元组（a），其中 $a \in dom$（s）；将 δ_+^A 应用于赋值集 A，会将 A 修改为 $A' = A \cup \{a\}$
- δ_-^A 是一个单元素元组（a），其中 $a \in dom$（s）；将 δ_-^A 应用于赋值集 A，会将 A 修改为 $A' = A \setminus \{a\}$

对于槽节点 s，我们用 dom_Δ（s）表示 dom（s）上所有可能的原子增量的集合，用$\mathbb{S}$（dom_Δ（s））表示 dom_Δ（s）中元素的所有可能序列的集合。

原子增量可通过某个 $apply$过程应用于模型或赋值集。我们将此过程重载，使其也适用于原子增量序列；此时，该过程会按照序列指定的顺序逐一应用各个增量。

我们称原子增量序列 Δ对于槽节点 s的内容是最小的，当且仅当对于所有可能的内容 $v \in dom(s)$，都有 $\nexists \Delta' \in \mathbb{S}(dom_\Delta(s))$：$apply(v, \Delta') = apply(v, \Delta)$；其中，我们只考虑在同构意义下相等的图。

为了能够通过 eGDN $G = (O, S, E, s, t)$ 对模型变更作出响应，具有输入槽 $in(o) = \{s_{i_1}, ..., s_{i_k}\}$和输出槽$out(o) = \{s_{o_1}, ..., s_{o_l}\}$的操作节点 $o \in O$可以配备一个 $update$过程。此过程以针对 G 的赋值函数为参数，并实现函数 γ^δ：$dom(s_{i_1}) \times ... \times dom(s_{i_k}) \times \mathbb{S}(dom_\Delta(s_{i_1})) \times ... \times \mathbb{S}(dom_\Delta(s_{i_k})) \times dom(s_{o_1}) \times ... \times dom(s_{o_l}) \to \mathbb{F}_\Delta$，其中 $\mathbb{F}_\Delta$ 是函数集合，函数形式为 f_Δ：$out(o) \to \bigcup_{s_{o_i} \in out(o)} \mathbb{S}(dom_\Delta(s_{o_i}))$，且满足 $\forall s_{o_i} \in out(o)$：$f_\Delta(s_{o_i}) \in \mathbb{S}(dom_\Delta(s_{o_i}))$。

为了存储增量以便稍后做出响应，o 也扩展了一个数组 $o.\Delta$，用于缓存其输入槽位和输出槽位的原子增量序列；实际上，这些序列可以通过通知机制和观察者设计模式 [36]收集。

使用 val 作为 G各槽位的赋值函数调用 $o.update(val)$，即可得到按 val 参数化的 γ^δ 的值，以及缓存的增量序列：$o.update(val) = \gamma^\delta(val(s_{i_1}) ..., val(s_{i_k}) \; o.\Delta[s_{i_1}], ..., o.\Delta[s_{i_k}], val(s_{o_1}) ..., val(s_{o_l}))$.

直观地说，操作节点的 $update$ 过程应生成一系列针对该节点输出槽位的增量，以更新这些输出槽位的内容，使其与操作节点当前内容保持一致

输入槽。因此，除了与 o 相邻的槽位内容外，$update$ 过程还可以考虑以输入槽增量形式提供的其他信息，从而实现更高效的实现。

形式化地，对于具有输入槽位$in(o) = \{s_{i_1}, ..., s_{i_k}\}$、输出槽位 $out(o) = \{s_{o_1}, ..., s_{o_l}\}$和关联函数$\gamma^\delta$的操作节点 o ，更新过程 $update$ 当且仅当对于参数 $\Delta_1 \in \mathbb{S}(dom_\Delta(s_{i_1}))$ ...，$\Delta_k \in \mathbb{S}(dom_\Delta(s_{i_k}))$、$v_{i_1} \in dom(s_{i_1})$ ...，$v_{i_k} \in dom(s_{i_k})$以及 $v_{o_1} \in dom(s_{o_1})$ ...，$v_l \in dom(s_{o_l})$，

$$\exists f \in \gamma_S(v_{i_1}, ..., v_{i_k}) : \forall s_{o_i} \in out(o) : apply(v_{o_i}, f_\Delta(s_o)) = f(s_{o_i}),$$

具有 $f_\Delta = \gamma^\delta$（Δ_1、...、Δ_k、v_{i_1}、...、v_{i_k}、v_{o_1}、...、v_{o_l}）且 $\forall i \in [1, k]$、$j \in [1, l]$：$s_{i_i} = s_{o_j} \to v_{i_i} = v_{o_{j_\circ}}$

在许多情况下，高效实现一个 $update$ 过程需要采用一种宽松的正确性概念：根据 o的语义函数应用增量之前，输出槽位的内容必须与输入槽位的内容保持一致。下文将这种宽松的正确性概念称为条件正确性，其形式化定义如下：

$$\begin{aligned}
&\exists v'_{i_1} \in dom(s_{i_1}), ..., v'_{i_k} \in dom(s_{i_k}) : \\
&\quad (apply(v'_{i_1}, \Delta_1) = v_{i_1} \wedge ... \wedge apply(v'_{i_k}, \Delta_k) = v_{i_k} \wedge \\
&\quad \exists f' \in \gamma_S(v'_{i_1}, ..., v'_{i_k}) : \\
&\qquad (\forall s_{i_i} \in in(o) \cap out(o) : v'_{i_i} = f'(s_{i_i}) \wedge \\
&\qquad \forall s_{o_i} \in out(o) \setminus in(o) : v_{o_i} = f'(s_{o_i}))) \\
&\to \\
&\exists f \in \gamma_S(v_{i_1}, ..., v_{i_k}) : \forall s_{o_i} \in out(o) : apply(v_{o_i}, f_\Delta(s_o)) = f(s_{o_i}),
\end{aligned}$$

与 $f_\Delta = \gamma^\delta(\Delta_1, ..., \Delta_k, v_{i_1}, ..., v_{i_k}, v_{o_1}, ..., v_{o_l})$ 及 $\forall i \in [1, k], j \in [1, l]$：$s_{i_i} = s_{o_j} \to v_{i_i} = v_{o_{j_\circ}}$

如果对于赋值函数 val 和缓存的增量 $\Delta_1, ..., \Delta_k$（其中$\sum_{i \in \{1,...,k\}} |\Delta_i| = 1$，即一个作为输入的原子差分），操作节点 o 的 $update$ 过程的实现满足以下条件，我们就称其为完全增量式：(i) 运行时间复杂度属于 $O(|\Delta_o|)$，其中 $\Delta_o = \bigcup_{s_o \in out(o)} o.update(val)(s_o)$；(ii) 每个输出槽产生的增量集合均为最小。

在某些情况下，例如双向模型转换或原位模型转换中，操作节点可能通过一条入边和一条出边与同一个槽位相连，使该槽位同时成为同一操作节点的输入槽和输出槽。因此，这种操作节点可能表现出递归行为，因为执行其 $update$ 过程也可能改变操作节点输入槽中的内容，从而需要进一步调用 $update$以恢复一致性。在此情境下，如果操作节点的 o $update$ 过程在执行一次 o的更新函数并

随后将产生的增量应用于 o的输出槽值之后，再以更新后的槽位值执行第二次时不会产生任何新增量，那么我们就称该过程为非递归。

形式上，如果对于任意可能的参数化 $v_{i_1} \in dom(s_{i_1}) ... v_{i_k} \in dom(s_{i_k})$、$\Delta_1 \in \mathbb{S}(dom_\Delta(s_{i_1})) ... \Delta_k \in \mathbb{S}(dom_\Delta(s_{i_k}))$ 以及 $v_{o_1} \in dom(s_{o_1}) ... v_{o_l} \in dom(s_{o_l})$，均满足以下条件，那么操作节点 o （其输入槽为 $in(o)$、输出槽为 $out(o)$）的 $update$ 过程是非递归的：

$$\forall s_o \in out(o) : \gamma^\delta(\Delta'_1, ..., \Delta'_k, v'_{i_1}, ..., v'_{i_k}, v'_{o_1}, ..., v'_{o_l})(s_o) = \emptyset,$$

其中

$$\Delta'_i = \begin{cases} f_\Delta(s_i) & \text{if } s_{i_i} \in out(o) \\ \Delta_i & \text{otherwise} \end{cases}$$

and

$$v'_{i_i} = \begin{cases} apply(v_{i_i}, f_\Delta(s_i)) & \text{if } s_{i_i} \in out(o) \\ v_{i_i} & \text{otherwise} \end{cases}$$

and

$$v'_{o_i} = apply(v_{o_i}, f_\Delta(s_{o_1})),$$

具有 $f_\Delta = \gamma^\delta$（Δ_1、...、Δ_k、v_{i_1}、...、v_{i_k}、v_{o_1}、...、v_{o_l}）。

对于输入槽集合 $S_i \subseteq in(o)$，操作节点 o 的更新过程的潜在更新方向由 $o.dir_\Delta(o, S_i)$ 给出，其中，对于槽位 $s_o \in out(o)$，

$$\begin{aligned} s_o \in o.dir_\Delta(o, S_i) \leftrightarrow \exists \Delta_1 \in \mathbb{S}(dom_\Delta(s_{i_1})), ..., \Delta_k \in \mathbb{S}(dom_\Delta(s_{i_k})), \\ v_{i_1} \in dom(s_{i_1}), ..., v_{i_k} \in dom(s_{i_k}), \\ v_{o_1} \in dom(s_{o_1}), ..., v_{o_l} \in dom(s_{o_l}) : \\ \forall s_{i_i} \in in(o) \setminus S_i : \Delta_i = \emptyset \wedge \\ \gamma^\delta(\Delta_1, ..., \Delta_k, v_{i_1}, ..., v_{i_k}, v_{o_1}, ..., v_{o_l})(s_o) \neq \emptyset \end{aligned}$$

直观而言，$o.dir_\Delta(o, S_i)$ 表示这样一个输出槽子集：如果 S_i 中至多输入槽的内容发生了变化，那么 o 的更新过程可能会为这些输出槽生成增量。

潜在更新方向的函数 dir_Δ 按定义是单调的，即 $\forall S_{i_1}, S_{i_2} \subseteq in(o)$：$S_{i_1} \subseteq S_{i_2} \rightarrow o.dir_\Delta(o, S_{i_1}) \subseteq o.dir_\Delta(o, S_{i_2})$。如果还满足 $\forall S_{i_1}, S_{i_2} \subseteq in(o)$：$o.dir_\Delta(S_{i_1}) \cup o.dir_\Delta(S_{i_2}) = o.dir_\Delta(S_{i_1} \cup S_{i_2})$，我们就称 dir_Δ 为并单调。

下面我们介绍基于操作节点的 $update$ 过程来增量执行 eGDN 的算法。对于这些算法，我们假设输入 eGDN 中缓存的增量是一致的，也就是说，它们对应于从与 eGDN 中所有操作的语义函数相一致的槽内容

到当前内容的修改。直观地说，这一假设意味着，只有当槽内容此前曾在某个时刻保持一致，且此后所有更改均已被跟踪并缓存于 eGDN 中时，所介绍的算法才能生成一致的槽内容。

5.2 保证终止的增量执行

对于每个操作节点，给定一个正确的 $update$ 函数，输入 eGDN $G =(O,\ S,\ E,\ s,\ t)$ 可以在赋值函数 val 的上下文中通过算法 1 执行增量执行。因此，算法 1 首先推导出G的操作节点的一个排序，然后通过执行各节点的$update$ 函数、将所得增量应用于相应的槽位，并更新缓存的增量，来更新 val 函数。

重要的是，所采用的排序必须保证结果正确，也就是说，执行后 G 的槽位内容必须与其所有操作节点的语义函数一致，即必须满足$\forall o \in O:\ o.valid(val)$。

如果 G 的形式是有向无环图，且操作节点不共享输出槽，那么只需对 G 的操作节点进行拓扑排序即可得到这样的排序。然而，要求采用有向无环图结构是一项重大限制，因为这实际上会禁止某些输入槽同时也是输出槽的双向转换。此外，完全不允许共享输出槽这一假设虽然是防止操作结果被覆盖所必需的，但也成为实现若干理想用例的另一项障碍，例如涉及双向转换链的用例。

根据 eGDN 操作节点在非递归性和潜在更新方向方面的性质，对于某些循环扩展图依赖网络，在放宽对共享输出槽的假设后，也可以找到合适的顺序。算法 2 展示了对一个 eGDN G 的分析，该 eGDN 仅包含采用非递归 $update$ 过程的节点，以及一个包含初始时内容已修改的槽位的集合 S_i 。如果成功，该算法将返回一个执行顺序，可用于替代算法 1 中的拓扑排序。重要的是，计算出的顺序仍能生成一个与所有操作语义一致的赋值函数。

该算法首先创建一个数组 C，其中每个操作节点对应一个单元格，并将其初始化为空集合。它还将队列 Q初始化为所有与集合 S_i 中的槽位相连的操作节点；对于每个这样的操作节点，则将其输入槽中同样位于 S_i 中的槽位集合存入C中的相应单元格。随后，使用初始化后的队列 Q ，在 eGDN结构上执行略作修改的广度优先搜索，从而在没有具体输入的情况下模拟G 的执行。

因此，该过程会循环执行，直到 Q为空。每次循环执行时，都会将 Q 中的第一个操作节点 o 出队。然后，根据 o的潜在更新方向以及可能涉及的槽位集合，获取因执行 o's 更新过程 S_o 而可能产生增量的所有输出槽节点，这些槽位可能

过程 ExecuteIncrementalDAG($G = (O, S, E, s, t)$ val**) 输入：** G**：** **eGDN**val**：** G**的槽位赋值函数**1 $D \leftarrow$ **FindValidUpdateOrder(**$O, \{s \in S | \exists o \in O : o.\Delta[s] \neq \emptyset\}$**); 2 如果** $D \neq$**为空，则 3 对每个** $o \in D$**，执行**4 $\Delta_o \leftarrow o.update($ val**); 5 对每个** $s_o \in out($$o$**)，执行**6 $val($$s_o$**)** $\leftarrow$ **apply(** $val($$s_o$**)** $\Delta_o($$s_o$**)); 7 对每个** $o' \in out($$s_o$**)，执行** 8 $o'.\Delta[s_o] \cup \Delta_o$**; 9 结束 10 结束 11 对每个** $s \in in($$s$**)** $\cup$ $out($$s$**)，执行** 12 $o.\Delta[s] \leftarrow \emptyset$**; 13 结束 14 结束 15 结束**

算法 1：基于 eGDN 操作节点顺序执行 eGDN 的增量算法

当前包含未处理增量，这些增量从 C 中取出。之后，将所有与 S_o 中某个槽位相连的操作节点 o' 添加到 Q 中（如果它们尚未包含在其中）。此外，根据 S_o 更新 C 中存储的、可能包含未处理增量的 o 输入槽集合。当前正在处理的节点 o 是一个例外：它不会再次加入队列，其可能包含未处理增量的输入槽集合会重置为空集；这是利用了 eGDN 中所有 $update$ 过程均为非递归这一假设。

执行期间，算法会在触发图 G_T 中跟踪 G 的操作之间的依赖关系。一旦检测到循环依赖，执行便会立即中止并返回 **null**；这可能表明，对于初始填充的槽位 S_i，G 的执行可能陷入无限循环。这也保证了第 10 行的循环完整执行后，G_T 是一个有向无环图。

最后，返回 G_T 的一种拓扑排序，作为可能的规范执行顺序；在上述假设下，该顺序会为输入 eGDN 的槽位生成一个赋值函数，并使其与该 eGDN 所有操作节点的语义函数一致。

尽管所提出的算法旨在处理模型网络及模型操作的增量变更，但对于批处理情况——即需要先执行模型操作，以便为一组初始的现有模型生成相应的查询结果和转换后的模型——也可以通过简单直接的方式加以处理。因此，可以通过推导出相应创建操作的平凡序列，来模拟对初始现有模型的增量构建，并将其作为算法的起点。这

查找有效更新顺序过程($G = (O, S, E, s, t)$ S_i) 输入： G: eGDNS_i: 初始变更槽位集合1 $C \leftarrow$新 $Array(|O|)$; 2 $C.init(\emptyset)$;3 $Q \leftarrow$ 新 $Queue$;4 $G_T =$新 $Graph$; 5 对每个 $o \in in(S_i) \cup out(S_i)$ 执行6 $Q.enqueue(o)$; 7 $C[o] \leftarrow S_i \cap in(o)$; 8 结束9 $G_T.addVertices(Q)$; 10 当 $\neg Q.isEmpty()$ 时11 $o \leftarrow Q.dequeue()$; 12 $S_o \leftarrow o.dir_{\Delta}(o, C[o])$;13 $O_o \leftarrow out(S_o) \cup in(S_o) \setminus \{o\}$; 14 对每个 $o' \in O_o$ 执行 15 如果 $\neg o' \in Q$ 则 16 $Q.enqueue(o')$; 17 结束 18 $C[o'] \leftarrow C[o'] \cup (S_o \cap in(o'))$; 19 $G_T.addVertexIfNotExists(o')$; 20 $G_T.createEdgeIfNotExists(o, o')$; 21 如果 $G_T.hasCycle()$ 则 22 返回 null; 23 结束 24 结束 25 $C[o] \leftarrow \emptyset$; 26 结束 27 返回 SortTopologically(G_T);

算法 2： 用于查找 eGDN 更新顺序的静态分析算法

只需假设：当 eGDN 的所有槽位均为空时，对于 eGDN 所有操作的语义而言，这构成一个一致的赋值；这似乎是合理的。实质上，还需要这一额外假设，才能满足初始缓存的增量与当前状态保持一致的要求。

终止

在算法 2 第 10 行的循环中加入额外的终止条件，要求所构建的依赖图必须是无环的，即可保证算法 2 终止。

定理 1. 算法 *2* 总是终止。

证明. 除了第 10 行的循环外，所有循环都只遍历有限集合，并且所有单独的操作总是会终止。由于终止条件要求 eGDN 的操作节点之间不存在循环依赖，第 10 行的循环也总是会终止：每次循环迭代都会从 Q 中移除一个操作节点，因此只有在操作节点不断被添加到 Q 时，终止才会受到威胁。由于操作节点的数量有限，无限执行只能由修改后的广度优先搜索中的环导致。然而，这类环会通过 G_T 检测到，并立即导致执行中止。

□

因此，如果输入 eGDN 的 $update$ 过程总是终止，也能保证算法 1 终止。

定理 2. 对于输入 *eGDN* G，如果 G 的操作节点的$update$过程总是终止，则算法 *1* 总是终止。

证明. 根据定理 1，算法 2 总会终止，要么中止，要么返回一个操作节点序列。返回这样的序列意味着该序列是有限的。因此，第 3 行的循环只会执行有限次。由于所有其他循环都只遍历有限集合，并且根据关于 G 的$update$ 过程的假设，各个操作总会终止，所以算法 1 总会终止。

□

正确性

下面的定理陈述了这样一种规范执行顺序的正确性：当所有 dir_Δ 函数均为并单调时，该顺序由算法 2 的一次执行产生。

定理 3. 对于输入 $G = (O, S, E, s, t)$ 和 val，如果其中所有 $update$ 过程均正确且非递归， G 中的所有 dir_Δ 函数均并单调，并且将缓存于 G 中的增量应用之前的赋值函数与 G 的操作节点语义一致，那么算法 *1* 将中止，或产生最终赋值函数 val，使得 $\forall o \in O : o.valid(val)$。

证明。 如果算法 1 未中止，则通过对算法 2 一次终止执行所产生的有向无环依赖图 G_T 进行拓扑排序，生成了 G 的操作节点的规范执行顺序 R 。

由于 G 的操作节点具有非递归性，我们知道，通过算法 1 执行操作节点 o 后， $o.valid(val)$ 成立。因此，对于操作节点 o，只有在执行完整序列 R 后， $\neg o.valid(val)$ 才可能成立；这要求存在某个操作节点 o’，它在 R 中位于 o 之后，并且会更改与 o 相邻的槽位内容，或者 $o \notin R$。考虑到所有具有初始内容被修改的相邻槽位的操作节点一开始都已加入 G_T，算法已经终止，并且在缓存修改 G 的槽位之前，槽内容与所有操作节点的语义一致，因此对于节点 $o \notin R$，只有在存在某个会更改与 o 相邻的槽位 s 内容的操作节点 $o' \in R$ 时， $\neg o.valid(val)$ 才可能成立。

无论哪种情况，我们都知道，在 G_T 中都不可能存在从 o' 到 o 的边，因为 o' 在拓扑排序中位于 o 之后，或者因为添加这样的边会导致 o 被加入 G_T ，进而导致 R。这意味着，根据 $o'.dir_\Delta$ 的定义以及假设的并集单调性，在执行 o' 之前，必定存在一个槽位 s' ，满足 $o'.\Delta[s] \neq \emptyset$ 且 $s \in o'.dir_\Delta(\{s'\})$，但在第 11 行将 o' 从队列 $C[o']$ 中取出时，该槽位从未属于槽位集合 $C[o']$。由于只有在将 o' 从 $C[o']$ 中取出并添加相应边时，槽位才会被移除，因此我们知道 $s' \notin S_i$ ，进而在算法 1 开始时 $o'.\Delta[s'] = \emptyset$ ；否则， o' 本应被加入 Q ，而 o' 与 o 之间的边最终也会建立。

因此，必须存在一个节点 o'' ，它在 R 中位于 o' 之前，并修改了 s' 的内容。此外，对于 o''，执行 o'' 之前必须存在一个槽位 s'' ，满足 $o''.\Delta[s''] \neq \emptyset$ 和$s' \in o''.dir_\Delta(\{s''\})$，且在 o'' 出队时，该槽位从未在 $C[o'']$ 中（否则， o' 和 o 之间最终会建立一条边）。因此，同样必须存在一个操作节点，在 R 中位于o'' 之前，并修改了 s'' 的内容；它适用的约束与 o'' 相同。最终，这意味着序列中的第一个操作节点必定有一个前驱，会改变某个槽节点的内容，这显然构成矛盾。

因此，在 R 中，不可能存在某个操作节点，其执行会改变与 R 中的先前操作节点或未包含在 R中的操作节点相邻的槽位的内容。因此，我们知道，执行 R后， $\forall o \in O : o.valid(val)$。 □

如果 eGDN 的 $update$ 函数仅在有条件的情况下正确，则必须针对 eGDN结构引入额外约束，以保证正确性。具体而言，如果输出槽并非所有共享该输出槽的操作节点的输入槽，操作节点就不得共享输出槽；而对于某个节点而言，若其输出槽并非同时也是该节点的输入槽，则用户不得修改这些输出槽的内容。

直观而言，这些条件对 eGDN结构施加了限制：操作节点的输出槽中的内容不得被

另一个操作节点或用户修改，除非该操作节点能够察觉并处理这些更改。

推论 *1.* 假设 $\forall o_1, o_2 \in O:\ o_1 \neq o_2 \rightarrow \forall s \in out(o_1) \cap out(o_2) : s \in in(o_1) \wedge s \in in(o_2)$ 且 $\forall o \in O:\ \forall s \in out(o):\ o.\Delta[s] = \emptyset$，对于输入 $G = (O, S, E, s, t)$ 和 val，若 G 中的所有 $update$ 过程都是条件正确且非递归的，G 中的所有 dir_Δ 函数都是并单调的，并且缓存在 G 中的增量之前的赋值函数与 G的操作节点的语义一致，则算法 *1* 会中止，或生成最终赋值函数 val，使得 $\forall o \in O:\ o.valid(val)$。

证明。 根据关于 G的操作节点输出槽位的附加假设可直接得出，条件正确性定义中的条件从未被违反。因此，定理 3 的陈述也适用于条件正确的 $update$ 函数。 □

值得注意的是，算法 2 第 5 行和第 14 行将操作节点加入队列 Q 的顺序并未定义。由于操作节点在 Q 中的顺序会影响算法的行为，这可能意味着算法 2 最终并非确定性的。

不过，我们可以证明，如果算法 2 没有因 G_T中的环而中止，那么最终的依赖图 G_T 是唯一确定的，与操作节点加入 Q的顺序无关。因此，算法 2 中影响结果的唯一剩余非确定性，源自算法末尾的拓扑排序；这是一种本质上具有非确定性的操作。

定理 4. 对于输入 $G = (O, S, E, s, t)$ 和 S_i，如果 G 中的所有 dir_Δ 函数都是并单调的，那么算法 *2* 第 *10* 行的循环完整执行后得到的依赖图 G_T在同构意义下是唯一确定的。

证明。最初加入 G_T 的顶点集合由$in(S_i) \cup out(S_i)$唯一确定。由于只有在创建边的同时才会添加额外顶点，因此第 10 行循环执行期间添加的顶点集合由已添加的边集合决定。

为了说明算法的输入唯一确定新增的边，我们证明，在循环终止的执行中，初始集合 S_i 与 eGDN G 结合后，唯一确定一组操作节点对 (o_1, o_2)，并在 G_T 中于这些节点对之间创建有向边。

S_i 唯一确定了最初加入 Q的操作节点集合 $O_Q = in(S_i) \cup out(S_i)$。对于其中的每个操作节点 $o_Q \in O_Q$，由于 $o_Q.dir_\Delta$ 的单调性，并且只有在 o_Q 出队并经过处理后，槽位才会从$C[o_Q]$ 中移除，因此至少会将对应于各对$edges_S(o_Q, S_i) = \{(o_Q, o_T) | o_T \in out(S_o) \cup in(S_o) \setminus \{o_Q\}\}$的边加入 G_T；当 o_Q 出队时，$S_o = o_Q.dir_\Delta(S_i \cap in(o_Q))$。根据关于并集单调性的假设，我们还可以写作 $edges_S(o_Q, S_i) = edges_\emptyset(o_Q) \cup \bigcup_{s_i \in S_i \cap in(o_Q)} edges_N(o_Q, s_i)$，其中 $edges_\emptyset(o_Q) = \{(o_Q, o_T) | o_T \in$

$out(o_Q.dir_\Delta(\emptyset)) \cup in(o_Q.dir_\Delta(\emptyset)) \setminus \{o_Q\}\}$ 以及 $edges_N(o_Q, s_i) = \{(o_Q, o_T) | o_T \in out(o_Q.dir_\Delta(in(o_Q) \cap \{s_i\})) \cup in(o_Q.dir_\Delta(in(o_Q) \cap \{s_i\})) \setminus \{o_Q\}\}$。

此外，对每个出队的$o_Q \in O_Q$ 所进行的 C 和 Q 修改，可能会导致后续添加更多边。具体而言，对于每个 $s_o \in o_Q.dir_\Delta(\{s_i\})$和每个 $o_T \in out(s_o) \cup in(s_o) \setminus \{o_Q\}$，如果 o_T尚未包含在其中，就会将其添加到 Q 中，随后按照处理 o_Q的相同方式进行处理；此时保证s_i 位于 $C[o_T]$ 中。这会导致添加所有与这些对相对应的边： $edges_N(o_T, s_o)$，并再次触发更多边的添加。由于 $o_T.dir_\Delta$ 具有单调性，并且只有在 o_T 出队时才会从 $C[o_T]$ 中移除槽位，因此这些边的添加独立于此期间可能对 $C[o_T]$ 进行的任何其他修改。此外，根据关于 $o_T.dir_\Delta$并集单调性的假设，对 $C[o_T]$ 进行组合修改所产生的额外边，不会多于分别处理 $C[o_T]$各成员时所产生的边。

由于无法以任何其他方式修改 $C[o_T]$，也无法以任何其他方式向 G_T 添加边，因此，在 G_T 中创建有向边的操作节点对 (o_1, o_2) 集合由函数 $edges(S_i) = \bigcup_{o_Q \in in(S_i) \cup out(S_i)} (edges_\emptyset(o_Q) \cup \bigcup_{s_i \in S_i \cap in(o_Q)} edges_R(o_Q, s_i))$ 给出，其中 $edges_R(o_Q, s_i) = edges_N(o_Q, s_i) \cup \bigcup_{o_T \in O_T)) \setminus \{o_Q\}} \bigcup_{s_o \in o_Q.dir_\Delta(in(o_Q) \cap \{s_i\}} edges_R(o_T, s_o)$，而 O_T 由 $O_T = out(o_Q.dir_\Delta(in(o_Q) \cap \{s_i\})) \cup in(o_Q.dir_\Delta(in(o_Q) \cap \{s_i\}$ 给出。

循环因 Q 变为空而终止，这意味着所有曾添加到 Q 中的节点都已处理完毕，因此所有相应的边也都已添加到 G_T中。由于已确保对于每一对操作节点（o_1， o_2）只添加一条相应的边，我们知道，无论具体的处理顺序如何，G_T 始终恰好包含每一对（o_1， o_2）$\in edges(S_i)$对应的一条有向边。

由于已添加顶点的集合由已添加边的集合唯一确定，且每个顶点只能添加一次，因此对于输入 G 和 S_i， G_T的顶点集合是唯一确定的。

因此，在算法 2 第 10 行的循环完整执行结束时，图 G_T对于输入 G 和 S_i 是唯一确定的，与在第 5 行和第 14 行将操作节点添加到 Q 中的顺序无关。□

G_T 由输入 G 和 S_i 唯一确定这一事实还意味着：如果算法 1 的一次执行在未中止的情况下终止，那么对于相同输入的任何可能执行也都会终止。

定理5。 对于输入 $G = (O, S, E, s, t)$ 和 S_i，算法 *1* 第 *10* 行中的循环执行若未中止而终止，当且仅当对于相同输入的任何其他执行也未中止而终止。

证明。 根据定理 1，算法 1 第 10 行中的循环总会终止，原因要么是违反了循环条件，要么是循环中止。由于当且仅当在 G_T 中任意时刻检测到环时循环才会中止，而且 G_T中的边绝不会被移除，因此，循环只有在循环执行期间添加到 G_T 中的边集不构成环时，才会在不中止的情况下终止。由于循环执行期间添加到 G_T 中的边集仅由输入 和 函数性地决定，因此

如果循环的一次执行在 G 和S_i的情况下未中止而终止，那么在相同输入下的任何执行也都会未中止而终止。

□

此外，我们可以证明，如果存在一个执行序列，能够保证最坏情况下结果正确，且每个操作节点最多执行一次，那么算法 2 就能找到这样的序列。

定理 6. 对于输入 *eGDN* $G=(O,S,E,s,t)$，其过程 $update$ 正确且非递归，函数 dir_Δ 并单调，槽位集合 $S_i \subseteq S$ 中的内容最初已修改，赋值函数为 val。假设

1. 对于任意操作节点 $o_1 \in O$，对于采用赋值函数 val' 和输入槽 S_Δ 的增量执行 $o_1.update(val')$，有 $\forall s_o \in o_1.dir_\Delta(S_\Delta) : o_1.update(val')(s_o) \neq \emptyset$;
2. 对于具有 $o_1 \neq o_2$ 的第二个节点 $o_2 \in O$，有 $\exists s_o \in out(o_1) \cap in(o_2) : o_1.update(val')(s_o) \neq \emptyset \rightarrow \neg o_2.valid(val'')$，其中对于 $s \in S$

$$val''(s) = \begin{cases} apply(val'(s), o_1.update(val')(s)) & if\ s \in out(o_1) \cap in(o_2) \\ val'(s) & otherwise \end{cases} \tag{1}$$

and

3. 成立：$\forall o \in in(S_i) \cup out(S_i)$：$\neg o.valid(val)$,

如果存在一个序列，通过算法 *1* 执行该序列即可保证得到正确的结果赋值函数，并且该序列中每个节点 $o \in O$ 仅出现一次，那么算法 *2* 将返回这样的序列。

证明。 在给定假设下，算法在终止或中止前创建的边集合 G_T 表示操作节点对之间所有关系的一个子集（o_1， o_2）。其中，必须至少执行一次 o_1 的 $update$ 过程才能生成正确的最终赋值函数，并且该执行会修改与 o_2 相邻的槽位的内容，因此根据假设（2），随后必须执行 o_2 。

这是因为，为恢复一致性，根据假设（2）和（3），$o \in in$（S_i）$\cup out$（S_i）中的所有操作节点都必须至少执行一次。所有这些操作节点 o_1 最初都会在算法 2 中被加入队列 Q 。根据假设（1），每次执行操作节点 o_1都会修改 $o_1.dir_\Delta$（$S_i \cap in$（o_1））中的所有槽位，因此根据假设（2），随后必须执行 $o_2 \in in$（$o_1.dir_\Delta$（$S_i \cap in$（o_1））$\cup out$（$o_1.dir_\Delta$（$S_i \cap in$（o_1））中的所有操作节点。算法 2 在 o_1 出队时，为所有这些节点对（o_1， o_2）创建边。

随后对任意操作节点 o_2 的执行同样需要执行所有节点 $o_3 \in in(o_2.dir_\Delta(S_i \cap in(o_2))) \cup out(o_2.dir_\Delta(S_i \cap in(o_2)))$；当 o_2 出队时，算法 2 所创建的边也反映了这一点，

依此类推。由于根据关于 dir_{Δ} 函数并集单调性的假设，该算法不会创建额外的边，因此 G_T 中的所有边都表示操作节点[5]排序中的此类必要关系。

由于当 G_T 无环时，算法 2 始终会生成正确的操作节点序列，因此可以假设，在算法未生成排序的情况下， G_T 中至少存在一个环。因此，不可能存在一个涉及该环中操作节点的序列，使得每个节点仅包含一次，且每个节点都至少在其前驱之后执行一次。由此通过逆否命题可知，如果存在一个能够保证结果正确、且每个操作节点仅包含一次的操作节点序列，那么算法 2 就能找到这样的序列。

□

请注意，根据定理 6 中的假设，可能存在一些能够保证结果正确的有限操作节点执行顺序，但算法 2 无法找到这些顺序。然而，这些顺序要求至少有一个操作节点被执行至少两次。

5.3 任意 eGDN 的增量执行

如果 eGDN 不是有向无环图，并且无法通过算法 2 找到其操作节点的合适排序，则可以改为通过一种简单的不动点迭代来实现增量执行，如算法 3 所示。

算法 3 首先将输入 eGDN 中所有输入槽或输出槽发生变化的操作节点，初始化为需要执行的操作节点集合D 。然后，算法持续迭代，直至达到不动点。

因此，下次迭代中需要执行的操作节点集合 D_n 初始化为空集。之后，对于当前迭代中需要执行的每个操作节点 o ，将该节点从集合 D 中移除。然后，调用 o 的 $update$ 过程，计算一组对 o输出槽内容的更改，使其与 o 的语义保持一致。

对于 o 的每个输出槽 s_o ，如果 $update$ 已计算出更改，则随后应用这些更改，并在每个操作节点 o' 处进行适当登记，其中 s_o 是该节点的输入槽。如果这样的 o' 在当前迭代中尚未需要执行，则将其加入 D_n，标记为在下一次迭代中执行。对于将 s_o 作为输出槽的操作节点，也以同样方式标记为需要执行。最后，在考虑完 D 中的所有操作节点后，用 D_n 替换 D ；如果 D_n 非空，则开始新一轮迭代。

与算法 1 类似，算法 3 也可以通过将现有模型编码为元素创建序列，来处理模型的 eGDN 初始执行的批处理情况。

[5]顺便一提，由于操作节点可以在输入槽集合不同时出队并执行，而这些输入槽可能已被修改，因此 G_T 中节点之间的边（o_1，o_2）并不一定意味着 o_2 必须在 o_1 执行后执行 any ，而只表示后续执行是必要的 $at\ least\ once$。

过程 ExecuteIncremental($G = (O, S, E, s, t)$ val)

输入： G：eGDN
val：G 的槽位赋值函数

$D \leftarrow \{o \in O | \exists s \in in(o) \cup out(o) : o.\Delta[s] \neq \emptyset\}$;
当 $D \neq \emptyset$ **时执行**
  $D_n \leftarrow \emptyset$;
  对于每个 $o \in D$ **执行**
    $D \leftarrow D \setminus \{o\}$;
    $\Delta_o \leftarrow o.update(val)$;
    对于每个 $s \in in(s) \cup out(s)$ **执行**
      $o.\Delta[s] \leftarrow \emptyset$;
    end
    对于每个 $s_o \in out(o)$ **执行**
      如果 $\Delta_o(s_o) \neq \emptyset$ **则**
        $val(s_o) \leftarrow$ 应用$(val(s_o)\ \Delta_o(s_o))$;
        对于每个 $o' \in out(s_o)$ **执行**
          $o'.\Delta[s_o] \cup \Delta_o$;
          如果 $o' \notin D$ **则**
            $D_n \leftarrow D_n \cup \{o'\}$;
          end
        end
        对于每个 $o' \in in(s_o)$ **执行**
          如果 $o' \neq o \wedge o' \notin D$ **则**
            $D_n \leftarrow D_n \cup \{o'\}$;
          end
        end
      end
    end
  end
  $D \leftarrow D_n$;
end

算法 3： 用于 eGDN 执行的增量算法

终止

与算法 1 相比，算法 3 不保证终止，因为操作节点的循环传递依赖可能导致某些槽节点的内容发生无限次循环变化。如果不限制开发者可以指定何种 eGDN，这个问题就不可避免。

然而，在实践中，即使存在循环结构，仍然可以确保类似 eGDN 的模型操作网络终止。例如，在某些情况下，网络层面的环并不一定对应模型操作的实际循环依赖，因为相关模型操作可能只影响槽内容中彼此不同的部分，例如某些不同类型的元素。在某些情况下，重构 eGDN 可以在保持语义的同时消除结构层面的环，例如将不具有有效自反依赖的原位模型转换改为具有不同输入模型和输出模型的模型转换。

此外，模型操作的环可能表现出单调行为，例如在每次迭代中删除某些永远不会重新创建的元素，从而保证收敛。然而，归根结底，开发者仍有责任创建不会在执行过程中导致无限循环的模型操作网络。

正确性

如果算法 3 终止，则可以保证结果赋值函数与输入 eGDN 中所有操作节点的语义一致。

定理 7。 对于输入 $G = (O, S, E, s, t)$ 和 val，如果算法 *3* 终止，则所有使用的 $update$ 过程都是正确且非递归的；并且，如果在应用 G 中缓存的增量之前，赋值函数与 G的操作节点的语义一致，则算法会生成最终赋值函数 val，使得 $\forall o \in O:\ o.valid(val)$。

证明。 我们通过对循环迭代次数进行归纳，证明循环第 2 行中的不变量 (1) $\forall o \in O:\ \neg o.valid(val) \rightarrow o \in D$ 成立。

不变量 (1) 的基本情形由于 D 的初始化以及关于初始缓存的增量和先前赋值函数的假设而成立。

为证明不变量 (1) 的归纳步骤，我们首先证明在归纳假设下，循环第 10 行中的不变量 (2) $\forall o \in O:\ \neg o.valid(val) \rightarrow o \in D \cup D_n$成立。这同样可以通过归纳法完成。

由于不变量 (1) 的归纳假设，不变量 (2) 的基础情形成立。

对于不变量 (2)，归纳步骤成立，因为在内层循环的每次迭代中，都只会通过其 $update$ 过程执行一个操作节点 o，并将其从 D中移除，同时更新 val 和缓存的增量。如果执行 o 不会改变其自身某个输入槽的内容，那么根据关于$update$ 过程正确性的假设，并且由于缓存的增量始终得到正确更新，我们知道之后 $o.valid(val)$。否则，在第 13 行的循环中，会将 o 添加到 D_n 中。第 13 行和第 19 行的循环

还会将所有操作节点 o' 添加到 D_n 中；这些节点的 $o'.valid(val)$结果可能受到对 val的更新的影响。因此，根据归纳假设，在第 10 行循环结束时，我们再次有 $\forall o \in O : \neg o.valid(val) \rightarrow o \in D \cup D_n$ ，所以归纳步骤成立。

由于第 10 行循环结束时， $D = \emptyset$，我们知道 $\forall o \in O : \neg o.valid(val) \rightarrow o \in D_n$。由于第 2 行循环的每次迭代结束时，集合 D 都会被替换为 D_n，因此（1）的归纳步骤成立。

只有在用 D_n替换后满足 $D = \emptyset$ 时，才会退出第 2 行的循环，因此我们知道，如果算法终止，则 $\forall o \in O : o.valid(val)$。 □

与算法 1 类似，如果所有使用的 $update$ 过程至少是条件正确的，e GDN 的节点不共享那些并非所有共享节点的输入槽的输出槽，并且不存在针对某节点输出槽的增量（该输出槽同时不是输入槽），那么该算法也会产生正确的结果。

推论2。假设 $\forall o_1, o_2 \in O : o_1 \neq o_2 \rightarrow \forall s \in out(o_1) \cap out(o_2) : s \in in(o_1) \wedge s \in in(o_2)$ 且 $\forall o \in O : \forall s \in out(o) : o.\Delta[s] = \emptyset$，对于输入 $G = (O, S, E, s, t)$ 和 val，如果算法 3 终止，所有使用的 $update$ 过程都是条件正确且非递归的，并且在 G 中缓存增量之前的赋值函数与 G 的操作节点语义一致，那么它会生成最终赋值函数 val ，使得 $\forall o \in O : o.valid(val)$。

证明。 根据关于 G 操作节点输出槽的附加假设，可以直接得出，条件正确性定义中的条件绝不会被违反。因此，定理 7 中的结论同样适用于条件正确的 $update$ 函数。

□

5.4 使用 eGDN 开发

由于算法 2 只考虑 eGDN结构，而不考虑具体的槽位内容，因此可将其用作静态分析 eGDN 的工具。具体而言，可通过该算法分析内容已修改的槽位配置，以判断相应 eGDN 执行是否会终止。例如，该算法可用于检查修改某个特定单个模型是否仍能保证终止。

如果所有用户可编辑模型都满足这一点，那么一种保守的方法是强制采用**直接传播策略**，从而始终保证 eGDN 执行终止且结果正确，同时避免对每种用户可编辑模型组合都执行分析所需的指数级开销。在此策略下，修改单个模型后，会立即传播相应更改以恢复一致性。只有完成这一步后，才允许修改另一个模型。

此外，如果通过算法 1 执行 eGDN，且初始修改的槽位为 S_i，则可调整算法 2，使其返回在 eGDN 操作下可能被自动修改的槽位集合 $closure_\Delta(S_i)$。这使得通过 eGDN 管理的模型网络能够进行协作式开发，并实现

借助**传播闭包锁定策略**，保证终止并实现无冲突一致性恢复。对于一组已修改的槽位 S_Δ，该策略仅允许在算法 2 针对集合 $S_\Delta \cup \{s\}$ 生成执行顺序时，修改另一个槽位 s 的内容。此外，为确保不会覆盖任何用户编辑，该策略会检查是否$S_\Delta \cup \{s\} \cap closure_\Delta(S_\Delta \cup \{s\}) = \emptyset$。请注意，即使同一用户希望编辑多个槽位的内容，该策略的限制也同样适用。

由于算法 3 无法保证终止，因此如果 eGDN 无法通过算法 1 执行，就需要谨慎考虑。不过，如果开发者确信其 eGDN 即使在结构层面存在循环依赖也能保证终止，则可以将算法 3 用作 eGDN 执行的后备选项。

所提出的算法还支持将子 eGDN 作为父 eGDN 的操作节点，因为它们实质上实现了所需的 $update$ 过程。

6 实现

我们以原型方式实现了若干具体的操作节点类型，用于构建 eGDN，以便在 Eclipse 建模框架（EMF） [26]的环境中使用。表 2 除了列出已实现操作的名称外，还简要说明了它们的行为。表 3 则根据本报告中定义的属性，对我们的实现进行了描述。

非递归性： </style}TGG 同步操作的 $update$过程在包含源模型、目标模型和对应模型的槽位各不相同时是非递归的。复合节点是否具有非递归性取决于子 eGDN 的具体构成。只有当输入槽位与输出槽位各不相同时，才能保证所有其他节点的 $update$ 过程是非递归的。勾号 ✓表示在此假设下过程具有非递归性 $update$ 。

潜在填充／更新方向： TGG 同步（$\leftrightarrow$）的潜在更新方向可描述如下（假设源模型、目标模型和对应模型各自具有不同的槽位）：如果所考虑的输入槽集合为空，则不会修改任何输出槽的内容。如果所考虑的输入槽集合仅包含源模型，则该操作只会修改目标模型和对应模型，反之亦然。其他情况下，所有模型都可能被修改。复合节点的潜在更新方向由子 eGDN 的确切结构决定。对于任何所考虑的输入槽集合，所有其他节点都可能修改其所有输出槽的内容。所有示例节点的潜在更新方向函数 dir_Δ 均为并单调。

正确性： 所有操作实现的 $update$过程都只是有条件地正确。实际上，这意味着操作不得共享

输出槽位，而且除非共享或编辑的输出槽位同时也是相关操作节点的输入槽位，否则不允许对操作节点的输出槽位进行用户编辑。

增量性： 勾号 ✓表示在假设数据结构理想的情况下，该过程具有完全增量性$update$ 。此外，列为非完全增量的操作在一定程度上也支持增量执行。增量性的程度取决于操作及其具体输入。自然地，只有当所考虑表达式的评估具有 $O(1)$ 的运行时间复杂度时，我们对表达式节点的实现才是完全增量的。对于模式匹配节点和 TGG 同步节点，在特定的输入模型、模式以及 TGG 下，可以实现完全增量执行。eGDN 或子 eGDN 的执行增量性程度取决于哪些槽位被指定为该 eGDN 的接口槽，以及其中包含的操作节点及其组合方式。尽管由于所采用的 OCL 解释器对集合的处理，组表达式节点也具有部分增量的更新过程，但通常无法实现完全增量执行。

作为这些操作之间的接口（即槽节点），我们的实现采用常规 EMF 模型作为模型槽，并采用基于哈希的索引作为赋值槽。与基于数组的索引相比，选择基于哈希的索引意味着理论上完全增量的操作实现与我们的槽实现结合使用时，可能无法实现完全增量；不过，基于哈希的数据结构通常因内存占用较低而更适合实际应用，并且在大多数场景中都能提供可接受的性能。

图 3 展示了图 3 中示例 eGDN 的一个更复杂版本，该版本可使用表 2 中介绍的示例 eGDN 节点实现。现在，类图到抽象语法图的转换具体通过单向 TGG 同步实现。此前由单个查询节点表示的查询操作，如今被分解为由多个子查询构成的复杂网络。该子 eGDN 包含两个标记为“$x \to y$”的模式匹配节点，用于查找由单条边组成的基本模式；还包含一个分组计数节点和一个分组求和节点，分别显示为标记“COUNT (X)”和“SUM (X)”的节点，以及一个标记为“$\bowtie$”的连接节点。此外，模式匹配节点也可以通过边输入实现。

7 评估

本章将报告基于原型实现进行的初步实证评估。此外，我们将介绍如何在典型应用场景中使用 eGDN，并根据第 3 章提出的需求评估所开发的方法。

Name	描述
RETE 节点	
节点输入	从模型中提取给定类型的单个节点
边输入	从模型中提取给定类型的单条边
Join	对存储在两个 输入赋值槽
反连接	对左侧输入赋值槽执行反连接 与右侧输入赋值槽进行比较
GDN 节点	
模式匹配	在模型中查找与给定模式匹配的内容；支持 以对象约束语言 [62] 编写的附加约束；支持 关于其他模式是否存在匹配项的约束 通过依赖关系关联的赋值 槽位
属性计算节点	
表达式	计算对象约束语言 [62] 表达式的值，适用于单个 赋值
组表达式	计算对象约束语言 [62] 表达式的值，适用于集合 按某些变量分组的赋值集合
分组计数	对集合中按某些变量分组的赋值进行计数 按某些变量分组的赋值
分组求和	计算特定变量数值的总和 能够以按特定条件分组的赋值集合的形式 变量
转换节点	
TGG 同步（→）	执行 单向 模型同步 of 通过三图文法将更改从源模型传播到目标模型和相关的对应关系模型 对应关系模型 [67]
TGG 同步 (↔)	执行更改的双向模型同步 在源模型、目标模型和相关的对应关系之间 通过三图文法生成模型 [67]
复合节点	
eGDN	执行子 eGDN，以更新已公开内容的 槽位，通过算法 1 或算法 3

表 2：示例 eGDN 节点类型

Name	非递归	方向	正确	增量
RETE 节点				
节点输入	✓	all	条件	✓
边输入	✓	all	条件	✓
Join	✓	all	条件。	✓
反连接	✓	all	条件。	✓
GDN 节点				
模式匹配	✓	all	条件。	(✓)
属性计算节点				
表达式	✓	all	条件。	(✓)
组表达式	✓	all	条件。	∼
分组计数	✓	all	条件。	✓
分组求和	✓	all	条件。	✓
转换节点				
TGG 同步 (→)	✓	all	条件。	(✓)
TGG 同步 (↔)	✓	*	条件。	(✓)
复合节点				
eGDN	?	?	条件。	(✓)

表 3：示例 eGDN 节点类型的属性

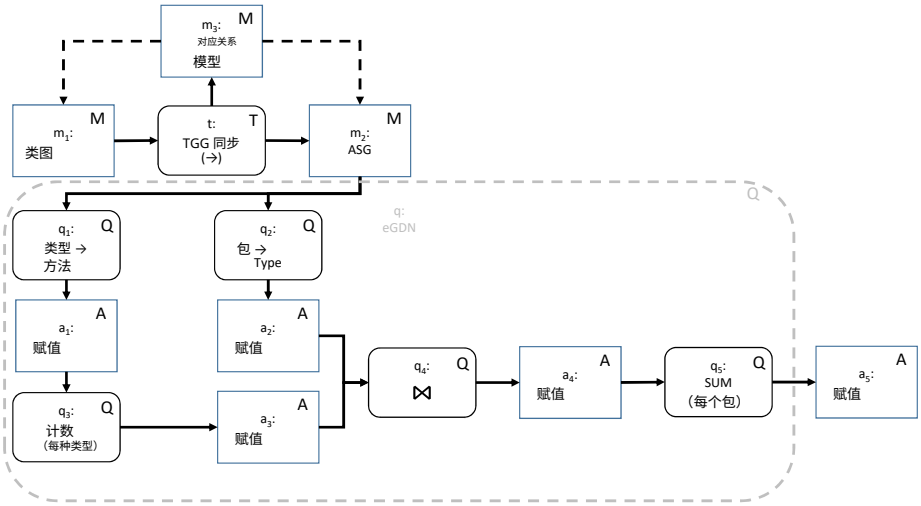


图 4：复杂示例 eGDN

7.1 性能评估

为了对所提出的方法进行初步实证评估，我们开展了一项实验，其灵感源自软件开发领域的一个应用场景：不断演变的类图作为生成面向对象代码的基础，随后对生成的代码进行分析，以计算代码度量指标。

因此，我们通过三图文法实现了一个从 Ecore 模型 [26] 到 Java 抽象语法图 [18] 的简单模型转换。对于类图中的每个类，该转换都会在第一个包中创建一个 Java 抽象语法图接口，并在第二个包中创建一个实现类。此外，对于类图中每个类的每个属性，该转换都会在抽象语法图中的相应接口和类中创建对应的字段以及相关的 getter 和 setter 方法。

此外，我们还实现了一个由多个子查询组成的模型查询，用于统计 Java 包中所有类型的方法数量。该转换和查询集成在一个 eGDN 中，其结构如图 4 所示。

使用我们的原型实现（可从 [**30**], 获取），我们将一个真实世界的 Ecore 模型 [18] 分配给类图模型槽位，并通过算法 1 对其余槽位执行初始填充。为了评估 eGDN 的可扩展性，我们随后对类图槽位中的模型应用若干次合成更新，每次更新都会为模型中的每个类添加一个属性，并通过算法 1（“增量”）测量 eGDN 处理每次此类更新所需的时间。我们将其与一个基线进行比较：在基线中，我们通过相应操作的非增量式实现，分别对模型转换和查询的结果进行完整重新计算（“批处理”）。[6]

图 5 展示了前 30 次更新的执行时间。在包含前 5 次更新的初始阶段，执行时间逐次更新递减；此后，通过“增量”策略处理类图更新的执行时间基本没有变化。特别是，随着不断执行更新，类图逐渐增大，但执行时间似乎没有呈现出随之增加的趋势。相比之下，随着类图不断增大，“批处理”的执行时间逐次更新增加。起初，它与“增量”的执行时间相近（高出 1.6 倍）；到第 30 次更新时，“批处理”的执行时间已增加到“增量”执行时间的 80 倍。

因此，测量结果表明，通过 IN- CREMENTAL 进行增量 eGDN 执行能够高效处理类图的更新：执行时间似乎仅取决于实际变更，而非模型的大小，这确实实现了增量性。因此，eGDN 似乎适合作为一种形式化方法，用于可扩展、模块化且增量式地实现

[6]所有实验均在一台运行 Linux SMP Debian 4.19.67-2 的机器上完成，该机器配备 Intel Xeon E5-2630 CPU（主频 2.3 GHz）和 386 GB 内存，并运行 OpenJDK 11.0.6 版。报告的执行时间测量值为相应实验 10 次运行的平均执行时间。

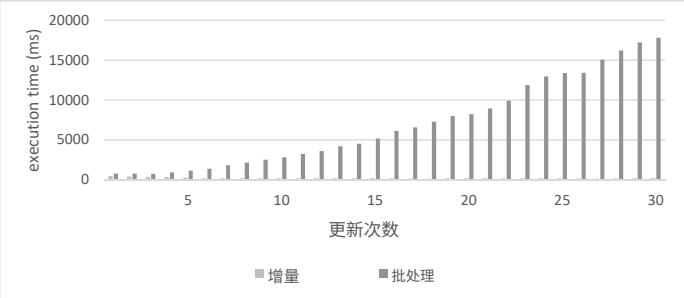


图 5：类图更新的执行时间测量结果

适用于此场景的模型操作网络。实验初期每次更新的执行时间逐渐缩短，这很可能是 Java 虚拟机预热效应所致。

Java 虚拟机的意外行为（尤其是垃圾回收）对我们结果的内部有效性构成了主要威胁。为减轻此类影响，文中报告的执行时间测量值取自多次实验运行结果的算术平均值；总体执行时间的标准差始终低于总体执行时间的 5%。

实验中使用的合成更新对外部效度构成了威胁。不过，该实验以真实应用场景为灵感，并以真实模型为基础，展示了 eGDN 方法在这一场景中的适用性。合成更新仅用于对我们技术的可扩展性进行系统评估。因此，我们并未对该方法在实际应用场景中的表现作出任何定量论断，而只是将实验结果视为所提出方法潜力的一个指标。此外，我们也未对该方法能否推广到其他应用领域作出论断；这需要进一步评估，将留待未来工作。

7.2 适用性评估

为了研究所开发技术的适用性，我们考虑以下需要全局模型管理的扩展示例场景：一个类图遵循类似于图1所示的元模型，用于通过包含在包中的类来建模正在开发的软件系统的结构。类可以包含方法，而这些方法又可以引用类作为其返回类型。单独模型中的对象约束语言表达式用于描述类图中某些方法的行为。因此，OCL模型有自己的类型表示，这些类型与类图中的类相对应。这种对应关系通过链接模型来表示，该模型只包含专用的链接顶点。一个链接顶点可以连接到类图中的一个类

以及 OCL模型中与之对应的类型；也可以连接到类图中的一个方法以及 OCL 模型中与之对应的表达式，即实现。

我们考虑以下适用于此设置的用例：

- **一致性检查：**开发者希望运行自动且增量式的一致性检查，以验证类图中某个方法的返回类型是否与该方法的对象约束语言实现所对应的类型相匹配。所开发的一致性检查还应适用于类似的设置，即使用不同于对象约束语言的表达式语言来实现方法。因此，此用例的实现需要满足需求 **R1.1**、**R1.2**、**R2.2** 和**R 2.3**的解决方案。

- **代码生成：**开发者希望根据类图自动且增量式地生成 Java代码，并以抽象语法图的形式表示。此外，还应根据类图中各方法的对象约束语言实现，生成这些方法的 Java 实现。最后，应整合两个模型对应的 Java 代码片段，并对其进行一些代码度量指标分析。因此，此用例的实现需要满足需求 **R2.1**、**R2.2** 和 **R2.3**的解决方案。

- **巨型模型复用：** 在开发出自动一致性检查和代码生成后，开发者希望在另一个具有类似模型集合的项目中复用同样的两项操作。因此，针对该用例的实现需要一种满足需求**R 3.1.2**、**R3.2**、**R3.3**和**R 3.4**的解决方案。

为了支持全部三个用例的全局模型管理，解决方案还必须能够在一组可能相互集成的模型上，对不同类型的模型操作构成的网络进行建模，即解决方案必须满足需求**R3.1.1**。

图7、图6和图8分别展示了针对**一致性检查**、**代码生成**和**巨型模型复用** 用例的示例 eGDN 实现。

如图7所示，**一致性检查**用例通过四个模式匹配查询节点实现，这些节点从基础模型中提取特定的简单模式，并通过赋值槽以通用格式使其可访问。随后，一个由三个连接查询操作和一个反连接查询操作（标记为 ▷）组成的复杂查询操作，通过分别查找类图中的操作、其在 OCL模型中的实现，以及相关的返回类和表达式类型来执行实际的一致性检查，其中返回类和表达式类型并不对应。因此，基于 eGDN 的方法在此情况下满足需求 **R1.1** 和**R2.2**，因为它对通过集成链接集成的一组模型执行了一致性检查。此外，对于提供类似

功能的不同建模语言而言，所得实现可通过赋值槽 a_1、 a_2、a_3 和 a_7 提供的通用接口实现复用，从而满足需求**R1.2**。示例 eGDN 还展示了如何由更简单的操作组合出更复杂的模型操作，从而满足需求**R2.1**，并为这些操作提供增量执行方案，从而满足需求**R2.3**。特别是，通过算法 2 可以验证，算法 1 提供了一种执行 eGDN 的方法，能够保证对三个基础模型任意组合的变更均获得正确的结果并实现终止。

图6所示的 eGDN 通过结合两个 TGG 同步，实现了**代码生成**用例：将类图和 OCL模型转换为 Java抽象语法图。两个 Java模型通过专用链接模型进行集成，该链接模型由原始链接模型与 TGG 同步所创建的对应模型进行单向模型转换生成。最后，可以在抽象语法图上执行查询操作，以计算代码度量指标。使用算法 2分析 eGDN，可以确定，对于三个基础模型的任意组合发生的更改，都能保证通过算法 1终止执行。该示例展示了 eGDN 如何提供统一且模块化的模型操作概念，以及增量执行方案；同时还展示了模型操作的组合，从而满足需求**R2.1**、**R2.2**和**R2.3**。

图7和图6中的 eGDN 还共同说明了 eGDN 如何作为巨模型建模语言使用，支持不同类型的模型操作，包括模型属性（如代码生成用例中计算的度量指标）、模型一致性（如一致性检查用例中的一致性条件），以及模型转换和同步（如代码生成用例中的转换和同步）。它还展示了如何在该语言中表示集成视图（如图7中槽位 a_8 内的跨模型一致性查询结果）和可追溯性链接（如 TGG 同步生成的对应模型）。尽管示例 eGDN 中未体现，类图和 OCL元模型本身也是模型，因此可以通过将其纳入专用模型槽而轻松显式表示。元模型或常规模型的良构性条件可以通过常规查询操作实现并进行处理。因此，eGDN 满足需求**R3.1.1**。

最后，图 8 中**巨型模型复用** 用例的 eGDN 实现，将图7和图 6 中的 eGDN 视为一个总体 eGDN 中的操作节点。这体现了 eGDN 如何通过将子 eGDN 视为可通过通用执行方案执行的常规操作，在巨模型层级实现模块化和增量性；该方案还允许在执行前累积多项变更。因此，该示例展示了需求**R3.1.2**、**R3.2**和**R3.4**均得到满足。该 eGDN 还展示了槽位如何作为这些巨模型操作的接口，从而满足需求**R 3.3**。

因此，eGDN 可用于实现三个示例用例所需的功能，满足本

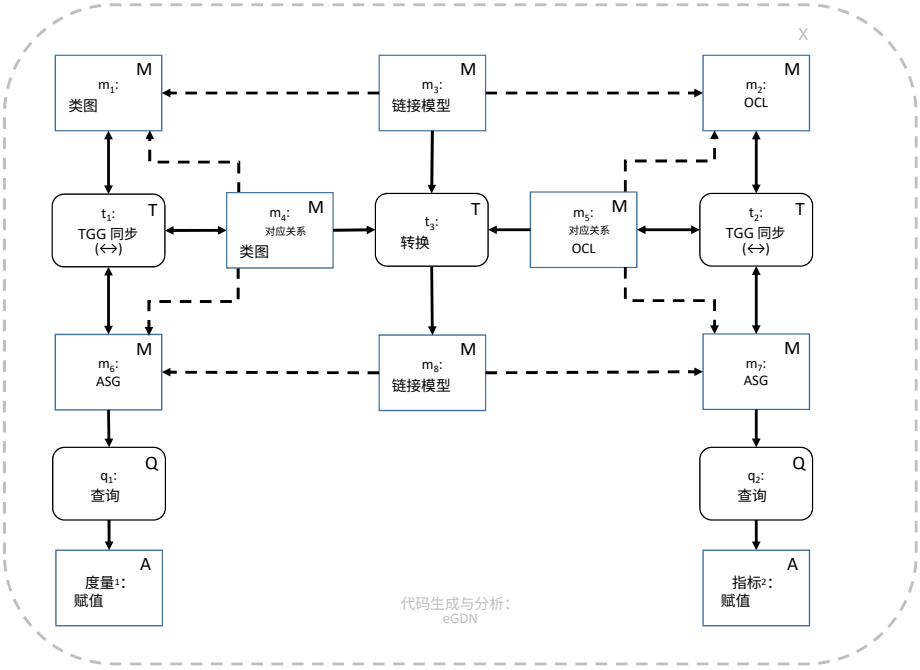


图6：实现代码生成用例的 eGDN 示例

场景。

表4总结了示例用例和 eGDN 方法对各项需求的覆盖情况，其中“○”表示某个用例的实现与某项需求相关，“✓”表示在此场景中，eGDN 能够满足某项需求。

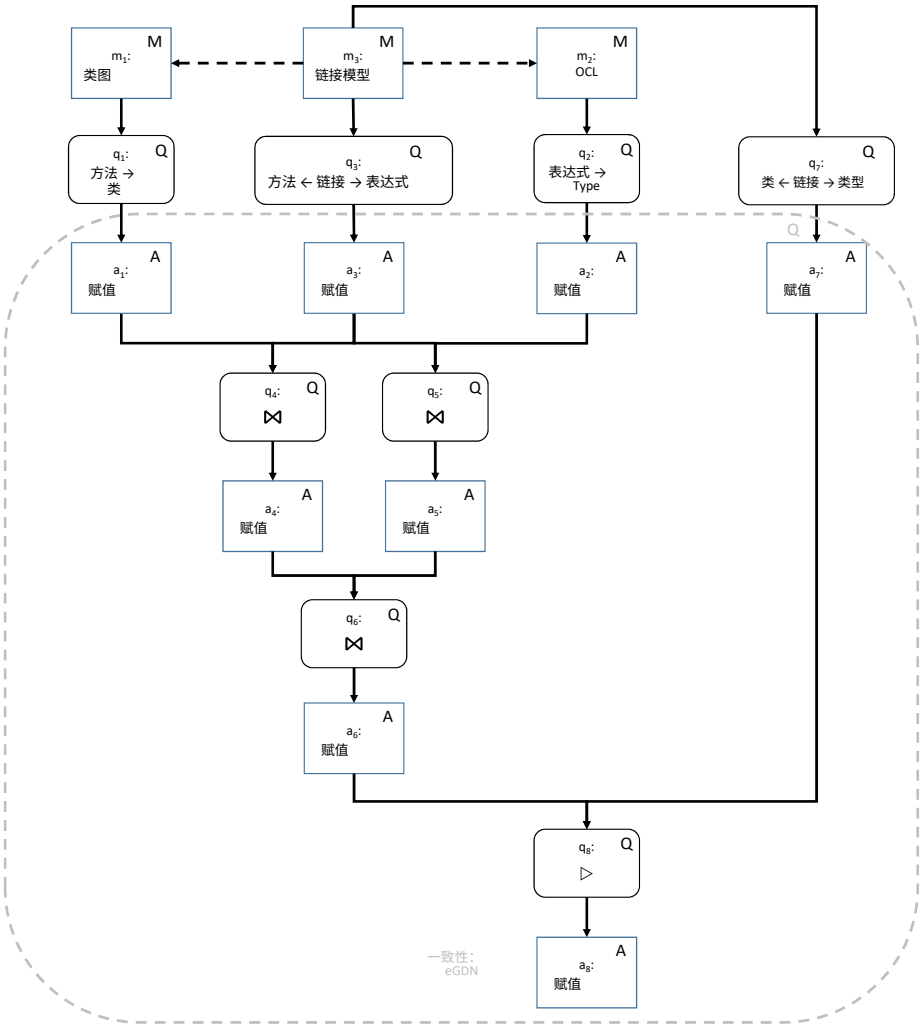


图7：实现一致性检查用例的 eGDN 示例

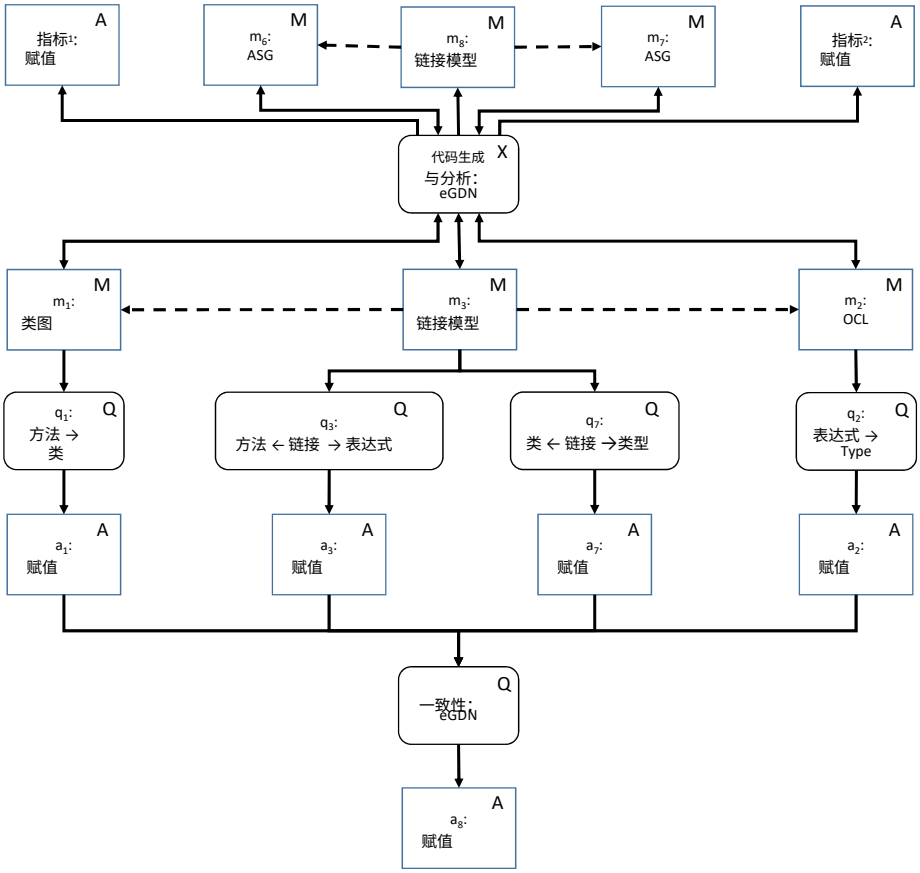


图 8：实现巨型模型复用用例的 eGDN 示例

R1.1：建模语言集成	○			✓
R1.2：建模语言的嵌入接口	○			✓
R2.1：模型操作的组合		○		✓
R2.2：针对集成模型的模型操作	○	○		✓
R2.3：模型操作的执行方案	○	○		✓
R3.1.1：巨模型建模语言	○	○	○	✓
R3.1.2：巨模型操作模块概念			○	✓
R3.2：健壮的巨模型执行方案			○	✓
R3.3：巨模型接口			○	✓
R3.4：异步巨模型执行方案			○	✓

表4：第 3 章需求覆盖情况

8 结论

在本报告中，我们进一步推广了 GDN 机制，提出了名为 eGDN 的机制，使复杂模型操作网络能够以模块化和增量方式进行构建与执行，其中包括模型属性、模型一致性、模型转换和模型同步。除了对 eGDN 进行形式化定义之外，我们还为其执行提供了增量算法。此外，我们介绍了若干 eGDN 节点，并对其进行了原型实现，以便针对该方法的可扩展性开展初步实证评估。我们的实验基于软件开发定义域中的一个应用场景，结果表明，所提出的技术可用于实现高效的全局模型管理。此外，我们还根据已识别的全局模型管理解决方案需求，从概念层面对该方法进行了评估。

在未来的工作中，我们计划针对 eGDN 在真实应用场景中的表达能力和性能开展更为广泛的评估。这还可能包括实现其他类型的 eGDN 节点，并最终实现对 eGDN 规范和执行的真正工具支持。此外，我们将研究如何将所提出的概念扩展到由多个不同版本组成的演化建模环境。我们还将探索，这种扩展如何通过派生其他版本，帮助缓解 eGDN 执行中可能出现的无限循环或覆盖用户编辑等问题。

致谢

这项工作主要是在模块化和增量式全局模型管理项目（项目编号 336677879）的开展过程中完成的，该项目由德国研究联合会资助。

参考文献

[1] *AM3* 项目主页。 https://wiki.eclipse.org/AM3。[2] *AMW* 项目主页。 https://projects.eclipse.org/projects/ modeling.gmt.amw/。[3] *ATL* 项目主页。 https://eclipse.org/atl/。[4] *ATLFlow* 项目主页。 http : / / opensource . urszeidler . de / ATLflow/。[5] *AToMPM* 项目主页。 http :/ /www- ens . iro. umontreal . ca/~syriani/atompm/atompm.htm。[6] Thomas Beyhl、Regina Hebig 和 Holger Giese。“用于维护可追溯性链接的模型管理框架”。载于：软件工程 *2013* 研讨会论文集。由 Stefan Wagner 和 Horst Lichter 编。第 P-215 卷。计算机科学讲义（LNI）。亚琛：Gesellschaft f¨ur Informatik (GI)，2013 年 2 月，第 453–457 页。[7] Thomas Beyhl 等人。“使用广义判别网络实现图查询的操作化”。载于：国际图变换会议。Springer。2016 年，第 170–186 页。[8] Jean B´ezivin、Fr´ed´eric Jouault 和 Patrick Valduriez。“关于巨模型的必要性”。载于：*OOPSLA/GPCE*：模型驱动软件开发最佳实践研讨会论文集，第*19*届 *ACM* 面向对象编程、系统、语言与应用年度会议。2004 年。 URL: http://atlanmod.emn.fr/www/papers/OOPSLA04/bezivin-megamodel.pdf。[9] Jean B´ezivin 等人。“大规模建模与小规模建模”。载于：模型驱动架构。第 3599/2005 卷。计算机科学讲义（LNCS）。施普林格出版社，2005 年，第 33–46 页。 URL：http://dx.doi. org/10.1007/11538097_3。[10] Jean B´ezivin 等人。“大规模建模与小规模建模”。载于：模型驱动架构。Springer，2004 年，第 33–46 页。[11] Xavier Blanc 等人。“基于模型操作的模型不一致增量检测”。载于：*CAiSE '09*：第*21*届高级信息系统工程国际会议论文集，荷兰阿姆斯特丹。第 5565/2009 卷。柏林、海德堡：施普林格出版社，2009 年 8 月，第 32–46 页。

[12] Arnaud Blouin 等人。“Kompren：模型切片器的建模与生成”。载于：软件与系统建模14.1（2015 年），第 321–337 页。 URL：http: // dx.doi.org/10.1007/s10270-012-0300-x。[13] Dominique Blouin、Yvan Eustache 和 Jean-Philippe Diguet。“借助元模型子集与模型同步实现可扩展的全局模型管理”。载于：与 *ACM/IEEE* 第*17*届模型驱动工程、语言与系统国际会议同期举行的第二届建模语言全球化国际研讨会论文集，*GEMOC@Models 2014*，瓦伦西亚。2014 年，第 43–52 页。 URL：http: // ceur-ws.org/Vol-1236/paper-07.pdf。[14] Dominique Blouin 等人。“Kaolin：用于 FPGA 设计复用、升级与迁移的系统级 AADL 工具”。载于：*NASA/ESA* 自适应硬件与系统国际会议（*AHS*）。蒙特利尔，加拿大，2015 年 6 月。[15] Mark Boddy 等人。“FUSED：协同系统工程的工具集成框架”。载于：网络物理系统分析式虚拟集成研讨会。2011 年。[16] Hugo Bruneliere 等人。“模型视图方法的基于特征的综述”。载于：软件与系统建模 18.3（2019 年），第 1931–1952 页。[17] Hugo Bruneliere 等人。“EMF Views：用于集成异构模型的视图机制”。载于：概念建模——第*34*届国际会议，*ER 2015*，瑞典斯德哥尔摩，*2015*年*10*月*19*日至*22*日，论文集。2015 年，第 317–325 页。 URL：http://dx.doi.org/10.1007/978-3-319-25264-3_23。[18] Hugo Bruneliere 等人。“MoDisco：用于模型驱动逆向工程的通用可扩展框架”。载于：*IEEE/ACM*自动化软件工程国际会议论文集。2010 年。 DOI：10.1145/1858996.1859032。[19] Jordi Cabot 和 Ernest Teniente。“对象约束语言约束的增量求值”。载于：*CAiSE'06*：第*18*届高级信息系统工程国际会议，卢森堡，卢森堡。第 4001/2006 卷。计算机科学讲义（LNCS）。施普林格出版社，2006 年 5 月，第 81–95 页。 URL：http://dx.doi.org/10.1007/11767138_7。[20] *CoEST* 项目主页。http://www.coest.org/。[21] 复合 *EMF* 模型项目主页。http://www.mathematik. uni-marburg.de/~swt/compoemf/。[22] Csaba Debreceni 等人。“查询驱动的视图模型增量同步”。载于：第二届基于视图、面向方面与正交软件建模研讨会论文集。VAO '14。美国纽约州纽约市：ACM，2014 年，31:31–31:38。 URL：http://doi.acm.org/10.1145/ 2631675.2631677。

[23] Alexander Egyed. “UML 的即时一致性检查”。载于：*ICSE ’06*：第 *28* 届软件工程国际会议论文集。中国上海，2006 年 5 月 20 日至 28 日，第 381–390 页。[24] Hartmut Ehrig 等。《代数图变换基础》。EATCS。Springer，2006 年。DOI：10.1007/3-540-31188-2。[25] *EMF-IncQuery* 项目主页。https://www.eclipse.org/incquery/。[26] *EMF*：*Eclipse* 建模框架。https://www.eclipse.org/modeling/ emf/。最后访问于 2022 年 11 月 25 日。[27] *EMF Views* 项目主页。http://atlanmod.github.io/emfviews/。[28] *Epsilon* 项目主页。http://eclipse.org/epsilon/。[29] Anne Etien 等。“组合独立模型转换”。载于：*2010* 年 *ACM* 应用计算研讨会论文集。SAC ’10。美国纽约州纽约市：ACM，2010 年，第 2237–2243 页。URL：http://doi.acm.org/10.1145/1774088.1774557。[30] 扩展广义判别网络评估工件。https: //github.com/hpi-sam/Extended- GDNs。最后访问于 2022 年 11 月 25 日。[31] Jean-Marie Favre。“模型（驱动）（逆向）工程基础——第一集：Fidus Papyrus 与 Solarus 的故事”。载于：*Dagstuhl* 模型驱动逆向工程研讨会后续论文集。2004 年。[32] Jean-Marie Favre、Ralf L¨ammel 和 Andrei Varanovich。“软件产品语言架构建模”。载于：第 *15* 届模型驱动工程语言与系统国际会议论文集。MODELS’12。柏林，海德堡：施普林格出版社，2012 年，第 151–167 页。URL：http://dx.doi.org/10.1007/978-3-642-33666-9_11。[33] Franck Fleurey 等。“软件工程中的模型”。载于：Holger Giese 编。柏林，海德堡：施普林格出版社，2008 年。章节：一种通用的自动模型组合方法，第 7–15 页。URL：http://dx. doi.org/10.1007/978-3-540-69073-3_2。[34] Charles L Forgy。“Rete：用于多模式/多对象模式匹配问题的快速算法”。载于：人工智能与数据库读本。爱思唯尔，1989 年，第 547–559 页。[35] *FUSED* 项目主页。http://www.adventiumlabs.com/。[36] Erich Gamma 等。《设计模式：可复用面向对象软件的要素》。培生德国有限公司，1995 年。[37] *Gaspard2* 项目主页。http://www.lifl.fr/west/gaspard/。

[38] Holger Giese、Stefan Neumann 和 Stephan Hildebrandt。“模型同步实践：保持 SysML 与 AUTOSAR 模型的一致性”。载于：图变换与模型驱动工程——纪念 *Manfred Nagl 65* 岁诞辰文集。Gregor Engels 等编。第 5765 卷。计算机科学讲义。Springer Berlin / Heidelberg，2010 年，第 555–579 页。[39] Holger Giese 和 Robert Wagner。“从模型转换到增量双向模型同步”。载于：软件与系统建模 8.1（2009 年），第 21–43 页。[40] Holger Giese 等。“基于模型的集成”。载于：嵌入式实时系统的模型驱动工程——达格施图尔国际研讨会，德国达格施图尔城堡，*2007*年*11*月*4*日至*9*日。修订精选论文。Holger Giese 等编。第 6100 卷。计算机科学讲义。Springer，2011 年，第 17–54 页。 URL：http://www.springerlink.com/content/p44512248g5u6717/。[41] Joshua Gleitze、Heiko Klare 和 Erik Burger。“为模型变换网络寻找通用执行策略”。载于：软件工程基础方法国际会议。Springer，Cham。2021年，第 87–107 页。[42] *GME* 项目主页。http://www.isis.vanderbilt.edu/projects/ gme/。[43] Iris Groher、Alexander Reder 和 Alexander Egyed。“动态约束的增量一致性检查”。载于：软件工程基础方法。DavidS. Rosenblum 和 Gabriele Taentzer 编。第 6013 卷。计算机科学讲义。Springer Berlin Heidelberg，2010年，第 203–217 页。 URL：http://dx.doi.org/10.1007/978-3- 642-12029-9_15。[44] Annegret Habel 和 Karl-Heinz Pennemann。“相对于嵌套条件的高层变换系统正确性”。载于：计算机科学中的数学结构 19.2（2009 年），第 245–296 页。[45] Eric N Hanson、Sreenath Bodagala 和 Ullas Chadaga。“使用优化的判别网络进行触发条件测试和视图维护”。载于：*IEEE* 知识与数据工程汇刊 14.2（2002 年），第 261–280 页。[46] Regina Hebig、Andreas Seibel 和 Holger Giese。“论巨模型的统一”。载于：第 *4* 届多范式建模国际研讨会论文集（*MPM 2010*）。Vasco Amaral 等编。第 42 卷。EASST 电子通讯。2011年。 URL：http://journal.ub.tu-berlin.de/eceasst/article/view/704/713。[47] Reiko Heckel、Jochen Malte K¨uster 和 Gabriele Taentzer。“带类型属性图变换系统的合流性”。载于：图变换国际会议。Springer。2002年，第 161–176 页。

[48] Sebastian J.I. Herzig、Ahsan Qamar 和 Christiaan J.J. Paredis。“一种识别基于模型的系统工程中不一致的方法”。载于：*Procedia Computer Science* 28.0（2014）。2014年系统工程研究会议，第354–362页。 ISSN: 1877-0509。 DOI: http: //dx.doi.org/10.1016/j.procs.2014.03.044。 URL: http://www. sciencedirect.com/science/article/pii/S1877050914001070。[49] Sebastian JI Herzig 和 Christiaan JJ Paredis。“基于模型的贝叶斯推理”。载于：第*11*届模型驱动工程、验证与确认研讨会 *MoDeVVa 2014*。2014年，第69页。[50] Anders Hessellund 和 Andrzej Wasowski。“模型与元模型的接口和元接口”。载于：模型驱动工程语言与系统。Krzysztof Czarnecki 等编。第5301卷。计算机科学讲义。Springer Berlin Heidelberg，2008年，第401–415页。 URL: http://dx.doi.org/10.1007/978-3-540-87875-9_29。[51] Jendrik Johannes 和 Uwe Assmann。“基于关注点的模型驱动软件开发流程（解）组合”。载于：模型驱动工程语言与系统。DorinaC. Petriu、Nicolas Rouquette 和 Oystein Haugen 编。第6395卷。计算机科学讲义。10.1007/978-3-642-16129-2 5。Springer Berlin / Heidelberg，2010年，第47–62页。 URL: http : //dx. doi . org/10 . 1007/978 - 3 - 642 - 16129 - 2%5C_5。[52] Stefan Jurack 和 Gabriele Taentzer。“具有继承和包含结构的类型化图的组件概念”。载于：图变换——第*5*届国际会议，*ICGT 2010*，荷兰恩斯赫德，*2010*年*9*月*27*日至*10*月*2*日，论文集。2010年，第187–202页。 URL: http://dx.doi.org/10.1007/978-3-642-15928-2_13。[53] Heiko Klare 等。“在基于视图的系统开发中实现一致性——Vitruvius 方法”。载于：*Journal of Systems and Software* 171（2021年），第110815页。[54] *Kompose* 项目主页。http://www.kermeta.org/mdk/kompose。[55] *Kompren* 项目主页。http://people.irisa.fr/Arnaud.Blouin/ software_kompren.html。[56] Harald K¨onig 和 Zinovy Diskin。“相关模型的一致性高效检查”。载于：欧洲建模基础与应用会议。Springer。2017年，第161–178页。[57] D. Langsweirdt、N. Boucke 和 Yolande Berbers。“使用 ACOL 的嵌入式系统架构驱动开发”。载于：对象/组件/*S*

服务-

面向实时分布式计算研讨会（*ISORCW*），*2010*年第*13*届 *IEEE* 国际研讨会。2010年5月，第138–144页。[58] Henrik Lochmann 和 Anders Hessellund。“使用多种领域特定语言建模的集成视图”。载于：*IASTED* 国际软件工程会议（*SE 2009*）论文集。ACTA Press，2009年2月，第1–10页。

[59] Levi L´ucio 等。“FTG+PM：用于研究模型转换链的集成框架”。载于：*SDL 2013*：模型驱动的可靠性工程。Ferhat Khendek 等编。第7916卷。计算机科学讲义。Springer Berlin Heidelberg，2013年，第182–202页。URL：http://dx.doi.org/10.1007/978-3-642-38911-5_11.[60] *ModeSoC* 项目主页。http://www.reuseware.org/index.php/ Reuseware_ModelSoC。[61] *MoTE* 项目主页。http://www.mdelab.org/mdelab-projects/mote-a-tgg-based-model-transformation-engine/.[62] 对象管理组织。对象约束语言规范。https://www.omg.org/spec/OCL/。最后访问于 2022 年 11 月 25 日。[63] *OSLC* 项目主页。http://open-services.net/。[64] Richard F. Paige 等。“模型管理语言工程的概念框架与技术基础设施设计”。载于：复杂计算机系统工程，*2009*年第*14*届 *IEEE* 国际会议。2009年6月，第162–171页。[65] Cosmina Cristina Rat¸iu 等。“跨多领域工程模型的反应式链接”。载于：第*25*届模型驱动工程语言与系统国际会议论文集。2022年，第76–86页。[66] Jos´e E. Rivera 等。“编排 ATL 模型转换”。载于：*MtATL 2009* 论文集：第*1*届使用 *ATL* 进行模型转换国际研讨会。Fr´ed´eric Jouault 编。法国南特，2009年7月，第34–46页。URL：http : / / www . lcc . uma . es / ~av / Publicaciones / 09 / MtATL2009.pdf。[67] Andy Sch¨urr。“使用三元图语法规范图转换器”。载于：《 20^th 计算机科学中的图论概念国际研讨会。E. W. Mayr、G. Schmidt 和 G. Tinhofer 编。第903卷。计算机科学讲义。德国赫尔兴：Spinger Verlag，1994年6月，第151–163页。[68] Andreas Seibel、Regina Hebig 和 Holger Giese。“模型驱动工程中的可追溯性：高效且可扩展的可追溯性维护”。载于：软件与系统可追溯性。Jane Cleland-Huang、Orlena Gotel 和 Andrea Zisman 编。Springer London，2012年，第215–240页。[69] Andreas Seibel、Stefan Neumann 和 Holger Giese。“动态分层大型模型：全面的可追溯性及其高效维护”。载于：软件与系统建模 9.4（2010年），第493–528页。[70] Andreas Seibel、Stefan Neumann 和 Holger Giese。“动态分层大型模型：全面的可追溯性及其高效维护”。载于：《软件与系统建模》9.4（2010年），第493–528页。URL：http://dx.doi.org/10.1007/s10270-009-0146-z。

[71] Andreas Seibel 等。“一种专用语言，用于上下文组合和真正的黑盒模型转换的执行”。载于：第*4*届软件语言工程国际会议（*SLE 2011*），葡萄牙布拉加。2011年7月。[72] Gabor Simko 等。“模型集成的基础：语义骨干”。载于：美国机械工程师学会*2012*年国际设计工程技术会议暨计算机与工程信息会议。美国机械工程师学会。2012年，第1077–1086页。[73] *SmartEMF* 项目主页。http : / / www . itu . dk / ~hessellund / smartemf/.[74] Perdita Stevens。“将软件构建与模型间一致性的维护相结合：朝着基于巨模型的可靠、最优且灵活的构建迈进”。载于：《软件与系统建模》19.4（2020年），第935–958页。[75] Perdita Stevens。“维护模型网络中的一致性：大规模双向转换”。载于：《软件与系统建模》19.1（2020年），第39–65页。[76] Zoltan Ujhelyi 等。“EMF-IncQuery：用于实时模型查询的集成开发环境”。载于：《计算机编程科学》98，第1部分（2015年）。《实验性软件与工具包》第五期（EST）：专题“使用 Eclipse 进行学术建模”（ACME2012），第80–99页。[77] Hans Vangheluwe、Juan de Lara 和 Pieter J Mosterman。“多范式建模与仿真导论”。载于：*AIS2002* 会议论文集（*2002*年）。2002年。[78] D´aniel Varr´o 等。“迈向响应式增量模型转换平台：VIATRA 框架的三代发展”。载于：《软件与系统建模》15.3（2016年），第609–629页。[79] Andr´es Vignaga 等。“巨模型建模中的工件类型化”。载于：《软件与系统建模》12（2013年），第105–119页。URL：http://dx.doi.org/ 10.1007/s10270-011-0191-2。[80] *Wires* 项目主页。http :/ / atenea. lcc .uma . es/ index .php / Main_Page/Resources/Wires*。