

基于形式化方法的程序代码合并分析

何清^{1,2}, 蒋建民^{1,2†}

(1. 成都信息工程大学软件工程学院, 四川 成都 610225; 2. 软件自动生成与智能服务四川省重点实验室, 四川 成都 610225)

摘要:在实际软件开发中,开发人员通常会使用多个团队开发的程序来构建软件系统。然而,当这些程序包存在不同的版本时,将它们合并到一个完整的应用程序中可能会引发冲突,导致程序错误。提出了一种新的形式化模型——统一结构作为包图的形式语义,对程序代码包合并提出了一种新的分析方法。该方法给出了三种包合并的策略,深入研究了将不同版本的代码转换成的包图在合并过程中可能出现的冲突和不一致性问题,并提供了具体的解决方案。实验表明这个方法能确保合并后的代码正确性,从而提高软件开发的质量和生产效率。

关键词:代码合并; UML; 包图; 形式化; 包合并; 合并冲突

中图分类号:TP311

文献标识码:A

Formal Methods-based Program Code Merging Analysis

HE Qing^{1,2}, JIANG Jianmin^{1,2†}

(1. School of Software Engineering, Chengdu University of Information Technology, Chengdu, Sichuan 610225, China;

2. Sichuan Key Laboratory of Software Automatic Generation and Intelligent Service, Chengdu, Sichuan 610225, China)

Abstract: In practical software development, developers often use programs developed by multiple teams to build software systems. However, when these programs exist in different versions, merging them into a complete application may lead to conflicts, resulting in program errors. This article proposes a new formal model—unified structure as package diagram formal semantics, for program code package merging, presenting a novel analysis approach. This method offers three strategies for package merging, delving into potential conflicts and inconsistencies that may arise when merging package diagrams representing different versions of code, along with concrete solutions. Experiments demonstrate that this method can ensure the correctness of merged code, thereby enhancing the quality and productivity of software development.

Key words: code merging; UML; package diagram; formalization; package merging; merging conflicts

在现代软件工程中,随着软件系统的复杂性不断增加,代码合并成为一项关键任务。为了解决代码合并带来的挑战,研究人员和开发者们一直在寻找更好的方法和工具来简化合并过程,以减少冲突和提高代码集成的效率。有许多工具支持自动合并程序或模型的两个版本中两个非冲突段的合并,但这些工具在解决程序或模型的冲突方面提供的帮助很少,冲突解决在很大程度上依赖于手动干

预。此外,开发人员可能对他正在合并的冲突块缺乏足够的了解,然后根据自己的经验和对问题领域的一些知识进行猜测。如果这些猜测是错误的,可能会导致不必要的重复工作:识别合并问题,与团队成员协商以正确解决冲突,重新部署,重新回归测试等。如果合并错误且在部署之前未被识别,后果将更加灾难性。在这样的背景下,使用严格的形式化方法和 UML 包合并对代码合并进行分析具

收稿日期:2024-08-13

基金项目:国家重点研发计划资助项目(2022YFB3305101);国家自然科学基金资助项目(61772004);成都信息工程大学人才科研基金资助(KYTZ202009)

作者简介:何清(1998—),女,四川巴中人,硕士研究生,研究方向:形式化方法。

†通信联系人,E-mail: jjm@cuit.edu.cn

有重要意义,然而,已有的工作并没有使用过形式化方法对代码合并进行分析。

本文提出了一种新的模型——统一结构模型作为 UML 包图的形式语义,并且将形式化与包合并结合对代码合并提出了一种新的分析方法。基于形式化模型,可以对包图进行形式化的分析、验证和推理。

UML 包图提供了对软件架构的可视化表示,能够更好地描述系统的结构和组织^[1-3]。包图也是一种面向对象程序代码的图形表示方法,用于展示软件系统中的包之间的关系和组织,从而帮助开发人员理解和管理代码。在程序代码中,类通常被组织在包内,这种组织结构与 UML 中的包图和类图之间存在紧密的关系。

包合并(Package Merging)是指将两个或多个 UML 包合并为一个更大的包的过程。这种合并可以在系统设计和软件开发过程中发挥重要作用,特别是当存在多个子系统或模块需要整合时^[2]。通过将相关的包合并为一个较大的包,可以更好地表达系统的模块化和分层设计,减少不必要的复杂性。需要明确的是,合并后的包是一个逻辑上的组织结构,而不会生成新的包。当一个包合并另一个包时,包层面的元素是不变的。包合并实际上是对包内的元素进行合并。

本文使用统一结构对 UML 包图进行了严格的形式化定义和描述,探讨了三种合并策略的性质以及不同的合并策略对合并结果的影响。然后通过详细分析包合并中的冲突情况,提供了一个冲突解决框架,这个框架可以帮助软件开发团队更好地识别和解决潜在的代码合并冲突,提高代码合并过程的效率和质量。

1 相关工作

形式化方法已经成功应用于各种硬件设计,特别是芯片的设计。各大硬件制造商都有一个非常强大的形式化方法团队为保障系统的可靠性提供技术支持,例如 IBM、Intel、AMD 等等。由于软件系统的复杂性和不确定性远远超出硬件系统,形式化方法在软件开发中应用程度并不高。但是在最近 10 多年中,随着形式验证技术和工具的发展,特别是在程序验证中的成功应用,形式化方法在处理软件开发复杂性和提高软件可靠性方面已显示出无可取代的潜力^[4]。

Carré 等介绍了对系统进行形式化建模的方

法^[5],该论文使用包图来表示系统示例,并采用了一种与本文所使用的形式化方法不同的描述方式对子模型的特殊形式,即子元模型,以及它们在模型空间结构化中的特定角色进行了分析。

在需要进行模型合并的场景中,开发人员通常需要指定如何合并 UML 模型元素。Bischoff 等^[6]指出,可能有许多不同的合并设计模型的方式。然而,由于合并策略的日益多样化以及 UML 表达特定合并语义甚至模型合并顺序的局限性,指定如何合并输入模型是一项特别具有挑战性的任务。因此,开发人员最终无法表达如何合并输入模型的各个部分。

迄今为止,UML 缺乏帮助开发人员表达合并关系的结构。在过去的研究中,发现了 UML 内置的合并机制(即 UML 包合并)存在一系列模糊和缺失的规则^[7-8]。尽管一些 UML 扩展被提出来解决这些问题^[9],但它们仍未能对合并关系提供具体的语义参数化。

尽管今天有许多不同的合并工具在使用,但众所周知,在实践中它们并不完美。因为它们无法考虑到开发人员可能会进行的每一种可能的并发变更,因此会出现无法自动解决的并发变更之间的冲突,导致合并失败^[10]。开发人员必须介入,分析各自的变更及其引起的合并冲突,并手动解决冲突。这是一项困难的任务,开发人员希望尽量避免^[11]。然而,据报道合并操作有 10%~20% 的失败率^[12],有些项目的失败率甚至接近 50%^[13]。

2 形式化模型

本节给出用于包图建模的形式化模型“统一结构”,然后讨论该模型的一些性质。统一结构(Unified Structure)是我们研究团队在以前的形式化模型——依赖结构(Dependency Structure)^[14-15]的基础上开发出的新的基于结构的形式化模型,统一结构的表达能力和兼容性比依赖结构更强,目的是将 UML 模型进行统一的形式化建模,从结构的角度对 UML 模型进行分析。

定义 1:一个统一结构模型是一个元组 $US = \langle ME \leq, \omega^1, \dots, \omega^n, \lambda_m, \lambda_d, \tau^1, \dots, \tau^m \rangle$, 其中:

1) ME , 模型元素的有限集合。

2) $\subseteq$ $ME \times ME$, 表示包含关系。

3) $\forall i \in \{1, \dots, n\}, \omega^i \subseteq ME \times ME$, 表示依赖关系,它是一个非自反的偏序。

4) $\lambda_m \subseteq \text{ME} \times \text{ME}$, 表示模型元素的约束条件。

5) $\lambda_d \subseteq \text{ME} \times (\cup_{\omega^1} \cup \dots \cup \omega^n)$, 表示依赖关系的约束条件。

6) $\forall j \in \{1, \dots, m\}, \tau_j \subseteq \text{ME}$, 表示模型元素的类型集, 如: $\forall e \in \text{ME}, \exists \tau \in \{\tau^1, \dots, \tau^m\} : e \in \tau$ 。

对于所有的 $x, y \in \text{ME}, x \omega_i y (i \in \{1, \dots, n\})$, 称为一个依赖, 解释为 x 依赖于 y (注意, i 表示依赖关系的类型)。 $x < y$ 表示 x 包含在 y 中。如果 $x < z, y < z$, 它们被简单地 $x, y < z$ 用来表示, 表示 x 和 y 都包含在 z 中。对于所有 $w, v \in \text{ME}$, 符号 $v^\circ w$ 表示 w 不包含 v 。

元组 $\tau^1, \dots, \tau^m$ 是模型元素的分组结构, 用来对模型元素进行分类。例如在软件开发过程中的模型元素可以分为需求、设计、实现、测试和编码类型, 相对应的模型元素的类型集表示为需求集合、设计集合、实现集合、测试集合、编码集合。用统一结构表示为 $\tau^{\text{requirement}}, \tau^{\text{design}}, \tau^{\text{implementation}}, \tau^{\text{testing}}, \tau^{\text{code}}$ 。

根据定义 1 所定义的统一结构将图 1 表示为 $\text{PD} = \langle \text{ME}, <, \omega, \lambda, \tau \rangle$, 其中 $\text{ME} = \{A, B, C, D, E, F\}$, $< = \{(B, A), (E, D), (C, D)\}$, $\omega^{\text{import}} = \{(B, F), (B, C)\}, \lambda = \emptyset, \tau^{\text{PName}} = \{A, B, C, D, E, F\}$ 。这里解释一下其中符号的含义: ME 表示模型中所有元素的集合。如图 1 中的 A 表示一个包; $<$ 表示包含关系, C 和 D 之间的连线是包含关系的另一种表示方式, 表示 D 包含 C ; ω^{import} 表示一个包使用或依赖另一个包中的元素, 它包含泛化 (generalization)/存取 (access) 关系。通过 import 关系, 一个包可以引用另一个包中定义的类、接口、枚举等元素, 以便在自身内部使用。例如 $(B, F) \in \omega^{\text{import}}$ 表示 B 引用了 F (B 依赖于 F); λ 表示依赖关系的约束; τ^{PName} 表示包的名称。

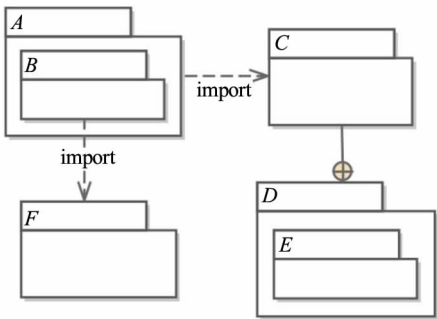


图 1 包图例子

3 包的合并

在生成所需的合并模型之前, 开发人员需要比较输入模型, 以便确定它们之间的共同点和差异。匹配策略定义了比较和对比输入模型的语义, 识别它们的重叠部分和差异。在匹配输入模型之后, 下一步是将它们合并在一起。为此, 可以使用合并策略。

基于匹配策略确定的输入模型之间的重叠点以及它们的差异, 合并策略定义了如何集成匹配部分。覆盖合并、一般合并和联合合并这三种合并策略被用于广泛的合并场景^[16], 包括设计模型的演化、本体合并和概念模型合并。接下来, 简要介绍这三种策略, 并假设 Ma 和 Mb 是输入模型。

(1) 覆盖合并

当包 A 使用覆盖合并策略合并包 B 时, 有如下两条规则: (1) 若 A 与 B 存在匹配的元素, 则只将 A 中的这些元素保留。 (2) B 中其他的元素直接复制到结果包中。覆盖合并满足幂等性和结合性, 但是不满足交换性。

(2) 一般合并

当包 A 使用一般合并策略合并包 B 时, 对于 A 和 B 中匹配的每一对元素, 这些元素应该进行合并。合并的方式取决于元素的类型, 即只有相同类型的模型元素可以进行合并。 A 和 B 中不相同的元素保持不变, 并直接插入到合并后的结果包中。一般合并满足幂等性和结合性。但是, 一般合并并不满足交换性。

(3) 联合合并

当包 A 使用联合合并策略合并包 B 时, 对于 A 和 B 中匹配的每一对元素, 它们的名称会被修改并插入到结果包中, 以保留它们的标识。这意味着在结果包中, 它们以不同的标识符或名称共存。同时, A 和 B 中没有匹配的部分将保持不变, 并直接插入到结果包中。

联合合并满足交换性和结合性, 不满足幂等性。在联合合并中, 重复执行合并操作会导致重复添加相同的元素, 使得结果不再一致。这是因为在每次联合合并时, 对于每一对对应的元素, 它们的名称会被修改并插入到合并后的包中, 以保留它们的标识。为了确保合并操作的正确性和一致性, 在选择合并策略时, 需要综合考虑各种因素, 并根据具体情况选择适合的策略。

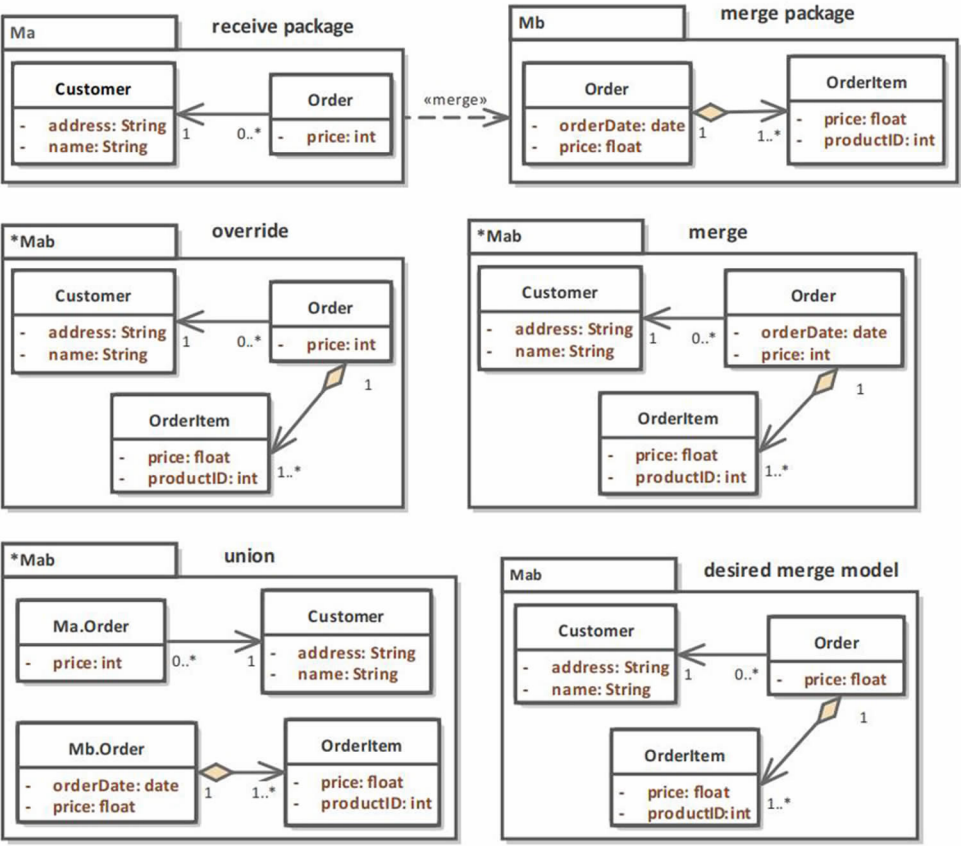


图 2 包合并示例

4 冲突和不一致

4.1 合并冲突

在 UML 中的包合并过程中,可能会出现一些冲突。如图 2 所示,在 Ma 中,Order 类的 price 属性被定义为整数类型(int),而在 Mb 中 Order 类的 price 属性为浮点数类型(float)。这两个属性类型的相互矛盾表示了一个需要开发人员解决的冲突。因此,开发人员需要回答一个关键问题:price 属性应该是整数还是浮点数?

根据图 2 所期望的合并结果 Mab,正确的答案是 Order 类的 price 属性应该是浮点数类型(float),即 Order, price = float(Order, price 表示 Order 中的属性 price)。然而,开发人员可能错误地将其分配为整数类型。这将导致不一致,即合并输出模型中的模型元素与开发人员所期望的合并模型之间存在分歧。

在包合并时,冲突存在于两个包之间的匹配的元素之间,如果没有相同名称的元素或没有需要比较的内容,就不会出现冲突。在包合并过程中,两个包之间可能会出现以下几种合并冲突类型:

- (1) 数据类型冲突:当合并两个包时,如果两个包中具有相同名称的属性或方法返回不同的数据类型,就会发生冲突。例如,一个包中的类 A 的属性 b 返回整数类型,而另一个包中的类 A 的属性 b 返回浮点数类型。
 - (2) 依赖关系冲突:如果合并的包中存在相同元素之间的不一致依赖关系,也会导致冲突。
 - (3) 可见性冲突:当合并两个包时,如果两个包中的元素的可见性不一致,冲突可能发生。可见性指定了元素(属性、方法、类等)对其他模块或类的可访问性。例如,一个包中的属性被定义为私有,而另一个包中的属性被定义为公共。
 - (4) 类型抽象性冲突:当合并两个包时,如果两个包中的类的抽象性不一致,就会发生冲突。例如,一个包中将类定义为抽象类,而另一个包中将其定义为具体类。
 - (5) 命名冲突:如果两个合并的包中存在相同名称的元素(例如类、接口、枚举等),则会出现名称冲突。这可能导致模型的不一致,因为相同名称的元素无法明确引用或使用。
- 统一结构的设计原则允许在后续的变更中对系统进行修改和扩展。因此,在描述合并冲突时,

引入了抽象性和可见性这两个集合,以满足新的需求和设计要求。这项变更旨在提高系统的灵活性和可扩展性,并确保以后的开发和维护能够更好地满足各种需求和变化。下面主要对部分类型的冲突进行讨论。

定义 2: 设 $PD = \langle ME, \prec, \overset{\text{import}}{\omega}, \overset{\text{PName}}{\lambda}, \tau \rangle$ 是一个包图, $M_A, M_B \in \tau$, 又设 $CD = \langle ME', \prec', \overset{\text{PName}}{\omega}, \overset{\text{CName}}{\lambda'}, \overset{\text{CProperty}}{\tau'}, \overset{\text{COperation}}{\tau'}, \overset{\text{DataType}}{\tau'} \rangle$ 和 $SD = \langle ME'', \prec'', \overset{\text{Associat}''}{\omega}, \overset{\text{Use}''}{\omega}, \overset{\text{Aggegat}''}{\omega}, \overset{\text{Composition}''}{\omega}, \overset{\text{Generalization}''}{\omega}, \overset{\text{Realization}''}{\omega}, \overset{\text{CName}}{\lambda''}, \overset{\text{CProperty}}{\lambda''}, \overset{\text{COperation}}{\lambda''}, \overset{\text{DataType}}{\lambda''} \rangle$ 分别是 M_A 和 M_B 中的类图。 $\overset{\text{visibility}}{\tau}$ 表示元素的可见性, $\overset{\text{visibility}}{\tau} = \{\text{public}, \text{private}, \text{protect}\}$ 。 $\overset{\text{Abs}}{\tau}$ 表示元素的抽象性, $\overset{\text{Abs}}{\tau} = \{\text{true}, \text{false}\}$ 。 M_A 和 M_B 两个包合并时的冲突表示为 $\text{Con}_{(M_A, M_B)}$ 。

(1) 如果 $m, n \in \tau \overset{\text{CName,}}{\cap} \tau'' \overset{\text{CName,}}{,} \alpha, \beta \in \tau \overset{\text{CProperty,}}{\cup} \tau'' \overset{\text{CProperty,}}{\cup} \tau \overset{\text{COperation,}}{\cup} \tau'' \overset{\text{COperation,}}{,} p, q \in \tau \overset{\text{DataType,}}{\cup} \tau'' \overset{\text{DataType,}}{,} ((p, \alpha), m) \in \lambda'_m \wedge ((q, \beta), n) \in \lambda''_m, m = n, \alpha = \beta, p \neq q$, 则 M_A 和 M_B 两个包合并时存在数据类型冲突, 表示为 $\text{Con}_{\langle M_A, M_B \rangle}(\tau \overset{\text{DataType,}}{\cup} \tau'' \overset{\text{DataType,}}{,})$ 。

(2) M_A 和 M_B 合并时的可见性冲突,表示为

$$\text{Con}_{(M_A, M_B)}(\overset{\text{visibility}}{\tau}, \overset{\text{visibility}}{\tau}'')。$$

(3) M_A 和 M_B 两个包合并时的抽象性冲突,表示为 $\text{Con}_{\langle M_A, M_B \rangle}(\tau^{\text{Abs}}, \tau^{\text{Abs}})$ 。

(4) 如果 $m, n, p, q \in \tau^{CName_1} \cap \tau^{CName_2}$, $(m_end: x, n_end: y) \in \lambda'_d$, $(p_end: z, q_end: c) \in \lambda''_d$, $x \neq z \vee y \neq c$ ($x, y, z, c \in N$), 则 M_A 和 M_B 两个包含并时存在依赖关系的约束冲突, 表示为 $Con_{M_A, M_B}(\lambda'_d, \lambda'_d)$ 。

(5) 如果 $m, n, p, q \in \tau \overset{\text{CName}}{\cap} \tau''$, $m = p$, $n = q$. $((m, n) \in \omega \overset{\text{Association}}{\quad}, (p, q) \in \omega'' \overset{\text{Aggregation}}{\quad}) \vee ((m, n) \in \omega \overset{\text{Aggregation}}{\quad}, (p, q) \in \omega'' \overset{\text{Association}}{\quad})$, 则 M_A 和 M_B 两个包含
 并时存在关联关系和聚合关系冲突, 表示为
 $\text{Con}_{(M_A, M_B)}(\omega \overset{\text{Association}}{\quad}, \omega'' \overset{\text{Aggregation}}{\quad})$.

(6) 如果 $m, n, p, q \in \tau' \cap \tau'', m = p, n = q, ((m, n) \in \omega' \xrightarrow{\text{Association}}, (p, q) \in \omega'') \vee ((m, n) \in \omega' \xrightarrow{\text{Composition}}, (p, q) \in \omega'')$, 则 M_A 和 M_B 两个包合并

时存在关联关系和组合关系冲突,表示为

$$\text{Con}_{\langle M_A, M_B \rangle} \left(\overset{\text{Association}}{\omega}, \overset{\text{Composition}}{\omega} \right).$$
$$(7) \text{ 如果 } m, n, p, q \in \tau \overset{\text{CName}}{\cap} \tau'', m = p, n = q. ((m, n) \in \omega \overset{\text{Aggregation}}{,}, (p, q) \in \omega'' \overset{\text{Composition}}{,}) \vee ((m, n) \in \omega \overset{\text{Composition}}{,}, (p, q) \in \omega'' \overset{\text{Aggregation}}{,}), \text{ 则 } M_A \text{ 和 } M_B \text{ 两个包合并时存在关联关系和组合关系冲突, 表示为 } \text{Con}_{\langle M_A, M_B \rangle}(\omega \overset{\text{Aggregation}}{,}, \omega'' \overset{\text{Composition}}{,}).$$

后面的四种冲突都是依赖关系冲突的具体细分,它们描述了不同类型的依赖关系冲突情况。如图 3 所示。

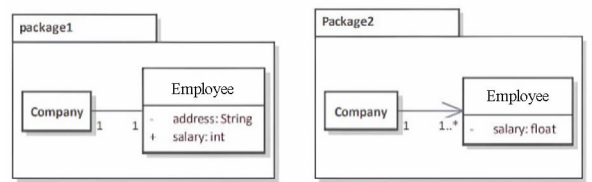


图 3 包合并冲突示例 1

(1) Package1 中 Employee 类的 salary 属性的数据类型为 int, Package2 中 Employee 类的 salary 属性的数据类型为 float, 当 Package1 和 Package2 合并的时候存在数据类型冲突。

$$\begin{array}{lll} \text{Package1::Employee.salary} & = & \text{int}, \\ \text{Package2::Employee.salary} & = & \text{float}, \\ \text{Con}_{(M_A, M_B)}(\tau^{\text{Data Type}_1}, \tau^{\text{Data Type}_2}) & = & (\text{int}, \text{float}). \end{array}$$

(2) Package1 中 Employee 类的 salary 属性的可见性为 public, Package2 中 Employee 类的 salary 属性的可见性为 private, 当 Package1 和 Package2 合并的时候存在可见性冲突。

$$\text{Package1}::\text{Employee. salary. visibility}=\text{public}, \text{Package2}::\text{Employee. salary. visibility}=\text{private}, \text{Con}_{(M_A, M_B)}(\tau^{\text{visibility}}, \tau^{\text{visibility}}) = (\text{public}, \text{private}).$$

(3) Package1 中 Employee 类不是抽象类, Package2 中 Employee 类是抽象类, 当 Package1 和 Package2 合并的时候存在抽象性冲突。

$$\begin{array}{l} \text{Package1} :: \text{Employee. isAbstract} = \text{false}, \\ \text{Package2} :: \text{Employee. isAbstract} = \text{true}, \\ \text{Con}_{\langle M_A, M_B \rangle} (\tau^{\text{Abs}}, \tau^{\text{Abs}}) = (\text{false}, \text{true}). \end{array}$$

在对依赖关系的冲突进行描述的时候,由于某些部分具有相同的表示方式,无法进行区分。因此引入了一个下标。该下标仅用于区分关系,对其他部分没有任何影响。

(4) 依赖关系的约束冲突: $Con_{(M_A, M_B)}(\lambda'_d, \lambda''_d) = (([Company_{end}: 1, Employee_{end}: 1], (Company, Employee)_{M_A}), ([Company_{end}: 1, Employee_{end}: 1 \cdots *], (Company, Employee)_{M_B}))$.

(5) 关联关系冲突: $Con_{(M_A, M_B)}(\overset{Association}{\omega}, \overset{Association}{\omega''}) = ((Employee, Complany), (Complany, Employee))_{M_{AAssociation}}, (Employee, Company)_{M_{BAssociation}}, (Company, Employee))$.

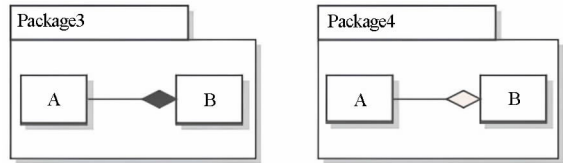


图 4 包合并冲突示例 2

(6)如图 4 所示:Package3 中类 A 和类 B 之间的关系为组合关系,Package4 中类 A 和类 B 之间的关系为聚合关系,当 Package3 和 Package4 合并的时候存在聚合关系和组合关系冲突。

$Con_{(M_A, M_B)}(\overset{Composition}{\omega}, \overset{Aggregation}{\omega''}) = ((A, B)_{M_{AComposition}}, (A, B)_{M_{BAggregation}})$.

4.2 冲突解决方案

一些冲突可以通过前文提出的三种合并策略来解决。然而,在这些合并策略中,对于依赖关系的冲突解决尚未明确处理。在本文的研究中,意识到了这个问题的重要性,并提出了一种解决方案,以解决这些冲突并维护系统的一致性。

在解决依赖关系的冲突时,遵循了关联、聚合和组合关系中约束关系的规则。当出现关系冲突时,选择约束关系最强的那个来解决冲突。这样做的目的是确保合并后的结构能够满足系统设计和需求的约束,并保持一致性。具体来说,如果合并的类之间存在关联关系,便选择具有更严格约束的关联类型。例如,如果一个类之间的关联关系被定义为双向关联和单向关联,并且冲突发生在关联的方向上,选择双向关联,因为它具有更强的约束性。同样地,对于聚合和组合关系,也选择具有更严格约束的关系类型。例如,如果一个类之间的关系被定义为聚合和组合,并且冲突发生在关系的强度上,选择组合关系,因为它具有更强的约束性。

对于聚合和关联之间的冲突,优先考虑具有更严格约束的关系类型。例如,如果一个类之间的关系既可以被视为聚合关系,又可以被视为关联关系,并且存在冲突时,选择聚合关系,因为它具有更

强的约束性。类似地,在组合和关联之间存在冲突的情况下,选择约束关系更强的那个。

例如在图 3 中,如果按照 Package1 覆盖合并 Package2 的顺序,可得到合并结果:Employee. salary=int,Employee. salary. visibility=public,Employee. isAbstract=false, $\overset{Association}{\omega} = ((Employee, Company), (Company, Employee))$ 。

合并冲突的处理确保了合并后的结构符合系统设计和需求的约束,从而提高系统的可维护性和可扩展性,并为后续的开发和维护提供了清晰的指导。虽然本文介绍了解决合并冲突的方案,但需要注意,由于每个项目和冲突场景都具有独特性,这些方案存在一定的不确定性和限制。此外,该解决方案是通过一系列实验和观察得出的,但由于合并冲突的复杂性和多样性,无法确保在所有情况下都能获得最佳效果。因此,需要在实际项目中进行进一步验证和调整,以适应具体的需求和限制。

4.3 不一致的发现与解决

在合并过程中,交换两个包的顺序可能会导致不同的合并结果。为了选择与预期结果最为一致的合并顺序和策略,对于不同顺序下采用不同策略的合并结果进行了比较。

在图 5 中展示了两个包的图示以及期望得到的合并结果包。随后,在图 6 中展示了使用不同合并顺序进行覆盖合并和联合合并的结果。从图中可以观察到,当 A 合并 B 时,覆盖合并和一般合并的结果都存在两个不一致:(1)属性 Employee. salary 的可见性是私有的,而不是公有的;(2)属性 Employee. salary 的数据类型是 int,而不是 float。当 B 合并 A 时,覆盖合并和一般合并所得到的结果相同,并且与期望的合并结果更为一致。因此,通过选择那些与预期结果不一致性较小的合并顺序和策略,能够更好地控制合并过程的结果。这种方法可以帮助开发人员更准确地合并模型,并减少合并过程中的不一致性。

开发人员需要能够明确指定应该应用哪种合并策略,以及在定义两个(或更多)合并关系时应该以何种顺序执行合并策略。如果不正确地表达合并语义和执行顺序,模型合并可能成为一个容易出错且耗时的任务。在这个阶段,接收包和合并包的模型元素可能会相互冲突,开发人员需要决定如何解决这些冲突。尽管冲突无法避免,但可以通过适当的解决方法来解决它们。通过解决冲突和不一致,并进行适当的调整和协商,可以确保在包合并

过程中实现模型的一致性和准确性。这有助于避免潜在的混淆和模糊,并促进模型的可理解性和可维护性。

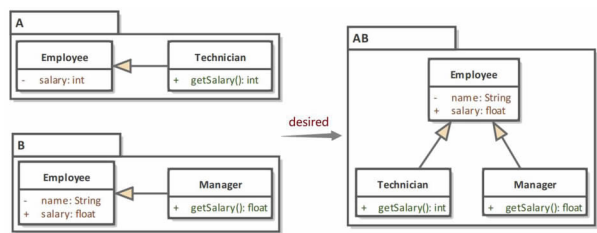


图 5 预期合并结果包示意图

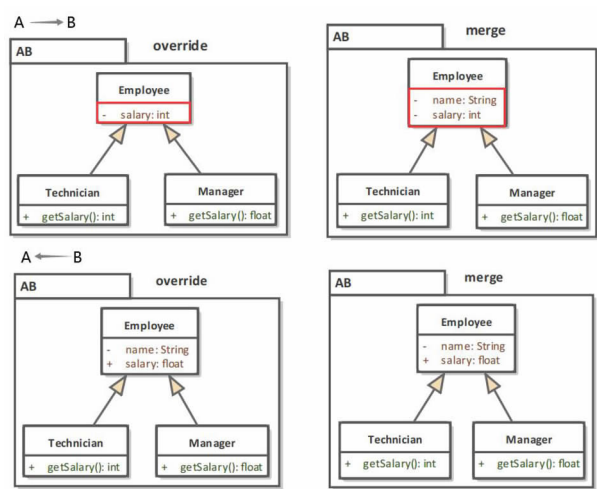


图 6 不同合并顺序的覆盖合并和一般合并结果

5 工具及实例研究

为了便于对包的合并进行分析,开发了绘图工具来绘制并将包图转化为形式化模型——统一结构。开发的原型工具是在使用 JavaScript 的 GoJS 图形库作为基础可视化操作界面,配合 jQuery 进行开发。JavaScript 是一种解释性的脚本语言,可以基于对象进行开发,不依赖于具体的操作系统,仅需要浏览器的支持。GoJS 是用于构建交互图表和图形的 JavaScript 和 TypeScript 库。图 7 是在工具中将 UML 图转换成统一结构的一个简单例子。

5.1 将代码转换成包图

将所有代码源文件生成的类图放置在一个包中。使用最外层的包作为标识来区分不同版本的开源项目生成的包图。因为一个项目通常包含大量的代码,将整个项目转换成图形表示可能会变得复杂和冗长。因此,为了简化分析过程,随机选择了部分代码进行转换,将其表示为包图。由于每个

开发人员的代码和合并需求可能不同,无法确定确切的需要合并或可能冲突的部分。因此,具体的项目转换策略应由开发人员根据实际情况自行确定。

为了进行实际操作,在 GitHub 上找了一个小型的公司管理项目(项目链接: <https://github.com/mingkong/JavaWeb>)。然后,将该项目的部分代码转换为可导入工具的 JSON 格式,并将其保存为一个 JSON 文件。接着,对这部分代码进行了一些修改,并生成了第二个 JSON 文件以表示修改后的版本。通过将这两个 JSON 文件导入工具中,可以得到两个不同版本的代码转换成的包图。这样,可以在包图上进行进一步的分析,以便检测、理解和解决可能存在的冲突。

5.2 冲突检测

将代码转化成包图之后,结合之前的形式化定义对生成的两个包图进行冲突检测。

从图 7 中可以看到包 Package1 和 Package2 之间一共有三处冲突: MimeTypeImpl 类的属性 _mimeType 存在可见性冲突; SecurityFilter 类的方法 doFilter() 存在可见性冲突,以及方法 destroy() 存在数据类型冲突; IndexActionBean 类和 BasicActionBean 类之间存在依赖关系冲突, Package1 中为 Compose 关系,而 Package2 中是 generalize 关系。上述实验结果表明工具检测出的冲突与图中存在的冲突一致。

本文的研究侧重于为代码合并提供合适的方案,实验旨在验证这一理论的可行性,而非进行对比实验。在实际代码合并中,该方法可以与版本控制系统(如 Git)结合使用,以提供更好的代码管理和合并流程。

6 结 论

采用统一结构模型作为 UML 包图的形式语义,系统地揭示了包合并的复杂性,并深入研究了包合并过程中可能出现的冲突和不一致性问题。通过形式化的包合并分析,将焦点放在了代码合并的实践层面,特别是包中类的合并方面。我们的研究发现,不同的合并策略可能导致多种潜在的合并结果,并且合并顺序对最终结果具有重要影响。实验也验证了提出的方法的正确性。这些发现提醒着研究者在包合并领域中需要综合考虑多样性和不确定性因素,以更好地解决代码合并过程中的挑战。

通过形式化的包合并分析对代码合并进行探

索,为软件开发领域提供了一种新的视角和方法。该方法旨在提高代码合并的准确性和效率,从而推动软件开发过程的质量和效率。在接下来的工作

中,我们将继续完善原型工具,使其投入实际的面向对象开发的系统建模中,为开发人员提供更好的支持和帮助。

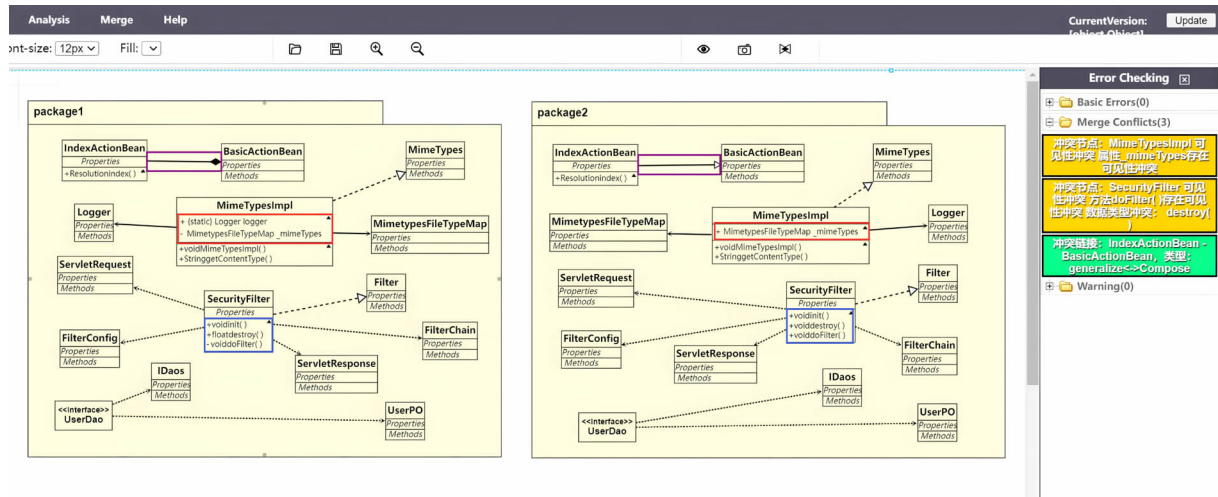


图 7 冲突检测

参考文献

[1] CHAUDRON M R V, HEIJSTEK W, NUGROHO A. How effective is UML modeling? [J]. Software and Systems Modeling, 2012, 11(4):571—580.

[2] HO-QUANG T, HEBIG R, ROBLES G, et al. Practices and perceptions of UML use in open source projects[C]. 2017 IEEE/ACM 39th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP). IEEE, 2017: 203—212.

[3] OMG: ‘UML: Infrastructure specification’, Version 2.5.1, <https://www.omg.org/spec/UML/2.5.1/PDF>, 2018.

[4] 王戟, 詹乃军, 冯新宇, 等. 形式化方法概貌[J]. 软件学报, 2019, 30(1): 33—61.

[5] CARRÉ B, VANWORMHOUDT G, CARON O. On submodels and submetamodels with their relation: a uniform formalization through inclusion properties[J]. Software and Systems Modeling, 2018, 17: 1105—1137.

[6] BISCHOFF V, FARIAS K, GONÇALES L J, et al. Integration of feature models: a systematic mapping study[J]. Information and Software Technology, 2019, 105: 209—225.

[7] DINGEL J, DISKIN Z, ZITO A. Understanding and improving UML package merge[J]. Software and Systems Modeling, 2008, 7: 443—467.

[8] KNAPP A, MOSSAKOWSKI T. Multi-view consistency in UML: a survey[J]. Graph Transformation, Specifications,

and Nets, 2018, 37—60.

[9] GUESSI M, OLIVEIRA L B R, NAKAGAWA E Y. Extensions of UML to model aspect-oriented software systems[J]. CLEI Electronic Journal, 2011,14(15): 1—18.

[10] MENS T. A state-of-the-art survey on software merging[J]. IEEE Transactions on Software Engineering, 2002, 28(5): 449—462.

[11] ROONEY G, BERLIN D. Practical Subversion[M]. Berkeley: Apress, 2007.

[12] BRUN Y, HOLMES R, ERNST M D, et al. Proactive detection of collaboration conflicts[C]. Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering, 2011: 168—178.

[13] ZIMMERMANN T. Mining workspace updates in CVS[C]. Fourth International Workshop on Mining Software Repositories (MSR’07: ICSE Workshops 2007). IEEE, 2007:11.

[14] JIANG J M, ZHU H, LI Q, et al. Isolation modeling and analysis based on mobility[J]. ACM Transactions on Software Engineering and Methodology, 2019, 28(2): 1—31.

[15] JIANG J M, ZHU H, LI Q, et al. Event-based functional decomposition[J]. Information and Computation, 2020, 271: 104484.

[16] GHIOTTO G, MURTA L, BARROS M, et al. On the nature of merge conflicts: a study of 2,731 open source java projects hosted by github[J]. IEEE Transactions on Software Engineering, 2018, 46(8): 892—915.