计算机系统应用 ISSN 1003-3254, CODEN CSAOBN
Computer Systems & Applications,2023,32(10):75−84 [doi: 10.15888/j.cnki.csa.009265]

E-mail: csa@iscas.ac.cn
http://www.c-s-a.org.cn
Tel: +86-10-62661041

基于 UML 活动图的用例识别①

陈卓然, 蒋建民, 唐国富, 谢嘉成, 陈华豪

(成都信息工程大学 软件工程学院, 成都 610225)
通信作者: 蒋建民, E-mail: jjm@cuit.edu.cn

摘　要: 在面向对象的软件开发过程中, 统一建模语言 (unified modeling language, UML) 的用例图用于捕获用户的需求. 传统描述用例的方法一般是开发者根据自己的经验, 从需求中人工获取用例. 然而, 如何自动生成准确的用例仍然是一个待解决的问题. 本文提出了一种通过用 UML 活动图半自动生成用例的方法. 首先通过引入形式化模型——统一结构来描述用例图与活动图, 其次给出分解活动图的算法, 然后根据分解活动图得到的依赖链生成对应的用例的事件流, 从而得到用例模型, 最后通过所开发的原型 CASE 工具进行案例的演示, 验证了本文所提出的方法的可行性

关键词: UML; 用例图; 活动图; 形式化模型; 用例识别

引用格式: 陈卓然,蒋建民,唐国富,谢嘉成,陈华豪.基于 UML 活动图的用例识别.计算机系统应用,2023,32(10):75–84. http://www.c-s-a.org.cn/1003-3254/9265.html

Use Case Identification Based on UML Activity Diagram

CHEN Zhuo-Ran, JIANG Jian-Min, TANG Guo-Fu, XIE Jia-Cheng, CHEN Hua-Hao

(School of Software Engineering, Chengdu University of Information Technology, Chengdu 610225, China)

Abstract: In the object-oriented software development process, use case diagrams of unified modeling language (UML) are applied to capture the user requirements. The traditional method of describing use cases is generally based on the developer's own experience to obtain use cases from the requirements manually. However, how to automatically generate accurate use cases is still a problem to be solved. This study proposes a method to generate use cases semi-automatically by using UML activity diagrams. Firstly, the study specifies the use case diagram and activity diagram by introducing a formal model, the unified structure. Secondly, it gives an algorithm for decomposing the activity diagram and then generates the event flow of the corresponding use cases, which is based on the dependency chain obtained from the decomposed activity diagram, to obtain the use case model. Finally, the case is demonstrated by the developed prototype CASE tool and the feasibility of the proposed method is verified.

Key words: unified modeling language (UML); use case diagram; activity diagram; formal model; use case identification

在面向对象的软件开发中, 统一建模语言 (unified modeling language, UML) 已经成为国际标准建模语言[1,2]. 用例图是在 UML 中一种重要的模型, 它用于捕获用户的功能需求, 而活动图也是在 UML 中的一种用于描述系统行为的模型视图, 它既可以用于描述业务过程, 也可以用于描述用例的事件流. 在实际的软件开发中, 活动图描述的业务过程通常对应到用例的事件流. 因此, 在真实的软件开发过程中可以通过业务流程来识别出不同用例的事件流, 从而进一步识别出用例[3].

用例图通常展示了最初一部分的用例模型, 它刻画了参与者与用例之间的关系, 并描述系统的整体功能需求. UML 创始人在他们的著作中提出, 用例是能

① 基金项目: 国家重点研发计划 (2022YFB3305104); 国家自然科学基金 (61772004); 成都信息工程大学人才科研基金 (KYTZ202009)
收稿时间: 2023-03-21; 修改时间: 2023-04-20, 2023-05-06; 采用时间: 2023-05-15; csa 在线出版时间: 2023-08-22
CNKI 网络首发时间: 2023-08-23

够向用户提供有价值的结果的系统中的一种功能, 它获取的是功能需求, 所有的用例加在一起组成用例模型, 应当构成目标系统所有的功能[3].

在面向对象的软件开发过程中, 用例模型还贯穿了软件开发的整个生命周期, 驱动软件开发的整个过程[3]. 用例模型是否正确直接决定了待开发系统的成败. 然而, 构建用例模型是非常复杂繁琐的工作, 用例模型是对业务流程的分析得出的结果, 它要考虑到参与者的识别、目标系统的行为、用例的粒度大小等多种因素, 一般的开发者难以正确识别出用例模型. 在捕获与识别用例的方面, 大部分开发者往往选择直接从用户的需求描述中根据自己软件开发的经验识别用例. 这样手工获取的用例难以完全表述用户的需要, 常常还会因为用户与开发者的交流不足问题, 产生用户原本不需要或与用户的期望偏差的功能, 导致识别出不正确的用例模型, 使开发出的系统不满足用户的需求.

自从 20 世纪 80 年代 Jacobson 提出用例图的概念以来, 很多研究者对用例图进行了深入的研究, 如用例的描述[4], 用例的迭代[5], 以及使用建模工具构造用例的工作[6], 但是这些工作都是手工完成的, 而没有采取半自动或自动识别用例的方法. 近年来, 一些研究者开始采用自然语言处理与深度学习技术[7–9] 提取需求文档中的需求, 但是, 这些工作都没有涉及到生成用例模型. 而在用例模型相关的研究中[10–13], 还没有直接从业务流程或业务建模中获取用例的方法. 因此, 本文提出了一种基于 UML 活动图的半自动用例识别方法. 首先, 通过对活动图进行形式化建模, 从业务流程方面整体描述系统功能; 其次, 识别出活动图中的依赖链, 对依赖链进行分割; 最后, 根据依赖链所对应的活动的执行顺序以及活动与活动所在的泳道之间的关系, 来生成对应的用例以及用例与参与者的关系.

本文描述的方法能够应用于软件开发的早期阶段, 协助建模人员建立用例模型. 由于用例模型来源于需求, 因此构造用例模型一般采用人工构造. 然而正确识别用例并不是一项容易的工作, 直接人工识别用例可能造成需求不明确的问题. 本文的方法通过活动图建模业务流程并分解, 可以快速、准确地识别用例. 本文的方法使用形式化方法对模型进行了建模, 能够保证模型的正确性与一致性, 降低后续维护的成本.

1 用例图与活动图

用例图把系统视为一个黑盒, 从参与者的角度看待系统, 非常适合用来做需求分析[14]. 需求分析作为软件开发至关重要的阶段, 常需要用例图的支持与驱动. 图 1 展示了一个简单的用例图.

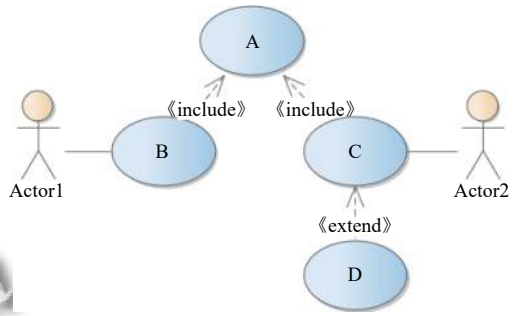

图 1 一个简单用例图

活动图广泛应用于业务流程建模, 着重表现从一个活动到另一个活动的流程. 在软件开发中, 活动图也用来表示整个业务的业务流程执行顺序. 活动图从需求分析阶段开始, 一直贯穿到软件测试阶段, 在软件开发的各个阶段都起着重要的作用.

图 2 展示了一个简单的活动图, 它描述了酒店预订业务的业务流程. 在第 2 节中, 本文将对图 2 的活动图进行形式化建模, 为之后的用例识别工作提供基础. 为了方便表示, 这里的节点都使用字母进行编号.

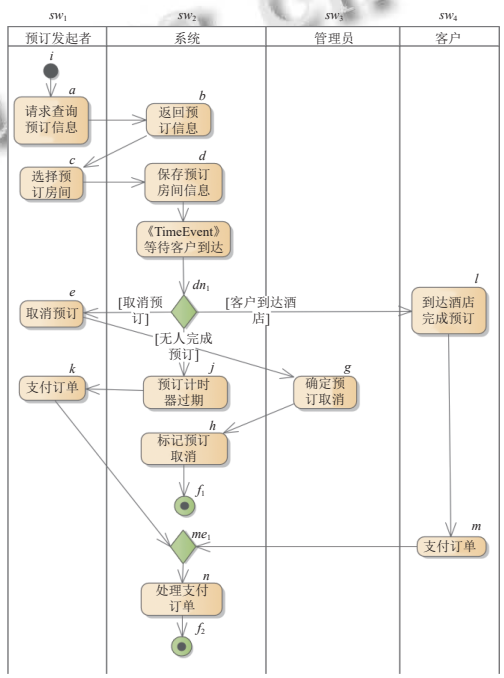

图 2 一个活动图

2 统一结构

本节给出用于形式化表达 UML 模型的形式化模型——统一结构 (unified structure). 统一结构基于团队先前的依赖结构 (dependency structure) 模型[15], 将 UML 模型进行统一的形式化建模, 从结构层面对 UML 模型进行分析与表示.

定义 1. 统一结构 (US) 是一个多元组$\langle ME, \prec, \overset{1}{\omega}, \cdots, \overset{n}{\omega}, \lambda, \overset{1}{\tau}, \cdots, \overset{m}{\tau}\rangle$, 其中:

1) ME为模型元素的有限集;

2) $\prec \subseteq ME \times ME$, 包含关系, 它是一个非自反的偏序;

3) $\forall i \in \{1, \cdots, n\}, \overset{i}{\omega} \in ME \times ME$为依赖关系;

4) $\lambda \subseteq ME \times (\prec \cup \overset{1}{\omega} \cup \cdots \cup \overset{n}{\omega})$为依赖上的限制;

5) $\forall j \in \{1, \cdots, m\}, \overset{j}{\tau} \in ME$为模型元素的类型集, 满足条件: $\forall e \in ME, \exists \tau \in \{\overset{1}{\tau}, \cdots, \overset{m}{\tau}\} : e \in \tau$.

模型元素在此处的概念与 UML 中的模型元素一致[5], 它可以根据视点的不同而有不同的类型. 包含关系是非自反的偏序, 用于建模父子关系. 依赖关系的种类与数量会随着模型的不同而变化. 包含与依赖关系都可以根据需要加入额外的限制条件. 显然, 根据定义 1, 可以用统一结构对用例图与活动图进行建模. 下面分别介绍如何用统一结构描述用例图与活动图.

图 1 展示了一个简单的用例图, 它可以用统一结构表示为 $UCD = \langle ME, \prec, \overset{\text{Sequence}}{\omega}, \overset{\text{Include}}{\omega}, \overset{\text{Extend}}{\omega}, \overset{\text{Associate}}{\omega}, \overset{\text{Generalize}}{\omega}, \lambda, \overset{\text{UseCase}}{\tau}, \overset{\text{Actor}}{\tau}, \overset{\text{Event}}{\tau}\rangle$, 其中:

1) $ME = \{A, B, C, D, Actor1, Actor2\}$

2) $\prec = \varnothing$

3) $\overset{\text{Sequence}}{\omega} = \varnothing$

4) $\overset{\text{Include}}{\omega} = \{(A, B), (A, C)\}$

5) $\overset{\text{Extend}}{\omega} = \{(C, D)\}$

6) $\overset{\text{Associate}}{\omega} = \{(B, Actor1), (D, Actor2)\}$

7) $\overset{\text{Genaralize}}{\omega} = \varnothing$

8) $\lambda = \varnothing$

9) $\overset{\text{UseCase}}{\tau} = \{A, B, C, D\}$

10) $\overset{\text{Actor}}{\tau} = \{Actor1, Actor2\}$

11) $\overset{\text{Event}}{\tau} = \varnothing$

符号分别有如下含义: ME表示模型中所有元素的集合, 如A代表用例A, $Actor1$代表参与者$Actor1$. $\prec$代表用例与事件的包含关系, 由于例子没有使用事件流, 因此此处为空. $\overset{\text{Sequence}}{\omega}$表示事件流的执行顺序, 此处为空. $\overset{\text{Include}}{\omega}$表示用例之间的包含关系, 如$(A, B)$表示$B$包含$A$. $\overset{\text{Extend}}{\omega}$表示用例之间的扩展关系, 如$(C, D)$表示$D$扩展$C$. $\overset{\text{Associate}}{\omega}$表示用例与参与者之间的关联关系, 如$(B, Actor1)$表示用例$B$与参与者$Actor1$之间存在关联关系. $\overset{\text{Generalize}}{\omega}$表示用例与用例, 或参与者与参与者之间的泛化关系, 此处没有体现, 因此为空. $\overset{\text{UseCase}}{\tau}$表示用例集合, $\overset{\text{Actor}}{\tau}$是参与者集合, $\overset{\text{Event}}{\tau}$表示用例的事件流的集合, 此处没有描述事件流, 因此为空.

图 2 的活动图用统一结构表示为 $AD = \langle ME, \prec, \overset{\text{Sequence}}{\omega}, \lambda, \overset{\text{InitialAct}}{\tau}, \overset{\text{FinalAct}}{\tau}, \overset{\text{Activity}}{\tau}, \overset{\text{Fork}}{\tau}, \overset{\text{Join}}{\tau}, \overset{\text{Decision}}{\tau}, \overset{\text{Merge}}{\tau}, \overset{\text{Swimlane}}{\tau}\rangle$, 其中:

1) $ME = \{a, b, c, d, e, g, h, j, k, l, m, n, i, f_1, f_2, te_1, dn_1, me_1\}$

2) $\prec = \{(i, sw_1), (a, sw_1), (c, sw_1), (e, sw_1), (k, sw_1), (b, sw_2), (d, sw_2), (te_1, sw_2), (dn_1, sw_2), (j, sw_2), (h, sw_2), (f_1, sw_2), (me_1, sw_2), (n, sw_2), (f_2, sw_2), (g, sw_3), (l, sw_4), (m, sw_4)\}$

3) $\overset{\text{Sequence}}{\omega} = \{(a, i), (b, a), (c, b), (d, c), (te_1, d), (dn_1, te_1), (e, dn_1), (g, e), (h, g), (f_1, h), (j, dn_1), (k, j), (me_1, k), (l, dn_1), (m, l), (me_1, m), (n, me_1), (f_2, n)\}$

4) $\lambda = \varnothing$

5) $\overset{\text{InitialAct}}{\tau} = \{i\}$

6) $\overset{\text{FinalAct}}{\tau} = \{f_1, f_2\}$

7) $\overset{\text{Activity}}{\tau} = \{a, b, c, d, e, g, h, j, k, l, m, n, i, f_1, f_2, te_1, dn_1, me_1\}$

8) $\overset{\text{Fork}}{\tau} = \varnothing$

9) $\overset{\text{Join}}{\tau} = \varnothing$

10) $\overset{\text{Decision}}{\tau} = \{dn_1\}$

11) $\overset{\text{Merge}}{\tau} = \{me_1\}$

12) $\overset{\text{Swimlane}}{\tau} = \{sw_1, sw_2, sw_3, sw_4\}$

此处解释符号的含义. ME表示模型元素中所有元素的集合, 如a表示一个活动节点, i表示一个初始化节点; $\prec$表示节点之间的包含关系, 如(i, sw_1)表示节点i归属于节点sw_1; $\overset{\text{Sequence}}{\omega}$表示节点之间的执行顺序, 如$(a, i)$表示执行顺序为先$i$后$a$; $\overset{\text{InitialAct}}{\tau}$表示初始化节点集合; $\overset{\text{FinalAct}}{\tau}$表示结束节点集合; $\overset{\text{Activity}}{\tau}$表示活动节点集合; $\overset{\text{Fork}}{\tau}$表示分支节点集合, 此处没有使用分支节点, 因此集合内没有元素; $\overset{\text{Join}}{\tau}$表示汇聚节点集合, 此处没有使用汇聚节点, 因此集合内没有元素; $\overset{\text{Decision}}{\tau}$表示决策节点集合; $\overset{\text{Merge}}{\tau}$表示合并节点集合; $\overset{\text{Swimlane}}{\tau}$表示泳道节点集合.

定义 2. 令$US=\langle ME,\prec,\overset{1}{\omega},\cdots,\overset{n}{\omega},\lambda,\overset{1}{\tau},\cdots,\overset{m}{\tau}\rangle$为一个统一结构.

如果序列$dc=x_1\cdots x_n$, $\forall i\in\{1,\cdots,n-1\}, x_i,x_{i+1}\in ME$, $(x_i,x_{i+1})\in(\overset{1}{\omega}\cup\cdots\cup\overset{n}{\omega})$, 则称$dc$为$US$的一个依赖链.

$\hat{dc}$表示为依赖链dc中的模型元素, 也就是说, $\hat{dc}=\{x_1\cdots x_n\}$, $DC(US)$表示US中所有可能的依赖链. $[dc]$表示依赖链$dc=x_1\cdots x_n$中模型元素的个数, 也就是说, $[dc]=n$.

在图 2 所示的活动图中, $iabcte_1dn_1eghf_1$是一条依赖链.

性质 1. 若$US=\langle ME,\prec,\overset{1}{\omega},\cdots,\overset{n}{\omega},\lambda,\overset{1}{\tau},\cdots,\overset{m}{\tau}\rangle$为一个统一结构, 则$ME=\overset{1}{\tau}\cup\cdots\cup\overset{n}{\tau}$.

证明: 由定义 1 直接可得.

性质 2. 令$US=\langle ME,\prec,\overset{1}{\omega},\cdots,\overset{n}{\omega},\lambda,\overset{1}{\tau},\cdots,\overset{m}{\tau}\rangle$为一个统一结构, 并且$dc=x_1\cdots x_n\in DC(US)$. 如果$\forall i\in\{1,\cdots,n-1\},(x_i,x_{i+1})\in\prec$, 那么在$dc$中没有环.

证明: 由于依赖链dc中只包含了包含关系, 根据定义 1 与定义 2, 包含关系$\prec$是一个非自反的偏序, 满足该偏序关系的元素无法关联到自身. 因此dc中不存在环.

性质 2 说明了只包含了包含关系的依赖链是没有环的.

3 基于活动图的用例的识别

本节介绍使用统一结构建模的活动图进行用例识别的方法.

定义 3. 令$AD=\langle ME,\prec,\overset{Sequence}{\omega},\lambda,\overset{InitialAct}{\tau},\overset{FinalAct}{\tau},\overset{Activity}{\tau},\overset{Fork}{\tau},\overset{Join}{\tau},\overset{Decision}{\tau},\overset{Merge}{\tau},\overset{Swimlane}{\tau}\rangle$为一个活动图, $DC(AD)$是AD中所有可能的依赖链. 如果依赖链$dc=x_1\cdots x_n\in DC(AD)$且$x_1\in\overset{InitialAct}{\tau}$, $x_n\in\overset{FinalAct}{\tau}$, 则称$dc$是完全的. 这里定义$CDC(AD)$表示活动图$AD$所有的完全的依赖链.

在图 2 的活动图中, 依赖链$iabcte_1dn_1eghf_1$. 的首个元素i属于$\overset{InitialAct}{\tau}$, 末尾的元素$f_1$属于$\overset{FinalAct}{\tau}$, 首尾之间的元素全部属于$ME$, 因此它是一个完全的依赖链.

命题 1. 令$AD=\langle ME,\prec,\overset{Sequence}{\omega},\lambda,\overset{InitialAct}{\tau},\overset{FinalAct}{\tau},\overset{Activity}{\tau},\overset{Fork}{\tau},\overset{Join}{\tau},\overset{Decision}{\tau},\overset{Merge}{\tau},\overset{Swimlane}{\tau}\rangle$为一个活动图, $CDC(AD)$是AD中所有可能的完全的依赖链. 如果$\forall dc=x_1\cdots x_n\in CDC(AD)$, 则$\exists x_i,x_{i+1}\in\{x_1,\cdots,x_n\}:(x_i,x_{i+1})\in\overset{Sequence}{\omega}$.

证明: $\forall dc=x_1\cdots x_n\in CDC(AD)$, 即活动图中所有的依赖链都是完全的依赖链, 每个依赖链中的每两个相邻元素组成的元素对都能在$\overset{Sequence}{\omega}$中找到, 即$\exists x_i$, $x_{i+1}\in\{x_1,\cdots,x_n\}:(x_i,x_{i+1})\in\overset{Sequence}{\omega}$, 命题得证.

命题 1 表明了一个活动图可以分解为多个独立的依赖链. 图 2 的活动图能够被分解为依赖链$iabcte_1dn_1eghf_1$与依赖链$iabcdte_1dn_1jkme_1nf_2$, 分解原始活动图得到的两个依赖链都是完全的依赖链.

定义 4. 令$AD=\langle ME,\prec,\overset{Sequence}{\omega},\lambda,\overset{InitialAct}{\tau},\overset{FinalAct}{\tau},\overset{Activity}{\tau},\overset{Fork}{\tau},\overset{Join}{\tau},\overset{Decision}{\tau},\overset{Merge}{\tau},\overset{Swimlane}{\tau}\rangle$为一个活动图, 并假定$\overset{TimeEvent}{\tau}\subseteq\overset{Activity}{\tau}$是时间事件. 如果一个链$dc=x_1\ldots x_n\in CDC(AD)$, 并且$\exists x\in\overset{TimeEvent}{\tau}\cap\hat{dc}$, 则称$dc$是可分解的.

定义 4 引入了时间事件的概念. 因为一些业务上的操作需要等待其他业务相关的执行者或用户完成其他操作才能继续进行, 所以将这些等待其他执行者或用户的操作描述为时间事件. 以这一类事件为分界, 将一个完整但较长的业务流程分解成多个独立的业务流程.

满足定义 4 的依赖链能够被分解为多个依赖链. 如果依赖链的性质不能满足定义 4, 说明依赖链所对应的业务流程在实际执行中不需要等待其他业务执行者完成操作, 这样的依赖链能够直接被识别为一个用例.

算法 1. 活动图生成已分解的依赖链的集合

输入: 活动图$AD=\langle ME,\prec,\overset{Sequence}{\omega},\lambda,\overset{Swimlane}{\tau},\overset{InitialAct}{\tau},\overset{FinalAct}{\tau},\overset{Activity}{\tau},\overset{Fork}{\tau},\overset{Join}{\tau},\overset{Decision}{\tau},\overset{Merge}{\tau},\rangle$

输出: 所有已分解的依赖链的集合 *DDC*

1) *CDC*←∅ //初始化完全的依赖链的集合
2) *DDC*←∅ //初始化已分解的依赖链的集合
3) *rc*←∅ // 初始化依赖链
4) Initialize *pathStack* as a stack //初始化路径栈
5) *InitialNodesSet*←{$e\in ME|\nexists x\in ME:(x,e)\in\overset{Sequence}{\omega}$} //找出起始节点
6) **FOR** each *i* in *InitNodesSet* **DO**
7) *pathStack*.push(*i*)
8) **IF** $\exists x\in ME:(x,i)\in\overset{Sequence}{\omega}$ **THEN**
9) *DFS*(*x*) //深度优先搜索
10) **END IF**
11) *pathStack*.pop()
12) **END FOR**
13) **FOR** each *dc* in *CDC* **DO**
14) **FOR** each *x* in *dc* **DO**
15) **SWITCH** the set where *x* in **DO** /* 根据 *x* 所属的集合做出不同的行为 */
16) **CASE** $\overset{TimeEvent}{\tau}$

```
17)      IF τ(TimeEvent) ⊆ τ(Activity) THEN /* 如果元素表示时间事件, 那么将时间事件之前的链存入 DDC 中, 之后直接遍历下一个元素. */
18)        DDC←DDC∪{rc}
19)        rc←∅ // 清空依赖链, 重新开始记录
20)      END IF
21)    CASE τ(FinalAct) /* 如果元素表示终止节点, 则将依赖链存入 DDC 中 */
22)        rc←rc∪{x}
23)        DDC←DDC∪{rc}
24)        rc←∅ // 清空依赖链, 重新开始记录
25)    OTHERWISE rc←rc∪{x} /* 将元素加入依赖链 */
26)    END
27)  END FOR
28) END FOR
29) PROCEDURE DFS(x) //深度优先搜索算法
30)    IF x∈τ(FinalAct) or x==NULL THEN /*找到终止节点, 或是为 NULL 的节点 */
31)      IF x∈τ(FinalAct) pathStack.push(i) /* 只在 x 不为 NULL 时将节点入栈*/
32)      END IF
33)      CDC←CDC∪pathStack.path /*将完整路径加入路径集合*/
34)      IF x∈τ(FinalAct) pathStack.pop() /*只在 x 不为 NULL 时将节点出栈 */
35)      END IF
36)    ELSE //找到其他节点
37)      IF !x.isVisited THEN
38)        x.isVisited ← true //遍历节点
39)        pathStack.push(x)
40)          IF ∃e∈ME:(e,x)∈ω(Sequence) THEN
41)            FOR each e in{e∈ME|(x,e) ∈ω(Sequence)}
42)              DFS(x) //递归深度优先
43)            END FOR
44)          END IF
45)        pathStack.pop()
46)      END IF
47)    END IF
48) END PROCEDURE
```

算法 1 的设计思路如下: 在构建活动图, 并使用统一结构对它进行描述后, 将这个统一结构作为算法的输入. 遍历整个活动图, 识别出活动图的依赖链. 在识别依赖链后, 对其中含有时间事件的依赖链进行分解, 并将分解的依赖链的集合作为算法的输出. 整个算法基于深度优先搜索, 在此基础上进行扩展.

算法 1 给出了输出依赖链的详细流程.

1) 输入活动图, 并获得活动图中的所有初始节点, 作为深度优先搜索的起点集合.

2) 将起始节点入栈.

3) 从起始节点开始, 深度优先搜索后继节点.

4) 在深度优先搜索结束之后, 弹出栈顶, 继续从其他起始节点开始搜索, 直到没有起始节点可以搜索.

5) 在搜索完完全的依赖链之后, 遍历所有完全的依赖链, 删除其中的时间事件.

我们修改了一部分深度优先搜索算法来适配需求.

1) 深度优先搜索算法首先接收一个节点作为搜索起点.

2) 如果搜索到了终止节点, 说明这一条路径已经搜索完毕, 将路径存入CDC中.

3) 对其他遍历到的节点, 如果它们没有被遍历过, 就将它们标记为已访问. 如果遍历到一个已经访问过的节点, 就回到分支处继续后续搜索.

在生成CDC之后, 将其中的所有可分解的依赖链dc分解.

1) 遍历依赖链dc, 将遍历到的每一个元素都保存在一个链中.

2) 遍历到时间事件时, 将链保存在DDC中, 然后, 不保存时间事件元素, 直接清空链, 从下一个节点继续遍历.

3) 遍历到结束节点时, 保存结束节点元素, 然后将链保存在DDC中, 最后清空链.

通过算法 1 分解依赖链的操作, 原本的依赖链会被分割成两部分. 将两部分依赖链各自作为独立的元素保存在了集合DDC中. $DDC(AD)$表示活动图AD中所有完全的依赖链分解之后的依赖链的集合. 如果依赖链没有时间事件, 那么依赖链本身也会存入DDC中.

定义 5. 令$AD=\langle ME,\prec,{}^{Sequence}\omega,\lambda,{}^{InitialAct}\tau,{}^{FinalAct}\tau,{}^{Activity}\tau,{}^{Fork}\tau,{}^{Join}\tau,{}^{Decision}\tau,{}^{Merge}\tau,{}^{Swimlane}\tau\rangle$为一个活动图. $\exists dc\in DDC(AD)$, 对$dc=x_1\cdots x_n$,

1) 如果$x_1\notin{}^{InitialAct}\tau$, 则称$dc$是缺首的依赖链.

2) 如果$x_n\notin{}^{FinalAct}\tau$, 则称$dc$是缺尾的依赖链.

3) 如果$x_1\notin{}^{InitialAct}\tau$且$x_n\notin{}^{FinalAct}\tau$, 则称$dc$是缺首尾的依赖链.

所有缺首的依赖链、缺尾的依赖链、缺首尾的依赖链统称为“不完整的依赖链”.

任何满足了定义 4 的可分解的依赖链都可以在分解之后形成定义 5 中所述的不完整的依赖链. 不完整的依赖链可以通过补充节点重新恢复成完全的依赖链.

定义 6. 令$AD=\langle ME,\prec,{}^{Sequence}\omega,\lambda,{}^{InitialAct}\tau,{}^{FinalAct}\tau,{}^{Activity}\tau,{}^{Fork}\tau,{}^{Join}\tau,{}^{Decision}\tau,{}^{Merge}\tau,{}^{Swimlane}\tau\rangle$为一个活动图, 并

设x_0, x_f分别是活动图的起始和终止节点. 如果$dc = x_1 \cdots x_n \in DDC(AD)$且$dc$是不完整的依赖链. 令$rc$为修复的依赖链. 根据$dc$所属的类型,

1) 如果dc是缺首的依赖链, 则$rc = x_0 x_1 \cdots x_n$.

2) 如果dc是缺尾的依赖链, 则$rc = x_1 \cdots x_n x_f$.

3) 如果dc是缺首尾的依赖链, 则$rc = x_0 x_1 \cdots x_n x_f$.

定义 6 的操作改变了活动图原本的依赖关系, 因此这个过程是单向不可逆的.

性质 3. 设 AD 是一个活动图, 所有已修复的依赖链 是完全的依赖链, 且不含有时间事件.

证明: 根据定义 5, 加入的依赖链中不会再含有时间事件. 根据定义 6 的操作, 修复的依赖链重新拥有了起始节点与终止节点, 满足定义 3 中完全的依赖链的性质, 该性质得证.

性质 3 说明了修复的依赖链具备完全的依赖链的性质, 并且已经无法再分解.

性质 4. 设 AD 是一个活动图, 如果$\forall dc \in DDC(AD)$, 且dc是完全的, 则$DDC(AD) = CDC(AD)$.

证明: 该证明可直接得到.

所有满足了性质 4 的依赖链集合所对应的活动图都不能再分解, 可以直接生成用例.

已经完成修复的依赖链可以描述一个完整的业务流程. 在统一开发过程[3] 中, 一个用例应当只描述系统的某一个功能, 而它的事件流描述了系统与用例相关的参与者之间的交互, 用例的事件流也可以反映为一个活动图. 完全的依赖链与修复的依赖链都具备了与用例相同的性质: 它们都描述了系统的某一个功能, 并且能够通过活动与泳道的关系来进一步描述用例与参与者的关系. 通过一个完整的依赖链能够生成它对应的用例的事件流, 进而描述一个完整的用例.

如果需要从活动图中识别用例, 那么可以采用以下几个步骤.

1) 将活动图建模为统一结构AD.

2) 从活动图的每一个起始节点开始, 搜索活动图中的路径, 并在搜索到终止节点时保存对应的路径为完全的依赖链.

3) 在搜索完整个活动图后, 寻找路径中的时间事件, 将时间事件分解为起始节点与终止节点, 并补全依赖关系, 即修复成完全的依赖链.

4) 每个完全的依赖链即为一个用例的事件流, 依赖链中的活动所属的泳道描述为参与者, 最后手动补全用例名与用例描述.

定义 7. 令 $AD = \langle ME, \prec, \overset{\text{Sequence}}{\omega}, \lambda, \overset{\text{InitialAct}}{\tau}, \overset{\text{FinalAct}}{\tau}, \overset{\text{Activity}}{\tau}, \overset{\text{Fork}}{\tau}, \overset{\text{Join}}{\tau}, \overset{\text{Decision}}{\tau}, \overset{\text{Merge}}{\tau}, \overset{\text{Swimlane}}{\tau} \rangle$为一个活动图, $UCD = \langle ME', \prec', \overset{\text{Sequence}'}{\omega}, \overset{\text{Include}}{\omega}, \overset{\text{Extend}}{\omega}, \overset{\text{Associate}}{\omega}, \overset{\text{Generalize}}{\omega}, \lambda', \overset{\text{UseCase}}{\tau}, \overset{\text{Actor}}{\tau}, \overset{\text{Event}}{\tau} \rangle$为一个用例图. 假定在$CDC(AD)$中有 n 个完全的依赖链 (包括修复的依赖链), 它们的名字分别对应为$N_1, N_2, \cdots, N_n$, 且活动图中所有的依赖链都不能再分解, 那么活动图与用例图有如下关系.

1) $\overset{\text{Swimlane}}{\tau} \cup \overset{\text{Activity}}{\tau} \cup \{N_1, N_2, \cdots, N_n\} \subseteq ME'$

2) $\prec' = \{(a,b) | a \in b, b \in \{N_1, N_2, \cdots, N_n\}\}$

3) $\lambda' = \lambda$

4) $\overset{\text{Sequence}'}{\omega} = \{(a,b) | a \in \overset{\text{Activity}}{\tau}, b \in \overset{\text{Activity}}{\tau}, (a,b) \in \overset{\text{Sequence}}{\omega}\}$

5) $\overset{\text{Associate}}{\omega} = \{(a,b) | dc \in CDC(AD), a \in \hat{dc}, b \in \overset{\text{Swimlane}}{\tau}, (a,b) \in \prec\}$

6) $\overset{\text{UseCase}}{\tau} = \{N_1, N_2, \cdots, N_n\}$

7) $\overset{\text{Actor}}{\tau} = \{Actor | Actor \in \overset{\text{Swimlane}}{\tau}\}$

8) $\overset{\text{Event}}{\tau} = \{e | e \in \overset{\text{Activity}}{\tau}\}$

定义 7 给出了活动图生成用例图时的对应关系. 用例图中所有的元素由泳道对应的参与者与依赖链对应的用例的事件流构成; 用例包含它自己的事件流, 从而形成统一结构上的包含关系, 此关系在活动图中由依赖链及其所属的活动构成; 用例图节点之间的关系对应活动图中泳道与活动的关系; 用例图中的用例对应活动图中的依赖链; 用例图的参与者对应活动图的泳道; 用例的事件对应依赖链所属的活动.

借助定义 7 中的关系, 图 3 中的所有依赖链都能够识别出对应的用例. 因为用例描述的是系统功能, 所以将依赖链转为事件流之后, 即可认为已经识别出了对应的用例. 获取事件流后, 根据事件流描述的业务, 手动补充用例名称和事件流描述. 最后, 将系统之外的泳道表示为参与者, 将泳道与活动节点的关系表示为参与者与用例的关联关系, 完成用例的半自动生成.

最终识别并生成的用例图如图 4 所示. 由于用例图的事件流一般都使用文字叙述, 不直接在图中体现, 因此此处使用文本框简略描述事件流.

4 案例分析

本节中, 我们将用一个牙科诊所系统的部分业务流程进行案例分析. 描述该系统业务流程的活动图如图 5 所示. 病人可以在系统上预约诊治, 也可以直接在诊所由医师助手安排安装矫正器.

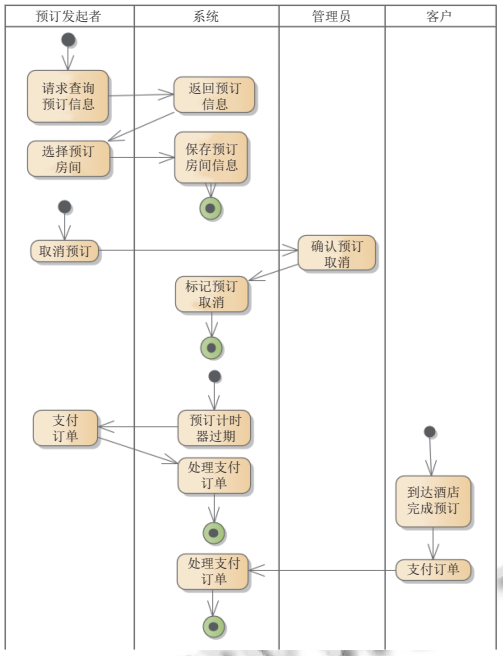

图 3 分解之后的活动链

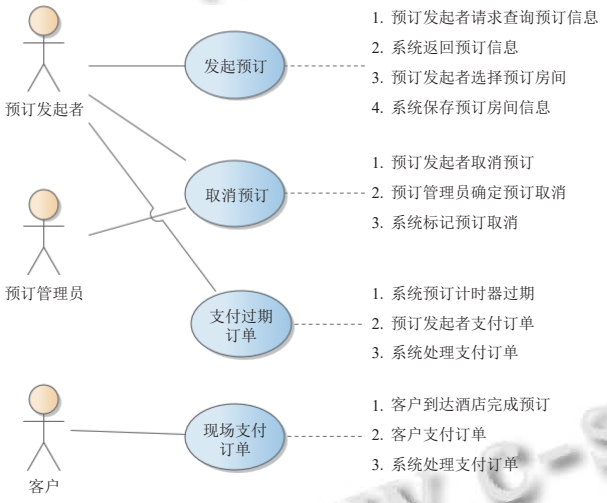

图 4 由活动图生成的用例图

预约就诊、诊治病人、支付诊治订单的流程由以下步骤组成.

1) 病人首先需要询问诊所接待员预约的日期是否可用, 如果预约的日期有医师能够诊治, 接待员就可以联系病人, 让病人提供信息, 由接待员输入信息之后保存在系统内.

2) 等待病人前来就诊. 在病人就诊完成之后, 医师助手记录病人就诊完成, 并根据病人的就诊情况安排病人是否需要后续诊治.

3) 如果需要安排复诊, 则医师助手在系统上输入诊治安排, 并在病人复诊结束后生成支付订单, 否则系统直接生成支付订单, 等待病人支付.

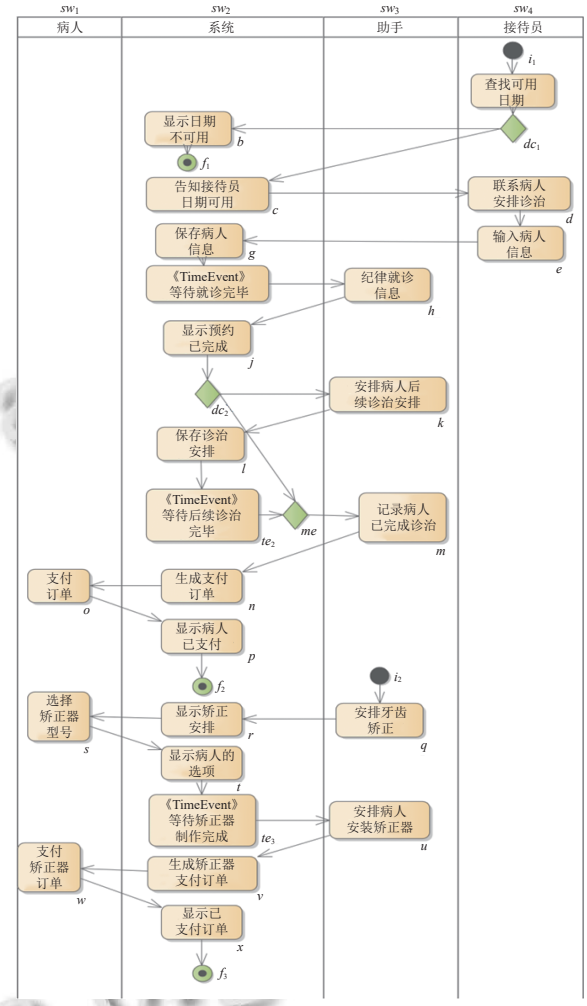

图 5 牙科诊所活动图

矫正器定制、矫正器安装的流程由以下步骤组成.

1) 病人来到诊所后, 医师助手安排病人矫正流程, 病人选择自己的矫正器型号. 注意, 病人来到诊所的步骤无需与系统交互, 因此不属于用例的一部分.

2) 在系统确认之后, 等待矫正器制作完成.

3) 助手给病人安装制作完成的矫正器.

4) 在安装完成之后, 系统生成矫正器订单, 等待病人支付.

根据以上步骤, 绘制出图 5 所示的活动图, 并应用第 3 节的方法进行用例识别.

首先, 给出活动图的统一结构. 图 5 的活动图可以用统一结构表示为 $AD=\langle ME,\prec,\overset{\text{Sequence}}{\omega},\lambda,\overset{\text{InitialAct}}{\tau},\overset{\text{FinalAct}}{\tau},\overset{\text{Activity}}{\tau},\overset{\text{Fork}}{\tau},\overset{\text{Join}}{\tau},\overset{\text{Decision}}{\tau},\overset{\text{Merge}}{\tau},\overset{\text{Swimlane}}{\tau}\rangle$, 其中:

1) $ME=\{sw_1,sw_2,sw_3,sw_4,i_1,a,dc_1,b,f_1,c,d,e,g,te_1,h,j,dc_2,me,k,l,m.n,o,p,f_2,i_2,q,r,s,t,te_3,u,v,w,x,f_3\}$

2) $<=\{(o,sw_1),(s,sw_1),(w,sw_1),(b,sw_2),(f_1,sw_2),(c,sw_2),(g,sw_2),(te_1,sw_2),(j,sw_2),(dc_2,sw_2),(l,sw_2),(te_2,sw_2),(me,sw_2),(n,sw_2),(p,sw_2),(f_2,sw_2),(r,sw_2),(t,sw_2),(te_3,sw_2),(v,sw_2),(x,sw_2),(f_2,sw_3),(h,sw_3),(k,sw_3),(m,sw_3),(i_2,sw_3),(q,sw_3),(u,sw_3),(i,sw_4),(dc_1,sw_4),(d,sw_4),(e,sw_4)\}$

3) $\overset{\text{Sequence}}{\omega}=\{(a,i_1),(dc_1,a),(b,dc_1),(f_1,b),(c,dc_1),(d,c),(e,d),(g,e),(te_1,g),(h,te_1),(j,h),(dc_2,j),(k,dc_2),(l,k),(te_2,l),(me,te_2),(me,dc_2),(m,me),(n,m),(o,n),(p,o),(f_2,p),(q,i_2),(r,q),(s,r),(t,s),(te_3,t),(u,te_3),(v,u),(w,v),(x,w),(f_3,x)\}$

4) $\lambda=\varnothing$

5) $\overset{\text{InitialAct}}{\tau}=\{i_1,i_2\}$

6) $\overset{\text{FinalAct}}{\tau}=\{f_1,f_2,f_3\}$

7) $\overset{\text{Activity}}{\tau}=\{a,b,c,d,e,g,h,j,k,l,m,n,o,p,q,r,s,t,u,v,w,x\}$

8) $\overset{\text{Fork}}{\tau}=\varnothing$

9) $\overset{\text{Join}}{\tau}=\varnothing$

10) $\overset{\text{Decision}}{\tau}=\{dc_1,dc_2\}$

11) $\overset{\text{Merge}}{\tau}=\{me\}$

12) $\overset{\text{Swimlane}}{\tau}=\{sw_1,sw_2,sw_3,sw_4\}$

其次, 将活动图的统一结构作为输入, 应用算法 1. 算法将识别依赖链, 删除时间事件, 并生成分解的依赖链集合. 图 5 的活动图能够识别出 $i_1adc_1bf_1$, $i_1adc_1cdegte_1hjdc_2klte_2memnopf_2$, $i_1adc_1cdegte_1hjdc_2memnopf_2$, $i_2qrstte_3euvwxf_3$ 这 4 个依赖链. 将它们之中的时间事件删除后, 得到如下 7 个依赖链 $i_1adc_1bf_1$, i_1adc_1cdeg, $hjdc_2kl$, $hjdc_2memnopf_2$, $memnopf_2$, i_2qrst, $uvwxf_3$. 其中除 $i_1adc_1bf_1$ 以外, 剩余 6 个依赖链都是不完整的依赖链.

随后修复依赖链. 在 6 个不完整的依赖链中补充起始节点与结束节点, 得到 i_1adc_1cdegf, $ihjdc_2klf$, $ihjdc_2memnopf_2$, $imemnopf_2$, i_2qrstf, $ieuvwxf_3$ 这 6 个修复的依赖链.

最后, 根据依赖链生成用例, 根据泳道生成参与者与关联关系, 并补充对应的用例描述, 完成用例识别. 将依赖链对应的活动作为对应用例的事件流, 手动补充用例名, 将泳道与活动的关系转为用例与参与者的关系, 识别用例并生成用例图的整个过程就完成了.

我们使用 JavaScript 的 GOJS 图形库结合 jQuery 开发的原型工具能够重新绘制图 5 所示的活动图, 工具使用 GOJS 指定的数据结构描述 UML 元素, 使用 jQuery 结合原生 JavaScript 进行程序逻辑与算法的编写. 图 5 所示的活动图描述了病人预约就诊的业务、病人就诊之后支付订单的业务与病人安装牙齿矫正器的业务.

工具可以在绘制活动图后生成它的统一结构, 分解活动图中的时间事件, 并识别活动图中的依赖链, 点击识别出的依赖链则可以高亮显示它, 部分效果如图 6 所示. 如果有至少一条完全的依赖链被识别, 那么就可以生成依赖链对应的用例, 并根据泳道与活动的关系, 自动生成参与者与关联关系. 对图 6 的活动图进行识别得到的用例图如图 7 所示. 此处的用例名需要手动指定. 该案例研究表明, 使用活动图描述业务流程, 将活动图转为形式化模型并进一步识别用例是一个可行的方法.

5 相关工作

用例识别的工作本质上是对需求的进一步解释与挖掘, 是对用户具体需求的完整捕获. 一些早期工作关注于直接从需求文档中捕获需求并识别出用例. Hamza 等人[7] 提出了直接使用自然语言处理 (natural language processing, NLP) 从需求文档中提取需求的方法. 通过对需求文档的内容进行拼写检查、切分、词性标记化、分块、语法模式标记等工作之后, 识别并生成新的用例. Narawita[8] 使用 UML Generator 配合机器学习来分析需求, 生成 XML, 进一步生成用例图与类图. Bajwa 等人[16] 使用人工智能技术与 LESSA 方法来阅读和分析用户提供的场景, 自动生成用例. 这些方法都使用到了 NLP 与人工智能相关的技术来自动化识别用例, 有着相对较高的生成准确率. 相较于通过 NLP 方法获取用例, 本文提出的方法直接从业务模型中识别用例, 适合已经进行了业务建模的需求或能够迅速进行业务建模的需求. 本文提出的方法与 NLP 获取用例的方法存在类似的问题. NLP 的识别是一种类似于基于规则的识别方法, 它对对应语言的语法与词性等要求很高, 如果需求文档的编写不规范或者有错误拼写, 那么使用 NLP 识别出来的用例可能会不完整, 甚至出错. 本文提出的方法也没有对业务流程的来源进行冲突检测工作, 如果业务本身存在冲突或错误描述, 识别的用例描述的系统功能也可能会存在冲突问题. 这一点也是我们后续研究的重点.

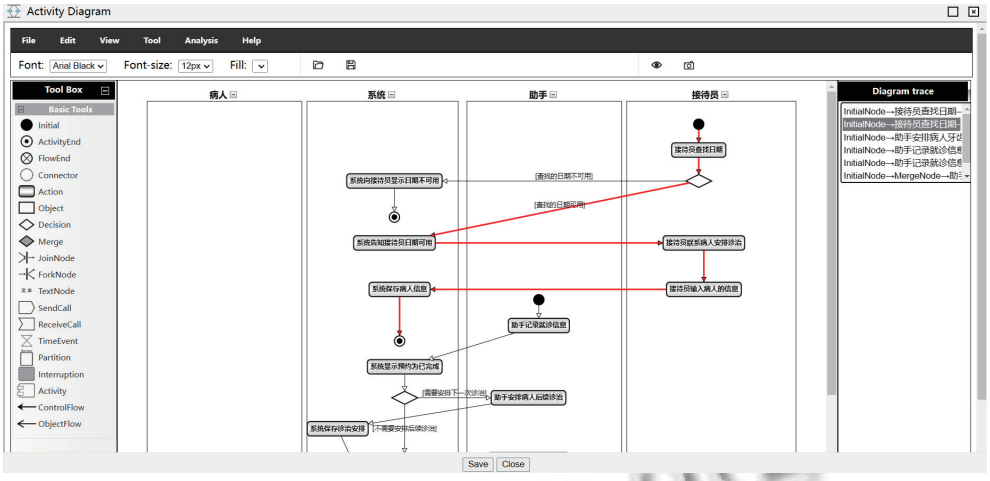

图 6 已识别的依赖链

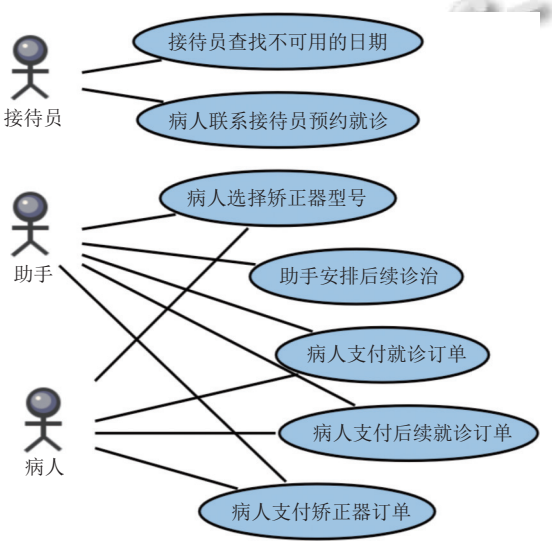

图 7 已识别的用例图

此外, 对用例进行识别的工作也与模型之间的可追溯性相关. 能够满足模型之间的可追溯性才能保证整体建模的正确性. Bouzidi 等人[17] 设计了一种基于业务流程建模与标注 (business process modeling notation, BPMN) 与 UML 用例之间的可追溯性研究, 通过建立一个包含所有 BPMN 与用例模型元素的元模型, 使用这个元模型定义 BPSUC 模型 (business process supported use cases) 之后, 通过 BPSUC 模型来建立 BPMN 与用例模型来确保模型之间的一致性. 该方法讨论了业务流程和需求模型之间的可跟踪性, 不过文中使用的用例图案例很简单, 无法验证 BPSUC的准确性. Khlif 等人[18] 提出了对设计追踪方法的改进和扩展, 以解决设计、需求和代码之间的追踪问题, 该方法给出了一些可追溯性规则, 并基于这些规则评估 UML 模型之间的一致性. 这些规则都有对应的图解与案例, 但没有验证每一个规则的正确性. Park 等人[19]提出了对 UML 用例元模型的扩展, 以将抽象用例细化到更细化的用例. 用例的改进允许在不同的抽象级别上跟踪用例, 但是作者没有解释如何建立这些跟踪. 本文的工作可以在一定程度上满足可追溯性. 在需求修改之后, 直接寻找需求对应的依赖链, 再进一步找到通过依赖链识别出的用例, 最后进行版本间的追溯性与一致性分析等工作.

在用例模型本身的研究上, Cockburn[20] 等处于 UML 早期阶段的研究人员大多根据自己的经验手工描述用例, 这类方法没有采用形式化方法, 无法保证前后建模的一致性, 进而影响到后续的软件开发. 在用例图的形式化研究方面, Kautz 等人[10] 给出了用例图的形式化语义, 并提出了自动识别语义差异的方法. 陈振庆[11] 提出了基于描述逻辑 SHOIN (D) 的形式化描述用例图的方法. 李智等人[12] 提出了结合问题框架与模型驱动技术, 将问题模型转换为用例图的方法. 该方法能够根据问题图中需求之间的关系来建立用例之间的扩展或包含关系, 但没有描述用例的事件流. Bensalem 等人[13] 提出了一个基于逻辑的形式化方法, 并将其应用在了两个复杂的用例上. 文献 [10–13] 采用的形式化方法与我们使用的形式化方法都不一样. 相较于之前的工作, 本文提出的识别用例的方法从业务建模中直接获取业务流程信息, 将业务流程通过形式化模型描述为依赖链, 并按时间跨度进行分解, 进一步识别为用例的事件流, 根据泳道与活动的关系自动生成用例与参与者的关联关系, 实现了用例的半自动生成. 这一思

路目前还没有在其他工作中体现.

软件需求文档中的信息提取一直是一个备受关注的研究课题. 虽然已经有了需求文档规范化相关的研究[21], 但是到目前为止, 还没有一种完整的自动化方法能够从软件需求文档的非结构化语言中提取关键信息[9]. 因此, 我们目前还没有发现任何一种能够直接自动识别用例的方法.

6 结语

本文首先给出了用例模型与活动图的基本概念, 随后使用形式化模型统一结构描述了活动图, 并给出了寻找依赖链的定义与识别方法. 通过原型工具进行案例分析, 说明了本文提出的方法能够识别出完全的依赖链, 并生成对应的用例.

本文提出的方法也有不足之处. 识别依赖链的方法没有考虑活动边的执行条件, 也没有对决策节点的分支做更加细致的处理, 因此只能根据泳道与活动之间的关系识别出用例图的关联关系. 在接下来的工作中, 我们会进一步改进识别用例的功能, 使识别出来的用例能够体现包含与扩展关系, 我们也将验证用例图与其他 UML 模型之间的一致性与正确性, 并继续完善原型工具, 使之成为可用且高效的 CASE 工具.

参考文献

1 Hausmann JH, Heckel R, Taentzer G. Detection of conflicting functional requirements in a use case-driven approach: A static analysis technique based on graph transformation. Proceedings of the 24th International Conference on Software Engineering. Orlando: ACM, 2002. 105–115.

2 Object Management Group. OMG unified modeling language™ (UML®) version 2.5. https://www.omg.org/spec/UML/2.5/PDF. (2015-03-01).

3 Jacobson I, Booch G, Rumbaugh J. 统一软件开发过程. 周伯生, 冯学民, 樊东平, 译. 北京: 机械工业出版社, 2002.

4 程勇, Cai ZM, 袁兆山. 从面向对象到面向目标的需求分析. 计算机科学, 2001, 28(12): 113–117.

5 梁玮. 基于 UML 的面向对象建模方法研究. 软件导刊, 2009, 8(1): 47–49.

6 丁峰, 毛少杰, 施振明. UML 和 ROSE 工具在指挥控制系统开发中的应用. 计算机工程, 2000, 26(10): 118–120.

7 Hamza ZA, Hammad M. Generating UML use case models from software requirements using natural language processing. Proceedings of the 8th International Conference on Modeling Simulation and Applied Optimization (ICMSAO). Manama: IEEE, 2019. 1–6.

8 Narawita CR. UML generator-use case and class diagram generation from text requirements. International Journal on Advances in ICT for Emerging Regions, 2017, 10(1): 1–10.

9 郑七凡. 软件需求文档的 UML 用例图抽取研究 [硕士学位论文]. 哈尔滨: 东北林业大学, 2020. [doi: 10.27009/d.cnki.gdblu.2020.000948]

10 Kautz O, Rumpe B, Wachtmeister L. Semantic differencing of use case diagrams. Journal of Object Technology, 2022, 21(3): 3.

11 陈振庆. UML 用例图的形式化及其推理. 贺州学院学报, 2017, 33(2): 144–148.

12 李智, 邓杰, 杨溢龙, 等. 从信息物理融合系统问题模型到 UML 用例图的变换方法. 计算机科学, 2020, 47(12): 65–72.

13 Bensalem S, Cheng CH, Huang XW, *et al*. Formal specification for learning-enabled autonomous systems. Proceedings of the 5th International Workshop on Formal Methods for ML-enabled Autonomous Systems, and the 15th International Workshop on Numerical Software Verification. Haifa: Springer, 2022. 131–143.

14 Jacobson I. Use cases —Yesterday, today, and tomorrow. Software & Systems Modeling, 2004, 3(3): 210–220.

15 Jiang JM, Zhu HB, Li Q, *et al*. Event-based functional decomposition. Information and Computation, 2020, 271: 104484. [doi: 10.1016/j.ic.2019.104484]

16 Bajwa IS, Hyder I. UCD-generator—A LESSA application for use case design. Proceedings of the 2007 International Conference on Information and Emerging Technologies. Karachi: IEEE, 2007. 1–5.

17 Bouzidi A, Haddar N, Haddar K. Traceability and synchronization between BPMN and UML use case models. Ingénierie des Systèmes d Information, 2019, 24(2): 215–228.

18 Khlif W, Kchaou D, Bouassida N. A complete traceability methodology between UML diagrams and source code based on enriched use case textual description. Informatica, 2022, 46(1): 27–47.

19 Park G, Fellir F, Hong JE, *et al*. Deriving use cases from business processes: A goal-oriented transformational approach. Proceedings of the 2017 Symposium on Applied Computing. Marrakech: ACM, 2017. 1288–1295.

20 Cockburn A. Writing Effective Use Cases. Boston: Addison-Wesley Longman Publishing, 2000.

21 陈杨杨, 蒋建民. 面向对象的需求规格说明文档研究. 软件导刊, 2020, 19(4): 102–106.

(校对责编: 牛欣悦)