文章编号：2096-1472(2022)-09-27-07
DOI:10.19644/j.cnki.issn2096-1472.2022.009.007

软件制品可追溯性的形式化建模与分析

李建清，蒋建民

(成都信息工程大学软件工程学院，四川 成都 610000)
✉1102418305@qq.com; jjm@fjnu.edu.cn

摘　要：可追溯性是软件制品管理的重要任务，良好的可追溯性可以大大降低软件开发成本和维护成本，但是传统的可追溯性操作指南难以保证可追溯性的正确性，因此有必要探索如何构建更好的可追溯性模型。与多数传统方法不同，本文使用形式化方法研究可追溯性，并提出了一种新的软件制品可追溯性的形式化模型，给出了软件制品可追溯性的形式化定义，研究了软件制品可追溯性的变更影响分析、制品分析和版本分析方法，最后开发出了原型工具，并用实例验证了该方法的正确性和有效性。实验结果表明，本文提出的形式化建模与分析方法可以较好地保持软件制品的可追溯性。

关键词：可追溯性；变更影响分析；制品分析；版本分析
中图分类号：TP311.5　**文献标识码**：A

Formal Modeling and Analysis of Software Product Traceability

LI Jianqing, JIANG Jianmin

(*College of Software Engineering*, *Chengdu University of Information Technology*, *Chengdu* 610000, *China*)
✉1102418305@qq.com; jjm@fjnu.edu.cn

Abstract: Traceability is an important task of software product management and good traceability can greatly reduce the cost of software development and maintenance. However, traditional traceability operation guidelines cannot guarantee the correctness of traceability, so it is necessary to explore how to build a better traceability model. Unlike most traditional methods, by using a formal approach to study traceability, this paper proposes a new formal model for software product traceability and provides a formal definition of software product traceability. It also investigates changing impact analysis, product analysis and version analysis methods for software product traceability, and finally develops a prototype tool and verifies the correctness and effectiveness of the method with examples. Experimental results show that the proposed formal modeling and analysis method can better maintain the traceability of software products.

Keywords: traceability; changing impact analysis; product analysis; version analysis

1 引言(Introduction)

在软件生命周期过程中，常常会产生基于文本的需求、需求模型、设计模型、源代码、测试用例、各种附加文档等复杂多样的软件制品。良好的软件制品可追溯性的潜在好处是更集中于开发、更低的维护成本、更清晰的文档和更精确的变更影响分析[1]。针对软件制品可追溯性已经研究多年[2−3]，然而在管理可追溯性时，从业者得到的科学文献支持很少[4]，有必要探索如何构建更好的可追溯性模型。一个好的可追溯性模型不仅可以保证各类软件制品之间的可追溯性关系以及这些软件制品内部元素之间的关系，而且还支持定制，并允许使用扩展性来定义新的可追溯性链接类型。我们研究的可追溯性模型完全具有这些性质。

可追溯性是描述和跟踪软件制品生命周期的能力，通常存在两种类型的软件制品可追溯性模型，一种是非形式化的，而另一种是形式化的。我们专注于形式模型，并使用形式方法来探索可追溯性。目前已有一些研究[5−6]使用形式化方法建模和分析可追溯性，但这些与我们的工作完全不同。传统的形式化模型通常采用行为模型，例如Petri网[7]、进程代数[8]、

基金项目：国家自然科学基金(61772004)；成都信息工程大学人才科研基金(KYTZ202009)；CCF−华为形式验证创新研究计划(CCF−HuaweiFM2021003).

迁移系统[9]。根据统一建模语言UML(Unified Modeling Language)和系统建模语言SysML(Systems Modeling Language)，软件系统模型分为行为模型和结构模型。我们认为结构模型足以对可追溯性进行建模和分析，因为可追溯性关系是一种结构关系。此外，如果用行为模型描述可追溯性，必须使用复杂的可达性算法来分析可追溯性，需要考虑状态爆炸问题，从而花费较大代价。在本文中，我们提出了一种形式化的可追溯性模型，称为“结构模型”，用于建模和可视化可追溯性，并基于该结构模型讨论了一些可追溯性的分析方法，比如变更影响、制品和版本分析，最后实现了基于结构模型的支持工具并用实验证明我们的新方法是有效的。

本文其余部分的结构如下：第2部分是可追溯性模型；第3部分是可追溯模型的组合；第4部分定义了可追溯性；第5部分是可追溯性分析；第6部分是案例研究和我们的支持工具介绍；第7部分是相关工作；第8部分总结了论文。

2 可追溯性模型(Traceability model)

在本部分中，我们将介绍结构模型来对可追溯性进行建模和分析。在SysML中，各种软件制品都被视为模型，其组成部分是模型元素。这种考虑有助于使用基于图形的工具对可追溯性进行分析和可视化表示，我们在这里采用这些概念。为方便起见，我们先给出一些辅助定义。令VN是版本号的集合，使得$\forall R \subseteq VN \times VN$，$R$是全序关系。Min($VN$)和Max($VN$)分别表示$VN$中的最小和最大版本号。令$\infty$为一个特殊的版本号，这意味着一个未确定的版本号并且$\infty \notin VN$。

定义2.1：结构模型(SM)是$vs: ME \to VN \cup \{\infty\}$一个元组$< ME, \prec, \overset{1}{\to}, \cdots, \overset{n}{\to}, vs, ve >$，其中，

- ME是模型元素的有限集；
- $\prec \subseteq ME \times ME$是包含关系，它是一个(非自反的)偏序；
- $\forall i \in \{1, \cdots, n\}, \overset{i}{\to} \subseteq ME \times ME$是依赖关系；
- $vs: ME \to VN \cup \{\infty\}$是初始版本函数；
- $ve: ME \to VN \cup \{\infty\}$是最终版本函数。

这里，对于所有的$x, y \in ME, x \overset{i}{\to} y (i \in \{1, \cdots, n\})$称为依赖，读作$x$依赖于$y$(注意：$i$表示依赖的类型)。而$x \prec y$表示$x$包含在$y$中。如果$x \prec z$，$y \prec z$，则它们简单地表示为$x, y \prec z$，其中$x$和$y$都包含在$z$中。结构模型可以指定软件制品的版本和软件制品之间的关系，并且可以进一步建模可追溯性。模型元素在创建时被分配了初始版本号，而在它被取消激活时被分配了最终版本号。接下来，我们将继续讨论如何实现该策略。为了使用结构模型对可追溯性进行建模，我们首先在这里举一个简单的例子。

例2.1：图1根据SysML中的图解符号展示了一个简单的用户账户管理系统中制品之间的偏序关系。显然，存在五种类型的关系：包含(containment)、跟踪(trace)、扩展(extend)、包括(include)和聚合(aggregation)。制品和制品之间的关系可用结构模型表示为$< ME, \prec, \overset{trace}{\to}, \overset{extend}{\to}, \overset{include}{\to}, \overset{aggregation}{\to}, vs, ve >$，其中$\overset{trace}{\to}$、$\overset{extend}{\to}$、$\overset{include}{\to}$和$\overset{aggregation}{\to}$分别对应制品间的追溯关系、用例间的扩展关系、用例间的包括关系、类间的聚合关系。接下来，我们将进行详细的说明。模型元素集ME具有元素RQ、R1、R2、R3、U1、U2、U3、U4、U5、C1、C2、D1。需求R1、R2、R3是RQ的子需求，即R1、R2、R3$\prec$RQ。

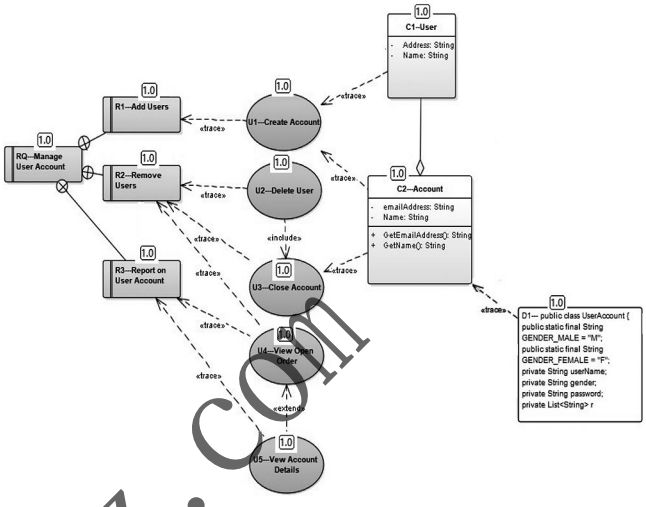

图1 系统中的偏序关系

Fig.1 The partial relationships in a system

根据图1中模型元素的关系，我们可以有如下依赖关系：$\overset{trace}{\to} = \{(U1, R1), (U2, R2), (U3, R2), (U4, R2), (U4, R3), (U5, R3), (D1, C2), (C1, U1), (C2, U1), (C2, U3)\}$，$\overset{extend}{\to} = \{(U5, U4)\}$，$\overset{include}{\to} = \{(U2, U3)\}$，$\overset{aggregation}{\to} = \{(C1, C2)\}$。我们可以假设当前版本号是1.0，其中版本号用圆角矩形表示(图1)，因此$vs(RQ) = vs(R1) = vs(R2) = vs(R3) = vs(U1) = vs(U2) = vs(U3) = vs(U4) = vs(U5) = vs(C1) = vs(C2) = s(D1) = 1.0$且$ve(RQ) = ve(R1) = ve(R2) = ve(R3) = ve(U1) = ve(U2) = ve(U3) = ve(U4) = ve(U5) = ve(C1) = ve(C2) = ve(D1) = \infty$。请注意，当前版本号通常是初始版本号。图1中标注的版本为初始版本号。一旦将模型元素分配给最终版本号，它将变为灰色。

该示例表明，结构模型可以指定软件系统中各种制品之间的关系。它可以直接使用原来的关系来表示用例模型之间的内部关系(比如include和extend)及类之间的关系(比如aggregation)。一旦结构模型明确了系统生命周期过程中模型的模型元素之间的关系，我们就可以从其结构的角度来分析该系统。

定义2.2：设$SM =< ME, \prec, \overset{1}{\to}, \cdots, \overset{n}{\to}, vs, ve >$为结构模型。

(1)给定序列$rc = x_1 \cdots x_n$，若$\forall i \in \{1, \cdots, n-1\}$，$x_i, x_{i+1} \in ME$，$(x_i, x_{i+1}) \in (\prec \cup \overset{1}{\to} \cup \cdots \cup \overset{n}{\to}) \vee (x_{i+1}, x_i) \in (\prec \cup \overset{1}{\to} \cup \cdots \cup \overset{n}{\to})$，则称序列$rc$是$SM$中的一个关系链。$\overline{rc}$表示关系链$rc$中的模型元素，即$\overline{rc} = \{x_1, \cdots, x_n\}$。RC($SM$)表示$SM$中所有可能的关系链。

(2)给定序列$dc = x_1 \cdots x_n$，若$\forall i \in \{1, \cdots, n-1\}$，$x_i, x_{i+1} \in ME$，$(x_i, x_{i+1}) \in (\prec \cup \overset{1}{\to} \cup \cdots \cup \overset{n}{\to})$，则称序列$dc$是在$SM$中的一个依赖链。$\hat{dc}$表示依赖链$dc$中所有的模型元素，即$\hat{dc} = \{x_1, \cdots, x_n\}$。

DC(SM)表示SM中所有可能的依赖链。令[dc]表示依赖链dc中模型元素的个数，即[dc]=n。

显然，关系链是间接的，而依赖链是直接的。例如，例2.1中存在依赖链 D1C2U3R2，依赖链 D1C2U3R2 也是关系链。

命题2.1：设SM为结构模型且$dc=x_1\cdots x_n\in \mathrm{DC}(SM)$。如果$\forall i\in\{1,\cdots,n-1\}$，$(x_i,x_{i+1})\in\prec$，则$dc$中不存在环。

证明：根据定义2.1，包含关系$\prec$是一个(非自反的)偏序，而且由于依赖链dc只包含了包含关系，因此dc中不存在环。

命题2.2：如果SM是一个结构模型，则$\mathrm{DC}(SM)\subseteq \mathrm{RC}(SM)$。

证明：这个命题直接可得。

显然，在一个结构模型中，关系链的数量大于等于依赖链的数量。

3 可追溯性模型的组合(Composition of traceability model)

复杂的软件系统包含许多制品，并分为由不同团队开发的多个子系统。每个团队都需要构建自己的软件制品可追溯性系统。一旦整个系统完成，所有的制品必须放在一起，并组成完整的可追溯性系统。因此，有必要讨论可追溯性的组合问题。

定义3.1：设有$SM'=<ME',\prec',\xrightarrow{1}',\cdots,\xrightarrow{n}',vs',ve'>$和$SM''=<ME'',\prec'',\xrightarrow{1}'',\cdots,\xrightarrow{n}'',vs'',ve''>$两个结构模型，若$\forall e\in ME'$，$vs''(e)=vs'(e)\wedge ve'(e)=ve''(e)$，则$SM'$和$SM''$的组合$SM'\oplus SM''=<ME,\prec,\xrightarrow{1},\cdots,\xrightarrow{n},vs,ve>$，其中$ME=ME'\cup ME''$，$\prec=\prec'\cup\prec''$，$\forall i\in\{1,\cdots,n\}$ $\xrightarrow{i}=\xrightarrow{i}'\cup\xrightarrow{i}''$，$\forall e\in ME':vs(e)=vs'(e)$，$ve(e)=ve'(e)$，$\forall e\in ME'':vs(e)=vs''(e)$，$ve(e)=ve''(e)$。$SM'$、$SM''$称为是可组合的。

需要注意的是，如果两个可组合的结构模型可能有不同数量的依赖关系类型，可能有不同的依赖关系，我们可以在组合之前将它们等价转换为两个具有相同数量依赖类型的结构模型。例如，SM'和SM''是可组合的，其中$SM'=<ME',\prec',\xrightarrow{a}',\xrightarrow{x}',vs',ve'>$和$SM''=<ME'',\prec'',\xrightarrow{x}'',\xrightarrow{y}'',\xrightarrow{z}'',vs'',ve''>$。显然，我们可以将$SM'$和$SM''$分别转化为$SM_1$和$SM_2$：$SM_1=<ME',\prec',\xrightarrow{a}',\xrightarrow{x}',\xrightarrow{y}',\xrightarrow{z}',vs',ve'>$，其中$\xrightarrow{y}'=\xrightarrow{z}'=\varnothing$；$SM_2=<ME'',\prec'',\xrightarrow{a}'',\xrightarrow{x}'',\xrightarrow{y}'',\xrightarrow{z}'',vs'',ve''>$，其中$\xrightarrow{a}''=\varnothing$。显然，$SM_1=SM'$，$SM_2=SM''$。此外，$SM_1$和$SM_2$具有相同数量的依赖类型。因此，$SM_1$和$SM_2$可以根据前面的定义组合。结构模型的组合具有以下性质。

命题3.1：设SM、SM'和SM''为三个结构模型，并且让三个结构模型中的每两个都是可组合的，则下列结论成立：

(1)$SM'\oplus SM''$是一个结构模型；

(2)$SM'\oplus SM''=SM''\oplus SM'$；

(3)$(SM\oplus SM')\oplus SM''=SM\oplus(SM'\oplus SM'')$。

证明：这些命题直接可得。

这个命题表明，结构模型的组合具有封闭性、交换性和结合性。

4 可追溯性的定义(Definition of traceability)

软件制品的可追溯性已被认为是支持软件开发和维护过程中各种活动的重要质量因素[10]。软件生命周期过程通常包含顺序阶段：需求、设计、实现、测试和维护，每个阶段都存在各种制品。一般认为，下一阶段的制品(模型元素)依赖于前一阶段的制品。文献[11]和文献[12]中的可追溯性思想侧重于需求和代码(实现)阶段之间的依赖关系。我们扩展了这些想法，并提出了以下横向和纵向可追溯性的形式化定义，并进一步给出了可追溯性的定义。

定义4.1：设$SM=<ME,\prec,\xrightarrow{1},\cdots,\xrightarrow{n},vs,ve>$为结构模型，令$RE\subseteq ME$是表示需求的模型元素集合，$MA\subseteq ME$是表示需求、设计、实现、测试或维护阶段的制品的模型元素集合。

(1)如果$\forall e\in ME$，$\exists dc=x_1\cdots x_n\in \mathrm{DC}(SM):e\in\hat{dc}\wedge x_n\in RE\wedge x_1\in MA$，则称$SM$是水平可追溯的。

(2)如果$\forall e\in ME$，$\exists dc=y_1\cdots y_n\in \mathrm{DC}(SM),\forall i\in\{1,\cdots,n-1\}:e\in\hat{dc}\wedge vs(y_{i+1})\leqslant vs(y_i)$，则称$SM$是垂直可追溯的。

(3)如果SM是水平和垂直可追溯的，则称SM是可追溯的。

这里，水平可追溯性考虑需求和维护阶段之间的直接和间接依赖关系，而垂直可追溯性则侧重于模型元素的版本变化[12]，即模型元素的版本号大于或等于其依赖模型元素，如图2所示。显然，虽然依赖链是直接的，但它们可以基于有向图反向遍历。因此，向前可追溯性和向后可追溯性包含在此定义中。定义不仅考虑了模型内部的可追溯性，还考虑了多个模型之间的可追溯性。

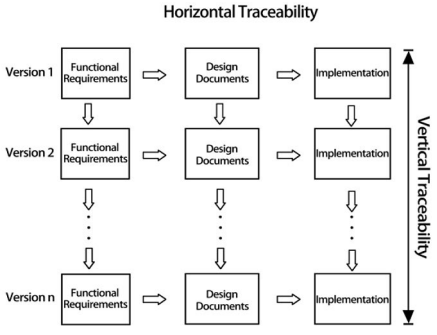

图2 水平方向和垂直方向的可追溯性

Fig.2 Horizontal and vertical traceability

例4.1：我们继续例2.1。在如图1所示的用户账户管理系统中存在模型元素RQ、R1、R2、R3、U1、U2、U3、U4、U5、C1、C2、D1。显然，对于任何模型元素，都存在依赖关系链使得模型元素直接或间接依赖于其他模型元素。根据定义4.1(1)，案例意味着这样的模型元素是可追溯的。因此，该系统是水平可追溯的。由于只有一个版本号，并且每个模型元素都具有相同的版本号，根据定义4.1(2)，系统是垂直可追溯的。根据定义4.1(3)，图1中描述的用户账户管理系统是可追溯的。

定理4.1：设$SM'=<ME',\prec',\xrightarrow{1}',\cdots,\xrightarrow{n}',vs',ve'>$且$SM''=<ME'',\prec'',\xrightarrow{1}'',\cdots,\xrightarrow{n}'',vs'',ve''>$为两个结构模型，$SM'$和$SM''$是可组合的，如果$SM'$和$SM''$是可追溯的，则$SM'\oplus SM''$是可追溯的。

证明：由命题2.2可知$(\mathrm{DC}(SM)\cup\mathrm{DC}(SM'))\subseteq\mathrm{DC}(SM\oplus SM')$，根据定义3.1，对于$ME'\oplus ME''$中的任何模型元素$e$，$e\in ME'$或$e\in ME''$。当$e\in ME'$时，由于$SM'$是可追溯的，根据定义4.1(1)和4.1(2)可得$\exists dc_1=x_1\cdots x_n\in\mathrm{DC}(SM):e\in\hat{dc}_1\wedge x_n\in RE\wedge x_1\in MA$且$\exists dc_2=y_1\cdots y_n\in\mathrm{DC}(SM)$，$\forall i\in\{1,\cdots,n-1\}:e\in\hat{dc}\wedge vs(y_{i+1})\leqslant vs(y_i)$。当$e\in ME''$时，由于$SM''$是可追溯的，根据定义4.1(1)和4.1(2)可得$\exists dc_3=x_1\cdots x_n\in\mathrm{DC}(SM):e\in\hat{dc}_3\wedge x_n\in RE\wedge x_1\in MA$并且$\exists dc_4=y_1\cdots y_n\in\mathrm{DC}(SM),\forall i\in\{1,\cdots,n-1\}:e\in\hat{dc}\wedge vs(y_{i+1})\leqslant vs(y_i)$。由于$e$是$ME'\oplus ME''$中的任何模型元素，根据定义4.1(1)和4.1(2)，$SM'\oplus SM''$是水平可追溯和垂直可追溯的。因此，根据定义4.1(3)，$SM'\oplus SM''$是可追溯的。

定理4.1表明可追溯性在一定条件下是可组合的。

5 可追溯性分析(Traceability analysis)

基于结构模型，这里介绍了一些可追溯性分析方法。

5.1 变更影响(覆盖)分析

在大型软件项目中，变更影响分析和变更覆盖分析在控制软件演化方面起着重要作用[12−13]。一旦创建、修改和删除模型元素，因为可能存在许多与该模型元素直接或间接相关的模型元素，因此相关的模型元素也可能会发生变化。我们的方法提供了更精确的变更影响分析，支持变更替代识别、消除误报和变更一致性检查。

定义5.1.1：设$SM=<ME,\prec,\xrightarrow{1},\cdots,\xrightarrow{n},vs,ve>$为结构模型，并且$e\in ME$为模型元素。$e$的影响集$\mathrm{ImpSet}(e)=\{e'\in ME\mid(\exists dc=d_1\cdots d_m\in\mathrm{DC}(SM):e'=d_1\wedge d_m=e)\}$。

模型元素的影响集包含直接或间接依赖于该模型元素的元素。该定义给出了一种计算模型元素影响集的方法。由于结构模型形成一个有向图，其中模型元素是顶点，包含或依赖关系是边，可以使用图遍历算法轻松计算任何模型元素的影响集，我们的工具已经实现了这个功能(见第6部分)。

例5.1.1：此处仍继续例2.1。显然，当在软件开发和维护过程中添加需求时，开发人员最重要的是快速准确地找到受影响的需求项、设计模型和代码片段，以便于开发和维护。例如，一旦添加了需求项R1.1，它就会直接影响模型元素R1，因为R1.1成为R1的子项。根据定义5.1.1，R1的影响集是$\mathrm{ImpSet}(R1)=\{U1,C1,C2,D1\}$(图1)。

根据R1的影响集，开发人员必须分析模型元素U1、C1、C2、D1是否必须进行修改，最终决定将U1、C1、D1分别修改为U1.1、C1.1、D1.1，而C2没有改变。图3描述了新的需求项R1.1的添加，以及在添加和修改操作下新得到的追溯管理系统。这个示例表明，开发人员在使用支持工具时很容易自动获得每个制品受影响的制品。

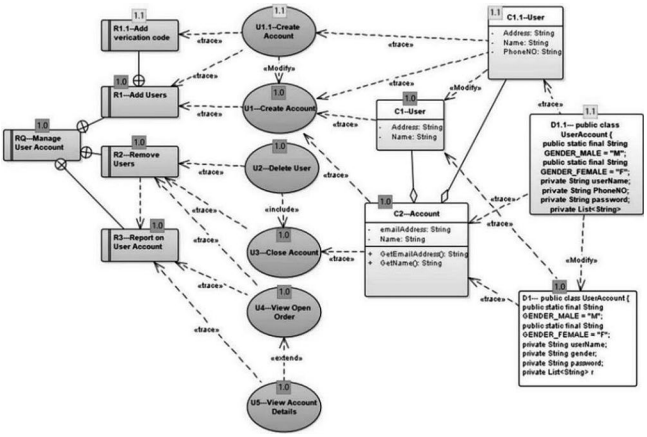

图3 可追溯性管理

Fig.3 Managing traceability

命题5.1.1：设$SM=<ME,\prec,\xrightarrow{1},\cdots,\xrightarrow{n},vs,ve>$为一个可追溯的结构模型，并且$\xrightarrow{x}\in\{\xrightarrow{1},\cdots,\xrightarrow{n}\}$，如果$e,e'\in ME$是两个模型元素，且$(e,e')\in\xrightarrow{x}\cup\prec$，则$\mathrm{ImpSet}(e)\subseteq\mathrm{ImpSet}(e')$。

证明：这个命题直接可得。

该命题表明，如果一个模型元素依赖于另一个，则前者的影响集包含在后者的影响集里。该结果可用于验证可追溯性管理系统中是否存在变更影响错误。例如，在图3中，由于R1.1是R1的孩子，根据命题5.1.1可知$\mathrm{ImpSet}(R1.1)\subseteq\mathrm{ImpSet}(R1)$。显然，由于$\mathrm{ImpSet}(R1)=\{R1.1,U1.1,C1.1,D1.1,U1,C1,C2,D1\}$和$\mathrm{ImpSet}(R1.1)=\{U1.1,C1.1,D1.1\}$，结论$\mathrm{ImpSet}(R1.1)\subseteq\mathrm{ImpSet}(R1)$成立。但是，如果由于某些开发者不正确的手动操作导致ImpSet(R1.1)不是ImpSet(R1)的子集，则必须探究为什么结论不成立，可以使用我们的方法和相应的支持工具找到变更影响错误。

5.2 制品分析

在一个软件产品线中，制品的多个版本共存并同时在多个产品中使用，需要跟踪哪个产品使用哪个版本[14]。显然，存在一些用于构建多个产品的制品。事实上，每一个新产品都对应一个新版本号，但其中也包含许多版本保持不变的制品(请注意，这样的版本号对应于我们的可追溯性模型中的初始版本号，不是我们的可追溯性中的最终版本号)。

在我们的解决方案中，无论何时创建产品，都可以跟踪任何产品的所有制品。当一个产品发布时，需要创建一个模型元素，其中包含代码元素和相应版本号的各种文档，以及一些未更改的旧版本制品。例如，如果某些需求不需要改变，而新产品满足了这些需求，我们可以根据新产品中包含的旧模型元素的直接或间接依赖关系来追溯这些需求。由于版本号通常是递增分配给新模型元素的，因此产品模型元素的版本号大于或等于产品中包含的制品的版本号。

定义5.2.1：设$SM=<ME,\prec,\xrightarrow{1},\cdots,\xrightarrow{n},vs,ve>$是一个结构模型，并且$p\in ME$为一个产品，则产品$p$的制品集$\mathrm{ProSet}(p)=\{e'\in ME\mid\exists dc=d_1\cdots d_m\in\mathrm{DC}(SM),\forall e''\in\hat{dc},vs(e'')\leqslant vs(p),p=d_1\wedge d_m=e'\}$。显然，如5.1部分所述，我们可以使用图遍

历算法轻松计算哪个产品包含哪些制品。例如，假设两个代码片段D1、D1.1是图3中的两个产品。那么ProSet(D1)={C1,C2,U1,U3,R1,R2,RQ}，ProSet(D1.1)={C1.1,U1.1,R1.1,D1,C1,C2,U1,U3,R1,R2,RQ}。我们分别找到了D1和D1.1的制品，D1的制品只有一个版本号——1.0，而D1.1的制品有两个版本号——1.0和1.1。

命题5.2.1：设$SM=<ME,\prec,\xrightarrow{1},\cdots,\xrightarrow{n},vs,ve>$为一个可追溯的结构模型，$p,p'\in ME$为两个产品。如果$p'$是$p$的新版本，则$\text{ProSet}(p)\subseteq\text{ProSet}(p')$。

证明：根据定义4.1(2)，新版本总是大于或等于旧版本。版本号满足定义5.2.1中产品中制品的版本要求。根据定义5.2.1，该结论成立。

该命题指出产品的新版本包含其旧版本的所有制品。在图3中，如果将两个代码片段D1.1、D1视为两个产品，则D1.1应包含D1中的所有制品。根据前面的讨论，ProSet(D1)只是ProSet(D1.1)的子集。

命题5.2.2：设$SM=<ME,\prec,\xrightarrow{1},\cdots,\xrightarrow{n},vs,ve>$为一个可追溯的结构模型。如果模型元素个数$|ME|>1$，则$\forall e\in ME,\exists dc=d_1\cdots d_m\in\text{DC}(SM):e\in\hat{dc}\wedge[dc]>1$。

证明：这个命题直接可得。

这个命题说明，在可追溯的管理系统中不存在孤立的软件制品，即孤立的软件制品是不可追溯的。

5.3 版本分析

作为可追溯性最重要的工作之一，版本控制就是管理代码等制品的变更和版本，并解决合并冲突和相关异常。在本部分中，我们将分析软件制品的版本可追溯性。

命题5.3.1：设$SM=<ME,\prec,\xrightarrow{1},\cdots,\xrightarrow{n},vs,ve>$为一个可追溯的结构模型，并且$e\in ME$为产品，则$\forall e'\in ME$，$vs(e')=vs(e)\Rightarrow e'\in\text{ProSet}(e)$。

证明：根据定义5.2.1，结果显然成立。

该命题表明，如果发布了一个产品，它应该包含版本号和与产品版本号相同的所有可追溯制品。

命题5.3.2：如果$SM=<ME,\prec,\xrightarrow{1},\cdots,\xrightarrow{n},vs,ve>$为一个可追溯的结构模型，并且$e\in ME$为产品，则$\forall e'\in\text{ProSet}(e),vs(e')\leqslant vs(e)$。

证明：根据定义5.2.1，该结论显然成立。

该命题指出，如果追溯管理系统中存在产品，则产品模型元素的版本号应大于或等于产品中包含的每个制品。

命题5.3.3：如果$SM=<ME,\prec,\xrightarrow{1},\cdots,\xrightarrow{n},vs,ve>$是一个可追溯的结构模型，则$\forall e\in ME$，$\exists e'\in ME$，$\exists\xrightarrow{x}\in\{\xrightarrow{1},\cdots,\xrightarrow{n}\}$，$(e,e')\in\xrightarrow{x}:vs(e)\geqslant vs(e')$。

证明：根据定义4.1(2)，新模型元素必须依赖于旧模型元素，并且新模型元素的版本号大于或等于旧模型元素的版本号。此外，同时创建多个模型元素，这些模型元素具有相同的版本号。因此，结果显然成立。

该命题指出，如果一个模型元素在追溯管理系统中依赖于另一个，则前者的版本号应大于或等于后者的版本号。通过这个命题，我们可以在追溯管理系统中使用版本号的比较来发现错误的依赖关系。例如，在图4中，存在依赖关系$C1\xrightarrow{\text{trace}}U1.1$。根据命题5.3.3，$vs(C1)\geqslant vs(U1.1)$。但是图4中的$vs(C1)=1.0$而$vs(U1.1)=1.1$。显然，$vs(C1)<vs(U1.1)$。这个结果与$vs(C1)\geqslant vs(U1.1)$相矛盾。因此，我们可以确定$C1\xrightarrow{\text{trace}}U1.1$是一个不正确的依赖关系。事实上，旧制品不可能依赖新制品，我们在图4中的依赖项上打了一个叉以表明依赖项是不正确的。除了变更影响、制品分析和版本分析之外，我们的工具还可以进行更多的自动化检测分析，如检测是否存在无效依赖项或依赖方向是否正确。例如，由于无法追溯孤立的模型元素，我们的工具可以找到所有孤立的模型元素。又例如，代码片段通常依赖于设计模型元素，并且依赖方向应该符合规则，如果某些依赖方向违反规则，我们的工具也可以检测到它们。

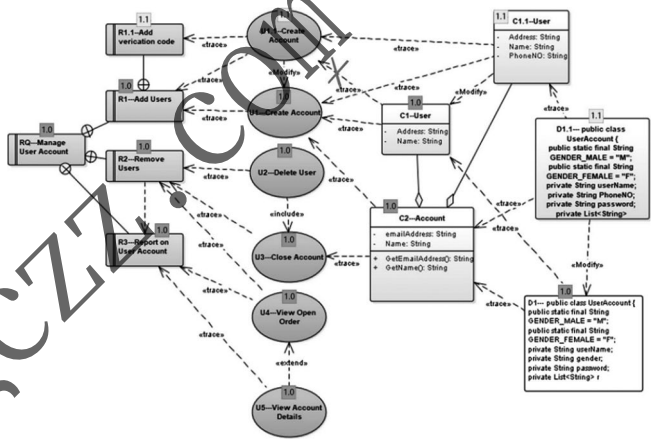

图4 异常的依赖关系和版本号

Fig.4 Abnormal dependencies and version numbers

为了构建基于结构模型的可追溯性管理系统，我们开发了一款Web端名为JLTool的工具。该工具主要由两个完整的模块组成，一个模块是绘制UML图等各种图，创建模型元素之间的可追溯性关系(包括自动化或半自动化UML图导入和可追溯性链接生成)；而另一个是执行可追溯性分析，如变更影响(覆盖)分析、制品分析和版本分析。该工具可以从以下地址使用：http://219.151.152.164:3000/。我校软件工程专业的学生正在使用该工具进行软件开发实验。

6 案例研究(Case study)

在本部分中，我们使用一个简单的地址簿系统演示本文中的结果。本案例(http://www.cs.gordon.edu/courses/cs211/AddressBookExample/)由美国Gordon学院计算机科学教授Russell C. Bjork开发。在初始版本中，简单的地址簿系统包含以下制品：10个需求项、15个用例、11个类、15个序列图和11个代码片段。每个制品都被视为一个模型元素。在我们的工具中，不同类型的模型元素可以包含不同的内容主体。例如，我们的工具提供了一个需求项的对话框，该对话框用于输入需求项的具体文本内容。类似地，我们的工具可以从XMI格式的文件中手动绘制或完全导入UML图作为模

型元素，例如，每个序列图都是一个模型元素(可以通过工具Help中的Case study导入该案例)。

为了证明我们的工具可以提供第5部分中用于可追溯性的管理操作，我们在这里进行修改操作。由于简单案例的初始版本没有提供搜索功能，我们需要将需求项R1修改为R1.1，使R1.1包含搜索功能需求。使用我们的工具，具体操作步骤如下：(1)选择模型元素R1。(2)右键单击R1，显示快捷菜单。(3)从快捷菜单中选择修改，将自动创建一个新的模型元素，它是R1的副本。而且，新模型元素与R1之间的修改关系是自动创建的(即新模型元素依赖于R1)。最后，我们可以手动更改文本描述并为新模型元素设置版本号。例如，我们将新模型元素命名为R1.1。由于需求的变化，相应的用例、类、序列图和代码片段必须发生变化。因此，新版本的简单地址簿系统具有以下制品：11 个需求项、16 个用例、16 个类、16 个序列图和16 个代码片段。

如果我们想修改需求项R1，可以通过我们的工具对R1进行变更影响分析，如图5所示。首先选择R1并右键单击它，会显示一个快捷菜单，其中包括用于分析和所有管理操作的选项。然后点击“变化分析”选项，在主屏幕的右侧浏览器(树视图窗口)中列出了所有直接或间接依赖于R1的模型元素，即ImpSet(R1)={U1,C2,⋯,C10}。如有必要，借助工具，我们可以轻松处理R1和所有直接或间接依赖于R1的模型元素。如果需要进行制品分析，例如进行模型元素S13的制品分析，应该首先选择S13，然后右键单击“制品分析”选项。S13直接或间接依赖的所有模型元素都列在右侧浏览器的“制品分析”选项下，即ProSet(S13)={C1.1,C2.1,⋯,R1.1,R1,RQ}。接下来，我们的工具可以自动检测不正确的可追溯性依赖关系。所有版本异常信息都显示在主屏幕右侧浏览器的“版本分析”条目下，并在追溯图中的红色依赖线中突出显示。例如，由于低版本代码片段依赖于地址簿系统中的高版本序列图元素，根据命题5.3.3，模型元素S13和模型元素B8之间的依赖关系为不正确。最后，我们的工具可以分析其他异常，例如孤立的模型元素和不正确的依赖方向，这些异常列在主屏幕右侧浏览器的其他分析条目下。

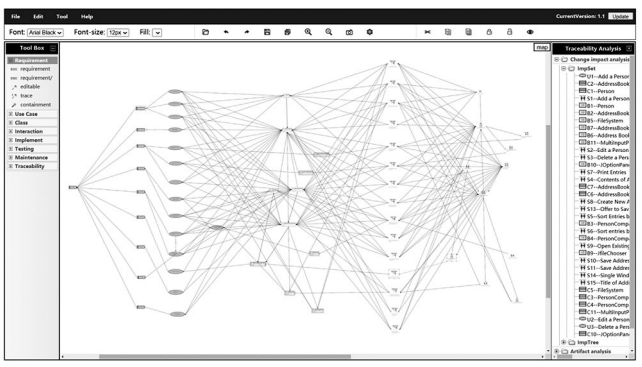

图5 变更影响分析

Fig.5 Changing impact analysis

7 相关工作(Related work)

软件追溯管理问题得到了广泛研究[2−3]，大多数研究[14−15]使用非形式化方法来探索如何管理软件生命周期过程中的可追溯性。我们的工作涉及形式化方法，该方法有助于为软件制品可追溯性提供严格的语义，以便对可追溯性进行建模、设计和推理。因此，我们在这里只讨论最相关的作品。WEN等人[11]使用称为行为树[16]的形式化模型来表示功能需求，以确保变更管理的完整性并自动获取记录设计变更历史的进化设计文档。在文献[17]中，ERATA等人介绍了一种新方法及称为Tarski的支持平台，该方法以交互方式表示可追溯性的语义，并使用户能够根据项目特定需求严格配置系统。GOKNIL等人[18]提出了需求元模型，包括具有在一阶逻辑中指定的形式语义的关系类型。BROY[5]区分了逻辑制品内容、制品内容表示及其物理表示，并定义了制品内容块的概念。在上面提到的工作中，其可追溯性语义很难处理软件制品的多对多关系，而我们的方法没有限制。

在文献[19]中，GOKNIL等人提出了一种在工具支持下生成和验证需求及架构之间跟踪的方法。DRIVALOS等人[20]提出了一种可追溯性元建模语言(TML)，该语言用于在高抽象级别上构建和维护严格的可追溯性元模型。PAIGE等人[21]提出了一种在基于模型的工程中识别、定义和实现语义丰富的跟踪链接的方法。DI等人[6]提出了一种基于模糊逻辑自动生成需求可追溯性矩阵的方法，该方法用于处理那些可能对需求可追溯性的确定产生负面影响的不确定性。SCHWARZ等人[22]介绍了基于图的方法来形式化和实现软件工程项目中的可追溯性信息，旨在实现基于需求的软件重用。SEIBEL等人[23]提出了一种综合的可追溯性方法，该方法针对模型驱动的工程和全局模型的能力方法，以动态分层的大型模型的形式进行管理。HOLTMANN等人[24]在基于模型的开发上下文中给出了可追溯性术语的定义，并开发了一组术语，使我们能够描述如何使用可追溯性，以及跟踪链接具有哪些属性。这些研究与我们的工作完全不同。

8 结论(Conclusion)

我们展示了如何在各种管理操作下保证可追溯性的正确，并提出了使用形式化方法构建、分析和可视化整个软件生命周期过程中的可追溯性的新的解决方案。我们开发了一个支持工具，用于促进不同形式的自动化分析，例如变更影响、制品和版本分析。该工具还可以帮助工程师半自动地建立和维护可追溯性信息。在本文中，由于我们关注的是管理追溯性的正确性，因此没有考虑如何建立一个存储库来存储追溯信息，但是我们的工具采用了一些节省存储空间的解决方案。在未来的工作中，我们将探索如何更有效地存储软件制品和可追溯性信息，并完善JLTool工具，希望使其成为真正的CASE工具。

参考文献(References)

[1] LAGO P, MUCCINI H, VAN V H. A scoped approach to

traceability management[J]. Journal of Systems and Software, 2009, 82(1):168−182.

[2] CHARALAMPIDOU S, AMPATZOGLOU A, KAROUNTZOS E, et al. Empirical studies on software traceability: A mapping study[J/OL]. Journal of Software: Evolution and Process, 2021:e2294 (2021−2−14) [2022−1−1]. https://doi.org/10.1002/smr.2294.

[3] 翟宇鹏,洪玫,杨秋辉.功能需求到测试用例的可追溯性研究[J].计算机科学,2017,44(Z11):480−484.

[4] MARO S, STEGHOFER J P, KNAUSS E, et al. Managing traceability information models: Not such a simple task after all?[J]. IEEE Software, 2021, 38(5):101−109.

[5] BROY M. A logical approach to systems engineering artifacts: Semantic relationships and dependencies beyond traceability—from requirements to functional and architectural views[J]. Software and Systems Modeling, 2018, 17(2):365−393.

[6] DI T A, RIBEIRO T, OLIVATTO G, et al. An automatic approach to detect traceability links using fuzzy logic[C]// IEEE. 2013 27th Brazilian Symposium on Software Engineering. Brasilia, Brazil: IEEE, 2013:21−30.

[7] MURATA T. Petri nets: Properties, analysis and applications[J]. Proceedings of the IEEE, 1989, 77(4):541−580.

[8] BOLOGNESI T, BRINKSMA E. Introduction to the ISO specification language LOTOS[J]. Computer Networks and ISDN Systems, 1987, 14(1):25−59.

[9] NIELSEN M, ROZENBERG G, THIAGARAJAN P S. Elementary transition systems[J]. Theoretical Computer Science, 1992, 96(1):3−33.

[10] SPANOUDAKIS G, ZISMAN A. Software traceability: A roadmap[C]// CHANG S K. Handbook of Software Engineering and Knowledge Engineering. Singapore: World Scientific Publishing Co., 2005, 3:395−428.

[11] WEN L, TUFFLEY D, DROMEY R G. Formalizing the transition from requirements' change to design change using an evolutionary traceability model[J]. Innovations in Systems and Software Engineering, 2014, 10(3):181−202.

[12] REMPEL P, MÄDER P. Preventing defects: The impact of requirements traceability completeness on software quality[J]. IEEE Transactions on Software Engineering, 2017, 43(8): 777−797.

[13] HUANG Y, JIANG J, LUO X, et al. Change−patterns mapping: A boosting way for change impact analysis[J]. IEEE Transactions on Software Engineering, 2022, 48(7): 2376−2398.

[14] ANQUETIL N, KULESZA U, MITSCHKE R, et al. A model−driven traceability framework for software product lines[J]. Software and Systems Modeling, 2010, 9(4):427−451.

[15] 邓刘梦,沈国华,黄志球,等.扩展SysML支持需求追踪模型的自动生成[J].计算机科学与探索,2019,13(6):950−960.

[16] DROMEY R G. From requirements to design: Formalizing the key steps[C]// IEEE. In 1st International Conference on Software Engineering and Formal Methods (SEFM 2003). Brisbane, Australia: IEEE, 2003:2−11.

[17] ERATA F, CHALLENGER M, TEKINERDOGAN B, et al. Tarski: A platform for automated analysis of dynamically configurable traceability semantics[C]// ACM. 32nd Annual ACM Symposium on Applied Computing. Marrakech, Morocco: ACM, 2017:1607−1614.

[18] GOKNIL A, KURTEV I, BERG K V D, et al. Semantics of trace relations in requirements models for consistency checking and inferencing[J]. Software and Systems Modeling, 2011, 10(1):31−54.

[19] GOKNIL A, KURTEV I, VAN D B K. Generation and validation of traces between requirements and architecture based on formal trace semantics[J]. Journal of Systems and Software, 2014, 88:112−137.

[20] DRIVALOS N, KOLOVOS D S, PAIGE R F, et al. Engineering a DSL for software traceability[C]// Software Language Engineering(SLE). International Conference on Software Language Engineering. Berlin, Heidelberg: Springer, 2008:151−167.

[21] PAIGE R F, DRIVALOS N, KOLOVOS D S, et al. Rigorous identification and encoding of trace−links in model−driven engineering[J]. Software and Systems Modeling, 2011, 10(4):469−487.

[22] SCHWARZ H, EBERT J, WINTER A. Graph−based traceability: A comprehensive approach[J]. Software and Systems Modeling, 2010, 9(4):473−492.

[23] SEIBEL A, NEUMANN S, GIESE H. Dynamic hierarchical mega models: Comprehensive traceability and its efficient maintenance[J]. Software and Systems Modeling, 2010, 9(4):493−528.

[24] HOLTMANN J, STEGHÖFER J P, RATH M, et al. Cutting through the jungle: Disambiguating model−based traceability terminology[C]// IEEE. 2020 IEEE 28th International Requirements Engineering Conference (RE). Zurich, Switzerland: IEEE, 2020:8−19.

作者简介：

李建清(1993−)，男，硕士生.研究领域：软件开发方法和形式化方法.

蒋建民(1972−)，男，博士，教授.研究领域：软件开发方法和形式化方法.