使用启发式规则的本体冲突检查方法

章少雷，吴毅坚+，陈叶旺，彭鑫，赵文耘

（复旦大学计算机科学技术学院 上海 200433）

+ 通讯作者：wuyijian@fudan.edu.cn

摘 要：本体(Ontology)作为语义 WEB 的核心已经被应用到社交，农业等多个领域。将本体开放给众多用户进行编辑的做法已经开始流行，但是在构建多用户本体系统时仍然存在一些问题。本体演化就是其中之一，在共同协作的本体环境中总是充斥着各种冲突，所以冲突检查是本体演化中重要的一个环节，本文将提出一个实用的基于启发式的方法来进行本体冲突检查，此方法是一个稳定的保证较高查全率和查准率的方法，并将给出演化中的冲突检查算法。

关键词：本体；协同构建；演化；冲突检查；语义 WEB

基金项目：国家“八六三”高技术研究发展计划(2007AA01Z179)、国家自然科学基金（90818009）、上海市科学技术委员会（08DZ2271800，09DZ2272800）

作者简介：章少雷，男，1984 年生，硕士研究生，研究方向为软件资源库；吴毅坚（通讯作者），男，1979 年生，博士、讲师，研究方向为软件工程；陈叶旺，1980 年生，男，博士研究生，研究方向为软件资源库；彭鑫，男，1979 年生，博士、讲师，研究方向软件工程；赵文耘，男，1964 年生，教授，博士生导师，研究方向软件工程。

A Method for Ontology Conflict Checking Based on Heuristic Rules

ZHANG Shao-lei, WU Yi-jian，CHEN Ye-wang, PENG Xin, ZHAO Wen-yun

[1]（*School of Computer Science, Fudan University, Shanghai 200433,China*）

Abstract: As a core of semantic web, ontology has been used in many different domains such as sociality, agriculture and so on. It’s being popular to open construction platform of ontology to public users, but there are many conflicts when users edit their ontology concurrently. Therefore, ontology conflict checking becomes an important problem in ontology evolvement. This paper will propose a heuristic method to solve this problem, and this method is a steady way to ensure recall ratio and precision ratio, and an algorithm will be proposed in this method.

Key words: Ontology, Collaborative construction, Ontology evolvement, Conflict checking, Semantic Web

1. 介绍

本体（Ontology）作为语义 Web 应用的基础技术之一，已经被广泛应用于特定领域知识建模以及基于知识的资源标注和检索。另一方面，集中式的开发方式也难以适应这些本体的规模要求。因此，基于广泛协作的分布式本体开发逐渐受到广泛关注。在这种本体开发模式下，广泛分布的领域知识专家共同参与本体模型的构建。在这样的环境下，协同冲突不可避免的成为了一个关键问题。

本文试图提供一个简单而实用的方法来进行冲突检查，并且作为知识管理员可以在此方法中进行自定义的冲突检查，本文提出的方法可以应用于大规模的本体知识开放构建应用中。

本文第二部分将介绍本体演化的相关工作；第三部分将介绍本文的系统架构和相关的基础知识；第四部分将阐述冲突检查方法；第五部分介绍本文作者开发的冲突检查工具；第六部分为总结及未来的工作。

2. 相关工作

这个部分将介绍本体演化的相关工作。文献[1]介绍了

当前流行的众多本体工具，并从应用性，方法学，互操作性和易用性方面对这些工具进行了对比阐述。文献[7]介绍了一个本体构建工具以及在工具中进行本体演化的关键的冲突检查问题，并且使用操作对本体编辑进行封装，然后通过每个操作的影响范围计算操作之间的相互影响系数来发现多用户协同构建中的本体冲突。文献[16]对已有本体演化方法进行了全面的调查和阐述，其中阐述本体演化分为六个步骤，变更捕捉，变更表示，语义变更发现，变更实现，变更传播和变更验证。文献[8]将本体构建中的冲突分为硬冲突，软冲突及潜在冲突，针对不同的形式来找出不同的冲突。文献[9]从本体结构出发，通过先定位两个版本相同的资源，然后基于这些相同资源对与这些资源相关的资源进行比较，来确定不同的部分是添加，修改还是删除。文献[11]给出了本体变更的一套完整的解决方案，提出了本体的融合方法，通过形式化的方式设计了一些公理，然后检查这些公理之间是否存在不一致性。文献[3]也介绍了一种本体演化的策略来保持本体一致性并且阐述了以 OWL 为本体描述基础的语义变更的模型。文献[10]阐述了在本体变更问题上的一些可能权衡的地方，比如是进行周期性的版本审核，还是连续性的进行版本编辑，是加入审核机制还是任由本体自由演化，最后文献介绍了相应的一些本体演化工具。文献[14]介绍了一种将本体映射为图的演化方案，通过对图进行操作，完成基于本体的演化工作。文献[15]提供了一种本体演化的框架和模型，希望通过此模型来解决领域知识动态变化中的演化和维护问题。文献[4]阐述了一种冲突检查的算法，通过计算每个本体编辑操作的影响范围，然后根据影响范围来计算两个操作的相互影响度，然后判断影响度是否超过预定义的阈值来判断操作之间是否产生冲突。文献[5]将本体编辑操作分为硬冲突，软冲突和潜在冲突，通过不同的检查方式可以检查出不同的冲突。文献[17]将本体的演化用图来进行表示，图的节点是演化中的本体，而图的边则代表从前一个版本到后一个版本演化所进行的变更。文献[18]引入了一种免疫理论到本体演化中，在演化框架中加入了免疫控制层，当免疫控制层发现本体当中有冲突时，它会按照一些相关规则来去除这种冲突，使整个本体保持一致。文献[19]提出了一种本体演化中有效减小变更影响范围的算法，并给出了影响范围的量化公式，有效的了减小演化时对本体的改变范围。

上述这些方法都提供了针对本体演化或变更的相关解决方案，但是它们都缺乏实际的针对大规模本体数据的演化方案，并且在冲突检查方法中引入了一些数值计算而导致了查准率的不稳定。本文将提出一种自启发式的冲突检查方法，来解决在大规模本体数据中时常发生冲突的问题，并且此方法是基于匹配算法，可以保证方法的稳定性。

3. 概述

3.1 架构

图 1 展示了本文的本体演化方案的架构，这个架构由多层结构组成。首先是客户层，这一层由多个客户机组成，用户在自己的终端（PC，手机等）进行本体编辑，然后从终端提交操作给门户层。门户层提供服务客户端的门户，这一层由多个服务器组成。

门户层在接收到用户编辑之后，将相关数据传送到本体演化层，使用操作包装器将这些编辑包装成操作(Command)，然后存入用户独有的工作库里面，在这个阶段用户之间的工作库是互相独立的，用户可以在自己的工作库里面做任何编辑。在经过一段时间之后，本体管理员认为可以生成新的本体版本，就会开启本体演化过程，在这个过程开始后，客户层用户的编辑行为将被禁止。冲突检查器从各个用户的工作库里面拿到每个用户的操作，然后为每个用户检查冲突，这一步是为了保证每个用户自己工作库里的编辑是不冲突的，这一步冲突检查结束后，操作将被存回自己的工作区供冲突检查器使用。

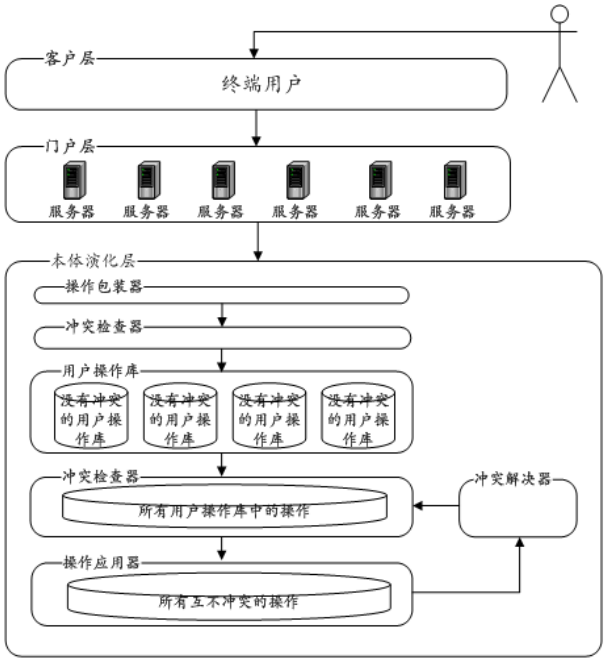

图 1 - 协同构建系统架构

Fig 1: Collaborative construction system architecture

冲突检查器是本文主要阐述的内容，它是本体演化的主要步骤之一，它从每个用户工作库提取出所有用户操作，然后将冲突发布给冲突解决器，冲突解决器在用户管理员协助下进行冲突解决，然后冲突检查器再进行冲突检查，然后再将冲突发布给冲突解决器，这个过程不断重复直到没有冲突。这个过程将所有用户操作融合在一起，这些操作被传给操作应用器。

操作应用器将所有的互不冲突的操作应用到上一个版本的本体上，然后生成新版本。每个操作的应用都只会涉及到单个操作影响的本体资源，而不需要读取所有本体数据。

整个架构不仅允许用户可以自由的进行编辑，并且也不需要读取大规模的本体数据。在实践方面具有一定的意义。

3.2 操作（Command）

在上面的架构中，所有的用户编辑都将被封装成操作（Command），然后被应用到本体上面。每个操作封装了用户每一次编辑行为的相关信息。每个操作包括了用户标识，实际进行的本体编辑行为，以及相应参数。

定义 1：定义操作 Command<U, Action, P>，其中，U 是做此编辑行为的用户，Action 指用户编辑的实际行为，P<P1, P2, ...>是一个集合，包含了操作所影响的所有知识资源。

表 1 列出了 8 个关于 Ontology Class 的操作定义。除了表 1 的 8 个操作之外，用户可以进行很多其他的本体操作，如 AddDatatypeProperty, SetObjectPropertyValue 等。

Command Name	Action	P
AddClass	添加一个本体类	P1:OntClassName
RemoveClass	删除一个本体类	P1:OntClassName
AddDisjointClassRelation	为两个本体类添加不相交类关系	P1:SuperClassName, P2:SubClassName
RemoveDisjointClassRelation	将两个本体类的不相交关系去掉	P1:OntClassName1, P2:OntClassName2
AddEquivalentClassRelation	为两个本体类添加相等关系	P1:OntClassName1, P2:OntClassName2
RemoveEquivalentClassRelation	删除两个本体类的相等关系	P1:OntClassName1, P2:OntClassName2
AddSuperClassRelation	添加两个类的父子关系	P1:SuperClassName1, P2:SubClassName2
RemoveSuperClassRelation	删除两个类的父子关系	P1:SuperClassName1, P2:SubClassName2

表 1 - 部分 Command 操作

Table 1: Some commands

下面将给出本体编辑的示例。

示例 1：用户执行一个 AddClass (Animal)操作和一个 AddClass(Plant)操作的时候，将添加两个本体，如图 2；然后用户执行一个 AddDisjointClassRelation(Animal, Plant)，就添加了一个不相交关系，如图 3：

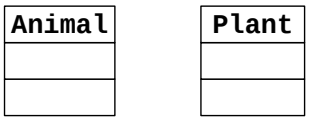

图 2 - 添加类

Fig2: Add classes

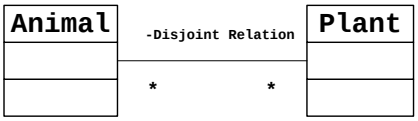

图 3 - 添加类关系

Fig3: Add class relationship

当用户执行一个 RemoveClass(Animal)操作时，之前 AddDisjointClassRelation 操作添加的不相交关系也被同样取消了，所以编辑模块会自动加入一个操作 RemoveDisjointClassRelation(Animal, Plant)。依上例所述，用户的编辑行为都可以转换成 Command 操作。

3.3 冲突

多个用户的协同编辑会产生很多的冲突，这里的冲突是因为多个用户在针对相同或者相互联系的资源定义时所产生不同意见而发生的，示例 2 给出了一个多用户进行本体编辑的例子。

示例 2：以图 3 为基础，用户 1 做了两个操作 AddClass(OgreishFlower)和 AddSuperClassRelation(Plant)将 OgreishFlower(食人花)定义为了 Plant(植物)的子类。而用户 2 此时做了一个操作 AddSuperClassRelation(Animal)将食人花定义为了动物，如图 4：

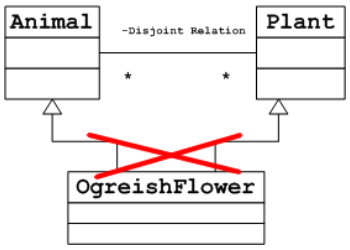

图 4 - 发生语义冲突

Fig4: Semantic conflict

食人草被分别定义为动物和植物，而动物和植物又被定义为两个完全不相同的概念。这种冲突在两个或两个以上用户编辑时会频繁出现，而所有由多个用户编辑而产生的冲突，本文称为协同构建冲突。

定义 2（协同构建冲突）：由多个（两个或两个以上）用户提交本体编辑操作作用于本体上，产生相反或不符合本体语义限制的结果，称为协同构建冲突。

单个用户在进行编辑时，也可能产生一些自相矛盾的操作，比如 RemoveClass(A)和 AddDisjointClassRelation(A, B)，当 A 已被删除后，而又添加 A 的不相交关系。这种由单个用户提交到工作库里而产生的操作，本文称为单用户冲突。

定义 3（单用户冲突）：由一个用户在编辑时产生的操作作用于本体上，产生相反、不符合本体语义限制或前后不一致的结果，称为单用户冲突。

本文将在后面详细阐述单用户冲突检查和协同构建冲突检查的方法，并阐述这两种冲突检查在整个冲突检查过程中的作用。

4. 冲突检查方法

4.1 单用户冲突检查和协同构建冲突检查

单用户冲突检查是为了保证每个用户工作库里面的操作互不冲突。冲突检查系统将在每次用户提交操作时检查操作是否和工作库里的操作发生冲突，如果发生，将不允许用户提交，用户可以做相应修改之后，直到不与之前的操作冲突才能完成提交。本文的冲突检查流程如图 5：

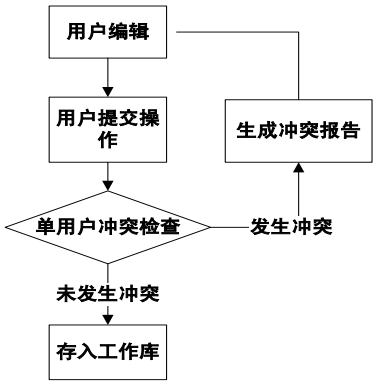

图 5 - 单用户冲突检查流程

Fig5: Checking process for single-user conflicts

当编辑行为进行了一段时间，本体管理团队就可以暂停本体编辑，开始本体审核以生成新的本体版本。所以就启动协同构建冲突检查，协同构建冲突检查只需要检查多个用户之间的操作冲突。在协同冲突检查过程中，同一个用户的操作之间可能会产生前后依赖的关系，所以当前面的操作产生冲突的时候，后面的操作将被锁定（Lock），直到这个冲突被解决，协同冲突检查的流程如图 6：

协同冲突检查是一个重复的过程，这个过程会持续到所有本体操作不发生冲突为止。

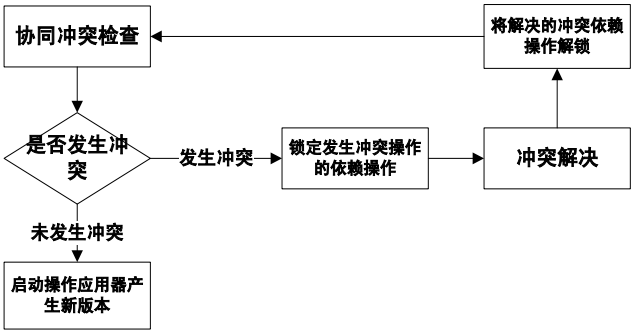

图 6 - 协同冲突检查流程

Fig6: Checking process for collaborative conflicts

4.2 操作冲突表

设计操作冲突表的目的是将语义含义赋予给所有操作，让冲突检查系统可以明白操作之间是否产生冲突。因为本文之前提过冲突只会发生在相关资源上，所以冲突表也是在相关资源的基础上建立起来的。冲突表由本体管理团队定制，定制后系统就会自动检查出符合要求的冲突。

因为需要检查单用户冲突和协同构建冲突，所以两种冲突表是不一样的，这就需要在冲突表里面体现。而本体是用统一资源标识符来标识惟一的资源，所以冲突表给出的冲突也是基于相同或相关 URI。

定义 4（冲突表）：冲突表 Conflict<Context, Ci, Cj, Handle>定义了当两个操作(Ci, Cj)同时发生时，根据 Ci，Cj 的不同组合情况(Handle)以及当前用户情况(Context)而采取的不同处理措施。

Context 有四种：

相同 URIs，相同用户

有重叠 URIs，相同用户

相同 URIs，不同用户

有重叠 URIs，不同用户

Handle 表示处理操作的方法，有 6 种：

Impossible - Ci, Cj 不可能同时发生

Merge - Ci, Cj 可以融合成一个操作 C

Conflict - Ci, Cj 产生明显冲突，需要冲突解决

Be Determined - Ci, Cj 还需要进一步检查

Need to Adjust - Ci, Cj 需要调整顺序进行应用

Not Conflict - Ci, Cj 不发生冲突

根据 Context 的四种情况，需要建立四个冲突表，下给出这四个冲突表。

词汇	实际操作
C1	AddClass
C2	RemoveClass
C3	AddDisjointClass
C4	RemoveDisjointClass
C5	AddEquivalentClass
C6	RemoveEquivalentClass
C7	AddSuperClass
C8	RemoveSuperClass

表 2 - 操作词汇表

Table2: Command vocabulary

词汇	Handle
IMP	Impossible
MG	Merge
BD	Be Determined
CF	Conflict
NTA	Need to Adjust
NC	Not Conflict

表 3 - Handle 词汇表

Table3: Handle vocabulary

	C1	C2	C3	C4	C5	C6	C7	C8
C1	MG	NTA	NC	NC	NC	NC	NC	NC
C2	BD	MG	CF	BD	CF	BD	CF	BD
C3	NC	CF	MG	NTA	CF	NC	CF	NTA
C4	NC	NTA	NTA	MG	NTA	NC	NTA	NC
C5	NC	CF	CF	NTA	MG	CF	CF	NC
C6	NC	BD	NC	NC	CF	MG	NC	NC
C7	NC	CF	CF	NTA	CF	NC	MG	CF
C8	NC	BD	NTA	NC	NC	NC	CF	MG

表 4 - 相同 URIs，相同用户冲突表

Table4: Conflict table of same URIs from same user

	C1	C2	C3	C4	C5	C6	C7	C8
C1	/	/	NC	NC	NC	NC	NC	NC
C2	/	/	CF	BD	CF	BD	CF	BD
C3	NC	CF	BD	BD	BD	NC	BD	NC
C4	NC	NTA	BD	NC	NC	NC	NC	NC
C5	NC	CF	BD	NC	BD	BD	BD	NC
C6	NC	BD	NC	NC	BD	NC	NC	NC
C7	NC	CF	BD	NC	BD	NC	BD	BD
C8	NC	BD	NC	NC	NC	NC	BD	NC

注：/表示和表 3 情况一样

表 5 - 重叠 URIs，相同用户冲突表

Table5: Table of overlapped URIs from same user

	C1	C2	C3	C4	C5	C6	C7	C8
C1	MG	IMP	NC	NC	NC	NC	NC	NC
C2	IMP	MG	CF	NTA	CF	NTA	CF	NTA
C3	NC	CF	MG	CF	CF	NC	BD	NC
C4	NC	NTA	CF	MG	NC	NC	NC	NC
C5	NC	CF	CF	NC	MG	CF	CF	NC
C6	NC	BD	NC	NC	CF	MG	NC	NC
C7	NC	CF	BD	NC	CF	NC	MG	CF
C8	NC	BD	NC	NC	NC	NC	CF	MG

表 6 - 相同 URIs，不同用户冲突表

Table6: Table of same URIs from different users

	C1	C2	C3	C4	C5	C6	C7	C8
C1	/	/	NC	NC	NC	NC	NC	NC
C2	/	/	CF	BD	CF	BD	CF	BD
C3	NC	CF	MG	BD	BD	NC	BD	NC
C4	NC	NTA	BD	MG	NC	NC	NC	NC
C5	NC	CF	BD	NC	MG	BD	BD	NC
C6	NC	BD	NC	NC	BD	MG	NC	NC
C7	NC	CF	BD	NC	BD	NC	BD	BD
C8	NC	BD	NC	NC	NC	NC	BD	MG

注：/表示和表 5 情况一样

表 7 - 重叠 URIs，不同用户冲突表

Table7: Table of overlapped URIs from different users

操作冲突表可以作为初步判断冲突的依据，根据不同的情况 (Handle)可以做相应的不同处理。初步判断可以剔除很多明显的冲突，以及允许明显不冲突的操作通过检查。下一部分将详细阐述每种冲突情况的处理方法。

4.3 处理方法(Handle)

本部分将阐述如何来处理 6 种冲突情况的处理方法。

4.3.1　处理 Impossible 情况

Impossible 情况一般较少出现，这种情况表示两个操作是不可能同时发生的，如上面表 5 中 Conflict(C1, C2)=IMP，在用户执行 AddClass 添加了一个独有的类(如 http://uri_prefix/unique_class1)后，另一个用户又执行了一个 RemoveClass 删除同样 URI 的类，但是之前第二个用户又没有添加过该类，所以这样的情况是不可能同时发生的。

如果同时发生，冲突检查器会怀疑客户端没有使用合法的编辑器，它会拒绝所有该操作来源(提交此操作的用户)所提交的所有操作。

4.3.2　处理 Merge 情况

Merge 情况主要是因为用户的重复操作，用户重复的定义了相同的概念，并进行了提交，如表 3 中 Conflict<C1, C1>=MG，表示同一个用户同时添加了同样名称的类，所以两个 AddClass 只需要合并为一个 AddClass 即可。

4.3.3　处理 Not Conflict 情况

这种情况下表示两个操作并不冲突，所以冲突检查器滤过这两个操作，进行下一个检查。

4.3.4　处理 Conflict 情况

这种情况下表示两个操作发生了明显的冲突，如表 6 中 Conflict<C2, C7>=CF，第一个用户执行了 RemoveClass 操作删除了一个类，而第二个用户执行了 AddSuperClass 在删除的类上面添加了一个父子关系，而第一个用户又要删除它，这样就产生了冲突。

当这样的情况发生时，冲突检查器直接将操作放入冲突报告，并锁定(Lock)依赖此冲突的操作的所有操作，然后等待冲突解决器解决，本文将在 4.4 阐述操作间的依赖关系。

4.3.5　处理 Need to Adjust 情况

Need to Adjust 表示两个操作可能会产生冲突，但是在以特定顺序执行的时候是没有冲突的，如表 5 中 Conflict<C2, C6>=NTA，在 RemoveEquivalentClass 先执行，而 RemoveClass 后执行的时候不会产生冲突，相反如果掉转过来执行就会出现冲突。在此情况下，操作应用器将以不冲突的顺序应用这两个操作。

4.3.6　处理 Be Determined 情况

Be Determined 是冲突当中最复杂的情况，需要引入冲突形式化库来判断两个冲突是否发生冲突，后面本文将在 4.5 详细阐述冲突形式化库。

4.4 操作依赖表

单个用户的操作之间总是存在一些依赖关系，后续的操作会建立在前面操作的结果之上，当前面的操作发生冲突时，后续的操作会因为前面的操作处理不同也产生不同的处理。如两个操作 AddClass，AddDisjointClassRelation，在 AddDisjointClassRelation 操作是依赖 AddClass 操作的，如果当 AddClass 操作被系统拒绝之后，AddDisjointClass 也没有意义了，所以也要被拒绝掉。

定义 5（依赖表）：依赖表 Dependency <Ci, Yi>=isDependent，表示 Ci 是否依赖 Yi，如果依赖 isDependent 等于 Y，否则等于 N。操作依赖表帮助处理上述的 Conflict 情况。

	Y1	Y2	Y3	Y4	Y5	Y6	Y7	Y8
C1	/	N	N	N	N	N	N	N
C2	Y	/	N	N	N	N	N	N
C3	Y	N	/	N	N	N	N	N
C4	Y	N	Y	/	N	N	N	N
C5	Y	N	N	N	/	N	N	N
C6	Y	N	N	N	Y	/	N	N
C7	Y	N	N	N	N	N	/	N
C8	Y	N	N	N	N	N	Y	/

表 8 – 8 种操作的单用户操作依赖表

Table8: Dependency table of eight commands

4.5 冲突的形式化

当遇到两个操作难以判断是否冲突的时候，就需要进一步的对两个操作做判断，引入冲突的形式化定义是一个可行的方法。每一种冲突都有自己的模式存在，以形式化的方式定义这个模式，就是冲突的形式化定义。

定义 6（冲突的形式化）：定义 Conflict_Pattern< CommSet | ConflictCondition | Handle>，CommSet 是一个集合，定义了冲突所包括的所有操作，ConflictCondition 定义了冲突成立的所有条件，而 Handle 定义了冲突匹配时需要采用的处理方法。

示例 3：在表 6 中，Conflict<C7, C7>=Be Determined，也就是示例 2 的情况，当两个 AddSuperClass 在不同用户之间发生，在冲突表里面无法直接判断它们是否冲突，所以就应该引入冲突形式化库，冲突形式化库中有一个冲突形式化定义为：

```
Conflict_Pattern1
<AddSuperClass a1,
 AddSuperClass a2,
 AddDisjointClass a3 |
 a1.subClassName=a2.subClassName &&
 a3.disjointClassName1=a1.superClassName &&
 a2.disjointClassName2=a2.superClassName |
 Conflict>
```

根据此条形式化冲突规则，在操作库中如果找个一个 AddDisjointClass 操作，并且和这条冲突是匹配的，说明冲突发生，所以按照 Conflict 处理方法，将 3 个冲突都加入到冲突报告，等待冲突解决。反之没有相应的操作，说明两个 AddSuperClass 操作是不冲突的，使用 Not Conflict 方法来处理。

冲突形式化定义需要精心的定义，可以在冲突形式化库中加入任意的冲突形式化规则，这样不仅使冲突检查更灵活，也可以使冲突检查器按照本体管理者的意愿去检查，而尽可能的避免了误查。以下是一些常用的冲突形式化规则：

示例 4：

```
Conflict_Pattern1:
<AddSuperClass a1,
 AddSuperClass a2,
 AddDisjointClass a3 |
 a1.subClassName=a2.subClassName &&
 a3.disjointClassName1=a1.superClassName &&
 a2.disjointClassName2=a2.superClassName |
 Conflict>

Conflict_Pattern2:
<AddDisjointClass a1,
 RemoveDisjointClass a2 |
 a1.disjointClassName1=a2.disjointClassName1 &&
 a1.disjointClassName2=a2.disjointClassName2 &&
 a1.user != a2.user|
 Conflict>

Conflict_Pattern3:
<AddDisjointClass a1,
 AddEquivalentClass a2 |
 a1.disjointClassName1=a2.equivalentClassName1 &&
 a1.disjointClassName2=a2.equivalentClassName2 |
 Conflict>
```

一般来讲，冲突形式化库越完善，冲突检查器检查出来的冲突就越准确。当然，冲突形式化库的扩大也会影响冲突检查的效率，所以本体管理团队需要在效率与查全率之间进行权衡。

4.6 冲突检查算法

这一部分将给出一个冲突检查算法，本文的算法是一个自启发式的方法，它通过一个用户的一个操作找出其它的冲

突操作，直到所有的操作都不发生冲突。这个算法同时试用于单用户冲突检查和协同冲突检查，不同处在于使用的冲突表以及在细节上的一些处理，所以下面的算法描述以协同构建冲突为主。

定义 7（用户工作库操作集合）：定义 UserCommSet <U1<O11, O12, O13, …, O1n1>, U2<O21, O22, O23, …, O2n2>, ……, Uw-1<O(w-1)1, O(w-1)2, O(w-1)3, …, O(w-1)n(w-1)>, Uw<Ow1, Ow2, Ow3, …, Ownw>>为用户工作库中的操作集合，Uw 表示用户标识，Owi 为第 w 个用户工作库中的第 i 个操作，nw 为每个用户工作库的操作数量。

算法 1 - 协同构建冲突检查算法

输入：用户工作库操作集合 UserCommSet

输出：按照顺序排列的不冲突的操作集合 CommSet

算法流程：

1) 初始 WSetLevel = 0 表示当前递归层数
   初始 lockNum=0 表示当前被锁定的操作数
2) 取出序号最低的用户工作库中最前面的未标识为不冲突并且不被锁定的操作 O1，如果 O1 不存在，执行步骤 10
3) 针对 O1，进行下列步骤
4) 将 O1 从所属工作库集合中删除，初始 WSet 为空集合
5) 锁定依赖于 O1 的所有操作，锁定一个操作，执行一次 lockNum=lockNum+1
6) 在其它工作区中的所有不被锁定的 Oi，使 Conflict<O1, Oi>!=Not Conflict，在工作库集合中删除所有 Oi，并将其加入 WSet
7) 如果 WSet 为空，标识 O1 为无冲突操作，加入到所属工作库集合中，如果 WSetLevel=0，执行步骤 2，否则 WSetLevel = WSetLevel-1，并执行步骤 9
8) 遍历 WSet 中所有操作 Oi，遍历结束后执行步骤 9：
9) 如果 Conflict<O1, Oi>=Impossible：
   删除 O1 和 Oi 所属的工作库，然后执行步骤 2
10) 如果 Conflict<O1, Oi>=Merge，融合 O1, Oi，将融合后的操作加入 O1 所属的工作库，并继续执行步骤 9
11) 如果 Conflict<O1, Oi>=Need to Adjust：
    调整 O1, Oi 执行顺序并标识两个操作的执行顺序，如 O1.precede=Oi，并继续执行步骤 9
12) 如果 Conflict<O1, Oi>=Conflict：
    将<O1, Oi>加入冲突报告，并继续执行步骤 9
13) 如果 Conflict<O1, Oi>=Be Determined：
    用算法 2 判断两个操作是否存在冲突，并找出冲突集合，将冲突集合加入冲突报告，并将冲突集合中为加入到 WSet 中的操作加入到 WSet 中，并继续执行步骤 9
14) 如果 WSet 为空，执行步骤 7，否则从 WSet 中任意取出一个操作 On，将 On 从 WSet 中删除，令 O1=On，执行步骤 3
15) 如果 lockNum>0，等待冲突解决器返回冲突解决结果，如果某个操作被拒绝，将依赖此操作的所有操作从工作库集合中删除，删除一个操作执行一次 lockNum--；如果操作被接受，将依赖此操作的所有操作解锁，解锁一个操作 lockNum--
    如果 lockNum=0，执行步骤 11
16) 按照算法 3 将工作库中所有标识为不冲突的操作加入 CommSet，输出 CommSet，算法结束

操作应用器使用 CommSet，按照其顺序应用操作到原有本体版本中，就可以生成新的本体版本了。

在处理 Be Determined 情况时，需要使用冲突形式化的模式匹配算法来判断两个操作的关系，下面给出冲突形式化匹配算法：

算法 2：冲突匹配算法

输入：一个操作集合 CSet(COMM1, COMM2, COMM3, …, COMMn)

输出：一个处理方法及其冲突集合(如果是 Conflict 的话)

算法流程：

1) 初始化冲突形式化库，将库中所有规则建立成以操作为 Key，规则集合为 Value 的映射 CommMap。（此步骤只需执行一次，第二次就从步骤 2 开始执行）
2) 遍历集合 CSet，每次取出一个操作 COMMi，遍历结束执行步骤 3：
   a) 从 CommMap 中以 COMMi 为 Key，取出形式化规则集合 CommSet
   b) 将 CommSet 中所有规则建立成以操作为 Key，规则集合为 Value 的映射 CommMapi
   c) 令 CommMap=CommMapi，并继续执行步骤 2
3) 如果 CommMap 为空，则 CSet 中的操作不冲突，输出 Not Conflict，算法结束；
   如果 CommMap 不为空，则执行步骤 4。
4) 遍历 CommMap，检查每一个形式化冲突规则的 Condition，如果有一个形式化冲突规则符合 Condition，则返回这个形式化冲突规则的 Handle(如果 Handle 为 Conflict，则返回相应的冲突集合)，算法结束。

当冲突检查基本结束之后，所有工作库里的操作将被按特定顺序被操作应用器应用到原来的本体版本中，所以在工作库中有一些操作之间有依赖关系，而有些操作之间通过检查器标识了前后应用的顺序，所以需要将这些操作排序，算法 3 给出了排序的算法。

算法 3：操作应用排序算法

输入：U - 所有用户工作库中的按时间顺序排列的操作集

合，U<

U1<Oi1，Oi2，Oi3，…，Oin>，

U2<Oj1，Oj2，Oj3,…，Ojm>，

U3<Ok1，Ok2，Ok3，…，Okl>，

… … ，

Uw<...>>

这些操作互不冲突，只存在依赖和标识前后执行的关系

输出：按照顺序排列的不冲突的操作集合 CommSet

算法流程：

1) 初始 CommSet 为空集合

2) 从序号最低的工作库中取出第一个操作 O1，如果 O1 不存在，执行步骤 5

3) 针对 O1 执行步骤 4

4) 如果 O1.precede=null，并且 O1 不存在依赖的操作，将 O1 加入 CommSet，并将 O1 从工作库中删除，并执行步骤 2

如果 O1.precede=Oi，令 O1=Oi，并执行步骤 3

如果 O1 依赖于操作 Od，令 O1=Od，并执行步骤 3

5) 输出 CommSet，算法结束

5. 工具实现及实验

本文根据上述的框架和算法创建了协同构建系统，这个系统包括了冲突检查模块。这个协同构建系统基于 B/S 架构，使用 Java1.5 和 MySQL 进行开发，支持 IE 和 Firefox 浏览器，它完整的控制了本体演化的整个周期。本部分 5.1 介绍该协同构建系统，5.2 展示了该系统中本体冲突检查的相关实验数据。

5.1 本体协同构建编辑器实现

图 7 展示了协同构建系统的编辑器，编辑器允许用户在本体审核阶段之前私有工作区做任意编辑，编辑器提供了几乎所有的本体编辑动作，用户可以灵活的编辑本体。

图 7 – 本体编辑器

Fig7: Ontology Editor

在协同管理器中，本体管理团队可以将本体平台置为开放状态，在此状态下，用户可以在编辑器做编辑。隔一段时间之后，管理团队可以将平台状态置为关闭状态，进入审核阶段。冲突检查器将生成冲突报告，供本体管理团队进行审核并解冲突，图 8 显示了冲突解决器。

图 8 – 冲突解决器

Fig8: Conflict resolver

管理团队可以在冲突解决器中进行所有冲突的解决，并且可以发起讨论主题与编辑者进行相关知识主题的讨论。

5.2 本体冲突检查实验

本实验将从效率和查准率上检验上述算法及解决方案，本体库类数量为 151257 个，本体大小 201,824KB。

本实验将递增预设的操作数量和冲突数量，以此来探寻冲突检测的时间与操作和冲突数量之间的关系，并且对查准率进行统计，预设的冲突包括直接在冲突表中检测到的冲突和通过形式化冲突定义而检测到的冲突。并且在预设冲突中，加入了冲突形式化定义未覆盖的冲突（每次加入 200 个未覆盖的冲突）。

实验运行环境如下：

CPU：英特尔酷睿双核，1.83GHz

内存：1.5G

硬盘：100G(5400rpm)

操作系统：Windows XP Professional SP2

本实验实现使用 Java 语言，运行在 JDK1.5 环境下，JVM 最大内存 768M，最小内存 128M，并基于 Jena 框架。为了使实验运行尽量准确，以及尽量避开运行中由于 Java 虚拟机回收资源带来的时间误差和硬件情况（温度，湿度等）所带来的扰动，本实验每个运行数据都要运行 20 次，并且去掉最高和最低的 2 个样本时间，最终时间来自于剩下的 16 个样本时间的平均值。

预设操作数	预设冲突数	检测到的冲突数量	检测时间(秒)	应用操作时间(秒)
300000	50000	49800	1085	15.22
600000	50000	49800	1818	26.68
900000	50000	49800	2103	35.97
1200000	50000	49800	2675	51.82
1500000	50000	49800	3481	71.81

表 9 – 递增操作数量的实验结果表

Table9 – Table for testing efficiency by increasing number of commands

从上面的实验数据可以看出，本文的算法是一个稳定的算法，只要在冲突表和形式化定义中定义的冲突，冲突检查器都可以检查出来，而定义中没有的 200 个冲突就无法检查出来，需要添加新的冲突形式化定义。而检测的时间与提交的操作数基本成正比关系，应用操作的时间也与提交的操作数成正比关系（后面当操作数较多时，时间稍高，可以认为是 JVM 垃圾回收导致的时间扰动）。

预设操作数	预设冲突数	检测到的冲突数量	检测时间（秒）	应用操作时间(秒)
750000	50000	49800	2050	36.90
750000	100000	99800	2545	43.09
750000	150000	149800	2285	35.04
750000	200000	199800	2275	40.01
750000	250000	249800	2528	44.07

表 10 – 递增冲突数量的实验结果表

Table10: Table for testing efficiency by increasing number of conflicts

上面的实验数据支持了前面的分析，当提交的操作数不变时，检测时间也基本相差不大，而应用操作的时间也基本保持在一个小的波动范围内。整个算法在处理 10 万级的操作和数量为 10 万级的本体时，冲突检查的效率是不错的。

6　总结和未来的工作

有效的本体演化管理是大规模领域本体开发的基本条件。本体模型不像软件代码一样存在清晰的逻辑层次结构，因此难以实现细粒度的版本和变更管理。另一方面，协同开发环境中不同参与者之间所存在的语法和逻辑冲突也会影响本体模型的一致性。针对这些问题，本文提供了一个新的方法来进行本体编辑中的冲突检查，本文建议将本体的编辑封装成操作，而将问题转化成检查操作之间的冲突，然后通过建立冲突表，依赖表以及冲突形式化模型来检查用户自己和用户之间的操作冲突。这种方法将大规模的本体检查转化成了较小规模的操作检查，提高了本体检查的效率，也增加了本体冲突检查的人性化和灵活性。在下一步工作中，我们将进一步对自动或半自动的冲突解决方法进行研究。

REFERENCES:

[1]. M. Denny. Ontology building, a survey of editing tools. Technical report. O'Reilly XML.com, November 06, 2002.

[2]. Erol Bozsak, etc. Kaon - towards a large scale semantic web. E-Commerce and Web Technologies, 2002, LNCS, Volumn 2455/2002, pp. 231-248.

[3]. N. F. Noy etc. Musen. Creating Semantic Web Contents with Protege-2000. Volumn 16, Issue 2, Pages 60-71. IEEE Intelligent Systems, 2001.

[4]. Yewang Chen, Shaolei Zhang, Xin Peng, Wenyun Zhao. A Collaborative Ontology Construction Tool with Conflicts Detection. The 4th International Conference on Semantics, Knowledge and Grid (SKG2008).

[5]. Yewang Chen, Xin Peng, Wenyun Zhao. An Approach to Detect Collaborative Conflicts for Ontology Development. The Joint International Conferences on Asia-Pacific Web Conference (APWeb) and Web-Age Information Management (WAIM), APWeb-WAIM 2009.

[6]. Clyde W.Holsapple, K.D. Joshi. A Collaborative Approach to Ontology Design. Communications of the ACM, Vol. 45, No. 2, Pages 42-47, 2002.

[7]. Michal Tury, M´aria Bielikov´a. An Approach to Detection Ontology Changes. ICWE'06 Workshops, July 10-14, 2006.

[8]. J. Bao, V.G. Honavar. Collaborative Ontology Building with Wiki@nt. Third International Workshop on Evaluation of Ontology Building Tools, Hiroshima 2004.

[9]. Peter Haase, Frank van Harmelen, Zhisheng Huang, Heiner Stuckenschmidt, and York Sure. A Framework for Handling Inconsistency in Changing Ontologies. ISWC 2005, LNCS 3729, pp, 353-367, 2005.

[10]. Web ontology working group. OWL Web ontology Language Overview. http://www.w3.org/TR/2003/PR---owl--features-20031215/, accessed Feb.02, 2004

[11]. Cohen, W. Ravikumar, P., Fienberg, S. . A comparison of string metrics for matching names and records. Proc. KDD-2003 Workshop on Data Cleaning and Object Consolidation, 2003

[12]. Natalya F. Noy, Abhita Chugh, William Liu, and Mark A. Musen. A Framework for Ontology Evolution in Collaborative Environments. ISWC 2006, LNCS 4273, pp. 544-558, 2006.

[13]. Tan Li. A graph based method for ontology evolvement, Proceedings of the 12th Annual National Academic Conference of China Artificial Intelligence Association, 2007, page 365. (in Chinese)

[14]. Wang Jin, Chen Enhong, Lin Le. An Ontology Evolution and Maintenance Model in Web Environment, Computer Science, Nov, 2003, Vol. 30, No. 12, page 126-page 129. (in Chinese)

[15]. Liu Bosong, Gao Ji. A Study on Ontology Evolution Management, Computer Science, May, 2005, Vol.31, No. 5, page 9-page 12. (in Chinese)

[16]. Peter Haase, York Sure. State-of-the-Art on Ontology Evolution. 2004 Institute AIFB, University of Karlsruhe.

[17]. Ljiljana Stojanovic, Alexander Maedche, Nenad Stojanovic, Rudi Studer. Ontology Evolution as Reconfiguration-Design Problem Solving. Proceedings of the 2nd international conference on knowledge capture, 2003. Pages 162-171.

[18]. Chengwei Zhang, Peng Liang, Mingyan Zhao. Research on Ontology Evolution Model Based on Immune Theory. Volumn 1, Pages 787-791 IEEE/SOLI 2008.

[19]. Liu Chen, Han Yanbo, Chen Wanghu, Wang Jianwu. MINI: An Ontology Evolution Algorithm for Reducing Impact Ranges. Chinese Journal of

Computers, May, 2008, Vol. 31, No. 5, page 711-page 720.(in Chinese)

附中文参考文献：

[13] 谭力. 一种基于图的本体演化方法, 中国人工智能学会第 12 届全国学术年会论文汇编, 2007 年, 365 页.
[14] 王进, 陈恩红, 林乐. 一种网络环境中的本体演化和维护模型, 计算机科学, 2003 年 12 月, 第 30 卷, 第 12 期, 126 页-129 页.
[15] 刘柏嵩, 高济. 本体演化管理研究, 计算机科学, 2004 年 5 月, 第 31 卷, 第 05 期, 9 页-12 页.
[19] 刘晨，韩燕波，陈旺虎，王建武. MINI-一种可减小变更影响范围的本体演化算法, 计算机学报 2008 年 5 月, 第 31 卷, 第 05 期, 711 页-720 页.