软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software,2015,26(9):2231−2249 [doi: 10.13328/j.cnki.jos.004735]
©中国科学院软件研究所版权所有.
E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel: +86-10-62562563

基于概念 R-MUPS 的本体调试方法*

欧阳丹彤[1,2], 苏 静[1,2], 叶育鑫[1,2,3], 崔仙姬[1,2,4]

[1](吉林大学 计算机科学与技术学院,吉林 长春 130012)

[2](符号计算与知识工程教育部重点实验室(吉林大学),吉林 长春 130012)

[3](国家地球物理探测仪器工程技术研究中心(吉林大学),吉林 长春 130026)

[4](大连民族大学 信息与通信工程学院,辽宁 大连 116600)

通讯作者: 叶育鑫, E-mail: yeyx@jlu.edu.cn

摘 要: 通过定义不可满足概念间的覆盖关系,发现 MUPS 和 MIPS 之间的内在关联,从而引出不可满足概念的 R-MUPS;给出并证明不一致本体术语集中至少存在一个不可满足概念的 R-MUPS 就是该术语集的 MIPS;利用这一结论,提出基于有序标签演算的 R-MUPS 算法,采用深度优先遍历原则合并分支计算 R-MUPS,同时缓存覆盖概念集合,加快 MIPS 的求解,实现本体调试.通过概念扩展树与概念 R-MUPS 算法的等价性,证明算法的正确性并分析其复杂度.最后,利用自动生成本体、现实本体及其扩建本体的数据进行全面测试.实验结果表明:基于 R-MUPS 的 MIPS 求解方法能够高效、准确地完成本体调试任务.

关键词: 本体调试;描述逻辑;定位;MUPS;MIPS

中图法分类号: TP181

中文引用格式: 欧阳丹彤,苏静,叶育鑫,崔仙姬.基于概念 R-MUPS 的本体调试方法.软件学报,2015,26(9):2231−2249. http://www.jos.org.cn/1000-9825/4735.htm

英文引用格式: Ouyang DT, Su J, Ye YX, Cui XJ. The ontology debugging method based on concept R-MUPS. Ruan Jian Xue Bao/Journal of Software, 2015,26(9):2231−2249 (in Chinese). http://www.jos.org.cn/1000-9825/4735.htm

The Ontology Debugging Method Based on Concept R-MUPS

OUYANG Dan-Tong[1,2], SU Jing[1,2], YE Yu-Xin[1,2,3], CUI Xian-Ji[1,2,4]

[1](College of Computer Science and Technology, Jilin University, Changchun 130012, China)

[2](Key Laboratory of Symbolic Computation and Knowledge Engineering of Ministry of Education (Jilin University), Changchun 130012, China)

[3](National Engineering Research Center of Geophysics Exploration Instruments(Jilin University), Changchun 130026, China)

[4](College of Information and Communication Engineering, Dalian Nationalities University, Dalian 116600, China)

Abstract: In this paper, an inherent relationship between MUPS and MIPS is obtained by defining covering relations of unsatisfiable concepts and developing the definition of R-MUPS of unsatisfiable concept. Then a proof is given that there exists at least one unsatisfiable concept whose R-MUPS is also a MIPS for every MIPS. Based on this conclusion, an R-MUPS algorithm based on the ordered label calculus is proposed. It applies depth-first traversal in merging branches to calculate R-MUPS, and also caches the concepts of covered at the same time to speed up the solving of MIPS to achieve ontology debugging. Through revealing the equivalence between concept expansion tree and R-MUPS algorithm, the correctness of R-MUPS algorithm is proved, and its complexity is further analyzed. Finally, a comprehensive test is performed using automatically generated ontology test sets, realistic ontology and its expansion ontology. The experimental results show that the solving of MIPS based on R-MUPS algorithm can efficiently and accurately complete the ontology debugging tasks.

* 基金项目: 国家自然科学基金(61272208, 61133011, 41172294, 61170092); 吉林省科技发展计划(201201011)
收稿时间: 2013-06-29; 修改时间: 2014-04-09; 定稿时间: 2014-09-19

Key words: ontology debugging; description logics; pinpoint; MUPS; MIPS

广义上,本体指各种形式化的表示,它包括分类、具有层次化结构的术语词汇以及基于某种逻辑理论的领域描述[1].人工智能学科下的本体特指对达成共识的领域概念的形式化描述[2].本体的构建一般由领域专家手工完成或计算机程序辅助半自动完成,建成的本体被广泛应用于计算机科学和工程的各个方面[3−5].在本体构建、使用、升级和维护的整个生命周期中,都离不开对其内涵知识不一致性的冲突消解,因此,本体调试(ontology debugging)工作成为本体工程中不可或缺的重要环节.本体调试对本体建模人员来说是一项巨大的挑战,尤其是随着本体语言描述能力复杂度的提高和本体规模的增长,纯手工完成对复杂本体语言或大规模本体的调试工作是不可能的[6].现有的标准本体推理任务和推理机具备判定逻辑是否不一致的推理能力,但无法提供有效的参考信息和帮助来解决或排查本体中的逻辑冲突.本体调试的研究就是期望提供有效的方法和手段,辅助本体建模人员实现逻辑冲突的快速定位和冲突解决方案的自动生成.

本体调试在确定和解决本体中逻辑冲突的过程中,一般分为诊断和修复两个阶段[7],人工智能中的自动推理技术更适于处理第一阶段的问题.这是因为自动推理领域的一个共识是:如何选择最理想的方案解决逻辑冲突,是领域专家需要考虑的问题.自动化系统并不具备代替领域专家完成本体修复阶段任务的能力,而更善于对逻辑冲突进行搜索和定位.因而本体调试的大多数研究工作实际上是研究逻辑冲突的诊断问题(即逻辑冲突的定位),而不是逻辑冲突的解决和修复.

理论上,本体中的逻辑冲突表现为 inconsistency 或 incoherency 两种不一致,其中,inconsistency 有时也泛指这两类冲突.根据 Flouris[8]的定义,这里将二者加以区分:inconsistency 指本体不存在任何模型,而 incoherency 特指本体中包含不可满足概念.Qi&Pan[9]提出了一种针对不一致本体的有效推理算法,并利用此算法实现 inconsistency 分层本体的调试;Meyer[10]将命题逻辑不一致管理技术加以改进并引入到描述逻辑本体,使用了 conjunctive maxi-adjustment 算法[11]实现对分层本体的调试过程的优化;Haase[12]提出使用本体进化方法避免本体的 inconsistency、使用特殊的推理技术忽略本体中的 inconsistency 或者使用版本控制技术间接处理本体的 inconsistency;Yue[13]利用超协调方法处理本体中的 inconsistency,达到调试本体的目的.

以上的工作均是处理 inconsistency 的本体调试.事实上,本体调试的大多数工作都集中在如何解决 incoherency[14].因为在本体的语境下,incoherency 是很重要的性质,不可满足概念的存在通常表明本体中概念的形式化定义是有问题的.这一类研究中,通常都借助 Tableau 演算定位本体中的 incoherency.Schlobach 等人[15−17]给出了概念不可满足和本体不一致原因的形式化定义,即,最小不可满足保持子术语集 MUPS 和最小不连贯保持子术语集 MIPS,并给出了扩展 Tableau 演算规则的不一致本体调试方法,但是只适用于非循环 $\mathcal{ALC}$ 不一致本体.Meyer[14]同样针对非循环 $\mathcal{ALC}$ 不一致本体计算不可满足概念的最大可满足术语集,并且做了大量的实验来分析方法的性能.Lam[6]扩展了 Schlobach 的工作,将 incoherency 从冲突公理集合进一步定位到冲突公理集合的某一局部,他将追踪 incoherency 的部分公理集的技术结合进来,提高了基于 Tableau 的本体调试算法的调试能力.Plesser[18]同样在 Tableau 演算的基础上构建概念依赖树得到 MUPS,这种方法更适用于本体进化过程.Kalyanpur[19−21]对同一本体中多个不可满足概念之间的关联性做了分析和研究,定义了不可满足概念间的依赖关系,针对一些特殊概念,结合碰集方法计算其辩护(比 MUPS 更宽泛的概念,MUPS 是辩护的特例),他们计算辩护的算法效率是比较高的.不修改 Tableau 演算,只调用外部推理机判断概念的可满足性实现本体错误的定位,因为不受本体语言和公理的类型限制,这种方法能够更好地应用于实际中.其中,Schlobach[15−17]可以计算单个 MUPS,加入启发式和模式匹配也不能保证得到所有的 MUPS[22],Kalyanpur[19−21]进一步地结合碰集树方法计算所有 MUPS.Kostyantyn[23]将模型诊断[24,25]中的基本定义和方法引入到本体调试中,给出了本体通用的诊断理论,提出一种不依赖于特定推理系统的正确完备求解最小不一致公理集的方法.文献[26,27]则对诊断使用二分策略以减少对本体蕴含的查询,能够快速给出目标诊断.另外一些工作致力于不同描述逻辑语言表示的本体的调试方法及其优化,本体模块化方法能够很好地提高现有本体调试算法的效率,尤其处理大规模本体数据.文献[28]使用本体模块化限制计算 OWL DL 本体不可满足概念的所有辩护过程的搜索空间.文献[29]提出一种基于

目标驱动的优化方法能够抽取出保证覆盖蕴含式所有辩护的模块,改进先前静态局部模块方法抽取模块规模大的弊端.文献[30]通过分析 DL-Lite 语言中不可满足概念或角色所具有的特点,针对 DL-Lite 本体给出计算概念 MUPS 的算法.

Haase 和 Qi[31]对本体调试研究进行系统性的分析和综述,对与调试直接或间接相关的方法进行了分类,并用各种标准对其进行较为严格的评价.通过分析和评测,他们认为:没有哪一种本体调试的方法是普遍适用于所有应用场景的,不同的方法是适合于不同调试目的和调试要求的.

就本体调试中的 incoherency 定位问题,本文针对描述逻辑中的非循环 $\mathcal{ALC}$ 本体术语集,借助 Tableau 演算推理方法研究不可满足概念与本体冲突的关联,提出基于概念 R-MUPS 的本体调试方法定位本体冲突.从问题定义来看,本文研究本体调试中的 incoherency 不一致冲突定位问题,而不是文献[9–12]的 inconsistency 不一致问题;从研究对象来看,本文研究针对非循环 $\mathcal{ALC}$ 本体的术语集(即 TBox)逻辑冲突定位,而不同于文献[25–27]中的具体本体模型(即特定 TBox 下的 ABox);从调试方法上来看,本文研究如何利用基于 Tableau 演算的推理机制(即白盒测试)直接解决 incoherency 的冲突定位,不同于文献[19–24]利用本体推理的标准问题结果或诊断方法(即黑盒测试)间接解决 incoherency 的冲突问题;在众多白盒测试方法的研究工作中,本文并不研究兼容于白盒测试的优化技术,如文献[27–30],而是研究导致本体冲突的本质原因;在探求冲突成因的研究中,不同于文献[15–17]中通过寻找不可满足概念的 MUPS 集合间接定位冲突,而是通过定义 R-MUPS 直接建立不可满足概念与冲突的联系.本文给出基于有序标签的本体调试的框架,其中以 $\mathcal{ALC}$ 语言为例,其他更复杂的描述逻辑本体可以在此基础上加以扩展,通过修改相应的 Tableau 规则予以实现.

本文第 1 节介绍 $\mathcal{ALC}$ 描述逻辑语言、本体中的基本概念以及标准的 Tableau 演算推演规则.第 2 节中形式化定义本体调试中的不一致(incoherency)定位问题,并给出概念覆盖、R-MUPS 等一系列概念和定理,论述本文定义的 R-MUPS 与 MUPS 和 MIPS 之间的关系.第 3 节对基于 R-MUPS 的 MIPS 求解算法进行了表述和实例分析,并进一步给出算法正确性证明及其复杂度和效率分析.第 4 节用自动生成的本体测试集和现实本体及其扩建本体对算法进行实验评估,并与经典的基于 Tableau 推理的调试方法进行了分析比较.最后给出结论和工作意义,并指出下一步工作.

1 $\mathcal{ALC}$ 及 Tableau 演算

描述逻辑(description logics,简称 DLs)是刻画本体的最常用形式化语言,它具有较强的表达能力,并能够提供可判定的推理服务.Baader&Nutt 2003[32]给出了关于描述逻辑的详细介绍.

用描述逻辑表示的本体知识库(knowledge base,简称 KB)是由 TBox 和 ABox 两部分组成:

- TBox 也叫术语集(terminology),是由 $C\sqsubseteq D$ 或 $C\equiv D$ 形式的公理构成,其中,$C\equiv D$ 等价于 $C\sqsubseteq D$ 和 $D\sqsubseteq C$,OWL 中的 $disjoint(C,D)$ 等价于 $C\sqsubseteq\neg D$ 和 $D\sqsubseteq\neg C$.如果公理左边是原子概念,称公理为该原子概念的一条定义公理;同时,该原子概念是命名符号,本体中没有定义公理的概念是基本符号.$\mathcal{ALC}$ 描述逻辑语言中的概念描述为

$$C,D\rightarrow\bot|\top|A|\neg C|C\sqcap D|C\sqcup D|\forall R.C|\exists R.C.$$

- 其中,$\bot$为空集,$\top$为全集,A 是原子概念,R 是抽象角色;
- ABox 是关于领域个体的断言集合,包括概念断言 $C(a)$和角色断言 $R(a,b)$两种:$C(a)$表示个体 a 满足概念 C 在 TBox 中的描述;$R(a,b)$表示个体 a 与 b 满足关系 R,或者说个体 b 是角色 R 中个体 a 的承载者.DL 的解释 $I=(\Delta^I,\bullet^I)$,Δ^I 表示解释的域(个体集);$\bullet^I$ 是映射函数,对原子概念 C 映射为一个集合 $C^I\subseteq\Delta^I$,对角色 R 映射为二元关系$\Delta^I\times\Delta^I$.解释 I 是公理 $C\sqsubseteq D$ 的模型,如果 I 满足 $C^I\subseteq D^I$;解释 I 是本体$\mathcal{O}$的模型,如果 I 满足$\mathcal{O}$中的所有公理.

概念 C 在本体$\mathcal{O}$中是不可满足的当且仅当$\mathcal{O}$的所有模型 I 都有 $C^I=\varnothing$.判定概念可满足性是本体推理中的基本问题,其他的推理任务都可以相应地转化为概念可满足判定问题,Tableau 演算可以判定 TBox 中概念的可满足性,$\mathcal{ALC}$ 语言的 Tableau 扩展规则如图 1 所示.

→⊓-规则
条件:$\mathcal{A}$ 包含$(C_1\sqcap C_2)(x)$,但并不同时包含 $C_1(x)$和 $C_2(x)$.
动作:$\mathcal{A}'=\mathcal{A}\cup\{C_1(x),C_2(x)\}$.
→⊔-规则
条件:$\mathcal{A}$ 包含$(C_1\sqcup C_2)(x)$,但既不包含 $C_1(x)$,也不包含 $C_2(x)$.
动作:$\mathcal{A}'=\mathcal{A}\cup\{C_1(x)\}$,$\mathcal{A}''=\mathcal{A}\cup\{C_2(x)\}$.
→∃-规则
条件:$\mathcal{A}$ 包含$(\exists R.C)(x)$,但不存在一个实例 z,使得 $C(z)$和 $R(x,z)$在 $\mathcal{A}$ 中.
动作:$\mathcal{A}'=\mathcal{A}\cup\{C(y),R(x,y)\}$,其中,$y$ 未在 $\mathcal{A}$ 中出现过的实例名
→∀-规则
条件:$\mathcal{A}$ 包含$(\forall R.C)(x)$和 $R(x,y)$,但并不包含 $C(y)$.
动作:$\mathcal{A}'=\mathcal{A}\cup\{C(y)\}$.

Fig.1 Tableau transformation rules of concept satisfiability for $\mathcal{ALC}$

图 1 $\mathcal{ALC}$ 语言判断概念可满足性的 Tableau 扩展规则

Tableau 演算根据不同的规则来扩展分解概念 C,其中每条扩展规则与描述逻辑中的构造算子相对应.其中,构造算子⊔会产生一个新的 ABox,其他的构造算子扩展当前的 ABox.初始时的 ABox $\mathcal{A}=\{C(a)\}$,当没有扩展规则可用或出现冲突时,Tableau 演算终止.此时,扩展得到一个 ABox 集 $\mathcal{S}=\{\mathcal{A}_1,\ldots,\mathcal{A}_n\}$,其中:如果存在一个 ABox $\mathcal{A}_i$ ($\mathcal{A}_i\in\mathcal{S}$)是不封闭的,则概念 C 是可满足的;反之,若所有的 $\mathcal{A}_i(\mathcal{A}_i\in\mathcal{S},1\leqslant i\leqslant n)$都是封闭的,则概念 C 是不可满足的. ABox $\mathcal{A}$是封闭的,如果$\mathcal{A}$满足$\{\bot(x)\}\subseteq\mathcal{A}$或者$\{A(x),\neg A(x)\}\subseteq\mathcal{A}$,其中:$x$ 是 Tableau 演算过程中产生的某个个体;A 是原子概念,将 $A(x),\neg A(x)$称为一个冲突对.

2 本体术语集调试中的 R-MUPS

2.1 不一致的定位问题

本体的逻辑冲突是本体工程中常见的问题,在诸如本体建模、本体融合、本体整合、本体迁移、本体进化等各种操作中极易产生.逻辑冲突一般表现为本体的逻辑不一致.文献[8]将本体不一致性解释为存在不可满足概念,定义如下.

定义 1(TBox 不一致(incoherency)[8]). TBox T 是不一致的当且仅当 T 中存在一个不可满足概念.

本体术语集不一致,即是本文所要研究解决的逻辑冲突.我们所要完成的本体调试工作,就是通过确定和修复逻辑冲突,使本体达到逻辑一致的过程.Schlobach[15]认为:基于描述逻辑系统的调试结果是一个存在逻辑冲突的公理集,去掉或修改该结果,可以使一个或者多个概念可满足,命题逻辑中的诊断是建立在极小冲突集[24]基础之上.如果通过本体调试能够确定出所有这样的极小冲突公理集,即达到定位本体不一致的效果.下边通过一个例子解释本体调试相关概念和过程,并简单说明概念不可满足原因和本体术语集不一致原因.

以下面的术语集 T_1 为例,试图构建一个包含 7 条公理的科研领域角色定义的知识术语集,其中,公理α_4定义了概念“大学老师”A_4 是必须指导学生(即表示为$\forall s.B$)的教师(即表示为 C).公理α_5 定义了概念“研究员”A_5 是至少指导一个不是学生的人,比如职工(即$\exists s.\neg B$).公理α_3 定义了身份既是大学老师又是研究员的,才是概念“科研人员”A_3.从逻辑认知的角度得知,如此定义的“科研人员”是不存在的,即概念 A_3 是不可满足的.原因在于根据定义 A_4 和 A_5 可知,“大学教师”和“研究员”是两个不相交的概念.即:一个实例要么所指导的人都是学生,要么所指导的人中至少存在一个不是学生,这两种情况不可能同时发生在同一个实例上.在人工智能的自动推理领域中,Tableau 演算可以有效模拟人类思维去判别概念逻辑冲突的这一过程.类似 A_3,可以得到 TBox T_1 中其他不可满足概念 A_1,A_6 和 A_7.但是经典的 Tableau 演算仅能够确定不可满足的概念,对概念为什么不可满足以及不可满足概念之间是否存在一些联系未加进一步推演.例如:概念 A_3 不可满足是公理集$\{\alpha_3,\alpha_4,\alpha_5\}$导致的,$A_3$ 是否可满足又决定概念 A_1 的可满足性,或者说本体术语集中一个不可满足概念制约其他某些概念的可满足性.此外,A_1 定义中的$\neg A$ 和 A_2 定义中的 A 构成一个冲突对,也导致概念 A_1 不可满足,说明促使概念不可满足的原因并不唯一.

例:TBox T_1:

α_1: $A_1 \sqsubseteq \neg A \sqcap A_2 \sqcap A_3$ α_2: $A_2 \sqsubseteq A \sqcap A_4$ α_3: $A_3 \sqsubseteq A_4 \sqcap A_5$

α_4: $A_4 \sqsubseteq \forall s.B \sqcap C$ α_5: $A_5 \sqsubseteq \exists s.\neg B$ α_6: $A_6 \sqsubseteq A_1 \sqcup \exists r.(A_3 \sqcap \neg C \sqcap A_4)$

α_7: $A_7 \sqsubseteq A_4 \sqcap \exists s.\neg B$

本体调试问题就是找出 TBox 不一致的所有原因,先给出概念不可满足原因的定义:

定义 2(MUPS[15]). 设概念 C 是 TBox T 中的不可满足概念,一个子集 $T' \subseteq T$ 称为 T 中 C 的极小不可满足保持子集(minimal unsatisfiability preserving sub-Tbox,简称 MUPS),如果 C 在 T' 下是不可满足的,则对于任意 $T'' \subset T'$,C 在 T'' 下都是可满足的.用 $mups(T,C)$表示 T 中关于 C 的所有 MUPS 的集合.

例子 TBox T_1 中,$mups(T_1,A_1)=\{\{\alpha_1,\alpha_2\},\{\alpha_1,\alpha_3,\alpha_4,\alpha_5\}\}$.由 MUPS 定义知,$\{\alpha_1,\alpha_2\}$和$\{\alpha_1,\alpha_3,\alpha_4,\alpha_5\}$任意子术语集都使得 A_1 是可满足的.类似地有 $mups(T_1,A_3)=\{\{\alpha_3,\alpha_4,\alpha_5\}\}$,$mups(T_1,A_6)=\{\{\alpha_1,\alpha_2,\alpha_4,\alpha_6\},\{\alpha_1,\alpha_3,\alpha_4,\alpha_5,\alpha_6\}\}$,$mups(T_1,A_7)=\{\{\alpha_4,\alpha_7\}\}$.MUPS 是概念不可满足原因的形式化定义,下面针对本体术语集给出其不一致原因 MIPS 的形式化定义:

定义 3(MIPS[15]). 设 TBox T 是不一致的,一个 TBox $T' \subseteq T$ 称为 T 的极小不一致保持子集(minimal incoherence-preserving sub-Tboxes,简称 MIPS),如果 T'是不一致的,对于任意 TBox $T'' \subset T'$,T''是一致的.用 $mips(T)$表示 T 中所有 MIPS 的集合.

本体调试的定义并不依赖某种特定的描述逻辑,本体术语集调试得到术语集的 MIPS,对应诊断中的极小冲突集.例子 TBox T_1 的 MIPS 为 $mips(T_1)=\{\{\alpha_1,\alpha_2\},\{\alpha_3,\alpha_4,\alpha_5\},\{\alpha_4,\alpha_7\}\}$,$A_1$ 在公理集$\{\alpha_3,\alpha_4,\alpha_5\}$下是可满足的,$A_3$ 却是不可满足的.TBox 不一致说明本体中至少存在一个不可满足概念,本体术语集的每个 MIPS 必定会使得本体中的某些概念不可满足,而且 MIPS 的任意一个子集都没有这样的性质.进一步地,Kalyanpur 在 MUPS 的基础上,将不可满足概念分为根和派生两类概念.

定义 4(根与派生不可满足概念[19]). TBox T 中,概念 C 是派生不可满足概念,如果 C 满足以下条件:存在 i,j 使得 $MUPS_i(D) \subseteq MUPS_j(C)$.这种情况下,称 C 是派生不可满足概念,简称派生概念.如果 C 不满足这个条件,则 C 是一个根不可满足概念,简称根概念.其中,D 是本体中另一个不可满足概念,$0 \leqslant i \leqslant |mups(T,D)|$,$0 \leqslant j \leqslant |mups(T,C)|$,即,任意的 $MUPS_i(D) \in mups(T,D)$,$MUPS_j(C) \in mups(T,C)$.

根概念指概念的可满足性不依赖本体中其他不可满足概念,这种不可满足性源于概念自身定义公理的矛盾,例如 T_1 中的 A_3.派生概念指概念不可满足直接或间接地依赖本体中另一个不可满足概念,例如 T_1 中的 A_1 依赖 A_3;同时,$\{\alpha_1,\alpha_2\}$也是 A_1 的不可满足错误.

2.2 相对极小不可满足保持子术语集(R-MUPS)

本体术语集由公理组成,每条公理是公理左端概念的一个形式化定义,每个命名概念都有相应的公理对其进行定义.给定本体术语集 T,称 T 中的概念 C 直接使用 D(D 是原子概念或者角色),如果 D 出现在 C 的定义公理的右端,称直接使用的传递闭包为使用.本体包含一个循环,如果存在一个使用概念自身的原子概念,也称本体是循环的.表达能力最强的描述逻辑语言 DL $\mathcal{SROIQ}$包含枚举型概念($\{a_1,...,a_n\}$,其中,a_i 是个体,$1 \leqslant i \leqslant n$),概念包含和等价公理,角色包含公理等[32].我们用 $\mathcal{U}(T,C)$为定义概念 C 所使用的符号集,其中包括概念、角色和个体符号.例子 TBox T_1 中,$\mathcal{U}(T_1,A_1)=\{A,A_2,A_3,A_4,A_5,B,C,s\}$.

定义 5(概念覆盖). 设 T 是一个不一致的本体术语集,C_1 和 C_2 是 T 中两个不可满足概念.称 C_2 覆盖 C_1,如果 $C_2 \in \mathcal{U}(T,C_1)$.用 $cover(T,C)$表示 T 中所有覆盖 C 的概念的集合.

概念覆盖能够反映不可满足概念之间的依赖关系,概念 C 依赖 $cover(T,C)$.根概念在一定程度上决定派生概念的可满足性,如果没有概念覆盖 C,那么 C 是根概念;否则,C 是派生概念.例子 T_1 中 $cover(T_1,A_1)=\{A_3\}$,$cover(T_1,A_3)=\{\}$,A_1 是 A_3 的派生概念.相反,根概念和派生概念不能完全体现概念覆盖.例如,TBox $T=\{\beta_1:B_1 \sqsubseteq \neg A \sqcap B_2, \beta_2:B_2 \sqsubseteq A \sqcap B_3, \beta_3:B_3 \sqsubseteq \neg A\}$,$mups(T,B_1)=\{\{\beta_1,\beta_2\}\}$,$mups(T,B_2)=\{\{\beta_2,\beta_3\}\}$,由 Kalyanpur 的根概念定义知,$B_1$ 和 B_2 都是根概念,因为β_1 中的$\neg A$ 与β_2 中 A 冲突,掩盖了β_2 中 A 与β_3 中$\neg A$ 的冲突,使得$\{\beta_1,\beta_2,\beta_3\}$不是 B_1 的 MUPS.根据概念覆盖有 $B_2 \in \mathcal{U}(T,B_1)$,即 B_1 覆盖 B_2.

定义 6(R-MUPS). 设概念 C 是 TBox T 中的不可满足概念,$\mathcal{M}=\cup_{C'\in cover(T,C)\cup\{C\}}mups(T,C')$.一个 TBox $T'\in\mathcal{M}$ 称为 C 的相对 MUPS(relative minimal unsatisfiability preserving sub-Tbox,简称 R-MUPS),如果不存在 $T''\in\mathcal{M}$ 使得 $T''\subset T'$.用 $r\text{-}mups(T,C)$表示 T 中关于 C 的所有 R-MUPS 的集合.

换句话说,R-MUPS 是合并概念及其覆盖概念的所有 MUPS 的极小值.例子 TBox T_1 中,$r\text{-}mups(T_1,A_1)=\{\{\alpha_1,\alpha_2\},\{\alpha_3,\alpha_4,\alpha_5\}\}$,$\{\alpha_3,\alpha_4,\alpha_5\}$对应的概念 A_3 被 A_1 覆盖.同样可以得到:

$$r\text{-}mups(T_1,A_3)=\{\{\alpha_3,\alpha_4,\alpha_5\}\},r\text{-}mups(T_1,A_6)=\{\{\alpha_1,\alpha_2\},\{\alpha_3,\alpha_4,\alpha_5\}\},r\text{-}mups(T_1,A_7)=\{\{\alpha_4,\alpha_7\}\}.$$

根据 R-MUPS 的定义,显然知,不可满足概念的 R-MUPS 具有如下性质,其中,T 是一个本体术语集,C 是 T 中的不可满足概念:

(1) 对任意 $C'\in cover(T,C)$,都有 $r\text{-}mups(T,C')\subseteq r\text{-}mups(T,C)$;

(2) 如果 T 中没有覆盖 C 的概念,那么 $r\text{-}mups(T,C)=mups(T,C)$;

(3) 对任意 C 都有 $r\text{-}mups(T,C)\neq\varnothing$.

引理 1. C 是本体术语集 T 中的不可满足概念,则对任意 $e\in mups(T,C)$都有 $Sig(e)\subseteq\mathcal{U}(T,C)$,其中,$Sig(e)$表示术语集 e 中使用的符号集(包括概念符号和角色符号).

定理 1. C 是本体术语集 T 中的不可满足概念,则 $r\text{-}mups(T,C)$中的每个元素都是 T 的 MIPS.

证明:根据 R-MUPS 的定义知 $r\text{-}mups(T,C)\subseteq\mathcal{M}$,其中,$\mathcal{M}=\cup_{C'\in cover(T,C)}mups(T,C)$.假设存在 $e_x\in r\text{-}mups(T,C)$不是 T 的 MIPS,即,存在 $e_y\subset e_x$ 是 T 的 MIPS.不妨设 e_x 和 e_y 分别为 C_x 和 C_y 的 MUPS,由于 $e_y\subset e_x$,根据引理 1 得 $C_y\in\mathcal{U}(T,C_x)$,并且由概念覆盖的定义有 $C_y\in cover(T,C_x)$.所以有 $C_y\in cover(T,C)$,进而 $e_y\in\mathcal{M}$,这与 $e_x\in r\text{-}mups(T,C)$矛盾.假设不成立,证毕. □

MIPS 表示的本体术语集是不一致的,并且它的任意真子集表示的术语集是一致的.也就是说,MIPS 使得某些概念不可满足,而这些概念基于 MIPS 的真子集所表示的术语集是可满足的.这样,MIPS 必然是术语集中的某些不可满足概念的 MUPS.同样地,由定理 1 可以知道:对于不一致本体的任意 MIPS 而言,必然存在某些不可满足概念的 R-MUPS 与之相等.所以,通过计算所有概念的 R-MUPS,可以得到本体术语集的所有 MIPS.但是由于不可满足概念的 R-MUPS 是建立在其覆盖概念的 MUPS 基础之上的,所以可以得到下边的定理:

定理 2. 本体术语集 T,任意 $r_t\in mips(T)$,T 中至少存在一个不可满足概念 C,使得 $r_t\in r\text{-}mups(T,C)$.

证明:$Cpt(T)$表示由 T 中所有不可满足概念构成的集合,令 $T_m=\cup_{C'\in Cpt(T)}mups(T,C')$.对 T 中的任意不可满足概念 C,令 $\mathcal{M}_C=\cup_{C'\in cover(T,C)\cup\{C\}}mups(T,C')$,那么 $T_m=\cup_{C'\in Cpt(T)}\mathcal{M}_{C'}$.用 $\min(S)$表示 S 在集合包含的偏序关系下的所有极小值构成的集合,根据 R-MUPS 定义,可知 $r\text{-}mups(T,C)=\min(\mathcal{M}_C)$,由 MIPS 的定义有 $mips(T)=\min(T_m)$.综合上边得到的等式,有 $mips(T)=\min(T_m)=\min(\cup_{C'\in Cpt(T)}\mathcal{M}_{C'})=\cup_{C'\in Cpt(T)}\min(\mathcal{M}_{C'})=\cup_{C'\in Cpt(T)}r\text{-}mups(T,C')$,证毕. □

定理 2 说明:求解本体术语集 T 的 MIPS 可以通过计算每个概念的 R-MUPS 可以得到;计算一个概念的 R-MUPS 等价于计算 T 的某个子术语集的 MIPS.因为概念之间的覆盖关系,这个子术语集包含覆盖的概念.也就是说,如果已经知道某个概念的 R-MUPS,那么不需要再对覆盖它的概念计算其 R-MUPS.结合二者,本文算法可以快速实现本体调试:计算本体术语集 T 的 $mips(T)$过程中,如果已经计算了不可满足概念 C 的 $r\text{-}mups(T,C)$,那么不用计算 $r\text{-}mups(T,C')$,其中,$C'\in cover(T,C)$.

3 基于 R-MUPS 的 MIPS 求解

Schlobach 方法计算 MIPS 是基于 MUPS,即:计算本体术语集中所有概念的 MUPS 并将其合并,再取其中的极小值得到术语集的 MIPS.换句话说,合并 TBox T 中所有概念的 MUPS 得到集合 $S_{mups}(T)$,$S_{mups}(T)$上的集合包含关系是一个部分序关系,记为 PS,则 $mips(T)$是由 $S_{mups}(T)$在关系 PS 下的所有极小元构成.例子 T_1 中,

$$S_{mups}(T_1)=\{\{\alpha_1,\alpha_2\},\{\alpha_1,\alpha_3,\alpha_4,\alpha_5\},\{\alpha_3,\alpha_4,\alpha_5\},\{\alpha_1,\alpha_2,\alpha_4,\alpha_6\},\{\alpha_1,\alpha_3,\alpha_4,\alpha_5,\alpha_6\},\{\alpha_4,\alpha_7\}\};$$

$$mips(T_1)=\{\{\alpha_1,\alpha_2\},\{\alpha_3,\alpha_4,\alpha_5\},\{\alpha_4,\alpha_7\}\}.$$

3.1 R-MUPS算法

为计算不可满足概念的 R-MUPS,在 Tableau 演算扩展规则(如图 1 所示)的基础上,定义标注有序标签的扩展规则(如图 2 所示),它是在 Tableau 演算的基础上为断言加上一个标签形如$(a{:}C)^{label}$,a 是个体,C 是概念,标签 *label* 是一个内部元素有序的公理集合.它以公理加入到标签中的先后为序,本文将带标签的断言简称为断言. R-MUPS 算法执行的终止条件是:没有规则可用或者存在某个分支没有冲突(概念可满足).R-MUPS 算法中,一个分支存在冲突,则该分支至少存在一个个体是⊥概念的实例或者同时是某个原子概念及其否定的实例.总是假定公式都是否定范式(negation normal form,简称 NNF,否定只出现在原子概念前边).

($\sqcap_l$-规则):如果$(a{:}C_1\sqcap C_2)^L\in B$,但不存在$\{(a{:}C_1)^L,(a{:}C_2)^L\}\subseteq B$
则 $B'{:=}B\cup\{(a{:}C_1)^L,(a{:}C_2)^L\}$.

($\sqcup_l$-规则):如果$(a{:}C_1\sqcup C_2)^L\in B$,但既不存在$(a{:}C_1)^L\in B$,也不存在$(a{:}C_2)^L\in B$
则 $B'{:=}B\cup\{(a{:}C_1)^L\}$,并且 $B''{:=}B\cup\{(a{:}C_2)^L\}$,其中,$L$ 是分支 B'和 B''的标签,
B 是 B' and B''的父结点分支.

(Ax_l-规则):如果$(a{:}A)^L\in B$ 并且$(A\sqsubseteq C)\in T$
则 $B'{:=}B\cup\{(a{:}C)^{L\bullet[A\sqsubseteq C]}\}$.

($\exists_l$-规则):如果$(a{:}\exists R_i.C)^L\in B$,再没有其他规则可应用于实例 a 的公式,
并且$\{(a{:}\forall R_i.C_1)L_1,\ldots,(a{:}\forall R_i.C_n)L_n\}\subseteq B$,
Then $B'=B\cup\{(b:C)^L,(b:C_1)^{L_1\infty L},\ldots,(b:C_n)^{L_n\infty L}\}$,其中,$b$ 是未出现过在 B 中的新实例.

Fig.2 Tableau transformation rules of R-MUPS for acyclic ALC (with labels)

图 2 非循环 $\mathcal{ALC}$ TBox 中概念的 R-MUPS 表扩展规则(带标签)

首先对标签及其运算进行说明如下,其中,L_1 和 L_2 是标签,β_i(其中,$1\leqslant i\leqslant m$)表示公理:

(1) 标签用$[\beta_1,\ldots,\beta_m]$表示,记法中,从左到右为公理加入到标签中的先后顺序,即标签的顺序;

(2) 用$\beta_i=NULL$ 表示公理无效,最终不会出现在 MUPS,MIPS 和 R-MUPS 中,用作标签运算中的分界标识.公理相等当且仅当公理的标记一样,并且均不为 *NULL*(公理的标记为β_i);

(3) $L_1\bullet L_2$,将 L_2 中元素(包括 *NULL*)逐个添加到 L_1 的末尾,返回 L_1;

(4) $L_1\infty L_2$,将 L_2 与 L_1 不同的部分合并到 L_1 中.合并方法:L_1 从头开始,在 L_1 和 L_2 第 1 次出现不等元素位置的前边插入无效公理,即 *NULL*(分界标识);然后,将 L_2 的剩余元素依次添加到 L_1 的末尾,返回 L_1.

使用图 2 给出的规则进行 R-MUPS 扩展时,对 B 运用$\sqcup_l$-rule 规则产生新分支,新分支则用产生分支断言的标签作为自己的标记,并且设置其父分支为当前分支.每个分支都有自己的标记(标签),记录从根分支到新分支扩展过程中所需的公理,标记中,末尾公理所描述的概念说明对其应用$\sqcup_l$-rule 规则产生标记所表示的分支.这样,n 次运用$\sqcup_l$-rule 规则之后产生新分支 B_n,产生分支路径为 B-B_1-…-B_{k-1}-B_k,称 B 为根分支,B_i 是 B_{i+1} 的父分支,B_{i+1} 是 B_i 的子分支,B_i 为 B_k 的祖先分支,B_k 为 B_i 的子孙分支,其中 $0\leqslant i\leqslant k-1$,$B_0=B$;对 B 运用$\sqcap_l$-rule 规则,满足条件则向 B 中加入新的断言,新断言继承标签;对 B 运用 Ax_l-rule 规则,向断言的标签末尾添加新公理;对 B 运用$\exists_l$-rule 规则,新增断言是由相应的存在($(a{:}\exists R_i.C)^L$)和全称限定断言($(a{:}\forall R_i.C_j)L_j$,$1\leqslant j\leqslant n$)共同产生,执行 $L\infty L_j$,在 L_j 与 L 共同公理部分之后加入分界标识 *NULL*,表示标签中 *NULL* 之后的公理受到限定断言的约束,并将存在限定断言中约束全称限定断言的公理主次添加至其末尾.

R-Mups-ComputeFunction 计算本体术语集 T 中概念 C 的所有 R-MUPS 和覆盖 C 的所有概念,返回值为$\langle R\text{-}mups,Cover\rangle$,其中,$R\text{-}mups=r\text{-}mups(T,C)$,$cover(T,C)\subseteq Cover$(如果 C 可满足,$cover(T,C)=\varnothing$).初始时,分支列表 $BranchList=\{B\}$只有一个分支 B,其中,$B=\{(a{:}C)^{\varnothing}\}$,$B$ 的标记为空,B 的父亲分支为其自身.从 *BranchList* 中获取分支,对分支使用图 2 中规则进行扩展,其中,运用$\sqcup_l$-rule 规则向 *BranchList* 头部插入新元素(新分支),使得 $BranchList=\{B',B'',\ldots\}$,实现深度优先遍历分支.没有规则可用时,当前分支扩展终止,得到:

$$BranchList=\{B',B_1,\ldots,B_n\}.$$

如果当前分支 B'存在冲突,首先调用 $conflictCompute(B',T)$计算 C 在 B'的 R-MUPS,并分别处理返回值

$\langle Unsat,ownCon,fatherCon\rangle$的各个部分:将 *Unsat* 表示的 B'中不可满足概念加入到 *Cover*,B'的自身冲突 *ownCon* 加入到 *R-mups*,当前分支和 *fatherCon* 压入栈顶;然后调用函数 *CombineBranch*(*Stack*,*T*),合并已经处理过的分支和当前分支,将合并后分支的自身冲突加入到 *R-mups*;最后,当所有分支都处理完毕,将栈中的剩余分支回溯至根分支,并获取根分支的冲突.如果分支 B'没有冲突,说明 C 是可满足的,那么 $R\text{-}mups=\varnothing,Cover=\{A|(a:A)^z\in B'\}$,即,*Cover* 保存 B'中各断言所对应的原子概念,这些概念是可满足的,然后函数直接返回,见算法 1.

算法 1. 计算所有 R-MUPS 算法 *R-Mups-ComputeFunction*(*C*,*T*).

输入:概念 *C*,非循环 $\mathcal{ALC}$ 本体术语集 *T*;

输出:公理集合簇(*r-mups*(*T*,*C*))*R-mups*,概念的集合 *Cover*.

Step 1. 初始化变量 *R-mups*,*Cover*,*Stack* 为空,$B=\{(a:C)^{\varnothing}\}$,*BranchList*={*B*};

Step 2. 遍历 *BranchList* 中的分支 *Br*,做如下操作:

  Step 2.1. 使用 Fig.1.1 中的规则对 *Br* 进行扩展,直到没有规则可以使用时终止;

  Step 2.2. 如果 *Br* 没有冲突,清空 *R-mups* 和 *Cover*,遍历 *Br* 的断言$(a:A)^z$,将 *A* 加入到 *Cover* 中,返回(*R-mups*,*Cover*);

  Step 2.3. 如果 *Br* 存在冲突,做如下操作:

    Step 2.3.1. 调用 *conflictCompute* 函数计算 *Br* 中的冲突,返回值为(*Unsat*,*ownCon*,*fatherCon*);

    Step 2.3.2. 将$\langle Br,fatherCon\rangle$入栈 *Stack*;

    Step 2.3.3. 调用 *CombineBranch* 函数计算分支合并后的所有冲突 *backCon*;

    Step 2.3.4. 取 $R\text{-}mups\cup ownCon\cup backCon$ 中的极小值保存到 *R-mups* 中;

    Step 2.3.5. 合并 *Unsat* 到 *Cover* 中;

  Step 2.4. 从 *BranchList* 中删除 *Br*;

Step 3. 将$\langle null,null\rangle$入栈 *Stack*,作为结束标识符;

Step 4. 调用 *CombineBranch* 函数计算最后分支合并后的所有冲突 *backCon*;

Step 5. 弹出 Stack 中的栈顶元素$\langle rootBr,rootCon\rangle$;

Step 6. 取 $R\text{-}mups\cup rootCon$ 中的极小值保存到 *R-mups* 中;

Step 7. 返回(*R-mups*,*Cover*);

冲突计算函数 *conflictCompute*(*C*,*B*,*T*),实现计算概念 *C* 在分支 *B* 下的 R-MUPS 和覆盖 *C* 的所有概念.变量 *father* 记录当前分支到根分支的所有标记中的元素,*Unsat* 记录 *B* 中的不可满足概念,*ownCon* 记录分支自身的冲突,*fatherCon* 记录除自身冲突之外的冲突,*ownCon* 和 *fatherCon* 共同保存概念 *C* 在分支 *B* 下 R-MUPS.关于冲突对$(a:A)^x$和$(a:\neg A)^y$,将标签 *x* 和 *y* 从头到尾第 1 次出现对应位置元素不相等的前一个公理记作α_B,那么合并 *x* 和 *y* 中从α_B到末尾间的公理集(由变量 *con* 表示),使得α_B所描述的概念是不可满足的;进一步地,*x* 或 *y* 中从头到α_B之间的公理所表示的概念都是不可满足的(保存在变量 *Unsat* 中).如果α_B不在 *father* 中,表示 *con* 可能是 *B* 上覆盖 *C* 的某个概念的 MUPS,称为当前分支的自身冲突,保存到 *ownCon* 中;否则,保存到 *fatherCon* 中,称为当前分支的祖先冲突,见算法 2.

算法 2. 计算分支冲突算法 *conflictCompute*(*C*,*B*,*T*).

输入:概念 *C*,分支 *B*,非循环 $\mathcal{ALC}$ 本体术语集 *T*;

输出:概念的集合 *Unsat*,公理集合簇 *ownCon*,公理集合簇 *fatherCon*.

Step 1. 初始化变量 *Unsat*,*ownCon*,*fatherCon*,*father* 为空,*Btmp* 为 *B*;

Step 2. 将分支 *Btmp* 的标记加入到 *father* 中;

Step 3. 做如下操作,直到 *Btmp* 的父亲分支不是其自身:

  Step 3.1. 修改 *Btmp* 为其父亲分支;

  Step 3.2. 将 *Btmp* 的标记加入到 *father* 中;

Step 4. 遍历 *B* 中冲突对$(a:A)^x$和$(a:\neg A)^y$,做如下操作:

Step 4.1. 将 *x* 和 *y* 中从开始,到第一次出现元素不相等位置之间的公理所描述的概念添加到 *Unsat* 中;

Step 4.2. 合并 *x* 和 *y* 中从第一次出现不等元素的前一位置的元素,到各自末尾之间的公理到 *con* 中,删除 *NULL*;

Step 4.3. 如果 *con* 和 *father* 不相交,取 $ownCon\cup\{con\}$中的极小值保存到 *ownCon* 中;

否则,取 $fatherCon\cup\{con\}$中的极小值保存到 *fatherCon* 中;

Step 5. 返回(*Unsat*,*ownCon*,*fatherCon*);

分支合并函数 *CombineBranch*(*Stack*,*T*),*R-Mups-ComputeFunction*(*C*,*T*)函数采用深度优先顺序处理各包含冲突的分支是分支合并的前提.如果栈顶元素$\langle curBr,curCon\rangle=\langle null,null\rangle$,表示 *C* 没有分支可扩展,则合并栈 *Stack* 中未处理的分支并回溯至根分支,*Stack* 为空时终止.如果栈顶元素$\langle curBr,curCon\rangle\neq\langle null,null\rangle$时,需要对当前分支和栈 *Stack* 中的分支进行祖孙关系判断,并作出相应的操作:如果栈 *Stack* 为空,表示当前分支是第 1 个分支,分支合并函数第 1 次被调用,将$\langle curBr,curCon\rangle$压栈并返回;如果栈 *Stack* 不为空,进行分支关系的判断,其中,*CommonBranch* 为 *preBr* 与 *curBr* 到根分支的分支序列中第 1 个相同的分支:(1) 如果 *preBr* 与 *curBr* 相等,判断是否是同一分支,是则合并两个分支(将 *preBr* 并入到 *curBr* 中),再将$\langle curBr,curCon\rangle$压栈返回;(2) 如果 *preBr* 是 *curBr* 的祖先分支,不能合并分支,则依次将$\langle preBr,preCon\rangle$和$\langle curBr,curCon\rangle$压栈返回,等待当前分支处理完毕;(3) 如果 *preBr* 与 *curBr* 不是同一分支并且 *preBr* 不是 *curBr* 的祖先分支,则将 *preBr* 回溯至 *preBr* 与 *curBr* 第 1 个共同分支 *CommonBranch* 停止:首先,*preBr* 回溯为其父分支,并分离新 *preBr* 的自身冲突(保存到 *backCon* 中)和祖先冲突(参与合并分支时的冲突计算),然后将 *CombineBr* 并入到 *preBr* 中(合并后分支的冲突是由参与合并分支的祖先冲突计算得到),称这样的过程为一次回溯.*preBr* 与 *CommonBranch* 同一分支时回溯终止,再判断 *preBr* 与 *curBr* 是否进行同一分支的合并操作,见算法 3.

算法 3. 分支合并算法 *CombineBranch*(*Stack*,*B*,*T*).

输入:栈 *Stack*,分支 *B*,非循环 $\mathcal{ALC}$ 本体术语集 *T*;

输出:公理集合簇 *backCon*.

Step 1. 初始化变量 *backCon* 为空;

Step 2. 弹出 *Stack* 的栈顶元素$\langle curBr,curCon\rangle$;

Step 3. 如果 Stack 非空,做如下操作:

Step 3.1. 弹出 *Stack* 的栈顶元素$\langle preBr,preCon\rangle$;

Step 3.2. 如果 *curBr* 为 *null*,则变量 *CommonFather* 为 *null*;

否则,*CommonBranch* 为 *curBr* 和 *preBr* 到根分枝序列中第一个共同的祖先分支;

Step 3.3. 如果 *preBr* 与 *curBr* 表示的是不同分支,且 $preBr\neq CommonBranch$,做如下操作,直到 *Stack* 为空:

Step 3.3.1. 将 *preCon* 中与 *preBr* 的所有祖先分支都无交集的元素加入到 *backCon* 中,保留 *backCon* 中的极小值;

Step 3.3.2. 弹出 *Stack* 的栈顶元素$\langle CombineBr,CombineCon\rangle$;

Step 3.3.3. 令 $preBr=CombineBr$;

Step 3.3.4. 取 *CombineCon* 和 *preCon* 笛卡尔乘积的极小值保存到 *preCon* 中;

Step 3.3.5. 如果 *CommonBranch* 和 *preBr* 表示同一分支,则跳出循环,即,跳至 Step 3.4;

Step 3.4. 如果 *curBr* 不为 *null*,并且 $curBr=preBr$,则取 *curCon* 和 *preCon* 笛卡尔乘积的极小值保存到 *curCon* 中;

否则,将$\langle preBr,preCon\rangle$入栈,即,加入到 Stack 中;

Step 4. 如果 *curBr* 不为 *null*,将$\langle curBr,curCon\rangle$入栈,即,加入到 Stack 中;

Step 5. 返回 *backCon*;

3.2 R-MUPS算法举例

以前面的 TBox T_1 为例,计算 *R-Mups-ComputeFunction*(A_6,T_1),初始 $BranchList=\{B\}$,$B=\{(a{:}A_6)^{\varnothing}\}$.

Step 1: 对 B 中$(a{:}A_6)^{\varnothing}$使用 Ax$_l$-rule 规则,得:$B'=\{(a{:}A_6)^{\varnothing},(a{:}A_1\sqcup\exists r.(A_3\sqcap\neg C\sqcap A_4))^{[\alpha_6]}\}$,$BranchList=\{B'\}$;

Step 2: 对 B 应用$\sqcup_l$-rule 规则:$BranchList=\{B',B''\}$,$B'=\{(a{:}A_6)^{\varnothing},(a{:}A_1\sqcup\exists r.(A_3\sqcap\neg C\sqcap A_4))^{[\alpha_6]},(a{:}A_1)^{[\alpha_6]}\}$(标记为$[\alpha_6]$,父分支为 B'),$B''=\{(a{:}A_6)^{\varnothing},(a{:}A_1\sqcup\exists r.(A_3\sqcap\neg C\sqcap A_4))^{[\alpha_6]},(a{:}\exists r.(A_3\sqcap\neg C\sqcap A_4))^{[\alpha_6]}\}$(标记为$[\alpha_6]$,父分支为 B');

Step 3: 对 B'使用 Ax$_l$-rule,$\sqcap_l$-rule 和$\exists_l$-rule 规则逐步扩展,没有规则可用时终止,图 3 给出 B'的扩展过程和结果;

Step 4: 调用 $conflictCompute(A_6,B',T_1)$计算 A_6 在 B'下的 R-MUPS,再将 B'和返回值 *fatherCon* 压栈.每个冲突对的计算结果如下,其中,变量 $father=\{\alpha_6\}$合并从 B'到根分支序列上各个分支的标记,*common* 是冲突对标签中共同部分的公理所描述的概念,*con* 是冲突对计算得到的冲突,*father* 和 *con* 与函数 *conflictCompute* 中的同名变量对应:

(1) $(a:A)^{[\alpha_6,\alpha_1,\alpha_2]}$ 和 $(a:\neg A)^{[\alpha_6,\alpha_1]}$:$con=\{\alpha_1,\alpha_2\}$,$common=\{A_6,A_1\}$;

(2) $(b:B)^{[\alpha_6,\alpha_1,NULL,\alpha_2,\alpha_4,\alpha_3,\alpha_5]}$ 和 $(b:\neg B)^{[\alpha_6,\alpha_1,\alpha_3,\alpha_5]}$:$con=\{\alpha_1,\alpha_2,\alpha_3,\alpha_4,\alpha_5\}$,$common=\{A_6,A_1\}$;

(3) $(b:B)^{[\alpha_6,\alpha_1,\alpha_3,NULL,\alpha_4,\alpha_5]}$ 和 $(b:\neg B)^{[\alpha_6,\alpha_1,\alpha_3,\alpha_5]}$:$con=\{\alpha_3,\alpha_4,\alpha_5\}$,$common=\{A_6,A_1,A_3\}$;

最后返回值:$Unsat=\{A_6,A_1,A_3\}$,$ownCon=\{\{\alpha_1,\alpha_2\},\{\alpha_3,\alpha_4,\alpha_5\}\}$,$fatherCon=\varnothing$;

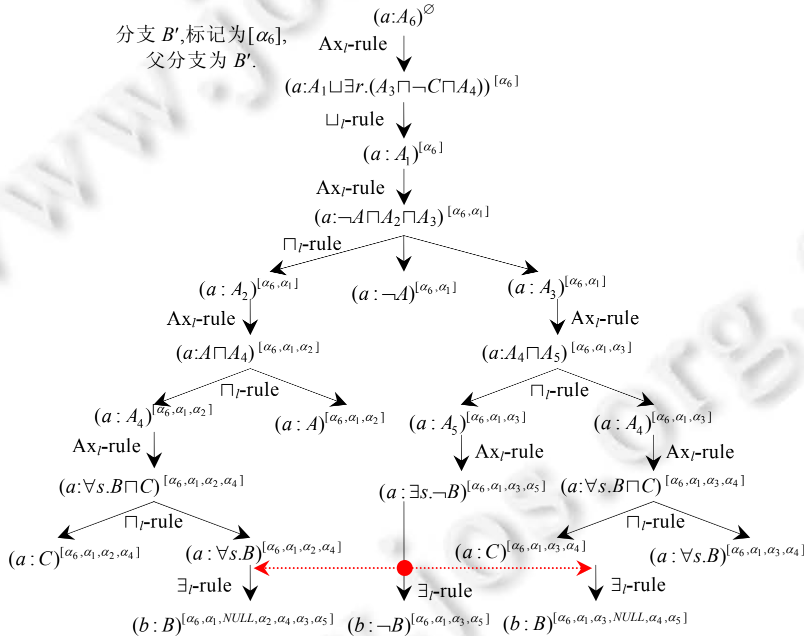

Fig.3 Branch B' which is R-MUPS transformating about concept A_6 for TBox T_1

图 3 *TBox* T_1 中 A_6 进行 R-MUPS 扩展中的分支 B'(标记为$[\alpha_6]$,父分支为 B')

第 5 步: 调用 $CombineBranch(Stack,T)$函数进行分支合并.*Stack* 中只有一个元素$\langle B',\varnothing\rangle$,直接返回,合并分支得到的 R-MUPS 为 $backCon=\varnothing$;

第 6 步: B'处理完毕,计算 *R-mups* 和 *Cover* 值,*R-mups* 是由 *R-mups*,*ownCon* 和 *backCon* 合并之后的极小值构成,*Cover* 是合并 *Cover* 和 *Unsat*,$R\text{-}mups=\{\{\alpha_1,\alpha_2\},\{\alpha_3,\alpha_4,\alpha_5\}\}$,$Cover=\{A_6,A_1,A_3\}$;

第 7 步: 对 B''应用 R-MUPS 扩展规则逐步扩展结果如图 4 所示.

第 8 步: 调用 $conflictCompute(A_6,B'',T_1)$计算 A_6 在 B''下的 R-MUPS,然后将 B''和返回值 *fatherCon* 压栈:

(1) $(b:C)^{[\alpha_6,\alpha_3,\alpha_4]}$ 和 $(b:\neg C)^{[\alpha_6]}$:$con=\{\alpha_6,\alpha_3,\alpha_4\}$,$common=\{A_6\}$;

(2) $(b:C)^{[\alpha_6,\alpha_4]}$ 和 $(b:\neg C)^{[\alpha_6]}$:$con=\{\alpha_6,\alpha_4\}$,$common=\{A_6\}$;

(3) $(c:B)^{[\alpha_6,\alpha_3,NULL,\alpha_4,\alpha_5]}$ 和 $(c:\neg B)^{[\alpha_6,\alpha_3,\alpha_5]}$:$con=\{\alpha_3,\alpha_4,\alpha_5\}$,$common=\{A_6,A_3\}$;

(4) $(c:B)^{[\alpha_6,NULL,\alpha_4,\alpha_3,\alpha_5]}$ 和 $(c:\neg B)^{[\alpha_6,\alpha_3,\alpha_5]}$:$common=\{A_6\}$,$con=\{\alpha_6,\alpha_3,\alpha_4,\alpha_5\}$;

最后返回值:$Unsat=\{A_6,A_3\}$,$ownCon=\{\{\alpha_3,\alpha_4,\alpha_5\}\}$,$fatherCon=\{\{\alpha_4,\alpha_6\}\}$;

第 9 步: 调用 *CombineBranch*(*Stack*,*T*)函数进行分支合并.*Stack* 中有两个元素:栈顶元素$\langle curBr,curCon\rangle=\langle B'',\{\{\alpha_4,\alpha_6\}\}\rangle$,另一个元素$\langle preBr,preCon\rangle=\langle B',\varnothing\rangle$.*curBr* 与 *preBr* 的标记相同,表示同一个分支,进行合并操作(*preBr* 并入到 *curBr* 中)得 $curCon=\varnothing$(其中元素为 *curCon* 与 *preCon* 作笛卡尔乘积的极小值).然后返回,返回值 $backCon=\varnothing$;

第 10 步: B''处理完毕,计算 *R-mups* 和 *Cover* 值,$R\text{-}mups=\{\{\alpha_1,\alpha_2\}\{\alpha_3,\alpha_4,\alpha_5\}\}$,$Cover=\{A_6,A_1,A_3\}$;

第 11 步: 所有分支处理完毕,向 *Stack* 中压入$\langle null,null\rangle$,然后调用 *CombineBranch*(*Stack*,*T*)函数做最后的分支合并处理.*Stack* 中只有一个元素$\langle preBr,preCon\rangle=\langle B',\varnothing\rangle$,已经是根分支,返回值 $backCon=\varnothing$.

最后计算 *R-mups* 为合并 *R-mups* 和 *backCon* 得到的极小值:$R\text{-}mups=\{\{\alpha_1,\alpha_2\}\{\alpha_3,\alpha_4,\alpha_5\}\}$,*Cover* 保持不变,即,$Cover=\{A_6,A_1,A_3\}$.函数返回值为$(\{\{\alpha_1,\alpha_2\},\{\alpha_3,\alpha_4,\alpha_5\}\},\{A_6,A_1,A_3\})$.这个例子简单验证了算法的正确性.

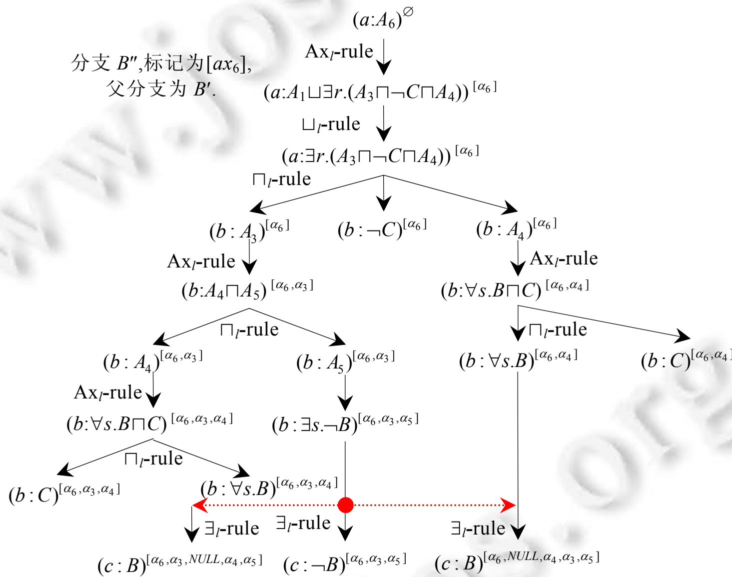

Fig.4 Branch B'' which is R-MUPS transformating about concept A_6 for TBox T_1

图 4 *TBox* T_1 中 A_6 进行 R-MUPS 扩展中的分支 B''(标记为$[\alpha_6]$,父分支为 B')

3.3 算法分析

本节给出概念扩展树的构建规则与概念 R-MUPS 扩展过程等价定理,再用扩展树证明关于本体术语集 *T* 及其概念 *C* 的 *R-Mups-ComputeFunction*(*C*,*T*)确实是 R-MUPS 问题的极小化函数.扩展树 *E-T* 中的结点记为(*a*,*C*,*label*),其中,*a* 为个体,*C* 为概念,*label* 为扩展 *C* 的有序公理集.如果 *C*=*NULL*,称结点(*a*,*NULL*,*label*)为空结点(*NULL-node*)或者关于个体 *a* 的空结点.

定义 7. 概念扩展树是由唯一的起始结点根结点出发的结点集合,其中:任何非根结点有且仅有一个前驱结点,称之为该结点的父结点;任何结点都可能有多个后继结点,称为该结点的子结点;若某结点没有后继结点,称之为叶子结点.若存在路径 n_0-…-n_m,其中,n_{i+1} 是 n_i 的后继结点,则称 n_m 是 n_0 的子孙结点,n_0 是 n_m 的祖先结点.一个结点到它的某个子孙结点有且仅有一条路径.

定义 8. *ETree* 为概念的一个扩展树,*node* 为 *ETree* 的一个结点,除去从根结点 *root* 到 *node* 路径中与 *node*

www.jos.org.cn

对应个体不相等的空结点,得到结点序列 $root=(a_0,C_0,l_0)$-$node_1=(a_1,C_1,l_1)$-…-$node=(a_n,C_n,l_n)$,$node$ 到序列中任一结点 $node_i(0\leqslant i\leqslant n-1)$的路径值为 $l_i\cdot l_{i+1}\cdot\ldots\cdot l_n$,其中,$node$ 到 $root$ 的路径为 $l_0\cdot l_1\cdot\ldots\cdot l_n$.用 $path(n_1,n_2,ETree)$表示树 $ETree$ 中结点 n_1 到结点 n_2 的路径值,$path(n,ETree)$表示树 $ETree$ 中结点 n 到根结点的路径值,路径值是一个有序公理集.

定义 9(LCF). n_1 和 n_2 是扩展树 $ETree$ 中两个结点,结点 N_L 是由根结点到 n_1 和 n_2 的路径上最后一个相同的非空结点(N_L 到根结点之间对应的结点都相同),并且 N_L 对应的概念是原子的,称 N_L 为 n_1 和 n_2 的 LCF (*LastCommonFather*).用 $lcf(n_1,n_2,ETree)$表示树 $ETree$ 中结点 n_1 和 n_2 的 LCF.

定义 10. 概念 R-MUPS 分支 B 中的断言$(a{:}C)^x$与概念扩展树中 ETr 中非空结点 $N_c=(a',C',y)$一一对应,如果 $a'=b',C=C',x=path(N_i,ETr)$(标签与路径值中对应元素相等,其中,$x$ 中的 $NULL$ 值与 N_i 路径结点序列中个体为 b' 的 $NULL$ 结点对应).如果对任意的$(a{:}C)^x\in B$ 都存在$(a',C',y)\in ETr$ 与之一一对应,称 B 与 ETr 一一对应,其中,C 为原子概念.

定义 11(标识结点). 概念的 R-MUPS 分支 B 与概念扩展树 ETr 一一对应,称结点 $curNode=(a_{cur},C_{cur},l_{cur})$为 ETr 的标识结点,如果 $path(curNode,ETr)$等于 B 的标记,并且 $curNode$ 的路径上不存在其他结点的路径值等于 B 的标记,称与 B 的父分支标记对应的标识结点为 ETr 的父标识结点.

概念 R-MUPS 扩展规则与图 4 中相应的扩展树生成规则一一对应,对概念 C 进行 R-MUPS 扩展;相应地,对 C 执行图 4 中相同的规则构建扩展树.初始时,R-MUPS 扩展 $B=\{(a{:}C)^{\varnothing}\}$,对应扩展树 $ETr=\{(a,C,\varnothing)\}$. R-MUPS 扩展终止时的每个分支都对应一棵扩展树,如果分支存在冲突,相应的扩展树也是冲突的,其中,与分支冲突对应的结点称为冲突结点.

(**⊓-ET**):如果结点 $N=(a,C_1\sqcap C_2,L)\in ETr$,但并不存在$\{N_1=(a,C_1,L),N_2=(a,C_2,L)\}\subseteq ETr$,其中,$N_F$ 是 N 的父结点
则 $ETr'=ETr\cup\{N_1,N_2\}-\{N\}$,其中,$N_F$ 是 N_1 和 N_2 的父结点.

(**⊔-ET**):如果结点 $N=(a,C_1\sqcup C_2,L)\in ETr$,但既不存在 $N_1=(a,C_1,L)\in Etr$,也不存在 $N_2=(a,C_2,L)\in ETr$,其中,N_F 是 N 的父结点
则 $ETr'=ETr\cup\{N_1\}-\{N\}$,$ETr''=ETr\cup\{N_2\}-\{N\}$,其中,$N_F$ 是 N_1 and N_2 的父结点,也是 ETr'和 ETr''的标记结点

(**Ax-ET**):如果结点 $N=(a,A,L)\in ETr$,并且 $A\sqsubseteq C\in T$,其中,N_F 是 N 的父结点
则 $ETr'=ETr\cup\{N',N_s\}-\{N\}$,其中,$N'=(a,A,L\bullet[A\sqsubseteq C])$,$N_s=(a,C,\varnothing)$,$N_F$ 是 N'的父结点,N'是 N_s 的父结点.

(**∃-ET**):如果结点 $N=(a,\exists R_i.C,L)\in ETr$,再没有其他可用于关于 a 的结点,并且$\{N_1=(a,\forall R_i.C_1,L_1),\ldots,N_n=(a,\forall R_i.C_n,L_n)\}\subseteq ETr$,
其中,N_F 是 N 的父结点,NF_i 是 N_i 的父结点$(1\leqslant i\leqslant n)$
则 $ETr'=ETr\cup\{N',N_1',\ldots,N_n',NULL_1,\ldots,NULL_n\}-\{N,N_1,\ldots,N_n\}$,
其中,$N'=(b,C,L)$, $N_i'=(b,C_i,L_i\bullet path(N',lcf(N_i',N',ETr),ETr)$,$NULL_i=(b,NULL,[NULL])$,
N_F 是 N'的父结点,NF_i 是 N_i 的父结点,$NULL_i$ 是 $lcf(N_i',N',ETr)$在 N_i' 路径上的子结点,
前提是如果它不包含 b 上的 $NULL$-$node$,其中,b 是一个新生成的实例.

Fig.5 Tableau transformation rules of concept transformation tree for acyclic ALC (with labels)

图 5 非循环 $\mathcal{ALC}$ TBox 概念扩展树的表扩展规则(带标签)

引理 2. R-MUPS 扩展终止时得到的每个分支都存在一棵扩展树与其对应,且 R-MUPS 扩展分支中对应断言的标签等于相应扩展树中非空结点的路径值.

定理 3. 关于本体术语集 T 及其不可满足概念 C,极小化函数 *R-Mups-ComputeFunction*(C,T)得到 *r-mups*(T,C)和 *cover*(T,C).

证明:*R-Mups-ComputeFunction*(C,T)的返回值为(*R-mups*,*Cover*),不可满足概念 C 的 R-MUPS 扩展得到所有分支 $BranchList=\{B_1,\ldots,B_n\}$,相应的森林 $ETList=\{ETr_1,\ldots,ETr_n\}$(森林是树的有限集合),其中,从 B_1 到 B_n 是采用深度优先的处理顺序.每个分支 $B_i(1\leqslant i\leqslant n)$对应一棵扩展树 ETr_i,分支 B_i 中的每个冲突对$(c_i{:}A_i)l_i$和$(c_i{:}\neg A_i)k_i$,分别对应扩展树 ETr_i 的冲突结点 $N_1'=(c_i,A_i,l_i')$ 和 $N_2'=(c_i,\neg A_i,k_i)$.

(1) 当 $n=1$ 时,C 在 R-MUPS 扩展终止时只有一个分支 B_1:结点 $Cfather=lcf(N_1',N_2')$,以 $Cfather$ 为根的子树是对其对应概念进行 R-MUPS 扩展得到的,$Cfather$ 子树中的冲突结点也是其祖先结点的冲突结点.$Cfather$ 对应概念不可满足的公理集 $con=path(N_1',Cfather,ETr)\cup path(N_2',Cfather,ETr)-\{NULL\}$(合并路径值并除去其中的 $NULL$ 值),其祖先结点 Anc 对应的不可满足公理集为 $con\cup path(Cfather,Anc,ETr)-\{NULL\}$.$Cfather$ 对应的概念

被其祖先结点的原子概念覆盖,有 $Cover=cover(T,C)=\{Cpt|Cpt$ 是公理β所描述的概念,其中,$\beta\in path(Cfather, ETr)\}$.用 $ownCon$ 保存这样的 con,则有 $R\text{-}mups=r\text{-}mups(T,C)=\vee_{con\in ownCon}(\wedge_{ax\in con}ax)$.

(2) 当 $n=2$ 时,C 在 R-MUPS 扩展终止时有两个分支 B_1 和 B_2:已知 ETr_1 和 ETr_2 的分支结点为 $Bnode=(a_b,C_b,l_b)$,$(UC_1,OC_1,FC_1)=conflictCompute(C,B_1,T)$,$(UC_2,OC_2,FC_2)=conflictCompute(C,B_2,T)$,$Cover=UC_1\cup UC_2$,$R\text{-}mups=\{con|con$ 是 $OC_1\cup OC_2\cup\{FC_1$ 与 FC_2 笛卡尔乘积$\}$的极小值$\}$,只要证明 $R\text{-}mups\subseteq mips(T)$成立.假设 $R\text{-}mups$ 中存在 $con_f\notin mips(T)$,即:存在 $con_m\in r\text{-}mups(T,C)$满足 $con_m\subset con_f$.由情形(1)得知,$OC_1\subseteq mips(T)$,$OC_2\subseteq mips(T)$,所以 $con_m\notin OC_1\cup OC_2$,$con_f\notin OC_1\cup OC_2$.设 C_m 是 con_m 对应的不可满足概念,C_m 对应 ETr_1 或 ETr_2 中根结点到 $Bnode$ 间某个结点 $N_m=(a_m,C_m,l_m)$,以 N_m 为根结点的子树对应 C_m 的 R-MUPS 扩展(同样有两个分支),即,$con_m=con_1\cup con_2$,其中,con_j 是 $ETr_j(j=1,2)$中得到 con_m 的冲突.设β_b 是描述 C_b 的公理,分两种情况讨论如下:

〈1〉 $\beta_b\notin con_m$:$\beta_b\notin con_1$,$\beta_b\notin con_2$,也就是说,使得 C_m 不可满足的原因 con_m 与分支无关,即 $con_1=con_2$.如果 $con_1\in FC_1$,必然有 $con_2\in FC_2$,那么 $con_1\cup con_2\in R\text{-}mups$,与 $con_f\in R\text{-}mups$ 矛盾.如果 $con_1\notin FC_1$,则存在 $con_{s1}\in FC_1$ 且 $con_{s1}\subset con_1$,$con_{s2}\in FC_2$ 且 $con_{s1}=con_{s2}$,那么 $con_{s1}\cup con_{s2}\in R\text{-}mups$ 且 $con_{s1}\cup con_{s2}\subset con_m$,与 $con_m\in r\text{-}mups(T,C)$矛盾.假设不成立.

〈2〉 $\beta_b\in con_m$:当$\beta_b\in con_1$,$\beta_b\notin con_2$ 时,有 $con_2\in FC_1$,那么 $con_2\cup con_2\in R\text{-}mups$ 且 $con_2\cup con_2\subset con_1\cup con_2$,与 $con_m\in r\text{-}mups(T,C)$矛盾.当 $ax_b\in con_1$,$ax_b\in con_2$ 时,如果 $con_j\in FC_j$,有 $con_1\cup con_2\in R\text{-}mups$,与 $con_f\in R\text{-}mups$ 矛盾;如果 $con_j\notin FC_j$,存在 $con_{sj}\in FC_j$,则 $con_{s1}\cup con_{s2}\subset con_1\cup con_2$,与 $con_m\in r\text{-}mups(T,C)$矛盾;如果 $con_1\in FC_1$,$con_2\notin FC_2$,则存在 $con_{s2}\in FC_2$ 且 $con_{s2}\subset con_2$,那么 $con_1\cup con_{s2}\subseteq con_1\cup con_2$,与 $con_m\in r\text{-}mups(T,C)$矛盾或者与 $con_f\in R\text{-}mups$ 矛盾.

另外,如果 C_b 不可满足的,$Cover=cover(T,C)$;否则,$cover(T,C)\subset Cover$.综上情形〈1〉、情形〈2〉,$FC_1\cap FC_2$ 中的元素是 FC_1 与 FC_2 笛卡尔乘积的极小值,$R\text{-}mups=r\text{-}mups(T,C)$,$cover(T,C)\subseteq Cover$.

(3) 假设当 $n=m$ 时成立,证明当 $n=m+1$ 时成立:从左向右扫描 $ETr_1,\ldots,ETr_{m+1}$,查找第 1 个连续的有相同分支结点的 $ETr_p,ETr_{p+1},\ldots,ETr_q$(其中,$1\leqslant p<q\leqslant m+1$),共 $q-p+1$ 棵扩展树,设它们的分支结点 $Bnode=(a_b,C_b,l_b)$.合并 $ETr_p,ETr_{p+1},\ldots,ETr_q$ 为一棵扩展树 ETr_b,对应的 B_b 为合并 $B_p,B_{p+1},\ldots,B_q$ 得到的分支,B_b 的标记是 $B_j(p\leqslant j\leqslant q)$父分支的标记.用 ETr_b 替换 $ETList$ 中的 $ETr_p,ETr_{p+1},\ldots,ETr_q$ 得到 $ETList=\{ETr_1,\ldots,ETr_{p-1},ETr_b,ETr_{q+1},\ldots,ETr_{m+1}\}$,相应的 $BranchList=\{B_1,\ldots,B_{p-1},B_b,B_{q+1},\ldots,B_n\}$.这样,$ETr_b$ 是 B_b 对应的一棵扩展树,即,ETr_b 中的分支结点不是 $Bnode$,不妨设 ETr_b 的分支结点为 $Fnode$.$(UC_p,OC_p,FC_p)=conflictCompute(C,B_p,T),\ldots,(UC_q,OC_q,FC_q)=conflictCompute(C,B_q,T)$,$UC_b=UC_p\cup\ldots\cup UC_q$,$OC_b=OC_p\cup\ldots\cup OC_q$,$FC_b=\{FC_p,\ldots,FC_q$ 的笛卡尔乘积的极小值$\}$.令 $FO_b=\{con|con\in FC$ AND $con\cap path(Fnode,ETr_b)=\varnothing\}$,$FC_b=FC_b-FO_b$,$OC_b=OC_b\cup FO_b$,其他扩展树不影响 $Fnode$ 的子孙结点的可满足性,则有$(UC_b,OC_b,FC_b)=conflictCompute(C,B_b,T)$.这样处理完之后,$BranchList$ 中分支数为 $m-(q-p)$,也就是说,C 的 R-MUPS 扩展得到的分支数小于 m,得到 $R\text{-}mups=r\text{-}mups(T,C)$,$cover(T,C)\subseteq Cover$.

综合情形(1)~情形(3),得证. □

R-MUPS 算法高效是因为当本体中不可满足概念描述的复杂度高:

(1) 析取构造符较多,即,概念的 R-MUPS 或者是 MUPS 扩展的分支较多.R-MUPS 算法根据概念之间的覆盖关系对分支中的冲突进行分类,直接分离出本体的 MIPS,将剩余的冲突参与到分支合并的计算,这样可以极大地减少分支合并过程中冲突极小化的计算量;

(2) 合取构造符较多,概念 R-MUPS 扩展的深度深.也就是说,不可满足概念之间的覆盖关系多.计算一个不可满足概念的 R-MUPS 过程中,大部分冲突都是分支自身的冲突,即,本体的 MIPS,也减少了计算量;同时,计算一个概念的 R-MUPS 过程中,可以直接得到覆盖概念的 R-MUPS.

不可满足概念的概念描述的复杂度和概念之间的覆盖关系,影响 R-MUPS 算法实现调试的效率.

概念的可满足性由 Tableau 算法判定,其计算复杂度为 PSPACE-完全的[32],Schlobach 计算概念 MUPS 的复杂度也是 PSPACE-完全的[17],本文计算 R-MUPS 并不增加其计算复杂度.

空间方面,函数 *R-Mups-ComputeFunction* 计算概念的 R-MUPS 过程中采用深度优先遍历,即:在内存中每次仅保留一个分支,而且计算 R-MUPS 的同时也得到了概念的覆盖概念集.在计算 R-MUPS 时缓存概念所覆盖的

概念,并且对可满足概念也保存其相关的可满足概念,实现快速计算 MIPS.

4 实验评测

本文实验是在 PC 机 Windows XP 操作系统(Intel(R) Core(TM)2 Duo CPU E7500 2.93GHz,2.00GB 的内存)下运行.在测试数据方面,首先利用公理数和公理长度自动生成不同规模本体,评价调试算法对不同数据强度的处理性能,然后利用现有开源本体及其扩建本体,评价调试算法对现实本体数据的处理性能.在进行比较的调试方法选取方面,我们选择 Schlobach 的 MUPS 方法进行比较.在所有本体调试策略中,该方法是与本文工作调试定位角度一致的,都是通过寻找不可满足概念间接实现极小不可满足术语集的判定.相对其他调试方法,如利用最大可满足术语集策略、利用术语集的扩展收缩策略、利用逻辑语言转换等方案,虽然都能从不同角度定位本体逻辑冲突,但无法为用户提供不可满足概念最小保持子集做调试参考.因此,在同一数据和测试平台下比较 MUPS 算法和 R-MUPS 算法具有现实意义.同时,我们还针对 R-MUPS 调试方法中是否引入概念覆盖进行了对比测试,即,比较计算所有不可满足概念的 R-MUPS 与计算部分不可满足概念的 R-MUPS,其中,计算部分不可满足概念的 R-MUPS 是利用概念间的覆盖关系,不对被已经计算过 R-MUPS 的概念覆盖的概念再计算.通过实验对比,说明在 R-MUPS 调试方法中引入概念覆盖关系的意义.

4.1 自动生成的本体测试集

自动生成不同规模的非循环 $\mathcal{ALC}$ 本体,需要设定以下几个参数:本体规模 *ConceptAmount*(TBox 的公理总数,每个公理描述 TBox 中唯一的原子概念)、最大公理长度 *maxAxiomLength*(公理右边部分中⊓和⊔构造符数量的最大值)、基本原子概念和角色.按照分层思想生成不同规模的非循环 $\mathcal{ALC}$ 本体,包含所有 $\mathcal{ALC}$ 构造符及其构造的复合概念形式,同时,在一定程度上控制概念扩展的最大深度.首先,非循环 $\mathcal{ALC}$ 本体保证算法的可判定性;其次,随机选择概念描述构造符保证算法的完备性;最后,本体中概念间覆盖关系的复杂度的不确定性,保证测试本体并不有利于本文算法.

自动生成本体方法:构建 TBox T,初始时 T 由基本原子概念和角色构成.逐个向 T 增加新概念及其公理,直到设定的本体规模值(*ConceptAmount*).完成新概念 C 及其公理的构造后,确定被 C 包含的概念数 Son_c(子概念数),并将 $\langle C_x,Son_c\rangle$ 插入到队列中,队列实现宽度优先生成新概念.采用队列的数据结构按照分层的方式完成新概念公理的构造,取队列头部元素 $\langle C_f,N_s\rangle$,接着构造概念 C_f 的 N_s 个子概念 C_s 及其公理.构造公理之前,首先确定公理长度,然后选择公理的构造符为⊓或者⊔,按照已经确定的构造符得到复合概念 *complex*,即,$C_s \sqsubseteq C_f \sqcap complex$.根据构造符构造 *complex*:(1) 构造符为⊓,则 $complex, C \rightarrow D|\neg D|\exists R.D|\exists R.\neg D|\forall R.D|\forall R.\neg D|complex \sqcap C$;(2) 构造符为⊔,则 $complex, C \rightarrow D|\neg D|\exists R.D|\exists R.\neg D|\forall R.D|\forall R.\neg D|complex \sqcup C$.其中,$D$ 为已经存在的且不超过 C_f 层概念或基本原子概念.

图 6 和图 7 给出最大公理长度为 5、本体规模分别为 3 000 和 5 000 的参数设置下随机生成 20 个本体的调试时间,横坐标表示随机生成的本体,由本体的不可满足概念数和 MIPS 数标记,以不可满足概念数递增的顺序排列;纵坐标表示本体调试时间.总体上看:图 6 规模相同的本体,本体调试时间随着本体中不可满足概念数的增加而增加,当本体中不可满足概念数相近时,调试时间随着本体 MIPS 数的增加而增加;图 7 扩大本体规模,生成本体的不可满足概念数增加,相应的两种方法实现本体调试时间均会提高,不可满足概念数相近时,本体的 MIPS 越多,本体调试时间越长.用公式$(T_m-T_r)/T_m$计算 R-MUPS 方法较 Schlobach 方法的提高效率,累加各组数据的提高效率,取平均值作为图中给定参数下 R-MUPS 方法较 Schlobach 方法的提高效率的平均值,其中,T_m 和 T_r 分别是 R-MUPS 方法和 Schlobach 方法的调试时间.

图 6 中,计算所有不可满足概念 R-MUPS 方法提高效率的最大值为 45%(第 17 组数据),平均值为 20%(计算 20 组数据提高效率的平均值);计算部分不可满足概念的 R-MUPS 方法提高效率的最大值为 49%(第 17 组数据),平均值为 26%.图 7 中,计算所有不可满足概念 R-MUPS 方法提高效率的最大值为 20%(第 16 组数据),平均值为 14%;计算部分不可满足概念的 R-MUPS 方法提高效率的最大值为 26%(第 5 组数据),平均值为 20%.

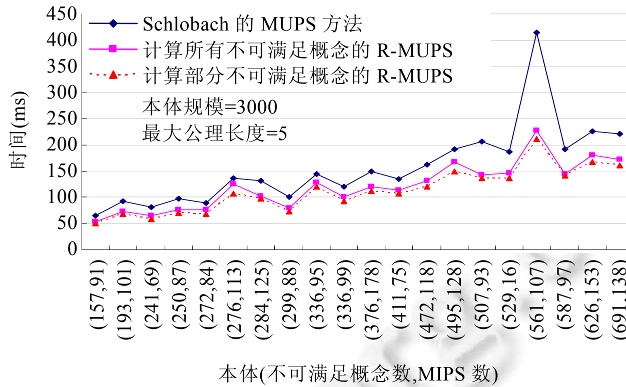

Fig.6 Texting automatically generated ontologies 1(different in scales)

图 6 自动生成本体测试 1(不同规模)

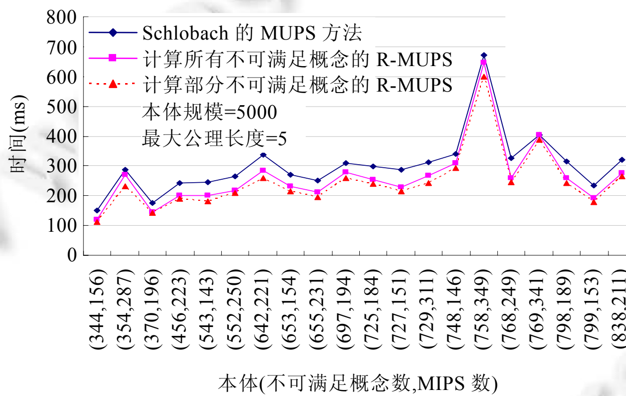

Fig.7 Texting automatically generated ontologies 2(different in scales)

图 7 自动生成本体测试 2(不同规模)

图 8 和图 9 给出本体规模为 1 000、最大公理长度分别为 5 和 10 的参数设置下,随机生成 20 个本体的调试时间.图 8 和图 9 满足前边说明的本体中不可满足概念数和 MIPS 数影响本体调试时间.对图 9 中 R-MUPS 算法调试时间高于 MUPS 调试时间的本体进行分析,都具有共同的特点是,本体中概念描述简单和概念间的覆盖关系较少.最大公理长度间接地限制本体中概念描述的复杂度,而概念描述的复杂程度隐含概念之间的覆盖关系;如果生成的本体的 MIPS 数小于不可满足概念数,说明存在概念之间的覆盖关系,本体不可满足概念数与 MIPS 数的差值越大,本体中概念的覆盖关系可能越复杂.图 9 中:计算所有不可满足概念 R-MUPS 方法提高效率的最小值为–36%(图 9 的第 7 组数据),平均值为 19%;计算部分不可满足概念的 R-MUPS 方法提高效率的最小值为–26%(图 9 的第 7 组数据),平均值为 23%.

综合自动生成本体测试集的实验结果,相较于增加本体规模,增加概念描述的复杂度消耗更多的本体调试时间.当本体概念描述复杂或不可满足概念之间的覆盖关系较多时,R-MUPS 算法相较于 MUPS 算法更为高效,效率平均提高 20%.

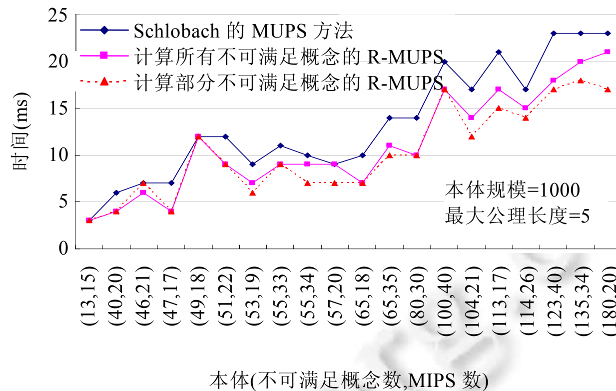

Fig.8 Texting automatically generated ontologies 3(same in the scale)

图 8 自动生成本体测试 3(相同规模)

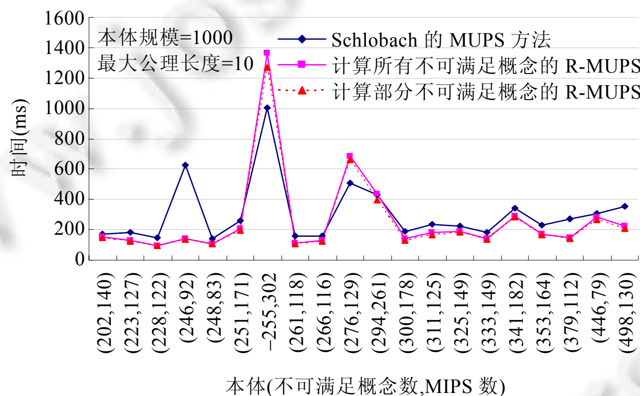

Fig.9 Texting automatically generated ontologies 4(same in the scale)

图 9 自动生成本体测试 4(相同规模)

4.2 现有本体及其扩建本体

现有本体 DICE-A[15],Mad-Cow,MGED(http://www.mindswap.org/ontlogies/debugging),OBI,hec-onto-drugs, openGalen(http://www.ontoware.org).openGalen 是开源型医学术语集,目前可用的开源资源包括一个复杂的本体开发环境和一个大型的描述逻辑本体用于医疗领域;DICE-A 是医学本体 DICE 中的解剖片段,包含有关解剖、病因和形态的概念(例如 $Brain\sqsubseteq CentralNervousSystem$,$Brain\sqsubseteq BodyPart$,$CentralNervousSystem\sqsubseteq NervousSystem$,$BodyPart\sqsubseteq\neg NervousSystem$);生物医学本体 OBI 包括研究设计、协议、使用的仪器、所产生的数据和对数据进行分析的类型,从功能基因组学研究本体,将包含功能基因组学研究和那些特定领域方面的术语;MGED 本体描述在微阵列实验所用的样品,主要目的是为微阵列实验的注解提供标准的条款,这些条款将使实验要素能够结构查询(例如);Mad-Cow 和 hec-onto-drugs 是规模较小的本体(见表 1).

先将本体修改为非循环 $\mathcal{ALC}$ 本体,例如删除原本体中的个体,将数据类型属性改成对象类型属性,同时修改相应的概念描述.如果概念 C 的描述中包含带有定义域 *Range* 或者值域 *Domain* 的属性 R,即 $C\sqsubseteq\exists R.C'$,则为 C 增加 $C\sqsubseteq Range$ 或者全称概念描述 $C\sqsubseteq\forall R.Domain$.修改之后,不改变原本体概念的可满足性,保证 C 在原本体中是可满足的,那么在对应的修改之后的本体中也是可满足的;反之亦然.然后修改一致本体,例如增加一些新概念,其概念描述与本体中原有的一些概念矛盾,进而使得本体不一致.