一种基于ILP和ASP的学习B语言描述的动作模型方法

刘 振 张志政
(东南大学计算机科学与工程学院 南京 211189)

摘 要 动作模型学习可以使Agent主动适应动态环境中的变化,从而提高Agent的自治性,同时也可为动态域建模提供一个初步模型,为后期的模型完善和修改提供了基础。通过结合归纳逻辑程序设计(Inductive Logic Programming,ILP)和回答集程序设计(Answer Set Programming,ASP),设计了一个学习B语言描述的动作模型算法,该算法可以在混合规模的动态域中进行学习,并采用经典规划实例验证了该学习算法的有效性。

关键词 动作模型学习,动作语言B,归纳逻辑程序设计,回答集逻辑程序设计

中图法分类号 TP311 **文献标识码** A **DOI** 10.11896/j.issn.1002-137X.2015.1.049

Learning Action Models Described in Action Language B by Combining ILP and ASP

LIU Zhen ZHANG Zhi-zheng
(School of Computer Science and Engineering, Southeast University, Nanjing 211189, China)

Abstract Action model learning is beneficial to autonomous and automated systems. If an Agent can update its action model according to the changes occurred in the environment, it can be more adaptable to the world and operate more effectively. Simultaneously, action model learning can provide modeling dynamic domain with an initial rough model which is the foundation for further improvement and modification. We designed an algorithm used for learning action models described in language B by combining ILP and ASP. This algorithm can work on dynamic domains consisting of objects of different scale. In the experiments, we tested the learning algorithm through using classic planning cases and verified the soundness of the learning algorithm.

Keywords Action model learning, Action language B, Inductive logic programming, Answer set programming

1 引言

作为一种关于动作规则的学习,动作模型学习是一个从一系列观察到的状态中进行动作规则归纳的过程,其主要任务是从状态序列中学习到每一个动作执行时所需要满足的条件,以及这个动作的成功执行所带来的影响。动作模型学习可以使Agent主动适应环境中的意外变化(例如环境中新的未知Agent的出现、硬件故障等),进而增强系统的自治性[1,2]。同时,动作模型学习也可以为动态域建模提供一个初步模型,为后期的模型完善和修改提供了基础[3]。

目前基于非单调推理语言的关于动作模型的学习主要有基于STRIPS语言和基于回答集编程的动作语言。前者以经典的统计、分类、归纳逻辑程序设计等机器学习算法为主要手段,但是所学习的动作模型中,动态域的流是不分类的,在实际规划、诊断等应用中不可避免地存在容变能力(elaboration tolerance[4])低的问题,但是已存在对观察不完全、噪音处理的高效方法。基于回答集程序的动作语言的动作模型学习主要通过非单调推理手段,来更好地解决动态域中的框架问题(frame problem)、分支问题(ramification problem)、资格问题(qualification problem)[5],但还存在如下问题:

1)部分算法基于命题逻辑语言,不能满足实际需要;

2)现有方法以逻辑公理来描述学习策略,效率低,不适合大数据上的学习。

本文结合归纳逻辑程序设计ILP和回答集程序设计ASP设计了一个学习B语言描述的动作模型的算法,并从国际规划竞赛(International Planning Competition,IPC[6])和文献[5]中选择4个标准动态域,完成了对算法有效性的验证。

本文第2节介绍动作语言B以及学习的输入和学习任务;第3节介绍学习B语言描述的动作模型算法;第4节介绍实验与分析;第5节介绍相关工作;最后,总结全文并对下一步工作展望。

2 背景知识

本文所研究的动作模型学习算法基于动作语言B,对B语言描述的动态域问题的求解是通过将域描述进行翻译得到ASP程序,然后使用ASP推理机对这一ASP程序进行求解。本节首先介绍基于ASP的动作语言B的语法,然后介绍本文学习算法的输入及学习任务。

到稿日期:2014-02-09 返修日期:2014-05-05 本文受国家自然科学基金项目(60803061),江苏省自然科学基金项目(BK2008293),东南大学科技基金项目(XJ2008315)资助。

刘 振(1989-),男,硕士生,主要研究方向为回答集逻辑程序,E-mail:53387534@qq.com;张志政(1980-),男,博士,副教授,硕士生导师,主要研究方向为回答集逻辑推理,E-mail:zzzhang@seu.edu.cn(通信作者)。

2.1 动作语言 B

动作语言是用于描述动态域的自然语言的一种形式化模，也是描述状态转换图的一种工具，下面将介绍动作语言 B 的语法。

首先介绍动作语言 B 的一些基本术语：一个动作符号集 Σ 由 3 个非空符号集析取构成：静态属性(statics)S、流(fluent)F 和动作(action)A。

静态属性用于描述对象的不可变属性部分，例如 *block*(*b*)表示是一块积木，这种“是一块积木”的属性是不会被改变的。

流表示对象的可变属性或者对象之间的关系，例如 *on*(*b*1,*b*2)表示积木 *b*1 在积木 *b*2 的上面，流集合 F 分为惯性流 F_i 和定义流 F_d。惯性流的真值会被某个动作所改变，如果没有发生这样的动作，那么这个流的真值将会从当前状态保持到后继状态。定义流的值不能被动作所改变，只能够通过其他流的真值来推出，并且根据封闭世界假设(Closed World Assumption,CWA)，当没有证据表明一个定义流的真值为真时，它的真值为假。

静态属性 S、流 F 统称为域属性。一个域文字是一个域属性 p 或者其否定形式 $-p$。如果一个域文字是一个流，那么它是一个流文字，否则它是一个静态文字。

动作集合 A 包含了动态域中的所允许的动作，例如动作 *stack*(*arm*0,*b*1,*b*2)表示利用机械臂 *arm*0 将积木 *b*1 放到积木 *b*2 上。

定义 1(声明 statements)　动作语言 B 允许下列 3 种形式的声明：

1)因果规则Causal Low

a causes l_i if p

2)状态约束 State Constraint

l if p

3)执行条件 Executability Condition

impossibile $a_0,\cdots,a_k$ if p

因果规则 a causes l_i if p 表示如果一个状态满足条件 p，那么在这个状态中执行动作 a 会产生影响 l_i，其中 p 是一个域文字集合，l_i 是惯性流，也被称为动作的结果或影响。

状态约束 l if p 表示任何一个满足条件 p 的状态一定满足条件 l，其中 l 是任意的域文字。状态约束一般用于定义动作的间接影响，如何定义动作的间接影响被称为分支问题(ramification problem)。

执行条件 impossibile $a_0,\cdots,a_k$ if p 表示在满足条件 p 的状态中至少有一个动作 $a_i(0\leqslant i\leqslant k)$不能够被执行，即 $a_0,\cdots,a_k$ 不能同时被执行。

动作语言 B 是动作语言族中一个新成员，相对于其他动作语言，其语法简练、直观、易用。目前 B 语言已经逐步在规划、诊断、问答系统中得到推广和应用。

鉴于回答集逻辑程序语言的固有优势，设计和实现基于 B 语言的能够自动学习动作模型的 Agent 具有潜在的实用价值。又由于 B 语言上的推理(包括模型更新)可归结为基于回答集程序的信念变化，因此基于 B 语言的动作模型学习也将为基于动作语言的动作模型学习和信念变化提供可借鉴的理论和方法。

2.2 训练集形式化及学习任务

我们称 Agent 对世界某一时刻描述为一个状态(state)，状态从语法上是一个实例化文字集和一个执行动作所构成的集合。集合中的各个元素(流)从语义上描述了世界的各方面属性以及对象之间的关系。

为了表示一个状态，我们需要引入关系 $holds(f,i)$，表示流 f 在第 i 时刻为真，我们用 $holds(f,i)$ 表示 f，$-holds(f,i)$表示 $-f$。1 个状态 σ 是 1 个由 $holds(f,i)$ 与 $-holds(f,i)$ 组成的集合。

为了表示 1 个动作，我们引入关系 $occurs(a,i)$，它表示在第 i 时刻发生了动作 a。

我们的训练集由规划组成，每一个规划是一个状态动作序列，即 $plan=\langle s_0,a_0,s_1,a_1,\cdots,s_n\rangle$。在任意一个状态 s_i 中，动作 a_i 的执行导致状态进入后继状态 s_{i+1}，s_0 为初始状态，s_n 为结束状态，从 s_0 到达 s_n 的过程中所执行的动作序列以及中间状态构成了一个规划。

例 1 给出动态域 Circuit 中的一个规划，图 1 为动态域 Circuit 中的电路图。

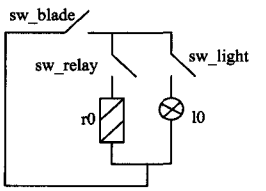

图 1　动态域 Circuit 电路图

例 1　动态域 Circuit 中的一个规划

$-holds(on(sw\_light),0)-holds(on(sw\_relay),0)-holds(active(r0),0)holds(on(sw\_balde),0)-holds(active(l0),0)occurs(toggle(sw\_blade),0)$

$-holds(on(sw\_blade),1)-holds(active(r0),1)-holds(active(l0),1)-holds(on(sw\_relay),1)-holds(on(sw\_light),1)occurs(toggle(sw\_light),1)$

$-holds(on(sw\_blade),2)-holds(active(r0),2)-holds(on(sw\_relay),2)-holds(active(l0),2)holds(on(sw\_light),2)$

$occurs(toggle(sw\_relay),2)$

我们的学习任务是从这些状态序列，即规划中学习因果规则，然后将因果规则和惯性定律结合在一起形成一个描述动态域的动作模型，该动作模型应与原动作模型等价。

状态约束 *State Constraint* 以及执行条件 *Executability condition* 可以使动作模型的表达更加优雅和简洁，它们可看作是因果规则的简化，所以目前本文仅考虑因果规则学习，状态约束和执行条件的学习将作为下一步工作。对于流的分类，我们可以通过如下措施达到等价效果：所有的流都是惯性流，并且在初始状态指定所有流的真值。由于我们在初始状态中给定了所有流的真值，后继状态中流的真值或者是因为惯性从前驱状态中保持而来或者是被动作所改变，因此流在每个状态中都有确定的真值。

通过在形如例 1 给出的规划中使用本文设计的动作模型学习算法，我们可以得到一条关于动作的因果规则。

例 2　动作 toggle

toggle(*Switch_relay*1) *causes on*(*Switch_relay*1) *if* $-active(Relay1),-on(Switch\_relay1)$

3 动作模型学习算法设计

本节首先介绍动作模型学习算法的框架，然后介绍极小化因果规则的生成、分类、泛化算法。

3.1 动作模型学习算法框架

如算法 1 所示，学习算法由 3 步组成：首先通过算法 Causal_Law_Generation 生成极小化(the most specific)因果规则，即从训练集中生成因果规则实例；然后通过算法 Causal_Law_Classification，依据规则表达的语义，即规则的动作谓词和结果谓词对因果规则实例进行分类；最后由算法 Causal_Law_Generalization 对每一类别中的实例化规则进行泛化，泛化分为常量的变量替换和规则条件部分不相关流的删除。

算法 1 B语言动作模型学习算法

```
输入：训练集；
输出：因果规则集；
BEGIN
1. causal_law_instances = Causal_Law_Generation(training_set);
2. causal_law_classification = Causal_Law_Classification(causal_law_
   instances);
3. causal_law_set = Causal_Law_Generalization(causal_law_classifi-
   cation);
4. return causal_law_set;
END
```

3.2 因果规则生成(Causal Law Generation)

一条因果规则 a causes l_i if p 由 3 部分组成：动作 a、前提条件 p 和一个结果文字 l_i。算法首先从训练集中抽取三元组$\langle state_i, a_i, state_{i+1}\rangle$，$state_i$ 表示发生动作 a_i 时的状态，记作 $pre\text{-}state(a_i)$，$state_{i+1}$表示 $state_i$ 的后继状态，即由动作 a_i 所导致进入的新状态，记为 $successor\text{-}state(a_i)$。

定义 2(动作的影响) 动作的影响是指动作的后继状态与发生动作时的状态的不同，记为 $delta\text{-}state(a_i)$，$delta\text{-}state(a_i)=\{l \mid l\in successor\text{-}state(a_i)$且 $l\notin pre\text{-}state(a_i)\}$。

$delta\text{-}state(a_i)$是一个流文字集合，它包含了真值被动作 a_i 所改变的流。

下面给出因果规则生成算法：

算法 2 因果规则生成算法 Causal Law Generation

```
输入：训练集；
输出：因果规则实例；
BEGIN
1. causal_law_instance = ∅;
2. for each plan in training_set do
3.     i=0;
4.     while i<plan. length−1 do
5.         delta-state(a_i)=state_{i+1}−state_i;
6.         for each fluent_i in delta-state(a_i) do
7.             new causal_law;
8.             causal_law. a= a_i;
9.             causal_law. p= pre-state(a_i);
10.            causal_law. l = fluent_i;
11.            causal_law_instances = causal_law_instances ∪
               causal_law;
12.        end for
13.        i++;
14.    end while
15. end for
16. return causal_law_instances;
END
```

算法 2(因果规则生成算法)首先会遍历训练集中所有的规划，在每个规划中遍历所有的状态，对于每个状态 $state_i$ 中发生的动作 a_i，计算出 $delta\text{-}state(a_i)$；然后对于集合 $delta\text{-}state(a_i)$中的每一个流生成一条极小化因果规则 r：a causes l_i if p，其中 $a=a_i$，$l_i=l(l\in delta\text{-}state(a_i))$，$p=state_i$。通过这种方式生成的规则可以在状态 $state_i$ 中应用并会导致进入新状态 $state_{i+1}$，由于其中的参数为常量并且规则的 if 部分包含了整个状态，因此我们称这样的规则为极小化规则(the most specific rule)。

生成的极小化因果规则的谓词参数是常量，并且规则的 if 部分有大量的流，这些流中，一部分是真正的前提条件，另外一些流实际上与动作及其影响无关。因此，极小化规则实际上是对样本空间的过度拟合。

下一步工作是对极小化规则进行泛化，包括变量替换常量和消除因果规则中的不相关流。

3.3 因果规则分类(Causal Law Classification)

在对规则进行泛化之前，需要对规则进行分类。我们需要将表达相同涵义的极小化因果规则分为一类，并假设这类规则是实例化自同一条因果规则，之后再对同一类中的规则进行泛化得到原始未被实例化的规则。下面我们介绍分类的标准。

定义 3(分类标准) 两条极小化因果规则 r：a causes l if p 和 r'：a' causes l' if p'属于同一类，如果有：

1)动作 a 和 a'谓词相同，并且其中的参数类型对应相同；

2)动作影响 l 和 l'谓词相同，并且其中的参数类型对应相同。

下面给出极小化因果规则分类算法：

算法 3 极小化因果规则分类算法 Causal Law Classification

```
输入 ：causal_law_instances;
输出 ：causal_law_classification;
BEGIN
1. causal_law_classification=∅;
2. for each r in causal_law_instances do
3.     bool flag = false;
4.     for each class_i in causal_law_classification do
5.         choose one causal law r' from class_i;
6.         if action_predicate_same(r,r') and effect_predicate_same
           (r,r') then
7.         class_i=class_i ∪r;
8.             flag = true;
9.             break;
10.        end if
11.    end for
12.    if flag == false then
13.        new class;
14.        class = class ∪r;
15.        causal_law_classification= causal_law_classification∪
           class;
16.    end if
17. end for
18. return causal_law_classification;
END
```

算法 3(极小化因果规则分类算法)遍历所有极小化规则,在遍历的过程中进行规则分类和创建新的类别。

经过分类,表达相同涵义的实例化规则聚在一起,接着通过对每一类中的规则进行泛化,得到原始未被实例化的规则。

3.4 因果规则泛化(Causal Law Generalization)

本小节将介绍如何泛化同一类别中的极小化规则。在泛化前我们需要提出一个假设,该假设为泛化的基础,然后给出泛化的流程。

假设 1 同一类别中的极小化因果规则实例化自同一条因果规则。

基于这样的假设,泛化包括两步:首先是进行变量替换,然后是删除规则条件部分的不相关流。

泛化流程如图 2 所示。

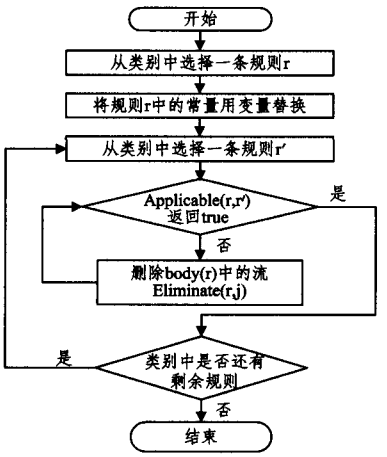

图 2 规则泛化流程图

首先从一个极小化因果规则类别中选取一条规则,然后对规则进行变量替换,使之可以在新的状态或场景中被实例化。接下来通过删除不相关流来保证经过变量替换的规则在同一类其它规则应用的状态中也可以被应用,即消除与训练集的过度拟合,不相关流的删除主用通过算法 Applicable(r, r')以及 Eliminate(r,j)来完成,具体操作步骤见下面 3 个小节。

3.4.1 规则选取

在对一个类别中的规则进行泛化时,我们首先需要选择一条类别中的规则 r 作为进行变量替换和不相关流删除的对象。这里的规则选取策略是选择执行条件,即 if 部分中流的数量最少的那条规则。

任意选择一条规则 r 进行变量化后,其 if 部分多余的流在与其他规则拟合时一定会被删除,最终剩余流的个数一定与 if 部分流数量最少的那条规则 r' 相同,否则 r 不能应用于原先应用的状态,因此我们首先选择条件部分流数量最少的规则。

3.4.2 变量替换

泛化的第一步是用变量替换常量。从一个类别中选择条件部分流数量最少的规则,将其中的常量用变量替换:同一常量用同一变量替换,不同常量用不同变量替换,如例 3 所示。

例 3 变量替换

将极小化规则 r:

$pickup(arm0,b3)$ $causes$ $-clear(b3)$ if $on\_table(b1)$, $-on(b3,b2)$, $-on(b0,b1)$, $-on(b2,b0)$, $empty(arm0)$, $clear(b3)$, $-holding(arm0,b3)$, $-holding(arm0,b2)$, $-holding(arm0,b1)$, $-holding(arm0,b0)$, $-on(b3,b3)$, $-on(b3,b1)$, $-on(b3,b0)$, $-on(b2,b3)$, $-on(b2,b2)$, $-on(b2,b1)$, $-on(b0,b3)$, $-on(b0,b2)$, $-on(b0,b0)$, $-on(b1,b2)$, $-on(b1,b0)$, $-on(b1,b1)$, $-on(b1,b3)$, $clear(b0)$, $clear(b1)$, $clear(b2)$, $on\_table(b0)$, $on\_table(b2)$, $on\_table(b3)$

进行变量替换,得到 r':

$pickup(Arm0,Block1)$ $causes$ $-clear(Block1)$ if $on\_table(Block2)$, $-on(Block1,Block3)$, $-on(Block4,Block2)$, $-on(Block3,Block4)$, $empty(Arm0)$, $clear(Block1)$, $-holding(Arm0,Block1)$, $-holding(Arm0,Block3)$, $-holding(Arm0,Block2)$, $-holding(Arm0,Block4)$, $-on(Block1,Block1)$, $-on(Block1,Block2)$, $-on(Block1,Block4)$, $-on(Block3,Block1)$, $-on(Block3,Block3)$, $-on(Block3,Block2)$, $-on(Block4,Block1)$, $-on(Block4,Block3)$, $-on(Block4,Block4)$, $-on(Block2,Block3)$, $-on(Block2,Block1)$, $-on(Block2,Block4)$, $-on(Block2,Block2)$, $clear(Block4)$, $clear(Block2)$, $clear(Block3)$, $on\_table(Block4)$, $on\_table(Block3)$, $on\_table(Block1)$, $Block4 \ != \ Block1$, $Block4 \ != \ Block3$, $Block4 \ != Block2$, $Block4 \ != \ Block1$, $Block3 \ != \ Block2$, $Block3 \ != \ Block1$, $Block2 \ != Block1$

经过变量替换后的极小化因果规则可以被实例化,但是规则的执行条件,即 if 部分仍然存在不相关流,这些流导致规则过度拟合训练集,在其他场景中不能够被满足,因此需要对不相关流进行删除。

3.4.3 删除不相关流

删除不相关流主要通过算法 Applicable(r,r')以及 Eliminate(r,j)来完成。算法 Applicable(r,r')将执行可应用性检查,规则 r 是极小化因果规则 r' 所属类别中经过变量替换而得的规则,Applicable(r,r')判断规则 r 在 $state_i$ 中是否可以应用,其中 $r'=a_i$ causes l_i if $state_i$, $l_i \in delta\text{-}state(a_i)$。

函数 Applicable(r,r')按照下述步骤工作:

1)生成 ASP 程序 $\pi(p)$,$p=r \cup state_i$,$\pi(p)$包含规则 r 所对应 ASP 形式的规则以及惯性公理并将 $state_i$ 中的时刻设为 0;

2)使用 ASP 推理机 Clingo 对程序 π 求回答集;

3)若第二步求得的回答集与动作 a_i 的后继状态,即 $successor\text{-}state(a_i)$相同,则函数 $Applicable(r,r')$返回 true;否则返回 false。

例 4 可应用性判定

r 为例 3 中经过变量替换的规则,r'为:

r':$pickup(arm0,b1)$ $causes$ $-clear(b1)$ if $on\_table(b1)$, $-on(b0,b2)$, $-on(b2,b1)$, $empty(arm0)$, $clear(b0)$, $-holding(arm0,b2)$, $-holding(arm0,b1)$, $-holding(arm0,b0)$, $-on(b2,b2)$, $-on(b2,b0)$, $-on(b0,b1)$, $-on(b0,b0)$, $-on(b1,b1)$, $-on(b1,b2)$, $on\_table(b2)$, $-on(b1,b0)$, $clear(b1)$, $clear(b2)$, $on\_table(b0)$

规则 r'与 r 属于同一类别,$r'=a_i$ causes l_i if $state_i$, $l_i \in delta\text{-}state(a_i)$, $state_i$ 为:

$holds(on\_table(b1),4)$ $-holds(on(b0,b2),4)$ $-holds(on(b2,b1),4)$ $holds(empty(arm0),4)$ $holds(clear(b0),4)$ $holds(on\_table(b2),4)$ $-holds(holding(arm0,b2),4)$ $-holds(holding(arm0,b1),4)$ $-holds(holding(arm0,b0),4)$ $-holds(on(b2,b2),4)$ $-holds(on(b2,b0),4)$ $-holds(on$

$(b0,b1),4)$ $-holds(on(b0,b0),4)$ $-holds(on(b1,b1),4)$ $-holds(on(b1,b2),4)$ $-holds(on(b1,b0),4)$ $holds(clear(b1),4)$ $holds(clear(b2),4)$ $holds(on\_table(b0),4)$

程序 $p=r\cup state_i$，对程序 $\pi(p)$ 求回答集得到 A，然而回答集 A 与动作 a_i 的后继状态不相同(忽略时刻的不同)，因此函数返回 false。

如果函数返回的是 false，表示经过变量替换后的规则 r 不能应用于 r' 被应用的状态，即规则 r 的执行条件不满足，原因在于规则 r 的 if 部分存在不相关流，过度拟合训练集。我们通过函数 Eliminate(r,j) 来完成不相关流的删除工作。函数 Eliminate(r,j) 将会删除规则 r 执行条件中的 j 个流，这样的删除方式共有 C_n^j 种(n 为规则 r 执行条件中流的数量)。算法 4 中对于不相关流的删除采用最少删除原则，即在函数返回 false 时，被删除流的数量 j 从 1 递增至 n，每种数量的删除有 C_n^j 种方式，在每进行一次流删除后使用函数 Applicable(r,r') 进行判断，一旦 Applicable(r,r') 返回 true，流删除工作便停止。此时的规则 r 可以应用于 r' 所被应用的状态，从而完成 r 与 r' 的拟合。

例 5　流删除

在例 4 中，r 未经过流删除时，Applicable(r,r') 返回 false，在 r:*a causes l if p* 的条件部分 p 中删除不相关流以及不等式后，r 为：

pickup(*Arm*0,*Block*1) *causes* $-$*clear*(*Block*1) *if on_table*(*Block*1), $-$*on*(*Block*2, *Block*3), $-$*on*(*Block*3, *Block*1), *empty*(*Arm*0), *clear*(*Block*2), $-$*holding*(*Arm*0, *Block*3), $-$*holding*(*Arm*0, *Block*1), $-$*holding*(*Arm*0, *Block*2), $-$*on*(*Block*3, *Block*3), $-$*on*(*Block*3, *Block*2), $-$*on*(*Block*2, *Block*1), $-$*on*(*Block*2, *Block*2), $-$*on*(*Block*1, *Block*1), $-$*on*(*Block*1, *Block*3), *on_table*(*Block*3), $-$*on*(*Block*1, *Block*2), *clear*(*Block*1), *clear*(*Block*3), *on_table*(*Block*2), *Block*3 != *Block*2, *Block*3 != *Block*1, *Block*2 != *Block*1

此时，Applicable(r,r') 的返回值为 true。可以看到，在经过流删除后规则 r 可以应用到原先应用 r' 状态中，即完成了规则 r 与 r' 的拟合。通过与同一类别中后续其它规则的拟合，r 中的不相关流会得到进一步删除，最终我们会得到与所有规则都进行过拟合，也即泛化完全的规则 r，下面给出极小化规则泛化算法。

算法 4　极小化规则泛化算法 Causal Law Generalization

```
输入 :causal_law_classification;
输出 :causal_law_set;
BEGIN
1. causal_law_set = ∅;
2. for each class_i in causal_law_classification do
3.     choose the shortest causal law r from class_i;
4.     r = Variable_Substitution(r);
5.     for each r' in class_i do
6.         bool flag = Applicable(r,r');
7.         if flag == false then
8.             for j = 1; j ≤ NumofFluents(Pre-condition(r)) and
               flag == false; ++j do
9.                 causal_law_temp = Eliminate(r,j);
10.                for each r'' in causal_law_temp do
11.                    bool flag2 = Applicable(r'',r');
12.                    if flag2 == true then
13.                        r = r'';
14.                        break;
15.                end if
16.            end for
17.        end for
18.    end If
19.    end for
20.    causal_law_set= causal_law_set∪r;
21. end for
22. return causal_law_set;
END
```

下面给出规则泛化完全、消除过度拟合的一个例子。

例 6　泛化完全的规则 r

例 5 中的规则 r 经过与同类别中的其他极小化规则拟合后，进一步删除不相关流，得到：

pickup(*Arm*0, *Block*1) *causes* $-$*clear*(*Block*1) *if clear*(*Block*2), $-$*holding*(*Arm*0, *Block*1), *on_table*(*Block*2), $-$*on*(*Block*1, *Block*1), $-$*on*(*Block*1, *Block*2), $-$*on*(*Block*2, *Block*1), $-$*holding*(*Arm*0, *Block*2), $-$*on*(*Block*2, *Block*2), *clear*(*Block*1), *empty*(*Arm*0), *on_table*(*Block*1), *Block*2 != *Block*1

因为 Block world 训练集中，规模最小的规划问题包含两块积木，Agent 的观察始终包含两块积木，因此学习的结果也包含描述两块积木之间关系的流。至此，规则经过与数据中其他所有规则的拟合，完成了泛化，泛化后的规则可以应用到其他规则原先所应用的状态或场景。

4　实验结果与分析

我们从国际规划竞赛 IPC 和文献[5]中选择了 4 个标准动态域 Block world、Driver log、Briefcase、Circuit，并完成了用动作语言 B 对这 4 个动态域的描述，通过加入规划模块使之成为一个经典规划问题，然后使用 AltoASP 转换器将规划问题转换为一个 ASP 程序，进而使用推理机 Clingo 求回答集，将所得回答集按照时间进行排序即为一个状态动作序列，训练集和测试集由状态动作序列组成，状态动作序列也称为规划，我们对算法的评估是通过正确率以及与原动态域描述对比来进行的。下面首先介绍用于评估学习到的动作模型的标准，即正确率的定义，然后介绍在这 4 个标准动态域中的实验情况。

本文实验均在 32 位 Windows 7 系统、4GB 内存、Interl Core2 Quad 双核 2.67GHz 环境下进行。

下面给出正确率的定义，正确率用于评估学习结果在测试集上的表现。

定义 4(正确率)　一个动作模型关于一个状态是正确的，如果有：

1)此状态满足原动作模型中的动作规则的发生条件，那么此状态也满足学习到的动作规则的发生条件；

2)结合这个动作模型和状态进行推理所得到的下一时刻状态与原动作规则所导致的后继状态相同。

记(CP)为一个规划中的动作模型关于状态正确的个数，$|P|$ 表示规划中状态的个数，学到的动作模型关于整个测试集的正确率为：

$$CP(T)=\frac{\sum\limits_{P_i\in T} C(P_i)}{\sum\limits_{P_i\in T}|P_i|}$$

下面首先给出在域 Block world 上学习的结果，然后给出在测试集上对学习结果的测试情况，最后选择一条学习到的因果规则进行分析说明。

在包含 120 个规划的训练集上学习过程耗时 726.03 秒。

由于我们的数据集是确定性数据集，数据集中不包含噪音以及观察不完全情况，因此正确率可以达到 100%。我们将学习到的动作模型在测试集(40 个规划)上进行测试，得到的正确率为 100%。下面对学习到的动作模型中的一条因果规则 r_l(l 表示 learned，即学习到的规则)进行分析说明。

r_l: *unstack* (*Arm*1, *Block*1, *Block*2) *causes* − *empty* (*Arm*1) if *on* (*Block*1, *Block*2), *empty* (*Arm*1), *clear* (*Block*1), −*on*(*Block*2, *Block*2), −*holding*(*Arm*1, *Block*2), −*holding* (*Arm*1, *Block*1), − *on _ table* (*Block*1), − *clear* (*Block*2), − *on* (*Block*1, *Block*1), − *on* (*Block*2, *Block*1), *Block*1!=*Block*2

这条规则表示机械臂 *Arm*1 将积木 *Block*1 从积木 *Block*2 上拿起会导致机械臂非空。下面将从两个方面对这条规则进行说明：

(1)原动作模型中不存在表示动作 *unstack*(*Arm*1, *Block*1, *Block*2)会带来影响-*empty*(*Arm*1)的规则。这是因为在原动作模型中我们通过状态约束来表示各个属性之间的关系以及动作的间接影响，而在学到的动作模型中，我们只使用了因果规则，所有引起状态的变化均作为动作的直接影响。状态约束可以使得动作模型表达更优雅和简洁，但是没有状态约束规则并不影响动作模型的实际应用。

(2)学到的动作模型中的规则其条件部分有相对较多的流，这是因为：

1)原动作模型中有执行条件声明，它表达动作在某些条件下不能被执行。在学习到的动作模型中没有执行条件声明，因此，若要表达动作在什么样的条件下能发生需要在前提条件，即 if 部分增加一些流，学习到的因果规则其表达能力相当于原动作模型中的因果规则声明和执行条件声明两者的结合。

2)泛化不够彻底。在规则 r_l 的前提条件部分中有流 −*holding*(*Arm*1, *Block*2), −*holding*(*Arm*1, *Block*1)以及不等式 *Block*1!=*Block*2，这 3 个文字可以进一步被泛化为 −*holding*(*Arm*1, *Block*)，其中 *Block* 变量取值于所有 *block* 常量。

3)没有状态约束。在学习到的动作模型中我们没有表达流之间关系的状态约束，例如，如果我们有状态约束：

−*holding*(*Arm*1, *Block*2) if *empty*(*Arm*1)

表示任意时刻一块积木只能处于一个位置的规则：

−*on_table*(*B*) if *on*(*B*, *B*1)

on(*B*, *B*1) if *on*(*B*, *B*2), *B*1!=*B*2

−*on*(*B*, *B*1) if *on_table*(*B*)

以及表示积木是否为 clear 的规则：

−clear(B1) if *on*(*B*, *B*1)

那么我们学习到的规则的前提条件将会得到进一步简化，例如，根据上面几条状态约束我们会得到 r_l'：

r_l': *unstack* (*Arm*1, *Block*1, *Block*2) *causes* − *empty* (*Arm*1) if *on* (*Block*1, *Block*2), *empty* (*Arm*1), *clear* (*Block*1), *Block*1!=*Block*2

被省略的流可由上述状态约束推出。值得一提的是，我们可以根据学习到的因果规则通过概率统计、数据挖掘等方法进一步学习状态约束，然后利用状态约束去进一步泛化学习到的动作模型。

在动态域 Driver log、Circuit、Briefcase 中，我们得到的正确率为 100%，与原动态域模型的对比分析与在动态域 Block world 中的类似。

5 相关工作

本文提出的动作模型学习算法基于动作语言 B，采用离线学习方式。我们的学习素材是规模混合的动态域规划实例，学习结果可以应用到新的动态域规划问题，并且离线学习方式使得我们可以在丰富的大规模训练实例中进行学习，从而使学习结果更为精确，与已有文献[1,7]相比，不同之处在于：

2012 年文献[1]提供了一种使用 Reactive ASP[8] 来进行动作模型学习的方法：该方法的学习过程是首先利用在线 ASP 求解器 oClingo[8] 对增长逻辑程序(incremental logic program)中的静态知识库 B(包含动作规则的生成公理)求回答集，求得的每一个回答集对应于一个动作模型，所有的回答集便组成了初始解空间，然后在训练集上对每一个动作模型进行测试，测试失败的动作模型被淘汰，最后剩下的动作模型即为学习到的结果。然而问题在于：

1)随着流(用于描述状态或世界中的某种属性)的种类增多，以及实例化后数量的增加，所产生的流组合数量庞大，这些组合都成为动作可能的执行条件和影响，这种类似穷举的方法会产生大量回答集，因此初始解空间很大；

2)该方法实际上基于命题逻辑语言，因此没有提供参数泛化的操作，不能满足实际应用。

文献[7]首先以一种精简的方式来为规则编码：一条规则由一个事实集合以及一张语义表来表示，当确定动作语言的语义后，学习规则的任务就被简化至寻找一个事实集合。在给出编码方式后，文献[7]中指出域描述 $DD=\langle A^D, H^{CT}\rangle$ (Domain Description，CT 表示当前时间，H 表示历史状态，主要包含之前的观察和发生的动作，AD 表示动作描述)。程序 $DD=AD\cup H^{CT}\cup\pi_r,\pi_r$ 是一个包含约束的规则集，当程序 DD 不一致，即出现症状时就需要学习新的规则，使得 DD 一致。L 是一个规则生成公理集合，对于 DD 的一个归纳修正，即新动作的学习就简化为求解 $DD\cup L$ 的回答集，求得的回答集消除了 DD 的不一致，成为新学习到的动作。但存在的问题有：

1)如果求得的若干回答集都能够消除不一致，那么它们是否都作为新学到的动作；

2)仅通过是否消除不一致来确定是否为学习到的新动作，这样学习到的动作描述是不精确的，动作的执行条件可能存在不相关描述。

结束语 B 语言是近几年出现的一种基于非单调推理的动作语言，比 STRIPS 语言有更强的表达能力。本文的主要工作是设计了一个学习动作语言 B 描述的动作模型算法，并完成了对学习算法的实验，验证了算法的有效性。

下一步工作主要围绕以下 4 方面问题：1)状态约束 l if p 的学习，通过学习状态约束我们可以表达动作的间接影响以及流之间的关系；2)进一步泛化学习到的因果规则，通过状态约束的学习，利用流之间的关系可以将动作规则的前提条件

部分进行简化，同时通过状态约束表达动作的间接影响可以减少动作规则的数量；3)将动作规则拆分成因果规则声明和执行条件声明；4)处理噪音和观察不完全情况，包含噪音和观察不完全的数据集更加接近实际情况。

参 考 文 献

[1] Certicky M. Action Learning with Reactive Answer Set Programming: Preliminary Report[C]// The Eighth International Conference on Autonomic and Autonomous Systems (ICAS 2012). 2012: 107-111

[2] Estlin T, Gaines D, Chouinard C, et al. Increased Mars rover autonomy using AI planning, scheduling and execution[C]// IEEE International Conference on Robotics and Automation, 2007. IEEE, 2007: 4911-4918

[3] Yang Q, Wu K, Jiang Y. Learning action models from plan examples using weighted MAX-SAT[J]. Artificial Intelligence, 2007, 171(2): 107-143

[4] McCarthy J. Elaboration tolerance[OL]. 1997-09-09. http://www-formal.stanford.edu/juc./elaboration.html

[5] Gelfond M, Kahl Y. Knowledge Representation, Reasoning, and the Design of Intelligent Agents[OL]. http://redwood.cs.ttu.edu/~mgelfond/FALL-2012/book.pdf

[6] IPC(2003) [OL]. http://www.cs.cmu.edu/afs/cs/project/jair/pub/volume20/long03a-html/node37.html

[7] Balduccini M. Learning Action Descriptions with A-Prolog: Action Language C[C]// AAAI Spring Symposium: Logical Formalizations of Commonsense Reasoning. 2007: 13-18

[8] Gebser M, Grote T, Kaminski R, et al. Reactive answer set programming[M]. Logic Programming and Nonmonotonic Reasoning. Springer Berlin Heidelberg, 2011: 54-66

[9] Wang X. Learning by observation and practice: An incremental approach for planning operator acquisition[C]// ICML. 1995: 549-557

[10] Sil A, Yates A. Extracting STRIPS Representations of Actions and Events[C]//RANLP. 2011: 1-8

[11] Boose J H, Gaines B R. Knowledge acquisition for knowledge-based systems: Notes on the state-of-the-art[J]. Machine Learning, 1989, 4(3/4): 377-394

[12] Benson S. Learning action models for reactive autonomous Agents[D]. Stanford university, 1996

[13] 谢颖. 归纳逻辑程序设计初探[D]. 北京：北京师范大学哲学系，2008

[14] Stuart R, Peter N. 人工智能——一种现代方法(第2版)[M]. 姜哲，金奕江，等译. 北京：人民邮电出版社，2010

[15] Lorenzo D. Learning non-monotonic causal theories from narratives of actions[C]// NMR. 2002: 349-355

[16] Pasula H, Zettlemoyer L S, Kaelbling L P. Learning Probabilistic Relational Planning Rules[C]// ICAPS. 2004: 73-82

[17] Yang Q, Wu K, Jiang Y. Learning Actions Models from Plan Examples with Incomplete Knowledge[C]// ICAPS. 2005: 241-250

(上接第 195 页)

软件的事件确定有限自动机模型的生成方法。该方法通过UML顺序图来描述系统对象之间的交互行为，选取一组相关的全局变量构成状态向量对顺序图中各场景的交互信息的前后置状态向量值进行分析，检测场景中存在的矛盾，得到一致的需求场景；然后从顺序图中提取每个对象的事件序列集合，构造各个对象的时间确定有限自动机模型，最后组合各对象的自动机模型得到系统的有限自动机模型。该方法可以根据系统顺序图建立具有严格数学基础的系统确定有限自动机模型。生成的系统有限自动机模型可作为系统测试用例生成的基础模型，也可用于对系统行为进行诊断分析，这是以后工作的重点。

参 考 文 献

[1] 张曙光. 高速铁路系统生命周期安全评估体系的研究[J]. 铁道学报，2007，29(2)：20-26

[2] The European Committee for Electro-technical Standardization. EN 50126 Railway Applications-the Specification and Demonstration of Reliability, Availability, Maintainability and Safety (RAMS)[S]. UK: BSI, 2002

[3] CENELEC. EN 50128 Railway Applications: Communications, signaling and processing systems-Software for railway control and protection systems[S]. UK: CENELEC, 2001

[4] CENELEC. EN 50126 Railway Applications: The specification and demonstration of Reliability, Availability, Maintainability and Safety(RAMS) [S]. UK: CENELEC, 1999

[5] IEC. IEC61508 Functional Safety of electrical/ electronic/programmable electronic safety- related systems-part3: software requirements[S]. UK: IEC, 2000

[6] IEC. IEC61508 Functional Safety of electrical/ electronic/programmable electronic safety- related systems-part7: Overview of techniques and measures[S]. UK: IEC, 2006

[7] 王曦，徐中伟，梅萌. 基于模型检测的软件安全性验证方法[J]. 武汉大学学报，2010，56(2)：156-160

[8] Haxthausen A E, Peleska J. Formal Development and Verification of a Distributed Railway Control System[J]. IEEE Transactions on Software Engineering, 2000, 26(8): 1546-1563

[9] Garmhausen V H, Campos S, Cimatti A. Verification of a safety-critical railway interlocking system with real-time constraints [J]. Elsevier Science of Computer Programming, 2000(36): 53-64

[10] Yang Shuang-hua, Yang Li-li. Automatic safety analysis of control systems[J]. Journal of Loss Prevention in the Process Industries, 2005, 18(3): 178-185

[11] Bozzano M, Villa-orita A. The FSAP/NuSMV-SA Safety Analysis Platform[J]. Software Tools for Technology Transfer, 2007, 9(1): 5-24

[12] Koh K Y, Seong P H. SMV model-based safety analysis of software requirements[J]. Reliability Engineering & System Safety, 2009, 94(2): 320-331

[13] 赵志熙. 计算机联锁系统技术[M]. 北京：中国铁道出版社，1999：20

[14] Booch G, Rumbaugh J, Jacobsn I. UML 用户指南[M]. 邵维忠，译. 北京：机械工业出版社，2001：59-78

[15] Kim H, Russell M. Learning UML 2.0[M]. California: O' Reilly, 2006: 1-286

[16] 张琛，段振华. 应用 UML2.0 模型的测试用例生成方法[J]. 西安交通大学学报，2011，45(8)：18-23