软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software,2015,26(10):2521−2544 [doi: 10.13328/j.cnki.jos.004738]

E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel: +86-10-62562563

基于关键迹和 ASP 的 CSP 模型检测*

赵岭忠[1,2], 翟仲毅[2], 钱俊彦[2], 郭云川[3]

[1](软件工程国家重点实验室(武汉大学),湖北 武汉 430072)

[2](广西可信软件重点实验室(桂林电子科技大学),广西 桂林 541004)

[3](中国科学院 信息工程研究所,北京 100093)

通讯作者: 钱俊彦, E-mail: qjy2000@gmail.com

摘 要: 模型检测是通信顺序进程(communicating sequential processes,简称 CSP)形式化验证的重要手段.当前,CSP 模型检测方法基于操作语义,需将进程转化为迁移系统,进而提取语义模型,但转化过程较为复杂;待验证性质采用 CSP 语言进行描述,虽然有利于精炼检测(refinement checking),但描述能力较弱,通用性不强.鉴于此,提出了一种新的CSP指称语义模型——关键迹模型(critical-trace model)及基于该指称语义模型的CSP模型检测方法,并证明了其验证的可靠性,避免了上述问题.关键迹模型采用递归策略计算,待验证性质采用线性时态逻辑(linear temporal logic,简称 LTL)描述.基于回答集程序设计(answer set programming,简称 ASP)实现了关键迹模型的自动生成及 LTL 的自动验证,并开发了一个 CSP 模型检测原型系统——T_ASP.实验结果表明:与类似系统相比,该系统的描述能力更强,验证结果的准确性更高,且可同时验证多条性质,在性质不满足时还可提供多条反例.

关键词: 模型检测;通信顺序进程;关键迹模型;线性时态逻辑;回答集程序设计

中图法分类号: TP311

中文引用格式: 赵岭忠,翟仲毅,钱俊彦,郭云川.基于关键迹和 ASP 的 CSP 模型检测.软件学报,2015,26(10):2521−2544. http://www.jos.org.cn/1000-9825/4738.htm

英文引用格式: Zhao LZ, Zhai ZY, Qian JY, Guo YC. Model checking CSP based on ASP and critical-trace model of CSP. Ruan Jian Xue Bao/Journal of Software, 2015,26(10):2521−2544 (in Chinese). http://www.jos.org.cn/1000-9825/4738.htm

Model Checking CSP Based on ASP and Critical-Trace Model of CSP

ZHAO Ling-Zhong[1,2], ZHAI Zhong-Yi[2], QIAN Jun-Yan[2], GUO Yun-Chuan[3]

[1](State Key Laboratory of Software Engineering (Wuhan University), Wuhan 430072, China)

[2](Guangxi Key Laboratory of Trusted Software (Guilin University of Electronic Technology), Guilin 541004, China)

[3](Institute of Information Engineering, The Chinese Academy of Sciences, Beijing 100093, China)

Abstract: Model checking is a mainstream method for formal verification of communicating sequential processes (CSP). Existing CSP model checkers are based on operational semantics, which need to translate processes into a label transition system, and to extract the semantic model based on the system. The conversion process is complex. Moreover, in most CSP model checkers, the properties to be verified are described by CSP, which is helpful for refinement checking, but at the same time leads to limited description power and weak generality. To address these issues, a new denotational semantic model of CSP, critical-trace model, is proposed and proved to be reliable for model checking CSP. Based on this model, a framework for model checking CSP is established to allow critical-trace model to be constructed inductively from the traces of its components, and properties to be specified by linear temporal logic (LTL), a universal property specification language. In addition, automatic mechanisms for generating critical-trace model and verifying LTL formulas are

* 基金项目: 国家自然科学基金(61262008, 61100186); 广西自然科学基金(2013GXNSFBA019267); 武汉大学软件工程国家重点实验室开放基金(SKLSE20100806); 广西教育厅重点项目; 广西高等学校高水平创新团队及卓越学者计划; 广西可信软件重点实验室基金(kx201113,kx201415); 桂林电子科技大学创新团队资助项目

收稿时间: 2013-09-10; 定稿时间: 2014-09-29

implemented with answer set programming (ASP). Finally, the two mechanisms are integrated into a CSP model checker T_ASP. Compared with the similar systems developed previously, experimental results indicate a higher ability of the proposed system to describe CSP processes and higher verification accuracy. Furthermore, T_ASP checks multiple properties in one execution of the system. When a property is not satisfied, the system also returns counterexamples for the property.

Key words: model checking; communicating sequential process; critical-trace model; linear temporal logic; answer set programming

CSP 是研究并发的重要理论及构建并发系统的经典方法[1].随着 CSP 在分布式系统、网络安全协议方面的应用,对 CSP 的形式化验证显得尤为重要.目前,CSP 验证主要采用两种方法:程序正确性证明和模型检测.其中,程序正确性证明[2−4]能够解决并发系统无穷序列验证问题,但通常采用手工证明或者半自动化证明方式,且需领域专家参与.模型检测是一种自动的、基于模型、面向性质、用于有限状态系统的自动化验证技术,通过检测系统模型的所有可能状态,验证模型是否符合性质要求.当模型满足相应的性质时,输出满足;反之,则输出不满足,并提供相关反例,用于模型的精炼[5].一般情况下,模型检测的对象是有限状态系统,但是基于不动点理论[6]、Craig interpolation 技术[7]以及归纳原理[8],可以对某些无穷状态系统进行验证.

传统的模型检测一般使用 3 种层次的系统模型:高层语言模型、语义模型和实现模型[9].其中,

- 高层语言模型是一种高度抽象的形式化描述语言,可作为模型检测工具前端的描述语言,如 Promela[10],CSS[11],CSP 等;
- 语义模型是一种较为直观的形式化规范方式,可表示高层语言模型的含义,也可作为模型检测算法的输入模型,还可作为高层语言模型与实现模型间的中介模型,常见的语义模型,如 Petri 网模型[12]、CSP 的迹模型[13]等;
- 实现模型用于模型验证算法的具体实施,分为显式和符号模型两种,其中,显式模型直接采用语义模型进行验证;符号模型是语义模型的一种隐式表示形式,具有较高的空间表示效率,如基于二叉决策图(BDD)、基于约束和基于表的模型检测,即是通过相应的符号模型进行验证的[14−16].

语言模型可通过操作语义或指称语义生成相应的实现模型.CSP 发展过程中,Hoare 提出了迹模型的概念,并指出了它的两方面用途:(1) 可为 CSP 语言提供清晰的、一致的定义;(2) 可通过迹模型完成性质验证[13].现有 CSP 验证工具主要通过操作语义生成迹语义模型,如:FDR[17,18]通过操作语义将进程转化为相应的迁移系统,并依据迁移系统显式导出迹模型;ARC[19]和 PAT[20]则是通过操作语义得到进程的 BDD 符号模型,然后依据符号模型导出相应的迹模型,并完成验证.基于操作语义的模型转化方法通常比较复杂,不易修改和扩展,且须领域专家的参与.与以上工作不同,本文提出了一种基于指称语义——关键迹模型的 CSP 验证方法,以关键迹模型为语义模型和实现模型,底层实现则采用 ASP 技术,实现策略简单、易于理解、可扩展性好.

通用模型检测采用时态逻辑(temporal logic)来描述待验证性质[21−23],然而在现有多数 CSP 模型检测工具中,待验证性质和模型都采用进程形式进行描述,性质使用进程描述虽有助于模型精炼,但通用性不强,且对活性不能很好地描述.本文采用线性时态逻辑 LTL 来描述性质,可方便地对待验证性质进行描述,并通过可满足性求解技术对 CSP 模型进行验证.

模型检测技术面向应用的主要瓶颈是状态爆炸问题,目前存在多种解决途径,如偏序规约[24,25]、有界模型检测[26]、模块化方法[27].为缓解状态爆炸问题,CSP 检测工具也采用了相应策略,如:SymFDR[28]采用有界模型检测方法来缩减验证空间,并利用归纳的方法完成无界模型的验证;ARC 通过 BDD 符号模型来降低存储空间;PAT 则通过 BDD 符号模型和有界模型相结合的方法来解决状态爆炸问题.本文提出的关键迹模型是面向迹语义的抽象模型,与迹模型相比,其包含的数据量较小,一定程度上缓解了状态爆炸问题.

我们已对基于 ASP 的 CSP 模型检测技术进行了初步研究,如:文献[29]提出了进程和性质的知识表示方法以及把性质验证归约为回答集求解的思想,但是,该验证体系对语义模型的解释不够明确,缺乏系统的验证方法,自动化程度不高;文献[30]提出了基于进程迹的 CSP 模型验证框架,该方法明确了待验证的语义模型(迹模型),但是该模型规模庞大,包含大量冗余信息,增加了验证复杂度.鉴于上述问题,提出了 CSP 进程的关键迹模型,该模型只关注进程所发生的迹,减小了待验证语义模型规模,并利用 ASP 技术实现了关键迹模型自动化生成及

基于关键迹模型的自动化验证,实现了 CSP 模型检测原型系统——T_ASP.

本文第 1 节介绍 ASP 和 CSP.第 2 节给出关键迹模型及其递归计算方法,并论证基于关键迹模型验证的可靠性.第 3 节讨论如何自动生成关键迹模型,并提出基于迹的并发机制.第 4 节详述基于关键迹模型的 LTL 自动验证方法,介绍原型系统 T_ASP 的整体结构及其组件功能.第 5 节探讨哲学家就餐问题的 CSP 模型,并使用 T_ASP 验证模型性质.第 6 节分析相关研究工作.最后总结本文工作,指出后续的研究方向和策略.

1 基础知识

1.1 ASP编程

ASP 是一种声明式程序设计(declarative programming)方法[31],由稳定模型语义(stable model semantics)[32]扩展而来.ASP 作为新型程序设计方法有其独特的优点:(1) 编码紧凑、易理解;(2) 能够有效地对信息不完全问题进行刻画和计算;(3) 适合解决知识密集型、复杂计算问题;(4) 存在高效 ASP 求解器,可对问题自动求解[33].经多次扩展,ASP 具有强大的知识表示能力.下面介绍扩展析取逻辑程序的语法以及相应的 ASP 语义.

若 A 是一个原子,则 A 或~A 称为文字,其中,A 为正文字,~A 为负文字,A 和~A 称为一对互补字.一个扩展析取逻辑程序Π是一个规则集,且每条规则 r 满足如下形式:

$$L_1\vee\ldots\vee L_k\leftarrow L_{k+1},\ldots,L_m,\text{not } L_{m+1},\ldots,\text{not } L_n,$$

其中,$n\geqslant m\geqslant k\geqslant 0$,$L_i$ 是一个文字,not 表示失败即否定(negation as failure).$head(r)=\{L_1,\ldots,L_k\}$表示 r 头部的文字集合,$pos(r)=\{L_1,\ldots,L_m\}$表示 r 体部的正文字集合,$neg(r)=\{L_{m+1},\ldots,L_n\}$表示 r 体部的负文字集合.当头部为空时,称此 r 为约束规则.

Generate-and-Test[34,35]是 ASP 程序设计的经典方法,包括 generate,define 和 test 这 3 个模块.其中,generate 模块由一组普通规则构成,可计算出问题的潜在解;test 模块由一组约束规则构成,主要消除 generate 模块生成的无效解.对于复杂程序,generate 和 test 模块并不能完全消除不合理的解,test 可结合 define 模块解决这一问题.define 模块通常由一组头原子为辅助谓词的规则组成.

本文 ASP 程序的书写规范采用 DLV[36]系统的相关规定(用:-代替←;每条规则以英文句号结束).例如,程序 $a\leftarrow b$, not c 可表示为 a:-b, not c.

1.2 CSP基础理论

CSP 理论中,进程是描述客体的基本单位,进程间通过事件进行交互.设 x 是一个事件,P 是一个进程,αP 表示进程 P 的字母表,若 $x\in\alpha P$,则 $x\rightarrow P$ 表示一个进程,该进程首先执行事件 x,然后按照进程 P 的行为进行.进程的基本结构包括:前缀、递归、一般选择、非确定选择.

- 前缀:对于进程 $x\rightarrow P$,若 P 是可终止的或 $P=STOP$($STOP$ 表示空进程),则 $x\rightarrow P$ 属于前缀进程;
- 递归:X 是不可终止进程,$F(X)$表示进程 X 的卫式,即,$F(X)$满足($a\rightarrow X$)形式,则 $X=F(X)$属于递归进程,形式表示为$\mu X.F(X)$.此类进程可表示无穷行为的客体;
- 一般选择:P.Q 是两个进程,则 $P\Box Q$ 属于一般选择进程.$P\Box Q$ 可选 P 行为,也可选 Q 行为,具体选择由环境控制.假设 P:$x\rightarrow T$,Q:$y\rightarrow S$,若环境提供选择事件 x,则选择 P;若环境提供选择事件 y,则选择 Q.$\alpha(P\Box Q)$,满足:$\alpha(P\Box Q)=\alpha P=\alpha Q$;
- 非确定选择:P,Q 是两个进程,则 $P\Pi Q$ 属于非确定进程.$P\Pi Q$ 可选择 P 的行为,也可选择 Q 的行为,它的选择是任意的,不受外界控制.$\alpha(P\Pi Q)$满足:$\alpha(P\Pi Q)=\alpha P=\alpha Q$.

进程的并发方式是构造并发系统的核心.

假设有事件 a,b,c,d,进程 P,Q,且 $a\in(\alpha P-\alpha Q)$,$b\in(\alpha Q-\alpha P)$,$c,d\in(\alpha Q\cap\alpha P)$,可以得到并发规则 CR_{csp}.

- CR_{csp}-1: $(c\rightarrow P)\|(c\rightarrow Q)=(c\rightarrow(P\|Q))$;
- CR_{csp}-2: $(c\rightarrow P)\|(d\rightarrow Q)=STOP$,若 $c\neq d$;
- CR_{csp}-3: $(a\rightarrow P)\|(c\rightarrow Q)=(a\rightarrow(P\|(c\rightarrow Q))$;

- CR_{csp}-4: $(c\rightarrow P)||(b\rightarrow Q)=(b\rightarrow((c\rightarrow P)||Q))$;
- CR_{csp}-5: $(a\rightarrow P)||(b\rightarrow Q)=(a\rightarrow(P||(b\rightarrow Q))|b\rightarrow((a\rightarrow P)||Q))$.

定义 1. 一个 CSP 进程 P 可递归地定义为

$$P=STOP|x\rightarrow P_1|\mu P_2.F(P_2)|P_3\square P_4|P_3\Pi P_4|P_3||P_4,$$

其中,进程 P_1 是可终止进程.

CSP 存在多种指称语义模型,如迹模型(trace model)、失败/发散集模型(failure-divergence model),它们分别提供了进程的不同行为信息.本文主要讨论 CSP 的迹模型,该模型包含一系列迹,其中每条迹表示进程的一条行为序列,该序列由进程可执行的事件组成,并以事件发生的先后顺序进行排列.该模型显式地提供了进程的所有可能行为序列.为了自动验证的需要,本文只在有穷迹语义下对进程进行解释.

定义 2(有穷迹语义)[37]. 给定进程 P,P 的有穷迹语义是从语言模型 $CSP_M(P)$到 P 的迹模型的一个函数.

定义 3(迹模型)[37]. 给定进程 P,P 的迹模型记为 $traces(P)$,$traces(P)$包含进程 P 所有可能的迹.

$traces(P)$的构建有两种途径:一是以递归方式由进程 P 所含组件进程的迹来计算 $traces(P)$;二是通过 CSP 的操作语义将进程转化为相应的迁移系统,进而提取进程的 $traces(P)$.文献[37]已经证明这两种方法的等价性.本文采用递归求解方式来构造迹模型,下面给出基本结构进程的迹模型计算方式,复杂的 CSP 进程可递归地调用基本结构模型计算规则进行求解.

- 若进程 P 为 $STOP$ 进程,则 P 的迹模型为
$$traces(STOP)=\{t|t=\langle\rangle\};$$
- 若进程 P 为 $P=a\rightarrow STOP$,则 P 的迹模型为
$$traces(P)=\{\langle\rangle\}\cup\{\langle a\rangle\};$$
- 若进程 P 为 $P=a\rightarrow Q$(其中,进程 $Q=a\rightarrow\ldots\rightarrow STOP$ 的形式),则 P 的迹模型为
$$traces(P)=\{\langle\rangle\}\cup\{\langle a\rangle\hat{}\,t|t\in traces(Q)\}(\hat{}\text{是迹之间的连接算子});$$
- 若进程 P 为 $P=a\rightarrow\ldots\rightarrow b\rightarrow P$,设进程 $Q=a\rightarrow\ldots\rightarrow b\rightarrow STOP$,进程 Q 的最长迹为 q,q 的所有子迹模型为 sub_T,则 P 的迹模型为
$$traces(P)=\{\langle\rangle\}\cup\{(q\hat{}\langle b,a\rangle)^n\}\cup\{t|t=r\hat{}\,s,r\in(q\hat{}\langle b,a\rangle)^n,s\in sub\_T,n=0,1,2,\ldots\};$$
- 若进程 P 为 $Q\square T$,Q 和 T 为前缀进程或递归进程,则进程 P 的迹模型为
$$traces(P)=\{q|q\in traces(Q)\}\cup\{t|t\in traces(Q)\};$$
- 若进程 P 为 $Q\Pi T$,Q 和 T 为前缀进程或递归进程,则进程 P 的迹模型为
$$traces(P)=\{q|q\in traces(Q)\}\cup\{t|t\in traces(Q)\};$$
- 若进程 P 为 $Q||T$,Q 和 T 为前缀进程或递归进程,则进程 P 的迹模型为
$$traces(P||Q)=\{t|(t\uparrow\alpha P)\in traces(P)\wedge(t\uparrow\alpha Q)\in traces(Q)\wedge t\in(\alpha P\cup\alpha Q)^*\}.$$

其中,$t\uparrow\alpha P$ 表示 t 中属于 αP 的事件组成的新迹,$(\alpha P\cup\alpha Q)^*$表示字母 $\alpha P\cup\alpha Q$ 任意排列组成的迹集合.

2 关键迹模型

文献[38]给出了评价语义模型的两个标准:一是语义模型能够很好地区分不同的程序;二是语义模型要尽可能地精简,摒弃无关信息.从模型检测角度看,语义模型的规模是主要的评价因素.迹模型为 CSP 进程提供了大量的可能行为,通常规模庞大,包含大量不影响性质验证结果的冗余信息.所以在 CSP 进程验证时,只关注有影响的关键迹是提高验证效率的有效途径.为此,下文首先构建一种新的模型——关键迹模型,该模型是迹模型的一个特殊子集,数据量远小于迹模型;然后论证基于关键迹模型验证的可靠性.

2.1 关键迹模型的定义

定义 4(子迹). 设 s,t 分别为两条迹,若存在任意一条迹 r,满足 $s\hat{}\,r=t$($\hat{}$是迹之间的连接算子),则称 s 是进程 t 的子迹,形式化地记为 $s\in prefix(t)$.

定义 5(最长迹(max-trace)). s,t 是进程 P 的任意两条迹,即 $s\in traces(P),t\in traces(P)$,若对于 $t\in traces(P)$,不存在 $s\in prefix(t)$,则 s 属于进程 P 的一条最长迹,形式化地记为 Max-$trace_P(s)$.

定义 6(关键迹模型(critical-trace model)). 进程 P 的关键迹模型定义如下:

$$critical\text{-}traces(P)=\{s|s\in traces(P)\wedge \text{Max-}trace_P(s)\}.$$

可见,进程的关键迹模型由迹模型中所有最长迹构成.下面给出基本进程关键迹模型的计算规则.

- 若进程 P 为 $STOP$,则 P 的关键迹模型为

$$critical\text{-}traces(STOP)=\{t|t=\langle\rangle\};$$

- 若进程 P 为 $P=a\rightarrow STOP$,则 P 的关键迹模型为

$$critical\text{-}traces(P)=\{\langle a\rangle\hat{}\langle\rangle\}=\{\langle a\rangle\};$$

- 若进程 P 为 $P=a\rightarrow Q$,且进程 $Q=b\rightarrow\ldots\rightarrow STOP$,则 P 的关键迹模型为

$$critical\text{-}traces(P)=\{\langle a\rangle\hat{}t|t\in critical\text{-}traces(Q)\};$$

- 若进程 P 为 $P=a\rightarrow\ldots\rightarrow b\rightarrow P$,设进程 $Q=a\rightarrow\ldots\rightarrow b\rightarrow STOP$,则进程 P 的关键迹模型为

$$critical\text{-}traces(P)=\{s|s=(t\hat{}\langle b,a\rangle)^n\wedge t\in traces(Q),n=0,1,2,\ldots\};$$

- 若进程 P 为 $Q\square T$,Q 和 T 为前缀进程或递归进程,则 P 的关键迹模型为

$$critical\text{-}traces(P)=\{q|q\in critical\text{-}traces(Q)\}\cup\{t|t\in critical\text{-}trace(T)\};$$

- 若进程 P 为 $Q\Pi T$,Q 和 T 为前缀进程或递归进程,则 P 的关键迹模型为

$$critical\text{-}traces(P)=\{q|q\in critical\text{-}traces(Q)\}\cup\{t|t\in critical\text{-}trace(T)\};$$

- 若进程 P 为 $Q\|T$,Q 和 T 为前缀进程或递归进程,则 P 的关键迹模型为

$$critical\text{-}traces(P)=\{t|(t\uparrow\alpha Q)\in traces(Q)\wedge(t\uparrow\alpha T)\in traces(T)\wedge t\in(\alpha Q\cup\alpha T)^*\wedge\text{Max-}trace_P(t)\wedge t\in CR_{csp}(Q\|T)\}.$$

其中,函数 $CR_{csp}(P)$表示进程 P 符合 CR_{csp} 的迹的集合(当 P 不是并发进程时,$CR_{csp}(P)$表示 P 的最长迹的集合).可见,$CR_{csp}(Q\|T)$表示进程 Q 和 T 并发时,符合 CR_{csp} 的迹的集合.

2.2 基于关键迹模型验证的可靠性

定义 7(前缀封闭)[1]. 给定进程 P,s,t 为两条迹,若 $s\hat{}t$ 是 P 的一条迹,则 s 也为 P 的一条迹.

性质 1[1]. 给定进程 P,$traces(P)$满足如下性质:(1) $traces(P)$是非空集合;(2) $traces(P)$是前缀封闭的.

定理 1. 给定进程 P,对于任意 $t\in traces(P)$,必存在唯一的迹 s 满足:$t\in prefix(s)\wedge$Max-$trace_P(s)$.

证明:由定义 6 可得,$critical\text{-}traces(P)\subset traces(P)$.

可知:对于任意的 $t\in traces(P)$,$t\in critical\text{-}traces(P)$或 $t\in traces(P)-critical\text{-}traces(P)$.

(1) 若 $t\in critical\text{-}traces(P)$,即,$t$ 满足 Max-$trace_P(t)$.由性质 1 可得:存在迹 $s\in traces(P)$,且 $s=t\hat{}r$.由定义 5 可知,$traces(P)$中的任何最长迹是唯一的,所以存在 $r=\langle\rangle$,$s=t$;

(2) 若 $t\in traces(P)-critical\text{-}traces(P)$,即,$t\in trace(P)$,$t\notin critical\text{-}traces(P)$.由性质 1 可得:存在迹 $s\in traces(P)$,$s=t\hat{}r$.由 t 不满足 Max-$trace_P(t)$,则一定存在迹 $r\neq\langle\rangle$满足$(s=t\hat{}r)\wedge$Max-$trace_P(s)$.$traces(P)$中的最长迹是唯一的,所以 r 也是唯一的. □

定理 2. 给定进程 P,$t\in traces(P)$且 t 是 P 所发生的迹,则 t 满足:

(1) t 可由最长迹复合或并发生成;

(2) Max-$trace_P(t)$.

证明:

1) 如果 P 不是并发进程,即 P 无需与其他进程交互.

(1) 若 P 为基本进程,则 P 将按照最长的行为序列一直进行,因此,P 所发生的迹 t 一定满足:

Max-$trace_P(t)$;

(2) 若 P 为由基本进程组成的复合进程,则 P 所发生的迹可由基本进程经过复合操作生成.基本进程所发生的迹是最长迹,且最长迹经过所有复合操作仍生成最长迹,故,P 所有发生的迹也为最长迹;

2) 如果 P 为并发进程,设 $P=Q||R$.

(1) 若 Q,R 为基本进程,假设 t 由 r 和 s 并发生成,即 $t=CR_{csp}(r||s),r\in traces(Q),s\in traces(R)$.由定理 1 可知:存在 $p\in traces(Q)\wedge r\in prefix(p)\wedge$Max-$trace_Q(p),q\in traces(R)\wedge s\in prefix(q)\wedge$Max-$trace_R(q)$.综上可得 $t=CR_{csp}(p||q)$.假设 $t=CR_{csp}(p||q)$不满足 $Max\text{-}trace_P(t)$,即:存在迹 $w\in traces(P),t\in prefix(w)$,且满足 Max-$trace_P(w)$.如果 $w=CR_{csp}(u||v)$,则满足 $r\in prefix(u)\wedge u\in traces(Q),s\in prefix(v)\wedge v\in traces(R)$.由定理 1 可知:存在 $p\in traces(Q)\wedge u\in prefix(p)\wedge$Max-$trace_Q(p),q\in traces(R)\wedge v\in prefix(q)\wedge$Max-$trace_R(q)$满足 $w=CR_{csp}(p||q)$.由 Max-$trace_P(w)$可知,$CR_{csp}(p||q)$生成的迹为最长迹.与已知矛盾,假设不成立.综上可得,$t=CR_{csp}(p||q)$满足 Max-$trace_P(t)$;

(2) 若进程 Q 或 R 是并发进程,则 Q,R 可以通过递归调用归结为基本进程的并发.通过情形(1)中类似方法容易证得 $t=CR_{csp}(p||q)$满足 Max-$trace_P(t)$. □

通过定理 2 可知:进程所发生的迹一定是最长迹,且这些迹一定可由相应子进程的最长迹生成.

基于迹模型进行性质判定时,需首先计算进程的迹模型,然后逐条判断每条迹是否符合 CR_{csp} 及是否满足相应性质要求.待验证性质采用 LTL 公式描述,其语法结构如下:

$$\psi::=p\in A|\neg\psi|\psi_1\wedge\psi_2|\psi_1\vee\psi_2|G\psi|F\psi|X\psi.$$

简要起见,这里只采用 G,F,X 这 3 个时态算子,其中,G 算子表示未来的所有状态,F 算子表示未来的某一状态,X 算子表示下一个状态.基于迹模型的 LTL 公式可满足如下语义定义:

定义 8(基于迹模型的 LTL 公式可满足语义($traces(P)\models S_{LTL}$)). 给定进程 P 和任意 LTL 公式 $S_{LTL},\pi_t\in trace(P)$ 且 $\pi_t\in CR_{csp}(P),\pi_t$ 满足 S_{LTL}(记做 $\pi_t|=S_{LTL}$)定义如下:

$\pi_t\models p$	iff	$\exists i.p=\pi_{t(i)}$
$\pi_t\models\neg p$	iff	$\pi_t\not\models p$
$\pi_t\models\psi_1\wedge\psi_2$	iff	$\pi_t\models\psi_1$ 且 $\pi_t\models\psi_2$
$\pi_t\models\psi_1\vee\psi_2$	iff	$\pi_t\models\psi_1$ 或 $\pi_t\models\psi_2$
$\pi_t\models X\psi$	iff	$\pi_t(2)\models\psi$
$\pi_t\models F\psi$	iff	$\exists i.\pi_t(i)\models\psi$
$\pi_t\models G\psi$	iff	$\pi_t\models\neg F\neg\psi$

其中,$\pi_{t(i)}$表示进程迹 π_t 的第 i 个事件,$\pi_t(i)=\pi_{t(i)}\rightarrow\pi_{t(i+1)}\rightarrow\ldots$是从 π_t 中删除前$(i-1)$个事件后所得到的子迹.如果对任意 $\pi_t\in trace(P)$且 $\pi_t\in CR_{csp}(P)$均满足 $\pi_t\models S_{LTL}$,则称 P 的迹模型满足 S_{LTL},记做 $traces(P)\models S_{LTL}$.

性质 2. 给定进程 $P,\{t|t\in traces(P)\wedge$Max-$trace_P(t)\}=critical\text{-}traces(P)$.

证明:由定义 6 可得上述性质. □

定理 3. 给定进程 P、LTL 公式 $S_{LTL},traces(P)\models S_{LTL}$ 当且仅当 $critical\text{-}traces(P)\models S_{LTL}$.

证明:

(1) 充分性.

由性质 2 可得,$critical\text{-}traces(P)=\{t|\forall t\in traces(P)\wedge$Max-$trace_P(t)\}$.由此可得:

$$critical\text{-}traces(P)\models S_{LTL}\Leftrightarrow\{t|=S_{LTL}|\forall t\in trace(P)\wedge\text{Max-}trace_P(t)\}.$$

由定理 2 可知,进程发生的迹 t 一定属于最长迹,即,$t\in CR_{csp}(P)$满足 Max-$trace_P(t)$.整理后得到:

$$\{t\models S_{LTL}|\forall t\in trace(P)\wedge t\in CR_{csp}(P)\}.$$

通过定义 8 可得,$traces(P)\models S_{LTL}$.综上可得,$critical\text{-}traces(P)\models S_{LTL}\Rightarrow traces(P)\models S_{LTL}$.

(2) 必要性.

由定义 8 可知,$traces(P)\models S_{LTL}$,即$\{t\models S_{LTL}|\forall t\in trace(P)\wedge t\in CR_{csp}(P)\}$.通过定理 2 可得,$traces(P)\models S_{LTL}$,即:

$$\{t\models S_{LTL}|\forall t\in trace(P)\wedge\text{Max-}trace_P(t)\}.$$

由性质 2 可得,$\{t\models S_{LTL}|\forall t\in trace(P)\wedge$Max-$trace_P(t)\}\Rightarrow critical\text{-}traces(P)\models S_{LTL}$.

综上可得:$traces(P)\models S_{LTL}\Rightarrow critical\text{-}traces(P)\models S_{LTL}$. □

由定理 3 可知:对于 LTL 公式的验证而言,进程的关键迹模型和迹模型是等价的.

3 关键迹模型的自动生成方法

3.1 关键迹模型的生成框架

利用 ASP 技术实现关键迹模型的自动生成包括以下过程:(1) 知识的翻译,即,把 CSP 进程转化为对应的 ASP 规则;(2) 生成基本进程的关键迹模型,即,利用 ASP 构造基本进程到关键迹模型的生成规则;(3) 并发操作下关键迹模型的生成,即,利用 ASP 构造基于迹的并发规则,通过此规则生成并发系统的关键迹模型.本节主要讨论过程(2)和过程(3),分别对应 ASP 平台下关键迹模型生成机制的两个模块:基本进程和并发操作模块.基本进程模块包括前缀进程、递归进程、一般选择进程和非确定选择进程的生成规则.并发操作模块包括:(1) 选择模块,为基本进程模块中的选择进程提供相应的选择知识,从而实现自动生成关键迹模型;(2) 迹的并发规则库,该规则库与 CR_{csp} 有等价作用,已知两进程的关键迹模型,调用此规则库可生成其并发进程的关键迹模型.

为了更直观地描述上述设计,图 1 给出了关键迹模型的自动生成机制,其中,空心箭头表示数据的输入或输出,实线箭头表示数据流向,虚线箭头表示数据访问.下文框图约定与此相同,这里不再赘述.

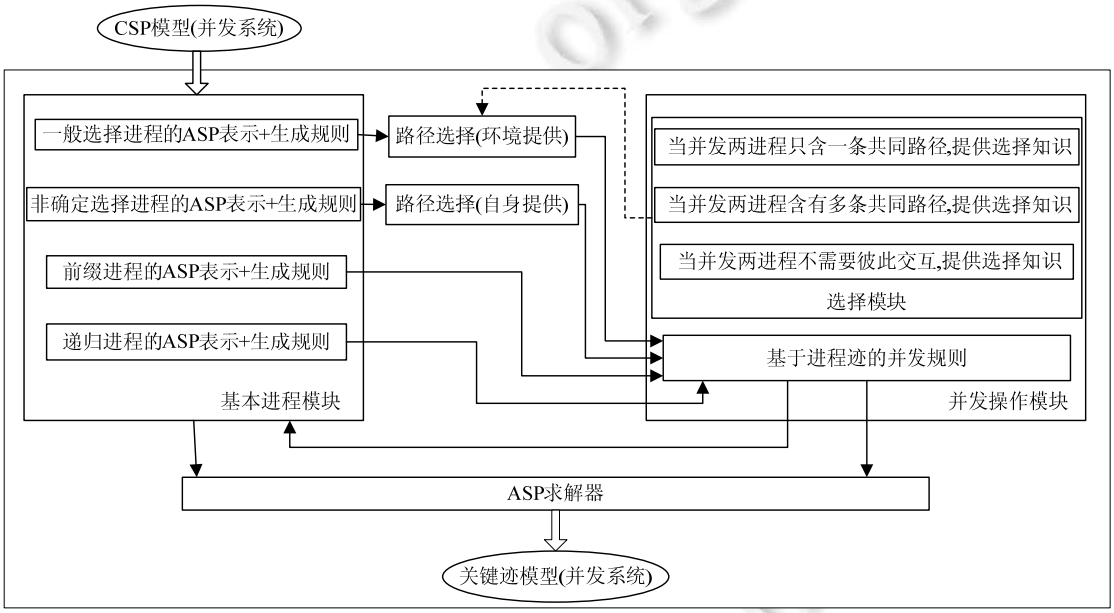


Fig.1 Automatic generation mechanism of critical-trace model

图 1 关键迹模型的自动生成机制

3.2 基本进程关键迹模型的生成

3.2.1 进程和迹的 ASP 表示

为了刻画进程和迹,引入谓词 *flow*(*X*,*Y*,*P*)表示在进程 *P* 的事件 *X* 比事件 *Y* 先发生,*preset*(*P*,*Q*)表示进程 *P* 是进程 *Q* 前缀式中所有事件组成的可终止进程(如 $Q=x\rightarrow\ldots\rightarrow z\rightarrow S$,则 $P=x\rightarrow\ldots\rightarrow z\rightarrow STOP$),*invoking*(*P*,*Q*)表示进程 *P* 的执行过程中将调用进程 *Q*,*repeat*(*P*,*Q*)表示进程 *Q* 的行为由进程 *P* 的自调用形成.

例如:

- 进程 *Q* 的一条迹为⟨*u*,*v*,*w*,*x*,*y*,*z*⟩,可用 ASP 表示为
  - *flow*(*u*,*v*,*q*);
  - *flow*(*v*,*w*,*q*);

- *flow*(*w*,*x*,*q*);
- *flow*(*x*,*y*,*q*);
- *flow*(*y*,*z*,*q*);

- 前缀进程 *P*=*w*→*x*→*y*→*z*→*STOP*,可用 ASP 表示为
  - *flow*(*w*,*x*,*q*);
  - *flow*(*x*,*y*,*q*);
  - *flow*(*y*,*z*,*q*);
  - *preset*(*q*,*p*);

  其中,*q* 表示前缀式构成的可终止进程,*p* 表示进程 *P*;
- 递归进程 *P*=*w*→*x*→*y*→*z*→*P*,可用 ASP 表示为
  - *flow*(*w*,*x*,*q*);
  - *flow*(*x*,*y*,*q*);
  - *flow*(*y*,*z*,*q*);
  - *preset*(*q*,*p*);
  - *repeat*(*p*,*q*).

一般选择进程的每条分支可采用前缀或递归进程的描述方法加以表示,此外,还需刻画不同分支形成的中间进程知识.引入谓词 *pre*(*X*,*Y*,*P*)表示在进程 *P* 中先执行事件 *X*,然后执行事件 *Y*,*isset*(*M*,*N*)表示进程 *M* 是 *N* 的子进程,且 *M* 由 *N* 的若干前缀事件序列构成.*aux_invoking*(*P*,*Q*)表示在进程 *P* 的执行过程中可能调用进程 *Q*.

一般选择进程 *P*=*u*→*v*→(*w*→*x*→*y*→*STOP*□*y*→*w*→*x*→*P*),可用 ASP 表示为

- *pre*(*u*,*v*,p_1);
- *pre*(*v*,*w*,p_2);
- *pre*(*w*,*x*,p_2);
- *pre*(*x*,*y*,p_2);
- *isset*(p_1,p_2);
- *preset*(p_2,*p*);
- *pre*(*v*,*y*,p_3);
- *pre*(*y*,*w*,p_3);
- *pre*(*w*,*x*,p_3);
- *isset*(p_1,p_3);
- *preset*(p_3,*p*);
- *aux_invoking*(p_3,*p*).

非确定选择进程的知识刻画与一般选择进程相同,这里不再赘述.

为了生成进程的关键迹模型,引入辅助谓词 *event*(*X*,*P*)表示事件 *X* 属于进程 *P*,*first*(*X*,*P*)表示 *P* 进程前缀表示式的第 1 个事件是 *X*,*not_first*(*X*,*P*)表示 *X* 不是 *P* 中第 1 个发生的事件,*not_last*(*X*,*P*)表示 *X* 不是 *P* 中最后发生的事件,*pre_last*(*Y*,*P*)表示事件 *Y* 是进程 *P* 的前缀式的最后一个事件.它可用 ASP 表示为

- *event*(*X*,*P*):-*flow*(*Y*,*X*,*P*);
- *event*(*Y*,*P*):-*flow*(*Y*,*X*,*P*);
- *first*(*X*,*P*):-not *not-first*(*X*,*P*),*event*(*X*,*P*);
- *not_first*(*X*,*P*):-*flow*(*Y*,*X*,*P*);
- *not_last*(*X*,*P*):-*flow*(*X*,*Y*,*P*);
- *pre_last*(*Y*,*P*):-not *not_last*(*Y*,*P*),*event*(*X*,*P*).

3.2.2 前缀进程的关键迹模型生成

前缀进程表示有限行为的客体,进程执行到前缀式的尾事件将正常终止.由于此类进程属于线性可终止结构,所以到关键迹模型的转化比较简单.引入 *last*(*X*,*P*)表示 *X* 是进程 *P* 的最后一个事件,即 *X* 是 *P* 正常结束的标志.转化过程可用 ASP 表示为

last(*Y*,*P*):-*flow*(*X*,*Y*,*P*),*pre_last*(*Y*,*P*),not *invoking*(*P*,*Q*),not *repeat*(*S*,*P*).

3.2.3 递归进程的关键迹模型生成

递归进程表示无穷行为的客体,可视为前缀表示式基础上形成的循环进程.递归进程执行完前缀表示式的最后一个事件,将继续调用自身或其他进程.

- 若调用自身,可表示为
  - *flow*(*X*,*Y*,*Q*):-*flow*(*X*,*Y*,*P*),*preset*(*P*,*Q*);
  - *flow*(*X*,*Y*,*Q*):-*pre_last*(*X*,*P*),*first*(*Y*,*P*),not *flow*(*X*,*Y*,*P*),*repeat*(*P*,*Q*);
- 若调用其他进程,可表示为
  - *flow*(*X*,*Y*,*Q*):-*flow*(*X*,*Y*,*P*),*invoking*(*Q*,*P*);
  - *flow*(*X*,*Y*,*Q*):-*invoking*(*Q*,*P*),*preset*(*R*,*P*),*preset*(*T*,*Q*),*pre_last*(*X*,*T*),*first*(*Y*,*R*),not *flow*(*X*,*Y*,*P*).

3.2.4 一般选择进程的关键迹模型生成

一般选择进程到关键迹模型的转化需经过两个步骤:选择进程的路径以及处理正常终止或递归调用.下面引入谓词 *choice*(*X*,*N*)表示选择 *N* 进程中的 *X* 事件,*choice_p*(*N*)表示进程 *N* 将被选择.下节中给出的选择机制可提供相应的选择知识——*choice*(*X*,*N*).一般选择进程的路径选择需经过以下 3 个步骤.

(1) 根据选择机制提供的选择知识,完成分支进程的选择,可表示为

flow(*X*,*Y*,*N*):-*pre*(*X*,*Y*,*N*),*choice*(*Y*,*N*);

(2) 已被选择的分支,其后续路径也将被选择,可表示为

flow(*Y*,*Z*,*N*):-*flow*(*X*,*Y*,*N*),*pre*(*Y*,*Z*,*N*);

(3) 当在选择结构中嵌套了选择结构时,需构造如下规则完成整个进程的选择:

- *choice_p*(*P*):-*choice*(*X*,*P*);
- *flow*(*Y*,*Z*,*N*):-*flow*(*Y*,*Z*,*M*),*isset*(*M*,*N*),*choice_p*(*N*).

执行上述 3 个步骤,进程将得到确定的路径.在路径的终端,进程或正常终止或递归调用.正常终止的处理方法与前缀进程相同.递归调用时,进程的处理方式与上文相同,但需先将递归调用的进程激活,可用 ASP 实现为

invoking(*P*,*Q*):-*pre_last*(*X*,*P*),*aux_invoking*(*P*,*Q*).

按照上述递归机制,进程可调用进程的任何分支,进程迹会产生多个环形路径,增加了性质验证复杂度.因此,建模时宜加强对模型的限定,如规定递归调用时,进程只调用已经选择的分支进程,该方式可表示为

repeat(*P*,*Q*):-*pre_last*(*X*,*P*),*aux_repeat*(*P*,*Q*).

3.2.5 非确定选择进程的关键迹模型生成

与一般选择进程不同,非确定选择进程的路径选择由进程内部完成,而内部选择方式可以是随机选择,也可人为制定.简化起见,采用不确定问题确定化的方法处理路径选择问题,其基本思想是:把非确定选择进程任意随机选择得到的模型作为确定性模型的子模型,合并所有子模型,并用确定性模型代替非确定模型.显而易见,非确定性模型的行为一定包含在确定性模型中,这保证了模型验证的可靠性.引入谓词 *un_choice*(*X*,*P*)表示进程 *P* 的事件 *X* 是非确定进程的一个随机选择方式,则相应的选择知识 *un_choice*(*X*,*P*) *v*…*v choice*(*Y*,*M*)可由进程自身提供,具体过程由第 4.4 节给出的翻译程序 1 中实现.

非确定选择进程到关键迹模型的转化与一般选择进程类似,也需经过上述两个步骤.当运用非确定选择来描述进程时,主要采用两种形式:包含非确定选择嵌套的方式和不包含非确定选择嵌套的方式.

- 如果进程描述采用不包含非确定选择嵌套的形式,则关键迹模型的生成过程可表示为
  - *flow*(*X*,*Y*,*N*):-*pre*(*X*,*Y*,*N*),*choice*(*X*,*N*);

- *flow*(*Y*,*Z*,*N*):-*flow*(*X*,*Y*,*N*),*pre*(*Y*,*Z*,*N*);
- *repeat*(*P*,*Q*):-*pre_last*(*X*,*P*),*aux_repeat*(*P*,*Q*).

- 如果进程描述采用包含非确定选择嵌套的方式,其生成过程可表示为
  - *choice*(*X*,*N*):-*undefined_choice*(*Y*,*M*),*undefined_choice*(*X*,*N*),*isset*(*M*,*N*);
  - *choice*(*Y*,*M*):-*undefined_choice*(*Y*,*M*),*undefined_choice*(*X*,*N*),*isset*(*M*,*N*);
  - *flow*(*X*,*Y*,*N*):-*pre*(*X*,*Y*,*N*),*choice*(*X*,*N*);
  - *flow*(*Y*,*Z*,*N*):-*flow*(*X*,*Y*,*N*),*pre*(*Y*,*Z*,*N*);
  - *choice_p*(*P*):-*choice*(*X*,*P*),*choice*(*Y*,*Q*),*isset*(*P*,*Q*);
  - *choice_p*(*Q*):-*choice*(*X*,*P*),*choice*(*Y*,*Q*),*isset*(*P*,*Q*);
  - *flow*(*Y*,*Z*,*N*):-*flow*(*Y*,*Z*,*M*),*isset*(*M*,*N*),*choice_p*(*N*);
  - *repeat*(*P*,*Q*):-*pre_last*(*X*,*P*),*aux_repeat*(*P*,*Q*).

3.3 并发操作下关键迹模型的生成

3.3.1 选择机制的 ASP 实现

为构建选择机制,引入辅助谓词 *aux_p*(*P*,*Q*),*aux_c*(*P*,*Q*),*process*(*Q*),*small*(*P*),*nsmall*(*P*),*auxn_first*(*X*,*P*),*auxfirst*(*X*,*P*),其中,*aux_p*(*P*,*Q*)表示进程 *P* 是进程 *Q* 的子进程,*aux_c*(*P*,*Q*)表示进程 *P* 与进程 *Q* 进行并发,*process*(*Q*)表示 *Q* 为一个进程,*small*(*P*)表示进程 *P* 为不含子进程的进程,*auxfirst*(*X*,*P*)表示 *X* 是子进程 *P* 的首事件.利用上述谓词构造如下辅助规则:

- *aux_p*(*P*,*Q*):-*preset*(*P*,*Q*);
- *aux_p*(*O*,*Q*):-*aux_p*(*P*,*Q*),*isset*(*O*,*P*);
- *event*(*X*,*P*):-*pre*(*X*,*Y*,*P*);
- *event*(*Y*,*P*):-*pre*(*X*,*Y*,*P*);
- *event*(*X*,*Q*):-*event*(*X*,*P*),*aux_p*(*P*,*Q*);
- *process*(*P*):-*isset*(*P*,*Q*);
- *process*(*Q*):-*isset*(*P*,*Q*);
- *auxn_first*(*X*,*P*):-*pre*(*Y*,*X*,*P*);
- *small*(*P*):-not *nsmall*(*P*),*process*(*P*);
- *auxfirst*(*X*,*P*):-not *auxn_first*(*X*,*P*),*event*(*X*,*P*);
- *nsmall*(*Q*):-*isset*(*P*,*Q*);
- *aux_choice*(*X*,*N*,*Y*,*M*):-*pre*(*T*,*X*,*N*),*pre*(*T*,*Y*,*M*),*auxfirst*(*T*,*N*),*auxfirst*(*T*,*M*),*s_layer*(*N*,*M*).

选择机制通过并发环境对路径进行选择,因而只能为一般选择进程提供相应的选择知识.设进程 *P* 是一般选择进程,*Q* 为任意进程,*P* 和 *Q* 并发时,*P* 的选择方式分为 3 种情况.

(1) *P* 有一条分支路径由 *P* 与 *Q* 的共有事件构成,并由此生成相应的选择知识,可用 ASP 表示为
- *choice*(*Y*,*M*):-*aux_choice*(*X*,*N*,*Y*,*M*),*small*(*N*),*small*(*M*),*aux_p*(*N*,*P*),*aux_p*(*M*,*P*),*aux_c*(*P*,*Q*),
  event(*Y*,*Q*),not *event*(*X*,*Q*),*X*!=*Y*,*N*!=*M*;
- *choice*(*Y*,*M*):-*choice*(*Z*,*R*),*aux_choice*(*X*,*N*,*Y*,*M*),*isset*(*R*,*N*),*isset*(*R*,*M*),*aux_p*(*N*,*P*),*aux_p*(*M*,*P*),
  aux_c(*P*,*Q*),*event*(*Y*,*Q*),not *event*(*X*,*Q*),*X*!=*Y*,*N*!=*M*;
- *choice*(*X*,*N*):-*choice*(*Z*,*R*),*aux_choice*(*X*,*N*,*Y*,*M*),*isset*(*R*,*N*),*isset*(*R*,*M*),*aux_p*(*N*,*P*),*aux_p*(*M*,*P*),
  aux_c(*P*,*Q*),not *event*(*Y*,*Q*),*event*(*X*,*Q*),*X*!=*Y*,*N*!=*M*;

(2) *P* 的所有分支都是私有事件,无需 *Q* 的参与.路径的选择由内部提供,可用 ASP 表示为
- *choice*(*X*,*N*) *v choice*(*Y*,*M*):-*aux_choice*(*X*,*N*,*Y*,*M*),*small*(*N*),*small*(*M*),*aux_p*(*N*,*P*),*aux_c*(*P*,*Q*),
  not *event*(*Y*,*Q*),not *event*(*X*,*Q*),*X*!=*Y*,*N*!=*M*;
- *choice*(*X*,*N*) *v choice*(*Y*,*M*):-*choice*(*Z*,*R*),*aux_choice*(*X*,*N*,*Y*,*M*),*isset*(*R*,*N*),*isset*(*R*,*M*),*aux_p*(*N*,*P*),

aux_p(*M*,*P*),*aux_c*(*P*,*Q*),not *event*(*Y*,*Q*),not *event*(*X*,*Q*),*X*!=*Y*,*N*!=*M*;

(3) *P* 的多条分支都由 *P* 与 *Q* 的共有事件构成,每个分支都需 *Q* 的参与.此时,环境提供了多条路径选择,也转化为非确定选择,可用 ASP 表示为

- *choice*(*Y*,*N*) *v choice*(*Z*,*M*):-*pre*(*X*,*Y*,*N*),*pre*(*X*,*Z*,*M*),*aux_p*(*N*,*P*),*aux_p*(*M*,*P*),*aux_c*(*P*,*Q*),*event*(*Y*,*Q*), *event*(*X*,*Q*),*N*!=*M*;
- *choice*(*X*,*N*) *v choice*(*Y*,*M*):-*choice*(*Z*,*R*),*aux_choice*(*X*,*N*,*Y*,*M*),*isset*(*R*,*N*),*isset*(*R*,*M*),*aux_p*(*N*,*Q*), *aux_p*(*M*,*Q*),*aux_c*(*P*,*Q*),*event*(*Y*,*P*),*event*(*X*,*P*),*X*!=*Y*,*N*!=*M*.

3.3.2 CSP 的并发机制与基于进程迹的并发机制

进程并发时,最重要的任务是调用 CR_{csp} 生成进程所发生的迹.CR_{csp} 提供了抽象的并发方式,可简洁地显示出进程的并发状况.但其不足之处是:只关注当前的并发事件,而后续的并发仍采用抽象方式描述,为了得到语义模型,需要频繁地调用 CR_{csp},在语言模型和语义模型间加以转换.为此,提出一种基于迹的并发机制.

为了构建基于迹的并发规则,需关注两方面问题:(1) CR_{csp} 的并发思想;(2) 迹并发的基本结构.其中,CR_{csp} 的总体思想是:共同事件共同参与,私有事件随机发生.迹并发的基本结构将在下节给出.在构建此种并发规则时,先确定迹并发的基本结构,然后确定该结构遵循的 CR_{csp} 的并发思想,进而设计相应的并发规则.

图 2 给出了 CR_{csp} 与基于迹的并发规则之间的关系.可见,基于迹的并发规则与 CR_{csp} 的作用是等价的.与之不同,基于迹的并发规则只在语义模型上计算,通过规则的匹配生成新进程的迹.该并发方式无需步步进行递归调用,因而可避免生成大量中间变量.此外,充分地利用已生成的关键迹模型,提高了知识重用.

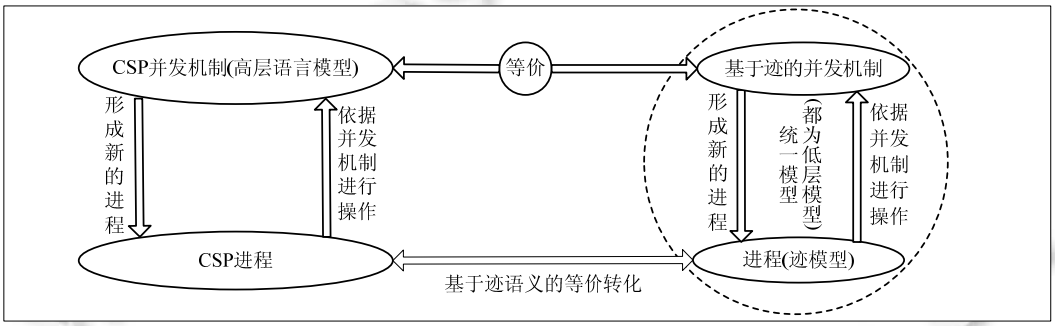


Fig.2 Relationship between CR_{csp} and concurrent manner based on traces of processes

图 2 CSP 的并发规则 CR_{csp} 与基于迹的并发规则之间的关系

3.3.3 基于迹的并发规则

进程并发时,需特别考虑以下情况:参与并发的两个进程,一个已经发生了若干事件,另一个还处于初始事件.为了准确定位将要参与并发的事件,需对参与并发的进程的事件流进行标记.

为此,引入谓词 *initial_flow*(*X*,*Y*,*P*)表示当前参与并发的事件流为 *flow*(*X*,*Y*,*P*),*next_flow*(*X*,*Y*,*P*)表示参与并发的下一事件流为 *flow*(*X*,*Y*,*P*).并发事件的定位方法是:首先标记两进程初始参与并发的事件,发生事件的进程的标记后移一位,另一进程标记不变.事件的定位策略可用 ASP 如下表示:

- *initial_flow*(*X*,*Y*,*P*):-*flow*(*X*,*Y*,*P*),*first*(*X*,*P*);
- *next_flow*(*Y*,*Z*,*P*):-*initial_flow*(*X*,*Y*,*P*),*flow*(*X*,*Y*,*P*),*flow*(*Y*,*Z*,*P*);
- *initial_flow*(*Y*,*Z*,*P*):-*initial_flow*(*X*,*Y*,*P*),*next_flow*(*Y*,*Z*,*P*),*flow*(*X*,*Y*,*T*),*concurrent*(*P*,*Q*,*T*).

为了等价地实现 CR_{csp} 的并发功能,本节给出了迹并发结构的最小完备集,共 10 种基本结构.进程的所有并发情况都可归纳为这 10 种基本结构.下面分 3 组情况分别来定义并发规则.

1. 图 3 给出了第 1 组基本结构,该结构需遵循规则 CR_{csp}-1 和 CR_{csp}-2,可用 ASP 表示为

flow(*X*,*Y*,*T*):-*initial_flow*(*X*,*Y*,*P*),*initial_flow*(*X*,*Y*,*Q*),not *fail_flow*(*X*,*Y*,*P*),not *fail_flow*(*X*,*Y*,*Q*),*event*(*X*,*P*),

event(*Y*,*P*),*event*(*X*,*Q*),*event*(*Y*,*Q*),*concurrent*(*P*,*Q*,*T*);

:-*initial_flow*(*X*,*Y*,*P*),*initial_flow*(*X*,*Z*,*Q*),not *fail_flow*(*X*,*Y*,*P*),not *fail_flow*(*X*,*Z*,*Q*),
event(*X*,*P*),*event*(*Y*,*P*),*event*(*X*,*Q*),*event*(*Y*,*Q*),*event*(*Z*,*P*),*concurrent*(*P*,*Q*,*T*);

:-*initial_flow*(*X*,_,*P*),*initial_flow*(*Y*,_,*Q*),not *fail_flow*(*X*,_,*P*),not *fail_flow*(*Y*,_,*Q*),*event*(*X*,*P*),
event(*Y*,*P*),*event*(*X*,*Q*),*event*(*Y*,*Q*),*concurrent*(*P*,*Q*,*T*).

其中,谓词 *fail_flow*(*X*,*Y*,*P*)表示事件流 *flow*(*X*,*Y*,*P*)已经失效,可生成辅助规则:

fail_flow(*X*,*Y*,*P*):-*flow*(*X*,*Y*,*T*),*concurrent*(*P*,*Q*,*T*).

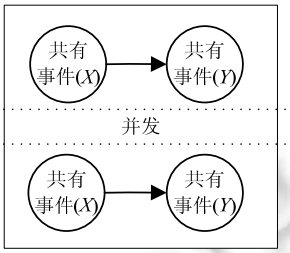


Fig.3 First group of basic structure of concurrency based on the traces of processes

图 3 基于迹并发的基本结构 1

2. 图 4 给出了第 2 组基本结构,共包含 4 种形式.其中,形式(1)、形式(2)和形式(4)都需遵循规则 CR_{csp}-1~CR_{csp}-4,形式(3)需遵循 CR_{csp}-3 和 CR_{csp}-4.下面分别加以描述.

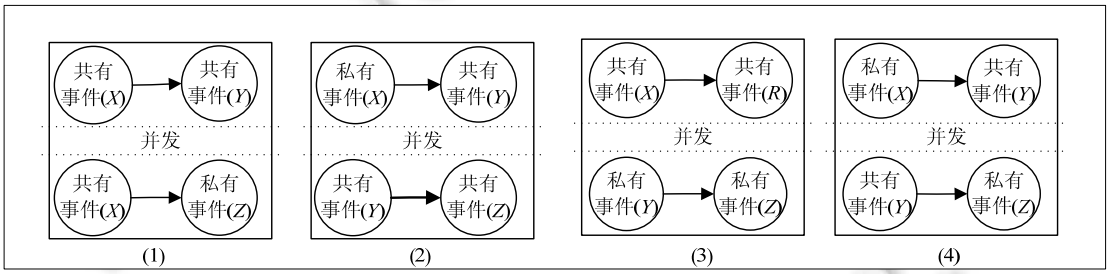


Fig.4 Second group of basic structure of concurrency based on the traces of processes

图 4 基于迹并发的基本结构 2

- 图 4-(1)可用 ASP 表示为

flow(*X*,*Y*,*T*):-*initial_flow*(*X*,*Y*,*P*),*initial_flow*(*X*,*Z*,*Q*),not *fail_flow*(*X*,*Y*,*P*),not *fail_flow*(*X*,*Z*,*Q*),*event*(*Y*,*Q*),
not *event*(*Z*,*P*),*concurrent*(*P*,*Q*,*T*);

:-*initial_flow*(*X*,*Y*,*P*),*initial_flow*(*Z*,*R*,*Q*),not *fail_flow*(*X*,*Y*,*P*),not *fail_flow*(*X*,*Y*,*Q*),*event*(*X*,*Q*),
event(*Z*,*P*),*event*(*Y*,*Q*),not *event*(*R*,*P*),*concurrent*(*P*,*Q*,*T*);

- 图 4-(2)可用 ASP 表示为

flow(*X*,*Y*,*T*):-*initial_flow*(*X*,*Y*,*P*),*initial_flow*(*Y*,*Z*,*Q*),not *fail_flow*(*X*,*Y*,*P*),not *fail_flow*(*Y*,*Z*,*Q*),not *event*(*X*,*Q*),
event(*Y*,*P*),*event*(*Z*,*P*),*concurrent*(*P*,*Q*,*T*);

:-*initial_flow*(*X*,*Y*,*P*),*initial_flow*(*R*,*Z*,*Q*),not *fail_flow*(*X*,*Y*,*P*),not *fail_flow*(*R*,*Z*,*Q*),not *event*(*X*,*Q*),
event(*Y*,*Q*),*event*(*R*,*P*),*event*(*Z*,*P*),*Q*≠*Y*,*concurrent*(*P*,*Q*,*T*);

- 图 4-(3)可用 ASP 表示为

flow(*Y*,*Z*,*T*):-*initial_flow*(*X*,*R*,*P*),*initial_flow*(*Y*,*Z*,*Q*),not *fail_flow*(*X*,*R*,*P*),not *fail_flow*(*Y*,*Z*,*Q*),*event*(*X*,*Q*),
event(*R*,*Q*),not *event*(*Y*,*P*),not *event*(*Z*,*P*),*concurrent*(*P*,*Q*,*T*);

- 图 4-(4)可用 ASP 表示为

flow(*X*,*Y*,*T*):-*initial_flow*(*X*,*Y*,*P*),*initial_flow*(*Y*,*Z*,*Q*),not *fail_flow*(*X*,*Y*,*P*),not *fail_flow*(*Y*,*Z*,*Q*),not *event*(*X*,*Q*),
event(*Y*,*P*),not *event*(*Z*,*P*),*concurrent*(*P*,*Q*,*T*);
:-*initial_flow*(*X*,*Y*,*P*),*initial_flow*(*R*,*Z*,*Q*),not *fail_flow*(*X*,*Y*,*P*),not *fail_flow*(*R*,*Z*,*Q*),not *event*(*X*,*Q*),
event(*Y*,*Q*),*event*(*R*,*P*),not *event*(*Z*,*P*),*Q*≠*Y*,*concurrent*(*P*,*Q*,*T*).

3. 图 5 给出了第 3 组基本结构,共包含 5 种形式.

图 5-(1)可用 ASP 表示为

flow(*X*,*Y*,*T*) *v flow*(*X*,*Y*,*T*):-*initial_flow*(*X*,*Y*,*P*),*initial_flow*(*X*,*Z*,*Q*),not *fail_flow*(*X*,*Y*,*P*),not *fail_flow*(*X*,*Z*,*Q*),
not *event*(*Y*,*Q*),*not event*(*Z*,*P*),*concurrent*(*P*,*Q*,*T*);
:-*initial_flow*(*X*,*Y*,*P*),*initial_flow*(*R*,*Z*,*Q*),not *fail_flow*(*X*,*Y*,*P*),not *fail_flow*(*R*,*Z*,*Q*),
event(*X*,*Q*),not *event*(*Y*,*Q*),not *event*(*Z*,*P*),*concurrent*(*P*,*Q*,*T*);

图 5-(2)可用 ASP 表示为

flow(*X*,*Y*,*T*) *v flow*(*Y*,*X*,*T*):-*initial_flow*(*X*,*Z*,*P*),*initial_flow*(*Y*,*Z*,*Q*),not *fail_flow*(*X*,*Z*,*P*),not *fail_flow*(*Y*,*Z*,*Q*),
not *event*(*X*,*Q*),not *event*(*Y*,*P*),*concurrent*(*P*,*Q*,*T*);

图 5-(3)可用 ASP 表示为

flow(*X*,*Y*,*T*) *v flow*(*Y*,*X*,*T*) *v flow*(*Y*,*X*,*T*):-*initial_flow*(*X*,*R*,*P*),*initial_flow*(*Y*,*Z*,*Q*),not *fail_flow*(*X*,*R*,*P*),
not *fail_flow*(*Y*,*Z*,*Q*),not *event*(*X*,*Q*),not *event*(*Y*,*P*),*event*(*R*,*P*),
not *event*(*Z*,*P*),*concurrent*(*P*,*Q*,*T*);

图 5-(4)可用 ASP 表示为

flow(*X*,*Y*,*T*) *v flow*(*Y*,*X*,*T*) *v flow*(*Z*,*X*,*T*) *v flow*(*X*,*Z*,*T*):-*initial_flow*(*X*,*Y*,*P*),*initial_flow*(*Z*,*R*,*Q*),not *fail_flow*(*X*,*R*,*P*),
not *fail_flow*(*Y*,*Z*,*Q*),not *event*(*X*,*Q*),not *event*(*Y*,*Q*),
not *event*(*R*,*P*),not *event*(*Z*,*P*),*concurrent*(*P*,*Q*,*T*);

图 5-(5)可用 ASP 表示为

flow(*R*,*Z*,*T*):-*initial_flow*(*X*,*Y*,*P*),*initial_flow*(*R*,*Z*,*Q*),not *fail_flow*(*X*,*Y*,*P*),not *fail_flow*(*R*,*Z*,*Q*),*event*(*X*,*Q*),
not *event*(*Y*,*Q*),not *event*(*R*,*P*),not *event*(*Z*,*P*),*concurrent*(*P*,*Q*,*T*).

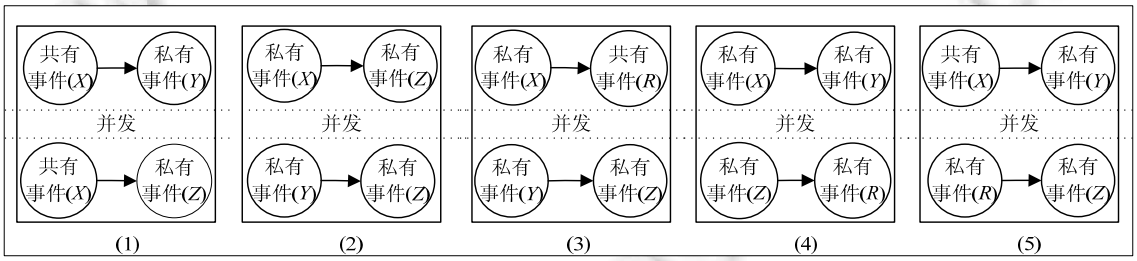


Fig.5 Third group of basic structure of concurrency based on the traces of processes

图 5 基于迹并发的基本结构 3

4 基于关键迹模型的自动化验证

4.1 ASP框架下LTL验证的基本思想

通过关键迹模型的生成机制,可将并发系统 CSP 模型转化为相应的关键迹模型,且每条回答集对应于关键迹中的一条迹.因此,通过验证回答集是否满足某性质,即可确定相应并发系统是否满足该性质.验证的基本思想是:当要验证一个模型是否满足性质ψ时,先对性质取反(即$\neg\psi$),接着构造性质变换目标程序$\Pi_{ans}(\neg\psi)$,然后调用求解器验证模型是否满足$\neg\psi$.若无回答集,则说明该模型满足性质ψ;若存在回答集,则说明该模型不满足性

质ψ,且相应的回答集为不满足的反例.验证流程如图 6 所示.

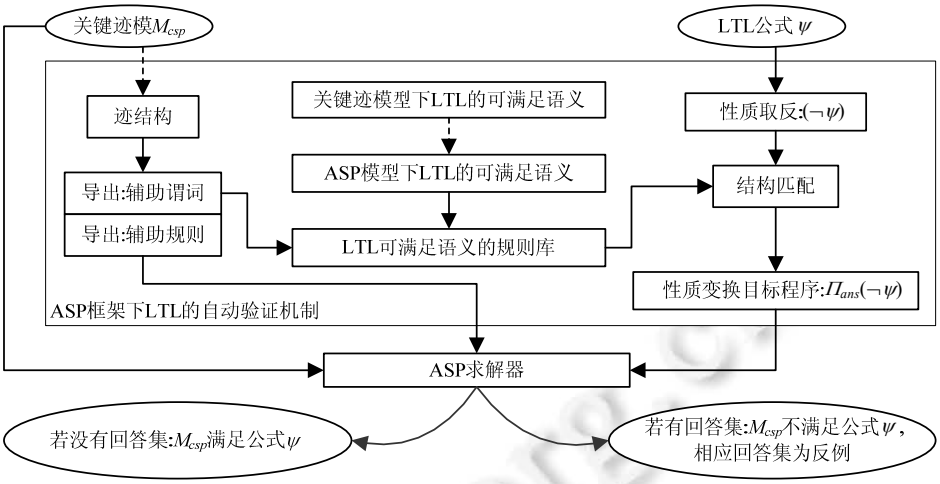


Fig.6 Verification of LTL based on ASP framework

图 6 基于 ASP 框架的 LTL 验证流程

4.2 回答集模型下LTL的可满足性语义

ASP 框架下得到的回答集与待验证进程关键迹模型中的迹是一一对应的,故可把关键迹模型下的可满足性语义归结为回答集模型下的可满足性语义.下面定义基于回答集模型的 LTL 可满足性语义.

定义 9. 给定待验证进程 Q,M_{ans} 是与 *critical-traces*(Q)对应的回答集的集合,LTL 公式在回答集$\pi_{ans}\in M_{ans}$下的可满足性定义为

$\pi_{ans}\models p$ iff $\exists X, flow(X,p,Q)\in\pi_{ans}\vee flow(p,X,Q)\in\pi_{ans}$

$\pi_{ans}\models\neg p$ iff $\pi_{ans}\not\models p$

$\pi_{ans}\models\psi_1\wedge\psi_2$ iff $\pi_{ans}\models\psi_1$ 且 $\pi_{ans}\models\psi_2$

$\pi_{ans}\models\psi_1\vee\psi_2$ iff $\pi_{ans}\models\psi_1$ 或 $\pi_{ans}\models\psi_2$

$\pi_{ans}\models F\psi$ iff $\exists X.\pi_{ans}(X)\models\psi$

$\pi_{ans}\models X\psi$ iff $first(X,Q)\wedge flow(X,Y,Q)\in\pi_{ans}\wedge\pi_{ans}(Y)\models\psi$

$\pi_{ans}\models G\psi$ iff $\pi_{ans}\models\neg F\neg\psi$

其中,p,ψ,ψ_1,ψ_2 的含义与定义 8 中相同,集合$\pi_{ans}(X)$的构造如下:若存在事件 X,Y 满足 $flow(X,Y,Q)\in\pi_{ans}$,则$\pi_{ans}(X)=\{flow(X,Y,Q)\}\vee\pi_{ans}(Y)$;若不存在事件 X,Y 满足 $flow(X,Y,Q)\in\pi_{ans}$,则$\pi_{ans}(X)=\varnothing$.如果对任意$\pi_{ans}\in M_{ans}$均满足$\pi_{ans}\models S_{LTL}$,则称回答集模型满足公式ψ,记作 $M_{ans}\models S_{LTL}$.

4.3 性质变换目标程序$\Pi_{ans}(\neg\psi)$

为了构建性质ψ的目标程序$\Pi_{ans}(\neg\psi)$,需用 ASP 刻画基于回答集模型的 LTL 可满足语义.由关键迹模型的生成机制可知,回答集与关键迹模型中的迹有一一对应关系.关键迹模型的迹包含下面两种结构:带环的递归结构和不带环的前缀结构,如图 7 所示.因此,回答集也只包含这两种结构,即,系统提供的反例包含这两种情形.

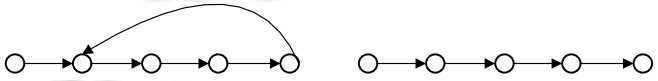


Fig.7 Two structures of traces included in the critical-trace model

图 7 关键迹模型所包含迹的两种结构

为了刻画基于回答集模型的LTL可满足语义,还需引入辅助谓词 *le*,*el*(*X*,*T*),*nl*(*X*,*Y*,*T*),*bel*(*X*,*Y*,*T*).其中,*le* 表示迹中存在环结构,*el*(*X*,*T*)表示事件 *X* 是进程 *T* 中环形路径的入口,*nl*(*X*,*Y*,*T*)表示 *flow*(*X*,*Y*,*T*)是前缀表示式末端到前缀中某一原子的流关系,*bel*(*X*,*Y*,*T*)表示 *flow*(*X*,*Y*,*T*)属于环形路经中的一个流关系.

当回答集中包含一个环时,可用 ASP 规则表示为 *el*(*X*,*T*):-*flow*(*Y*,*X*,*T*),*flow*(*Z*,*X*,*T*).

辅助谓词间的相互关系可用 ASP 表示为

- *le*:-*el*(*X*,*T*);
- *nl*(*X*,*Y*,*T*):-*flow*(*X*,*Y*,*T*),*el*(*Y*,*T*);
- *bel*(*X*,*Y*,*T*):-*el*(*X*,*T*),*flow*(*X*,*Y*,*T*);
- *bel*(*Y*,*Z*,*T*):-*flow*(*X*,*Y*,*T*),*bel*(*Y*,*Z*,*T*).

依据上述辅助规则,LTL 可满足语义的转化结果见表 1,其中,*pl*(*X*,*T*)表示 *X* 是待验证进程 *T* 发生的事件.

Table 1 Rule base of satisfiability semantics of LTL under ASP model

表 1 ASP 模型下 LTL 可满足语义的规则库

公式 f	对应的 ASP 形式
p(p 为原子命题)	f:-$f(X,Y,T)$; $f(X,Y,T)$:-$flow(X,Y,T)$,$pl(X,T)$.
$\neg p$	f:-$f(X,Y,T)$; $f_1(X,Y,T)$:-$flow(X,Y,T)$,$pl(X,T)$; $f(X,Y,T)$:-not $f_1(X,Y,T)$.
$f_1 \wedge f_2$	f:-f_1,f_2.
$f_1 \vee f_2$	f:-f_1; f:-f_2.
$G f_1$	f:-le,not q; q:-$f_2(X,Y,T)$; $f_2(X,Y,T)$:-not $f_1(X,Y,T)$,$flow(X,Y,T)$.
$F f_1$	f:-le,q; q:-$f_1(X,Y,T)$,$bel(X,Y,T)$.
$X f_1$	f:-$f(Z,X,T)$; $f(Z,X,T)$:-$f_1(X,Y,T)$,$flow(Z,X,T)$,$first(Z,T)$.

通过表 1,可构建性质变换目标程序 $\Pi_{ans}(\neg\psi)$,进而把性质验证归结为回答集求解.$\Pi_{ans}(\neg\psi)$的构建过程如下:对所有待验证的 LTL 公式 ψ取反,为$\neg\psi$,然后,由外向内分析$\neg\psi$的基本结构;最后,递归调用表 1 中的相应规则,即可得到目标程序 $\Pi_{ans}(\neg\psi)$.下面举例说明.

给定待验证进程 *P* 及 LTL 公式 $h=F1.up \wedge F2.up \wedge F3.up$,其中,1.*up*,2.*up* 和 3.*up* 是进程 *P* 的原子事件.构建步骤如下:

(1) 取反,设 $f=\neg h=G\neg 1.up \vee G\neg 2.up \vee G\neg 3.up$;

(2) 由外向内分析 *f*,设 $f=f_1 \vee f_2 \vee f_3$,$f_1=G\neg 1.up$,$f_2=G\neg 2.up$,$f_3=G\neg 3.up$,$t_1=\neg 1.up$,$s_1=\neg 2.up$,$r_1=\neg 3.up$;

(3) 依上面(2)中的结构调用表 1,可得如下目标程序:

- f:-f_1;
- f:-f_2;
- f:-f_3;
- f_1:-le,not q;
- q:-$t_2(X,Y,T)$;
- $t_2(X,Y,T)$:-not $t_1(X,Y,T)$,$flow(X,Y,T)$;
- $t_1(X,Y,T)$:-not $t_3(X,Y,T)$;
- $t_3(X,Y,T)$:-$flow(X,Y,T)$,$pl(X,T)$;
- pl(1.*up*,p);
- f_2:-le,not p;
- p:-$s_2(X,Y,T)$;

- $s_2(X,Y,T)$:-not $s_1(X,Y,T)$,$flow(X,Y,T)$;
- $s_1(X,Y,T)$:-not $s_3(X,Y,T)$;
- $s_3(X,Y,T)$:-$flow(X,Y,T)$,$pl(X,T)$;
- f_3:-le,not o;
- o:-$r_2(X,Y,T)$;
- $r_2(X,Y,T)$:-not $r_1(X,Y,T)$,$flow(X,Y,T)$;
- $r_1(X,Y,T)$:-not $r_3(X,Y,T)$;
- $r_3(X,Y,T)$:-$flow(X,Y,T)$,$pl(X,T)$;
- $pl(2.up,p)$;
- $pl(3.up,p)$.

其中,p 表示进程 P.

4.4 CSP模型检测原型系统——T_ASP

T_ASP 系统的结构如图 8 所示,其输入为 CSP 模型和 LTL 公式:CSP 模型用来描述并发系统;LTL 公式用来表示待验证性质,由用户提供.性质可满足时,系统输出 Yes;性质不满足时,输出 No+反例集 CE.

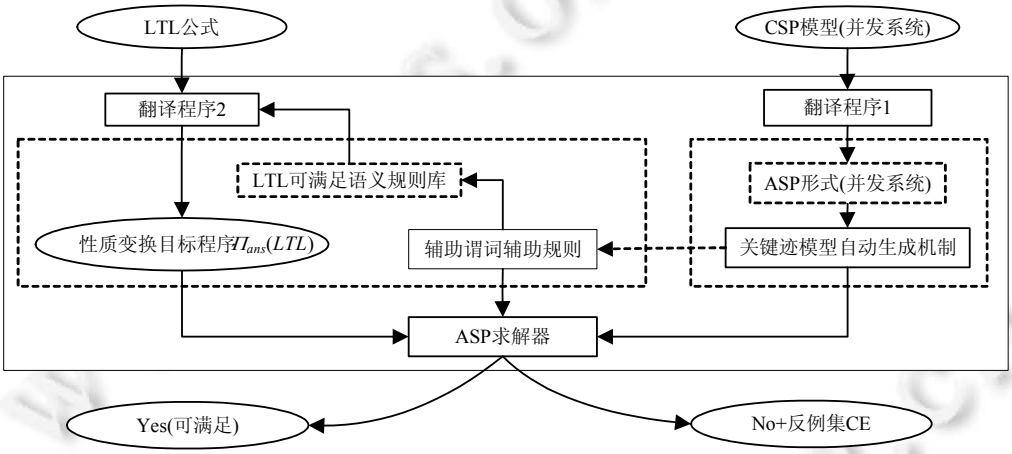


Fig.8 CSP model checker—T_ASP

图 8 CSP 模型检测系统——T_ASP

T_ASP 主要含有两大核心组件:关键迹模型自动生成机制和 LTL 的自动验证机制,其中,

- 关键迹模型生成机制包括基本进程转化规则库、选择机制规则库以及迹的并发规则库,具体结构如图 1 所示;
- LTL 的自动验证机制包括基于迹结构的辅助规则、LTL 可满足语义规则库以及构建性质变换目标程序$\Pi_{ans}(LTL)$,具体结构如图 6 所示.

此外,T_ASP 还包含辅助组件翻译程序 1 和翻译程序 2,它们为相应的输入提供接口.

- 翻译程序 1 的任务是:对 CSP 进程进行编译,将其转化为 ASP 求解器可以识别的进程知识.其基本思想是:先将 CSP 模型转化为进程并发树 CTP(CTP 中包含一棵主树 *CTP_Host*、若干棵子树 *CTP_Sub*);然后,逐个分析 *CTP_Host* 的所有节点,并生成相应的知识,具体算法如图 9 所示,该算法所调用的函数 $ConcurTree_ofProcesses(CSP_M,P)$如图 10 所示;
- 翻译程序 2 的作用为:将 LTL 公式ψ取反,从外向内依次匹配可满足语义规则(即表 1),生成性质变换目标程序$\Pi_{ans}(\neg\psi)$.其基本思想是:递归地分析 LTL 公式的具体结构,并构造出相应的基本结构库;然后,

根据这些基本结构和可满足语义规则库生成相应的$\Pi_{ans}(\neg\psi)$.具体算法如图 11 所示,其调用的函数 *RecurAnaly*($Unit_{\psi}$)如图 12 所示.

算法 1. 将并发系统 *M* 的 CSP 模型——CSP_M转换为对应的 ASP 程序——*ASP*(CSP_M).

Input: CSP_M,P_V; //P_V表示待验证的进程名;

Output: *ASP*(CSP_M).

```
1   判定 P_V 是否为并发系统的最顶层进程:若是,将进程 P_V 置为顶层进程 P;若不是,找出 CSP_M 中的顶层进程 P;
2   ConcurTree_ofProcesses(CSP_M,P);  //此函数的作用是构建根据 CSP_M 的进程并发树,即,构建 CTP;
3   while CTreeNode∈CTP_Host do
4     if CTreeNode 为并发进程 then  //构建主树上每个节点的知识库
5       判定 CTreeNode 的孩子节点的个数 Num_ChildNodes,构建 CTreeNode=ChildNode(1)||…||ChildNode(N);
6       构建 CTreeNode 知识库 K1;更新 ASP(CSP_M)=ASP(CSP_M)∪K1;
7     else  //若 CTreeNode 为 CTP_Host 的叶子节点,则构造其对应子树 SubTree(CTreeNode)的知识库;
8       查找子树 SubTree(CTreeNode);
9       while STNode∈SubTree(CTreeNode) do
10        if STNode 有唯一孩子节点 STCNode,且该孩子节点不是 CTP_Host 中的节点 then
11          生成知识库 K2,更新 ASP(CSP_M)=ASP(CSP_M)∪K2;
12        elseif STNode 有多个孩子节点 STCNode(1…i…),且孩子节点不是 CTP_Host 中的节点 then
13          构造知识库 K3,更新 ASP(CSP_M)=ASP(CSP_M)∪K3;
14        else  //STNode 的孩子节点是 CTP_Host 中的节点
15          构造知识库 K4,更新 ASP(CSP_M)=ASP(CSP_M)∪K4;
16        endif
17      endwhile
18    endif
19  endwhile
```

Fig.9 Pseudocode for translator program one

图 9 翻译程序 1 算法的伪代码

Function: *ConcurTree_ofProcesses*(CSP_M,*P*) //用于构建 CTP;CTP 主树由进程名组成,叶子节点对应一棵子树,由行为事件构成.

```
设置并发进程集合 CP={P},CTP_Host(主树)=∅,CTP_Sub(子树集合)=∅;
1   while CP 中的任意元素 do
2     if CP_element=P(1)||…||P(n) then
3       在 CTP_Host 主树中添加 CP_element 为根节点,P(1),…,P(n)为其孩子节点,并将{P(1),…,P(n)}加入集合 CP;
4     else  //CP_element 不是并发进程,构建 CTP 的子树
5       if CP_element 属于前缀形式 then
6         依据进程的行为,构建相应的树形结构 CTP_Sub(CP_element),并标识子树所属形式;
7       elseif CP_element 属于递归形式 then
8         依据进程的行为,构建相应的树形结构 CTP_Sub(CP_element),并标识子树所属形式;
9       elseif CP_element 属于一般选择形式 then
10        依据进程的行为结构,构建相应的树形结构 CTP_Sub(CP_element),并标识子树所属形式;
11      else CP_element 属于非确定选择形式 then
12        依据进程的行为结构,构建相应的树形结构 CTP_Sub(CP_element),并标识子树所属形式;
13      endif
14      CTP_Sub=CTP_Sub∪CTP_Sub(CP_element)
15    endif
16  endwhile
17  CTP=CTP_Host∪CTP_Sub;
18  Return CTP
```

Fig.10 Pseudocode for *ConcurTree_ofProcesses*(CSP_M,*P*)

图 10 函数 *ConcurTree_ofProcesses*(CSP_M,*P*)的伪代码

算法 2. 将 ψ转化为$\Pi_{ans}(\neg\psi)$; //ψ表示待验证的性质,$\Pi_{ans}(\neg\psi)$是相应的目标程序.
Input: ψ,*BeSatisfiedRules*,P_V; //*BeSatisfiedRules* 表示可满足语义规则库,即表 1 内容;
Out put: $\Pi_{ans}(\neg\psi)$.
将ψ转化为$\neg\psi$,$SubUnit_{\psi}=\neg\psi$,$\Pi_{ans}(\neg\psi)=\varnothing$;
1 *RecurAnaly*($SubUnit_{\psi}$);
2 **While** 每个元素 $AtomSubUnit_{\psi}\in RecurAnaly(SubUnit_{\psi})$ **do**
3 **if** $AtomSubUnit_{\psi}$属于原子结构 **then**
4 根据 *BeSatisfiedRules* 中的原子结构规则和 P_V,生成相应的知识集合 K_1;更新$\Pi_{ans}(\neg\psi)=\Pi_{ans}(\neg\psi)\cup K_1$;
5 **elseif** $AtomSubUnit_{\psi}$属于∧结构 **then**
6 根据 *BeSatisfiedRules* 中的∧结构规则,生成相应的知识集合 K_2;更新$\Pi_{ans}(\neg\psi)=\Pi_{ans}(\neg\psi)\cup K_2$;
7 **elseif** $AtomSubUnit_{\psi}$属于∨结构 **then**
8 根据 *BeSatisfiedRules* 中的∨结构规则,生成相应的知识集合 K_3;更新$\Pi_{ans}(\neg\psi)=\Pi_{ans}(\neg\psi)\cup K_3$;
9 **elseif** $AtomSubUnit_{\psi}$属于 *G* 结构 **then**
10 根据 *BeSatisfiedRules* 中的 *G* 结构规则,生成相应的知识集合 K_4;更新$\Pi_{ans}(\neg\psi)=\Pi_{ans}(\neg\psi)\cup K_4$;
11 **elseif** $AtomSubUnit_{\psi}$属于 *F* 结构 **then**
12 根据 *BeSatisfiedRules* 中的 *F* 结构规则,生成相应的知识集合 K_5;更新$\Pi_{ans}(\neg\psi)=\Pi_{ans}(\neg\psi)\cup K_5$;
13 **else** //$AtomSubUnit_{\psi}$属于 *X* 结构
14 根据 *BeSatisfiedRules* 中的 *X* 结构规则,生成相应的知识集合 K_6;更新$\Pi_{ans}(\neg\psi)=\Pi_{ans}(\neg\psi)\cup K_6$;
15 **endif**
16 **endwhile**
17 **return** $\Pi_{ans}(\neg\psi)$;

Fig.11 Pseudocode for translator program two
图 11 翻译程序 2 算法的伪代码

Function: *RecurAnaly*($Unit_{\psi}$) //该函数的作用是为算法 2 提供待转化的性质组件;设置 $SetUnit_{\psi}$的初始值为∅.
1 **if** $Unit_{\psi}$属于原子结构 **then**
2 更新 $SetUnit_{\psi}=Unit_{\psi}\cup SetUnit_{\psi}$;
3 **elseif** $Unit_{\psi}$属于∧结构 **then**
4 划分 $Unit_{\psi}$的各个子组件 $SubUnit_{\psi}(i)$,用 s_i 表示 $SubUnit_{\psi}(i)$,得到$\{s_1\vee\ldots\vee s_i\}$,$SetUnit_{\psi}=SetUnit_{\psi}\cup\{s_1\vee\ldots\vee s_i\}$;
5 **for** s_i 中的每个元素 **do**
6 *RecurAnaly*(s_i);
7 **endfor**
8 **elseif** $Unit_{\psi}$属于∨结构 **then**
9 划分 $Unit_{\psi}$的各个子组件 $SubUnit_{\psi}(i)$,用 k_i 表示 $SubUnit_{\psi}(i)$,得到$\{k_1\vee\ldots\vee k_i\}$,$SetUnit_{\psi}=SetUnit_{\psi}\cup\{k_1\vee\ldots\vee k_i\}$;
10 **for** k_i 的每个元素 **do**
11 *RecurAnaly*(k_i);
12 **endfor**
13 **elseif** $Unit_{\psi}$属于 *G* 结构 **then**
14 去除 $Unit_{\psi}$中的 *G* 算子,用 *r* 表示剩余的字符串,得到 *Gr*,$SetUnit_{\psi}=SetUnit_{\psi}\cup\{Gr\}$;
15 *RecurAnaly*(*r*);
16 **elseif** $Unit_{\psi}$属于 *F* 结构 **then**
17 去除 $Unit_{\psi}$中的 *F* 算子,用 *t* 表示剩余的字符串,得到 *Ft*,$SetUnit_{\psi}=SetUnit_{\psi}\cup\{Ft\}$;
18 *RecurAnaly*(*t*);
19 **elseif** $Unit_{\psi}$属于 *X* 结构 **then**
20 去除 $Unit_{\psi}$中的 *X* 算子,用 *w* 表示剩余的字符串,得到 *Xw*,$SetUnit_{\psi}=SetUnit_{\psi}\cup\{Xw\}$;
21 *RecurAnaly*(*w*);
22 **endif**
23 **return** $SetUnit_{\psi}$;

Fig.12 Pseudocode for *RecurAnaly*($Unit_{\psi}$)
图 12 函数 *RecurAnaly*($Unit_{\psi}$)的伪代码

5 实 验

5.1 哲学家就餐问题的CSP模型

为对不同进程算子进行实验,分别采用 3 种方法构造该问题.

- 第 1 种方法采用递归技术,具体描述如下:

$$Ph_i=i.sit\rightarrow i.pick\_fork.i\rightarrow i.pick\_fork.i\oplus 1\rightarrow i.down\_fork.i\rightarrow i.down\_fork.i\oplus 1\rightarrow i.up\rightarrow Ph_i \quad (i)$$

该方法对哲学家 *Phi* 的要求比较严格,其行为满足:先坐下来,拿起左边叉子,再拿右边的叉子,接着进餐完毕;然后放下左边叉子,再放右边叉子,最后离开座位.如此反复.

叉子进程:要么被左边的哲学家拿起,然后放下;要么被右边的科学家拿起,然后放下.可描述为

$$Fork.i=(i.pick.fork.i\rightarrow i.down.fork.i\rightarrow Fork.i\square i\ominus 1.pick.fork.i\rightarrow i\ominus 1.down.fork.i\rightarrow Fork.i) \quad \text{(ii)}$$

- 第 2 种方法采用一般选择及其嵌套方式进行描述,具体描述如下:

$$\begin{aligned} Ph_i=i.sit\rightarrow(&i.pick\_fork.i\rightarrow i.pick\_fork.i\oplus 1\rightarrow(i.down\_fork.i\rightarrow i.down\_fork.i\oplus 1\rightarrow i.up\rightarrow Ph_{i_c}\ \square i.down\_fork.i\\ &\oplus 1\rightarrow i.down\_fork.i\rightarrow i.up\rightarrow Ph_{i_c})\square i.pick\_fork.i\oplus 1\rightarrow i.pick\_fork.i\rightarrow(i.down\_fork.i\rightarrow\\ &i.down\_fork.i\oplus 1\rightarrow i.up\rightarrow Ph_{i_c}\ \square i.down\_fork.i\oplus 1\rightarrow i.down\_fork.i\rightarrow i.up\rightarrow Ph_{i_c})) \end{aligned} \quad \text{(iii)}$$

与进程(i)不同,哲学家进程开始时,对于拿起或放下叉子的左、右顺序不作限制,当选定某一方式后,哲学家按照已经选择的方式进行.每条分支最后调用的 Ph_{i_c} 与 Ph_i 被选择的路径有关,表示由 Ph_i 被选择的路径形成的自递归进程.例如:

若 Ph_i 满足选择机制的分支是 $i.sit\rightarrow i.pick\_fork.i\rightarrow i.pick\_fork.i\oplus 1\rightarrow i.down\_fork.i\rightarrow i.down\_fork.i\oplus 1\rightarrow i.up$,

那么 $Ph_{i_c}=i.sit\rightarrow i.pick\_fork.i\rightarrow i.pick\_fork.i\oplus 1\rightarrow i.down\_fork.i\rightarrow i.down\_fork.i\oplus 1\rightarrow i.up\rightarrow Ph_{i_c}$.

该哲学家进程分支的选择由与其交互的其他哲学家进程或者叉子进程提供.

叉子进程的 CSP 描述与进程(ii)相同.

- 第 3 种方法采用非确定选择及其嵌套进行描述,具体描述如下:

$$\begin{aligned} Ph_i=i.sit\rightarrow(&i.pick\_fork.i\rightarrow i.pick\_fork.i\oplus 1\rightarrow(i.down\_fork.i\rightarrow i.down\_fork.i\oplus 1\rightarrow i.up\rightarrow Ph_{i_c})\Pi(i.down\_fork.i\\ &\oplus 1\rightarrow i.down\_fork.i\rightarrow i.up\rightarrow Ph_{i_c})\Pi(i.pick\_fork.i\oplus 1\rightarrow i.pick\_fork.i\rightarrow(i.down\_fork.i\rightarrow\\ &i.down\_fork.i\oplus 1\rightarrow i.up\rightarrow Ph_{i_c})\Pi(i.down\_fork.i\oplus 1\rightarrow i.down\_fork.i\rightarrow i.up\rightarrow Ph_{i_c})) \end{aligned} \quad \text{(iv)}$$

进程(iii)与进程(ii)的含义基本相同,Ph_{i_c} 的定义也与进程(ii)相同;不同点在于:进程(ii)的选择由其他哲学家或叉子提供,进程(iii)的选择由自身提供.且进程(iii)通过规则 $x\rightarrow(P\Pi Q)=(x\rightarrow P)\Pi(x\rightarrow Q)$可转化为如下形式:

$$\begin{aligned} Ph_i=(&i.sit\rightarrow i.pick\_fork.i\rightarrow i.pick\_fork.i\oplus 1\rightarrow i.down\_fork.i\rightarrow i.down\_fork.i\oplus 1\rightarrow i.up\rightarrow Ph_{i_c})\Pi(i.sit\rightarrow i.pick\_fork.i\rightarrow\\ &i.pick\_fork.i\oplus 1\rightarrow i.down\_fork.i\oplus 1\rightarrow i.down\_fork.i\rightarrow i.up\rightarrow Ph_{i_c})\Pi(i.sit\rightarrow i.pick\_fork.i\oplus 1\rightarrow i.pick\_fork.i\rightarrow\\ &(i.down\_fork.i\rightarrow i.down\_fork.i\oplus 1\rightarrow i.up\rightarrow Ph_{i_c})\Pi(i.sit\rightarrow i.pick\_fork.i\oplus 1\rightarrow i.pick\_fork.i\rightarrow i.down\_fork.i\oplus 1\rightarrow\\ &i.down\_fork.i\rightarrow i.up\rightarrow Ph_{i_c}). \end{aligned}$$

叉子进程的 CSP 描述与进程(ii)相同.

哲学家就餐问题形成的并发系统 COLLGE 可用 CSP 描述如下:

$$Ph=Ph_1||Ph_2||Ph_3 \quad \text{(v)}$$

$$Fork=Fork_1||Fork_2||Fork_3 \quad \text{(vi)}$$

$$COLLGE_3=Ph||Fork \quad \text{(vii)}$$

5.2 实验结果

本实验分为两部分:(1) T_ASP 的验证能力和验证准确性实验;(2) 一次验证多条性质的实验.实验中,分别采用了 DLV,Smodels 和 Cmodels 这 3 种 ASP 求解器.实验平台配置如下:Win7,Inter(R) Core(TM) i3-2000 CPU @ 3.10GHz,RAM:4.00GB.

1. T_ASP 的验证能力和验证的准确性

为了说明 T_ASP 系统的验证效果,以哲学家就餐模型为例进行实验,并与文献[29]中的方法进行比较.实验主要从两方面进行:(1) 系统的验证能力;(2) 验证结果的准确性.简化起见,称文献[29]中的方法为 CS_ASP.

分别设定 M_1,M_2,M_3,M_4 和 M_5 对性质 P_1 验证.其中,$M_1=ph_1||ph_2$,表示哲学家 ph_1 和 ph_2 并发,$ph_i(i=1,2)$采用形式(i)进行描述;$M_2=ph_1||ph_2||ph_3$,表示哲学家 ph_1,ph_2 和 ph_3 并发,$ph_i(i=1,2,3)$也采用形式(i)进行描述;$M_3=ph_1||ph_2$,$ph_i(i=1,2)$采用形式(iii)进行描述;$M_4=ph_1||ph_2$,$ph_i(i=1,2)$采用形式(iv)进行描述;$M_5=ph_1||ph_2||ph_3$,$ph_i(i=1,2,3)$采用形式(iii)进行描述.待验证性质为 $P_1=F1.up\wedge F2.up$,即,哲学家 ph_1 和 ph_2 都可以用餐.表 2 给出了相应的实验结果

(CS_ASP 的并发步骤限制在 25 步以内).结果表明:T_ASP 调用 Cmodels 时验证效率最高,Smodels 次之,DLV 最低;CS_ASP 和 T_ASP 都可以完成对 M_1 的验证,在同种求解器下验证效率基本相当;CS_ASP 不能对 M_2,M_3,M_4 和 M_5 进行验证,这是由于 CS_ASP 中不具备分支进程的转化机制和并发的递归调用机制.

Table 2 Verification capabilities of CS_ASP and T_ASP

表 2 CS_ASP 和 T_ASP 验证能力的比较

验证方法			CSP 模型				
			M_1	M_2	M_3	M_4	M_5
CS_ASP	时间(s)	DLV	1.30	ND	ND	ND	ND
		Smodels	1.27	ND	ND	ND	ND
		Cmodels	1.30	ND	ND	ND	ND
	是否满足 P_1		是	不可验证	不可验证	不可验证	不可验证
T_ASP	时间(s)	DLV	1.39	2.77	4.79	4.07	6.87
		Smodels	1.22	2.23	3.90	3.43	4.95
		Cmodels	1.13	1.81	3.55	2.95	4.12
	是否满足 P_1		是	是	是	是	是

注:ND 表示相应的模型无法描述

T_ASP 不限定验证的步数,而 CS_ASP 是在有穷步骤内对模型进行验证的.为了比较这两种验证方法的效率和准确性,在 M_1 上对性质 P_1,P_2 和 P_3 进行验证.其中,P_2=G(1.*sit*→F1.*up*),即,ph_1 必然可以吃到东西;P_3=G(1.*up*→X2.*up*),即 ph_1 起身后离开,接着 ph_2 起身离开.由表 3 的实验结果可见:CS_ASP 和 T_ASP 在同种求解器下的验证效率相当,且调用 Cmodels 和 Smodels 的验证效率高于 DLV.此外,CS_ASP 验证结果满足 P_2,T_ASP 则输出不满足 P_2.这是由于 CS_ASP 的并发步骤数限定较少时产生了误判.验证 P_3 时,两种方法的验证结果都是不满足,但 T_ASP 还输出反例集 *CE*.

Table 3 Verification accuracy of CS_ASP and T_ASP

表 3 CS_ASP 和 T_ASP 验证准确性的比较

验证方法			性质		
			P_1	P_2	P_3
CS_ASP	验证时间(s)	DLV	1.30	1.44	1.87
		Smodels	1.27	1.31	1.66
		Cmodels	1.30	1.21	1.62
	验证结果		$S(P_1)$	$F(P_2)$(并发≤12)	$F(P_3)$
	是否误判		否	是	否
T_ASP	验证时间(s)	DLV	1.39	1.53	1.77
		Smodels	1.22	1.23	1.49
		Cmodels	1.13	1.02	1.44
	验证结果		$S(P_1)$	$S(P_2)$	$F(P_3)$+CE_1
	是否误判		否	否	否

注:$F(P_i)$表示不满足性质 P_i,$S(P_i)$表示满足性质 P_i,CE_i 表示反例集

2. 一次验证多条性质

在哲学家就餐模型上,实现 P_3 和 P_4 的同时验证以及 P_3 和 P_4 的独立验证.刻画 3 个哲学家就餐模型 M_6,M_7,M_8,其中,M_6=$COLLGE_3$,且 ph_i(i=1,2,3)采用形式(i)进行描述;M_7=$COLLGE_3$,且 ph_i(i=1,2,3)采用形式(ii)进行描述;M_8= $COLLGE_3$,且 ph_i(i=1,2,3)采用形式(iii)进行描述.对 $COLLGE_3$ 进行验证时,发生死锁的迹已经除去.此外,约定 $COLLGE_3$ 满足基本的安全性质,如,一个叉子不能同时被两个哲学家拿起.P_3 与前文相同,P_4=F1.*up*∧F2.*up*∧F3.*up* 表示 3 个哲学家都可以吃到饭.

由表 4 的实验结果可见:大部分情况下,P_3 和 P_4 同时验证消耗的时间要小于独立验证所消耗的时间之和,表明 1 次验证多条性质可提高性质验证效率.此外,M_6 的验证时间要远小于 M_7 和 M_8.这是由于建模方式越严格,模型的验证效率越高.

Table 4 Efficiency of independent and simultaneous verification of P_3 and P_4

表 4 性质 P_3 和 P_4 同时验证及其独立验证的效率

性质		模型					
		M_6		M_7		M_8	
		时间(s)	验证结果	时间(s)	验证结果	时间(s)	验证结果
P_3 与 P_4	DLV	28	$F(P_3),S(P_4)+CE_2$	187	$F(P_3),S(P_4)+CE_3$	>200	$F(P_3),S(P_4)+CE_4$
	Smodels	16.5	$F(P_3),S(P_4)+CE_2$	143	$F(P_3),S(P_4)+CE_3$	173	$F(P_3),S(P_4)+CE_4$
	Cmodels	12	$F(P_3),S(P_4)+CE_2$	108.5	$F(P_3),S(P_4)+CE_3$	167.5	$F(P_3),S(P_4)+CE_4$
P_3	DLV	13	$F(P_3)+CE_5$	87	$F(P_3)+CE_6$	96	$F(P_3)+CE_7$
	Smodels	8.5	$F(P_3)+CE_5$	63	$F(P_3)+CE_6$	80	$F(P_3)+CE_7$
	Cmodels	7	$F(P_3)+CE_5$	53	$F(P_3)+CE_6$	70.5	$F(P_3)+CE_7$
P_4	DLV	19	$S(P_4)$	121	$S(P_4)$	143	$S(P_4)$
	Smodels	10.5	$S(P_4)$	101	$S(P_4)$	119	$S(P_4)$
	Cmodels	9.5	$S(P_4)$	89	$S(P_4)$	116	$S(P_4)$

注:$F(P_i)$表示不满足性质 P_i,$S(P_i)$表示满足性质 P_i,CE_i 表示反例集

6 相关工作

CSP 的形式化验证包括程序正确性证明和模型检测两种方法:程序正确性证明主要在早期使用,并逐渐演化成为 Hoare 逻辑的公理语义[39];模型检测是当前 CSP 验证的主流方法,如 FDR[17,18],ARC[19],PAT[20],SymFDR[28]都采用模型检测技术对 CSP 进行验证.FDR 是应用最广的 CSP 验证工具,它通过操作语义将 CSP 转化为相应的迁移系统,进而显式地导出相应的指称语义模型,并完成性质验证.然而,FDR 中性质采用 CSP 语言进行描述,有利于精炼检测,但其通用性不强.ARC 是一种高效的 CSP 验证工具,它是通过操作语义将进程转化为相应的抽象语义模型——基于 OBDD 的符号模型.该方式可以有效地缓解状态爆炸问题,但是转化过程比较复杂.此外,ARC 也采用 CSP 语言进行性质规约,不利于性质的描述.与 ARC 类似,PAT 也是将进程转化为基于 OBDD 的符号模型;不同的是,PAT 采用有界模型检测方式,并基于时态逻辑描述性质.PAT 的不足之处是,对包含无穷序列的模型无法进行完全正确的判定.SymFDR 是在 FDR 的基础上结合 BMC 和 k-归纳方法构造的 CSP 检测工具:BMC 技术使 SymFDR 快速找到错误,k-归纳使其具有无界模型检测的能力.但是,SymFDR 继承了 FDR 的语义模型生成方式和性质规约方式.与上述验证工具不同,本文给出的 CSP 检测工具——T_ASP 并非基于操作语义,而是采用一种新的指称语义模型——关键迹模型,作为语义和实现模型.该模型包含数据量较小,一定程度上缓解了状态爆炸问题,且模型生成采用递归计算策略,易于理解,实现简单.与 PAT 类似,T_ASP 也采用基于事件的 LTL 公式对性质进行规范,通用性好,描述能力强.此外,T_ASP 还可以一次验证多条性质,一定程度上提高了验证效率.

与本文工作类似,Heljanko 提出了一种基于 ASP 的有界模型检测方法,其基本思想是:把 LTL 性质验证归约为回答集求解,利用 ASP 技术检测 Petri 网在有限步骤内的并发状况,从而完成系统的有界 LTL 模型验证[40].与此类似,文献[41]提出了一种基于 ASP 的 CTL 公式的可满足判定方法,把 CTL 的可满足性判断问题转化为偏好回答集计算.基于 Heljanko 的思想,文献[42]实现了有界抽象状态机(ASM)的 LTL 验证;文献[40,41]中的方法可以快速地对性质进行验证,并找出相应的反例,但是无法对系统进行无界模型验证.与上述工作不同,本文给出的 T_ASP 系统可对具有有穷迹语义的 CSP 进程进行无界模型检测,且输入为高层语言模型,由系统自动生成语义模型,极大地方便了并发系统的验证.

文献[29]初步探讨了基于 ASP 的 CSP 模型验证方法,实现了 CSP 模型和时态逻辑公式的 ASP 描述以及基于 ASP 求解器的性质验证方法,但其自动化程度不高,且相应的理论支撑不够.文献[30]中首次提出了基于进程迹的 CSP 模型验证思想,通过递归计算策略生成进程的迹模型,并通过 SAT 判定思想对 CSP 的迹模型进行性质验证.然而,迹模型通常比较庞大,所以验证效率不是很高.本文在文献[30]的基础上所提出的关键迹模型,只生成并发过程中实际发生的迹,数据量远小于迹模型,从而提高了验证效率.此外,系统地构建了关键迹模型的自动生成机制及基于关键迹模型的自动验证机制,与文献[30]相比,本系统可完全自动化.

7 总 结

本文主要工作可分为理论和实现两个方面.

- 理论方面,定义了一种 CSP 指称语义模型——关键迹模型,并给出了该模型的递归计算策略;论证了基于关键迹模型验证的可靠性;依据可满足性判定技术,提出了基于关键迹的性质验证方法;
- 实现方面,利用 ASP 技术构造了关键迹模型的自动生成方法以及 LTL 的自动验证方法.

在此基础上实现的 T_ASP 系统可完成 CSP 的基本验证,并具有以下特色:(1) 一次提供多条反例,便于系统诊断模块综合分析,错误较多时可减少诊断时间;(2) 采用 LTL 进行性质规约,通用性好,描述能力强;(3) 可一次验证多条性质;(4) 基于指称语义,相关理论易理解,系统实现简单、便于扩展和修改.

T_ASP 初步实现了上述功能,但在验证规模、反例分析以及适用场景方面还存在不足之处.未来研究集中在以下几个方面:

(1) 基于反例的系统调试技术.

反例生成与模型调整在验证过程中相辅相成,反例生成在 T_ASP 系统中已经完成,但尚不能进行模型调整. T_ASP 可利用非实例化 ASP 程序的调试技术分析已验证性质的支撑原因集合,进而对模型进行调整.目前,已存在多种实例化 ASP 程序的调试技术,如不一致性分析技术[43]、Justification 技术[44]、基于 ASP 的调试技术[45]、Meta-ASP 的调试技术[46].下一步工作可将实例化程序的调试技术引入非实例化程序[47],并应用于 T_ASP 的调试模块.

(2) 基于有界模型检测技术的关键迹模型验证.

有界模型检测技术只检测模型的有限前缀,可以减少内存需求,提高验证效率,但有可能提供错误的反例.下一步的研究中,将把增加完备性阈值[48]、*k*-归纳或 Craig interpolation 策略引入基于 ASP 的有界模型检测中,从而为性质验证提供可靠反例.

(3) 实现 ASP 平台下的增量式验证技术.

增量式验证可以提高知识重用,从而提高验证效率.但 ASP 平台下存在的理论难题是:如何确定新的性质加入后,系统的已验证性质继续可满足.因为只有已验证性质继续保持满足,该验证方法才是有效的.显然,该问题在单调逻辑中不会出现,然而对于非单调逻辑则是可能的.为此,可借助 ASP 语义研究的有关理论,如 Splitting 定理[49],研究满足上述关系的充分或充要条件.

(4) 面向可扩展需求的应用场景研究.

T_ASP 系统的优势之一在于其具有良好的可扩展性,可以较方便地把一些新的处理机制接入系统.因此,具有较强扩展需求的应用领域非常适合采用 T_ASP 系统.例如,轻量级 Web 组合中的适配问题,除了传统的适配验证之外,可以把基于 ASP 的适配机制引入 T_ASP,达到自动适配的目的.

References:

[1] Hoare CAR. Communicating Sequential Processes. Electronic Version, London: Prentice Hall, 2004. 1−122.

[2] Isobe Y, Roggenbach M. A complete axiomatic semantics for the CSP stable failures model. In: Proc. of the CONCUR 2006. LNCS, 2006. 158−172. [doi: 10.1007/11817949_11]

[3] Isobe Y, Roggenbach M. Proof principles of CSP—CSP-Prover in practice. In: Proc. of the LDIC 2007. Springer-Verlag, 2008. 425−442. [doi: 10.1007/978-3-540-76862-3_42]

[4] O'Reilly L, Roggenbach M, Isobe Y. CSP-CASL-Prover: A generic tool for process and data refinement. Electronic Notes in Theoretical Compututer Science, 2009,250(2):69−84. [doi: 10.1016/j.entcs.2009.08.018]

[5] Baier C, Katoen JP. Principles of Model Checking. Cambridge: MIT Press, 2007. 1−18.

[6] Winskel G. Topics in concurrency lecture notes. 2009. 20−24. http://www.cl.cam.ac.uk/~gw104/TIC.pdf

[7] McMillan KL. Interpolation and SAT-based model checking. In: Proc. of the CAV. 2003. 1−13. [doi: 10.1007/978-3-540-45069-6_1]

[8] Een N, Sorensson N. Temporal induction by incremental SAT solving. Electronic Notes in Theoretical Compututer Science, 2003, 89(4):543−560. [doi: 10.1016/S1571-0661(05)82542-3]

[9] Garavel H, Lang F. NTIF: A general symbolic model for communicating sequential processes with data. In: Proc. of the FORTE 2002. 2002. 276−291. [doi: 10.1007/3-540-36135-9_18]

[10] Holzmann G. The model checker SPIN. IEEE Trans. on Software Engineering, 1997,23(5):279−295. [doi: 10.1109/32.588521]

[11] Milner R. A Calculus of Communicating Systems. Springer-Verlag, 1980. [doi: 10.1007/978-1-4615-4020-5_2]

[12] Reisig W. Petri nets and algebraic specifications. Theoretical Computer Science, 1991,80(1):1−34. [doi: 10.1016/0304-3975(91)90203-E]

[13] Hoare CAR. A model for communicating sequential processes. In: Mc Keag RM, Mc Naghton AM, eds. On the Construction of Programs. Cambridge University Press, 1980. 33−44.

[14] SMV System. 2001. http://www.cs.cmu.edu/~modelcheck/smv.html

[15] Kronos. 1999. http://www-verimag.imag.fr/DIST-TOOLS/TEMPO/kronos/index-english.html

[16] Truth. 2002. http://www-i2.informatik.rwth-aachen.de/Forschung/MCS/Truth/index.html

[17] Armstrpng P, Goldsmith MH, Lowe G, Ouaknine J, Palikareva H, Roscoe A W, Worrell J. Recent developments in FDR2. In: Proc. of the CAV. 2012. 699−704. [doi: 10.1007/978-3-642-31424-7_52]

[18] Failures-Divergence Refinement: FDR2 User Manual. 2010. http://web.comlab.ox.ac.uk/projects/concurrency-tools/

[19] Parashkevov A, Yantchev J. ARC—A tool for efficient refinement and equivalence checking for CSP. In: Proc. of the IEEE Int'l Conf. on Algorithms and Architectures for Parallel Processing. 1996. 68−75. [doi: 10.1109/ICAPP.1996.562859]

[20] Sun J, Liu Y, Dong JS, Sun J. Bounded model checking of compositional processes. In: Proc. of the TASE. IEEE, 2008. 23−30. [doi: 10.1109/TASE.2008.12]

[21] Zhou CH, Liu ZF, Wang CD. Bounded model checking for probabilistic computation tree logic. Ruan Jian Xue Bao/Journal of Software, 2012,23(7):1656−1668 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/4089.htm [doi: 10.3724/SP.J.1001.2012.04089]

[22] Xia W, Yao YP, Mu XD. Model checking for event graphs and event temporal logic. Ruan Jian Xue Bao/Journal of Software, 2013, 24(3):421−432 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/4162.htm [doi: 10.3724/SP.J.1001.2013.04162]

[23] Clarke EM, Emerson EA, Sifakis J. Model checking: Algorithmic verification and debugging. Communications of the ACM, 2009,52:74−84. [doi: 10.1145/1592761.1592781]

[24] Meulen JV, Pecheur C. Combining partial order reduction with bounded model checking. In: Proc. of the Communicating Process Architectures 2009-WoTUG-32 on Concurrent Systems Engineering Series, Vol.67. 2009. 29−48. [doi: 10.3233/978-1-60750-065-0-29]

[25] Peled D. Ten years of partial order reduction. In: Proc. of the 10th Int'l Conf. on Computer Aided Verification (CAV'98). 1998. 17−28. [doi: 10.1007/BFb0028727]

[26] Biere A. Bounded model checking. In: Frontiers in Artificial Intelligence and Applications. 2003,Chapter 14:457−481. [doi: 10.3233/978-1-58603-929-5-457]

[27] Grumberg O, Long DE. Model checking and modular verification. ACM Trans. on Programming Languages and Systems, 1994(16): 843−871. [doi: 10.1145/177492.177725]

[28] Palikareva H, Ouaknine J, Roscoe AW. SAT-Solving in CSP trace refinement. Science of Computer Programming, 2012,77(10-11): 1178−1197. [doi: 10.1016/j.scico.2011.07.008]

[29] Zhao LZ, Zhang C, Qian JY. ASP-Based verification of concurrent systems described by CSP. Computer Science, 2012,39(12): 133−136 (in Chinese with English abstract).

[30] Zhao LZ, Zhai ZY, Qian JY. The framework for model checking CSP with traces of processes. Computer Science, 2013,40(11): 181−186 (in Chinese with English abstract).

[31] Baral C. Knowledge Representation, Reasoning, and Declarative Problem Solving. Cambridge Press, 2003. 1−60.

[32] Gelfond M, Lifschitz V. The stable model semantics for logic programming. In: Proc. of the Int'l Logic Programming Conf. and Symp. 1988. 1070−1080. [doi: 10.2307/2275201]

[33] Anger C, Konczak K, Linke T. A glimpse of answer set programming. Kunstliche Intelligenz, 2005,19(1):12−17.

[34] Smith AM, Mateas M. Answer set programming for procedural content generation: A design space approach. IEEE Trans. on Computational Intelligence and AI in Games, 2011,3(3):187−200. [doi: 10.1109/TCIAIG.2011.2158545]

[35] Lifschitz V. What is answer set programming? In: Proc. of the AAAI Conf. on Artificial Intelligence. Menlo Park: AAAI Press, 2008. 1594−1597.

[36] Leone N, Pfeifer G, Faber W, Eiter T, Gottlob G, Perri S, Scarcello F. The DLV system for knowledge representation and reasoning. ACM Trans. on Computational Logic, 2006,3(7):499−562. [doi: 10.1145/1149114.1149117]

[37] Roscoe AW. The Theory and Practice of Concurrency. London: Prentice Hall (Pearson), 1998. 183−218.

[38] Roscoe AW. Understanding Concurrent Systems. Heidelberg: Springer-Verlag, 2011. 191−317. [doi: 10.1007/978-1-84882-258-0]

[39] Apt KR. Ten years of Hoare's logic, A survey—Part I. ACM TOPLAS, 1981,3(4):431−483. [doi: 10.1145/357146.357150]

[40] Heljanko K, Niemelä I. Bounded LTL model checking with stable models. Theory and Practice of Logic Programming, 2003,4(3): 519−550. [doi: 10.1017/S1471068403001790]

[41] Tang CKF, Ternovska E. Model checking abstract state machines with answer set programming. In: Proc. of the LPAR. 2005. 443−458. [doi: 10.1007/11591191_31]

[42] Heymans S, Van Nieuwenborgh D, Vermeir D. Synthesis from temporal specications using preferred answer set programming. Lecture Note in Computer Sciences, 2005,3701:280−294. [doi: 10.1007/11560586_23]

[43] Syrjänen T. Debugging inconsistent answer set programs. In: Dix J, Hunter A, eds. Proc. of the NMR 2006. 2006. 77−83.

[44] Pontelli E, Son T. Justifications for logic programs under answer set semantics. In: Proc. of the ICLP 2006. Springer-Verlag, 2006. 196−210. [doi: 10.1007/11799573_16]

[45] Brain M, Gebser M, Puhrer J, Schaub T, Tompits H, Woltran S. Debugging ASP programs by means of ASP. In: Proc. of the LPNMR 2007. Springer-Verlag, 2007. 31−43. [doi: 10.1007/978-3-540-72200-7_5]

[46] Gebser M, Pührer J, Schaub T, Tompits H. A meta-programming technique for debugging answer-set programs. In: Fox D, Gomes CP, eds. Proc. of the 23rd AAAI Conf. on Artificial Intelligence. Menlo Park: AAAI Press, 2008. 13−17.

[47] Oetsch J, Puhrer J, Tompits H. Catching the ouroboros: Towards debugging non-ground answer-set programs. Theory and Practice of Logic Programming, 2010,10(4-6):513−529.

[48] Clarke E, Kröning D, Ouaknine J, Strichman O. Completeness and complexity of bounded model checking. In: Proc. of the VMCAI. 2004. 85−96. [doi: 10.1007/978-3-540-24622-0_9]

[49] Babb J, Lee JY. Module theorem for the general theory of stable models. Theory and Practice of Logic Programming, 2012,12(4-5): 719−735.

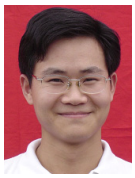
附中文参考文献:

[21] 周从华,刘志峰,王昌达.概率计算逻辑的限界模型检测.软件学报,2012,23(7):1656−1668. http://www.jos.org.cn/1000-9825/4089.htm [doi: 10.3724/SP.J.1001.2012.04089]

[22] 夏薇,姚益平,慕晓冬.面向事件图和事件时态逻辑的模型检验方法.软件学报,2013,24(3):421−432. http://www.jos.org.cn/1000-9825/4162.htm [doi: 10.3724/SP.J.1001.2013.04162]

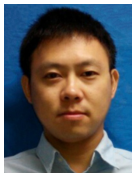
[29] 赵岭忠,张超,钱俊彦.基于 ASP 的 CSP 并发系统验证研究.计算机科学,2012,39(12):133−136.

[30] 赵岭忠,翟仲毅,钱俊彦.基于进程迹的 CSP 模型验证框架.计算机科学,2013,40(11):181−186.

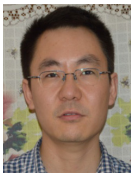


赵岭忠(1977－),男,河南社旗人,博士,教授,CCF 会员,主要研究领域为软件模型检测,形式化技术.

钱俊彦(1973－),男,博士,教授,CCF 高级会员,主要研究领域为软件验证,模型检验.



翟仲毅(1986－),男,博士生,主要研究领域为并发系统验证,Web 服务组合.



郭云川(1977－),男,博士,副教授,主要研究领域为隐私保护技术,安全评估与形式验证技术.