基于ASP的CSP并发系统验证研究

赵岭忠 张 超 钱俊彦
(桂林电子科技大学计算机科学与工程学院 桂林 541004)

摘 要 传统并发通信顺序进程(CSP)性质的验证通常使用3个不同的模型层面,从而增加了系统的复杂性和验证工具开发的难度;同时,主流的并发系统模型验证工具不支持在系统的一次运行中验证多个性质,这也降低了性质验证的效率。首先将CSP程序转换为ASP程序,然后将CSP进程并发规则和以LTL/CTL公式表示的待验证性质转换为ASP规则,从而建立了基于ASP验证CSP并发系统性质的统一框架。实验结果表明,基于ASP的CSP并发系统验证技术易于实现,在保持较高验证效率的同时,能够支持在验证软件的一次执行中验证多条LTL/CTL公式。

关键词 通信顺序进程,回答集编程,LTL/CTL
中图法分类号 TP311 **文献标识码** A

ASP-based Verification of Concurrent Systems Described by CSP

ZHAO Ling-zhong ZHANG Chao QIAN Jun-yan
(School of Computer Science and Engineering, Guilin University of Electronic Technology, Guilin 541004, China)

Abstract Traditionally, the verification of properties of Communicating Sequential Processes (CSP) is carried out on three different levels of models, which increases the complexity and difficulty of developing verification tools. At the same time, mainstream tools for verifying concurrent systems cannot verify multiple properties at one run of the tools, which decreases the verification efficiency of the tools. To deal with the problems, this paper proposed an ASP-based unified framework for verifying concurrent systems described by CSP. The method first transforms CSPs into the Answer Set Program, and then transforms the execution rules for concurrent CSP processes and the properties in the form of LTL/CTL formulae into the rules of ASP. At last, the properties can be verified by a computation of the answer sets of the resulting ASP program. It is shown that the ASP based method for verifying CSP concurrent system is easy to implement, is able to verify multiple LTL/CTL formulae at one execution of the verification software, and at the same time achieves acceptable time efficiency.

Keywords Communicating sequential processes, Answer set program, LTL/CTL

1 引言

通信顺序进程(CSP)是一种具有严格数学理论支撑的描述并发进程之间相互作用模式的语言[1]。为了验证CSP模型的性质,通常采用系统分析和模型检测技术[2-6],其中后者是目前研究的重点。典型的并发系统模型检测技术使用了3个层次的系统模型:高级语言、中介模型和低级模型。高级语言的设计具有表达能力强和方便用户使用的特征,其实例包括现有的形式化描述技术(如国际标准Lotos、E-Lotos等)以及专门设计的并发系统描述语言(如PROMELA等)[2]。低级模型则直接服务于模型验证算法的实现,典型实例包括标签转换系统、Kripke结构、Petri网、二叉决策图等[3,5,6]。高级语言和低层模型设计目标的差异性,导致直接把高级语言模型转化为低层模型通常具有一定的困难和较高的复杂度。为此,Garavel提出在高级语言和低级模型之间增加作为必要过度的中介模型[2]。虽然使用分层模型解决了并发系统验证中高层模型和低层模型差异较大转化困难的问题,但其也存在一些不足。一方面,验证系统的研发人员需同时考虑多个系统模型,增加了系统的复杂性,加重了系统开发的任务;另一方面,在验证过程中要多次进行高层模型到低级模型的翻译,保证该过程的正确性是对验证工具研发人员的挑战。此外,目前主流的并发系统验证方法,如基于Kripke结构的模型检测[3-6]、基于各种并发系统描述工具或语言(如Petri网、CSP等)的专用系统分析技术,其共同特点是不支持在验证工具的一次运行中验证多个性质,从而也限制了性质验证的效率。

解决以上问题的出路之一是建立基于统一模型的并发系统验证框架。为此面临的主要问题就是选择合适的并发系统

到稿日期:2012-02-13 返修日期:2012-06-08 本文受国家自然科学基金(61063002),广西自然基金(2011GXNSFA018166,2011GXNSFA018164),武汉大学软件工程国家重点实验室开放基金(SKLSE2010-08-06),广西可信软件重点实验室基金项目(kx201113),广西研究生教育创新计划项目(2010105950812M28)资助。
赵岭忠(1977—),男,博士,教授,主要研究方向为形式化技术,E-mail:zhaolingzhong163@163.com;张 超(1986—),男,硕士生,主要研究方向为并发系统验证;钱俊彦(1973—),男,博士,教授,主要研究方向为模型检验。

规格语言对系统进行建模。为了能够使验证工具在一次运行中验证多条系统性质，本文借助了具有声明特征的规格语言。ASP(Answer Set Programming)是采用回答集语义的逻辑程序设计(LP)语言，可以方便地用于非单调知识的表示和推理，且具有很强的声明性[7,8]。选择利用 ASP 构建并发程序 CSP 模型验证的统一框架具有以下优势：首先，作为一种较 CLP 和 Prolog 等语言更具声明性的 LP 语言[9,10]，ASP 不但使用方便，且具有强大的模型表达能力；其次，作为一种主流的知识表示和推理工具，ASP 存在众多高效的 ASP 回答集求解器。这些求解器具有较强的推理能力，能够胜任验证并发系统的特性所需的知识推理。

与本文工作类似，Angelis 等人提出了一种利用 ASP 进行并发进程合成的技术，利用该技术可以从多个并发进程生成一个满足所有行为特性和结构特性的新进程[11]。与此不同，本文的着重点则在于对多个并发进程是否满足用户期待的性质进行验证。此外，在将 LTL 公式和 CTL 公式转化为 ASP 方面，与本文工作类似的有 Heljanko 提出的基于稳定模型的 LTL 模型检测[12]。该方法提出将 Petri 网和 LTL 公式转化为 ASP 程序的方法。本文借鉴了基于稳定模型的 LTL 模型检测方法，探讨了将 CTL 公式转化为 ASP 规则的技术，从而使本文方法可同时应用于 CTL 公式性质的验证。开发基于 ASP 的 CSP 并发进程验证工具包括以下几个步骤：(1)将 CSP 进程转换为 ASP 程序；(2)将 CSP 进程并发规则和待验证系统性质转换为 ASP 规则；(3)利用回答集求解器进行系统性质的验证。验证工具开发的初步实践表明，基于以上思想的验证工具具有较高的开发效率；实验结果表明，本文方法在保持一定验证效率的同时，成功地实现了在验证工具的一次运行中验证多条系统性质的目标。

2 基础知识

2.1 通信进程并发系统

为了描述一个进程，首先要给出所感兴趣的进程事件。例如，一个简单自动售货机(VMS)通常关心投币(coin)、取出饮料(drink)两个事件。在 CSP 进程描述系统中，有以下约定：

序号	约定描述惯例	举例
1	用小写字母组合表示一个进程的不同的事件	coin、drink
2	用大写字母组合表示特定的进程或用 P、Q、R 表示任意进程	VMS、P、Q、R
3	用字母 x、y、z 表示事件的变量	x、y、z
4	用字母 A、B、C 代表事件的集合	A、B、C
5	用 X、Y 表示进程的变量	X、Y
6	进程 P 的字母表用 αP 表示	αVMS={coin、drink}
7	具有字母表 A，但不执行 A 的任何事件的进程记作 $STOP_A$	$STOP_A$

定义 1 设 x 是一个事件，P 是一个进程，且 $x\in\alpha P$，则 $(x\rightarrow P)$ 表示以 x 为 P 前缀的进程。

例如：一个简单自动售货机在为两名顾客服务后损坏了可以用 $(coin\rightarrow(choc\rightarrow(coin\rightarrow(choc\rightarrow STOP_{\alpha VMS}))))$ 表示。

如果发生的事件是线性序列，则可省略括号。于是上例可写作：$VMS=(coin\rightarrow choc\rightarrow coin\rightarrow choc\rightarrow STOP_{\alpha VMS})$。

定义 2 前缀形式表示的进程称作被约束的进程。如果 $F(X)$ 是一个包含进程名 X 的约束表达式，A 是 X 的字母表，则要求等式 $X=F(X)$ 在 A 中有唯一解决方法，简写为 $\mu X:A\cdot F(X)$。

定义 3 设 x 和 y 分别是进程 P 和进程 Q 初始参与的两个不同事件且 $\alpha P=\alpha Q$，则 $(x\rightarrow P|y\rightarrow Q)$ 表示如果发生 x，则随后发生的是进程 P 的行为；如果发生的是 y，则随后发生的是进程 Q 的行为。

并发

两进程并发进行时，一般会相互作用、相互影响，而相互影响和作用的事件则要求是两进程所共有。

定义 4 如果 P 和 Q 是两进程且字母表相同，即 $\alpha P=\alpha Q$，则 $P\|Q$ 表示 P 和 Q 交叉并行发生，相互影响产生的行为系统。

下面给出两个进程的并发执行规则。

设 $a\in(\alpha P-\alpha Q)$，$b\in(\alpha Q-\alpha P)$，$\{c,d\}\in(\alpha Q\cap\alpha P)$。

1. $(c\rightarrow P)\|(c\rightarrow Q)=(c\rightarrow(P\|Q))$
2. $(c\rightarrow P)\|(d\rightarrow Q)=$STOP if $c\neq d$
3. $(a\rightarrow P)\|(c\rightarrow Q)=(a\rightarrow(P\|(c\rightarrow Q))$
4. $(c\rightarrow P)\|(b\rightarrow Q)=(b\rightarrow((c\rightarrow P)\|Q))$
5. $(a\rightarrow P)\|(b\rightarrow Q)=(a\rightarrow(P\|(b\rightarrow Q))|\ b\rightarrow((a\rightarrow P)\|Q))$

每一条规则均可用 LISP 语言进行描述。以规则 1 为例，假设 A、B 分别是 P、Q 的进程序列，用 $concurrent(P,A,B,Q)$ 表示 $P\|Q$，则 $concurrent(P,A,B,Q)=aux(P,Q)$。那么，

$aux(A,B)=\lambda x,\mu y\cdot$ if $ismember(x,car(A))$ and $ismember(y,car(B))$ and $x=y$ then $aux(cdr(car(A)),cdr(car(B)))$ else "BLEEP"

其中，

$ismember(x,A)=$if $B=$NIL then false

else if $x=car(A)$ then true else $ismember(x,cdr(car(A)))$。

在上述语言中，$car(A)$ 表示一个有限序列 A 的头部事件，$cdr(x)$ 表示某事件 x 的后续序列。

可见，在研究并发程序 CSP 模型时，LISP 在描述方面缺乏直观性和简便性。并发的进程是否会产生死锁，进程并发时哪些事件可并发执行，哪些事件不会并发执行，这些描述在上述规则中不易看出。

2.2 线性时态逻辑与计算树逻辑

线性时态逻辑

线性时态逻辑(Linear Temporal Logic，LTL)是应用最广泛的时态性质描述语言之一。在模型检测中常常首先利用 LTL 公式描述出系统待验证的性质，然后通过检测工具检验系统是否满足 LTL 公式。如果有不满足的行为，那么输出不符合性质的行为。

LTL 公式定义如下：

$$\Phi::=\top|\bot|p|(\neg\Phi)|(\Phi\wedge\Phi)|(\Phi\vee\Phi)|(\Phi\rightarrow\Phi)|(X\Phi)|(F\Phi)|(G\Phi)|(\Phi U\Phi)|(\Phi W\Phi)|(\Phi R\Phi)$$

式中，p 是取自某原子集 Atoms 的任意原子命题。连接词 X、F、G、U、R 和 W 称为时态连接词。X 意为“下一个状态”，F 意为“某未来状态”，G 意为“所有未来状态”。接下来的 3 个

连接词 U、R 和 W 分别称为“直到”、“释放”和“弱-直到”。线性时态逻辑的语义在此不详细叙述。

下面举例说明如何利用 LTL 公式描述系统性质：

• started 成立但 ready 不成立的状态是不可能到达的：$G\neg$(started$\wedge\neg$ready)。

• 不管发生什么情况，一个特定过程最终达到永久死锁的状态：F G deadlock。

然而有些性质是 LTL 所不能表达的。例如，从所有状态出发，都存在一条路径到达一个满足 restart 的状态。LTL 不能表达是因为它不能直接判定这些路径的存在性。

计算树逻辑

计算树逻辑(或简称 CTL)是一种分支时间逻辑，即其时间模型是一个树状结构，其中未来是不确定的。未来有不同的路径，其中的任何一个都可能是现实的“实际”路径。CTL 公式定义如下：

$\Phi ::= \top \mid \bot \mid p \mid (\neg\Phi) \mid (\Phi\wedge\Phi) \mid (\Phi\vee\Phi) \mid (\Phi\rightarrow\Phi) \mid$ AXΦ|EXΦ|AFΦ| EFΦ| AGΦ| EGΦ| A[ΦUΦ]|E[ΦUΦ]

需要注意的是，每个 CTL 公式中连接词和量词均成对出现。量词 A 表示“沿所有路径”，E 表示“沿至少一条路径”；X、F、G 和 U 的含义与 LTL 中连接词一样，分别为“下一状态”、“某个未来状态”、“所有未来状态”和“直到”。下面举例说明：

• 可以到达一个使 started 成立，但 ready 不成立的状态：EF (started$\wedge\neg$ready)。

• 不管发生什么，一个特定进程最终将永久死锁：AF (AG deadlock)。

• 一个进程总可以请求进入其关键段。该性质用 LTL 是不可表达的，用 CTL 公式可写成 AG($n_1\rightarrow$EX$t1$)。

2.3 回答集编程

ASP 是一种声明式程序设计框架，其基础是逻辑程序的回答集语义或稳定模型语义。设 A 是一个原子，一个文字可以是 A 也可以是 $\sim A$，其中 A 表示正文字，$\sim A$ 表示负文字。ASP 程序的规则 r 形如：

$$L_1\vee\cdots\vee L_k :- L_{k+1},\cdots,L_m,\ \text{not}\ L_{m+1},\cdots,\text{not}\ L_n$$

其中，$n\geqslant m\geqslant k\geqslant 0$。每一个 L_i 是一个文字，not 表示失败即否定(negation as failure)。将 $head(r)=\{L_1,\cdots,L_k\}$ 称为规则头部；$pos(r)=\{L_{k+1},\cdots,L_m\}$ 称为规则体的正文字，而 $neg(r)=\{L_{m+1},\cdots,L_n\}$ 称为规则体的负文字。如果一个规则 r 没有头部，称之为约束。如果 $pos(r)$ 和 $neg(r)$ 均为空，则称 r 为事实。

特别地，正规逻辑程序是一种特殊的逻辑程序，正规逻辑程序规则 r 形如 $L_0 :- L_1,\cdots,L_m,\text{not}\ L_{m+1},\cdots,\text{not}\ L_n$，其中 L_i $(0\leqslant i\leqslant n)$ 是原子。对任意规则 r，定义：$head(r)=\{L_0\}$，$pos(r)=\{L_1,\cdots,L_m\}$，$neg(r)=\{L_{m+1},\cdots,L_n\}$，$body(r)=pos(r)\cup neg(r)$，已知正规逻辑程序 P，$Lit(P)$ 表示 P 中出现的所有文字的集合。如果 P 中只有一条规则 r，则 $Lit(\{r\})=head(r)\cup pos(r)\cup neg(r)$。为了方便，$Lit(\{r\})$ 简记为 $Lit(r)$；规则 r 可表示为：$head(r)$:- $pos(r)$, not $neg(r)$ 或者 $head(r)$:-$body(r)$。

设 P 为不含约束规则的正规逻辑程序，$S\subseteq Lit(P)$，P 基于 S 的 Gelfond-Lifschitz 规约($GL(P,S)$)可通过如下步骤得到[11]：

(1) 对 P 中的任意规则 r，如果其规则体中含有公式 not a，且 $a\in S$，则删除此规则；

(2) 删除剩下规则中所有形如 not a 的公式。

已知一个逻辑程序 P，$S\subseteq Lit(P)$，$GL(P,S)$ 的最小模型为 X，如果 $X=S$，则 X 称为回答集。从回答集的定义可以看出，如果 X 为程序 P 的一个回答集，则 X 满足下面两个条件：

1)对于程序中任一规则 r，如 $pos(r)\subseteq X$ 且 $neg(r)\cap X=\emptyset$，则 $head(r)\in X$；

2)如果 X 中存在一对相反的文字(即不一致)，则 $X=Lit(P)$。

例如，已知正规逻辑程序 $P=\{g$:- b, not f, not h. f :- a, not h, not g. h :- e, not f, not g. a :- b, e. e :-. d. b.$\}$，该程序有 3 个回答集：$\{d,b,h,a,e\}$，$\{d,b,f,a,e\}$，$\{d,b,g,a,e\}$。

3 基于 ASP 的 CSP 并发系统验证

3.1 基于 ASP 的 CSP 规则的转换

CSP 顺序、循环进程的 ASP 表示

CSP 的顺序进程可视为一系列有序序列的集合。为了描述进程序列中动作的先后顺序，引入谓词 $pre(X,Y,P)$ 表示在进程 P 中动作 X 在动作 Y 之前发生。在有限顺序进程中，可以根据每一个行为推断出顺序进程 P 发生的第一个动作 X 和最后一个发生的动作 Y，分别用 $first(X,P)$ 和 $last(Y,P)$ 表示。推断方法可用下面的步骤来实现：

(1) 引入谓词 $ism(X,P)$ 表示 X 是 P 中的动作。当程序中给定 $pre(X,Y,P)$ 时，推断出动作 X 和动作 Y 在进程 P 中。用 ASP 规则表示为：

$ism(X,P)$:- $pre(X,Y,P)$. $ism(Y,P)$:- $pre(X,Y,P)$.

(2) 引入谓词 $not\text{-}first(X,P)$ 表示 X 不是 P 中第一个发生的动作。在 P 中存在动作 Z 在动作 X 之前发生，说明 X 不是 P 中第一个发生的动作。用 ASP 规则表示为：

$not\text{-}first(X,P)$:- $pre(Z,X,P)$.

如果 P 中没有在 X 之前发生的动作，则表明 X 在 P 中是第一个发生的动作。用 ASP 规则表示为：

$first(X,P)$:- not $not\text{-}first(X,P)$, $ism(X,P)$.

(3) 引入谓词 $not\text{-}last(X,P)$ 表示 X 不是 P 中最后一个发生的动作。在 P 中存在动作 Z 在动作 Y 之后发生，说明 Y 不是 P 中最后一个发生的动作。用 ASP 规则表示：

$not\text{-}last(Y,P)$:- $pre(Y,Z,P)$.

如果 P 中没有在 Y 之后发生动作，则表明 Y 在 P 中是最后一个发生的动作。用 ASP 规则表示为：

$last(Y,P)$:- not $not\text{-}last(Y,P)$, $ism(Y,P)$.

以一个自动售货机为例，如果一个人在投完 1 元硬币后，选择巧克力，然后又投了 2 元硬币选择了饼干。用 CSP 表示该顺序进程为：$VMS=in1p\rightarrow choc\rightarrow in2p\rightarrow toffee$，将其转化为 ASP 表示为：

$pre(in1p,choc,VMS)$. $pre(choc,in2p,VMS)$. pre

$(in2p, toffee, VMS)$.

按照 $first(X,P)$和 $last(X,P)$定义,有以下结论成立:

$first(in1p, VMS)$. $last(toffee, VMS)$.

CSP 的循环进程和顺序进程有所不同,循环进程可以看作是顺序进程的重复执行。设循环进程 Q 是顺序进程 P 的重复执行,则 Q 的执行可以看成在执行完 P 的最后一个动作后,继续执行 P 的第一个动作,其他动作执行顺序与 P 中执行顺序一样。为了用 ASP 规则表示循环进程 Q,只需在 P 的 ASP 表示中增加如下 ASP 规则:

$pre(Y,X,Q)$:- $first(X,P)$, $last(Y,P)$. (1)

$pre(X,Y,Q)$:- $pre(X,Y,P)$. (2)

同样以自动售货机为例,有一个售货机只提供巧克力,顾客可以通过投硬币取到巧克力。用 CSP 表示循环进程为:$VMS=coin\rightarrow choc\rightarrow VMS$.,该进程可以看作 $VMS=P\rightarrow P\rightarrow\cdots$,其中 P 是顺序进程,且 $\alpha P=\{coin, choc\}$。用 ASP 表示为:

$pre(coin, choc, P)$.

再加入上述规则(1)、(2),就可以得到 VMS 的 ASP 表示:

$pre(coin, choc, P)$. $pre(Y,X,VMS)$:- $first(X,P)$, $last(Y,P)$. $pre(X,Y,VMS)$:- $pre(X,Y,P)$.

由以上规则可得:$pre(coin, choc, VMS)$和 $pre(choc, coin, VMS)$。这显然与循环进程所要表达的性质一致。

CSP 并发执行规则的 ASP 表示

本节讨论如何将 2.1 节提到的 CSP 并发执行规则用 ASP 规则表示。本文关注的是两个进程并发时,在每一个步骤上并发执行的动作。为此,引入谓词 $aux(X,Y,J)$表示事件 X 和 Y 在第 J 步并发执行。

设 $a\in(\alpha P-\alpha Q)$, $b\in(\alpha Q-\alpha P)$, $\{c,d\}\in(\alpha Q\cap\alpha P)$。

规则 1 $(c\rightarrow P)\parallel(c\rightarrow Q)=(c\rightarrow(P\parallel Q))$。表示如果两个并发进程中当前待执行的动作均为 c,则二者并发执行的结果相当于首先执行动作 c,然后并发执行两个进程中剩余的动作序列(分别由 P 和 Q 表示)。该执行规则可转化为以下 ASP:

$aux(G,H,I)$:-$aux(X,Y,J)$, $ism(X,P)$, $ism(X,Q)$, $ism(Y,P)$, $ism(Y,Q)$, $pre(X,G,P)$, $pre(Y,H,Q)$, $\#succ(J,I)$, $X=Y$, $P!=Q$. (i)

式中,$\#succ(J,I)$是回答集求解器自带的算术谓词,表示 $I=J+1$。该 ASP 规则表示:X 和 Y 分别都是 P 和 Q 中的动作,G、H 分别为 X 和 Y 的后继动作,且 X 和 Y 相同。那么,在 X 和 Y 在第 J 步并发执行后,下一步 I 中并发执行的动作为 G 和 H。

规则 2 $(c\rightarrow P)\parallel(d\rightarrow Q)=$STOP if $c\neq d$。表示如果两个并发进程中当前执行的动作为 c 和 d,且 c 和 d 不相同,则这两个进程将停止并发执行。用 ASP 规则表示为:

:- $aux(X,Y,I)$, $ism(X,P)$, $ism(X,Q)$, $ism(Y,P)$, $ism(Y,Q)$, $X!=Y$, $P!=Q$. (ii)

该 ASP 规则表示:X 和 Y 同为进程 P 和 Q 中的动作,且 X 和 Y 不相同时,二者无论何时都不能并发执行,即 $aux(X,Y,I)$与以上条件不能同时成立。

规则 3 $(a\rightarrow P)\parallel(c\rightarrow Q)=(a\rightarrow(P\parallel(c\rightarrow Q)))$。表示如果两个并发进程 $a\rightarrow P$ 和 $c\rightarrow Q$ 中当前执行的动作分别为 a 和 c,其中 a 是进程 P 独有的动作,c 是二进程的共有动作,则二者并发的结果相当于首先执行动作 a,然后进程 $a\rightarrow P$ 的剩余动作序列(P)和进程 $c\rightarrow Q$ 动作序列并发执行。用 ASP 规则表示为:

$aux(G,Y,I)$:-$aux(X,Y,J)$, $ism(X,P)$, not $ism(X,Q)$, $ism(Y,P)$, $ism(Y,Q)$, $pre(X,G,P)$, $\#succ(J,I)$, $P!=Q$. (iii)

该 ASP 规则表示:X 只是 P 中的动作,而 Y 既是 P 的中又是 Q 中的动作,G 是 X 的后继动作。那么,在 X 和 Y 并发执行后,由 G 和 Y 继续并发执行。

规则 4 $(c\rightarrow P)\parallel(b\rightarrow Q)=(b\rightarrow((c\rightarrow P)\parallel Q))$。表示如果两个并发进程 $c\rightarrow P$ 和 $b\rightarrow Q$ 中当前执行的动作分别为 c 和 b,其中 b 是进程 Q 独有的动作,c 是二进程的共有动作,则二者并发的结果相当于首先执行动作 b,然后进程 $b\rightarrow Q$ 的剩余动作序列(Q)和进程 $c\rightarrow P$ 动作序列并发执行。用 ASP 规则表示为:

$aux(X,H,I)$:-$aux(X,Y,J)$, $ism(X,P)$, $ism(X,Q)$, not $ism(Y,P)$, $ism(Y,Q)$, $pre(Y,H,Q)$, $\#succ(J,I)$, $P!=Q$. (iv)

该 ASP 规则表示:Y 只是 Q 中的动作,而 X 既是 P 的又是 Q 的动作,H 是 Y 的后继动作。那么,当 X 和 Y 并发执行后,由 X 和 H 继续并发执行。

规则 5 $(a\rightarrow P)\parallel(b\rightarrow Q)=(a\rightarrow(P\parallel(b\rightarrow Q))\mid b\rightarrow((a\rightarrow P)\parallel Q))$。表示如果两个并发进程 $a\rightarrow P$ 和 $b\rightarrow Q$ 中当前执行的动作为 a 和 b,其中 a 和 b 分别为进程 P 和 Q 所独有,则二者并发的结果相当于:首先执行动作 a,然后并发执行进程 $a\rightarrow P$ 的剩余动作序列(P)和进程 $b\rightarrow Q$。或者,首先执行动作 b,然后并发执行进程 $b\rightarrow Q$ 的剩余动作序列(Q)和进程 $a\rightarrow P$。用 ASP 规则表示为:

$aux(G,Y,I)$:-$aux(X,Y,J)$, $ism(X,P)$, not $ism(X,Q)$, $ism(Y,Q)$, not $ism(Y,P)$, $pre(X,G,P)$, $pre(Y,H,Q)$, $\#succ(J,I)$. (v)

$aux(X,H,I)$:-$aux(X,Y,J)$, $ism(X,P)$, not $ism(X,Q)$, $ism(Y,Q)$, not $ism(Y,P)$, $pre(X,G,P)$, $pre(Y,H,Q)$, $\#succ(J,I)$. (vi)

以上 ASP 规则分别表示:1)X 只在 P 中发生,Y 只在 Q 中发生,G、H 分别是 X、Y 的后继动作。那么,当 X 和 Y 并发执行后,由 G 和 Y 继续并发执行。2)X 只在 P 中发生,Y 只在 Q 中发生,G、H 分别是 X、Y 的后继动作。那么,当 X 和 Y 并发执行后,由 X 和 H 继续并发执行。

下面以两个进程并发为例来说明以上规则的应用。

例 设有两个进程 $P1$ 和 $P2$:$\alpha P1=\{1,3\}$, $\alpha P2=\{2,3\}$. $P1=1\rightarrow3\rightarrow P1$. $P2=3\rightarrow2\rightarrow P2$.

首先,$P1$ 可转化为 $Q1$:

{$first(1,p1)$. $last(3,p1)$. $pre(1,3,p1)$.

$pre(Y,X,P)$:- $first(X,P)$, $last(Y,P)$.

$ism(X,P)$:- $pre(X,Y,P)$. $ism(Y,P)$:- $pre(X,Y,P)$.};

$P2$ 可转化为 $Q2$：

{*first*(3,*p*2). *last*(2,*p*2). *pre*(3,2,*p*2).

pre(*Y*,*X*,*P*):- *first*(*X*,*P*),*last*(*Y*,*P*).

ism(*X*,*P*):- *pre*(*X*,*Y*,*P*). *ism*(*Y*,*P*):- *pre*(*X*,*Y*,*P*).}.

将 CSP 并发执行规则的 ASP 表示记作 Q。然后，求 $Q1 \cup Q2 \cup Q$ 的回答集。由此可得 $P1$ 和 $P2$ 并发过程中的所有并发事件。这是两个无限循环执行的进程间的并发，为了能够了解有限次数的并发情况，加入 # maxint=N 规定并发的次数。以 5 次为例，ASP 程序如下：

```
#maxint=5. aux(1,3,0).
%The ASP rules from P1、P2.
first(1,p1). last(3,p1). pre(1,3,p1). first(3,p2). last(2,p2). pre(3,2,p2).
pre(Y,X,P):- first(X,P),last(Y,P).  ism(X,P):- pre(X,Y,P).
ism(Y,P):- pre(X,Y,P).
%The ASP rules from CSP Concurrent execution rule.
aux(G,H,I):- aux(X,Y,J),pre(X,G,P),pre(Y,H,Q),X=Y,P! =Q,#succ(J,I).
:- aux(X,Y,I),ism(X,P),ism(X,Q),ism(Y,P),ism(Y,Q),X! =Y,P! =Q.
aux(G,Y,I):- aux(X,Y,J),pre(X,G,P),not ism(X,Q),ism(Y,P),ism(Y,Q),P! =Q,#succ(J,I).
aux(X,H,I):- aux(X,Y,J),ism(X,P),ism(X,Q),not ism(Y,P),pre(Y,H,Q),P! =Q,#succ(J,I).
aux(G,Y,I):- aux(X,Y,J),not ism(X,Q),not ism(Y,P),pre(X,G,P),pre(Y,H,Q),P! =Q,#succ(J,I).
aux(X,H,I):- aux(X,Y,J),not ism(X,Q),not ism(Y,P),pre(X,G,P),pre(Y,H,Q),P! =Q,#succ(J,I).
```

利用 DLV 对 ASP 程序求解，求得的回答集为：

{*ism*(1,*p*1),*ism*(3,*p*1),*ism*(2,*p*2),*ism*(3,*p*2),*first*(1,*p*1),*last*(3,*p*1),*first*(3,*p*2),*last*(2,*p*2),*pre*(1,3,*p*1),*pre*(3,2,*p*2),*pre*(3,1,*p*1),*pre*(2,3,*p*2),*aux*(1,3,0),*aux*(1,3,3),*aux*(3,3,1),*aux*(3,1,3),*aux*(1,2,2),*aux*(2,1,2),*aux*(2,3,3),*aux*(3,2,3),*aux*(3,3,4),*aux*(1,2,5),*aux*(2,1,5)}

由此可知 $P1$ 和 $P2$ 并发时前 5 次并发执行的情况。更直观地，原始的 CSP 并发程序推导执行过程可用图 1 表示；而基于 ASP 的 CSP 并发程序推导执行过程可用图 2 表示。

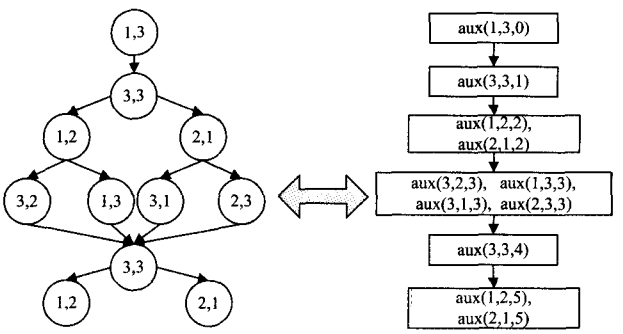

图 1

图 2

图 1 中，圆圈内表示并发执行的事件。箭头指向的是后继并发事件。图中的任意一条从初始并发事件(1,3)到其它并发事件的路径均对应一次系统的并发执行。下文中称该路径为并发执行路径。图 2 中，方框内表示并发事件的集合。箭头指向的是后继并发事件的集合。由 ASP 程序可以得出在有限步骤内事件并发状态的集合。下面给出确定这些并发状态间的联系方法。

首先，将第 i 步产生的所有并发事件状态用集合 S_i 表示，在相邻步骤并发事件集合 S_i、S_{i+1} 中：

1)如果 $aux(X,X,I) \in S_i$，$aux(Y,Z,I+1) \in S_{i+1}$，则存在一条 X 和 X 并发的状态到达 Y 和 Z 并发的状态路径。

2)如果 $aux(X,Y,I) \in S_i$，$aux(Z,Z,I+1) \in S_{i+1}$，则存在一条 X 和 Y 并发的状态到达 Z 和 Z 并发的状态路径。

3)如果 $aux(X,Y,I) \in S_i$，$aux(X,Z,I+1) \in S_{i+1}$，且 X、Y、Z 互不相同，则存在一由 X 和 Y 并发的状态到达 X 和 Z 并发的状态路径；

4)如果 $aux(X,Y,I) \in S_i$，$aux(Z,Y,I+1) \in S_{i+1}$，且 X、Y、Z 互不相同，则存在一条 X 和 Y 并发的状态到达 Z 和 Y 并发的状态路径；

5)在 S_i 和 S_{i+1} 中，不满足情况(1)－(4)的两个并发状态间不存在路径。

根据以上方法，可将求得的并发程序状态作为 ASP 程序的已知条件 $C1$，然后利用下述路径判定规则 L：

{:- *aux*(*X*,*Y*,*I*),*aux*(*M*,*N*,*J*),*J*=*I*+1,*X*! =*M*,*Y*! =*N*,*X*! =*Y*.}

将不在一条路径上的状态排除。如在图 2 中由 $C_1 \cup L$ 可求得一条路径：

{*aux*(1,3,0),*aux*(3,3,1),*aux*(1,2,2),*aux*(3,2,3),*aux*(3,3,4),*aux*(1,2,5)}

可见，CSP 与 ASP 模型的重要区别在于，CSP 由当前并发事件能推导出后继并发事件，但每次执行中只能考虑一个分支；而 ASP 能同时计算出所有的后继并发事件，并由此确定所有分支。该特征对并发系统的性质验证具有重要意义，即 ASP 可支撑多个性质的同时验证，而其他模型检测工具则有一定困难。

3.2 LTL\CTL 公式到 ASP 规则的转换

在验证 CSP 并发系统时，待验证性质可用 LTL\CTL 公式描述。下面讨论 LTL/CLT 公式的转换。LTL 公式对应的 ASP 规则如下[17]。

首先，定义原子公式：$AP=\{p(a,b,i) \mid a,b \in$ 共同事件$\}$，对于所有 $0 \leqslant i \leqslant N$($N$ 为规定的并发次数)。

公式 1 $p(a,b,i)$，对所有 $p(a,b,i) \in AP$。表示 $p(a,b,i)$是一个为真的原子命题。用 ASP 规则表示为：$f(i)$:- $p(a,b,i)$。

公式 2 $\neg p(a,b,i)$，对所有 $p(a,b,i) \in AP$。表示 $p(a,b,i)$是一个为假的原子命题。用 ASP 规则表示为：$f(i)$:- not $p(a,b,i)$。

公式 3 $f \equiv f1 \vee f2$。表示在某一状态 $f1$ 为真，或者 $f2$ 为真，则性质 f 为真。用 ASP 规则表示为：$f(i)$:- $f1(i)$. $f(i)$:- $f2(i)$.(其中，$f1(i)$、$f2(i)$是用 ASP 规则表示的谓词，下同)。

公式 4 $f \equiv f1 \wedge f2$。表示在某一状态 $f1$ 为真且 $f2$ 也为真，则性质 f 为真。用 ASP 规则表示为：$f(i)$:- $f1(i)$,

$f2(i)$。

公式5　$f\equiv Xf1$。表示下一个状态 $f1$ 为真，则性质 f 为真用 ASP 规则表示为：$f(i)$:- $f1(i+1)$。

公式6　$f\equiv Ff1$。表示在未来某个状态 $f1$ 为真，则性质 f 为真。用 ASP 规则表示为：$f(i)$:- $j\geqslant i, f1(j)$.

公式7　$f\equiv Gf1$。表示在某状态之后 $f1$ 一直为真，则性质 f 为真。用 ASP 规则表示为：*not_*$f1(i)$:- $j\geqslant i, j\leqslant n$, not $f1(j)$.　$f(i)$:- not *not_*$f1(i)$.

公式8　$f\equiv f1Uf2$。表示在某个未来状态 $f2$ 为真，并且在 $f2$ 为真之前 $f1$ 一直为真，则 f 为真。用 ASP 规则表示为：*not_*$f1(i)$:- $j\geqslant i, j\leqslant k-1, f2(k)$, not $f1(j)$.　$f(i)$:- not *not_*$f1(i)$.

公式9　$f\equiv f1\rightarrow f2$。表示在某一状态 $f1$ 为真使得 $f2$ 为真，则性质 f 为真。用 ASP 规则表示为：$f2(i)$:- $f1(i)$.

公式10　$f\equiv f1R\ f2$。表示在某状态之后 $f2$ 必须为真，直到 $f1$ 为真的时候，或者 $f2$ 一直为真。用 ASP 规则表示为：

*false_*$f1(i)$:- $j\leqslant n, j\geqslant i$, not $f1(j)$.

*not_*$f2(i)$:- $j\leqslant n, j\geqslant i$, *false_*$f1(i)$, not $f2(j)$.

*not_*$f2(i)$:- $j\leqslant k, j\geqslant i, f1(k)$, not $f2(j)$.

$f(i)$:- not *not_*$f2(i)$.

下面给出 CTL 公式转换为 ASP 规则的方法。

首先，定义原子公式：$AP=\{p(a,b,i)\mid a,b\in$ 共同事件$\}$，对于所有 $0\leqslant i\leqslant N$（N 为规定的并发次数）。

在 CTL 公式中，符号 X,F,G 和 U 在前面没有 A 或 E 的情况下不能单独存在，每一个 A 或 E 必须伴随着 X,F,G 或 U 之一出现。因此 CTL 公式1－公式4转换后的 ASP 规则与 LTL 公式1－公式4转换后的 ASP 规则相同。主要区别在于含有 X,F,G,U 的公式：

公式5　$f\equiv AXf1$。表示系统所有路径中，下一个状态 $f1$ 为真，则性质 f 为真。因此，所有路径加入 ASP 规则：$f(i)$:- $f1(i+1)$。如求得所有回答集中含有 $f(i)$，则表明性质 f 为真。

公式6　$f\equiv EXf1$。表示系统存在一条路径，使得下一个状态 $f1$ 为真，则性质 f 为真。换言之，系统中不是所有的路径，使得下一状态 $f1$ 为假，性质 f 为真。因此，所有路径加入 ASP 规则：*non_*$f(i)$:- not $f1(i+1)$ $f(i)$:- not *non_*$f(i)$。如求得某回答集中含有 $f(i)$，则表明性质 f 为真。

公式7　$f\equiv AFf1$。表示系统所有路径的未来某个状态 $f1$ 为真，则性质 f 为真。因此，所有路径加入 ASP 规则：$f(i)$:- $j\geqslant i, f1(j)$。如求得所有回答集中含有 $f(i)$，则表明性质 f 为真。

公式8　$f\equiv EFf1$。表示系统存在一条路径使得未来某个状态 $f1$ 为真，则性质 f 为真。换言之，系统中不是所有路径，使得未来某个状态 $f1$ 为假，则性质 f 为真。因此，所有路径加入 ASP 规则：$f(i)$:- $j>i, f1(j)$ *non_*$f(i)$:- not $f(i)$。如求得某回答集中含有 $f(i)$，则表明性质 f 为真。

公式9　$f\equiv AGf1$。表示系统所有路径上某状态之后 $f1$ 一直为真，则性质 f 为真。因此，所有路径加入 ASP 规则：*not_*$f1(i)$:- $j\geqslant i, j\leqslant n$, not $f1(j)$.　$f(i)$:- not *not_*$f1(i)$。如求得所有回答集中含有 $f(i)$，则表明性质 f 为真。

公式10　$f\equiv EGf1$。表示系统存在一条路径上某状态之后 $f1$ 一直为真，则性质 f 为真。因此，所有路径需加入 ASP 规则：*not_*$f1(i)$:- $j\geqslant i, j\leqslant n$, not $f1(j)$. $f(i)$:- not *not_*$f1(i)$。如求得某回答集中含有 $f(i)$，则表明性质 f 为真。

公式11　$f\equiv Af1Uf2$。表示系统所有路径中某个未来状态 $f2$ 为真，并且在 $f2$ 为真之前 $f1$ 一直为真，则性质 f 为真。因此，所有路径加入 ASP 规则：*not_*$f1(i)$:- $j\geqslant i, j\leqslant k-1, f2(k)$, not $f1(j)$.　$f(i)$:- not *not_*$f1(i)$. 若求得所有回答集中含有 $f(i)$，则表明性质 f 为真。

公式12　$f\equiv Ef1Uf2$。表示系统某条路径中存在某个未来状态 $f2$ 为真，并且在 $f2$ 为真之前 $f1$ 一直为真，则性质 f 为真。因此，所有路径加入 ASP 规则：*not_*$f1(i)$:- $j\geqslant i, j\leqslant k-1, f2(k)$, not $f1(j)$.　$f(i)$:- not *not_*$f1(i)$. 若求得某个回答集中含有 $f(i)$，则表明性质 f 为真。

3.3　验证方法流程

至此，可给出基于 ASP 对 CSP 并发程序进行验证的基本流程：

1)将 CSP 程序转化为 ASP 程序，记作 C_0。

2)将 CSP 并发进程规则转换为 ASP 规则，记作 Π。

3)将 $C_0\cup\Pi$ 的回答集中相同步数下并发事件状态转换成析取规则 C_1。

4)将 LTL/CTL 公式描述的待验证并发程序性质用 ASP 规则表示，记作 P。

5)计算 $C_1\cup P\cup F$ 的回答集，F 是路径判断规则。根据求得的回答集结果判定所验证性质是否满足。

按照以上步骤就可对 CSP 并发系统进行验证，下面以哲学家进餐问题为例来阐述该方法的运用。

4　哲学家进餐问题

哲学家进餐问题是并发程序研究中的经典问题。几个哲学家在思考问题，当他们感觉饿的时候，会到桌子旁准备进餐。假设有5位哲学家命名为 PH_1、PH_2、PH_3、PH_4、PH_5。他们准备进餐时，要先坐下来，然后先拿起左边餐叉，再拿起右边的餐叉，当双手分别有餐叉时，就可以进餐了。进完餐之后，离开座位继续去思考问题。哲学家进餐座位示意图如图3所示。

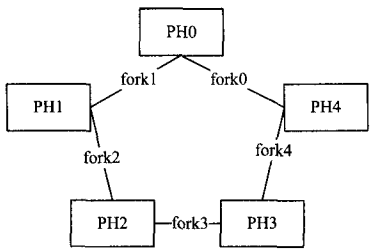

图3　哲学家进餐座位示意图

通常，哲学家进餐问题的求解方案需要满足如下性质：1. 一把叉子不能同时被两个哲学家拿起。2. 一个哲学家只要处于等待用餐状态，最终可进入用餐状态。3. 存在哲学家2坐下后始终等待就餐的状态。每一个哲学家的行为可用 CSP 规则描述为 $PH_i=siti\rightarrow pick\_forki\rightarrow pick\_forki\oplus 1\rightarrow eati$

$\rightarrow down\_fork i \rightarrow down\_fork i \oplus 1 \rightarrow up i \rightarrow Phi$，其中“⊕”是模5运算。以并发5步为例：

首先，将并发程序CSP模型转换为ASP：

以邻座(PH_1，PH_2)为例。给出ASP描述的CSP规则C_0：

{$first(sit1, ph1)$，$pre(sit1, pick\_fork1, ph1)$，$pre(pick\_fork1, pick\_fork2, ph1)$，$pre(pick\_fork2, eat1, ph1)$，$pre(eat1, down\_fork1, ph1)$，$pre(down\_fork1, down\_fork2, ph1)$，$pre(down\_fork2, up1, ph1)$，$last(up1, ph1)$，$first(sit2, ph2)$，$pre(sit2, pick\_fork2, ph2)$，$pre(pick\_fork2, pick\_fork3, ph2)$，$pre(pick\_fork3, eat2, ph2)$，$pre(eat2, down\_fork2, ph2)$，$pre(down\_fork2, down\_fork3, ph2)$，$pre(down\_fork3, up2, ph2)$，$last(up2, ph2)$，$aux(sit1, sit2, 0)$.}

接着，将$C_0 \cup \Pi$的回答集(Π是CSP并发进行规则转换所得ASP程序)中相同步数下并发事件状态转换成ASP的析取规则C_1。

其次，将LTL公式转化为ASP规则：

性质1 $G\neg(pick\_fork i \wedge pick\_fork j \wedge i=j)$，转化为ASP规则集$P1$：

$P1=\{f(I) :\!\!-\ aux(X,Y,I), X=Y, X=pick\_fork2.\quad non\_f1 :\!\!-\ f(I), I>0, I\leqslant 5.\quad f1 :\!\!-\ not\ non\_f1.\quad f1?\}$

性质2 $G(sit1 \rightarrow F(eat1))$，转化为ASP规则集$P2$：

$P2=\{f2 :\!\!-\ aux(sit1,X,I), aux(eat1,Y,J), J>I.$

$f2 :\!\!-\ aux(sit1,X,I), aux(Y,eat1,J), J>I.\quad f2?\}$

性质3 用CTL对性质描述为：$EG(sit2 \rightarrow AF(\neg eat2))$。可转化为ASP规则集$P3$：

$P3=\{non\_f3 :\!\!-\ aux(sit2,X,I), aux(eat2,Y,J), J>I.$

$non\_f3 :\!\!-\ aux(X,sit2,I), aux(Y,eat2,J), J>I.$

$non\_f3 :\!\!-\ aux(sit2,X,I), aux(Y,eat2,J), J>I.$

$non\_f3 :\!\!-\ aux(X,sit2,I), aux(eat2,Y,J), J>I.$

$f3 :\!\!-$ not $non\_f3.\quad f3?\}$

接下来就可结合路径判断规则L：

{$:\!\!-\ aux(X,Y,I), aux(M,N,J), J=I+1, X!=M, Y!=N, X!=Y.$}

验证以上性质。

利用ASP的谨慎推理(Cautious reasoning)可对性质1和性质2同时进行验证，在$C_1 \cup P1 \cup P2 \cup L$上运行ASP回答集求解器，结果为：

$f2$ is cautiously true. $f1$ is cautiously false, evidenced by {$aux(sit1, sit2, 0)$，$aux(pick\_fork1, sit2, 1)$，$aux(pick\_fork2, sit2, 2)$，$aux(pick\_fork2, pick\_fork2, 3)$，$aux(pick\_fork3, eat1, 4)$，$aux(pick\_fork3, down\_fork1, 5)$}。

该结果表示，性质2满足而性质1不满足，且给出了不满足的原因。

运用ASP的果敢推理(Brave reasoning)可对性质3进行验证，在$C_1 \cup P3 \cup L$上运行ASP回答集求解器，结果为：

$f3$ is bravely true, evidenced by {$aux(sit1, sit2, 0)$，$aux(pick\_fork1, sit2, 1)$，$aux(pick\_fork2, sit2, 2)$，$aux(pick\_fork2, pick\_fork2, 3)$，$aux(pick\_fork3, eat1, 4)$，$aux(pick\_fork3, down\_fork1, 5)$，$f3$}。

该回答集表示性质3满足并给出了满足性质3的一条路径。

谨慎推理和果敢推理是稳定模型语义中两种比较有代表性的推理。谨慎推理表示基原子A在程序P中为真，当且仅当A在P的所有稳定模型中都为真；而果敢推理则表示在程序P中一个基原子A是真的当且仅当A在P的某些稳定模型中为真。因此，在系统中对LTL公式描述的性质或者对CTL只含有“A”的公式描述的性质进行验证时，运用DLV工具的谨慎推理机制；而对CTL只含有“E”的公式描述性质进行验证时，运用DLV工具的果敢推理机制。对于运用同一种推理机制的多条性质，验证可同时进行。例如，性质1和性质2是可在谨慎推理机制下同时验证的。由上述例子可见，运用DLV工具不但可验证系统中的性质是否满足，而且对于不满足的性质，也可给出相应的回答集来解释不满足的原因。

5 实验

在以上讨论的基础上，利用VC开发了基于ASP的CSP并发系统验证工具。为了检验其验证效率，以CSP并发进程转化所得的ASP程序和ASP表示的待验证系统性质为输入，对不同ASP回答集求解器验证程序性质的效率进行了比较。表1给出验证上述哲学家进餐问题的同时验证性质1和性质2所需的执行时间，其中N是验证时选择的并发执行步数。实验平台配置如下：Windows XP，Intel(R) Core(TM) 2 Duo，CPU E7400 2.80GHz，内存3.25GB。

表1 基于ASP验证CSP并发进程的效率比较

并发步数N	DLV (s)	CMODELS (s)	SMODELS (s)	性质1	性质2
2	0.06	0.07	0.08	满足	不满足
5	0.25	0.26	0.28	不满足	满足
8	0.37	0.39	0.43	不满足	满足
12	0.52	0.55	0.53	不满足	满足
15	0.73	0.72	0.70	不满足	满足
18	0.98	0.99	0.93	不满足	满足
20	1.15	1.09	1.12	不满足	满足
28	1.44	1.46	1.43	不满足	满足
33	2.11	2.13	1.98	不满足	满足
35	2.29	2.26	2.17	不满足	满足
40	2.78	2.76	2.69	不满足	满足
42	2.91	2.93	2.89	不满足	满足

由表1可见，随着并发步数的增加，3种求解器均能进行有效的性质验证。相对而言，DLV在较小并发步数时执行时间最短；而SMODELS在并发步数较大时运行时间最短；CMODELS验证效率与DLV相当。以上实验也表明了，基于SAT的ASP程序回答集求解器SMODELS适用于较大规模问题的求解。验证工具的开发和初步试验结果表明，基于ASP的并发系统验证工具开发效率高，利用ASP求解器对并发程序验证的执行效率较高，并实现了在验证工具的一次运行中验证多条系统性质的目标。

结束语 本文给出了一种基于ASP的并发CSP系统验证统一框架，及将CSP描述的并发程序以及CSP并发执行规则转化为ASP程序的技术和把CTL公式转化为ASP程序的方法，并在此基础上验证了CSP系统性质技术。开发实践和实验结果表明，基于ASP的CSP并发系统验证技术易于实现，在保持较高验证效率的同时能够支持在验证软件的一

次执行中验证多条 LTL/CTL 公式。本文研究为开发更为高效的并发系统验证工具提供了新思路。下一步除了不断完善本文提出的验证框架之外，还将进一步通过更多的实验对该方法的效率进行测试和对比，研究在待验证的性质不满足时给出反例，及基于所给反例进行系统诊断的技术。

参 考 文 献

[1] Hoare C A R. Communicating Sequential Processes [M]. http://www.usingcsp.com/cspbooks,2004

[2] Garavel H, Lang F. NTIF: A General Symbolic Model for Communicating Sequential Processes with Data [C]// Proc. FORTE'02. 2002:276-291

[3] Clarke E M, Talupur M, Veith H. Proving Ptolemy Right, The Environment Abstraction Framework for Model Checking Concurrent Systems [C]// Proc. TACAS. 2008:33-47

[4] Chaki S, Clarke E M, Ouaknine J, et al. Concurrent software verification with states, events, and deadlocks[J]. Formal Aspects of Computing, 2005, 17(4):461-483

[5] Dubrovin J. Efficient Symbolic Model Checking of Concurrent Systems [D]. Aalto University School of Science Department of Information and Computer Science, Doctoral Dissertation, 2011

[6] 张兆庆，蒋昌俊，等. PVM 并行程序验证系统的原理与实现[J]. 计算机学报，1999，22(4)：409-414

[7] Gelfond M, Lifschitz V. The stable model semantics for logic programming[C]// Proc. ICLP'1988. 1988:1070-1080

[8] Baral C. Knowledge Representation, Reasoning, and Declarative Problem Solving [M]. Cambridge Press, 2003:5-64

[9] Zhao Ling-zhong, Qian Jun-yan, Chang Liang, et al. Using ASP for Knowledge Management with User Authorization [J]. Data & Knowledge Engineering, 2010, 69(8):737-762

[10] Gavanelli M, Rossi F. Constraint logic programming [C]// LNCS. 2010, 6125:64-86

[11] De Angelis E, Pettorossi A, Proietti M. Synthesizing Concurrent Programs using Answer Set Programming [C]// Proc. CS&P 2011. 2011:85-98

[12] Heljanko K, Niemelä I. Answer set programming and bounded model checking [C]// Proc. AAAI Spring 2001 Symposium on Answer Set Programming. 2001:90-96

(上接第 93 页)

5 性能分析与讨论

5.1 隐蔽性分析

采用基于增量链接的隐藏算法，将隐秘信息嵌入到函数指令代码之间的填充字节中、使得隐藏的信息与原程序中函数的指令代码融合在一起、其隐蔽性高。同时，在此方法中，不需要修改 PE 文件结构中的其它数据，不会增加文件长度，也不会影响程序性能，大大提高了隐蔽性。

5.2 嵌入容量分析

对 Windows 窗口应用程序，编译器将自动添加众多的初始化函数和窗口框架函数，函数指令代码之间的填充空间多，隐藏容量大。隐藏容量与应用程序功能有关，一般来说，程序功能越复杂，程序中的函数(包括用户自定函数和静态链接库函数)越多，填充空间越大，隐藏容量也就越大。表 1 为基于增量链接的信息隐藏方案隐藏容量分析。

表 1 基于增量链接的信息隐藏方案隐藏容量分析

文件名	文件大小 (bytes)	ILT 表的大小(bytes)	文件中函数个数	填充字节 (bytes)	嵌入容量 (bytes)
helloworld. exe	88576	575	115	2190	1960
hideinfo. exe	245248	4540	908	19170	17354
inforemix. exe	254026	315	63	2651	2525
movecode. exe	286797	895	179	11866	11508
iocnHide. exe	135263	310	62	6467	6343
import. exe	110686	185	37	1768	1694

5.3 鲁棒性

PE 文件结构复杂，代码节数据组织更为复杂，并且 PE 文件与图像、文本文件不一样，即使是一个字节的修改都可能导致程序不能运行。此方法隐藏信息后，文件长度不会增加，程序运行正常，性能也不受影响。

结束语 现有 PE 文件隐藏算法存在隐藏信息过于集中、没有与程序指令代码结合在一起、安全性不高等问题。本文利用编译器增量链接的特性，将隐秘信息隐藏在函数代码之间的填充字节中，使得隐藏的信息与程序指令代码紧密结合在一起，极大地提高了抗攻击性。由于对 Windows 窗口应用程序，编译器将自动添加众多的初始化函数和窗口框架函数，因此函数指令代码之间的填充空间多，隐藏容量大。隐藏信息后的 PE 文件的长度不会增加，程序性能不受影响，隐蔽性好。

参 考 文 献

[1] Appel A W, Ginsburg M. 现代编译原理：C 语言描述[M]. 赵克佳，黄春，沈志宇，译. 北京：人民邮电出版社，2006

[2] Aho A V, Lam M S, Sethi R, et al. Compilers: Principles, Techniques, and Tools (2nd Edition)[M]. Addison Wesley, 2008

[3] 俞甲子，石凡，潘爱民. 程序员的自我修养—链接、装载与库[M]. 北京：电子工业出版社，2009

[4] Levine J R. Linkers and Loaders[M]. Morgan Kaufmann, 2005

[5] 关于 Visual C++增量链接以及. textbss[OL]. http://www.cnblogs.com/Dahaka/archive/2011/08/01/2124256.html, accessed September, 2012

[6] Pietrek M. Peering Inside the PE: A Tour of the Win32 Portable Executable File Format[J]. Microsoft Systems Journal, 1994, 9(3):15-34

[7] Richter J, Nasarre C. Windows 核心编程[M]. 葛子昂，周靖，廖敏，译. 北京：清华大学出版社，2008:509-522

[8] Pietrek M. Windows 95 System Programming Secrets[M]. 候俊杰，译. 台湾：旗标出版社，1997:559-623

[9] 陈凯明，刘宗田. 反编译研究现状及进展[J]. 计算机科学，2001，28(5)：113-115

[10] Cifuentes C. Reverse Compilation Techniques [D]. School of Computing Science, Queensland University of Technology, July 1994