中国科学院大学
University of Chinese Academy of Sciences

博士学位论文

中国科学院大学学位论文 LaTeX 模板

作者姓名：刘坚

指导教师：蒋颖 研究员

中国科学院软件研究所

学位类别：工学博士

学科专业：计算机软件与理论

培养单位：中国科学院软件研究所

2018 年 02 月

LaTeX Thesis Template
of
The University of Chinese Academy of Sciences

By

Jian Liu

A thesis submitted to
University of Chinese Academy of Sciences
in partial fulfillment of the requirement
for the degree of
Doctorin Computer Software and Theory

Institute of Software, Chinese Academy of Sciences

February, 2018

中国科学院大学
研究生学位论文原创性声明

本人郑重声明：所呈交的学位论文是本人在导师的指导下独立进行研究工作所取得的成果。尽我所知，除文中已经注明引用的内容外，本论文不包含任何其他个人或集体已经发表或撰写过的研究成果。对论文所涉及的研究工作做出贡献的其他个人和集体，均已在文中以明确方式标明或致谢。

作者签名：　　日　　期：

中国科学院大学
学位论文授权使用声明

本人完全了解并同意遵守中国科学院有关保存和使用学位论文的规定，即中国科学院有权保留送交学位论文的副本，允许该论文被查阅，可以按照学术研究公开原则和保护知识产权的原则公布该论文的全部或部分内容，可以采用影印、缩印或其他复制手段保存、汇编本学位论文。

涉密及延迟公开的学位论文在解密或延迟期后适用本声明。

作者签名：　　导师签名：　　日　　期：

摘 要

本文是中国科学院大学学位论文模板 ucasthesis 的使用说明文档。主要内容为介绍 LaTeX 文档类 ucasthesis 的用法，以及如何使用 LaTeX 快速高效地撰写学位论文。

关键词：中国科学院大学，学位论文，LaTeX 模板

Abstract

This paper is a help documentation for the LaTeX class ucasthesis, which is a thesis template for the University of Chinese Academy of Sciences. The main content is about how to use the ucasthesis, as well as how to write thesis efficiently by using LaTeX.

Keywords: University of Chinese Academy of Sciences (UCAS), Thesis, LaTeX Template

目 录

插 图

表 格

符号列表

Characters

Symbol	Description	Unit
R	the gas constant	$\mathrm{m}^2 \cdot \mathrm{s}^{-2} \cdot \mathrm{K}^{-1}$
C_v	specific heat capacity at constant volume	$\mathrm{m}^2 \cdot \mathrm{s}^{-2} \cdot \mathrm{K}^{-1}$
C_p	specific heat capacity at constant pressure	$\mathrm{m}^2 \cdot \mathrm{s}^{-2} \cdot \mathrm{K}^{-1}$
E	specific total energy	$\mathrm{m}^2 \cdot \mathrm{s}^{-2}$
e	specific internal energy	$\mathrm{m}^2 \cdot \mathrm{s}^{-2}$
h_T	specific total enthalpy	$\mathrm{m}^2 \cdot \mathrm{s}^{-2}$
h	specific enthalpy	$\mathrm{m}^2 \cdot \mathrm{s}^{-2}$
k	thermal conductivity	$\mathrm{kg} \cdot \mathrm{m} \cdot \mathrm{s}^{-3} \cdot \mathrm{K}^{-1}$
T	temperature	K
t	time	s
p	thermodynamic pressure	$\mathrm{kg} \cdot \mathrm{m}^{-1} \cdot \mathrm{s}^{-2}$
$\hat{p}$	hydrostatic pressure	$\mathrm{kg} \cdot \mathrm{m}^{-1} \cdot \mathrm{s}^{-2}$
$\boldsymbol{f}_b$	body force	$\mathrm{kg} \cdot \mathrm{m}^{-2} \cdot \mathrm{s}^{-2}$
S	boundary surface	m^2
V	volume	m^3
$\boldsymbol{V}$	velocity vector	$\mathrm{m} \cdot \mathrm{s}^{-1}$
u	x component of velocity	$\mathrm{m} \cdot \mathrm{s}^{-1}$
v	y component of velocity	$\mathrm{m} \cdot \mathrm{s}^{-1}$
w	z component of velocity	$\mathrm{m} \cdot \mathrm{s}^{-1}$
c	speed of sound	$\mathrm{m} \cdot \mathrm{s}^{-1}$
$\boldsymbol{r}$	position vector	m
$\mathbf{n}$	unit normal vector	1
$\hat{\mathbf{t}}$	unit tangent vector	1
$\tilde{\mathbf{t}}$	unit bitangent vector	1
C_R	coefficient of restitution	1
Re	Reynolds number	1
Pr	Prandtl number	1

Ma	Mach number	1
α	thermal diffusivity	$\mathrm{m}^2 \cdot \mathrm{s}^{-1}$
μ	dynamic viscosity	$\mathrm{kg} \cdot \mathrm{m}^{-1} \cdot \mathrm{s}^{-1}$
ν	kinematic viscosity	$\mathrm{m}^2 \cdot \mathrm{s}^{-1}$
γ	heat capacity ratio	1
ρ	density	$\mathrm{kg} \cdot \mathrm{m}^{-3}$
σ_{ij}	stress tensor	$\mathrm{kg} \cdot \mathrm{m}^{-1} \cdot \mathrm{s}^{-2}$
S_{ij}	deviatoric stress tensor	$\mathrm{kg} \cdot \mathrm{m}^{-1} \cdot \mathrm{s}^{-2}$
τ_{ij}	viscous stress tensor	$\mathrm{kg} \cdot \mathrm{m}^{-1} \cdot \mathrm{s}^{-2}$
δ_{ij}	Kronecker tensor	1
I_{ij}	identity tensor	1

Operators

Symbol	**Description**
Δ	difference
∇	gradient operator
$\delta^{\pm}$	upwind-biased interpolation scheme

缩略词

Acronym	**Description**
NASA	National Aeronautics and Space Administration
CPT	Continuation Passing Tree
CFL	Courant-Friedrichs-Lewy
CJ	Chapman-Jouguet
EOS	Equation of State
JWL	Jones-Wilkins-Lee
TVD	Total Variation Diminishing
WENO	Weighted Essentially Non-oscillatory
ZND	Zel'dovich-von Neumann-Doering

第 1 章　引言

第 2 章　定理证明与模型检测的结合

本章首先介绍逻辑系统 CTL_P，CTL_P 是计算树逻辑 CTL 的一个扩展；然后介绍针对 CTL_P 的一个证明系统 SCTL; 之后介绍对证明系统 SCTL 的一个实现 SCTLProV，最后介绍案例分析以及相关实验结果的对比。

2.1　CTL_P

我们用逻辑 $\mathsf{CTL}_P(\mathcal{M})$ 来刻画要验证的系统 $\mathcal{M}$ 的性质，其中 $\mathcal{M}$ 通常指的是一个 Kripke 结构，其定义如下。

定义 2.1.1 (Kripke 结构). *一个 Kripke 结构 $\mathcal{M} = (\mathcal{S}, \longrightarrow, \mathcal{P})$ 包含如下三个部分：*

1. *S 是一个有穷的状态集合；*
2. *$\longrightarrow \subseteq S \times S$ 是一个一个二元关系；对于每一个状态 $s \in S$，至少存在一个 $s' \in S$ 使得 $s \longrightarrow s'$;*
3. *$\mathcal{P}$ 是一个有穷的关系符号的集合；对于每个关系符号 $P \in \mathcal{P}$，都存在自然数 n 使得 $P \in S^n$ 。*

对于一个状态 $s \in S$，我们将 s 的所有的下一个状态的集合定义为

$$\mathsf{Next}(s) = \{s' \mid s \longrightarrow s'\}。$$

一个路径是一个有穷或无穷的状态序列，通常形式为 $s_0, ..., s_n$ 或者 $s_0, s_1, ...$，其中，对于任意自然数 i，如果 s_i 不是该序列的最后一个元素，那么就有 $s_{i+1} \in \mathsf{Next}(s_i)$ 。

我们称 T 是一棵路径树当且仅当对于 T 上的所有由 s 标记的非叶子节点，该节点的所有后继节点正好由 $\mathsf{Next}(s)$ 中的所有元素一一标记。一棵路径树上的所有节点既可以是有穷个也可以是无穷个。

语法。　一个 Kripke 结构 $\mathcal{M}$ 的性质由 $\mathsf{CTL}_P(\mathcal{M})$ 公式表示：

定义 2.1.2. *对于一个给定的 Kripke 模型 $\mathcal{M} = (\mathcal{S}, \longrightarrow, \mathcal{P})$，$CTL_P(\mathcal{M})$ 公式的语法定义如下：*

$$\phi := \begin{cases} \top \mid \bot \mid P(t_1, ..., t_n) \mid \neg P(t_1, ..., t_n) \mid \phi \wedge \phi \mid \phi \vee \phi \mid \\ AX_x(\phi)(t) \mid EX_x(\phi)(t) \mid AF_x(\phi)(t) \mid EG_x(\phi)(t) \mid \\ AR_{x,y}(\phi_1, \phi_2)(t) \mid EU_{x,y}(\phi_1, \phi_2)(t) \end{cases}$$

其中，x 与 y 为变量，取值范围为 S，而 $t_1, ..., t_n$ 既可以是代表状态的常量，也可以是取值范围为 S 的变量。

在定义 2.1.2 中，我们用模态词来绑定公式中的变量。比如，模态词 AX，EX，AF 以及 EG 在公式 ϕ 中绑定了变量 x；而模态词 AR 和 EU 则在公式 ϕ_1 和 ϕ_2 中分别绑定了变量 x 和 y. 变量的替换则写为 $(t/x)\phi$，表示将公式 ϕ 中所有自由出现的变量 x 都替换为 t。

不失一般性地来说，我们假定所有的否定符号都出现在原子命题上；而且有如下缩写：

- $\phi_1 \Rightarrow \phi_2 \equiv \neg\phi_1 \vee \phi_2$,
- $EF_x(\phi)(t) \equiv EU_{z,x}(\top, \phi)(t)$,
- $ER_{x,y}(\phi_1, \phi_2)(t) \equiv EU_{y,z}(\phi_2, ((z/x)\phi_1 \wedge (z/y)\phi_2))(t) \vee EG_y(\phi_2)(t)$, 其中变量 z 既不在 ϕ_1，也不在 ϕ_2 中出现，
- $AG_x(\phi)(t) \equiv \neg(EF_x(\neg\phi)(t))$,
- $AU_{x,y}(\phi_1, \phi_2)(t) \equiv \neg(ER_{x,y}(\neg\phi_1, \neg\phi_2)(t))$.

我们称模态词 AF，EF，AU，以及 EU 为归纳模态词；模态词 AR，ER，AG，以及 EG 为余归纳模态词。

语义。 相应地，对于一个给定的 Kripke 模型 $\mathcal{M}$，$\mathsf{CTL}_P(\mathcal{M})$ 的语义定义如下：

- $\mathcal{M} \models P(s_1, ..., s_n)$：如果 $\langle s_1, ..., s_n\rangle \in P$，而且 P 是一个 $\mathcal{M}$ 上的 n 元关系；
- $\mathcal{M} \models \neg P(s_1, ..., s_n)$：如果 $\langle s_1, ..., s_n\rangle \notin P$，而且 P 是一个 $\mathcal{M}$ 上的 n 元关系；
- $\mathcal{M} \models \top$ 永远成立；
- $\mathcal{M} \models \bot$ 永远不成立；
- $\mathcal{M} \models \phi_1 \wedge \phi_2$：如果 $\mathcal{M} \models \phi_1$ 和 $\mathcal{M} \models \phi_2$ 同时成立；
- $\mathcal{M} \models \phi_1 \vee \phi_2$：如果 $\mathcal{M} \models \phi_1$ 成立，或者 $\mathcal{M} \models \phi_2$ 成立；
- $\mathcal{M} \models AX_x(\phi_1)(s)$：如果对于每个状态 $s' \in \mathsf{Next}(s)$，都有 $\mathcal{M} \models (s'/x)\phi_1$ 成立；
- $\mathcal{M} \models EX_x(\phi_1)(s)$：如果存在一个状态 $s' \in \mathsf{Next}(s)$，使得 $\mathcal{M} \models (s'/x)\phi_1$ 成立；

- $\mathcal{M} \models AF_x(\phi_1)(s)$：如果存在一个有无穷个节点的树 T，而且 T 的根节点是 s, 那么对于 T 的任何一个非叶子节点 s'，s' 的子节点为 $\mathsf{Next}(s')$，对于 T 的任何一个叶子节点 s'，$\vdash (s'/x)\phi_1$ 成立；
- $\mathcal{M} \models EG_x(\phi_1)(s)$：如果存在 $\mathcal{M}$ 上的一个无穷路径 $s_0, s_1, ...$（其中 $s_0 = s$），那么对于任意的自然数 i，都有 $\mathcal{M} \models (s_i/x)\phi_1$ 成立;
- $\mathcal{M} \models AR_{x,y}(\phi_1, \phi_2)(s)$：如果存在一棵路径树 T，T 的根节点由 s 标记，对于任意节点 $s' \in T$ 都有 $\mathcal{M} \models (s'/y)\phi_2$ 成立，而且对于任意的叶子节点 $s'' \in T$ 都有 $\mathcal{M} \models (s''/x)\phi_1$ 成立；
- $\mathcal{M} \models EU_{x,y}(\phi_1, \phi_2)(s)$：如果存在一个无穷路径 $s_0, s_1, ...$（其中 $s_0 = s$）和一个自然数 j，$\mathcal{M} \models (s_j/y)\phi_2$ 成立，而且对于任意的自然数 $i < j$ 都有 $\mathcal{M} \models (s_i/x)\phi_1$ 成立。

CTL vs. CTL$_P$。 在计算树逻辑（CTL）[8,9] 的语法中，原子公式通常用命题符号来表示，而命题符号在计算树逻辑的语义中通常解释为一个 Kripke 结构上的状态集合。在逻辑系统 CTL$_P$ 中，相比于计算树逻辑，我们通过引入多元谓词来增加逻辑系统中公式的表达能力。CTL$_P$ 相比于 CTL 的表达能力的提升可由如下的例子表示出来：

例子 2.1.1. 本例子受多机器人路径规划系统[6,15] 启发。在原例子中，多机器人路径规划系统的规范可以写成 *CTL* 公式：在一个多个区块的地图上，每从初始位置出发的机器人都能到达指定的最终位置，而且在行进的同时，每个机器人都会避免经过某些位置。

在本例子中，除了 *CTL* 所能表示的时序性质之外，我们考虑一种“空间”性质，即表示状态之间的关系。

假定有一个无人车正在一个星球表面行驶，这个星球的表面已经被分成了有穷个小的区域。无人车一次能从一个区域行走到另一个区域，那么我们将无人车的位置看作成一个状态，无人车所有的可能的所在位置则可看作为状态空间，而且无人车从一个位置到另一个位置的移动规律则可看作成迁移关系。无人车的设计需要满足一个基本的性质，即无人车不能永远在一个很小的范围内移动。准确地说，对于给定的距离 σ，在任意状态 s，随着无人车的移动会到达状态 s'，使得 s 和 s' 的位置之间的距离大于 σ。该性质可以由公式 $AG_x(AF_y(D_\sigma(x,y))(x))(s_0)$ 来刻画，其中 s_0 是初始状态，即无人车的降落点；原子公式 $D_\sigma(x,y)$ 则刻画了一种空间性质，即状态 x 和 y 的位置的距离大于 σ。

例子2.1.1中的性质可以很容易由 CTL$_P$ 中的公式进行刻画，然而很难用传统的时序逻辑的公式进行表示。原因是在传统的时序逻辑的语法中通常没有表述一

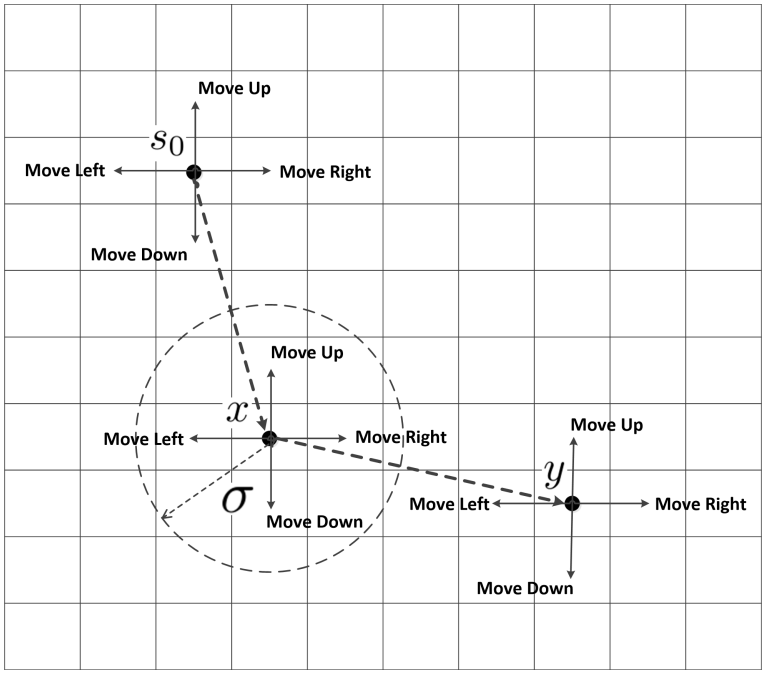


图 2.1: 无人车可能的所在位置

个特定的状态或者多个状态之间的关系的机制，即使在语义中，传统的时序逻辑通常只考虑当前的状态，而无法考虑多个状态之间的关系。

2.2 CTL_P 的证明系统：SCTL

在本节，我们针对逻辑 $\mathsf{CTL}_P(\mathcal{M})$ 给出一个证明系统 $\mathsf{SCTL}(\mathcal{M})$ （Sequent-calculus-like proof system for CTL_P）。在通常意义下的证明系统中，一个公式是可证的当且仅当该公式在所有的模型中都成立，而在 $\mathsf{SCTL}(\mathcal{M})$ 中，一个公式是可证的当且仅当该公式在模型 $\mathcal{M}$ 中是可证的。

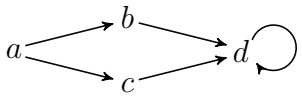
首先，让我们考虑一个 $\mathsf{CTL}_P(\mathcal{M})$ 公式 $AF_x(P(x))(s)$。该公式在模型 $\mathcal{M}$ 中成立当且仅当存在一个路径树 T，T 的根节点由 s 标记，而且 T 上的每个叶子节点都满足 P。

然后,我们考虑一个具有嵌套模态词的 $\mathsf{CTL}_P(\mathcal{M})$ 公式 $AF_x(AF_y(P(x,y))(x))(s)$。如果试图说明该公式在模型 $\mathcal{M}$ 中是成立的，那么就需要找到一个路径树 T，使得 T 的根节点由 s 标记，而且对于 T 中的所有叶子节点 a，$AF_y(P(a,y))(a)$ 是成立的。为了说明 $AF_y(P(a,y))(a)$ 是成立的，则需要又找到一棵路径树 T' 使得 T' 的根节点由 a 标记，而且 T' 上的所有叶子节点 b 都满足 $P(a,b)$。我们可以用以下的两个规则来刻画当前的嵌套的路径树。

$$\frac{\vdash (s/x)\phi}{\vdash AF_x(\phi)(s)}\ \mathbf{AF}\text{-}\mathsf{R}_1$$

$$\frac{\vdash AF_x(\phi)(s_1) \quad \ldots \quad \vdash AF_x(\phi)(s_n)}{\vdash AF_x(\phi)(s)}\ \mathbf{AF}\text{-}\mathsf{R}_2\ {\scriptstyle \{s_1,\ldots s_n\}=\mathsf{Next}(s)}$$

例子 2.2.1. 假设一个模型有如下图所示的迁移规则，



和一个原子谓词 $P = \{b, c\}$，那么公式 $AF_x(P(x))(a)$ 的一个证明如下。

$$\dfrac{\dfrac{\dfrac{}{\vdash P(b)}\ \mathit{atom}\text{-}\mathsf{R}}{\vdash AF_x(P(x))(b)}\ \mathbf{AF}\text{-}\mathsf{R}_1 \qquad \dfrac{\dfrac{}{\vdash P(c)}\ \mathit{atom}\text{-}\mathsf{R}}{\vdash AF_x(P(x))(c)}\ \mathbf{AF}\text{-}\mathsf{R}_1}{\vdash AF_x(P(x))(a)}\ \mathbf{AF}\text{-}\mathsf{R}_2$$

在此证明树中，除了 $\mathbf{AF}$-R_1 和 $\mathbf{AF}$-R_2，我们还应用了如下规则。

$$\frac{}{\vdash P(s_1, ..., s_n)}\ \mathit{atom}\text{-}\mathsf{R}\ {\scriptstyle \langle s_1,\ldots,s_n\rangle \in P}$$

例子 2.2.2. 假设另一个模型，该模型的迁移规则与例子*2.2.1*中相同，除此之外还有原子谓词 $Q = \{(b, d), (c, d)\}$。公式 $AF_x(AF_y(Q(x, y))(x))(a)$ 的证明如下。

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{Q(b,d)}\ \mathit{atom}\text{-}\mathsf{R}}{\vdash AF_y(Q(b,y))(d)}\ \mathbf{AF}\text{-}\mathsf{R}_1}{\vdash AF_y(Q(b,y))(b)}\ \mathbf{AF}\text{-}\mathsf{R}_2}{\vdash AF_x(AF_y(Q(x,y))(x))(b)}\ \mathbf{AF}\text{-}\mathsf{R}_1 \qquad \dfrac{\dfrac{\dfrac{\dfrac{}{Q(c,d)}\ \mathit{atom}\text{-}\mathsf{R}}{\vdash AF_y(Q(c,y))(d)}\ \mathbf{AF}\text{-}\mathsf{R}_1}{\vdash AF_y(Q(c,y))(c)}\ \mathbf{AF}\text{-}\mathsf{R}_2}{\vdash AF_x(AF_y(Q(x,y))(x))(b)}\ \mathbf{AF}\text{-}\mathsf{R}_1}{\vdash AF_x(AF_y(Q(x,y))(x))(a)}\ \mathbf{AF}\text{-}\mathsf{R}_2$$

在 SCTL 中，每个相继式都有 $\Gamma \vdash \phi$ 形式，其中 Γ 是一个可能为空的 SCTL 公式集合，ϕ 是一个 SCTL 公式。不同于通常的相继式演算，

So, as all sequents have the form $\vdash \phi$, the left rules and the axiom rule can be dropped as well. In other words, unlike the usual sequent calculus and like Hilbert systems, SCTL is tailored for deduction, not for hypothetical deduction.

As the left-hand side of sequents is not used to record hypotheses, we will use it to record a different kind of information, that occur in the case of co-inductive modalities, such as the modality EG.

Indeed, the case of the co-inductive formula, for example $EG_x(P(x))(s)$, is more complex than that of the inductive one, such as $AF_x(P(x))(s)$. To justify its validity, one needs to provide an infinite sequence starting from s, and each state

in the infinite sequence verifies P. However, as the model is finite, we can always restrict to regular sequences and use a finite representation of such sequences. This leads us to introduce a rule, called **EG**-merge, that permits to prove a sequent of the form $\vdash EG_x(P(x))(s)$, provided such a sequent already occurs lower in the proof. To make this rule local, we re-introduce hypotheses Γ to record part of the history of the proof. The sequent have therefore the form $\Gamma \vdash \phi$, with a non empty Γ in this particular case only, and the **EG**-merge rule is then just an instance of the axiom rule, that must be re-introduced in this particular case only.

SCTL($\mathcal{M}$) 的证明规则如图2.2所示。

$$\frac{}{\vdash P(s_1,...,s_n)}\ \text{atom-R}_{\langle s_1,...,s_n\rangle\in P} \qquad \frac{}{\vdash \neg P(s_1,...,s_n)}\ \neg\text{-R}_{\langle s_1,...,s_n\rangle\notin P}$$

$$\frac{}{\vdash \top}\ \top\text{-R} \qquad \frac{\vdash \phi_1 \quad \vdash \phi_2}{\vdash \phi_1 \wedge \phi_2}\ \wedge\text{-R} \qquad \frac{\vdash \phi_1}{\vdash \phi_1 \vee \phi_2}\ \vee\text{-R}_1 \qquad \frac{\vdash \phi_2}{\vdash \phi_1 \vee \phi_2}\ \vee\text{-R}_2$$

$$\frac{\vdash (s'/x)\phi}{\vdash EX_x(\phi)(s)}\ \mathbf{EX}\text{-R}_{s'\in \mathsf{Next}(s)} \qquad \frac{\vdash (s_1/x)\phi \quad \dots \quad \vdash (s_n/x)\phi}{\vdash AX_x(\phi)(s)}\ \mathbf{AX}\text{-R}_{\{s_1,...,s_n\}=\mathsf{Next}(s)}$$

$$\frac{\vdash (s/x)\phi}{\vdash AF_x(\phi)(s)}\ \mathbf{AF}\text{-R}_1 \qquad \frac{\vdash AF_x(\phi)(s_1) \quad \dots \quad \vdash AF_x(\phi)(s_n)}{\vdash AF_x(\phi)(s)}\ \mathbf{AF}\text{-R}_2{}_{\{s_1,...,s_n\}=\mathsf{Next}(s)}$$

$$\frac{\vdash (s/x)\phi \quad \Gamma, EG_x(\phi)(s) \vdash EG_x(\phi)(s')}{\Gamma \vdash EG_x(\phi)(s)}\ \mathbf{EG}\text{-R}_{s'\in \mathsf{Next}(s)} \qquad \frac{}{\Gamma \vdash EG_x(\phi)(s)}\ \mathbf{EG}\text{-merge}_{EG_x(\phi)(s)\in\Gamma}$$

$$\frac{\vdash (s/y)\phi_2 \quad \Gamma' \vdash AR_{x,y}(\phi_1,\phi_2)(s_1)\ ...\ \Gamma' \vdash AR_{x,y}(\phi_1,\phi_2)(s_n)}{\Gamma \vdash AR_{x,y}(\phi_1,\phi_2)(s)}\ \mathbf{AR}\text{-R}_1{}_{\{s_1,...,s_n\}=\mathsf{Next}(s),\Gamma'=\Gamma,AR_{x,y}(\phi_1,\phi_2)(s)}$$

$$\frac{\vdash (s/x)\phi_1 \quad \vdash (s/y)\phi_2}{\Gamma \vdash AR_{x,y}(\phi_1,\phi_2)(s)}\ \mathbf{AR}\text{-R}_2 \qquad \frac{}{\Gamma \vdash AR_{x,y}(\phi_1,\phi_2)(s)}\ \mathbf{AR}\text{-merge}_{AR_{x,y}(\phi_1,\phi_2)(s)\in\Gamma}$$

$$\frac{\vdash (s/y)\phi_2}{\vdash EU_{x,y}(\phi_1,\phi_2)(s)}\ \mathbf{EU}\text{-R}_1 \qquad \frac{\vdash (s/x)\phi_1 \quad \vdash EU_{x,y}(\phi_1,\phi_2)(s')}{\vdash EU_{x,y}(\phi_1,\phi_2)(s)}\ \mathbf{EU}\text{-R}_2{}_{s'\in\mathsf{Next}(s)}$$

图 2.2: SCTL($\mathcal{M}$)

有效性与完备性。 命题2.2.1和命题2.2.2的作用是将有穷结构转换为无穷结构，这两个命题被用来证明 SCTL 的有效性；命题2.2.3和命题2.2.4的作用是将无穷结构转换到有穷结构，这两个命题被用来证明 SCTL 的完备性。

命题 2.2.1 (有穷状态序列到无穷状态序列). 给定一个有穷的状态序列 $s_0, ..., s_n$，其中对于任意 $0 \leq i \leq n-1$ 都有 $s_i \longrightarrow s_{i+1}$，而且存在 $0 \leq p \leq n-1$ 使得 $s_n = s_p$。那么，一定存在一个无穷的状态序列 $s'_0, s'_1, ...$ 使得 $s_0 = s'_0$，而且对于任意 $i \geq 0$ 都有 $s'_i \longrightarrow s'_{i+1}$，同时此无穷状态序列中的每个状态都在 $s_0, ..., s_n$ 中。

证明. 本命题所述无穷序列为：$s_0, ..., s_{p-1}, s_p, ..., s_{n-1}, s_p, ...$，其中 $s_0 = s'_0$. □

命题 2.2.2 (有穷路径树到无穷路径树). 设 Φ 为一个状态集合，T 为一个有穷的路径树，T 的每个叶子节点都由某个状态 s 来标记，其中，$s \in \Phi$；或者存在从 T 的根结点到当前叶子节点的分支上的一个节点，使得该节点同样由 s 所标记。那么，一定存在一棵可能无穷的路径树 T'，而且 T' 的所有叶子节点都由 Φ 中的某个状态标记，同时用来标记 T' 节点的状态都用来标记 T 的节点。

证明. 令 T' 的根结点为 T 的根结点，而且对于 T 的每个节点的标记 s 来说，如果 $s \in \Phi$，那么 s 标记 T' 的叶子节点；否则，s 的后继节点分别由 $\mathsf{Next}(s)$ 中的每个元素标记。显然，标记 T' 中节点的状态都标记 T 中的节点。 □

命题 2.2.3 (无穷状态序列到有穷状态序列). 给定一个无穷状态序列 $s_0, s_1, ...$，其中对于任意 $i \geq 0$ 都有 $s_i \longrightarrow s_{i+1}$。那么，一定存在一个有穷的状态序列 $s'_0, ..., s'_n$，对于任意 $0 \leq i \leq n-1$，都存在一个 $0 \leq p \leq n-1$, 使得 $s'_n = s'_p$，而且 $s'_0, ..., s'_n$ 中的所有状态都在 $s_0, s_1, ...$ 中出现。

证明. 由于 Kripke 模型的状态集是有穷的，因此在状态序列 $s_0, s_1, ...$ 一定存在 $p, n \geq 0$，使得 $s_p = s_n$。本命题所述有穷状态序列即为 $s_0, ..., s_n$。 □

命题 2.2.4 (可能无穷的路径树到有穷路径树). 设 Φ 为一个状态集合；T 为一个可能无穷的路径树，其中 T 的所有叶子节点都由 Φ 中的某个状态所标记。那么，一定存在一个有穷的路径树 T'，使得对于 T' 的每个叶子节点的标记 s，$s \in \Phi$，或者存在从 T' 的根结点到该叶子节点的分支上的一个节点，该节点同样由 s 标记。

证明. 由于 Kripke 模型的状态集是有穷的，因此对于 T 的每个无穷分支，都存在 $0 \leq p < n$，使得 $s_p = s_n$。将 T 的每个这样的无穷分支在 s_n 处截断，所得到的路径树即为 T'。显然，由于 T' 具有有穷个分支，同时 T' 的每个分支都是有穷的，因此 T' 也是有穷的。 □

定理 2.2.1 (有效性). 设 $\mathcal{M}$ 为一个 *Kripke* 模型，ϕ 为一个 $CTL_P(\mathcal{M})$ 闭公式。如果相继式 $\vdash \phi$ 具有一个证明，则 $\mathcal{M} \models \phi$ 成立。

证明. 假设相继式 $\vdash \phi$ 具有证明 π，以下对证明 π 的结构做归纳：

- 如果 π 的最后一条规则为 atom-R，那么 $\vdash \phi$ 具有 $\vdash P(s_1, ..., s_n)$ 形式，因此 $\mathcal{M} \models P(s_1, ..., s_n)$。
- 如果 π 的最后一条规则为 $\neg$-R，那么 $\vdash \phi$ 具有 $\vdash \neg P(s_1, ..., s_n)$ 形式，因此 $\mathcal{M} \models \neg P(s_1, ..., s_n)$。
- 如果 π 的最后一条规则为 $\top$-R，那么 $\vdash \phi$ 具有 $\vdash \top$ 形式，因此 $\mathcal{M} \models \top$。
- 如果 π 的最后一条规则为 $\wedge$-R，那么 $\vdash \phi$ 具有 $\vdash \phi_1 \wedge \phi_2$ 形式。根据归纳假设，$\mathcal{M} \models \phi_1$ 与 $\mathcal{M} \models \phi_2$ 均成立，因此 $\mathcal{M} \models \phi_1 \wedge \phi_2$。
- 如果 π 的最后一条规则为 $\vee$-R，那么 $\vdash \phi$ 具有 $\vdash \phi_1 \vee \phi_2$ 形式。根据归纳假设，$\mathcal{M} \models \phi_1$ 成立或 $\mathcal{M} \models \phi_2$ 成立，因此 $\mathcal{M} \models \phi_1 \vee \phi_2$。
- 如果 π 的最后一条规则为 AX-R，那么 $\vdash \phi$ 具有 $\vdash AX_x(\phi_1)(s)$ 形式。根据归纳假设，对于任意 $s' \in \mathsf{Next}(s)$，都有 $\mathcal{M} \models (s'/x)\phi_1$ 成立，因此 $\mathcal{M} \models AX_x(\phi_1)(s)$。
- 如果 π 的最后一条规则为 EX-R，那么 $\vdash \phi$ 具有 $\vdash EX_x(\phi_1)(s)$ 形式。根据归纳假设，存在 $s' \in \mathsf{Next}(s)$，使得 $\mathcal{M} \models (s'/x)\phi_1$ 成立，因此 $\mathcal{M} \models EX_x(\phi_1)(s)$。
- 如果 π 的最后一条规则为 $\mathbf{AF}$-R_1 或 $\mathbf{AF}$-R_2，那么 $\vdash \phi$ 具有 $\vdash AF_x(\phi_1)(s)$ 形式。根据证明 π，我们利用归纳的方式构造一棵路径树 $|\pi|$。构造方式如下：
  - 如果 π 的最后一条规则为 $\mathbf{AF}$-R_1，而且 ρ 为 $\vdash (s/x)\phi_1$ 的证明，则路径树 $|\pi|$ 只包含一个节点 s；
  - 如果 π 的最后一条规则为 $\mathbf{AF}$-R_2，而且 $\pi_1, ..., \pi_n$ 分别为 $\vdash AF_x(\phi_1)(s_1), ..., AF_x(\phi_n)(s_n)$ 的证明，其中 $\{s_1, ..., s_n\} = \mathsf{Next}(s)$，那么令 $|\pi|$ 等于 $s(|\pi_1|, ..., |\pi_n|)$。

  路径树 $|\pi|$ 的根结点为 s，而且对于 $|\pi|$ 的每个叶子节点 s' 来说，$\vdash (s'/x)\phi_1$ 都有一个比 π 小的证明。根据归纳假设，对于 $|\pi|$ 的每个叶子节点 s' 来说，都有 $\mathcal{M} \models (s'/x)\phi_1$ 成立，因此，$\mathcal{M} \models AF_x(\phi_1)(s)$ 成立。
- 如果 π 的最后一条规则为 $\mathbf{EG}$-R，则 $\vdash \phi$ 具有 $\vdash EG_x(\phi_1)(s)$ 形式。根据证明 π，我们归纳构造一个状态序列 $|\pi|$。构造方式如下：
  - 如果 π 的最后一条规则为 $\mathbf{EG}$-merge，那么 $|\pi|$ 只包含一个单独的状态 s；
  - 如果 π 的最后一条规则为 $\mathbf{EG}$-R，而且 ρ 和 π_1 分别为 $\vdash (s/x)\phi_1$ 和 $\Gamma, EG_x(\phi_1)(s) \vdash EG_x(\phi_1)(s')$ 的证明，其中 $s' \in \mathsf{Next}(s)$，那么令 $|\pi|$ 等于 $s|\pi_1|$。

对于状态序列 $|\pi| = s_0, ..., s_n$，$s_0 = s$；对于任意 $0 \leq i \leq n-1$，$s_i \longrightarrow s_{i+1}$；对于任意 $0 \leq i \leq n$，$\vdash (s_i/x)\phi_1$ 都有一个比 π 小的证明；而且存在 $p < n$ 使得 $s_n = s_p$。根据归纳假设，对于任意 $i \geq 0$，都有 $\mathcal{M} \models (s_i/x)\phi_1$ 成立。由命题2.2.1可知，存在一个无穷的状态序列 $s'_0, s'_1, ...$，其中对于任意 $i \geq 0$ 都有 $s'_i \longrightarrow s'_{i+1}$，同时 $\mathcal{M} \models (s'_i/x)\phi_1$ 成立。因此，$\mathcal{M} \models EG_x(\phi_1)(s)$ 成立。

- 如果 π 的最后一条规则为 **AR**-R_1 或 **AR**-R_2，那么 $\vdash \phi$ 具有 $\vdash AR_x(\phi_1, \phi_2)(s)$ 形式。根据 π，我们归纳构造一个有穷的路径树 $|\pi|$。构造方式如下：
  - 如果 π 的最后一条规则为 **AR**-R_1，而且 ρ_1 和 ρ_2 分别为 $\vdash (s/x)\phi_1$ 和 $\vdash (s/x)\phi_2$ 的证明，那么 $|\pi|$ 只包含一个节点 s；
  - 如果 π 的最后一条规则为 **AR**-merge，那么 $|\pi|$ 只包含一个节点 s；
  - 如果 π 的最后一条规则为 **AR**-R_2，而且 $\rho, \pi_1, ..., \pi_n$ 分别为 $\vdash (s/y)\phi_2$, $\Gamma, AR_{x,y}(\phi_1, \phi_2)(s) \vdash AR_{x,y}(\phi_1, \phi_2)(s_1), ..., \Gamma, AR_{x,y}(\phi_1, \phi_2)(s) \vdash AR_{x,y}(\phi_1, \phi_2)(s_n)$ 的证明，其中 $\{s_1, ..., s_n\} = \mathsf{Next}(s)$，那么令 $|\pi|$ 等于 $s(|\pi_1|, ..., |\pi_n|)$。

  路径树 $|\pi|$ 以 s 为根结点，而且对于 $|\pi|$ 的每个节点 s' 来说，$\vdash (s'/y)\phi_2$ 都有一个比 π 小的证明；对于 $|\pi|$ 的任意叶子节点 s' 来说，$\vdash (s'/x)\phi_1$ 有一个比 π 小的证明，或者在从 $|\pi|$ 的根结点到当前叶子节点的分支上存在一个节点，使得 s' 标记此节点。根据归纳假设，对于 $|\pi|$ 的任意节点 s'，$\models (s'/y)\phi_2$ 成立，而且对于 $|\pi|$ 的任意叶子节点 s'，$\models (s'/x)\phi_1$ 成立，或者在从 $|\pi|$ 的根结点到当前叶子节点的分支上存在一个节点，使得 s' 标记此节点。根据命题2.2.2，存在一个可能无穷的路径树 T'，使得对于 T' 的每个节点 s'，都有 $\models (s'/y)\phi_2$ 成立，而且对于 T' 的每个叶子节点 s'，都有 $\models (s'/x)\phi_1$ 成立。因此，$\models AR_{x,y}(\phi_1, \phi_2)(s)$ 成立。

- 如果 π 的最后一条规则为 **EU**-R_1 或 **EU**-R_2，那么 $\vdash \phi$ 具有 $\vdash EU_{x,y}(\phi_1, \phi_2)(s)$ 形式。根据 π，我们归纳构造一个有穷状态序列 $|\pi|$。构造过程如下：
  - 如果 π 的最后一条规则为 **EU**-R_1，那么 $|\pi|$ 只包含一个状态 s；
  - 如果 π 的最后一条规则为 **EU**-R_2，而且 ρ 和 π_1 分别为 $\vdash (s/x)\phi_1$ 和 $\vdash EU_{x,y}(\phi_1, \phi_2)(s')$ 的证明，那么令 $|\pi|$ 等于 $s|\pi_1|$。

  在状态序列 $|\pi| = s_0, ..., s_n$ 中，$s_0 = s$；对于任意 $0 \leq i \leq n-1$，$s_i \longrightarrow s_{i+1}$；对于任意 $0 \leq i \leq n-1$，$\vdash (s_i/x)\phi_1$ 有一个比 π 小的证明；而且 $\vdash (s_n/y)\phi_2$ 有一个比 π 小的证明。根据归纳假设，对任意 $0 \leq i \leq n-1$，$\models (s_i/x)\phi_1$ 和 $\models (s_n/y)\phi_2$ 均成立。因此，$\models EU_{x,y}(\phi_1, \phi_2)(s)$ 成立。

- π 的最后一条规则不能为 merge 规则。

□

定理 2.2.2 (完备性). 设 ϕ 是一个 $CTL_P(\mathcal{M})$ 闭公式。如果 $\mathcal{M} \models \phi$，则 $\vdash \phi$ 在 $SCTL(\mathcal{M})$ 中是可证的。

证明. 对 ϕ 的结构作归纳：

- 如果 $\phi = P(s_1, ..., s_n)$，那么由 $\mathcal{M} \models P(s_1, ..., s_n)$ 可知，$\vdash P(s_1, ..., s_n)$ 是可证的。
- 如果 $\phi = \neg P(s_1, ..., s_n)$，那么由 $\mathcal{M} \models \neg P(s_1, ..., s_n)$ 可知，$\vdash \neg P(s_1, ..., s_n)$ 是可证的。
- 如果 $\phi = \top$，那么显然 $\vdash \top$ 是可证的。
- 如果 $\phi = \bot$，那么显然 $\vdash \bot$ 是不可证的。
- 如果 $\phi = \phi_1 \wedge \phi_2$，那么由于 $\mathcal{M} \models \phi_1 \wedge \phi_2$，因此 $\mathcal{M} \models \phi_1$ 和 $\mathcal{M} \models \phi_2$ 均成立。根据归纳假设，$\vdash \phi_1$ 和 $\vdash \phi_2$ 均可证。因此，$\vdash \phi_1 \wedge \phi_2$ 是可证的。
- 如果 $\phi = \phi_1 \vee \phi_2$，那么由于 $\mathcal{M} \models \phi_1 \vee \phi_2$，因此 $\mathcal{M} \models \phi_1$ 或 $\mathcal{M} \models \phi_2$ 成立。根据归纳假设，$\vdash \phi_1$ 或 $\vdash \phi_2$ 是可证的。因此，$\vdash \phi_1 \vee \phi_2$ 是可证的。
- 如果 $\phi = AX_x(\phi_1)(s)$，那么由于 $\mathcal{M} \models AX_x(\phi_1)(s)$，因此对于任意 $s' \in \mathsf{Next}(s)$，都有 $\mathcal{M} \models (s'/x)\phi_1$ 成立。根据归纳假设，对于任意 $s' \in \mathsf{Next}(s)$，$\vdash (s'/x)\phi_1$ 都是可证的。因此，$\vdash AX_x(\phi_1)(s)$ 是可证的。
- 如果 $\phi = EX_x(\phi_1)(s)$，那么由于 $\mathcal{M} \models EX_x(\phi_1)(s)$，因此存在 $s' \in \mathsf{Next}(s)$ 使得 $\mathcal{M} \models (s'/x)\phi_1$ 成立。根据归纳假设，$\vdash (s'/x)\phi_1$ 是可证的，因此 $\vdash EX_x(\phi_1)(s)$ 是可证的。
- 如果 $\phi = AF_x(\phi_1)(s)$，那么由于 $\mathcal{M} \models AF_x(\phi_1)(s)$，因此存在一棵有穷的路径树 T，并且 T 以 s 为根结点；对于 T 的每个非叶子节点 s'，s' 的后继节点分别由 $\mathsf{Next}(s)$ 中的元素所标记；对于 T 的每个叶子节点 s'，都有 $\mathcal{M} \models (s'/x)\phi_1$ 成立。根据归纳假设，$\vdash (s'/x)\phi_1$ 是可证的。然后，对于 T 的每个以子树 T'（设 T' 的根结点为 s'），我们归纳构造 $\vdash AF_x(\phi_1)(s')$ 的一个证明 $|T'|$。构造过程如下：
  - 如果 T' 只包含一个节点 s'，那么 $|T'|$ 的最后一条规则为 $\mathbf{AF}$-R_1，同时 $|T'|$ 中包含 $\vdash (s'/x)\phi_1$ 的证明；

– 如果 $T' = s'(T_1, ..., T_n)$，那么 $|T'|$ 的最后一条规则为 **AF**-R_2，同时 $|T'_1|, ..., |T'_n|$ 分别为 $\vdash AF_x(\phi_1)(s_1), ..., \vdash AF_x(\phi_1)(s_n)$ 的证明，其中 $\{s_1, ..., s_n\} = \mathsf{Next}(s)$。

因此，$|T|$ 是 $\vdash AF_x(\phi_1)(s)$ 的一个证明。

- 如果 $\phi = EG_x(\phi_1)(s)$，那么由于 $\mathcal{M} \models EG_x(\phi_1)(s)$，因此存在一个状态序列 $s_0, ..., s_n$ 使得 $s_0 = s$，而且对于任意 $0 \leq i \leq n$ 都有 $\mathcal{M} \models (s_i/x)\phi_1$ 成立。根据归纳假设，$\vdash (s_i/x)\phi_1$ 是可证的。根据命题2.2.3，存在一个有穷的状态序列 $T = s_0, ..., s_n$ 使得对任意 $0 \leq i \leq n-1$，$s_i \longrightarrow s_{i+1}$，同时 $\vdash (s_i/x)\phi_1$ 是可证的，而且存在 $p < n$ 使得 $s_n = s_p$。对于 T 的每个后缀 $s_i, ..., s_n$，我们归纳构造 $|s_i, ..., s_n|$ 为 $EG_x(\phi_1)(s_0), \ldots, EG_x(\phi_1)(s_{i-1}) \vdash EG_x(\phi_1)(s_i)$ 的证明。构造方式如下：

  – $|s_n|$ 的最后一条规则为 **EG**-merge；
  – 如果 $i \leq n-1$，根据归纳假设，由于 $\vdash (s_i/x)\phi_1$ 是可证的，而且 $|s_{i+1}, ..., s_n|$ 是 $EG_x(\phi_1)(s_0), ..., EG_x(\phi_1)(s_i) \vdash EG_x(\phi_1)(s_{i+1})$ 的一个证明。因此，$|s_i, ..., s_n|$ 是 $EG_x(\phi_1)(s_0), \ldots, EG_x(\phi_1)(s_{i-1}) \vdash EG_x(\phi_1)(s_i)$ 的一个证明，而且最后一条规则为 **EG**-R。

因此，$|s_0, ..., s_n|$ 是 $\vdash EG_x(\phi_1)(s)$ 的一个证明。

- 如果 $\phi = AR_{x,y}(\phi_1, \phi_2)(s)$，那么由于 $\mathcal{M} \models AR_{x,y}(\phi_1, \phi_2)(s)$，因此存在一棵以 s 为根节点的可能无穷的路径树，对于该路径树的每个节点 s'，都有 $\mathcal{M} \models (s'/x)\phi_2$；对于该路径树的每个叶子节点 s'，都有 $\mathcal{M} \models (s'/x)\phi_1$。根据归纳假设，对于该路径树的每个节点 s'，$\vdash (s'/y)\phi_2$ 是可证的，而且对于该路径树的每个叶子节点 s'，$\vdash (s'/y)\phi_1$ 是可证的。由命题2.2.4可知，存在一棵有穷的路径树 T，对于该路径树的每个节点 s'，$\vdash (s'/y)\phi_2$ 是可证的；对于该路径树的每个叶子节点 s'，$\vdash (s'/y)\phi_1$ 是可证的，或者 s' 为从 T 的根节点到该叶子节点分支上的节点。然后，对于 T 的每个子树 T'，我们归纳构造 $AR_{x,y}(\phi_1, \phi_2)(s_1), ..., AR_{x,y}(\phi_1, \phi_2)(s_m) \vdash AR_{x,y}(\phi_1, \phi_2)(s')$ 的一个证明 $|T'|$，其中 s' 为 T' 的根节点，而且 $s_1, ..., s_m$ 为从 T 的根节点到 T' 的根节点的分支。构造方式如下：

  – 如果 T' 只包含一个单独的节点 s'，同时 $\vdash (s'/x)\phi_1$ 是可证的，那么根据归纳假设，$\vdash (s'/x)\phi_1$ 和 $\vdash (s'/y)\phi_2$ 皆可证，而且 $|T'|$ 的最后一条规则为 **AR**-R_1；
  – 如果 T' 只包含一个单独的节点，同时 s' 包含在 $s_1, ..., s_m$ 中，那么 $|T'|$ 的最后一条规则为 **AR**-merge；

– 如果 $T' = s'(T_1, ..., T_n)$，那么根据归纳假设，$|T_1|, ..., |T_n|$ 分别为

$$AR_{x,y}(\phi_1, \phi_2)(s_1), ..., AR_{x,y}(\phi_1, \phi_2)(s_m),$$
$$AR_{x,y}(\phi_1, \phi_2)(s') \vdash AR_{x,y}(\phi_1, \phi_2)(s'_1)$$
$$...$$
$$AR_{x,y}(\phi_1, \phi_2)(s_1), ..., AR_{x,y}(\phi_1, \phi_2)(s_m),$$
$$AR_{x,y}(\phi_1, \phi_2)(s') \vdash AR_{x,y}(\phi_1, \phi_2)(s'_n)$$

的证明，同时 $|T'|$ 的最后一条规则为 $\mathbf{AR}$-R_2，其中 $s'_1, ..., s'_n = \mathsf{Next}(s')$。

因此，$|T|$ 是 $\vdash AR_{x,y}(\phi_1, \phi_2)(s)$ 的一个证明。

- 如果 $\phi = EU_{x,y}(\phi_1, \phi_2)(s)$，那么由于 $\mathcal{M} \models EU_{x,y}(\phi_1, \phi_2)(s)$，因此存在一个有穷的状态序列 $T = s_0, ..., s_n$ 使得 $\mathcal{M} \models (s_n/y)\phi_2$ 成立，而且对于任意 $0 \leq i \leq n-1$，$\mathcal{M} \models (s_i/x)\phi_1$ 成立。根据归纳假设，$\vdash (s_n/y)\phi_2$ 是可证的，而且对于任意 $0 \leq i \leq n-1$，$\vdash (s_i/x)\phi_1$ 是可证的。然后，对于 T 的每个后缀 $s_i, ..., s_n$，我们归纳构造 $|s_i, ..., s_n|$ 为 $\vdash EU_{x,y}(\phi_1, \phi_2)(s_i)$ 的证明。构造方式如下：

  – $|s_n|$ 的最后一条规则为 $\mathbf{EU}$-R_1；
  – 如果 $i \leq n-1$，那么根据归纳假设，由于 $|s_{i+1}, ..., s_n|$ 是 $\vdash EU_{x,y}(\phi_1, \phi_2)(s_{i+1})$ 的证明，而且 $\vdash (s_i/x)\phi_1$ 是可证的，因此，$|s_i, ..., s_n|$ 是 $\vdash EU_{x,y}(\phi_1, \phi_2)(s_i)$ 的证明。

因此，$|s_0, ..., s_n|$ 是 $\vdash EU_{x,y}(\phi_1, \phi_2)(s)$ 的一个证明。

□

2.3 SCTL 的工具实现：SCTLProV

本节介绍 SCTL 的一个实现—SCTLProV（图2.3）以及该工具与其他模型检测工具的对比。SCTLProV 的工作方式如下：首先，SCTLProV 读入一个输入文件，并将该输入文件解析到一个 Kripke 模型以及若干个 SCTL 公式；然后，对于每个公式，SCTLProV 搜索该公式的证明，如果该公式可证，则并输出该证明（或者只输出 True），如果该公式不可正，则输出该公式的非的证明（或者只输出 False）。

2.3.1 证明搜索

SCTLProV 的证明搜素方法如下：首先，对于要证明的相继式，以及 SCTL 规则将该公式的所有的前提给定一个序；然后，依次对这些前提进行证明搜索。我们

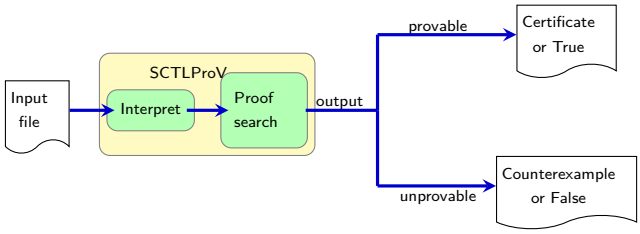


图 2.3: SCTLProV.

将以上证明搜索方法定义成一系列对于连续传递树（定义2.3.1）的重写规则。下面我们介绍连续传递树的概念。

2.3.1.1 连续传递树

在连续传递树中，连续一个基本的概念。在计算机程序设计语言理论[1,19] 中，连续是计算机程序将要执行的部分的显示表示。

定义 2.3.1 (连续传递树). 一个连续传递树（*Continuation Passing Tree*，简写为 *CPT*）指的是一棵同时满足以下条件的二叉树：

- 每个叶子节点被 $\mathfrak{t}$ 或 $\mathfrak{f}$ 标记，其中 $\mathfrak{t}$ 和 $\mathfrak{f}$ 是不同的两个符号；
- 每个非叶子节点都被一个 *SCTL* 相继式标记。

对于 *CPT* 的每个非叶子节点来说，它的左子树称之为该节点的 $\mathfrak{t}$-连续；它的右子树称之为该节点的 $\mathfrak{f}$-连续。对于一个 *CPT* c 来说，若 c 的根节点为 $\Gamma \vdash \phi$，以及 c 的 $\mathfrak{t}$-连续和 $\mathfrak{f}$-连续分别为 c_1 和 c_2，那么我们将 c 记作 $cpt(\Gamma \vdash \phi, c_1, c_2)$，或者可以表示为如下形式：

$$\begin{array}{c} \Gamma \vdash \phi \\ \diagup \quad \diagdown \\ c_1 \quad c_2 \end{array}$$

在 SCTLProV 的证明搜索算法可总结为：对于给定的 SCTL 相继式 $\vdash \phi$，我们构造一个连续传递树 $c = \mathsf{cpt}(\vdash \phi, \mathfrak{t}, \mathfrak{f})$，然后根据图2.4所示的重写规则将 c 重写到 $\mathfrak{t}$ 或 $\mathfrak{f}$。如果 c 最终重写到 $\mathfrak{t}$，那么 $\vdash \phi$ 是可证的；如果 c 最终重写到 $\mathfrak{f}$，那么 $\vdash \phi$ 是不可证的。

在 CPT 的重写规则中，对一个 CPTc 的一步重写只需判断 c 的根节点，而与 c 的子表达式无关。例如，根据重写规则，CPT$\mathsf{cpt}(\vdash \phi_1 \wedge \phi_2, \mathfrak{t}, \mathfrak{f})$ 重写到 $\mathsf{cpt}(\vdash \phi_1, \mathsf{cpt}(\vdash \phi_2, \mathfrak{t}, \mathfrak{f}), \mathfrak{f})$，此步重写意味着：如果搜索 $\vdash \phi_1$ 的证明成功，则继续搜索 $\vdash \phi_2$ 的证明；如果搜索 $\vdash \phi_1$ 的证明失败，则直接判定 $\vdash \phi_1 \wedge \phi_2$ 不可证。接下来，根据 ϕ_1 的结构，继续对 $\mathsf{cpt}(\vdash \phi_1, \mathsf{cpt}(\vdash \phi_2, \mathfrak{t}, \mathfrak{f}), \mathfrak{f})$ 进行重写。

$\mathsf{cpt}(\vdash \top, c_1, c_2) \rightsquigarrow c_1 \quad \mathsf{cpt}(\vdash \bot, c_1, c_2) \rightsquigarrow c_2$
$\mathsf{cpt}(\vdash P(s_1, ..., s_n), c_1, c_2) \rightsquigarrow c_1 \quad [\langle s_1, ..., s_n \rangle \in P]$
$\mathsf{cpt}(\vdash P(s_1, ..., s_n), c_1, c_2) \rightsquigarrow c_2 \quad [\langle s_1, ..., s_n \rangle \notin P]$
$\mathsf{cpt}(\vdash \neg P(s_1, ..., s_n), c_1, c_2) \rightsquigarrow c_2 \quad [\langle s_1, ..., s_n \rangle \in P]$
$\mathsf{cpt}(\vdash \neg P(s_1, ..., s_n), c_1, c_2) \rightsquigarrow c_1 \quad [\langle s_1, ..., s_n \rangle \notin P]$
$\mathsf{cpt}(\vdash \phi_1 \wedge \phi_2, c_1, c_2) \rightsquigarrow \mathsf{cpt}(\vdash \phi_1, \mathsf{cpt}(\vdash \phi_2, c_1, c_2), c_2)$
$\mathsf{cpt}(\vdash \phi_1 \vee \phi_2, c_1, c_2) \rightsquigarrow \mathsf{cpt}(\vdash \phi_1, c_1, \mathsf{cpt}(\vdash \phi_2, c_1, c_2))$
$\mathsf{cpt}(\vdash AX_x(\phi)(s), c_1, c_2) \rightsquigarrow \mathsf{cpt}(\vdash (s_1/x)\phi, \mathsf{cpt}(\vdash (s_2/x)\phi, \mathsf{cpt}(...\mathsf{cpt}(\vdash (s_n/x)\phi, c_1, c_2), ..., c_2), c_2), c_2)$ $[\{s_1, ..., s_n\} = \mathsf{Next}(s)]$
$\mathsf{cpt}(\vdash EX_x(\phi)(s), c_1, c_2) \rightsquigarrow \mathsf{cpt}(\vdash (s_1/x)\phi, c_1, \mathsf{cpt}(\vdash (s_2/x)\phi, c_1, \mathsf{cpt}(...\mathsf{cpt}(\vdash (s_n/x)\phi, c_1, c_2)...)))$ $[\{s_1, ..., s_n\} = \mathsf{Next}(s)]$
$\mathsf{cpt}(\Gamma \vdash AF_x(\phi)(s), c_1, c_2) \rightsquigarrow c_2 \quad [AF_x(\phi)(s) \in \Gamma]$
$\mathsf{cpt}(\Gamma \vdash AF_x(\phi)(s), c_1, c_2) \rightsquigarrow$ $\mathsf{cpt}(\vdash (s/x)\phi, c_1, \mathsf{cpt}(\Gamma' \vdash AF_x(\phi)(s_1), \mathsf{cpt}(...\mathsf{cpt}(\Gamma' \vdash AF_x(\phi)(s_n), c_1, c_2)..., c_2), c_2))$ $[\{s_1, ..., s_n\} = \mathsf{Next}(s), AF_x(\phi)(s) \notin \Gamma, \text{and } \Gamma' = \Gamma, AF_x(\phi)(s)]$
$\mathsf{cpt}(\Gamma \vdash EG_x(\phi)(s), c_1, c_2) \rightsquigarrow c_1 \quad [EG_x(\phi)(s) \in \Gamma]$
$\mathsf{cpt}(\Gamma \vdash EG_x(\phi)(s), c_1, c_2) \rightsquigarrow$ $\mathsf{cpt}(\vdash (s/x)\phi, \mathsf{cpt}(\Gamma' \vdash EG_x(\phi)(s_1), c_1, \mathsf{cpt}(...\mathsf{cpt}(\Gamma' \vdash EG_x(\phi)(s_n), c_1, c_2)...)), c_2)$ $[\{s_1, ..., s_n\} = \mathsf{Next}(s), EG_x(\phi)(s) \notin \Gamma, \text{and } \Gamma' = \Gamma, EG_x(\phi)(s)]$
$\mathsf{cpt}(\Gamma \vdash AR_{x,y}(\phi_1, \phi_2)(s), c_1, c_2) \rightsquigarrow c_1 \quad [(AR_{x,y}(\phi_1, \phi_2)(s) \in \Gamma]$
$\mathsf{cpt}(\Gamma \vdash AR_{x,y}(\phi_1, \phi_2)(s), c_1, c_2) \rightsquigarrow$ $\mathsf{cpt}(\vdash (s/y)\phi_2, \mathsf{cpt}(\vdash (s/x)\phi_1, c_1, \mathsf{cpt}(\Gamma' \vdash AR_{x,y}(\phi_1, \phi_2)(s_1), \mathsf{cpt}(...\mathsf{cpt}(\Gamma' \vdash AR_{x,y}(\phi_1, \phi_2)(s_n),$ $c_1, c_2)..., c_2), c_2)), c_2) \quad [\{s_1, ..., s_n\} = \mathsf{Next}(s), AR_{x,y}(\phi_1, \phi_2)(s) \notin \Gamma, \text{ and } \Gamma' = \Gamma, AR_{x,y}(\phi_1, \phi_2)(s)]$
$\mathsf{cpt}(\Gamma \vdash EU_{x,y}(\phi_1, \phi_2)(s), c_1, c_2) \rightsquigarrow c_2 \quad [EU_{x,y}(\phi_1, \phi_2)(s) \in \Gamma]$
$\mathsf{cpt}(\Gamma \vdash EU_{x,y}(\phi_1, \phi_2)(s), c_1, c_2) \rightsquigarrow$ $\mathsf{cpt}(\vdash (s/y)\phi_2, c_1, \mathsf{cpt}(\vdash (s/x)\phi_1, \mathsf{cpt}(\Gamma' \vdash EU_{x,y}(\phi_1, \phi_2)(s_1), c_1, \mathsf{cpt}(...\mathsf{cpt}(\Gamma' \vdash EU_{x,y}(\phi_1, \phi_2)(s_n),$ $c_1, c_2)...)), c_2)) \quad [\{s_1, ..., s_n\} = \mathsf{Next}(s), EU_{x,y}(\phi_1, \phi_2)(s) \notin \Gamma, \text{ and } \Gamma' = \Gamma, EU_{x,y}(\phi_1, \phi_2)(s)]$

图 2.4: CPT 的重写规则.

2.3.2 证明搜索的可终止性

在证明利用图2.3.1表示的重写规则，使得 SCTLProV 的证明搜索是可终止的之前，我们需要引入以下定义和命题。

定义 2.3.2 (字典路径序（lexicographic path ordering）[7,18]). 设 $\succeq$ 是函数符号集合 F 的一个拟序（*quasi-ordering*），其中 F 的每个符号的元数（*arity*）是固定不变的。集合 $T(F)$（由 F 生成的项的集合）上的字典路径序 $\succeq_{\mathsf{lpo}}$ 的归纳定义如下：

$s = f(s_1, ..., s_m) \succeq_{\mathsf{lpo}} g(t_1, ..., t_n) = t$ 当且仅当以下至少一条断言成立：

- 存在 $i \in \{1, ..., m\}$，使得 $s_i \succeq_{\mathsf{lpo}} t$ 成立。
- 对于任意 $j \in \{1, ..., n\}$，$f \succ g$ 和 $s \succ_{\mathsf{lpo}} t_j$ 同时成立。

- 对于任意 $j \in \{2,...,n\}, f = g$，$(s_1,...,s_m) \succeq'_{\mathsf{lpo}} (t_1,...,t_n)$ 和 $s \succ_{\mathsf{lpo}} t_j$ 都成立，其中 $\succeq'_{\mathsf{lpo}}$ 是由 $\succeq_{\mathsf{lpo}}$ 产生的字典序。

命题 2.3.1 (字典路径序的良基性). 如果 $\succeq$ 是函数符号集合 F 的一个拟序（*quasi-ordering*），其中 F 的每个符号的元数（*arity*）是固定不变的，那么根据集合 $T(F)$（由 F 生成的项的集合）上的字典路径序 $\succeq_{\mathsf{lpo}}$ 是良基的当且仅当 $\succeq$ 是良基的。

证明. 证明由 Dershowitz 提出，参考[7]。 □

定义 2.3.3 (相继式的权重). 假设一个 *Kripke* 模型的状态集的基数为 n；$\Gamma \vdash \phi$ 是一个 *SCTL*($\mathcal{M}$) 相继式；$|\phi|$ 是公式 ϕ 的大小；$|\Gamma|$ 是 Γ 的基数。相继式 $\Gamma \vdash \phi$ 的权重为

$$w(\Gamma \vdash \phi) = \langle |\phi|, (n - |\Gamma|) \rangle$$

命题 2.3.2 (可终止性). 假设 $\mathcal{M}$ 是一个 *Kripke* 模型，ϕ 是一个 $CTL_P(\mathcal{M})$ 闭公式，那么 *cpt*($\vdash \phi, \mathfrak{t}, \mathfrak{f}$) 能在有限步之内重写到 $\mathfrak{t}$ 或 $\mathfrak{f}$。

证明. 令 $F = \{\mathfrak{t}, \mathfrak{f}, \mathsf{cpt}\} \cup \mathsf{Seq}$，其中 Seq 是在 $\mathsf{cpt}(\vdash \phi, \mathfrak{t}, \mathfrak{f})$ 的重写步骤中所出现的相继式的集合；cpt 的元数是 3，F 中其他符号的元数是 0。F 上的拟序 $\succeq$（$\forall f, g \in F$, $f \succ g$ 是指 “$f \succeq g$ 同时 $f \neq g$”）定义如下：

- $\mathsf{cpt} \succ \mathfrak{t}$;
- $\mathsf{cpt} \succ \mathfrak{f}$;
- 对于每个相继式 $\Gamma \vdash \phi$ 都有 $\Gamma \vdash \phi \succ \mathsf{cpt}$;
- $\Gamma \vdash \phi \succ \Gamma' \vdash \phi'$ 当且仅当 $w(\Gamma \vdash \phi) > w(\Gamma' \vdash \phi')$，其中 $>$ 是自然数对上的字典序。

令 $\succeq_{\mathsf{lpo}}$ 为由 $\succeq$ 生成的关于 CPT 的字典序。显然，$\succeq$ 是良基的，因此根据命题2.3.1可知，$\succeq_{\mathsf{lpo}}$ 也是良基的。

若要证明重写系统是可终止的，只需证明对于每一步重写 $c \rightsquigarrow c'$，都有 $c \succ_{\mathsf{lpo}} c'$。下面我们针对重写规则逐条进行分析：

假设 c 是 $\mathsf{cpt}(\Gamma \vdash \phi, c_1, c_2)$ 形式的。

- 如果 $\phi = \top, \bot, P(s_1,...,s_m)$ 或 $\neg P(s_1,...,s_m)$，那么由于 c_1 和 c_2 是 $\mathsf{cpt}(\Gamma \vdash \phi, c_1, c_2)$ 的子项，因此，$\mathsf{cpt}(\Gamma \vdash \phi, c_1, c_2) \succ_{\mathsf{lpo}} c_1$ 以及 $\mathsf{cpt}(\Gamma \vdash \phi, c_1, c_2) \succ_{\mathsf{lpo}} c_2$。
- 如果 $\phi = \phi_1 \wedge \phi_2$, 那么由 $\succ_{\mathsf{lpo}}$ 的定义可知，由于 $\vdash \phi_1 \wedge \phi_2 \succ \vdash \phi_1$，$\mathsf{cpt}(\vdash \phi_1 \wedge \phi_2, c_1, c_2) \succ_{\mathsf{lpo}} \mathsf{cpt}(\vdash \phi_2, c_1, c_2)$ 以及 $\mathsf{cpt}(\vdash \phi_1 \wedge \phi_2, c_1, c_2) \succ_{\mathsf{lpo}} c_2$，因此 $\mathsf{cpt}(\vdash \phi_1 \wedge \phi_2, c_1, c_2) \succ_{\mathsf{lpo}} \mathsf{cpt}(\vdash \phi_1, \mathsf{cpt}(\vdash \phi_2, c_1, c_2), c_2)$;

- 如果 $\phi = \phi_1 \vee \phi_2$，那么由 $\succ_{\mathsf{lpo}}$ 的定义可知，由于 $\vdash \phi_1 \vee \phi_2 \succ \vdash \phi_1$，$\mathsf{cpt}(\vdash \phi_1 \wedge \phi_2, c_1, c_2) \succ_{\mathsf{lpo}} c_1$ 以及 $\mathsf{cpt}(\vdash \phi_1 \vee \phi_2, c_1, c_2) \succ_{\mathsf{lpo}} \mathsf{cpt}(\vdash \phi_2, c_1, c_2)$，因此 $\mathsf{cpt}(\vdash \phi_1 \vee \phi_2, c_1, c_2) \succ_{\mathsf{lpo}} \mathsf{cpt}(\vdash \phi_1, c_1, \mathsf{cpt}(\vdash \phi_2, c_1, c_2))$；
- 如果 $\phi = AX_x(\phi_1)(s)$，那么根据 $\succ_{\mathsf{lpo}}$ 的定义可知，由于 $\Gamma \vdash AX_x(\phi_1)(s) \succ \vdash (s_i/x)\phi_1$ 以及 $\mathsf{cpt}(\Gamma \vdash AX_x(\phi_1)(s), c_1, c_2) \succ_{\mathsf{lpo}} \mathsf{cpt}(\vdash (s_i/x)\phi_1, \mathsf{cpt}(...\mathsf{cpt}(\vdash (s_n/x)\phi_1, c_1, c_2)..., c_2), c_2)$,因此 $\mathsf{cpt}(\Gamma \vdash AX_x(\phi_1)(s), c_1, c_2) \succ_{\mathsf{lpo}} \mathsf{cpt}(\vdash (s_1/x)\phi_1, \mathsf{cpt}(...\mathsf{cpt}(\vdash (s_n/x)\phi_1, c_1, c_2), ..., c_2), c_2)$，其中 $\mathsf{Next}(s) = \{s_1, ..., s_n\}$，而且 $i \in \{1, ..., n\}$；
- 对于 EX 情况的分析与 AX 类似；
- 如果 $\phi = EG_x(\phi_1)(s)$，那么
  - 当 $EG_x(\phi_1)(s) \in \Gamma$ 时，此时与第一种情况类似：$c \succ_{\mathsf{lpo}} c'$；
  - 当 $EG_x(\phi_1)(s) \notin \Gamma$ 时，根据 $\succ_{\mathsf{lpo}}$ 的定义可知，由于 $\Gamma \vdash EG_x(\phi_1)(s) \succ \vdash (s/x)\phi_1$ 以及 $\forall i \in \{1, ..., n\}, \Gamma \vdash EG_x(\phi_1)(s) \succ \Gamma' \vdash EG_x(\phi_1)(s_i)$，其中 $\mathsf{Next}(s) = \{s_1, ..., s_n\}$ 以及 $\Gamma' = \Gamma \cup \{EG_x(\phi_1)(s)\}$；
- 对于 AF 情况的分析与 EG 类似；
- 如果 $\phi = AR_{x,y}(\phi_1, \phi_2)(s)$，那么
  - 当 $AR_{x,y}(\phi_1, \phi_2)(s) \in \Gamma$ 时，此时与第一种情况类似：$c \succ_{\mathsf{lpo}} c'$;
  - 当 $AR_{x,y}(\phi_1, \phi_2)(s) \notin \Gamma$ 时,由 $\succ_{\mathsf{lpo}}$ 的定义可知,由于 $\Gamma \vdash AR_{x,y}(\phi_1, \phi_2)(s) \succ \vdash (s/y)\phi_2$，$\Gamma \vdash AR_{x,y}(\phi_1, \phi_2)(s) \succ \vdash (s/x)\phi_1$ 以及 $\forall i \in \{1, ..., n\}, \Gamma \vdash AR_{x,y}(\phi_1, \phi_2)(s) \succ \Gamma' \vdash AR_{x,y}(\phi_1, \phi_2)(s_i)$，因此 $c \succ_{\mathsf{lpo}} c'$，其中 $\mathsf{Next}(s) = \{s_1, ..., s_n\}$ 以及 $\Gamma' = \Gamma \cup \{AR_{x,y}(\phi_1, \phi_2)(s)\}$；
- 对于 EU 情况的分析与 AR 类似。

□

2.3.3 证明搜索算法的正确性

证明搜索算法的正确性可由以下命题来表示。

命题 2.3.3 (证明搜索算法的正确性). *对于给定闭公式 ϕ，$\mathsf{cpt}(\vdash \phi, \mathfrak{t}, \mathfrak{f}) \rightsquigarrow^* \mathfrak{t}$ 当且仅当 $\vdash \phi$ 是可证的。*

证明. Induction on the structure of ϕ. The details are presented in **??**. □

2.3.4 证明搜索算法的优化

2.3.5 其他 CTL$_P$ 模型检测方法的对比

在这里，我们讨论 SCTLProV 的证明搜索算法与其他 CTL$_P$ 模型检测方法的对比。

2.3.5.1 基于 BDD 的符号模型检测

当 Kripke 模型中绝大多数状态变量是布尔类型（比如在硬件模型检测问题中）的时候，BDD 的应用可以用来减少模型检测算法的空间占用。迄今为止，最好的基于 BDD 的符号模型检测工具是 NuSMV[5,12] 以及 NuSMV 的扩展 NuXMV[4]。下面我们举例说明基于 BDD 的符号模型检测方法的原理：假设存在一个以 s_0 为初始状态，T 为迁移规则的 Kripke 模型 $\mathcal{M}$。若要验证 $\mathcal{M}, s_0 \models EF\phi$，基于 BDD 的符号模型检测工具（例如 NuSMV）会首先计算一个最小不动点 $\mathrm{lfp} = \mu Y.(\phi \vee EXY)$，然后，若 $s_0 \in \mathrm{lfp}$，则 $\mathcal{M}, s_0 \models EF\phi$ 成立，反之则不成立。计算不动点的过程中会不断地对迁移规则 T 进行展开直到得到不动点，其中 s_0 不可达的状态也可能被计算在不动点之内。

与基于 BDD 的符号模型检测工具不同，SCTLProV 在验证过程中没有必要计算不动点：迁移规则 T 是动态展开，即展开 T 直到可以判定公式是否可证为止。SCTLProV 的验证过程只访问初始状态 s_0 可达的状态，因此验证过程会节省空间的占用。不止如此，当 Kripke 模型的状态变量绝大多数为布尔类型的时候，SCTLProV 可以用 BDD 来记录访问过的状态，并以此来进一步节省空间占用；反之，当 Kripke 模型中包含多个非布尔类型的状态变量的时候，SCTLProV 可以选择直接记录（通常用哈希表）访问过的状态。不同于符号模型检测中将 Kripke 结构和要证明的性质都编码到 BDD 的做法，SCTLProV 在 Kripke 结构上直接做状态搜索，BDD 只被用来记录搜索过的状态。

2.3.5.2 动态（On-the-fly）模型检测

在验证时序逻辑公式的正确性时，利用动态模型检测的方法可以避免访问 Kripke 模型的整个状态空间，而只是访问由初始状态可达的状态集合。传统的 CTL 动态模型检测方法[2,20] 通常是基于递归的：即子公式的验证以及迁移规则的展开都是递归进行的。基于递归的 CTL 动态模型检测通常会涉及到大量的栈操作，尤其是在验证大型系统的过程中。验证算法通常会在栈操作上浪费大量的时间。

与传统的 CTL 动态模型检测方法不同，在 SCTLProV 中，公式和迁移规则均按需展开，而且验证算法基于连续（连续传递风格）而不是递归。基于连续的算法只需占用常数大小的栈空间[1,17,19]，而在递归算法中，栈空间的占用大小与递归深度成正比。

由于目前没有完整的基于传统的动态模型检测算法的工具存在，因此，为了将其与 SCTLProV 的证明搜索算法进行对比，我们开发了一个递归版本的 SCTLProV，即 SCTLProV$_R$[1]。不同于 SCTLProV，SCTLProV$_R$ 利用基于递归的算法来证明子公式并搜索状态空间。SCTLProV 与 SCTLProV$_R$ 的实验结果对比见第2.4节。

2.3.5.3 限界模型检测

在传统的限界模型检测工具中，若要证明一个时序逻辑公式，首先需要人为规定（或程序给定）一个限界，并将 Kripke 模型的迁移规则在限界之内展开，然后判断该公式在迁移规则的有限步展开之内满足与否。若在当前的限界之内可以判断公式满足与否，则算法终止；否则，继续扩大限界。举例来说，若要判断 $\mathcal{M}, s_0 \models_{k+1} EF\phi$ 满足与否，则要先在限界 $k+1$ 之内展开公式与迁移规则[3]：

$$[\mathcal{M}, EF\phi]_{k+1} := \bigwedge_{i=0}^{k-1} T(s_i, s_{i+1}) \wedge \bigvee_{j=0}^{k} \phi(s_j)$$

与限界模型检测工具不同，SCTLProV 不需要限界展开迁移规则，而是在展开证明公式的同时按需展开迁移规则。例如在证明 $\vdash EF_x(\phi)(s_0)$ 的时候，公式和迁移规则的展开方式如下：

$$\mathsf{unfold}(S, \vdash EF_x(\phi)(s_i)) := \phi(s_i) \vee ((s_i \notin S) \wedge T(s_i, s_{i+1}) \wedge \mathsf{unfold}(S \cup \{s_i\}, \vdash EF_x(\phi)(s_{i+1})))$$

其中，S 是在证明过程中已经访问过的状态集合。

2.4 案例分析与实验结果

在本节中，我们首先讨论 SCTLProV 的两个应用案例：一个进程互斥问题和一个针对 NASA 提出的小型飞机场控制系统的形式化验证问题；然后，分别对比 SCTLProV 和其他 5 个工具在 5 个测试集上的实验结果。被用于与 SCTLProV 对比的 5 个工具分别为：基于 BDD 的符号模型检测工具 NuSMV 及 NuXMV，基于 QBF 的限界模型检测工具 Verds，基于消解（Resolution）的定理证明工具 iProver Modulo，以及形式验证工具包 CADP。本节所有工具的运行环境均为：Linux 操作系统，内存 3.0GB，2.93GHz×4 CPU；每个测试用例的最大运行时间限制为 20 分钟。本节所有的测试用例均可在因特网[2]下载。

[1] https://github.com/terminatorlxj/SCTLProV_R

[2] https://github.com/terminatorlxj/ctl_benchmarks

2.4.1 案例一：进程互斥问题

例子 2.4.1 (进程互斥问题[16]). 本例讲的是关于两个进程（进程 A 和进程 B）的互斥问题。进程的互斥指的是在程序执行的任意时刻，最多只有一个进程在临界区内。一个关于此类问题的算法描述如图**??**所示，其中 $flag$ 是一个布尔变量，指的是当前是否有进程在运行，而 $mutex$ 是一个整型变量，指的是当前进入临界区的进程个数（初始值为 0）。如果在程序执行的某个时刻有多于 1 个进程进入临界区（即 $mutex = 2$），那么则称该程序违反了进程互斥性质。

```
/* Process A */          /* Process B */
1: while(flag);/*wait*/ 1: while(flag);/*wait*/
2: flag = true;          2: flag = true;
3: mutex ++;             3: mutex ++;
/*critical section*/     /*critical section*/
4: mutex --;             4: mutex --;
5: flag = false;         5: flag = false;
```

图 2.5: 进程 A 和进程 B 的一个简单描述。

如图*2.6*所示，输入文件*"mutual.model"* 中变量 a 和变量 b 分别指代进程 A 和进程 B 的程序计数器（*Program Counter*），变量 ini 指代初始状态。本例中要验证的性质为：是否存在程序执行的某个时刻使得两个进程同时进入临界区。利用 *SCTLProV*，我们可以在计算机的命令行中输入以下命令来验证该性质。

```
sctl -output output.out mutual.model
```

SCTLProV 的运行结果如下所示：该程序存在漏洞，即该程序违反了进程互斥性质。

```
verifying on the model mutual...
find_bug: EU(x,y, TRUE, bug(y), ini)
find_bug is true.
```

该性质的证明树输出在文件*"output.out"* 中，如下图所示。证明树的每个节点被输出为 $id : seqt\ [id_1, ..., id_n]$ 形式，其中 id 是该节点的 ID，$seqt$ 是当前的相继式，而 $id_1, ..., id_n$ 则是当前的相继式的所有的前提的 ID。

```
0: |- EU(x,y,TRUE,bug(y),{flag:=false;mutex:=0;a:=1;b:=1}) [4, 1]
4: {flag:=false;mutex:=0;a:=1;b:=1}
```

```
Model mutual()
{
  Var {
   flag : Bool; mutex : (0 .. 2); a : (1 .. 5); b : (1 .. 5);
  }
  Init {
   flag := false; mutex := 0; a := 1; b := 1;
  }
  Transition {
    a = 1 && flag = false : {a := 2;};
    a = 2 : {a := 3; flag := true;};
    a = 3 : {a := 4; mutex := mutex + 1;}; /*A has entered the critical section*/
    a = 4 : {a := 5; mutex := mutex - 1;}; /*A has left the critical section*/
    a = 5 : {flag := 0;};
    b = 1 && flag = false : {b := 2;};
    b = 2 : {b := 3; flag := true;};
    b = 3 : {b := 4; mutex := mutex + 1;}; /*B has entered the critical section*/
    b = 4 : {b := 5; mutex := mutex - 1;}; /*B has left the critical section*/
    b = 5 : {flag := 0;};
    /*If none of the conditions above are satisfied, then the current state goes to itself.*/
    (a = 1 || b = 1) && flag = true: {}
  }
  Atomic {
    bug(s) := s(mutex = 2);
  }
  Spec {
    find_bug := EU(x, y, TRUE, bug(y), ini);
  }
}
```

图 2.6: 输入文件 “mutual.model”。

```
|- EU(x,y,TRUE,bug(y),{flag:=false;mutex:=0;a:=2;b:=1}) [7, 5]
1: |- TRUE []
7: {flag:=false;mutex:=0;a:=1;b:=1}
{flag:=false;mutex:=0;a:=2;b:=1}
|- EU(x,y,TRUE,bug(y),{flag:=false;mutex:=0;a:=2;b:=2}) [23, 20]
5: |- TRUE []
23:{flag:=false;mutex:=0;a:=1;b:=1}
{flag:=false;mutex:=0;a:=2;b:=1}
{flag:=false;mutex:=0;a:=2;b:=2}
|- EU(x,y,TRUE,bug(y),{flag:=true;mutex:=0;a:=3;b:=2}) [27, 24]
20: |- TRUE []
27:{flag:=false;mutex:=0;a:=1;b:=1}
{flag:=false;mutex:=0;a:=2;b:=1}
{flag:=false;mutex:=0;a:=2;b:=2}
{flag:=true;mutex:=0;a:=3;b:=2}
|- EU(x,y,TRUE,bug(y),{flag:=true;mutex:=1;a:=4;b:=2}) [31, 28]
```

```
24: |- TRUE []
31:{flag:=false;mutex:=0;a:=1;b:=1}
{flag:=false;mutex:=0;a:=2;b:=1}
{flag:=false;mutex:=0;a:=2;b:=2}
{flag:=true;mutex:=0;a:=3;b:=2}
{flag:=true;mutex:=1;a:=4;b:=2}
|- EU(x,y,TRUE,bug(y),{flag:=true;mutex:=1;a:=4;b:=3}) [35, 32]
28: |- TRUE []
35:{flag:=false;mutex:=0;a:=1;b:=1}
{flag:=false;mutex:=0;a:=2;b:=1}
{flag:=false;mutex:=0;a:=2;b:=2}
{flag:=true;mutex:=0;a:=3;b:=2}
{flag:=true;mutex:=1;a:=4;b:=2}
{flag:=true;mutex:=1;a:=4;b:=3}
|- EU(x,y,TRUE,bug(y),{flag:=true;mutex:=2;a:=4;b:=4}) [37]
32: |- TRUE []
37: |- bug({flag:=true;mutex:=2;a:=4;b:=4}) []
```

由以上的证明树输出可知，当进程 A 进入临界区之后，进程 B 同样进入临界区。

如图*2.7*所示，*SCTLProV* 验证该例子时的输出（证明树和 *Kripke* 模型）可由可视化工具 *VMDV*（*Visualization for Modeling, Demonstration, and Verification*）实现三维可视化显示。

通过对原程序进行修改[16]，可以使程序满足进程互斥性质，修改后的程序如图*2.8*所示。修改后的程序可形式化描述为图*2.9*所示的输入文件。在此输入文件中，变量 x 和变量 y 均为布尔变量，分别指代当前状态下进程 A 和进程 B 是否在运行，而 $turn$ 则表示进程 A 和进程 B 轮流处在临界区内。

如下所示，修改后的程序满足进程互斥性质。

```
verifying on the model mutual...
find_bug: EU(x, y, TRUE, bug(y), ini)
find_bug is false.
```

2.4.2 案例二：小型飞机场运输系统

在本小节中，我们介绍对于一个工程问题的形式化验证：由美国国家航空与航天局（National Aeronautics and Space Administration，简称 NASA）为主导提出的小型飞机场运输系统（Small Aircraft Transportation System，简称 SATS）[13,14]。在 SCTLProV 中，我们对 SATS 系统进行形式化描述，并验证该系统的安全性。

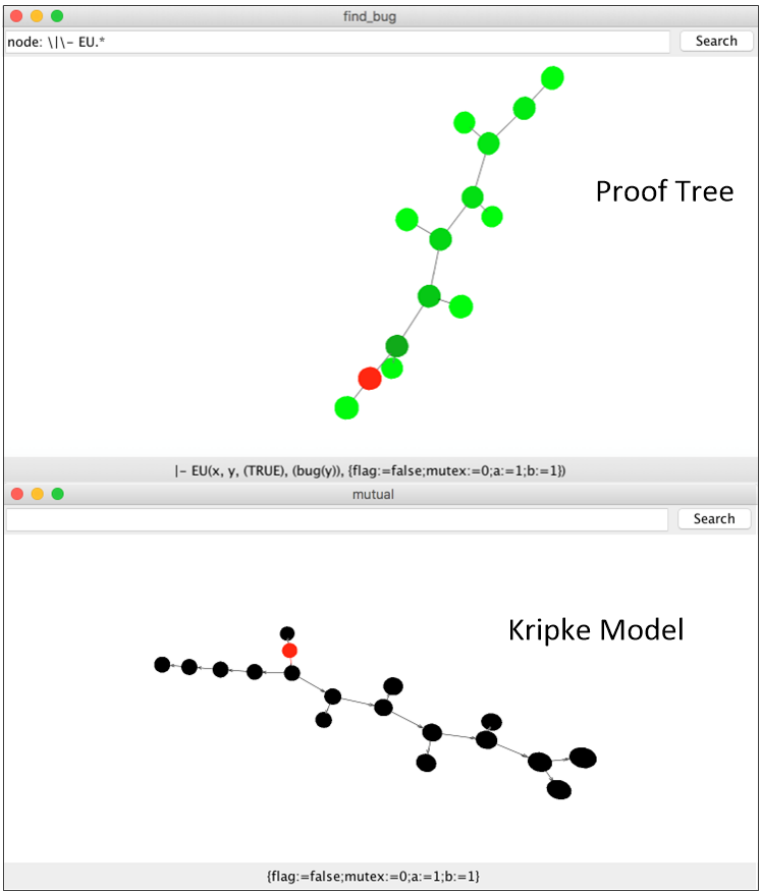


图 2.7: 进程互斥问题的验证中证明树和模型的可视化

```
/* Process A */                  /* Process B */
1: x = true;                     1: y = true;
2: turn = 1;                     2: turn = 2;
3: while(y&&turn!=2); /*wait*/   3: while(x&&turn!=1); /*wait*/
4: mutex ++;                     4: mutex ++;
/*critical section*/             /*critical section*/
5: mutex --;                     5: mutex --;
6: x = false;                    6: y = false;
```

图 2.8: 修改后的进程互斥程序。

在 SATS 系统中，整个飞机场区域被称为自我控制区（Self Control Area，简称 SCA）。如图2.10所示，该模型将 SCA 被分为 15 个子区域：

- **holding3(right/left)**：等待航线，高度 3000 英尺（右/左）；
- **holding2(right/left)**：等待航线，高度 2000 英尺（右/左）；

```
Model mutual()
{
  Var {
    x:Bool; y:Bool; mutex:(0 .. 2); turn:(1 .. 2); a:(1 .. 6); b:(1 .. 6);
  }
  Init {
    x := false; y := false; mutex := 0; turn := 1; a := 1; b := 1;
  }
  Transition {
    a = 1 : {a := 2; x := true;};
    a = 2 : {a := 3; turn := 1;};
    a = 3 && (y = false || turn = 2): {a := 4;};
    /*A has entered the critical section*/
    a = 4 : {a := 5; mutex := mutex + 1;};
    /*A has left the critical section*/
    a = 5 : {a := 6; mutex := mutex - 1;};
    a = 6 : {x := false;};
    b = 1 : {b := 2; y := true;};
    b = 2 : {b := 3; turn := 2;};
    b = 3 && (x = false || turn = 1): {b := 4;};
    /*B has entered the critical section*/
    b = 4 : {b := 5; mutex := mutex + 1;};
    /*B has left the critical section*/
    b = 5 : {b := 6; mutex := mutex - 1;};
    b = 6 : {y := false;};
    /*If none of the conditions above are satisfied,
    then the current state goes to itself.*/
    (a != 3 && (y = true && turn = 1)) || (b != 3 && (x = true && turn = 2)) : {};
  }
  Atomic {
    bug(s) := s(mutex = 2);
  }
  Spec {
    find_bug := EU(x, y, TRUE, bug(y), ini);
  }
}
```

图 2.9: 输入文件“mutual_solution.model”

- **lez(right/left)**：水平降落航线（右/左）；
- **base(right/left)**：基地航线（右/左）；
- **intermediate**：中间航线；
- **final**：最终航线；
- **runway**：机场跑道；
- **maz(right/left)**：重新降落航线（右/左）；
- **departure(right/left)**：起飞航线（右/左）。

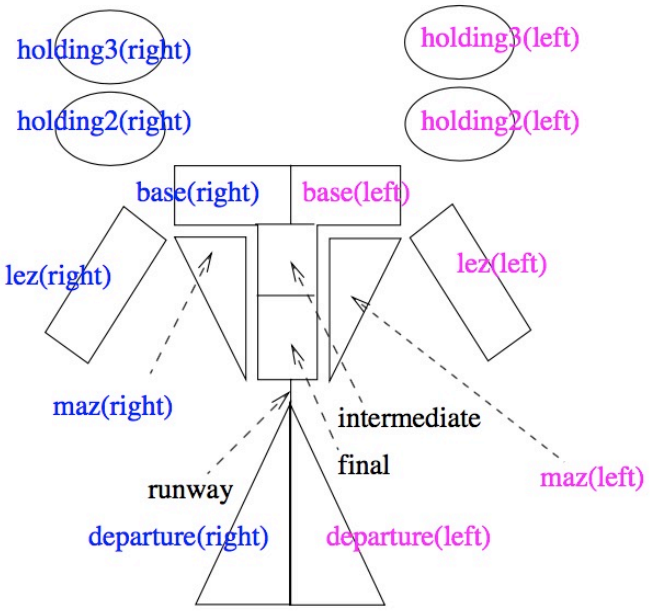


图 2.10: 飞机场自我控制区域的划分（以飞行员视角区分左右）

在任意时刻，SCA 的每个子区域内都有若干架飞机，每个子区域内的飞机遵循先进先出的顺序依次进出。在整个 SCA 中，飞机的进出子区域的方式有 24 种，分别如下：

- 进入 3000 英尺高度等待航线（右/左）；
- 进入水平降落航线（右/左）；
- 从 3000 英尺等待航线进入 2000 英尺等待航线（右/左）；
- 从 2000 英尺等待航线进入基地航线（右/左）；
- 从水平降落航线进入基地航线（右/左）；
- 从基地航线（右/左）进入中间航线；
- 从中间航线进入重新降落航线（右/左）；
- 从中间航线进入最终航线；
- 准备降落，从最终航线进入机场跑道；
- 降落成功，飞机驶离机场跑道；
- 降落失败，从最终航线进入重新降落航线（右/左）；

- 从重新降落航线进入高度最低，而且可以进入（当前等待航线中没有飞机）的等待航线；
- 进入机场跑道并准备起飞；
- 从机场跑道进入起飞航线；
- 从起飞航线（右/左）离开，最终离开 SCA。

在任意时刻，飞机进出 SCA 的子区域的方式必须是安全的，即 SATS 系统必须满足 8 个性质：

- SCA 中不超过 4 架飞机同时准备降落：即等待航线，水平降落航线，重新降落航线，基地航线，中间航线，以及最终航线上飞机的总数不超过 4;
- SCA 中左右两侧的航线（等待航线，水平降落航线，以及重新降落航线）中，同侧的飞机数总和不超过 2，同时 SCA 中不超过 2 架飞机准备从同侧重新降落航线重新降落；
- 进入 SCA 的航线（等待航线和水平进入航线）中，每个航线每侧的飞机数不超过 2，同时基地航线的飞机数总和不超过 3；
- 在每侧的水平进入航线中最多只有一架飞机，同时如果某侧水平进入航线中有飞机，则 SCA 的同侧其他航线（水平降落航线以及重新降落航线）中没有飞机；
- SCA 中的飞机按照事先给定的顺序依次进入 SCA；
- 最先进入 SCA 中飞机一定最先降落；
- 机场跑道上最多只有一架飞机；
- 起飞航线上的飞机彼此必须相隔足够远的距离。

在 SCTLProV 中，我们针对 SCA 建立一个 Kripke 模型：模型中的状态用 15 个状态变量来表示，分别代表 SCA 中 15 个子区域，每个状态变量都是列表类型，代表该子区域内的若干架飞机；模型中包含 24 条迁移规则，分别指代 SCA 中的所有飞机在每个子区域间的 24 种进出方式；要验证的性质是一个 $\mathsf{CTL}_P(\mathcal{M})$ 公式，即在每个状态上，SCA 都满足安全性质。该模型的输入文件可在因特网[3]上下载。

SCTLProV 验证该模型用时 26 秒左右，运行环境为：Linux 操作系统，内存 3.0GB，2.93GHz×4 CPU。验证过程中共访问 54221 个状态（Dowek，Muñoz 和

[3]https://github.com/terminatorlxj/SATS-model

Carreño 提出了的对 SATS 系统的一个简化版的 PVS 建模[13] 中，该模型中可访问的状态数为 2811）。经 SCTLProV 验证，该模型满足安全性性质。

值得注意的是，虽然这是一个典型的模型检测问题，但是传统的模型检测工具均无法验证该模型[13]，理由如下：

1. 模型的状态由复杂的数据结构所表示：每个状态变量的值均为列表类型，而且列表的长度可能为无穷。
2. 状态的迁移规则必须由复杂的算法所描述：在某些状态迁移的过程中需要对飞机列表进行递归操作。
3. 模型的性质必须由复杂的算法所描述：某些原子命题的定义需要对飞机列表进行递归操作。

SCTLProV 的输入语言表达能力强于绝大多数模型检测工具，并能完整的表示该模型，同时成功进行验证。

2.4.3 随机生成的布尔程序的验证

本小节包含三个测试集：测试用例集一在首次提出[21] 时被用作对比限界模型检测工具 Verds 和符号模型检测工具 NuSMV 的性能；并紧接着被用作对比定理证明器 iProver Modulo 与 Verds 的性能；在测试集一的基础上，我们通过增大模型中的状态变量的个数而得到测试集二与测试集三。每个测试集中均包含 2880 个测试用例，每个测试用例的 Kripke 模型都是随机生成的，每个模型中的状态变量绝大多数为布尔类型。大量的随机的测试用例对于 SCTLProV 与不同的工具来说都是相对公平的，而且通过对比不同工具的实验结果数据，我们可以清晰的得出有关各个工具在验证不同的模型以及不同的性质时的优势与劣势的结论。

以下分别介绍这三个测试集。

2.4.3.1 测试集一

测试集一中包含两类测试用例：并发进程（Concurrent Processes，简称 CP）和并发顺序进程（Concurrent Sequential Processes，简称 CSP）。

并发进程 在描述并发进程需要用到以下 4 个变量：

a: 进程个数
b: 所有进程的共享变量和局部变量的个数
c: 进程间共享变量的个数
d: 每个进程的局部变量的个数

进程间的共享变量的初始值均为 $\{0,1\}$ 中的随机值，而每个进程的局部变量的初始值均为 0。每个进程的共享变量和局部变量的每次赋值均为随机选择的某个变量的值的逻辑非。我们令每个测试用例中进程个数为 3，即 $a=3$；令 b 在 $\{12,24,36\}$ 中取值；同时令 $c=b/2$，以及 $d=c/a$。对于每个 b 的取值有 20 个 Kripke 模型，然后在每个 Kripke 模型分别验证 24 个 CTL 性质。因此，此测试集中共有 $3\times 20\times 24=1440$ 个并发进程测试用例。

并发顺序进程 在并发顺序进程测试用例中，除了以上定义的 a,b,c,d 变量之外，描述该类型测试用例还需用到以 2 个变量：

t: 每个进程的迁移的个数
p: 在每个迁移过程中同时进行的赋值的个数

除了在并发进程中介绍的 b 个布尔变量之外，在每个并发顺序进程中还用到一个局部变量来表示进程当前执行的位置，共有 c 个取值。进程间的共享变量的初始值均为 $\{0,1\}$ 中的随机值，而每个进程的局部变量的初始值均为 0。每个进程共有 t 种迁移（状态变换，即对变量的赋值操作），在每个迁移种对随机选择的 p 个共享变量和局部变量进行赋值操作。随着进程的运行，所有的迁移依次周期性地进行。我们令每个测试用例包含 2 个进程，即 $a=2$；令 b 在 $\{12,16,20\}$ 中取值；同时令 $c=b/2, d=c/a, t=c, p=4$。对于每个 b 的取值有 20 个 Kripke 模型，然后在每个 Kripke 模型分别验证 24 个 CTL 性质。因此，此测试集中共有 $3\times 20\times 24=1440$ 个并发顺序进程测试用例。

在本测试集中，我们验证 24 个 CTL 性质，其中性质 P_{01} 至 P_{12} 如图2.11所示，而性质 P_{13} 至 P_{24} 为依次将 P_{01} 至 P_{12} 中的 $\wedge$ 替换成 $\vee$，以及将 $\vee$ 替换成 $\wedge$。

P_{01}	$AG(\bigvee_{i=1}^{c} v_i)$	P_{07}	$AU(v_1, AU(v_2, \bigvee_{i=3}^{c} v_i))$
P_{02}	$AF(\bigvee_{i=1}^{c} v_i)$	P_{08}	$AU(v_1, EU(v_2, \bigvee_{i=3}^{c} v_i))$
P_{03}	$AG(v_1 \Rightarrow AF(v_2 \wedge \bigvee_{i=3}^{c} v_i))$	P_{09}	$AU(v_1, AR(v_2, \bigvee_{i=3}^{c} v_i))$
P_{04}	$AG(v_1 \Rightarrow EF(v_2 \wedge \bigvee_{i=3}^{c} v_i))$	P_{10}	$AU(v_1, ER(v_2, \bigvee_{i=3}^{c} v_i))$
P_{05}	$EG(v_1 \Rightarrow AF(v_2 \wedge \bigvee_{i=3}^{c} v_i))$	P_{11}	$AR(AXv_1, AXAU(v_2, \bigvee_{i=3}^{c} v_i))$
P_{06}	$EG(v_1 \Rightarrow EF(v_2 \wedge \bigvee_{i=3}^{c} v_i))$	P_{12}	$AR(EXv_1, EXEU(v_2, \bigvee_{i=3}^{c} v_i))$

图 2.11: 测试集一中需要验证的性质 P_{01} 至 P_{12}

2.4.3.2 测试集二、三

在测试集一的基础上，我们分别将并发进程测试用例中 b 的值分别扩大为 48，60，以及 72，将并发顺序进程测试用例中 b 的值扩大为 24，28，以及 32。并由此

得到包含 2880 个新的测试用例的测试集二。与测试集一一样，测试集二中的测试用例的模型的初始状态和迁移规则也是随机生成的。测试集二中要验证的性质与测试集一一致。

在测试集三中，我们将并发进程和并发顺序进程测试用例的 b 的值同时进一步扩大到 252，504，以及 1008，同时要验证的性质与测试集一、二保持一致。

2.4.3.3 实验数据

在测试集一、二、三上，我们分别对比了 SCTLProV 与 iProver Modulo、Verds、NuSMV，以及 NuXMV 的实验结果。

测试集一的实验结果 由表??与表2.2可知：在测试集一的 2880 个测试用例中，iProver Modulo、Verds、NuSMV、NuXMV、SCTLProV 分别能验证 1816（63.1%）、2230（77.4%）、2880（100%）、2880（100%）、2862（99.4%）个测试用例；同时 SCTLProV 分别在 2823（98.2%）、2858（99.2%）、2741（95.2%）、2763（95.9%）的测试用例上占用时间和空间少于 iProver Modulo、Verds、NuSMV、NuXMV。各个工具的时间占用随着状态变量的个数的变化趋势如图2.12所示；各个工具占用空间随着状态变量的个数的变化趋势如图2.13所示。

Programs	iProver Modulo	Verds	NuSMV	NuXMV	SCTLProV
CP $(b = 12)$	467(97.3%)	433(90.2%)	480(100%)	480(100%)	480(100%)
CP $(b = 24)$	372(77.5%)	428(89.2%)	480(100%)	480(100%)	480(100%)
CP $(b = 36)$	383(79.8%)	416(86.7%)	480(100%)	480(100%)	470(97.9%)
CSP $(b = 12)$	177(36.9%)	370(77.1%)	480(100%)	480(100%)	480(100%)
CSP $(b = 16)$	164(34.2%)	315(65.6%)	480(100%)	480(100%)	474(98.8%)
CSP $(b = 20)$	253(52.7%)	268(55.8%)	480(100%)	480(100%)	478(99.6%)
Sum	1816(63.1%)	2230(77.4%)	2880(100%)	2880(100%)	2862(99.4%)

表 2.1: 测试集一中 5 个工具能成功验证的测试用例个数

测试集二的实验结果 由表2.3与表2.4可知：在测试集二的 2880 个测试用例中，iProver Modulo、Verds、NuSMV、NuXMV、SCTLProV 分别能验证 1602（55.6%）、1874（65.1%）、728（25.3%）、736（25.6%）、2597（90.2%）个测试用例；同时 SCTLProV 分别在 2597（90.2%）、2594（90.1%）、2588（89.9%）、2588（89.9%）的测试用例上占用时间和空间少于 iProver Modulo、Verds、NuSMV、NuXMV。各个工具的时间占用随着状态变量的个数的变化趋势如图2.14所示；各个工具占用空间随着状态变量的个数的变化趋势如图2.15所示。

Programs	iProver Modulo	Verds	NuSMV	NuXMV
CP $(b = 12)$	480(100%)	480(100%)	430(89.6%)	431(89.8%)
CP $(b = 24)$	480(100%)	480(100%)	456(95.0%)	458(95.4%)
CP $(b = 36)$	454(94.6%)	467(97.3%)	441(91.9%)	446(92.9%)
CSP $(b = 12)$	480(100%)	480(100%)	464(96.7%)	465(96.9%)
CSP $(b = 16)$	474(98.6%)	473(98.5%)	472(98.3%)	474(98.6%)
CSP $(b = 20)$	455(94.8%)	478(99.6%)	478(99.6%)	479(99.8%)
Sum	2823(98.2%)	2858(99.2%)	2741(95.2%)	2763(95.9%)

表 2.2: 测试集一中 SCTLProV 相比其他工具占用资源（时间和空间）少的测试用例个数

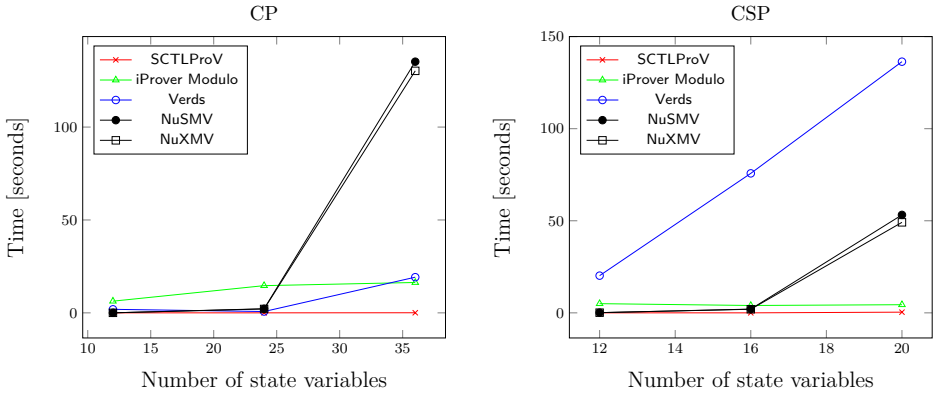


图 2.12: Average verification time in benchmark #1.

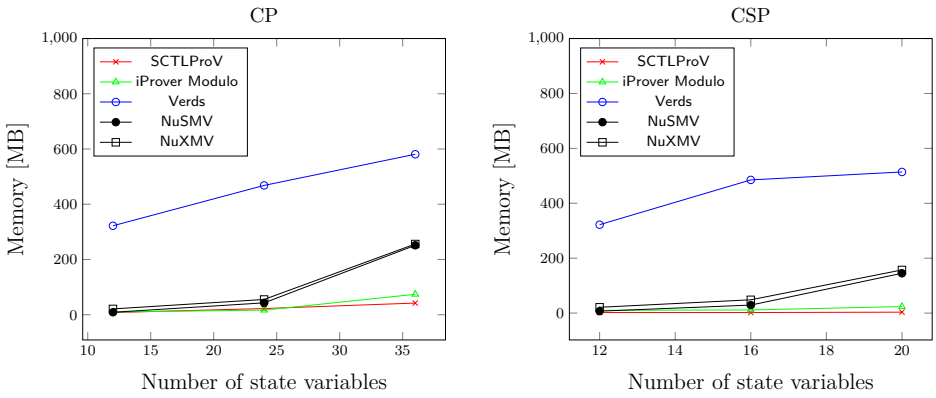


图 2.13: Average memory usage in benchmark #1.

测试集三的实验结果 由表2.5可知：在测试集三的 2880 个测试用例中，iProver Modulo、Verds、SCTLProV 分别能验证 1146（39.8%）、352（12.2%）、1844（64.0%）个测试用例，然而 NuSMV 与 NuXMV 无法验证本测试集中的测试用例。

Programs	iProver Modulo	Verds	NuXMV	NuXMV	SCTLProV
CP $(b = 48)$	375(78.1%)	400(83.3%)	171(35.6%)	176(36.7%)	446(92.9%)
CP $(b = 60)$	360(75.0%)	403(84.0%)	22(4.6%)	23(4.8%)	440(91.7%)
CP $(b = 72)$	347(72.3%)	383(79.8%)	0	0	437(91.0%)
CSP $(b = 24)$	190(39.6%)	235(49.0%)	421(87.7%)	423(88.1%)	430(89.6%)
CSP $(b = 28)$	172(35.8%)	229(47.7%)	106(22.1%)	108(22.5%)	426(88.8%)
CSP $(b = 32)$	158(32.9%)	224(46.7%)	8(1.7%)	6(1.3%)	418(87.1%)
Sum	1602(55.6%)	1874(65.1%)	728(25.3%)	736(25.6%)	2597(90.2%)

表 2.3: 测试集二中 5 个工具能成功验证的测试用例个数

Programs	iProver Modulo	Verds	NuSMV	NuXMV
CP $(b = 48)$	446(92.9%)	444(92.5%)	442(92.1%)	442(92.1%)
CP $(b = 60)$	440(91.7%)	440(91.7%)	440(91.7%)	440(91.7%)
CP $(b = 72)$	437(91.0%)	437(91.0%)	437(91.0%)	437(91.0%)
CSP $(b = 24)$	430(89.6%)	429(89.4%)	426(88.8%)	426(88.8%)
CSP $(b = 28)$	426(88.8%)	426(88.8%)	425(88.5%)	425(88.5%)
CSP $(b = 32)$	418(87.1%)	418(87.1%)	418(87.1%)	418(87.1%)
Sum	2597(90.2%)	2594(90.1%)	2588(89.9%)	2588(89.9%)

表 2.4: 测试集二中 SCTLProV 相比其他工具占用资源（时间和空间）少的测试用例个数

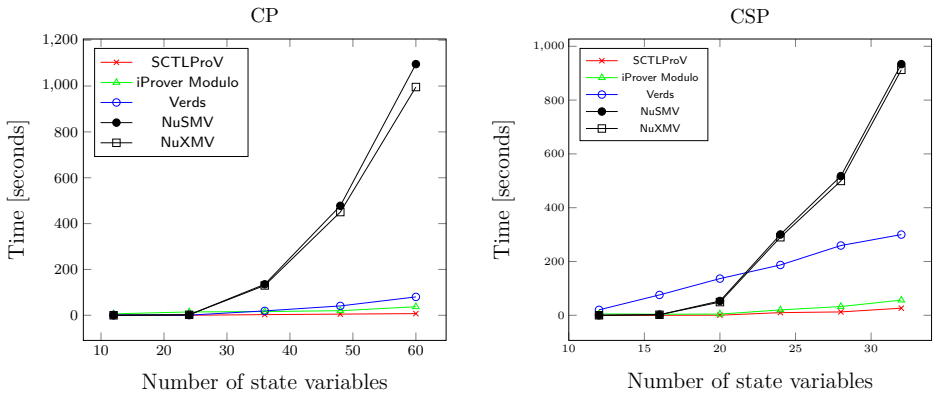


图 2.14: Average verification time in benchmark #1 and #2.

2.4.3.4 连续 vs. 递归

To show the importance of using continuation-passing style, we have implemented a recursive version of our tool and compared the time efficiency. In benchmark #1, #2, and #3, SCTLProV solves about 10% more test cases than SCTLProV$_R$, and it outperforms SCTLProV$_R$ in almost all solvable cases (Table 2.6).

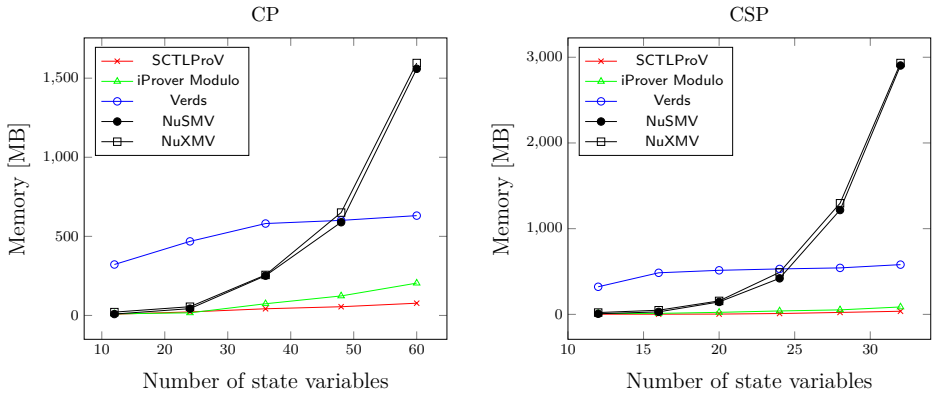


图 2.15: Average memory usage in benchmark #1 and #2.

Programs	iProver Modulo	Verds	NuSMV	NuXMV	SCTLProV
`CP` $(b = 252)$	299(62.3%)	216(45.0%)	0	0	371(77.3%)
`CP` $(b = 504)$	292(60.8%)	0	0	0	335(69.8%)
`CP` $(b = 1008)$	271(56.5%)	0	0	0	278(57.9%)
`CSP` $(b = 252)$	114(23.6%)	136(28.3%)	0	0	312(65.0%)
`CSP` $(b = 504)$	108(22.5%)	0	0	0	295(61.5%)
`CSP` $(b = 1008)$	62(12.9%)	0	0	0	253(52.7%)
Sum	1146(39.8%)	352(12.2%)	0	0	1844(64.0%)

表 2.5: 测试集三中 5 个工具能成功验证的测试用例个数

SCTLProV$_R$ is more sensitive to the number of variables than SCTLProV (Figure 2.16).

In the comparison of average verification time of SCTLProV and SCTLProV$_R$, we extend the number of variables in Concurrent Sequential Processes to 52 and 72, on the basis of benchmark #2.

Bench	SCTLProV solvable	SCTLProV$_R$ solvable	t(SCTLProV) < t(SCTLProV$_R$)
#1	2862(99.4%)	2682(93.1%)	2598(90.2%)
#2	2597(90.2%)	2306(80.1%)	2406(83.5%)
#3	1849(64.2%)	1520(52.8%)	1735(60.2%)

表 2.6: SCTLProV vs. SCTLProV$_R$

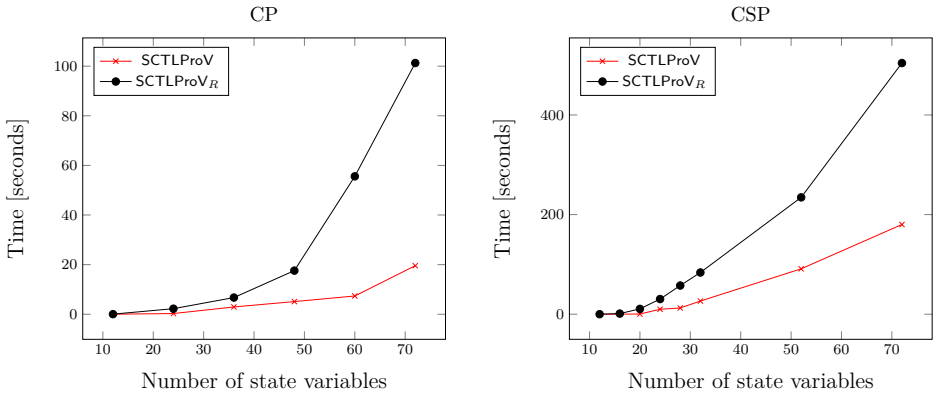


图 2.16: Average verification time in SCTLProV vs. SCTLProV$_R$.

2.4.4 带有公平性性质的程序的验证

In this part, we evaluate benchmark #4, which models mutual exclusion algorithms and ring algorithms[4]. Then, we compare the evaluation results of SCTLProV, Verds, NuSMV, and NuXMV, and we do not consider iProver Modulo because iProver Modulo cannot handle CTL properties with fairness constraints[11].

互斥算法和环算法 Mutual exclusion and ring algorithms. This benchmark consists of two sets of concurrent programs: the mutual exclusion algorithms and the ring algorithms. Both kinds of algorithms consist of a set of concurrent processes running in parallel.

In the mutual exclusion algorithms, the scheduling of processes is simple: for all i between 0 and $n-2$, process $i+1$ performs a transition after process i, and process 0 performs a transition after process $n-1$. Each formula in the algorithms needs to be verified under the fairness constraint that each process does not starve, i.e., no process waits infinitely long.

Each process in the mutual exclusion algorithms has three internal states: noncritical, trying, and critical. The number of processes vary from 6 to 51. There are five properties specified by CTL formulae are to be verified in mutual exclusion algorithms, as in Table 2.17. In these formulae, non_i (try_i, cri_i) indicates that process p_i has internal state noncritical (trying, critical). Note that because of the scheduling algorithm, processes 0 and 1 are not symmetric, as exemplified by the difference in performance between the properties P_4 and P_5.

Each process in the ring algorithms consists of 5 Boolean internal variables indicating the internal state, and a Boolean variable indicating the output. Each

[4] http://lcs.ios.ac.cn/~zwh/verds/verds_code/bp12.rar

Prop	Mutual Exclusion Algorithms
P_1	$EF(cri_0 \wedge cri_1)$
P_2	$AG(try_0 \Rightarrow AF(cri_0))$
P_3	$AG(try_1 \Rightarrow AF(cri_1))$
P_4	$AG(cri_0 \Rightarrow Acri_0U(\neg cri_0 \wedge A\neg cri_0Ucri_1))$
P_5	$AG(cri_1 \Rightarrow Acri_1U(\neg cri_1 \wedge A\neg cri_1Ucri_0))$
Prop	**Ring Algorithms**
P_1	$AGAFout_0 \wedge AGAF\neg out_0$
P_2	$AGEFout_0 \wedge AGEF\neg out_0$
P_3	$EGAFout_0 \wedge EGAF\neg out_0$
P_4	$EGEFout_0 \wedge EGEF\neg out_0$

图 2.17: Properties to be verified in benchmark #4.

process receives a Boolean value as the input during its running time. For a ring algorithm with processes $p_0, p_1, ..., p_n$, the internal state of p_i depends on the output of process p_{i-1}, and the output of p_{i-1} depends on its internal state, where $1 \leq i \leq n$. The internal state of p_0 depends on the output of process p_n, and the output of p_n depends on the internal state of its own. The number of processes vary from 3 to 10. There are four properties specified by CTL formulae are to be verified in ring algorithms, as in Figure 2.17. In these formulae, out_i indicates that the output of process p_i is Boolean value $true$.

The experimental results (Table 2.7 and Table 2.8) show that SCTLProV solves more test cases than Verds, NuSMV, and NuXMV. At the same time, SCTLProV is more time and space efficiency in more than 75 percent of the test cases than the other three tools.

The detailed experimental data is shown in **??**.

2.4.5 工业级测试用例的验证

In this part, we evaluate benchmark #5, also called the VLTS (Very Large Transition Systems) benchmark[5], which was originally proposed as a part of the CADP[6] (Construction and Analysis of Distributed Processes) toolbox[10]. As a formal verification toolbox, CADP focuses on action-based models, for instance, LTSs

[5] http://cadp.inria.fr/resources/vlts/

[6] http://cadp.inria.fr/

Programs	Verds	NuSMV	NuXMV	SCTLProV
`mutual exclusion`	136 (59.1%)	50 (21.7%)	50 (21.7%)	191 (83.0%)
`ring`	16 (50.0%)	21 (65.6%)	21 (65.6%)	20 (62.5%)
Sum	152(58.0%)	71(27.1%)	71(27.1%)	211 (80.5%)

表 2.7: Solvable cases in Verds, NuSMV, NuXMV, and SCTLProV.

Programs	Verds	NuSMV	NuXMV
`mutual exclusion`	187 (81.3%)	191 (83.0%)	191 (83.0%)
`ring`	13 (40.6%)	20 (62.5%)	20 (62.5%)
Sum	200(76.3%)	211(80.5%)	211(80.5%)

表 2.8: Cases where SCTLProV both runs faster and uses less memory.

and Markov Chains. As pointed out by the authors of CADP: *this benchmark has been obtained from the modeling of various communication protocols and concurrent systems, many of which corresponds to real life and industrial systems.*

There are 40 test cases in benchmark #5, where deadlocks and livelocks are to be detected for each test case. All test cases in benchmark #5 are encoded as BCG (Binary-Coded Graphs) format, which is a binary format designed for encoding large state spaces[10]. When verifying test cases in this benchmark in SCTLProV, each BCG file was parsed into a Kripke model, and perform the proving procedure on the Kripke model. We compare the experimental results of detecting deadlocks and livelocks both in SCTLProV and CADP.

The experimental result is depicted in Table 2.9: when detecting deadlocks, SCTLProV uses less time than CADP in 31 (77.5%) test cases, and uses less memory than CADP in 35 (87.5%) test cases; when detecting livelocks, SCTLProV uses less time than CADP in 22 (55%) test cases, and uses less memory than CADP in 27 (67.5%) test cases.

For brevity's sake, more details about the experiment on benchmark #5 are explained in **??**.

Cases	t(SCTLProV) < t(CADP)	m(SCTLProV) < m(CADP)
`deadlock`	31 (77.5%)	35 (87.5%)
`livelock`	22 (55%)	27 (67.5%)

表 2.9: Test cases where SCTLProV uses less time and less memory, respectively.

2.4.6 关于实验结果的讨论

In the evaluation of benchmark #1 to benchmark #4 in this paper, the performances of NuSMV, NuXMV, Verds, iProver Modulo, and SCTLProV in the comparisons are affected by two factors: the number of state variables, and the type of the property to be checked. The performances of NuSMV and NuXMV are mainly affected by the number of state variables, while the performances of iProver Modulo, Verds, and SCTLProV are mainly affected by the type of the property to be checked. When the number of state variables is rather small (such as test cases in benchmark #1), NuSMV and NuXMV solves more test cases than iProver Modulo, Verds and SCTLProV, but when the number of state variables becomes larger (such as test cases in benchmark #2 and #3), they performs worse than the other three tools. When checking properties where nearly all states must be searched (such as AG properties), NuSMV and NuXMV usually perform better than iProver Modulo, Verds and SCTLProV. However, for most properties, iProver Modulo, Verds and SCTLProV usually search much less states than NuSMV and NuXMV to check them, and are more time and space efficiency. Thus, iProver Modulo, Verds and SCTLProV scale up better than NuSMV and NuXMV when checking these properties. Moreover, SCTLProV scales up better than both iProver Modulo and Verds, and outperforms these two tools in most solvable cases.

In the evaluation of benchmark #5, the performances of SCTLProV and CADP are affected by both the size of the state space, and the properties to be verified. The verification processes in SCTLProV and CADP tend to use less time and memory in test cases which have smaller state spaces. For test cases where there exists a deadlock or livelock, the verification processes tend to use less time and memory than those where there are no deadlock or livelock. Moreover, SCTLProV outperforms CADP in more than half of the test cases.

参考文献

[1] APPEL A W, 2006. Compiling with continuations (corr. version)[M]. UK: Cambridge University Press.

[2] BHAT G, CLEAVELAND R, GRUMBERG O, 1995. Efficient on-the-fly model checking for *ctl**[C]//Proceedings of LICS'95. San Diego, California, USA: IEEE Computer Society, USA: 388–397.

[3] BIERE A, CIMATTI A, CLARKE E, et al., 1999. Symbolic model checking without BDDs[C]//CLEAVELAND W R. LNCS: volume 1579 Proceedings of TACAS'99. Amsterdam, the Netherlands: Springer, USA: 193–207.

[4] CAVADA R, CIMATTI A, DORIGATTI M, et al., 2014. The nuxmv symbolic model checker[C]//Proceedings of Computer Aided Verification - 26th International Conference, CAV 2014. Vienna, Austria: Springer International Publishing, Switzerland: 334–342.

[5] CIMATTI A, CLARKE E M, GIUNCHIGLIA F, et al., 1999. Nusmv: A new symbolic model verifier[C]//Proceedings of CAV'99. Trento, Italy: Springer-Verlag, Berlin: 495–499.

[6] CRAIG J J, 1989. Introduction to robotics - mechanics and control (2. ed.)[M]. USA: Prentice Hall.

[7] DERSHOWITZ N, 1987. Termination of rewriting[J/OL]. J. Symb. Comput., 3(1/2): 69–116. https://doi.org/10.1016/S0747-7171(87)80022-6.

[8] EMERSON E A, CLARKE E M, 1982. Using branching time temporal logic to synthesize synchronization skeletons[J]. Sci. Comput. Program., 2(3): 241–266.

[9] EMERSON E A, HALPERN J Y, 1985. Decision procedures and expressiveness in the temporal logic of branching time[J]. J. Comput. Syst. Sci., 30(1): 1–24.

[10] GARAVEL H, LANG F, MATEESCU R, et al., 2013. CADP 2011: a toolbox for the construction and analysis of distributed processes[J/OL]. STTT, 15(2): 89–107. https://doi.org/10.1007/s10009-012-0244-z.

[11] JI K, 2015. CTL Model Checking in Deduction Modulo[C]//Proceedings of Automated Deduction - CADE-25. Berlin: Springer International Publishing, Switzerland: 295–310.

[12] MCMILLAN K L, 1993. Symbolic model checking[M]. USA: Springer.

[13] MUÑOZ C A, DOWEK G, CARREÑO V, 2004. Modeling and verification of an air traffic concept of operations[C]//Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2004. Boston, Massachusetts, USA: ACM, USA: 175–182.

[14] NASA/TM-2004-213006, 2004. Abstract model of sats concept of operations: Initial results and recommendations[M]. USA: NASA.

[15] PARTOVI A, LIN H, 2014. Assume-guarantee cooperative satisfaction of multi-agent systems[C]//Proceedings of American Control Conference, ACC 2014. USA: IEEE, USA: 2053–2058.

[16] PETERSON G L, 1981. Myths about the mutual exclusion problem[J]. Inf. Process. Lett., 12(3): 115–116.

[17] REYNOLDS J C, 1993. The discoveries of continuations[J]. Lisp and Symbolic Computation, 6(3-4): 233–248.

[18] SARNAT J, SCHÜRMANN C, 2009. Lexicographic path induction[C]//Proceedings of Typed Lambda Calculi and Applications, 9th International Conference, TLCA 2009. Brasilia, Brazil: Springer, Berlin: 279–293.

[19] SESTOFT P, 2012. Undergraduate topics in computer science: volume 50 programming language concepts[M]. Switzerland: Springer International Publishing.

[20] VERGAUWEN B, LEWI J, 1993. A linear local model checking algorithm for CTL [C]//Proceedings of CONCUR '93, 4th International Conference on Concurrency Theory. Hildesheim, Germany: Springer-Verlag, Berlin: 447–461.

[21] ZHANG W, 2014. QBF Encoding of Temporal Properties and QBF-based Verification [C]//Proceedings of IJCAR 2014. Vienna: Springer-Verlag, Berlin: 224–239.

发表学术论文

学术论文

[1] Jian Liu, Ying Jiang, Yanyun Chen. VMDV: A 3D Visualization Tool for Modeling, Demonstration, and Verification. TASE 2017. accepted.

[2] Ying Jiang, Jian Liu, Gilles Dowek, Kailiang Ji. SCTL: Towards Combining Model Checking and Proof Checking. The Computer Journal. submitted.

项目资助情况

中法合作项目 LOCALI (项目编号 NSFC 61161130530 和 ANR 11 IS02 002 01)。

简历

基本情况

刘坚，男，山东省茌平县人，1989 年出生，中国科学院软件研究所博士研究生。

教育背景

- 2011 年 9 月至今：中国科学院软件研究所，计算机软件与理论，硕博连读
- 2007 年 9 月至 2011 年 7 月：山东农业大学，信息与计算科学，本科

联系方式

通讯地址：北京市海淀区中关村南四街 4 号，中国科学院软件研究所，5 号楼 3 层计算机科学国家重点实验室

邮编：100090

E-mail: dreammaker2010@yeah.net

致　谢

值此论文完成之际，谨在此向多年来给予我关心和帮助的老师、学长、同学、朋友和家人表示衷心的感谢！