中国科学院大学

University of Chinese Academy of Sciences

博士学位论文

基于模型检测和定理证明的形式化验证：基础理论与相关工具

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 6 月

Towards Combing Model Checking and Theorem Proving: Theory and Related Tools

A dissertation submitted to the
University of Chinese Academy of Sciences
in partial fulfillment of the requirement
for the degree of
Doctor of Philosophy
in Computer Software and Theory

By

Jian Liu

Supervisor: Professor Ying Jiang

Institute of Software, Chinese Academy of Sciences

June, 2018

中国科学院大学
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。本人完全意识到本声明的法律结果由本人承担。

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大学有关保存和使用学位论文的规定，即中国科学院大学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：　　日　　期：

摘 要

模型检测和逻辑推演是目前用于形式化验证系统正确性的主要的两种方法。模型检测的优点是可以实现完全自动化，验证过程不需要人工干预，缺点是在处理大型系统或者无穷状态系统的过程中会面临状态爆炸问题；逻辑推演的优点是通常不关心系统的状态，从而避免了状态爆炸问题的产生，而缺点则是通常不能实现完全自动化，在验证的过程中需要人工干预。因此，如何结合模型检测和逻辑推演两种方法的优点来建立新的形式化验证方法是本文的主要研究内容。

本文的第一个工作是，建立一个逻辑系统 CTL_P，CTL_P 以 Kripke 模型为参数，对于一个给定的 Kripke 模型 $\mathcal{M}$，一个 $\mathrm{CTL}_P(\mathcal{M})$ 公式是有效的当且仅当这个公式在 M 中是满足的。相比于在 CTL 逻辑中只能讨论模型中当前状态的性质，在 CTL_P 中我们可以定义模型中的状态之间的关系，从而丰富了 CTL 逻辑的表达性；然后，我们根据逻辑 CTL_P 建立了一个证明系统 SCTL（Sequent-calculus-like proof system for CTL_P），并证明了 SCTL 系统的可靠性和完备性，使得一个公式在 SCTL 中是可证的当且仅当它在给定的模型中是满足的。

本文的第二个工作是，提出了一个 SCTL 证明系统的工具实现 SCTLProV，SCTLProV 既可以看作为定理证明器也可看作为模型检测工具：相比于定理证明器，SCTLProV 可以应用更多的优化策略，比如利用 BDD（Binary Decision Diagram）来存储状态集合从而减小内存占用；相比于模型检测工具，SCTLProV 的输出是完整的证明树，比模型检测工具的输出更丰富。

本文的第三个工作是，实现了一个 3D 证明可视化工具 VMDV（Visualization for Modeling, Demonstration, and Verification）。目前，VMDV 可以完整的显示定理证明工具 SCTLProV 的输出（证明树以及 Kripke 模型）以及高亮显示 SCTLProV 的证明过程。与此同时，VMDV 是一个一般化的证明可视化工具，并提供了一个统一的接口用来与不同的定理证明器（比如 Coq）协同工作。

关键词：形式化验证，模型检测，定理证明，工具实现

Abstract

Model checking and automated theorem proving are two pillars of formal methods. The advantage of model checking is its complete automaticity, while the main obstacle is the state-explosion problem. Theorem proving usually does not enumerate the states of a system, thus does not have the state-explosion problem. However, theorem proving usually cannot be fully automatic. The main content of this thesis is to investigate model checking from an automated theorem proving perspective, aiming at combining the expressiveness of automated theorem proving and the complete automaticity of model checking.

The first work of the thesis is to build a logic CTL_P, which is a slight extension of the Computation Tree Logic (CTL). Then, we build a sequent-calculus-like proof system for CTL_P, called SCTL. SCTL is parameterized by a Kripke model, and unlike the usual sequent calculus, a CTL_P formula is provable in SCTL($\mathcal{M}$) if and only if the formula is valid in the given Kripke model $\mathcal{M}$. The soundness and completeness of SCTL are also proved. In order to search proofs efficiently, we present a proof search method for SCTL, which is based on the notion of continuation, and a proof search algorithm based on the proof search method.

The second work of the thesis is the implementation of a theorem prover, called SCTLProV. For a given input file, which consists of a Kripke model and its properties specified, SCTLProV searches for a proof for each of the properties for the given Kripke model using the proof search algorithm presented. SCTLProV can be seen as either a theorem prover, or a model checker. Comparing with traditional theorem provers, SCTLProV can use more optimization strategies in the model checking of CTL properties, such as using BDD to store visited states to reduce space occupation; comparing with traditional model checkers, the output of SCTLProV is much more expressive: instead of expressing counterexamples as sequences of states when the verified property does not hold, SCTLProV outputs a proof tree of the given property when the verification succeeds, and a proof tree of the negation of the given property otherwise.

The third work of the thesis is an implementation of a 3D visualization tool, called VMDV. Using VMDV, we can visualize the proof produced by SCTLProV and the

Kripke model for a better comprehension of the verification results. Furthermore, VMDV provides a generalized protocol for integration with various kinds of theorem provers, such as Coq.

Keywords: Formal Verification, Model Checking, Theorem Proving, Implementation

目 录

图形列表

表格列表

第 1 章　引言

1.1　选题背景和意义

近年来，计算机软硬件系统的飞速发展极大地提高了人类的生产效率，也极大地方便了人类的生活。但与此同时，由于软硬件故障而导致的人身以及生产事故同样不可忽视，尤其是对于安全攸关的系统，比如航空航天系统、高速铁路控制系统、医疗卫生系统等，这些系统的故障往往会导致灾难性的后果。比如，1994 年，Intel 公司由于其发售的奔腾系列处理器的指令集出现漏洞而宣布召回所有缺陷产品，并因此造成 4.75 亿美元的经济损失。1996 年，由于飞行控制程序中存在由 64 位浮点数到 16 位整数的错误转换，阿丽亚娜 5 号运载火箭在首次发射中仅仅飞行 37 秒就发生爆炸，并造成有史以来损失最惨重的航天事故。因此，随着软硬件系统的复杂度的提高，如何确保系统的正确性成为系统开发面临的关键性问题之一。

传统的计算机软硬件系统的开发一般依靠仿真和测试的方法来评估系统是否满足设计要求。基于仿真和测试的方法在系统开发的初始阶段可以很大地提高开发效率，并便于即时调试。随着技术的不断发展和完善，此类方法已经可以越来越方便地发现系统设计的一些隐匿的错误。由于其高效性，现如今的大量软硬件系统的开发过程依然采用此类方法。然而，基于仿真和测试的方法往往无法确保找到系统的所有错误，因而无法保证系统的正确性。

除了基于仿真和测试的方法，对系统正确性的评估还可通过形式化验证来进行。与基于仿真和测试的方法不同，通过形式化方法验证过的系统通常可确保其正确性。在利用形式化方法验证系统性质的过程中，系统的所有状态或行为通常用形式化描述语言来定义，然后分别利用数学的方法枚举由形式语言所定义的所有的系统状态，直到得出验证结论。

1.2　形式化方法

在众多的形式化验证方法中，模型检测和定理证明是最被普遍应用的两种。

模型检测　20 世纪 80 年代，模型检测[1–3] 最早由 Clark 和 Emerson 等人提出，并很快得到迅猛发展。在模型检测中，被验证的系统通常被抽象为一个有限状态模型，模型中的有限个状态对应于系统的状态，同时模型中的状态迁移通常用来描

述系统在运行过程中的状态变化。系统的性质通常由时序逻辑公式来描述。在模型检测中，验证系统的性质通常指的是判定时序逻辑公式在有限状态模型上的可满足性。模型检测工具通常会通过遍历状态空间来验证公式的可满足性。当需要验证的公式是可满足的时候，模型检测工具会返回 true；反之，当需要验证的公式不可满足时，模型检测工具会产生一个反例用来定位系统的错误。近年来，随着技术和理论的不断发展，一大批优秀的模型检测工具不断地涌现出来，其中被广泛应用的模型检测工具以符号模型检测工具 NuSMV[4]、限界模型检测工具 Verds[5]、即时模型检测工具 SPIN[6] 以及形式化验证工具集 CADP[7] 为代表。

- NuSMV：NuSMV 的提出是基于最早由 McMillan 在其博士论文[8] 中提出的符号模型检测方法。符号模型检测主要应用于对计算树逻辑（Computation Tree Logic，简称 CTL）公式的验证，符号模型检测最大的特点在于将要验证的性质看成系统状态的集合，而状态集合通常用 BDD[9] 来进行表示。一个 CTL 公式的可满足行则表示为系统的初始状态是否属于该状态集合。在系统的验证过程中，系统的状态空间不是显示地生成出来，而是由 BDD 来符号化地表示，这种状态集合的表示方式在表达相同数量的状态时相比显示表示每个状态的方式通常占用更少的空间。

- Verds：限界模型检测工具 Verds 由 Zhang[5] 提出，不同于符号模型检测工具，Verds 在验证公式的过程中首先规定一个界，然后在界内搜索模型的状态。在实际的验证过程中，相比于 NuSMV，Verds 通常只需访问较少的状态就能完成公式的验证。另外，不同于基于 SAT[10] 的限界模型检测工具，Verds 在验证过程中将时序逻辑公式翻译到一个 QBF（Quantified Boolean Formula），相比于 SAT 公式，QBF 可用来表示更加复杂的时序逻辑公式。

- SPIN：SPIN 是在 1980 年由美国贝尔实验室的 Holzmann 等人编写，并在 1991 年免费开放的一种即时模型检测工具。不同于 NuSMV 和 Verds，SPIN 更多的时候被用来验证分布式系统模型的线性时序逻辑（Linear Temporal Logic，简称 LTL）性质。在 SPIN 的验证过程不需要提前规定模型搜索的界，而是通过深度优先的方式逐个访问模型的状态，直到可以计算出 LTL 公式的可满足性为止。另外，不同于 NuSMV 和 Verds，模型的状态在 SPIN 中通常显式存储，不过由于需要访问的状态通常比前两种工具少，因此在某些模型的验证中空间占用比前两种工具更少。

- CADP：CADP 全称 Construction and Analysis of Distributed Processes，是一

系列用于通讯协议和分布式系统设计和验证的工具集合。CADP 目前由法国国家信息与自动化研究所的 CONVECS 小组开发（起初由 VASY 小组开发），并在工业界得到广泛应用。不同于验证 Kripke 模型的工具，CADP 主要用来验证基于动作的形式化系统，比如符号迁移系统、马尔可夫链等。CADP 工具集中包含可验证多种时序逻辑公式的工具，并可采用即时模型检测或符号模型检测的状态搜索策略。

模型检测的主要优点是验证过程完全自动化，主要缺点是状态爆炸问题，尤其是在对多进程的并发系统的验证过程中，随着进程数量的增长，系统状态的数量可能呈现指数级的增长。状态爆炸问题是困扰模型检测在大型系统的验证中应用的主要问题。

定理证明 不同于基于状态的模型检测方法，定理证明[11–13]是基于公理与推理规则的。在利用定理证明方法形式化验证系统的性质时，被验证的系统通常在一种数学语言中被表述为一系列数学定义，而要验证的性质则被表述为数学逻辑公式，然后从公理出发，利用推理规则寻找逻辑公式的形式化证明。除了验证过程，定理证明工具的验证结果也与模型检测方法不同：定理证明的验证结果一般是以证明树的形式给出，比模型检测的验证结果包含更详细的信息。在应用定理证明方法进行形式化验证时，定理证明工具的产生极大地增强了证明的效率。根据基于的数学逻辑的不同，定理证明工具可大体被分为以下几类：

- **经典高阶逻辑**：经典高阶逻辑的表达能力较强，通常可方便地用来描述复杂的系统行为和性质。然而，较好的表达性也意味着验证的自动化程度较低。基于经典高阶逻辑的代表性定理证明工具有 HOL[14]、Isabelle/HOL[15]、PVS[16] 等。

- **构造逻辑**：由柯里-霍华德同构[17]可知，一个公式的构造性证明一一对应于一个可执行的计算机算法。因此，在自动化程度上，基于构造逻辑的定理证明工具相比基于经典高阶逻辑的工具更好。但是，公式的构造性证明往往比较难以找到。基于构造逻辑的代表性定理证明工具有 Coq[18]、Nuprl[19]等。

- **一阶逻辑**：在表达能力上，基于一阶逻辑的定理证明工具通常可以完成绝大多数系统的验证，同时此类定理证明器的自动化程度较高。基于一阶逻辑的代表性定理证明工具有 ACL2[20]、Isabelle/FOL[21] 等。

定理证明的优点在于表达能力强，而且验证过程不是基于状态的，因此可以被用来验证大型系统，甚至是无穷状态系统。不过，相比于模型检测，定理证明的缺点就是证明过程通常需要人工干预，而不是完全自动化的。

1.3 相关工作

模型检测和定理证明的优缺点是互补的：模型检测是完全自动化的，而定理证明通常是半自动化的；定理证明能对复杂的或者具有无穷状态的系统进行建模和验证，而模型检测通常无法验证大型或者无穷状态系统。因此，如何将这两种形式化验证方法相结合成为近年来的热门研究课题。截至目前，国际上已经有多个试图结合两种验证方法的工作。

- Rajan，Shankar 和 Srivas 提出了一种基于 PVS 的模型检测和定理证明的结合方法[22]。在这种方法中，作者首先在 PVS 中定义了一个 μ-演算理论，并用 μ-演算定义了时序逻辑 CTL 的模态词，当证明 CTL 公式的时候，PVS 则会调用调用内置的模型检测命令来完成证明。在这种方法中，模型检测方法被用来完成 PVS 证明的子目标，而不是贯穿于整个证明过程。这种将模型检测和定理证明相结合方式的最大优点是利用了定理证明器强大的表达和抽象能力将系统抽象成一个有穷状态模型，然后利用模型检测算法在有穷状态模型中完成状态搜索。

- 在证明 PLTL 公式方面，Cavalli 和 Cerro 给出了一种基于经典的 resolution 的方法[23]。在这种方法中，作者将 PLTL 公式翻译到一种时序逻辑的合取范式，然后定义了一组这种合取范式的 resolution 规则。与这种方法类似，Venkatesh 提出了一种基于 resolution 的判定 LTL 公式可满足性的方法[24]。这种方法的特点是将给定的 LTL 公式转换成与该公式的可满足性等价的带有嵌套模态词的公式，然后在转换后的公式上应用经典的 resolution 规则。同样基于 resolution 的方法还有 Fisher，Dixon 和 Peim 提出的证明 LTL 公式有效性的方法[25]，以及 Zhang，Hustadt 和 Dixon 提出的证明 CTL 公式有效性的方法[26] 等。

- Ji 提出了一种基于 resolution modulo 的用于验证 CTL 公式有效性的方法[27]。Resolution modulo 方法首次由 Dowek，Hardin 和 Kirchner 提出[28,29]，基本思想是将一阶逻辑的 resolution 方法转换成重写系统，因此公式的推导过程更多地用计算（重写）来实现。Ji 的方法已经在定理证明器 iProver Modulo

中实现，并在与现有的模型检测工具（NuSMV 和 Verds）的实验结果对比中表现良好。

1.4 本文的内容和组织结构

本文的主要内容是研究模型检测和定理证明两种方法的结合，并不同于已有的工作，本文既没有将模型检测作为定理证明的一个判定过程，也没有将已有时序逻辑公式的模型检测问题转化为定理证明问题，而是提出基于传统时序逻辑的一个扩展，并在保持模型检测完全自动化优点的基础上可以验证更复杂的性质。本论文内容最初基于蒋颖教授与 Gilles Dowek 教授的工作[30]。本文的主要内容如下：

1. 建立一个针对 CTL 逻辑的扩展：CTL_P。相比于 CTL，在 CTL_P 中可以定义多元谓词。除了可以定义状态的时序性质之外，利用多元谓词可以表达某些“空间”性质，即多个状态之间的关系。针对 CTL_P，我们还定义了一个证明系统 SCTL 用来表达 CTL_P 公式的证明，并证明了该证明系统的可靠性和完备性。除此之外，我们还提出了一种针对 SCTL 系统的证明搜索算法并证明了该算法的终止性和正确性。

2. 给出了一个对证明系统 SCTL 的编程实现—定理证明工具 SCTLProV。在不丢失效率的前提下，SCTLProV 可被用来实现复杂系统的验证，SCTLProV 的输出结果相比于传统的模型检测工具更丰富。为了对比 SCTLProV 与现有工具的效率，我们分别在若干个随机生成以及工业级测试用例集上与多个业界顶尖的形式化验证工具进行实验数据的对比。

3. 设计并实现了定理证明可视化工具 VMDV，并将 VMDV 用于 SCTLProV 的证明输出的可视化。同时 VMDV 还被设计并实现为一般化的证明可视化工具，并可实现不同的定理证明工具输出的可视化。

除本章外，本文其余章节的组织结构如下：

第2为预备知识。

第3章介绍了 CTL_P 逻辑和证明系统 SCTL，以及针对 SCTL 的证明搜索算法。证明了 SCTL 证明系统的可靠性和完备性，以及证明搜索算法的终止性和正确性。证明搜索算法的伪代码也在本章给出。

第4章介绍了定理证明工具 SCTLProV 的架构和实现细节，并在多个测试用例集上与多个形式化验证工具进行实验结果对比。同时给出 SCTLProV 验证复杂系统（模型检测工具无法处理）的一个案例的分析。

第5章介绍了定理证明可视化工具 VMDV 的架构和实现细节。

第7章是全文总结和对未来工作的展望。

第 2 章　预备知识

本章介绍的是本文中用到的一些相关的预备知识。

2.1　模型检测的一般流程

如图2.1所示，利用模型检测方法来对计算机软硬件系统进行形式化验证的一般流程可总结为：

1. 针对要验证的软硬件系统建立一个形式化模型；
2. 将要验证的性质用一种逻辑公式表示出来；
3. 利用算法来判断要验证的性质是否在给定的形式化模型上是可满足的，如果要验证的性质在给定的形式化模型上可满足，则输出 $True$，否则输出 $False$ 并给出反例来说明性质不满足的原因。

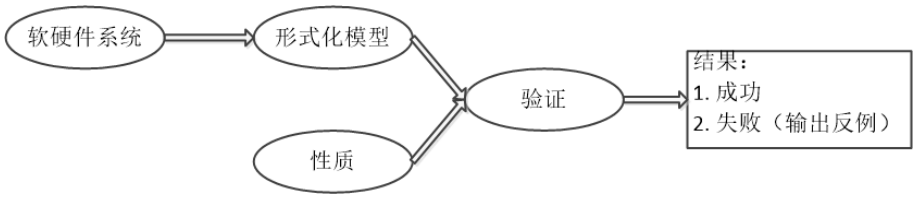


图 2.1 模型检测流程

2.1.1　形式化模型

在模型检测中，要验证软硬件系统的性质，首先必须针对该系统建立一个形式化模型。在这里，一个形式化模型通常指的是一个 Kripke 模型。模型检测中的经典的 Kripke 模型的定义如下：

定义 2.1 (经典的 Kripke 模型)**.** 设 AP 是一个有穷的原子命题集合，AP 上的一个经典的 *Kripke* 模型可表示为一个四元组 $M = (S, S_0, \longrightarrow, L)$，其中

1. S 是一个有穷状态集合；
2. $S_0 \subseteq S$ 是初始状态集合；
3. $\longrightarrow \subseteq S \times S$ 是一个二元关系，同时对 $\forall s \in S$，都存在一个状态 $s' \in S$ 使得 $s \longrightarrow s'$；

4. $L : S \to 2^{AP}$ 是一个标签函数，L 将 S 中的每个状态都映射到一个 AP 的子集。

在 Kripke 模型 M 中，我们将一条从状态 s 出发的一条无穷序列 $\pi = s_0, s_1, \ldots$ 称为一条路径，其中 $s = s_0$，而且对于任意 $i \geq 0$，都有 $s_i \longrightarrow s_{i+1}$。

2.1.2 性质

Kripke 模型的性质通常用时序逻辑公式来表示，而根据所需要描述的系统性质类型的不同，模型检测中常用的时序逻辑有线性时序逻辑 LTL[31]、计算树逻辑 CTL[32] 以及 LTL 与 CTL 的超集 CTL*[33]。本文着重介绍计算树逻辑 CTL 以及基于 CTL 的扩展。

定义 2.2 (计算树逻辑 CTL). 设 AP 是一个有穷的原子命题集合，那么基于 AP 的所有 *CTL* 公式可表示为：

$$\phi ::= \top \mid \bot \mid P \mid \neg\phi \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \mid AX\phi \mid EX\phi \mid AF\phi \mid EG\phi \mid A[\phi_1 R \phi_2] \mid E[\phi_1 U \phi_2]$$

其中 $\top$ 表示永真，$\bot$ 表示永假，$P \in AP$ 是一个原子命题。

然后，我们可以在经典的 Kripke 模型上定义 CTL 的语义。

定义 2.3 (CTL 的语义). 设 $M = (S, S_0, \longrightarrow, L)$ 是在 AP 上的一个经典的 *Kripke* 模型。我们用 $M, s \models \phi$ 来表示 *CTL* 公式 ϕ 在状态 s 上满足。$\models$ 的归纳定义如下。

- $M, s \models \top$ 永远成立；
- $M, s \models \bot$ 永远不成立；
- $M, s \models P$ 当且仅当 $p \in L(s)$；
- $M, s \models \neg\phi$ 当且仅当 $M, s \not\models \phi$；
- $M, s \models \phi_1 \wedge \phi_2$ 当且仅当 $M, s \models \phi_1$ 且 $M, s \models \phi_2$；
- $M, s \models \phi_1 \vee \phi_2$ 当且仅当 $M, s \models \phi_1$ 或 $M, s \models \phi_2$；
- $M, s \models AX\phi$ 当且仅当 $\forall s' \in \{s' \in S \mid s \longrightarrow s'\}$，$M, s' \models \phi$；
- $M, s \models EX\phi$ 当且仅当 $\exists s' \in \{s' \in S \mid s \longrightarrow s'\}$，$M, s' \models \phi$；

- $M, s \models AF\phi$ 当且仅当对于任意从 s 出发的路径 $s_0, s_1, \ldots$ 都存在 $i \geq 0$ 使得 $M, s_i \models \phi$；
- $M, s \models AF\phi$ 当且仅当存在从 s 出发的路径 $s_0, s_1, \ldots$ 使得对于任意 $i \geq 0$ 都有 $M, s_i \models \phi$；
- $M, s \models A[\phi_1 R \phi_2]$ 当且仅当对于任意从 s 出发的路径 $s_0, s_1, \ldots$：如果存在 $j \geq 0$ 使得 $M, s_j \models \phi_1$ 成立，那么对于所有的 $0 \leq i \leq j$ 都有 $M, s_i \models \phi_2$ 成立；否则，对任意 $i \geq 0$ 都有 $M, s_i \models \phi_2$ 成立。
- $M, s \models E[\phi_1 U \phi_2]$ 当且仅当存在从 s 出发的路径 $s_0, s_1, \ldots$ 使得：存在 $j \geq 0$ 使得 $M, s_j \models \phi_2$ 成立且对于所有的 $0 \leq i < j$ 都有 $M, s_i \models \phi_1$ 成立；

2.1.3 验证

对于给定的 Kripke 模型 M，状态 s 和 CTL 公式 ϕ，CTL 模型检测的验证过程即判断 $M, s \models \phi$ 是否成立。如果要验证的性质在给定的形式化模型上可满足，则输出 $True$，否则输出 $False$ 并给出反例来说明性质不满足的原因。对于以“$A*$”形式模态词开头的 CTL 公式，一个反例指的是一条路径。比如如果 $M, s \not\models AF\phi$，那么模型检测工具给出的反例是一条无穷路径，其中对于该无穷路径上的任意状态 s' 都有 $M, s' \not\models \phi$。对于以“$E*$”形式模态词开头的 CTL 公式，一个反例指的是一系列路径。比如如果 $M, s \not\models E[\phi_1 U \phi_2]$，那么模型检测工具给出的反例是所有满足以下条件的路径：即以 s 为起始状态，而且如果该路径为有穷路径，那么对于该路径上的最后一个状态 s' 有 $M, s' \not\models \phi_1$ 且 $M, s' \not\models \phi_2$，同时对于该路径上的任意其他状态（如果存在）s'' 有 $M, s'' \models \phi_1$；如果该路径为无穷路径，那么对于该路径上的任意状态 s' 有 $M, s' \models \phi_1$ 且 $M, s' \not\models \phi_2$。另外，值得注意的是，当要验证的 CTL 公式以“$E*$”形式模态词开头时，一个更自然的选择是：当该公式在给定的 Kripke 模型上不满足时不给出反例，而当该公式在给定的 Kripke 模型上满足时，给出一个证据（witness）来说明该公式可满足的原因[3]。比如，如果 $M, s \models E[\phi_1 U \phi_2]$，那么模型检测工具通常给出一个以 s 为起始状态的有穷路径，使得对于该路径上的最后一个状态 s' 有 $M, s' \models \phi_2$，而且对于该路径上的任意其他状态（如果存在）s'' 有 $M, s'' \models \phi_1$。满足这个条件的路径即为 $M, s \models E[\phi_1 U \phi_2]$ 的证据。

2.2 相继式演算（Sequent Calculus）

在本文中，我们需要用到相继式演算的概念。在数理逻辑与证明论中，相继式演算指的是一系列具有特定形式的证明系统：每个证明判断（proof judgement）包含两部分：上下文（context）和命题（proposition）。在相继式演算中，证明判断被称为相继式（sequent）。例如，在 $\Gamma \vdash A$ 形式的相继式中，命题集合 Γ 是上下文，A 是命题。$\Gamma \vdash A$ 可非形式化地解释为：若 Γ 中所有的命题均为真，那么命题 A 为真。在某些相继式演算中，$\vdash$ 后也可有多个命题，被称为多结论的相继式演算，比如 $\Gamma \vdash A_1, ..., A_n$，这种形式的相继式可非形式化地解释为：若 Γ 中所有命题均为真，那么 $A_1, ..., A_n$ 中存在某个为真的命题。在本文中，我们用到的是单结论的相继式演算，即每个相继式中 $\vdash$ 之后只有一个命题。

下面，我们在谓词逻辑中形式化地解释相继式演算的概念。在这之前，我们先介绍谓词逻辑语言、项、命题以及相继式的概念。

定义 2.4 (谓词逻辑语言)**.** 一个谓词逻辑语言 $\mathcal{L}$ 可用一个三元组 $(\mathcal{S}, \mathcal{F}, \mathcal{P})$ 来表示，其中

- $\mathcal{S}$ 是一个非空的有穷集合，集合的元素被称为项的类别（*term sort*）；
- $\mathcal{F}$ 是函数符号的集合；
- $\mathcal{P}$ 是谓词符号的集合。

每个函数符号都有一个元数（*arity*），用 S 上的一个 $(n+1)$-元组来表示；每个谓词符号也具有一个元数，用 S 上的一个 n-元组来表示。

定义 2.5 (项)**.** 假设一个谓词逻辑语言 $\mathcal{L} = (\mathcal{S}, \mathcal{F}, \mathcal{P})$，并且令 $(\mathcal{V}_s)_{s \in S}$ 为一系列无穷的变量集合，并由项的类别 s 区分开来。$\mathcal{L}$ 中项的集合由如下的方式归纳定义：

- 对于每个类别 $s \in S$，$\mathcal{V}_s$ 中的每个变量都是具有类别 s 的项；
- 对于每个函数符号 $f \in \mathcal{F}$，假设 f 的元数为 $(s_1, ..., s_n, s')$，那么对于任意 n 个项 $t_1, ..., t_n$，其中 t_i 的类别为 s_i，$f(t_1, ..., t_n)$ 是具有类别 s' 的项。

定义 2.6 (命题)**.** 假设一个谓词逻辑语言 $\mathcal{L} = (\mathcal{S}, \mathcal{F}, \mathcal{P})$，并且令 $(\mathcal{V}_s)_{s \in S}$ 为一系列无穷的变量集合，并由项的类别 s 区分开来。那么 $\mathcal{L}$ 中基于 $(\mathcal{V}_s)_{s \in S}$ 的命题的集合由如下的方式归纳定义：

- 对于每个谓词符号 $P \in \mathcal{P}$，假设 P 的元数为 $(s_1, ..., s_n)$，那么对于任意 n 个项 $t_1, ..., t_n$，其中 t_i 的类别为 s_i，$P(t_1, ..., t_n)$ 是一个命题；
- $\top$ 和 $\bot$ 均为命题；
- 如果 A 是一个命题，那么 $\neg A$ 也是一个命题；
- 如果 A 和 B 均为命题，那么 $A \wedge B$、$A \vee B$、$A \Rightarrow B$ 都是命题；
- 如果 A 是一个命题，且 x 是一个变量，那么 $\forall xA$ 和 $\exists xA$ 均为命题。

对于一个谓词语言 $\mathcal{L} = (\mathcal{S}, \mathcal{F}, \mathcal{P})$，如果 $\mathcal{S}$ 只包含一个元素，那么 $\mathcal{L}$ 中函数符号和谓词符号的元数可简单地表示为自然数，即参数的个数。

定义 2.7 (相继式)**.** 一个相继式被表示为一个具有 $\Gamma \vdash A$ 形式的二元组，其中 Γ 是一个命题的有穷集合，A 是一个命题。

相继式的证明是基于规则的，相继式演算的证明规则如下。

定义 2.8 (相继式演算的规则)**.** 证明规则如下方框中所示：

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

2.3 信息可视化

第 3 章　模型检测与定理证明的结合

在本章中，我们给出一种将模型检测与定理证明两种形式化验证方法的结合：首先，将模型检测中常用的时序逻辑—计算树逻辑 CTL 进行扩展，我们将扩展后的逻辑称为 CTL_P；然后，针对 CTL_P 逻辑建立一个带参的证明系统 SCTL($\mathcal{M}$)，其中参数 $\mathcal{M}$ 是任意的 Kripke 模型，并使得一个 CTL_P 公式在 SCTL($\mathcal{M}$) 中是可证的当且仅当该公式在 $\mathcal{M}$ 中成立。本章内容安排如下：首先介绍介绍 CTL_P 逻辑；然后介绍证明系统 SCTL; 最后介绍针对证明系统 SCTL 的一个证明搜索策略及其伪代码。

在形式化方法中，一个 CTL 模型检测问题则可表示为对于给定的 Kripke 模型 M 以及初始状态 s_0，判定 $M, s_0 \models \phi$ 是否成立，其中 ϕ 是一个 CTL 公式。当 ϕ 为原子命题 P 时，上述 CTL 模型检测问题则为 $M, s \models P$。这时，我们不仅仅将 P 看作一个原子命题，而且可以将 P 看作为一个一元谓词，因而上述模型检测问题可表示为：$M \models P(s)$。不仅如此，当谓词的元数大于 1 时，我们甚至可以表示多个状态之间的关系，比如 $M \models Q(s_1, ..., s_n)$。其他的 CTL 公式的可满足性同样可用这种方式表示，比如 $M, s \models EXP$ 可表示为 $M \models EX_x(P(x))(s)$，其中 P 是一个一元谓词，而且变量 x 由模态词 EX 绑定。这种表示 CTL 公式的方式可以用来定义一个对 CTL 的扩展，即不仅能表示状态的时序性质，而且能表达状态之间的“空间”性质。比如，对于一个 Kripke 模型上的路径 $s_0, s_1, ...$，在状态 s_m 上，某个人捡到一只鞋，而且在 s_m 之后的某个状态 s_n 上，这个人又捡到跟之前的鞋成对的另一只，其中 $0 \leq m < n$。这个性质可表示为公式 $EF_x(EF_y(Pair(x, y))(x))(s_0)$。接下来，我们来详细定义这种针对 CTL 的扩展。

3.1　针对计算树逻辑 CTL 的扩展：CTL_P

为了能在公式中表达多元谓词，首先我们需要在经典的 Kripke 模型的定义的基础上添加多元谓词的定义，同时使新模型的定义包含经典的 Kripke 模型的定义。为了叙述方便起见，我们也称新模型为 Kripke 模型，新的 Kripke 模型定义如下：

定义 3.1 (Kripke 模型)**.** *一个 Kripke 结构 $\mathcal{M} = (S, S_0, \longrightarrow, \mathcal{P})$ 包含如下四个部分：*

1. S 是一个有穷的状态集合；

2. $S_0 \subseteq S$ 是初始状态集合；

3. $\longrightarrow \subseteq S \times S$ 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 $s \longrightarrow s'$;

4. $\mathcal{P}$ 是一个有穷的谓词符号的集合；对于每个谓词符号 $P \in \mathcal{P}$，都存在自然数 n 使得 $P \in S^n$ 。

由以上定义可知，当 $\mathcal{P}$ 中的所有谓词符号均为一元时，以上定义与经典的 Kripke 模型的定义等价。除此之外，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

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

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 $s_{i+1} \in \text{Next}(s_i)$ 。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 $\text{Next}(s)$ 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。

我们用逻辑 $\text{CTL}_P(\mathcal{M})$ 来刻画要验证的 Kripke 模型 $\mathcal{M}$ 的性质，其定义如下。

定义 3.2 (CTL_P). 对于一个给定的 *Kripke* 模型 $\mathcal{M} = (\mathcal{S}, \mathcal{S}_{\prime}, \longrightarrow, \mathcal{P})$，$CTL_P(\mathcal{M})$ 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 3.2 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：

- $\phi_1 \Longrightarrow \phi_2 \equiv \neg\phi_1 \vee \phi_2$,
- $EF_x(\phi)(t) \equiv EU_{z,x}(\top, \phi)(t)$,
- $ER_{x,y}(\phi_1, \phi_2)(t) \equiv EU_{y,z}(\phi_2, ((z/x)\phi_1 \wedge (z/y)\phi_2))(t) \vee EG_y(\phi_2)(t)$, 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*。最外层模态词为归纳模态词的公式被称为*归纳公式*；最外层模态词为余归纳模态词的公式被称为*余归纳公式*。

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

- $\mathcal{M} \models \top$ 永远成立；
- $\mathcal{M} \models \bot$ 永远不成立；
- $\mathcal{M} \models P(s_1, ..., s_n)$：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；
- $\mathcal{M} \models \phi_1 \wedge \phi_2$：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 $s' \in \mathrm{Next}(s)$，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 $s' \in \mathrm{Next}(s)$，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 $\mathrm{Next}(s')$，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 $s_0, s_1, \ldots$（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）[34,35] 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 3.1. 本例子受多机器人路径规划系统[36,37] 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3.1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。

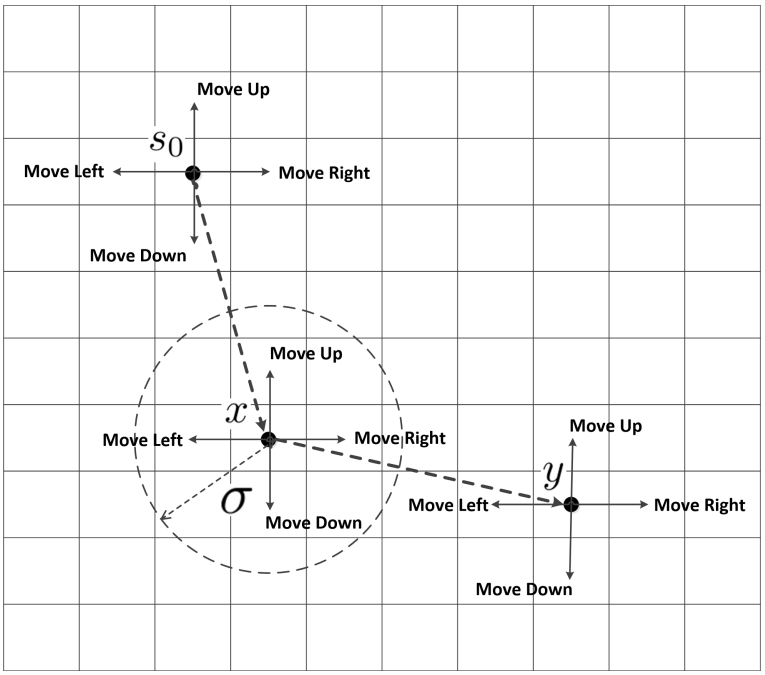


图 3.1 无人车可能的所在位置

3.2 CTL_P 的证明系统：SCTL

在本节，我们针对逻辑 $\mathrm{CTL}_P(\mathcal{M})$ 给出一个证明系统 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 SCTL($\mathcal{M}$) 中，一个公式是可证的当且仅当该公式在模型 $\mathcal{M}$ 中成立。

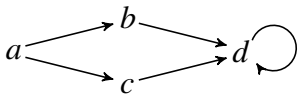
首先，让我们考虑一个 $\mathrm{CTL}_P(\mathcal{M})$ 公式 $AF_x(P(x))(s)$。该公式在模型 $\mathcal{M}$ 中成立当且仅当存在一个路径树 T，T 的根节点由 s 标记，而且 T 上的每个叶子节点都满足 P。

然后，我们考虑一个具有嵌套模态词的 $\mathrm{CTL}_P(\mathcal{M})$ 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。

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

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

例子 3.2. 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。

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

在此证明树中，除了 **AF**-R$_1$ 和 **AF**-R$_2$，我们还应用了如下规则。

$$\dfrac{}{\vdash P(s_1, \ldots, s_n)}\ \underset{\langle s_1,\ldots,s_n\rangle \in P}{\textit{atom-R}}$$

例子 3.3. 假设另一个模型，该模型的迁移规则与例子*3.2*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。

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

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 CTL$_P$ 公式的集合，ϕ 是一个 CTL$_P$ 公式。当 Γ 为空时，$\Gamma \vdash \phi$ 也可记作 $\vdash \phi$。在不包含余归纳子公式的相继式的证明中，每个相继式中 $\vdash$ 左侧均为空。在证明以余归纳公式开头的相继式时，往往需要访问无穷路径，比如若要证明 $\vdash EG_x(P(x))(s)$，则必须找到一个以 s 为起始的无穷路径使得 P 在这条无穷路径上的所有状态上都成立。为了可以在有穷步内完成以余归纳公式开头的相继式的证明，我们引入 merge 规则，即利用 $\vdash$ 左侧的 Γ 来记录已访问过的状态，进一步，为了区分相对于不同余归纳公式的 merge 规则，我们用 Γ 来记录已在证明中出现过的余归纳公式。由于 Kripke 模型中的状态是有穷的，因此每个 Γ 只包含有穷个公式。

SCTL($\mathcal{M}$) 的证明规则如图3.2所示。

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

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

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

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

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

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

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

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

图 3.2 SCTL($\mathcal{M}$)

SCTL 的可靠性与完备性 命题3.1和命题3.2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 的可靠性；命题3.3和命题3.4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 3.1 (有穷状态序列到无穷状态序列)**.** 给定一个有穷的状态序列 $s_0, \ldots, s_n$，其中对于任意 $0 \le i \le n-1$ 都有 $s_i \longrightarrow s_{i+1}$，而且存在 $0 \le p \le n-1$ 使得 $s_n = s_p$。那么，一定存在一个无穷的状态序列 $s'_0, s'_1, \ldots$ 使得 $s_0 = s'_0$，而且对于任意 $i \ge 0$ 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 $s_0, \ldots, s_n$ 中。

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

命题 3.2 (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 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 3.3 (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 3.4 (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 3.1 (可靠性). 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 ¬-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 ⊤-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 $s' \in \text{Next}(s)$，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 $s' \in \text{Next}(s)$，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 **AF**-R_1 或 **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 **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 **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 $\{s_1, ..., s_n\} = \text{Next}(s)$，则令 $|\pi|$ 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 **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 **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 **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 $s' \in \text{Next}(s)$，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3.1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 $\{s_1, ..., s_n\} = \text{Next}(s)$，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3.2，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 3.2 (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 $s' \in \text{Next}(s)$，都有 $\mathcal{M} \models (s'/x)\phi_1$ 成立。根据归纳假设，对于任意 $s' \in \text{Next}(s)$，$\vdash (s'/x)\phi_1$ 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 $s' \in \text{Next}(s)$ 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 $\text{Next}(s)$ 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 **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 $\{s_1, ..., s_n\} = \text{Next}(s)$。

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3.3，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3.4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 **AR**-R_2，其中 $s'_1, ..., s'_n = \text{Next}(s')$。

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 **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。

□

3.3 证明搜索策略

在本节，我们介绍一种在 SCTL 中进行证明搜索的策略，然后，我们证明该证明搜索策略的终止性和正确性，然后，我们提出两种针对该证明搜索策略的优

化方法，最后，我们将该证明搜索策略以伪代码的形式给出。

该证明搜素策略如下：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。我们将以上证明搜索方法定义成一系列对于连续传递树（定义3.3）的重写规则。下面我们介绍连续传递树的定义。

3.3.0.1 连续传递树

连续传递树的定义是基于连续的概念。在计算机程序设计语言理论[38,39] 中，连续指的是计算机程序将要执行的部分，一个函数被成为具有连续传递风格（Continuation-Passing Style）指的是将计算机程序将要执行的部分在函数的定义中显示表示出来。

定义 3.3 (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 t 或 f 标记，其中 t 和 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 t-连续；它的右子树称之为该节点的 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 t-连续和 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：

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

因此，SCTL 的该证明搜索策略可总结为：对于给定的 SCTL 相继式 $\vdash \phi$，我们构造一个连续传递树 $c = \mathrm{cpt}(\vdash \phi, \mathrm{t}, \mathrm{f})$，然后根据图3.4所示的重写规则将 c 重写到 t 或 f。如果 c 最终重写到 t，那么 $\vdash \phi$ 是可证的；如果 c 最终重写到 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，$\mathrm{CPTcpt}(\vdash \phi_1 \wedge \phi_2, \mathrm{t}, \mathrm{f})$ 重写到 $\mathrm{cpt}(\vdash \phi_1, \mathrm{cpt}(\vdash \phi_2, \mathrm{t}, \mathrm{f}), \mathrm{f})$，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 $\mathrm{cpt}(\vdash \phi_1, \mathrm{cpt}(\vdash \phi_2, \mathrm{t}, \mathrm{f}), \mathrm{f})$ 进行重写。下面，我们用一个完整的例子来说明该证明搜索策略。

例子 3.4. 根据重写规则，我们将例子*3.2*中公式的证明搜索过程表示为图*3.3*中的重写步骤。

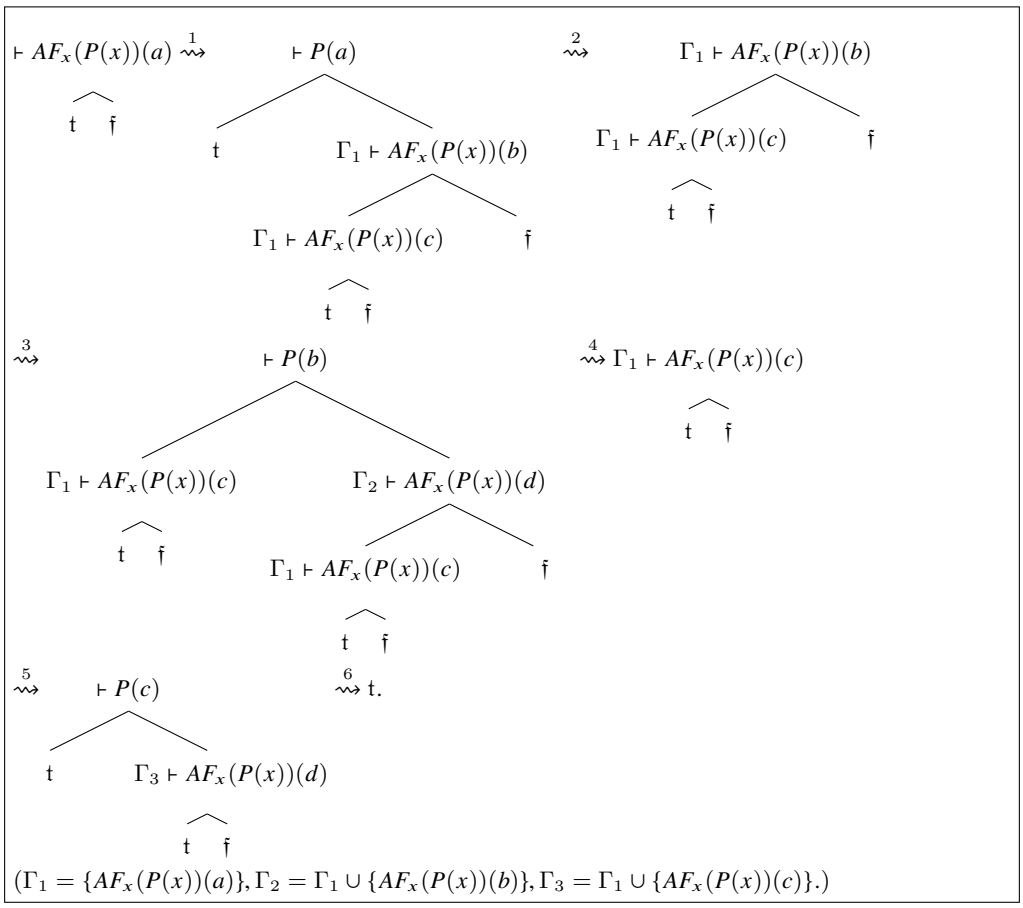


图 3.3 一个重写 CPT 的例子

下边我们分别解释图*3.3*中的每一步重写。

1. 在这一步中，$\overset{1}{\rightsquigarrow}$ 左侧的 *CPT* 中的根节点为 $\vdash AF_x(P(x))(a)$。我们暂时还无法判定 $\vdash AF_x(P(x))(a)$ 是否可证，因此根据证明规则，我们需要首先判定 $\vdash P(a)$ 是否可证，如果 $\vdash P(a)$ 可证，则 $\vdash AF_x(P(x))(a)$ 可证；否则，我们需要依次判定 $AF_x(P(x))(a) \vdash AF_x(P(x))(b)$ 和 $AF_x(P(x))(a) \vdash AF_x(P(x))(c)$ 是否可证，如果 $AF_x(P(x))(a) \vdash AF_x(P(x))(b)$ 和 $AF_x(P(x))(a) \vdash AF_x(P(x))(c)$ 均可证，则 $\vdash AF_x(P(x))(a)$ 可证，否则 $\vdash AF_x(P(x))(a)$ 可证不可证。我们将以上介绍的判定 $\vdash AF_x(P(x))(a)$ 是否可证过程编码到 $\overset{1}{\rightsquigarrow}$ 右侧的 *CPT* 中。
2. 在这一步中，由于 $\vdash P(a)$ 时不可证的，因此将 $\overset{2}{\rightsquigarrow}$ 左侧的 *CPT* 重写到它的右子树（f-连续），即 $\overset{2}{\rightsquigarrow}$ 右侧的 *CPT*。

3. 这一步与第 *1* 步类似，我们暂时无法判定 $AF_x(P(x))(a) \vdash AF_x(P(x))(b)$ 是否可证，因此根据证明规则，我们需要首先判定 $\vdash P(b)$ 是否可证，如果 $\vdash P(b)$ 可证，那么 $AF_x(P(x))(a) \vdash AF_x(P(x))(b)$ 可证，然后接着第 *2* 步证明 $\overset{3}{\rightsquigarrow}$ 左侧的 *CPT* 的左子树；否则，我们需要判定 $AF_x(P(x))(a), AF_x(P(x))(b) \vdash AF_x(P(x))(d)$ 是否可证，若 $AF_x(P(x))(a), AF_x(P(x))(b) \vdash AF_x(P(x))(d)$ 可证，则接着第 *2* 步证明 $\overset{3}{\rightsquigarrow}$ 左侧的 *CPT* 的左子树，若 $AF_x(P(x))(a), AF_x(P(x))(b) \vdash AF_x(P(x))(d)$ 不可证，则 $AF_x(P(x))(a) \vdash AF_x(P(x))(b)$ 不可证。我们将以上介绍的判定 $AF_x(P(x))(a) \vdash AF_x(P(x))(b)$ 是否可证过程编码到 $\overset{3}{\rightsquigarrow}$ 右侧的 *CPT* 中。

4. 这一步与第 *2* 步类似，我们可以判定 $\vdash P(b)$ 可证，因此 $\overset{4}{\rightsquigarrow}$ 左侧的 *CPT* 重写到它的左子树，即 $\overset{4}{\rightsquigarrow}$ 右侧的 *CPT*。

5. 这一步与第 *1*、*3* 步类似，我们暂时无法判定 $AF_x(P(x))(a) \vdash AF_x(P(x))(c)$ 是否可证，因此我们需要首先判定 $\vdash P(c)$ 是否可证，若 $\vdash P(c)$ 可证，则 $AF_x(P(x))(a) \vdash AF_x(P(x))(c)$ 可证；否则，判定 $AF_x(P(x))(a), AF_x(P(x))(c) \vdash AF_x(P(x))(d)$ 是否可证，若 $AF_x(P(x))(a), AF_x(P(x))(c) \vdash AF_x(P(x))(d)$ 可证，则 $AF_x(P(x))(a) \vdash AF_x(P(x))(c)$ 可证。我们将以上介绍的判定 $AF_x(P(x))(a) \vdash AF_x(P(x))(c)$ 是否可证过程编码到 $\overset{5}{\rightsquigarrow}$ 右侧的 *CPT* 中。

6. 在这一步中，由于 $\vdash P(c)$ 可证，因此 $\overset{6}{\rightsquigarrow}$ 左侧的 *CPT* 重写到它的左子树，即 t。至此，判定 $\vdash AF_x(P(x))(a)$ 是否可证的过程结束，$\vdash AF_x(P(x))(a)$ 可证。

3.3.1 证明搜索策略的可终止性

在证明利用图3.3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 3.4 (字典路径序（lexicographic path ordering）[40,41]). 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 $\succeq_{\text{lpo}}$ 的归纳定义如下：

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

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

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

图 3.4 CPT 的重写规则

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

命题 3.5 (字典路径序的良基性)**.** 如果 $\geq$ 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 $\geq_{\mathrm{lpo}}$ 是良基的当且仅当 $\geq$ 是良基的。

证明. 证明由 Dershowitz 提出，参考[40]。 □

定义 3.5 (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 3.6 (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 $cpt(\vdash \phi, \mathrm{t}, \mathrm{f})$ 能在有限步之内重写到 t 或 f。

证明. 令 $F = \{\mathrm{t}, \mathrm{f}, \mathrm{cpt}\} \cup \mathrm{Seq}$，其中 Seq 是在 $\mathrm{cpt}(\vdash \phi, \mathrm{t}, \mathrm{f})$ 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：

- $\mathrm{cpt} \succ \mathrm{t}$；
- $\mathrm{cpt} \succ \mathrm{f}$；
- 对于每个相继式 $\Gamma \vdash \phi$ 都有 $\Gamma \vdash \phi \succ \mathrm{cpt}$；
- $\Gamma \vdash \phi \succ \Gamma' \vdash \phi'$ 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 $\succeq_{\mathrm{lpo}}$ 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3.5可知，$\succeq_{\mathrm{lpo}}$ 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 $c \succ_{\mathrm{lpo}} c'$。下面我们针对重写规则逐条进行分析：

假设 c 是 $\mathrm{cpt}(\Gamma \vdash \phi, c_1, c_2)$ 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 $\mathrm{cpt}(\Gamma \vdash \phi, c_1, c_2)$ 的子项，因此，$\mathrm{cpt}(\Gamma \vdash \phi, c_1, c_2) \succ_{\mathrm{lpo}} c_1$ 以及 $\mathrm{cpt}(\Gamma \vdash \phi, c_1, c_2) \succ_{\mathrm{lpo}} c_2$。
- 如果 $\phi = \phi_1 \wedge \phi_2$, 那么由 $\succ_{\mathrm{lpo}}$ 的定义可知，由于 $\vdash \phi_1 \wedge \phi_2 \succ \vdash \phi_1$，$\mathrm{cpt}(\vdash \phi_1 \wedge \phi_2, c_1, c_2) \succ_{\mathrm{lpo}} \mathrm{cpt}(\vdash \phi_2, c_1, c_2)$ 以及 $\mathrm{cpt}(\vdash \phi_1 \wedge \phi_2, c_1, c_2) \succ_{\mathrm{lpo}} c_2$，因此 $\mathrm{cpt}(\vdash \phi_1 \wedge \phi_2, c_1, c_2) \succ_{\mathrm{lpo}} \mathrm{cpt}(\vdash \phi_1, \mathrm{cpt}(\vdash \phi_2, c_1, c_2), c_2)$；
- 如果 $\phi = \phi_1 \vee \phi_2$，那么由 $\succ_{\mathrm{lpo}}$ 的定义可知，由于 $\vdash \phi_1 \vee \phi_2 \succ \vdash \phi_1$，$\mathrm{cpt}(\vdash \phi_1 \wedge \phi_2, c_1, c_2) \succ_{\mathrm{lpo}} c_1$ 以及 $\mathrm{cpt}(\vdash \phi_1 \vee \phi_2, c_1, c_2) \succ_{\mathrm{lpo}} \mathrm{cpt}(\vdash \phi_2, c_1, c_2)$，因此 $\mathrm{cpt}(\vdash \phi_1 \vee \phi_2, c_1, c_2) \succ_{\mathrm{lpo}} \mathrm{cpt}(\vdash \phi_1, c_1, \mathrm{cpt}(\vdash \phi_2, c_1, c_2))$；

- 如果 $\phi = AX_x(\phi_1)(s)$，那么根据 $\succ_{\text{lpo}}$ 的定义可知，由于 $\Gamma \vdash AX_x(\phi_1)(s) \succ \vdash (s_i/x)\phi_1$ 以及 $\text{cpt}(\Gamma \vdash AX_x(\phi_1)(s), c_1, c_2) \succ_{\text{lpo}} \text{cpt}(\vdash (s_i/x)\phi_1, \text{cpt}(...\text{cpt}(\vdash (s_n/x)\phi_1, c_1, c_2)..., c_2), c_2)$，因此 $\text{cpt}(\Gamma \vdash AX_x(\phi_1)(s), c_1, c_2) \succ_{\text{lpo}} \text{cpt}(\vdash (s_1/x)\phi_1, \text{cpt}(...\text{cpt}(\vdash (s_n/x)\phi_1, c_1, c_2), ..., c_2), c_2)$，其中 $\text{Next}(s) = \{s_1, ..., s_n\}$，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：$c \succ_{\text{lpo}} c'$；
  - 当 $EG_x(\phi_1)(s) \notin \Gamma$ 时，根据 $\succ_{\text{lpo}}$ 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 $\text{Next}(s) = \{s_1, ..., s_n\}$ 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：$c \succ_{\text{lpo}} c'$；
  - 当 $AR_{x,y}(\phi_1, \phi_2)(s) \notin \Gamma$ 时，由 $\succ_{\text{lpo}}$ 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 $c \succ_{\text{lpo}} c'$，其中 $\text{Next}(s) = \{s_1, ..., s_n\}$ 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。

□

3.3.2 证明搜索策略的正确性

证明搜索策略的正确性可由以下命题来表示。

命题 3.7 (证明搜索策略的正确性)**.** *对于给定闭公式 ϕ，$cpt(\vdash \phi, \mathrm{t}, \mathrm{f}) \rightsquigarrow^* \mathrm{t}$ 当且仅当 $\vdash \phi$ 是可证的。*

证明. 在证明该命题之前，需要注意的是，对任意不同的两个 CPT c_1 和 c_2，$\text{cpt}(\Gamma \vdash \phi, c_1, c_2)$ 总会在有限步之内重写到 c_1 或 c_2。这是由于根据 CPT 重写系统的终止性，$\text{cpt}(\vdash \phi, \mathrm{t}, \mathrm{f})$ 总会在有限步之内重写到 t 或 f，而且在重写 $\text{cpt}(\vdash \phi, c_1, c_2)$ 的过

程中;c_1 和 c_2 的结构都不会影响重写的步骤，直到需要重写 c_1 或者 c_2 本身。因此，我们可以在 $\mathrm{cpt}(\Gamma \vdash \phi, \mathsf{t}, \mathsf{f})$ 的重写步骤中将 t 替换成 c_1，将 f 替换成 c_2，并由此得到一个由 $\mathrm{cpt}(\Gamma \vdash \phi, c_1, c_2)$ 重写到 c_1 或 c_2 的步骤。

为了保持证明的简洁性，我们证明一个更一般化的命题，即：对于一个给定的公式 ϕ, 以及任意不同的两个 CPT c_1 和 c_2，$\mathrm{cpt}(\vdash \phi, c_1, c_2)$ 在有限步内首先重写到 c_1 而不是 c_2 当且仅当 $\Gamma \vdash \phi$ 是可证的。

本证明由两部分组成：

1. $\Rightarrow$：对于一个给定的公式 ϕ, 以及任意不同的两个 CPT c_1 和 c_2，如果 $\mathrm{cpt}(\vdash \phi, c_1, c_2)$ 在有限步内首先重写到 c_1 而不是 c_2，那么 $\Gamma \vdash \phi$ 是可证的。

2. $\Leftarrow$：对于一个给定的公式 ϕ, 以及任意不同的两个 CPT c_1 和 c_2，如果 $\Gamma \vdash \phi$ 是可证的，那么 $\mathrm{cpt}(\vdash \phi, c_1, c_2)$ 在有限步内首先重写到 c_1 而不是 c_2。

两个部分的证明都是在公式 ϕ 的结构上作归纳。

在第一部分中，我们证明 $\Rightarrow$，其中假设 c_1 和 c_2 是两个不同的 CPT：

- 如果 $\phi = \top$ 或 $\bot$，显然得证。

- 如果 $\phi = P(s_1, ..., s_n)$，其中 $P(s_1, ..., s_n)$ 是一个原子公式，而且 $\mathrm{cpt}(\vdash P(s_1, ..., s_n), c_1, c_2)$ 在有限步内首先重写到 c_1 而不是 c_2，那么 $\mathrm{cpt}(\vdash P(s_1, ..., s_n), c_1, c_2) \rightsquigarrow c_1$，其中 $\langle s_1, ..., s_n \rangle \in P$，因而 $\vdash P(s_1, ..., s_n)$ 是可证的。

- 如果 $\phi = \neg P(s_1, ..., s_n)$，其中 $P(s_1, ..., s_n)$ 是一个原子公式，而且 $\mathrm{cpt}(\vdash \neg P(s_1, ..., s_n), c_1, c_2)$ 在有限步内首先重写到 c_1 而不是 c_2，那么 $\mathrm{cpt}(\vdash \neg P(s_1, ..., s_n), c_1, c_2) \rightsquigarrow c_1$，其中 $\langle s_1, ..., s_n \rangle \notin P$，因而 $\vdash \neg P(s_1, ..., s_n)$ 是可证的。

- 如果 $\phi = \phi_1 \wedge \phi_2$，而且 $\mathrm{cpt}(\vdash \phi_1 \wedge \phi_2, c_1, c_2)$ 在有限步内首先重写到 c_1 而不是 c_2，那么 $\mathrm{cpt}(\vdash \phi_1 \wedge \phi_2, c_1, c_2) \rightsquigarrow \mathrm{cpt}(\vdash \phi_1, \mathrm{cpt}(\vdash \phi_2, c_1, c_2), c_2) \rightsquigarrow^* \mathrm{cpt}(\vdash \phi_2, c_1, c_2) \rightsquigarrow^* c_1$，然后由归纳假设可知，$\vdash \phi_1$ 和 $\vdash \phi_2$ 均可证，否则, 如果 $\vdash \phi_1$ 或 $\vdash \phi_2$ 不可证，那么 $\mathrm{cpt}(\vdash \phi_1, \mathrm{cpt}(\vdash \phi_2, c_1, c_2), c_2)$ 会在有限步内首先重写到 c_2 而不是 c_1。由证明规则可知，$\vdash \phi_1 \wedge \phi_2$ 是可证的。

- 如果 $\phi = \phi_1 \vee \phi_2$，那么有两种方法使得 $\mathrm{cpt}(\vdash \phi_1 \vee \phi_2, c_1, c_2)$ 在有限步内首先重写到 c_1 而不是 c_2：

1. $\mathrm{cpt}(\vdash \phi_1 \vee \phi_2, c_1, c_2) \rightsquigarrow \mathrm{cpt}(\vdash \phi_1, c_1, \mathrm{cpt}(\vdash \phi_2, c_1, c_2))$，而且 $\mathrm{cpt}(\vdash \phi_1, c_1, \mathrm{cpt}(\vdash \phi_2, c_1, c_2))$ 在有限步内首先重写到 c_1 而不是 $\mathrm{cpt}(\vdash \phi_2, c_1, c_2)$。由归纳假设可知，$\vdash \phi_1$ 是可证的，然后由证明规则可知，$\vdash \phi_1 \vee \phi_2$ 是可证的。

2. $\mathrm{cpt}(\vdash \phi_1 \vee \phi_2, c_1, c_2) \rightsquigarrow \mathrm{cpt}(\vdash \phi_1, c_1, \mathrm{cpt}(\vdash \phi_2, c_1, c_2))$，$\mathrm{cpt}(\vdash \phi_1, c_1, \mathrm{cpt}(\vdash \phi_2, c_1, c_2))$ 在有限步内首先重写到 $\mathrm{cpt}(\vdash \phi_2, c_1, c_2)$ 而不是 c_1，而且 $\mathrm{cpt}(\vdash \phi_2, c_1, c_2)$ 在有限步内首先重写到 c_1 而不是 c_2。由归纳假设可知，$\vdash \phi_1$ 是不可证的，而 $\vdash \phi_2$ 是可证的，然后由证明系统的规则可知，$\vdash \phi_1 \vee \phi_2$ 是可证的。

- 如果 $\phi = EX_x(\phi_1)(s)$，而且 $\{s_1, ..., s_n\} = \mathrm{Next}(s)$，那么 $\mathrm{cpt}(\vdash EX_x(\phi_1)(s), c_1, c_2)$ 在有限步内首先重写到 c_1 而不是 c_2 可以推出：存在 $s_i \in \mathrm{Next}(s)$ 使得 $\vdash (s_i/x)\phi_1$ 是可证的。否则，如果对所有 $s_i \in \mathrm{Next}(s)$，$\vdash (s_i/x)\phi_1$ 都是不可证的，那么由归纳假设可知，

  $\mathrm{cpt}(\vdash EX_x(\phi_1)(s), c_1, c_2) \rightsquigarrow$

  $\mathrm{cpt}(\vdash (s_1/x)\phi_1, c_1, \mathrm{cpt}(...\mathrm{cpt}(\vdash (s_n/x)\phi_1, c_1, c_2)...))$

  $\rightsquigarrow^* \ldots \rightsquigarrow^*$

  $\mathrm{cpt}(\vdash (s_n/x)\phi_1, c_1, c_2) \rightsquigarrow^* c_2$，而且在上述步骤中没有对 c_1 的重写。因此，由证明系统的规则可知，$\vdash EX_x(\phi_1)(s)$ 是可证的。

- 如果 $\phi = AX_x(\phi_1)(s)$，那么由于 EX 是 AX 的对偶，因此对这种情况的分析与 EX 类似。

- 如果 $\phi = EG_x(\phi_1)(s)$，而且 $\{s_1, ..., s_n\} = \mathrm{Next}(s)$，那么 $\mathrm{cpt}(\vdash EG_x(\phi_1)(s), c_1, c_2)$ 在有限步内首先重写到 c_1 而不是 c_2 可推出：存在一条起始于 s 的无穷路径，使得对于这条路径上的任意状态 s'，$\vdash (s'/x)\phi_1$ 都是可证的。这是由于，根据重写系统的规则可知，如果 $\mathrm{cpt}(\vdash EG_x(\phi_1)(s), c_1, c_2) \rightsquigarrow^* c'$，其中 c' 为 c_1 或 c_2 而且还未被重写，那么在以上的重写步骤中，c_1 不是以上出现过的任何 CPT 的 f-连续，而只可能作为具有 $\mathrm{cpt}(\Gamma' \vdash EG_x(\phi_1)(s'), c_1, c_2')$ 形式的 CPT 的 t-连续。因此，只有一种方式使得 $\mathrm{cpt}(\vdash EG_x(\phi_1)(s), c_1, c_2)$ 在有限步内首先重写到 c_1 而不是 c_2，即 $\mathrm{cpt}(\Gamma' \vdash EG_x(\phi_1)(s'), c_1, c_2') \rightsquigarrow c_1$ 形式的重写是上述重写步骤的一部分，其中 $EG_x(\phi_1)(s') \in \Gamma'$。在这种情况下，根据证明系统的规则可知，所有的公式 $EG_x(\phi_1)(s') \in \Gamma', \vdash (s'/x)\phi_1$ 都是可证的。因此，存在一条无穷路径使得对于该路径上的任意节点 s'，$\vdash (s'/x)\phi_1$ 都是可证的，由于证明系统是可靠和完备的，因此 $\vdash EG_x(\phi_1)(s)$ 是可证的。

- 如果 $\phi = AF_x(\phi_1)(s)$，那么由于 EG 是 AF 的对偶，因此对这种情况的分析与 EG 类似。

- 如果 $\phi = EU_{x,y}(\phi_1, \phi_2)(s)$，而且 $\{s_1, ..., s_n\} = \text{Next}(s)$，那么 $\text{cpt}(\vdash EU_{x,y}(\phi_1, \phi_2)(s), c_1, c_2)$ 在有限步内首先重写到 c_1 而不是 c_2 可推出：存在一条起始于 s 的有穷路径使得对于该路径上的最后一个状态 s'，$\vdash (s'/y)\phi_2$ 是可证的，且对于该路径上的所有其他（如果存在）状态 s''，$\vdash (s''/x)\phi_1$ 是可证的。这是由于，对于重写步骤 $\text{cpt}(\vdash EU_{x,y}(\phi_1, \phi_2)(s), c_1, c_2) \rightsquigarrow^* c'$，其中 c' 为 c_1 或 c_2 且还未被重写，由重写系统的规则可知，c_1 不是以上步骤中出现过的任何 CPT 的 f-连续，而只可能作为具有 $\text{cpt}(\Gamma' \vdash EU_{x,y}(\phi_1, \phi_2)(s_1), c_1, c_2')$ 或 $\text{cpt}(\vdash (s_2/x)\phi_2, c_1, c_2'')$ 形式的 CPT 的 t-连续，且具有前者形式的 CPT 总会重写到具有后者形式的 CPT。因此，只有一种方式使得 $\text{cpt}(\vdash EU_{x,y}(\phi_1, \phi_2)(s), c_1, c_2)$ 在有限步内首先重写到 c_1 而不是 c_2，即具有 $\text{cpt}(\Gamma' \vdash EU_{x,y}(\phi_1, \phi_2)(s'), c_1, c_2') \rightsquigarrow \text{cpt}(\vdash (s'/y)\phi_2, c_1, c_2'') \rightsquigarrow^* c_1$ 形式的重写是以上重写步骤的不一份。在这种情况下，由归纳假设可知，$\vdash (s'/y)\phi_2$ 是可证的，而且由重写规则可知，对于所有的公式 $EU_{x,y}(\phi_1, \phi_2)(s_1) \in \Gamma'$，$\vdash (s_1/x)\phi_1$ 是可证的。因此，存在一条起始于 s 的有穷路径，使得对于该路径上的最后一个状态 s'，$\vdash (s'/y)\phi_2$ 是可证的，而且对于该路径上的任意其他（如果存在）状态 s''，$\vdash (s''/x)\phi_1$ 是可证的。然后，由证明系统的可靠性和完备性可知，$\vdash EU_{x,y}(\phi_1, \phi_2)(s)$ 是可证的。

- 如果 $\phi = AR_{x,y}(\phi_1, \phi_2)(s)$，那么由于 EU 是 AR 的对偶，因此对这种情况的分析与 EU 类似。

在第二部分中，我们证明 $\Leftarrow$，其中假设 c_1 和 c_2 是两个不同的 CPT：

- 如果 $\phi = \top$ 或 $\bot$，显然得证。

- 如果 $\phi = P(s_1, ..., s_n)$，其中 $P(s_1, ..., s_n)$ 是一个原子公式，且 $\vdash P(s_1, ..., s_n)$ 是可证的，那么 $\langle s_1, ..., s_n \rangle \in P$，因此 $\text{cpt}(\vdash P(s_1, ..., s_n), c_1, c_2) \rightsquigarrow c_1$。

- 如果 $\phi = \neg P(s_1, ..., s_n)$，其中 $P(s_1, ..., s_n)$ 是一个原子公式，且 $\vdash \neg P(s_1, ..., s_n)$ 是可证的，那么 $\langle s_1, ..., s_n \rangle \notin P$，因此 $\text{cpt}(\vdash \neg P(s_1, ..., s_n), c_1, c_2) \rightsquigarrow c_1$。

- 如果 $\phi = \phi_1 \wedge \phi_2$ 且 $\vdash \phi$ 是可证的，那么由证明系统的规则可知，$\vdash \phi_1$ 和 $\vdash \phi_2$ 均可证，那么由归纳假设可知，$\text{cpt}(\vdash \phi_1 \wedge \phi_2, c_1, c_2) \rightsquigarrow \text{cpt}(\vdash \phi_1, \text{cpt}(\vdash \phi_2, c_1, c_2), c_2) \rightsquigarrow^* \text{cpt}(\vdash \phi_2, c_1, c_2) \rightsquigarrow^* c_1$，而且 c_2 在上述步骤中未被重写。

- 如果 $\phi = \phi_1 \vee \phi_2$ 且 $\vdash \phi$ 是可证的，那么由证明系统的规则可知，$\vdash \phi_1$ 或 $\vdash \phi_2$ 是可证的：

  1. 如果 $\vdash \phi_1$ 是可证的，那么由归纳假设可知，$\text{cpt}(\vdash \phi_1 \vee \phi_2, c_1, c_2) \rightsquigarrow \text{cpt}(\vdash \phi_1, c_1, \text{cpt}(\vdash \phi_2, c_1, c_2)) \rightsquigarrow^* c_1$，而且 c_2 在上述步骤中未被重写。
  2. 如果 $\vdash \phi_1$ 是不可证的，而 $\vdash \phi_2$ 是可证的，那么由归纳假设可知，$\text{cpt}(\vdash \phi_1 \vee \phi_2, c_1, c_2) \rightsquigarrow \text{cpt}(\vdash \phi_1, c_1, \text{cpt}(\vdash \phi_2, c_1, c_2)) \rightsquigarrow^* \text{cpt}(\vdash \phi_2, c_1, c_2) \rightsquigarrow^* c_1$，而且 c_2 在上述步骤中未被重写。

- 如果 $\phi = EX_x(\phi_1)(s)$，$\{s_1, ..., s_n\} = \text{Next}(s)$，而且 $\vdash \phi$ 是可证的，那么由证明系统的规则可知，存在 $s_i \in \text{Next}(s)$ 使得 $\vdash (s_i/x)\phi_1$ 是可证的，而且对于任意 $1 \leq j < i$，$\vdash (s_j/x)\phi_1$ 是不可证的。那么，由归纳假设可知，

  $\text{cpt}(\vdash EX_x(\phi_1)(s), c_1, c_2) \rightsquigarrow$

  $\text{cpt}(\vdash (s_1/x)\phi_1, c_1, \text{cpt}(...\text{cpt}(\vdash (s_n/x)\phi_1, c_1, c_2)...))$

  $\rightsquigarrow^* \ldots \rightsquigarrow^*$

  $\text{cpt}(\vdash (s_i/x)\phi_1, c_1, \text{cpt}(...\text{cpt}(\vdash (s_n/x)\phi_1, c_1, c_2)...))$

  $\rightsquigarrow^* c_1$，而且 c_2 在上述步骤中未被重写。

- 如果 $\phi = AX_x(\phi_1)(s)$，那么由于 EX 是 AX 的对偶，因此对这种情况的分析与 EX 类似。

- 如果 $\phi = EG_x(\phi_1)(s)$，$\{s_1, ..., s_n\} = \text{Next}(s)$，而且 $\vdash EG_x(\phi_1)(s)$ 是可证的，那么由证明系统的可靠性和完备性可知，存在一条起始于 s 的无穷路径，使得对于这条路径上的任意状态 s'，$\vdash (s'/x)\phi_1$ 是可证的。由重写规则可知，在对 EG 公式的证明搜索过程中，对于状态的访问是按照深度优先的方式进行的，而且对于每个分支，有新的状态加入该分支只有当在该分支的最后一个状态上 ϕ_1 是满足的，而且该分支不包含重复的状态。由于搜索到的状态总数是有穷的，因此对于搜索到的每个分支，在该分支的最后一个状态上 ϕ_1 不满足，或者该分支包含重复的状态。那么，假设 $\text{cpt}(\vdash EG_x(\phi_1)(s), c_1, c_2)$ 在有限步内首先重写到 c_2 而不是 c_1，那么，所有搜索到的分支均不包含重复的状态，否则具有 $\text{cpt}(\Gamma' \vdash EG_x(\phi_1)(s'), c_1, c_2') \rightsquigarrow c_1$ 形式的重写是上述重写步骤的一部分，其中 $EG_x(\phi_1)(s') \in \Gamma'$，从而使得 $\text{cpt}(\vdash EG_x(\phi_1)(s), c_1, c_2)$ 在有限步内首先重写到 c_1 而不是 c_2。由此，我们可知，在每一个搜索到的分支的最后一个节点上，ϕ_1 均不满足，从而不存在一个无穷路径使得对于

该路径上的任意状态 s'，$\vdash (s'/x)\phi_1$ 是可证的。由此，我们可以得出结论，$\mathrm{cpt}(\vdash EG_x(\phi_1)(s), c_1, c_2)$ 总会在有限步内首先重写到 c_1 而不是 c_2。

- 如果 $\phi = AF_x(\phi_1)(s)$，那么由于 EG 是 AF 的对偶，因此对这种情况的分析与 EG 类似。

- 如果 $\phi = EU_{x,y}(\phi_1, \phi_2)(s)$，$\{s_1, ..., s_n\} = \mathrm{Next}(s)$ 而且 $\vdash \phi$ 是可证的，那么由证明系统的可靠性和完备性可知，存在一条起始于 s 的有穷路径使得对于该路径上的最后一个状态 s'，$\vdash (s'/y)\phi_2$ 是可证的，而且对于此路径上的任意其他（如果存在）状态 s''，$\vdash (s''/x)\phi_1$ 是可证的。由重写规则可知，在对 EU 公式的证明搜索中的每个状态分支，有新的状态加入该分支仅当在该分支的最后一个状态上 ϕ_1 满足且 ϕ_2 不满足，而且该分支不包含重复的状态。因此，对于在 EU 公式的证明搜索过程中所有搜索过的分支，在该分支的最后一个状态上，ϕ_2 满足，或者 ϕ_1 不满足，或者该分支包含重复的状态。假设 $\mathrm{cpt}(\vdash EU_{x,y}(\phi_1, \phi_2)(s), c_1, c_2)$ 在有限步内首先重写到 c_2 而不是 c_1，那么对于在该重写步骤中每个搜索到的状态分支，在该分支的最后一个状态上 ϕ_1 不满足，或者该分支包含重复的状态。否则，如果在该分支的最后一个状态上，ϕ_2 满足，那么由重写规则可知，具有
  $\mathrm{cpt}(\Gamma' \vdash EU_{x,y}(\phi_1, \phi_2)(s'), c_1, c_2') \rightsquigarrow^*$
  $\mathrm{cpt}(\vdash (s'/y)\phi_2, c_1, c_2'') \rightsquigarrow^* c_1$ 形式的重写出现在上述重写步骤中。因此，如果 $\mathrm{cpt}(\vdash EU_{x,y}(\phi_1, \phi_2)(s), c_1, c_2)$ 在有限步内首先重写到 c_2 而不是 c_1，那么不存在一条有穷的路径使得在该路径的最后一个状态上，ϕ_2 满足，而且在该路径上的所有其他（如果存在）状态上，ϕ_1 满足。因此，我们可以得出结论，$\mathrm{cpt}(\vdash EU_{x,y}(\phi_1, \phi_2)(s), c_1, c_2)$ 在有限步内首先重写到 c_1 而不是 c_2。

- 如果 $\phi = AR_{x,y}(\phi_1, \phi_2)(s)$，那么由于 EU 是 AR 的对偶，因此对这种情况的分析与 EU 类似。

□

3.3.3 证明搜索策略的优化

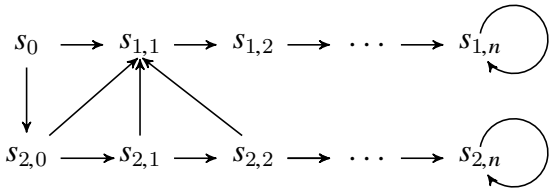
在余归纳公式（尤其是 EG 和 AR 公式）的证明搜索中，我们通常利用 merge 规则来证明某个性质在无穷路径上满足。对于每一个 merge 规则，上下文 Γ（我们称之为 merge）中的公式都是一种类型的，而且都与要证明的公式是相同类型的，唯一的不同是公式中相应的状态常量不同。本质上来说，每个 merge 对应的

是无穷路径上的状态，其中所有的状态都满足某个性质。因此，在 merge 规则的实现中，我们可以对于每一种公式类型，只在 Γ 中记录状态常量。

需要注意的是，对于归纳公式（尤其是 *AF* 和 *EU* 公式），虽然在证明规则中没有 merge 规则，但是在证明搜索中仍然需要 merge。对于这些公式，merge 的存在可避免无穷的证明搜索。这是由于当归纳公式是不可证的时候，即它们的非（余归纳公式）是可证的时候，我们需要 merge 来表示一个无穷路径，而且这个无穷路径上的状态都不满足某些性质。对于归纳公式，之所以在证明规则中没有 merge，而在证明搜索中需要 merge 是因为，在证明规则中我们只关心证明树的形状，而不是证明树的构造过程，只有在证明树的构造过程中才需要记录归纳公式的 merge。因此，作为对证明搜索策略的另一个优化，我们需要在归纳公式的证明搜索中同样记录 merge。

另外，在证明搜索过程中，我们对每个子公式都用用一个全局的数据结构来保存访问过的状态，以避免在证明子公式的时候同一个状态被重复访问。在证明搜索的实际实现中，用来记录状态集合的全局数据结构既可以是哈希表也可以是 BDD。两种数据结构各有优缺点：当模型中的状态变量绝大多数为布尔类型的时候，用 BDD 往往能减少空间的占用；而当模型中包含许多非布尔类型的状态变量时，如果用 BDD 记录的话则需将所有的非布尔变量转化成布尔变量，这个转化的过程会增多状态变量的个数，从而消耗掉更多的时间与空间，因此这时用哈希表来记录状态往往效率更高。接下来，我们利用一个例子来说明利用全局数据结构来记录状态之后，验证的效率会提升。

例子 3.5. *考虑一个具有如下状态迁移图的 Kripke 模型，其中 $n > 1$，而且在该模型中有集合 $P = \{s_0, s_{1,1}, s_{1,2}, \ldots, s_{1,n-1}, s_{2,0}, \ldots, s_{2,n}\}$。*



在该模型中，我们考虑公式 $EG_x(P(x))(s_0)$ 的验证，为了验证该公式，我们需要重写 CPT $cpt(\vdash EG_x(P(x))(s_0), \mathrm{t}, \mathrm{f})$。设在集合 $Next(s_0)$ 中，对状态 $s_{1,1}$ 的搜索在状态 $s_{2,0}$ 之前；对于任意的 $0 \le i \le n-1$，在集合 $Next(s_{2,i})$ 中，对状态 $s_{1,1}$ 的搜索在状态 $s_{2,i+1}$ 之前。重写步骤如图3.5所示，其中，令

$$\Gamma_0 = \{EG_x(P(x))(s_0)\},$$

$$\Gamma_{1,i} = \{EG_x(P(x))(s_0), EG_x(P(x))(s_{1,1}), \ldots, EG_x(P(x))(s_{1,i})\},$$

以及

$$\Gamma_{2,i} = \{EG_x(P(x))(s_0), EG_x(P(x))(s_{2,0}), \ldots, EG_x(P(x))(s_{2,i})\}$$

$\mathrm{cpt}(\vdash EG_x(P(x))(s_0), \mathrm{t}, \mathrm{f}) \rightsquigarrow^*$
$\mathrm{cpt}(\Gamma_0 \vdash EG_x(P(x))(s_{1,1}), \mathrm{t}, \mathrm{cpt}(\Gamma_0 \vdash EG_x(P(x))(s_{2,0}), \mathrm{t}, \mathrm{f})) \rightsquigarrow^*$
$\mathrm{cpt}(\Gamma_{1,1} \vdash EG_x(P(x))(s_{1,2}), \mathrm{t}, \mathrm{cpt}(\Gamma_0 \vdash EG_x(P(x))(s_{2,0}), \mathrm{t}, \mathrm{f})) \rightsquigarrow^*$
$\ldots \rightsquigarrow^*$
$\mathrm{cpt}(\Gamma_{1,n-1} \vdash EG_x(P(x))(s_{1,n}), \mathrm{t}, \mathrm{cpt}(\Gamma_0 \vdash EG_x(P(x))(s_{2,0}), \mathrm{t}, \mathrm{f})) \rightsquigarrow^*$
$\mathrm{cpt}(\Gamma_0 \vdash EG_x(P(x))(s_{2,0}), \mathrm{t}, \mathrm{f}) \rightsquigarrow^*$
$\mathrm{cpt}(\Gamma_{2,0} \vdash EG_x(P(x))(s_{1,1}), \mathrm{t}, \mathrm{cpt}(\Gamma_{2,0} \vdash EG_x(P(x))(s_{2,1}), \mathrm{t}, \mathrm{f})) \rightsquigarrow^*$
$\ldots \rightsquigarrow^*$
$\mathrm{cpt}(\Gamma_{2,0} \vdash EG_x(P(x))(s_{2,1}), \mathrm{t}, \mathrm{f}) \rightsquigarrow^*$
$\ldots \rightsquigarrow^*$
$\mathrm{cpt}(\Gamma_{2,n-1} \vdash EG_x(P(x))(s_{2,n}), \mathrm{t}, \mathrm{f}) \rightsquigarrow^*$
$\mathrm{cpt}(\Gamma_{2,n} \vdash EG_x(P(x))(s_{2,n}), \mathrm{t}, \mathrm{f}) \rightsquigarrow \mathrm{t}$

图 3.5 cpt($\vdash EG_x(P(x))(s_0), \mathrm{t}, \mathrm{f}$) 的重写步骤

在图*3.5*中，在重写 $cpt(\Gamma_{2,0} \vdash EG_x(P(x))(s_{2,1}), \mathrm{t}, \mathrm{f})$ 之前，路径 $\pi = s_{1,1}, s_{1,2}, ..., s_{1,n}$ 上的所有的状态都已被访问过一遍。而且，对于任意 $0 \le i \le n-1$，在每次重写 $cpt(\Gamma_{2,i} \vdash EG_x(P(x))(s_{2,i+1}), \mathrm{t}, \mathrm{f})$ 之前，π 上的所有状态都会重新被访问一遍。

当我们用一个全局数据结构 M 来记录访问在证明 EG 公式的过程中访问过的状态时，可以避免 π 被重复访问。在本例中，我们将所有不满足公式 $EG_x(P(x))(s)$ 的状态 s 存入到 M 中，而且当重写 $c = cpt(\Gamma \vdash EG_x(P(x))(s), c_1, c_2)$ 的时候，如果 $s \in M$，那么直接在当前的重写步骤中将 c 替换成 c_2。因此在本例中，对于任意的 $0 \le i \le n-1$，在重写 $cpt(\Gamma_{2,i} \vdash EG_x(P(x))(s_{2,i+1}), \mathrm{t}, \mathrm{f})$ 之前，π 上的所有状态都已记录在 M 中，因此不用重复被访问。

有关全局数据结构的详细解释见下一小节。

3.3.4 证明搜索策略的伪代码

在本小节，我们给出以上所述的证明搜索策略的中用到的数据结构及伪代码，如图3.6所示。

Input: A CPT c.

Output: A pair (r, t), where r is a Boolean, and t is either pt or ce.

function PROOFSEARCH
  $c := cpt^{\emptyset}(\vdash \psi, t^{\emptyset}, f^{\emptyset})$
  $pt := ce := \langle$tree with a single node: $\vdash \psi\rangle$
  $M^t := M^f :=$ visited $:= \langle$empty hash table$\rangle$
  while c has the form $cpt^{A_0}(\Gamma \vdash \phi, c_1^{A_1}, c_2^{A_2})$ **do**
    $\forall a \in A_0$, perform a
    case ϕ **is**
      $\top$: $c := c_1^{A_1}$
      $\bot$: $c := c_2^{A_2}$
      $P(s_1, ..., s_n)$:
        if $\langle s_1, ..., s_n\rangle \in P$ **then** $c := c_1^{A_1}$ **else** $c := c_1^{A_2}$ **end if**
      $\neg P(s_1, ..., s_n)$:
        if $\langle s_1, ..., s_n\rangle \in P$ **then** $c := c_1^{A_2}$ **else** $c := c_1^{A_1}$ **end if**
      $\phi_1 \wedge \phi_2$: ProveAnd$(\vdash \phi_1 \wedge \phi_2)$
      $\phi_1 \vee \phi_2$: ProveOr$(\vdash \phi_1 \vee \phi_2)$
      $EX_x(\phi_1)(s)$: ProveEX$(\vdash EX_x(\phi_1)(s))$
      $AX_x(\phi_1)(s)$: ProveAX$(\vdash AX_x(\phi_1)(s))$
      $EG_x(\phi_1)(s)$: ProveEG$(\Gamma \vdash EG_x(\phi_1)(s))$
      $AF_x(\phi_1)(s)$: ProveAF$(\Gamma \vdash AF_x(\phi_1)(s))$
      $EU_{x,y}(\phi_1, \phi_2)(s)$: ProveEU$(\Gamma \vdash EU_{x,y}(\phi_1, \phi_2)(s))$
      $AR_{x,y}(\phi_1, \phi_2)(s)$: ProveAR$(\Gamma \vdash AR_{x,y}(\phi_1, \phi_2)(s))$
    end case
  end while
  if $c = t^A$ **then**
    $\forall a \in A$, perform a
    return (true, pt)
  end if
  if $c = f^A$ **then**
    $\forall a \in A$, perform a
    return (false, ce)
  end if
end function

图 3.6 证明搜索策略的伪代码

在解释该伪代码之前，我们需要介绍公式模式的概念：对于以模态词 AF、

EG、AR、EU 开头的公式 ϕ，用 _ 来代替 ϕ 中的常量后得到即为 ϕ 的公式模式 ϕ^-。比如，$EU_{x,y}(\phi_1,\phi_2)(s)$ 的模式为 $EU_{x,y}(\phi_1,\phi_2)(\_)$；$AF_x(\phi')(s')$ 的模式为 $AF_x(\phi')(\_)$。一个公式的模式可以看作是当前公式中状态常量的上下文。

在该伪代码中，c 是当前需要被重写的 CPT；pt 和 ce 是在证明搜索的过程构造的树结构，分别指的是证明树和反例；M^t 和 M^f 记录的是，对于当前要证明的公式的每一个子公式，分别使得该子公式满足与不满足的状态的集合；visited 记录的是在对于每个以 AF、EG、AR、EU 开头的公式的证明过程中访问过的状态集合。

需要注意的是，对于伪代码中重写步骤中的每个 CPT c，我们人为关联一个动作集合 A，使得当 c 是当前需要重写的 CPT 时，执行 A 中的所有动作。将 CPT 关联动作集合的目的是为了构造证明树和反例，以及更新 M^t 和 M^f。在伪代码的初始时刻，我们分别给三个 CPT $\mathrm{cpt}(\vdash \phi, \mathrm{t}, \mathrm{f})$、t、f 均关联一个空动作集合，其中 ϕ 是要验证的公式。

另外需要注意的是，该伪代码中，三个全局变量 M^t、M^f、visited 的值都是哈希表。实际上，这三个全局变量都是对于每个公式模式记录一个状态集合，因此，M^t、M^f、visited 均可看作是以公式模式为键，以状态集合为值的哈希表。在该伪代码中，$M^t_{EG_x(\phi)(\_)}$ 用来表示一个状态集合 S，其中 $\forall s \in S$，$EG_x(\phi)(s)$ 是可证的；$M^f_{EG_x(\phi)(\_)}$ 用来表示一个集合 S，其中 $\forall s \in S$，$EG_x(\phi)(s)$ 是不可证的；$\mathrm{visited}_{EG_x(\phi)(\_)}$ 用来表示一个集合 S，其中 $\forall s \in S$，s 在证明以 $EG_x(\phi)(\_)$ 为模式的公式的过程中已被访问过。

该伪代码中的“while”循环的作用是重复重写 CPT c，直到重写到 t 或 f。该伪代码的返回值是一个由一个布尔值和一棵树组成的二元组，布尔值表示要验证的公式的满足与否，树则表示此公式的证明树或者反例。

该伪代码中，CPT 的重写方式由公式 ϕ 的形状决定，其中 ϕ 为原子公式和原子公式的非的情况在伪代码主体（表3.6）中给出，对于 ϕ 的更复杂的情况，我们在接下来的小节中依次加以讨论。

3.3.4.1 ProveAnd 和 ProveOr

该伪代码中合取公式的证明搜索如图3.7所示，其中 $\mathrm{ac}(\mathrm{t}, \mathrm{parent}, \mathrm{children})$ 表示将 children 的每个元素都作为证明树 t 上 parent 的子节点。ProveAnd 的解释如下：

- 第 4 行：当从 c′ 重写到 c_1 时，由于 $\vdash \phi_1$ 和 $\vdash \phi_2$ 均可证，那么将 $\vdash \phi_1$ 和

1: let $A_{12} = A_1 \cup \{ac(pt, \vdash \phi_1 \wedge \phi_2, \{\vdash \phi_1, \vdash \phi_2\})\}$
2: let $A_{21} = A_2 \cup \{ac(ce, \vdash \phi_1 \wedge \phi_2, \{\vdash \phi_1\})\}$
3: let $A_{22} = A_2 \cup \{ac(ce, \vdash \phi_1 \wedge \phi_2, \{\vdash \phi_2\})\}$
4: let $c' = cpt^{\emptyset}(\vdash \phi_1, cpt^{\emptyset}(\vdash \phi_2, c_1^{A_{12}}, c_2^{A_{22}}), c_2^{A_{21}})$
5: $c := c'$

图 3.7 ProveAnd($\vdash \phi_1 \wedge \phi_2$)

$\vdash \phi_2$ 都加在证明树中作为 $\vdash \phi_1 \wedge \phi_2$ 的子节点（第 1 行）。否则，当从 c' 重写到外部的 c_2 或内部的 c_2 时，由于 $\vdash \phi_1$ 或 $\vdash \phi_2$ 是不可证的，这时将 $\vdash \phi_1$ 或 $\vdash \phi_2$ 加到反例中作为 $\vdash \phi_1 \wedge \phi_2$ 的子节点（第 2、3 行）。

- 第 5 行: 将 c 重写到 c'。

ProveOr 是 ProveAnd 的对偶情况，伪代码细节如图3.8所示。

1: let $A_{22} = A_2 \cup \{ac(ce, \vdash \phi_1 \vee \phi_2, \{\vdash \phi_1, \vdash \phi_2\})\}$
2: let $A_{11} = A_1 \cup \{ac(pt, \vdash \phi_1 \vee \phi_2, \{\vdash \phi_1\})\}$
3: let $A_{12} = A_1 \cup \{ac(pt, \vdash \phi_1 \vee \phi_2, \{\vdash \phi_2\})\}$
4: let $c' = cpt^{\emptyset}(\vdash \phi_1, c_1^{A_{11}}, cpt^{\emptyset}(\vdash \phi_2, c_1^{A_{12}}, c_2^{A_{22}}))$
5: $c := c'$

图 3.8 ProveOr($\vdash \phi_1 \vee \phi_2$)

3.3.4.2 ProveEX 和 ProveAX

1: */* For notation purpose, here we refer "k" to $EX_x(\phi_1)(\_)$, and "k(s)" to $EX_x(\phi_1)(s)$. */*
2: $A_2 := A_2 \cup \{ac(ce, \vdash k(s), \{\vdash (s_1/x)\phi_1, ..., \vdash (s_n/x)\phi_1\})\}$
3: $\forall i \in \{1, ..., n\}$, let $A_{1i} = A_1 \cup \{ac(pt, \vdash k(s), \{\vdash (s_i/x)\phi_1\})\}$
4: let $c' = cpt^{\emptyset}(\vdash (s_1/x)\phi_1, c_1^{A_{11}}, cpt^{\emptyset}(...cpt^{\emptyset}(\vdash (s_n/x)\phi_1, c_1^{A_{1n}}, c_2^{A_2})...))$
5: $c := c'$

图 3.9 ProveEX($\vdash EX_x(\phi_1)(s)$)

ProveEX 的伪代码细节如图3.9所示，其中令 $\{s_1, ..., s_n\} = \text{Next}(s)$。ProveEX 与 ProveAnd 的分析过程类似，不同点在于第 4、5 行，当 c' 重写到第 i 个 c_1 的时候，$\vdash (s_i/x)\phi_1$ 应该作为 $\vdash EX_x(\phi_1)(s)$ 的子节点被加入到证明树中（第 3 行）；相反，当 c' 重写到 c_2 的时候，$\vdash (s_1/x)\phi_1, ..., \vdash (s_n/x)\phi_1$ 都应该作为 $\vdash EX_x(\phi_1)(s)$ 的子节点被加入到反例中（第 2 行）。

ProveAX 是 ProveEX 的对偶情况，伪代码细节如图3.10所示。

1: /*For notation purpose, here we refer "k" to $AX_x(\phi_1)(\_)$, and "k(s)" to $AX_x(\phi_1)(s)$.*/
2: $A_1 := A_1 \cup \{ac(pt, \vdash k(s), \{\vdash (s_1/x)\phi_1, ..., \vdash (s_n/x)\phi_1\})\}$
3: $\forall i \in \{1, ..., n\}$, let $A_{2i} = A_2 \cup \{ac(ce, \vdash k(s), \{\vdash (s_i/x)\phi_1\})\}$
4: let $c' = cpt^{\emptyset}(\vdash (s_1/x)\phi, cpt^{\emptyset}(...cpt^{\emptyset}(\vdash (s_n/x)\phi, c_1^{A_1}, c_2^{A_{2n}}), ...), c_2^{A_{21}})$
5: $c := c'$

图 3.10 ProveAX($\vdash AX_x(\phi_1)(s)$)

3.3.4.3 ProveEG and ProveAF

1: /* For notation purpose, here we refer "k" to $EG_x(\phi_1)(\_)$ and "k(s)" to $EG_x(\phi_1)(s)$,
2: and let Γ' be $\Gamma \cup \{k(s)\}$.*/
3: **if** $s \in M_k^f$ **then**
4: $c := c_2^{A_2}$
5: **else if** $s \in M_k^t$ or $k(s) \in \Gamma$ **then**
6: $c := c_1^{A_1}$
7: $M_k^t := M_k^t \cup states(\Gamma)$
8: $M_k^f := M_k^f \cup visited_k \setminus states(\Gamma)$
9: **else**
10: let $A_{20} = A_2 \cup \{ac(ce, \Gamma \vdash k(s), \{\vdash (s/x)\phi_1\})\}$
11: let $A_{2n} = A_2 \cup \{ac(ce, \Gamma \vdash k(s), \{\Gamma' \vdash k(s_1), ..., \Gamma' \vdash k(s_n)\})\}$
12: $\forall i \in \{1, ..., n\}$, let
13: $A_{1i} = A_1 \cup \{ac(pt, \Gamma \vdash k(s), \{\vdash (s/x)\phi_1, \Gamma' \vdash k(s_i)\})\}$
14: **if** $\Gamma = \emptyset$ **then**
15: $visited_k := \{s\}$
16: **else**
17: $visited_k := visited_k \cup \{s\}$
18: **end if**
19: $A_{20} := A_{20} \cup \{M_k^f := M_k^f \cup \{s\}\}$;
20: $A_{2n} := A_{2n} \cup \{M_k^f := M_k^f \cup \{s\}\}$;
21: let $c' = cpt^{\emptyset}(\vdash (s/x)\phi_1, cpt^{\emptyset}(\Gamma' \vdash k(s_1), c_1^{A_{11}}, cpt^{\emptyset}(...cpt^{\emptyset}(\Gamma' \vdash k(s_n), c_1^{A_{1n}}, c_2^{A_{2n}})...)), c_2^{A_{20}})$
22: $c := c'$
23: **end if**

图 3.11 ProveEG($\Gamma \vdash EG_x(\phi_1)(s)$)

ProveEG 的伪代码细节如图3.11所示，其中，states(□) 表示 □ 中出现的状态，并且令 $\{s_1, ..., s_n\} = Next(s)$。ProveEG 的解释如下：

- 第 3、4 行：如果已知 $EG_x(\phi_1)(s)$ 是不可证的，那么将 c 重写到 c_2。
- 第 5 – 8 行：如果已知 $EG_x(\phi_1)(s)$ 是可证的，或者可应用 merge 规则，那么

1: */* For notation purpose, here we refer "k" to $AF_x(\phi_1)(\_)$ and "k(s)" to $AF_x(\phi_1)(s)$,*
2: *and let Γ' be $\Gamma \cup \{k(s)\}$.. */*
3: **if** $s \in M_k^t$ **then**
4: $c := c_1^{A_1}$
5: **else if** $s \in M_k^f$ or $k(s) \in \Gamma$ **then**
6: $c := c_2^{A_2}$
7: $M_k^f := M_k^f \cup states(\Gamma)$
8: $M_k^t := M_k^t \cup visited_k \setminus states(\Gamma)$
9: **else**
10: let $A_{10} = A_1 \cup \{ac(pt, \Gamma \vdash k(s), \{\vdash (s/x)\phi_1\})\}$
11: let $A_{1n} = A_1 \cup \{ac(pt, \Gamma \vdash k(s), \{\Gamma' \vdash k(s_1), ..., \Gamma' \vdash k(s_n)\})\}$
12: $\forall i \in \{1, ..., n\}$, let
13: $A_{2i} = A_2 \cup \{ac(ce, \Gamma \vdash k(s), \{\vdash (s/x)\phi_1, \Gamma' \vdash k(s_i)\})\}$
14: **if** $\square = \emptyset$ **then**
15: $visited_k := \{s\}$
16: **else**
17: $visited_k := visited_k \cup \{s\}$
18: **end if**
19: $A_{10} := A_{10} \cup \{M_k^t := M_k^t \cup \{s\}\}$
20: $A_{1n} := A_{1n} \cup \{M_k^t := M_k^t \cup \{s\}\}$;
21: let $c' = cpt^{\emptyset}(\vdash (s/x)\phi_1, c_1^{A_{10}}, cpt^{\emptyset}(\Gamma' \vdash k(s_1), cpt^{\emptyset}(...cpt^{\emptyset}(\Gamma' \vdash k(s_n), c_1^{A_{1n}}, c_2^{A_{2n}})...), c_2^{A_{21}}))$
22: $c := c'$
23: **end if**

图 3.12 ProveAF($\Gamma \vdash AF_x(\phi_1)(s)$)

将 c 重写到 c_1，同时，由于 Γ 中的每个公式都是可证的，因此将 Γ 中出现的所有状态加入到 $M^t_{EG_x(\phi_1)(\_)}$ 中。另外，由于状态的访问是深度优先的，因此将所有的访问过的除了在 Γ 出现的状态之外都加入集合 $M^f_{EG_x(\phi_1)(\_)}$ 中。

- 第 21 行：CPT c' 的构造方式如下：

  1. 如果 c' 重写到外层的 c_2，那么 $\vdash (s/x)\phi_1$ 是不可证的，因此将 $\vdash (s/x)\phi_1$ 作为 $\Gamma \vdash EG_x(\phi_1)(s)$ 的子节点加到反例中（第 10 行）。另外，如果 $\Gamma \vdash EG_x(\phi_1)(s)$ 是不可证的，那么不存在以 *s* 开头的无穷路径使得该路径上的所有状态都满足 ϕ_1，在这种情况下，我们将 *s* 加入到 $M^f_{EG_x(\phi_1)(\_)}$ 中（第 19、20 行）。
  2. 如果 c' 重写到内层的 c_2，那么 $\Gamma' \vdash EG_x(\phi_1)(s_1), ..., \Gamma' \vdash EG_x(\phi_1)(s_n)$ 都是不可证的，因此将其都作为 $\Gamma \vdash EG_x(\phi_1)(s)$ 的子节点加入到反例中

（第 11 行）。

3. 如果 c′ 重写到第 i 个 c_1，那么 $\vdash (s/x)\phi_1$ 和 $\Gamma' \vdash EG_x(\phi_1)(s_i)$ 都是可证的，因此将其都作为 $\Gamma \vdash EG_x(\phi_1)(s)$ 的子节点加入到证明树中（第 12、13 行）。

- 第 22 行：将 c 重写到 c′。

ProveAF 是 ProveEG 的对偶情况，伪代码细节如图3.12所示。

3.3.4.4 ProveEU 和 ProveAR

ProveEU 的伪代码细节要比 ProveEG 更加复杂。原因是，对于后者我们只需找到一个满足某个公式的环即可，而对于前者我们需要找到一条终点状态满足某公式的有穷路径，而且在找到这条路径之前可能已访问若干个环。ProveEU 的伪代码细节如图3.13所示。在 ProveEG 中，我们令 $\{s_1, ..., s_n\} = \text{Next}(s)$；令 reachable(S) 表示能经过有穷路径到达 S 的某个状态的状态集合。ProveEU 的解释如下：

- 第 3、4 行：如果 $EU_{x,y}(\phi_1, \phi_2)(s)$ 是可证的，那么将 c 重写到 c_1。
- 第 5、6 行：如果 $EU_{x,y}(\phi_1, \phi_2)(s)$ 是不可证的，或者 $EU_{x,y}(\phi_1, \phi_2)(s) \in \Gamma$，那么将 c 重写到 c_2。
- 第 26、27 行：CPT c′ 的构造方式如下：

1. 如果 c′ 将来能重写到第一个 c_1，那么 $\vdash (s/y)\phi_2$ 是可证的，因此将其作为 $\Gamma \vdash EU_{x,y}(\phi_1, \phi_2)(s)$ 的子节点加到证明树中（第 9 行）。与此同时，由于 $EU_{x,y}(\phi_1, \phi_2)(s)$ 是可证的，那么对于任意访问过的状态 s'，如果存在以 s' 为起点的有穷路径而 s 在这条路径上，那么 $EU_{x,y}(\phi_1, \phi_2)(s')$ 也是可证的。因此，可将状态集合 reachable(states(Γ) ∪ {s}) 中的所有状态加入到 $M^t_{EU_{x,y}(\phi_1,\phi_2)(\_)}$ 中（第 10 行）。每个 reachable(states(Γ) ∪ {s}) 中的状态 s′ 或者在 states(Γ) ∪ {s} 中，或者在与 states(Γ) ∪ {s} 重叠的某个环上。因此，通过记录并选取与 states(Γ) ∪ {s} 重叠的所有的环，就可以计算状态集合 reachable(states(Γ) ∪ {s}) 了。然后，我们将所有被访问过的状态加入到 $M^f_{EU_{x,y}(\phi_1,\phi_2)(\_)}$ 中，然后去掉可经过有穷路径到达 states(Γ) ∪ {s} 中某个状态的状态（第 11 行）。

1: /* For notation purpose, here we refer "k" to $EU_{x,y}(\phi_1,\phi_2)(\_)$ and "k(s)" to $EU_{x,y}(\phi_1,\phi_2)(s)$,
2: and let Γ' be $\Gamma \cup \{k(s)\}$. */
3: **if** $s \in M_k^t$ **then**
4: $c := c_1^{A_1}$
5: **else if** $s \in M_k^f$ or $k_s \in \Gamma$ **then**
6: $c := c_2^{A_2}$;
7: **else**
8: let $A_{10} = A_1 \cup \{$
9: $ac(pt, \Gamma \vdash k(s), \{\vdash (s/y)\phi_2\})$,
10: $M_k^t := M_k^t \cup reachable(states(\Gamma) \cup \{s\})$,
11: $M_k^f := (M_k^f \cup visited_k) \setminus reachable(states(\Gamma) \cup \{s\})$
12: let $A_{20} = A_2 \cup \{ac(ce, \Gamma \vdash k(s), \{\vdash (s/x)\phi_1, \vdash (s/y)\phi_2\})\}$
13: let $A_{2n} = A_2 \cup \{ac(ce, \Gamma \vdash k(s), \{\Gamma' \vdash k(s_1), ..., \Gamma' \vdash k(s_n)\})\}$
14: $\forall i \in \{1, ..., n\}$, let
15: $A_{1i} = A_1 \cup \{$
16: $ac(pt, \Gamma \vdash k(s), \{\Gamma' \vdash k(s_i)\})$,
17: $M_k^t := M_k^t \cup reachable(states(\Gamma') \cup \{s_i\})$,
18: $M_k^f := (M_k^f \cup visited_k) \setminus reachable(states(\Gamma') \cup \{s_i\})\}$
19: **if** $\Gamma = \emptyset$ **then**
20: $visited_k := \{s\}$
21: **else**
22: $visited_k := visited_k \cup \{s\}$
23: **end if**
24: $A_{20} := A_{20} \cup \{M_k^f := M_k^f \cup \{s\}\}$
25: $A_{2n} := A_{2n} \cup \{M_k^f := M_k^f \cup \{s\}\}$
26: let $c' = cpt^{\emptyset}(\vdash (s/y)\phi_2, c_1^{A_{10}}, cpt^{\emptyset}(\vdash (s/x)\phi_1, cpt^{\emptyset}(\Gamma' \vdash k(s_1),$
27: $c_1^{A_{11}}, cpt^{\emptyset}(...cpt^{\emptyset}(\Gamma' \vdash k(s_n), c_1^{A_{1n}}, c_2^{A_{2n}})...)), c_2^{A_{20}}))$
28: $c := c'$
29: **end if**

图 3.13 ProveEU($\Gamma \vdash EU_{x,y}(\phi_1,\phi_2)(s)$)

2. 如果 c' 能重写到第 i 个其他的 c_1，那么 $\vdash (s/x)\phi_1$ 和 $\Gamma' \vdash EU_{x,y}(\phi_1,\phi_2)(s_i)$ 是可证的，因此将其都作为 $\Gamma \vdash EU_{x,y}(\phi_1,\phi_2)(s)$ 的子节点加入到证明树中（第 16 行）。与此同时，$reachable(states(\Gamma') \cup \{s_i\})$ 中的所有状态都应加到 $M^t_{EU_{x,y}(\phi_1,\phi_2)(\_)}$ 中（第 17 行）；所有被访问过的状态都加到 $M^f_{EU_{x,y}(\phi_1,\phi_2)(\_)}$ 中并且去掉可经过有穷路径到达 $states(\Gamma') \cup \{s_i\}$ 中某个元素的状态（第 18 行）。

3. 如果 c' 重写到外层的 c_2，那么 $\vdash (s/x)\phi_1$ 和 $\vdash (s/y)\phi_2$ 是不可证的，因此将其都作为 $\Gamma \vdash EU_{x,y}(\phi_1,\phi_2)(s)$ 的子节点加入到反例中（第 12 行）。

1: */* For notation purpose, here we refer "k" to* $AR_{x,y}(\phi_1, \phi_2)(\_)$ *and "k(s)" to* $AR_{x,y}(\phi_1, \phi_2)(s)$*, and let* Γ' *be* $\Gamma \cup \{k(s)\}$*. */*
2: **if** $s \in M_k^f$ **then**
3: $c := c_2^{A_2}$
4: **else if** $s \in M_k^t$ or $k(s) \in \Gamma$ **then**
5: $c := c_1^{A_1}$;
6: **else**
7: let $A_{10} = A_1 \cup ac(pt, \Gamma \vdash k(s), \{\vdash (s/x)\phi_1, \vdash (s/y)\phi_2\})\}$
8: let $A_{1n} = A_1 \cup \{ac(pt, \Gamma \vdash k(s), \{ \Gamma' \vdash k(s_1), ..., \Gamma' \vdash k(s_n)\})\}$
9: let $A_{20} = A_2 \cup \{$
10: $ac(ce, \Gamma \vdash k(s), \{\vdash (s/y)\phi_2\})$,
11: $M_k^f := M_k^f \cup reachable(states(\Gamma) \cup \{s\})$,
12: $M_k^t := (M_k^t \cup visited_k) \setminus reachable(states(\Gamma) \cup \{s\})$
13: $\forall i \in \{1, ..., n\}$, let
14: $A_{2i} = A_2 \cup \{$
15: $ac(ce, \Gamma \vdash k(s), \{\vdash (s/x)\phi_1, \Gamma' \vdash k(s_i)\})$,
16: $M_k^f := M_k^f \cup reachable(states(\Gamma') \cup \{s_i\})$,
17: $M_k^t := (M_k^t \cup visited_k) \setminus reachable(states(\Gamma') \cup \{s_i\})\}$
18: **if** □ $= \emptyset$ **then**
19: $visited_k := \{s\}$
20: **else**
21: $visited_k := visited_k \cup \{s\}$
22: **end if**
23: $A_{10} := A_{10} \cup \{M_k^t := M_k^t \cup \{s\}\}$
24: $A_{1n} := A_{1n} \cup \{M_k^t := M_k^t \cup \{s\}\}$
25: let $c' = cpt^{\emptyset}(\vdash (s/y)\phi_2, cpt^{\emptyset}(\vdash (s/x)\phi_1, c_1^{A_{10}}, cpt^{\emptyset}(\Gamma' \vdash k(s_1),$
26: $cpt^{\emptyset}(...cpt^{\emptyset}(\Gamma' \vdash k(s_n), c_1^{A_{1n}}, c_2^{A_{2n}})...), c_2^{A_{21}})), c_2^{A_{20}})$
27: $c := c'$
28: **end if**
29:

图 3.14 ProveAR($\Gamma \vdash AR_{x,y}(\phi_1, \phi_2)(s)$)

4. 如果 c' 重写到内层的 c_2，那么 $\Gamma' \vdash EU_{x,y}(\phi_1, \phi_2)(s_1)$, ..., $\Gamma' \vdash EU_{x,y}(\phi_1, \phi_2)(s_n)$ 都是不可证的，因此将其全部作为 $\Gamma \vdash EU_{x,y}(\phi_1, \phi_2)(s)$ 的子节点加入到反例中（第 13 行）。

5. 如果 $\Gamma \vdash EU_{x,y}(\phi_1, \phi_2)(s)$ 是不可证的，那么将 *s* 加入到 $M^f_{EU_{x,y}(\phi_1,\phi_2)(\_)}$ 中，并以此在对这个 *EU* 公式的证明搜索中避免重复访问 s（第 24、25 行）。值得注意的是 s 可从 $M^f_{EU_{x,y}(\phi_1,\phi_2)(\_)}$ 被移除（第 11 – 18 行）。这种情况只当存在 Γ' 和 s' 使得 $\Gamma' \vdash EU_{x,y}(\phi_1, \phi_2)(s')$ 是可证的，$s', \vdash (s'/y)\phi_2$ 是可证的，以及 $s \in reachable(states(\Gamma') \cup \{s'\})$ 的时候发生。

- 第 28 行：将 c 重写到 c′。

ProveAR 是 ProveEU 的对偶情况，伪代码细节如图3.14所示。

3.4 公平性性质的验证

在并发系统的形式化验证中，公平性是比较重要的一个方面。在状态搜索中，公平性前提通常可以避免一些不切实际的无穷的路径，即忽略掉一些不切实际的系统行为[3]。然而，无论是 CTL 还是 CTL_P 均无法在语法中表示公平性假设。这是由于在这两种逻辑中缺乏描述特定路径的性质的机制。即使如此，我们仍然可以基于本章中介绍的证明搜索策略来完成带有公平性前提的 CTL_P 公式的验证。

在本节中，我们介绍 CTL_P 中公平性的定义，以及如何完成对公平性性质的验证。

3.4.1 公平性

在介绍公平性之前，让我们首先看如下的例子。

例子 3.6 (进程调度的公平性). 假设有一个具有 $n+1$ 个进程的计算机系统，其中，进程 *Server* 为服务端进程，进程 $P_1, ..., P_n$ 为 n 个客户端进程。在系统的运行过程中，每个客户端进程可向服务端进程发出请求，而服务端进程则会根据收到的请求来为相应的客户端进程提供服务。该系统的一个可能的进程调度策略如下：系统从 P_1 到 P_n 依次检查是否有客户端进程发出请求，如果是，则 *Server* 为第一个系统找到的发出请求的客户端进程提供服务，当 *Server* 服务完成后，系统重复上述过程。

在该系统中，如果客户端进程 P_1 持续向服务端进程 *Server* 发出请求，那么根据系统的进程调度策略，*Server* 将持续为 P_1 提供服务，而忽略其他客户端进程的请求。这样的进程调度策略就被称为是不公平的。在一个公平的进程调度策略中，当一个客户端进程发出请求后，服务端进程总会在有限时间内提供服务，而不会让发出请求的客户端进程无限等待。比如，在轮询调度（*Round-Robin Scheduling*）[42] 中，系统按照轮询的顺序依次检查每个进程是否发出请求，而且当处理完一个进程的请求后，再按照顺序检查下一个进程。经过轮询调度的进程不会饿死（即无限等待），因此轮询调度是一种公平的进程调度策略。

本文中公平性的定义与 McMillan 提出的公平性定义[8] 一致，即一个无穷路径是公平的指的是某种性质在该路径上无穷次成立。一个带有公平性前提的性

质通常具有 $E_C f$ 或者 $A_C f$ 形式，其中 C 表示一个 CTL_P 公式的集合，而 f 表示路径的性质。在本文中，$E_C G(\phi)(s)$ 在给定的 Kripke 模型中成立当且仅当存在以 s 为起始状态的一个无穷路径，集合 C 中的每个公式在该路径上的无穷次成立，而且在该路径上的每个状态上 ϕ 都成立，而且集合 C 中的每个公式 ϕ'，ϕ' 在 π 上的无穷个状态上都是成立的；$A_C F(\phi)(s)$ 在给定的 Kripke 模型中是有效的当且仅当对于所有以 s 为起始状态的无穷路径，集合 C 中的每个公式都在每条路径上无穷次成立，而且每条路径上都存在一个状态，使得 ϕ 在该状态上成立。另外，与 McMillan 的定义类似，带有其他模态词的公平性性质可用 $E_C G$ 与 $A_C F$ 公式进行定义：

$$E_C X_x(\phi)(t) = EX_x(\phi \wedge E_C G_x(\top)(x))(t)$$

$$A_C X_x(\phi)(t) = AX_x(\phi \vee A_C F_x(\bot)(x))(t)$$

$$E_C U_{x,y}(\phi_1, \phi_2)(t) = EU_{x,y}(\phi_1, \phi_2 \wedge E_C G_z(\top)(y))(t)$$

$$A_C R_{x,y}(\phi_1, \phi_2)(t) = AR_{x,y}(\phi_1, \phi_2 \vee A_C F_z(\bot)(y))(t)$$

3.4.2 验证带有公平性假设的公式

根据上一小节内容可知，要使得公式 $E_C G_x(\phi)(s)$ 成立，只需找到一条以 s 为起始状态的公平的无穷路径使得 ϕ 在这条路径上永远成立；要使得公式 $A_C F(\phi)(s)$ 成立，只需确定不存在一条以 s 为其实状态的无穷路径使得 ϕ 在这条路径上永远不成立。因此，要验证公平性性质，只需要一个机制来判断公平的无穷路径的存在与否。若要达到此目的，我们需要证明以下两个命题。

命题 3.8. *令 C 为一个 CTL_P 公式的集合，而且 $\sigma = s_0, s_1, \ldots$ 是一条无穷路径，其中对于任意 $i \geq 0$ 都有 $s_i \rightarrow s_{i+1}$。如果 C 中的每个公式都在 σ 中无穷次成立，那么一定存在一条有穷路径 $\sigma_f = s'_0, s'_1, \ldots, s'_n$，使得对任意 $0 \leq i \leq n-1$ 都有 $s'_j \rightarrow s'_{j+1}$ 而且存在 $0 \leq p \leq n-1$ 使得 $s'_n = s'_p$，每个状态 s'_j 都在 σ 出现，而且对于 C 中的任意公式，都存在某个状态 s'_q 使得该公式成立，其中 $p \leq q \leq n$。*

证明. 由于 Kripke 模型中的状态是有穷的，因此存在状态集合 S 使得每个状态 $s \in S$ 都在 σ 中无穷次出现，而且对于每个公式 $f \in C$ 都存在 S 中的某个状态使得 f 成立。否则，如果对于任意状态集合 S' 中的状态在 σ 无穷次出现，都存在公式 $f \in C$ 使得 f 在 S' 中的所有状态上都不成立，那么 f 一定在所有在 σ 无穷次

出现的状态上都不成立，因此 f 无法在 σ 中无穷次成立。假设 $S = \{s_{i_1}, s_{i_2}, ..., s_{i_k}\}$，而且 $i_1 \leq i_2 \leq ... \leq i_k$，那么令 $s'_p = s_{i_1}$，$s'_n = s_{i_{k'}}$ 使得 $i_{k'} \geq i_k$ 以及 $s_{i_{k'}} = s_{i_1}$. □

命题 3.9. 令 C 一个 CTL_P 公式的集合，而且 $\sigma_f = s_0, s_1, ..., s_n$ 是一条有穷路径，其中对任意 $0 \leq i \leq n-1$ 都有 $s_i \rightarrow s_{i+1}$，而且存在 $0 \leq p \leq n$ 使得 $s_p = s_n$，而且对任意 C 中的公式，都存在 s_p 和 s_n 之间的某个状态使得该公式在该状态上成立。那么一定存在一条无穷路径 $\sigma = s'_0, s'_1, ...$ 使得对任意 $i \geq 0$ 都有 $s'_i \rightarrow s'_{i+1}$，而且每个状态 s'_j 都在 $s_0, s_1, ..., s_n$ 中出现，而且 C 中的每个公式都在该无穷路径上无穷次成立。

证明. 令 $\sigma = s_0, ..., s_{p-1}, s_p, ..., s_{n-1}, ...$ 即可得证。 □

由命题3.8与命题3.9可知，我们可以在证明搜索中利用 merge 来判断公平的无穷路径的存在与否：存在一条具有公平性前提 C 的无穷路径当且仅当存在一个 merge，使得 C 中的每个公式都在该 merge 上的环中的某个状态上成立。

这种判断公平性性质的可满足性的过程可用以下重写规则来刻画。

$\text{cpt}(\Gamma \vdash A_C F_x(\phi)(s), c_1, c_2) \rightsquigarrow c_2$ $[A_C F_x(\phi)(s) \in \Gamma,$ and $\forall \psi \in C, \exists s \in cycle(\Gamma), \psi$ holds on $s]$
$\text{cpt}(\Gamma \vdash A_C F_x(\phi)(s), c_1, c_2) \rightsquigarrow c_1$ $[A_C F_x(\phi)(s) \in \Gamma,$ and $\exists \psi \in C, \forall s \in cycle(\Gamma), \psi$ does not hold on $s]$
$\text{cpt}(\Gamma \vdash A_C F_x(\phi)(s), c_1, c_2) \rightsquigarrow$ $\text{cpt}(\vdash (s/x)\phi, c_1, \text{cpt}(\Gamma' \vdash A_C F_x(\phi)(s_1), \text{cpt}(...\text{cpt}(\Gamma' \vdash A_C F_x(\phi)(s_n), c_1, c_2)..., c_2), c_2))$ $[\{s_1, ..., s_n\} = \text{Next}(s), A_C F_x(\phi)(s) \notin \Gamma,$ and $\Gamma' = \Gamma, A_C F_x(\phi)(s)]$
$\text{cpt}(\Gamma \vdash E_C G_x(\phi)(s), c_1, c_2) \rightsquigarrow c_1$ $[E_C G_x(\phi)(s) \in \Gamma,$ and $\forall \psi \in C, \exists s \in cycle(\Gamma), \psi$ holds on $s]$
$\text{cpt}(\Gamma \vdash E_C G_x(\phi)(s), c_1, c_2) \rightsquigarrow c_2$ $[E_C G_x(\phi)(s) \in \Gamma,$ and $\exists \psi \in C, \forall s \in cycle(\Gamma), \psi$ does not hold on $s]$
$\text{cpt}(\Gamma \vdash E_C G_x(\phi)(s), c_1, c_2) \rightsquigarrow$ $\text{cpt}(\vdash (s/x)\phi, \text{cpt}(\Gamma' \vdash E_C G_x(\phi)(s_1), c_1, \text{cpt}(...\text{cpt}(\Gamma' \vdash E_C G_x(\phi)(s_n), c_1, c_2)...)), c_2)$ $[\{s_1, ..., s_n\} = \text{Next}(s), E_C G_x(\phi)(s) \notin \Gamma,$ and $\Gamma' = \Gamma, E_C G_x(\phi)(s)]$

在上述重写规则中，$cycle(\Gamma)$ 代表在 Γ 中出现的环上的状态集合。当同时应用上述重写规则与图3.4中的重写规则来进行证明搜索时，即可实现对于公平性性质的验证。

3.5 本章总结

在本章中，我们首先定义了一个针对计算树逻辑 CTL 的扩展：CTL_P 逻辑；接着定义了一个针对 CTL_P 的证明系统 SCTL 并证明了 SCTL 的可靠性和完备

性；然后，我们针对 SCTL 提出了一种基于连续的证明搜索策略，并以重写系统的形式给出，同时证明了该证明搜索策略的可终止性与正确性；最后，我们给出一个完全自动化的、线性的证明搜索算法，并以伪代码的形式给出。通过本章内容可知，我们不仅扩展了在传统的模型检测中常用的 CTL 逻辑，还给出了在扩展后的逻辑中的一种自动地、高效的证明搜索算法。由此，我们则可设计一种基于扩展后的逻辑的定理证明工具，即本章的内容可作为下一章内容的理论基础。

第 4 章　定理证明工具 SCTLProV

在上一章中，我们针对 SCTL 证明系统设计了一种证明搜索策略并给出了证明搜索算法的伪代码。在本章中，我们介绍对 SCTL 的证明搜索算法的一个编程实现—定理证明工具 SCTLProV（图4.1）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），如果该公式不可证，则输出该公式的非的证明（或者只输出 False）。

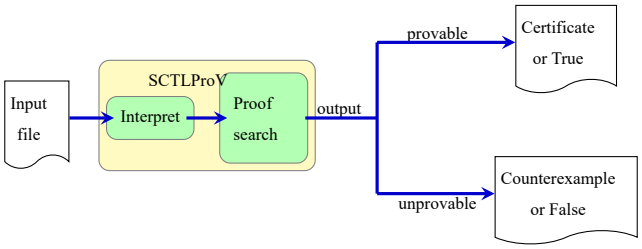


图 4.1 SCTLProV

4.1　SCTLProV 的输入语言

本节讨论的是 SCTLProV 的输入语言。在 SCTLProV 中，输入语言的作用是定义 SCTLProV 的输入文件内容，即对计算机系统进行形式化建模，并定义要验证的性质，然后，SCTLProV 可在该模型上验证该性质。

在绝大多数传统的 CTL 模型检测工具的输入语言中，Kripke 模型中的每个状态通常用固定个数的值来表示，其中值的类型通常相对较简单，比如布尔或枚举类型等。这类输入语言通常无法将状态表示为复杂的数据结构，比如任意有限长度的链表。在 SCTLProV 的输入语言中，我们可以利用多种类型的表达式来定义 Kripke 模型的状态，比如元组（tuple）、记录（record）、变体（variant）或者任意有限长度的链表等。当用这种表达式来表示状态时，状态之间的迁移则可用一个从状态映射到它的所有后继的函数来表示，而且原子公式（即状态的性质或者状态之间的关系）则可用一个从一个或者多个状态映射到一个布尔值的函数来表示。此类输入语言通常可以用来对复杂的系统进行建模，比如第4.3.2节中的小型飞机场运输系统。SCTLProV 的输入语言的详细描述见附录A。

4.2 其他 CTL 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 模型检测方法的对比。

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[4,8] 以及 NuSMV 的扩展 NuXMV[43]。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 $\text{lfp} = \mu Y.(\phi \vee EXY)$，然后，若 $s_0 \in \text{lfp}$，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。

即时（On-the-fly）模型检测 在验证时序逻辑公式的正确性时，利用即时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 即时模型检测方法[44,45] 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 即时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 即时模型检测方法不同，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[38,39,46]，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即时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 SCTLProV 的一个递归版本，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4.3节。

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[10]：

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

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：

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

其中，S 是在证明过程中已经访问过的状态集合。

4.3 案例分析

在本节中，我们讨论 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。

4.3.1 案例一：进程互斥问题

例子 4.1 (进程互斥问题[47]). 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4.2所示，其中 $flag$ 是一个布尔变量，指

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

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 4.2 进程 *A* 和进程 *B* 的一个简单描述

如图*4.3*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}
|- 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}
```

```
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 4.3 输入文件“mutual.model”

```
{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*4.4*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[47]，可以使程序满足进程互斥性质，修改后的程序如图*4.5*所示。修改后的程序可形式化描述为图*4.6*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.
```

4.3.2 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）[48,49]。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。

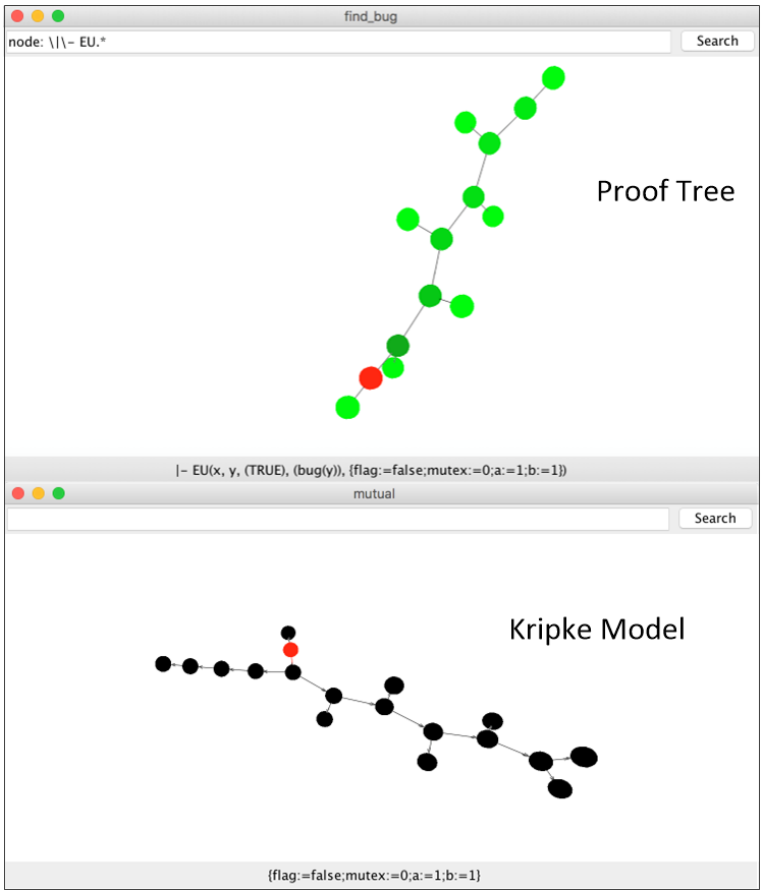


图 4.4 进程互斥问题的验证中证明树和模型的可视化

```
/* 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 4.5 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4.7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 4.6 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；

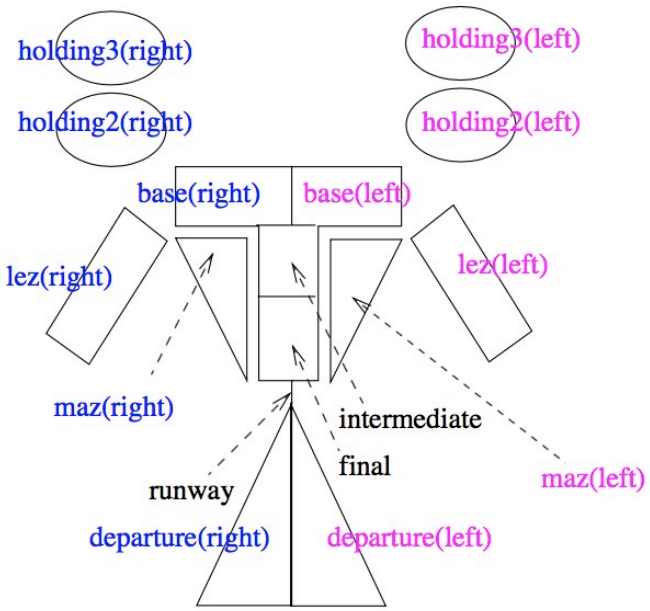


图 4.7 飞机场自我控制区域的划分（以飞行员视角区分左右）

- **maz(right/left)**：重新降落航线（右/左）；
- **departure(right/left)**：起飞航线（右/左）。

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 $\mathrm{CTL}_P(\mathcal{M})$ 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[2]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 Carreño 提出了对 SATS 系统的一个简化版的 PVS 建模[48]，在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[48]，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。

4.4 与相关工具的实验结果对比

在本节中，我们分别对比 SCTLProV 和其他 5 个工具在 4 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[3]下载。

[2] https://github.com/sctlprov/sctlprov_sats

[3] https://github.com/sctlprov/sctlprov_benchmarks

4.4.1 随机生成的布尔程序的验证

本小节包含两个测试集：测试用例集一在首次提出[5]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。测试集一中包含 2880 个测试用例，测试集二中包含 5760 个测试用例。测试集一、二中的每个测试用例的 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。

以下分别介绍这两个测试集。

4.4.1.1 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4.8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 4.8 测试集一中需要验证的性质 P_{01} 至 P_{12}

4.4.1.2 测试集二

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、72、252、504、1008，将并发顺序进程测试用例中 b 的值分别扩大为 24、28、32、252、504、1008。由此，我们得到包含 5760 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。

4.4.1.3 实验数据

在测试集一、二上，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4.1与表4.2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%）个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4.9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4.10所示。

程序类型	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 4.1 测试集一中 5 个工具能成功验证的测试用例个数

程序类型	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 4.2 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

测试集二的实验结果 由表4.3与表4.4可知：在测试集二的 5760 个测试用例中，iProver Modulo、Verds、NuSMV、NuXMV、SCTLProV 分别能验证 2748（44.7%）、2226（38.6%）、728（12.6%）、736（12.8%）、4441（77.1%）个测试用例；同时SCTLProV 分别在 4441（77.1%）、4438（77.0%）、4432（76.9%）、4432（76.9%）个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4.9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4.10所示。

程序类型	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%)
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（$b = 24$）	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%)
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	2748(47.7%)	2226(38.6%)	728(12.6%)	736(12.8%)	4441(77.1%)

表 4.3 测试集二中 5 个工具能成功验证的测试用例个数

程序类型	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%)
CP（$b = 252$）	371(77.3%)	371(77.3%)	371(77.3%)	371(77.3%)
CP（$b = 504$）	335(69.8%)	335(69.8%)	335(69.8%)	335(69.8%)
CP（$b = 1008$）	278(57.9%)	278(57.9%)	278(57.9%)	278(57.9%)
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%)
CSP（$b = 252$）	312(65.0%)	312(65.0%)	312(65.0%)	312(65.0%)
CSP（$b = 504$）	295(61.5%)	295(61.5%)	295(61.5%)	295(61.5%)
CSP（$b = 1008$）	253(52.7%)	253(52.7%)	253(52.7%)	253(52.7%)
Sum	4441(77.1%)	4438(77.0%)	4432(76.9%)	4432(76.9%)

表 4.4 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

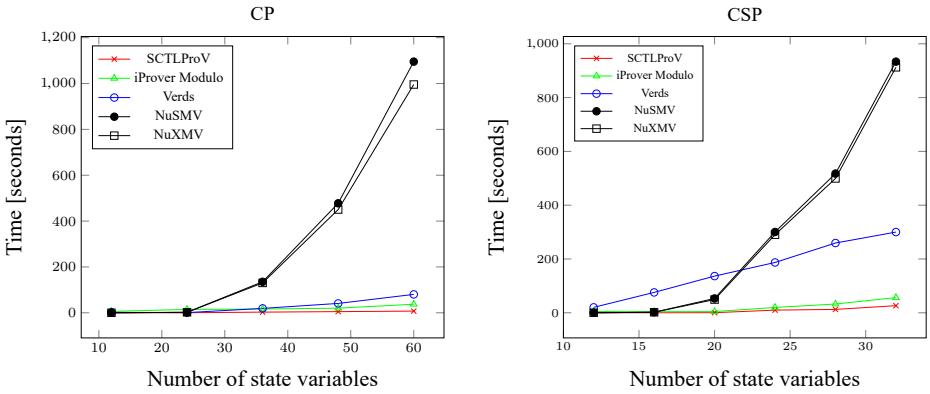


图 4.9 在测试集一、二上各个工具的平均占用时间

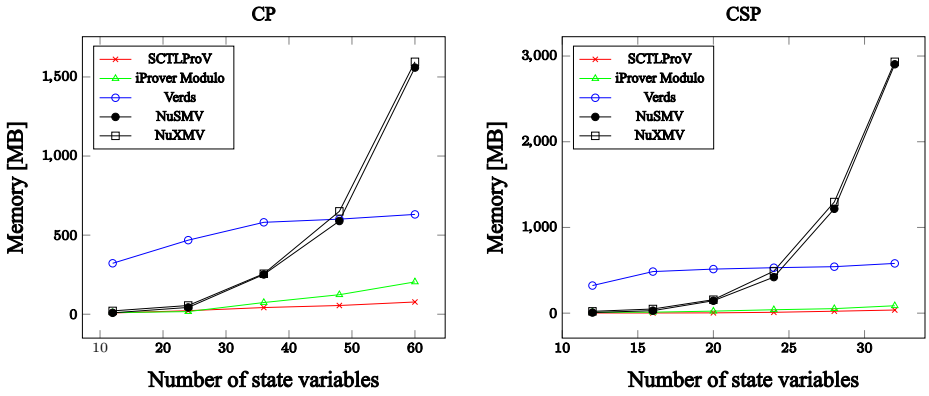


图 4.10 在测试集一、二上各个工具的平均占用内存

4.4.1.4 连续 vs. 递归

传统的即时模型检测工具通常使用递归算法来进行公式的证明和状态空间的搜索。不同于递归算法，SCTLProV 的验证算法是连续传递风格（Continuation-Passing Style，简称 CPS），CPS 的应用可以大大减少栈的操作，从而节省验证所需的时间。为了对比使用连续传递风格的算法和递归算法的效率，我们对比了 SCTLProV 和 SCTLProV$_R$ 分别在测试集一、二上的实验数据。其中 SCTLProV$_R$ 与 SCTLProV 的唯一不同是使用递归算法来证明公式和搜索状态空间。如表4.5所示，SCTLProV 能成功验证的测试用例个数比 SCTLProV$_R$ 多 10%，而且 SCTLProV 在绝大多数能成功运行的测试用例中比 SCTLProV$_R$ 所用时间短。如图4.11所示，随着状态变量数的增加，SCTLProV$_R$ 的平均运行时间多于 SCTLProV，而且时间的变化幅度更大。

测试集	Solvable		t(SCTLProV) < t(SCTLProV$_R$)
	SCTLProV	SCTLProV$_R$	
一	2862(99.4%)	2682(93.1%)	2598(90.2%)
二	4446(77.2%)	3826(66.4%)	3841(71.9%)

表 4.5 在测试集一、二上 SCTLProV 和 SCTLProV$_R$ 的实验数据对比

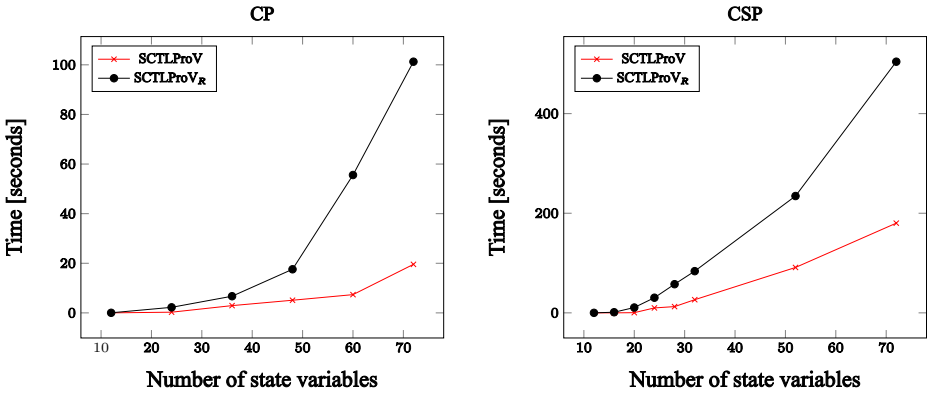


图 4.11 SCTLProV 和 SCTLProV$_R$ 平均运行时间

4.4.2 公平性性质的验证

在本小节，我们来对比 SCTLProV 和 Verds、NuSMV、NuXMV 在验证公平性性质时的效率。此次对比没有考虑 iProver Modulo，这是因为 iProver Modulo 无法验证公平性性质。此次对比所用的所有测试用例（测试集四）同样分为两种：互斥算法和环算法[4]。下面我们介绍这两种测试用例。

4.4.2.1 测试集三

测试集三种包含两类测试用例：互斥算法和环算法。

每个互斥算法包含 n 个进程，n 个进程的调度方式如下：对于 $0 \leq i \leq n-2$，进程 $i+1$ 的迁移在进程 i 之后，进程 0 的迁移在进程 $n-1$ 之后。互斥算法中 n 的取值范围为 $\{6, ..., 51\}$。要验证的性质如表4.12所示，其中 non_i，try_i 以及 cri_i 分别表示进程 i 的内部状态为 noncritical，trying 以及 critical。所有的性质必须在公平性的前提下进行验证，即在互斥算法的执行过程中没有进程饿死（永远处在等待状态）。

每个环算法包含 n 个进程，n 个进程的调度方式如下：对于 $1 \leq i \leq n-1$，进程 i 的内部状态由进程 $i-1$ 的输出决定，进程 0 的内部状态由进程 $n-1$ 的输

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

性质	互斥算法公式
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	$AG(cri_1 \Rightarrow Acri_1U(\neg cri_1 \wedge A\neg cri_1Ucri_0))$

图 4.12 互斥算法的性质

出决定；每个进程的输出取决于它的内部状态；每个进程的内部状态由 5 个布尔变量的值表示；每个进程的输出由 1 个布尔变量表示。环算法中 n 的取值范围为 $\{3, ..., 10\}$。要验证的性质如表4.13所示，其中 out_i 表示进程 i 的输出为布尔值 true。所有的性质必须在公平性的前提下进行验证，即在环算法的执行过程中没有进程饿死（永远处在等待状态）。

性质	环算法公式
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 4.13 环算法的性质

测试集三的实验结果 如表4.6所示，在本测试集的 262 个测试用例中，Verds、NuSMV、NuXMV、SCTLProV 分别能解决 152 (58.0%)、71 (27.1%)、71 (27.1%)、211 (80.2%) 个测试用例。如表4.7所示，SCTLProV 分别在 200 (76.3%)、211 (80.2%)、211 (80.2%) 个测试用例上占用的时间和空间少于 Verds、NuSMV、NuXMV。测试集三上各个工具的详细实验数据如附录C中表C.1和表C.2所示。

4.4.3 工业级测试用例的验证

在本小节中，我们在工业级测试用例上（测试集四）对比 SCTLProV 与其他工具的效率。不同于测试集一、二、三，本次对比所用的测试用例均为符号迁移系统（Labeled Transition System，简称 LTS），而且所有的 LTS 均以 BCG（Binary-Coded Graph）格式表示，其中 BCG 格式可以用来表示较大的状态空间。

程序类型	Verds	NuSMV	NuXMV	SCTLProV
互斥算法	136 (59.1%)	50 (21.7%)	50 (21.7%)	191 (83.0%)
环算法	16 (50.0%)	21 (65.6%)	21 (65.6%)	20 (62.5%)
总和	152(58.0%)	71(27.1%)	71(27.1%)	211 (80.5%)

表 4.6 测试集三中各个工具能成功验证的测试用例的个数

程序类型	Verds	NuSMV	NuXMV
互斥算法	187 (81.3%)	191 (83.0%)	191 (83.0%)
环算法	13 (40.6%)	20 (62.5%)	20 (62.5%)
总和	200(76.3%)	211(80.5%)	211(80.5%)

表 4.7 测试集三中 SCTLProV 占用资源更少的的测试用例的个数

测试集四是形式化验证工具包 CADP[7] 的一部分，也被称作 VLTS（Very Large Transition Systems）测试集。不同于本文中其他的形式化验证工具，CADP 是专门验证基于动作的形式化系统的工具，比如符号迁移系统、马尔可夫链等。测试集四中的例子均由对不同的传输协议以及并发系统的建模而得到的，其中许多例子是对工业级系统的建模[5]。

测试集四中共有 40 个测试用例，针对每个测试用例我们分别验证有无死锁与活锁。由于 SCTLProV 是基于 Kripke 模型的验证工具，因此在验证之前，我们需将每个 LTS 转换到相应的 Kripke 模型，然后在转换后的 Kripke 模型中进行验证。

给定一个 LTS $\mathcal{L} = \langle s_0, S, Act, \rightarrow \rangle$，其中 s_0 是初始状态；s 是一个有穷的状态集合；Act 是一个有穷的动作集合；$\rightarrow \subseteq S \times Act \times S$ 是迁移规则。那么 $\mathcal{L}$ 到相应的 Kripke 模型 $\mathcal{M} = \langle s'_0, S', \longrightarrow, \mathcal{P} \rangle$ 的转换过程如下：

- 令 s'_0 为 $(s_0, \cdot)$，其中 $\cdot \notin Act$ 是一个特殊的动作符号。
- 分别将 $(s_d, \cdot)$ 与 S' 与 $(s_d, \cdot) \longrightarrow (s_d, \cdot)$ 添加到 S' 与 $\longrightarrow$ 中，其中 $(s_d, \cdot)$ 区分于 S' 中的所有其他状态。
- 重复以下步骤直到没有更多的状态和迁移被添加到 $\mathcal{M}$ 中：
  - 如果 $\mathcal{L}$ 中存在一个迁移 $s_1 \xrightarrow{a} s_2$，那么对于所有的 $(s_1, b) \in S'$，将

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

$(s_1, b) \longrightarrow (s_2, a)$ 添加到 $\mathcal{M}$ 的迁移关系 $\longrightarrow$ 中，其中 $a, b \in Act \cup \{\cdot\}$；同时将 (s_2, a) 添加到 $\mathcal{M}$ 的状态集合 S' 中；

- 对于 S' 中的状态 (s, a)，如果 s 在 $\mathcal{L}$ 中没有后继，那么将 $(s, a) \longrightarrow (s_d, \cdot)$ 添加到 $\mathcal{M}$ 的迁移关系 $\longrightarrow$。

- 最后，令 $P = \{(s_d, \cdot)\}$ 而且 $Q = \{(s, a) \mid s \in S \wedge a = \tau\}$。

经过从 LTS 到 Kripke 模型的转换之后，我们就可以在 SCTLProV 中验证死锁与活锁的存在了。

死锁 在 LTS 中，死锁状态指的是没有后继的状态。当验证一个 LTS $\mathcal{L}$ 中是否存在可达的死锁状态时，我们首先将 $\mathcal{L}$ 经过以上的转换方法转换到一个 Kripke 模型 $\mathcal{M}$。经过观察得知，$\mathcal{L}$ 中存在一个可达的死锁状态当且仅当 $\mathcal{M}$ 中状态 $(s_d, \cdot)$ 时可达的。

然后，我们可以通过证明如下的公式来验证 $\mathcal{M}$ 中 $(s_d, \cdot)$ 是否可达：

$$EF_x(P(x))((s_0, \cdot))$$

这个公式是可证的当且仅当 $\mathcal{M}$ 中存在一条形式为 $(s_0, \cdot) \longrightarrow^* (s, a) \longrightarrow (s_d, \cdot)$ 的路径，其中 a 是一个动作符号，而且 s 在 $\mathcal{L}$ 中没有后继。因此，这个公式可以被用来验证 $\mathcal{L}$ 中有没有可达的死锁状态。

活锁 在 LTS 中，活锁指的是所有动作均为 τ 的无穷的环状路径。在 LTS 中检测活锁相比检测死锁更复杂，原因是当观察一个迁移的时候，状态和动作都要考察到。与在检测死锁的方法一样，当验证一个 LTS $\mathcal{L}$ 中是否存在可达的活锁时，我们首先将 $\mathcal{L}$ 经过以上的转换方法转换到一个 Kripke 模型 $\mathcal{M}$。通过以下分析可知，$\mathcal{L}$ 中存在一个可达的活锁当且仅当 $\mathcal{M}$ 中存在一个环状路径，而且该路径上的所有状态均满足 Q。

- ($\Rightarrow$) 如果 $\mathcal{L}$ 中存在一个可达的环状路径，那么这个环状路径具有 $s_p \xrightarrow{\tau} s_{p+1} \xrightarrow{\tau} \cdots \xrightarrow{\tau} s_n \xrightarrow{\tau} s_p$ 形式，其中 $s_p = s_0$，或者存在一个从 s_0 到 s_p 的路径 $s_0 \xrightarrow{a_0} \cdots \xrightarrow{a_{p-1}} s_p$。那么根据由 $\mathcal{L}$ 到 $\mathcal{M}$ 的转换方法可知，$\mathcal{M}$ 中一定存在一个环状路径 $(s_p, a) \longrightarrow (s_{p+1}, \tau) \longrightarrow \cdots \longrightarrow (s_n, \tau) \longrightarrow (s_p, \tau) \longrightarrow (s_{p+1}, \tau)$，其中 $a = a_{p-1}$，而且 $(s_0, \cdot) \longrightarrow^* (s_p, a)$。

- ($\Leftarrow$) 如果 $\mathcal{M}$ 中存在一个具有 $(s_p,\tau) \longrightarrow (s_{p+1},\tau) \longrightarrow \cdots \longrightarrow (s_n,\tau) \longrightarrow (s_p,\tau)$ 形式的环状路径，而且其中对于此路径中的某个状态 (s_m,τ)，有 $(s_0,\cdot) \longrightarrow^* (s_m,\tau)$，那么在 $\mathcal{L}$ 中存在一个环状路径 $s_p \xrightarrow{\tau} \cdots \xrightarrow{\tau} s_m \xrightarrow{\tau} \cdots \xrightarrow{\tau} s_p$，其中此路径是从 s_0 状态可达的。

然后，我们可以通过在 $\mathcal{M}$ 中证明如下的公式来验证 $\mathcal{L}$ 中是否有可达的活锁：

$$EF_x(EG_y(Q(y))(x))((s_0,\cdot))$$

这个公式是可证的当且仅当 $\mathcal{L}$ 中存在一个可达的活锁。

测试集四的实验结果 我们用 SCTLProV 与 CADP 分别验证了测试集四中所有测试用例。如表4.8所示，在 40 个测试用例中，SCTLProV 和 CADP 均能成功验证所有的例子。在验证死锁性质时，SCTLProV 在 33 (82.5%) 个测试用例中用时比 CADP 短；在 7 (17.5%) 个测试用例中占用内存比 CADP 少。在验证活锁性质时，SCTLProV 在 22 (55.0%) 个测试用例中用时比 CADP 短；在 6 (15.0%) 个测试用例中占用内存比 CADP 少。测试集四的详细实验数据见附录C中表C.3和表C.4。

性质	t(SCTLProV) < t(CADP)	m(SCTLProV) < m(CADP)
死锁	33 (82.5%)	7 (17.5%)
活锁	22 (55.0%)	6 (15.0%)

表 4.8 测试集四中 SCTLProV 相比 CADP 用时短以及占用内存少的测试用例的个数

4.4.4 关于实验结果的讨论

由测试集一、二、三的实验结果可知，NuSMV、NuXMV、Verds、iProver Modulo、SCTLProV 的实验数据主要受两方面因素影响：状态变量的个数和要验证的公式的类型。其中，NuSMV、NuXMV 的实验数据主要受状态变量个数的影响，而 Verds、iProver Modulo、SCTLProV 的实验数据主要受公式类型的影响。当状态变量的个数较小的时候（比如测试集一），NuSMV 和 NuXMV 的表现往往优于 Verds，iProver Modulo 以及 SCTLProV；然而，在状态变量数较大的测试集中（比如测试集二、三），Verds、iProver Modulo、SCTLProV 的表现更好。如果在验证公式的过程中需要访问到几乎整个状态空间（比如一些 *AG* 性质），那么 NuSMV 和 NuXMV 的表现优于 Verds，iProver Modulo 以及 SCTLProV；然而，

在验证其他公式的时候，Verds，iProver Modulo、SCTLProV 的表现更好，这是因为在验证此类性质的时候 Verds，iProver Modulo 以及 SCTLProV 不必访问整个状态空间。因此，Verds，iProver Modulo、SCTLProV 相比 NuSMV、NuXMV 在状态变量个数上扩展性更好，其中 SCTLProV 在状态变量个数上比 Verds 和 iProver Modulo 扩展性更好，而且在几乎所有能验证的例子上比 Verds 和 iProver Modulo 占用资源更少。

由测试集四的实验结果可知，SCTLProV 和 CADP 的实验数据也受两方面因素影响：状态空间的大小以及是否满足要验证的性质。当状态空间较小时，SCTLProV 和 CADP 均占用资源较少；而且当要验证的性质（死锁和活锁）满足的时候，SCTLProV 和 CADP 的占用资源也较少。同时，在超过一半的例子中，SCTLProV 用时相比 CADP 更少。另外，值得注意的是，在验证测试集四种的测试用例时，SCTLProV 需要将 LTS 转换成 Kripke 模型，而在转换的过程中往往状态空间也会增大，因此在某些测试用例中 SCTLProV 的空间占用比 CADP 大。如果 SCTLProV 能直接在 LTS 上进行验证，那么则可避免增大状态空间，这是本文的下一步工作。

4.5 本章总结

作为上一章内容的工具实现，本章介绍的是定理证明工具 SCTLProV 及其应用。SCTLProV 的功能包括形式化建模和验证，其中形式化模型和要验证的性质是以输入文件的形式输入到 SCTLProV 中。SCTLProV 能处理所有的 CTL 模型检测问题，且的验证过程是完全自动化的，这个特点与模型检测工具相同；同时，当 SCTLProV 的验证过程结束时可输出证明树或者反例（性质的非的证明树），这个特点与定理证明工具一致。因此，SCTLProV 即可看作是模型检测工具，也可看作是定理证明工具。通过案例分析，并在多个测试集上与多个业界领先的模型检测工具的实验数据的对比可知，无论从输入语言的表达性，还是验证效率上来说，SCTLProV 已经具备了处理现实的、大型的计算机系统的验证问题的能力。

第 5 章　定理证明的可视化

根据上一章的内容可知，SCTLProV 已经具备了对真实世界的计算机系统进行建模和验证其 CTL_P 性质的能力，本章的工作在上一章工作的基础上，主要是为了解决如何更好的理解定理证明工具 SCTLProV 的证明结果这一问题。

在数理逻辑领域，一个逻辑公式的形式化证明通常被表示成一个公式序列，其中这个公式序列中的每个公式既可以是公理，也可以是之前公式的逻辑推导结论。一种更加自然的表示公式的形式化证明的方式是将证明表示成一棵树，其中这颗树的每一个节点都用一个公式标记，而该节点的子节点则标记为相应公式的逻辑前提。如今，随着计算机定理证明工具的应用，逻辑公式的形式化证明树可由计算机以自动或者半自动（需要人工干预）的方式生成。然而，目前定理证明工具的输出都是文本格式的，这就使得当证明树的结构较复杂或者节点较多的时候不容易使人理解。同样的问题也出现在模型检测领域，当 Kripke 模型的结构较复杂的时候，文本格式的输出通常无法直观地展示由模型检测工具生成的反例。另外，在上一章中我们提到，SCTLProV 在验证 Kripke 模型的性质时，相比于传统的模型检测工具，能生成更丰富的验证结果。为了能将 SCTLProV 的证明输出结果以及证明过程得以清晰并完整地展现出来，在本章我们介绍 VMDV[1]（Visualization for Modeling, Demonstration and Verification）。

VMDV 是一个将证明树及其他数据结构（比如 Kripke 模型）在 3D 空间内进行动态可视化布局和显示的工具。相比于其他的证明可视化工具，VMDV 具有四个特点：

1. 目前大多数的证明可视化工具[50–53] 都使用 2D 图形来完成证明树的绘制，并且可以用不同的颜色来区分证明树中不同的分支或节点。不同于这些工具，VMDV 是在 3D 空间中实现证明树的绘制，除了具备 2D 图形相比文本格式更直观的优点之外，3D 空间相比 2D 空间可以容纳更多的节点和更复杂的图形结构，从而在展示大型证明树时更有优势。

2. 另一方面，除了 VMDV 之外，目前也有其他可在 3D 空间内实现证明树绘制的工具[54,55]。然而，目前的 3D 证明可视化工具的布局算法是比较原始的，当在这些工具中展示较多的节点时，所有的节点往往按照一个或若干

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

个既定的方向做延伸状排列，这种布局方式使得 3D 图形的展示既不美观，又降低了空间利用率。不同于这些工具，VMDV 利用动态布局算法，将所有的节点看作是相互之间具有斥力的物体，而连接节点之间的边则可看作是具有引力的弹簧，通过每个节点所受的引力和斥力来随时更新节点在 3D 空间内的位置，并在整个布局过程中随时更新节点间的引力和斥力，直到所有节点的受力达到平衡状态。整个证明树就可看作一个自适应的物理系统，由此得出的节点布局更自然美观，也能充分填充 3D 空间。

3. 在传统的具有人工干预的定理证明工具（比如 Coq[2]）中，证明树的节点通常随时被加入或者删除。VMDV 可实现证明树结构的动态更新的可视化。

4. 不同于传统的证明可视化工具通常只针对于一种定理证明工具的输出的可视化，我们在 VMDV 中定义了可与不同的定理证明工具的接口[3]。实现这个接口的定理证明工具或者其插件都可以利用 VMDV 的功能来实现证明的可视化。

VMDV 利用 OpenGL 的绘图接口来编写显示引擎，并用 Java 编程语言来实现布局算法与其他工具的数据通信。接下来，我们分别介绍 VMDV 的相关技术细节及其应用。

5.1 背景知识

这一小节讲的是有关可视化的相关背景知识。

5.1.1 OpenGL

OpenGL，全称 Open Graphics Library，是一个跨编程语言并且跨平台的编程接口的标准。在计算机系统中，通常由显卡提供 OpenGL 的实现。一个典型的基于 OpenGL 的 3D 图形的显示过程为：首先，用户程序通过调用 OpenGL 的绘图 API（Application Programming Interface）来定义一组要显示的图形的数据和命令，并将这些数据和命令存储在计算机的主内存（RAM）中；然后，计算机的中央处理器（CPU）会通过 CPU 时钟将这些数据和命令发送到显卡的显存（VRAM）中，并在图形处理器（GPU）的控制下完成图形的渲染；最后，图形渲染的结果会被存入帧缓冲区中，而帧缓冲区中的帧则最终被发送到计算机的显示器上，并显示出结果。

[2] https://coq.inria.fr/

[3] https://github.com/terminatorlxj/VMDV/blob/master/protocol.md

5.1.2 信息可视化

信息可视化是一个研究如何利用计算机图形学技术将数据中的抽象信息可视化的研究领域。抽象信息的可视化表达可以用来帮助人们揭示数据的隐匿模式，用来帮助更好地理解和发现数据中的规律。得益于计算机图形学理论的日益成熟以及计算机硬件的飞速发展，信息可视化系统能可视化越来越大的数据集以及越来越复杂的数据结构，并在许多行业中发挥着举足轻重的作用。比如，在知识管理领域，信息可视化系统可以用来研究不同学科领域知识之间的关系和演化[56]；在城市交通系统的研究中，信息可视化系统可以将城市的公共交通工具的卫星定位数据进行可视化，以帮助人们研究和改善公共交通的效率[57]。

5.2 VMDV 的实现

这一小节讲的是 VMDV 的架构，接口以及 3D 图形的自动布局算法。

5.2.1 VMDV 的架构

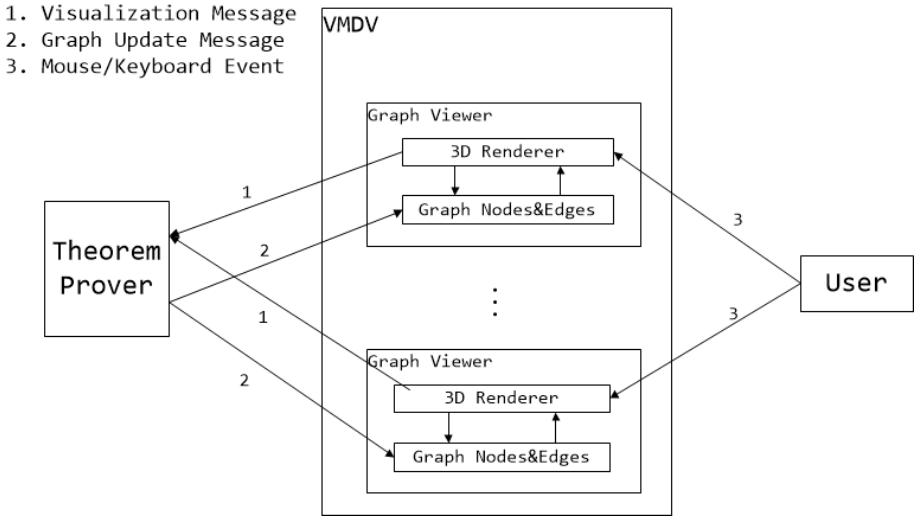


图 5.1 VMDV 的架构

如图5.1所示，VMDV 可同时在多个程序窗口中显示 3D 图形，在 VMDV 中，每个程序窗口都分别包含一个 3D 渲染器和一个由多个点和边组成的图。3D 渲染器的作用是读入图中的点和边的信息，然后根据布局算法确定点和边的位置，最后在 3D 空间中将图渲染出来。其中，图中的点和边可以通过接收定理证明器传递的消息来动态的添加、删除以及更改，同时 3D 渲染器对图的更改进行动态的显示，然后向定理证明器发送反馈消息。在 3D 渲染器工作的同时，VMDV 同样接收用户传递的交互消息（键盘或鼠标消息），并及时地在 3D 渲染器中做出

反应。

可以看出，VMDV 同时可以接收两种消息：定理证明器传递过来的消息以及用户给定的消息。其中，定理证明器的消息是以 TCP 数据包的形式传输的。因此，定理证明器与 VMDV 在网络上通过给定的通讯接口进行交互，而不一定在同一机器上甚至同一进程中。这种设计方式大大减少了 VMDV 对特定的定理证明器的依赖。用户通过传递给 VMDV 消息，可实现对 3D 图形的操作：旋转、放大、缩小、高亮、搜索以及触发自定义操作。通过这两种消息，VMDV 可以实现与定理证明器以及用户的交互。

下面我们分别介绍 VMDV 与定理证明器和用户的交互。

5.2.2 VMDV 与定理证明器的交互

上面我们说到，VMDV 与定理证明器的交互是通过 TCP 数据包形式的消息传递来实现的。其中，每个 TCP 数据包都包含一个 JSON 格式的数据结构。VMDV 与定理证明工具的交互协议的详细描述见附录B。

5.2.3 VMDV 与用户的交互

用户可通过鼠标和键盘实现与 VMDV 的交互。用户通过鼠标可以实现 3D 图形的放缩、旋转、选中、高亮以及触发自定义操作等；通过键盘可以实现符合特定条件的节点的搜索。下面我们依次介绍这些交互。

图形放大与缩小 在对于图形的所有操作中，放大和缩小是最简单直观的操作，此类操作通常由鼠标控制。在 VMDV 中，我们用鼠标滚轮来控制图形的放大和缩小，通常滚轮向上滚动代表放大图形，滚轮向下滚动代表缩小图形。

图形旋转 在可视化中，3D 图形相比于 2D 图形最大的优势是可以旋转，从而变换角度详细观察图形的细节。带有旋转功能的 3D 图形相比于 2D 图形的另一个优势是前者通常可以显示更多的内容。虽然后者可通过延伸图形来利用更多平面空间，然而当要显示的内容过多时整个图形往往变得过于稀疏，从而难以观察图形的全局；当不通过延伸图形来显示更多的内容时，2D 图形往往会有重叠显示的内容，从而使整个图形变得更加难以理解。因此，具有旋转功能的图形对于理解整个图形的内容是非常重要的。在 VMDV 中，我们用鼠标左键来控制旋转，按住鼠标左键在图形上滑动即可实现图形的旋转（如图5.2）。

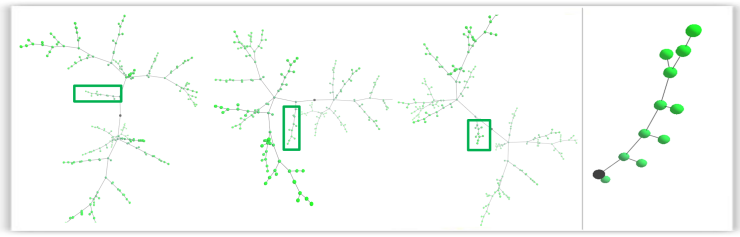


图 5.2 从不同角度观察图形

图形选中与高亮 通常在一个 3D 图形中，在需要呈现局部信息（比如公式的结构、Kripke 模型中的状态等）时，我们需要用鼠标来选中，从而高亮某个节点，这时就可以观察被选中的节点的详细信息。VMDV 中的选中与高亮是可在不同的图形中同步进行的，比如当高亮一棵 SCTLProV 的证明树上的节点时，这个节点对应的公式的相关状态也会在显示 Kripke 模型的图形中高亮出来。在 VMDV 中，选中节点的操作既可以通过鼠标左键在节点上点击，也可以通过键盘搜索满足相应条件的节点（如图5.3）。

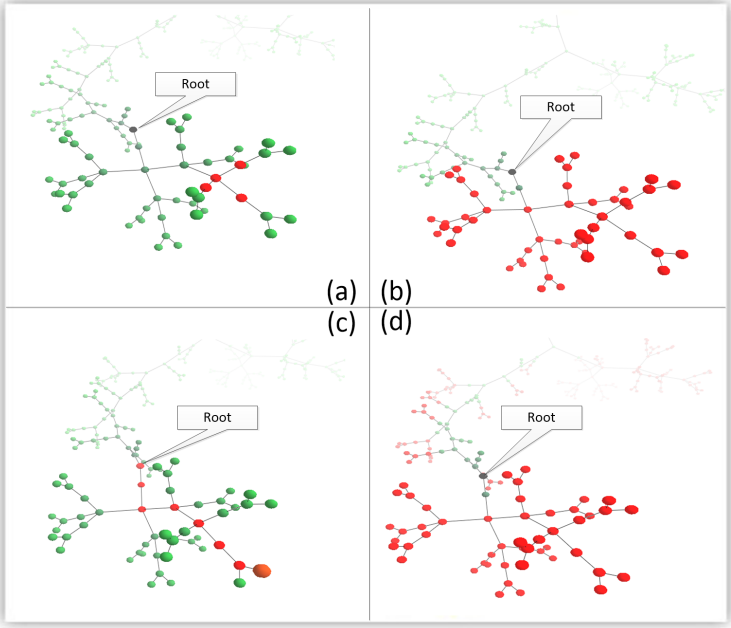


图 5.3 高亮图形的不同部分：(a) 高亮单个节点及其后继；(b) 高亮一棵子树；(c) 高亮一个节点的所有祖先；(d) 高亮所有同一类型的公式的子证明

触发自定义操作 除了以上介绍的用户操作之外，VMDV 还提供了一种功能来实现对图形的自定义操作。由于 VMDV 可与不同的定理证明工具适配，因此，在一些常用操作之外，VMDV 中还应该定义一些与特定的定理证明工具相关的操作，比如在 SCTLProV 中，当高亮证明树上的节点时，状态图上的节点也随

之自动高亮；在 Coq 中，当选中证明树上的某个节点时，用户可自行选择是否继续证明该节点上的公式。在 VMDV 中实现自定义操作的方式分为两步：第一步，详细定义自定义操作的方式，即对图形的操作；第二步，定义触发自定义操作的方式。在第一步中，我们利用 Java 语言的面向对象的特性，首先定义一个 `AssistAffect` 抽象类，然后将所有的对图形的操作均包装成 `AssistAffect` 的子类，这样在程序的编译过程中编译器不需要知道图形的操作的细节，而是在运行时按需调用相应的 `AssistAffect` 子类。由此，我们实现了主程序代码与自定义的图形操作代码的分离。在第二步中，我们同样将触发自定义操作的代码与主程序代码分离开来，具体做法是将触发自定义操作的代码定义为实现 Java 接口 `PopupItem` 的类，由类的继承关系可知，所有实现接口 `PopupItem` 的类均可视为弹出菜单项，且可在窗口中通过鼠标右键触发。因此，当点击由鼠标右键触发的弹出菜单项时，相应的自定义操作即可执行（如图5.4）。

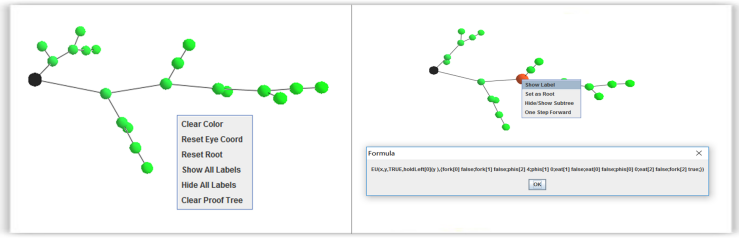


图 5.4 VMDV 中的自定义操作

节点搜索 在 VMDV 中，虽然节点可通过鼠标逐个高亮显示，但这种方式往往效率过低。为了解决这个问题，我们在 VMDV 中增加了节点搜索的功能以实现节点的批量高亮显示。VMDV 中的节点搜索是通过将每个节点的标签（即所代表的公式的字符串表示）与输入的正则表达式[4]进行匹配，然后将匹配成功的节点高亮显示出来。（如图5.5）。

5.2.4 VMDV 中 3D 图形的动态布局算法

在 3D 图形的可视化系统中，布局算法的作用是计算图形的每个部分在 3D 空间坐标系中的坐标使得图形的每个部分尽量减少重叠。由于 VMDV 中图形的所有节点均可动态添加或删除，因此 VMDV 的布局算法必须是连续的，即算法是连续运行、随时更新的。而且，同样由于 VMDV 的图形具有动态更新的特性，图形中的所有部分必须用同一种方式显示，比如所有的节点统一显示为实心球，

[4] https://en.wikipedia.org/wiki/Regular_expression

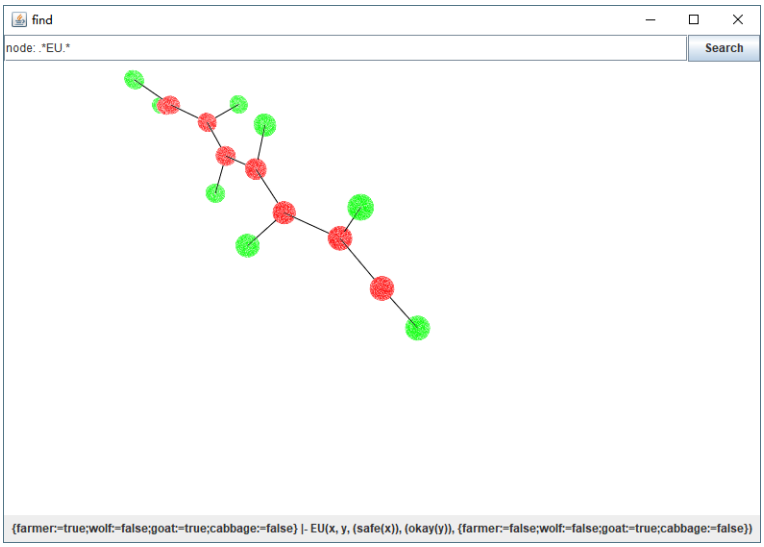


图 5.5 VMDV 中搜索节点

所有的边统一显示为实线。同样，为了保持图形显示的统一性，图形的布局算法也必须是统一的，即图形的每个部分用同一种方式布局。VMDV 中的布局算法基于 ForceAtlas2 算法[58]，该算法同时满足了 VMDV 对布局算法连续性和统一性的要求。类似的布局算法还有 OpenOrd 算法[59] 以及 Yifan Hu 算法[60]。不同于 ForceAtlas2 算法，OpenOrd 算法既不是连续的也不是统一的，而 Yifan Hu 算法是统一的，但不是连续的。与 ForceAtlas2 算法一样，VMDV 中的布局算法的基本原理为：首先，图形中每个部分的坐标是根据其受力的大小而变化的，其中每对节点之间具有斥力，力的大小与节点间距离的大小成反比，同时每对以边相连的节点之间具有相互的引力，引力大小与边的长度成正比；然后，当根据每个节点的受力情况更新完其坐标后，再根据新的坐标计算其在新位置上的受力，直至达到受力平衡状态。

5.3 VMDV 的应用

在本小节中，我们首先考虑 VMDV 在定理证明工具 SCTLProV 的一个应用：可视化证明树和 Kripke 模型，然后，我们介绍如何将 VMDV 应用在定理证明工具 Coq 中。

VMDV 在定理证明工具 SCTLProV 的一个应用

例子 5.1 (过河问题). 一位农夫带着一头狼，一只羊和一筐白菜准备过河，河边有一条小船，农夫划船每次只能载狼、羊、白菜三者中的一个过河。农夫不在旁边时，狼会吃羊，羊会吃白菜。问农夫该如何过河。

过河问题可以被描述为一个模型检测问题：假设农夫准备带着狼、养、白菜从河岸 A 到达河岸 B，Kripke 模型中的每个状态由四个布尔类型的状态变量$farmer$、$wolf$、$goat$ 以及 $cabbage$ 组成，分别代表农夫、狼、羊以及白菜在河岸 A 或者河岸 B。，那么对于一个状态 s，如果 $farmer$ 的赋值为 $true$，那么代表在状态 s 中农夫在河岸 B，其他状态变量的赋值以此类推。因此，Kripke 模型中的初始状态为：

$$s_0 = \{farmer := false, wolf := false, goat := false, cabbage := false\}$$

Kripke 模型中的迁移则表示为农夫在过河过程中所有的状态变量的赋值的变化。该模型检测问题则可表述为是否存在一个由 s_0 可达的状态 s 使得

$$s = \{farmer := true, wolf := true, goat := true, cabbage := true\}$$

而且在从 s_0 到 s 的路径上的所有的状态 s' 上，狼不能单独和羊在一起，且羊不能单独和白菜在一起。该模型检测问题可用 CTL_P 公式表示为：

$$EU_{x,y}(safe(x), complete(y))(s_0)$$

其中 $safe(x)$ 表示在状态 x 中狼不能单独和羊在一起，且羊不能单独和白菜在一起，而 $complete(y)$ 表示在状态 y 中农夫、狼、羊以及白菜均安全到达河岸 B。

利用 VMDV，我们可以将 SCTLProV 对该问题的验证过程进行可视化（见图5.6），其中上述 CTL_P 公式的证明树在 VMDV 中逐步显示出来，而且随着证明树的逐步显示（图5.6上半部分），该问题的 Kripke 模型也逐步构造出来（图5.6下半部分）。当证明构造完成之后，且将所有的对应于带有 EU 模态词的公式的节点进行高亮显示时，在状态图中则对应着一个路径的高亮显示，该路径即为从 s_0 到 s 的路径，即农夫、狼、羊以及白菜均安全从河岸 A 到达河岸 B 的过程（见图5.7）。

需要注意的是，在 SCTLProV 中，在对不同公式的验证过程中，需要访问的状态个数也往往不同，比如在上述过河问题中，对 EU 公式的验证意味着需要在 Kripke 模型中寻找一条满足特定条件的路径的过程，因此当对 EU 公式的验证结束并成功之后，相应的状态图中只包含该路径上的节点及其后继。然而，当验证非 EU 的公式，比如 AG 公式时，如果要验证的公式成立，那么 SCTLProV 往往需要访问 Kripke 模型的所有状态，这时状态图也会相对更大（见图5.8），此时，由于 AG 公式的证明树及其状态图往往很大，我们可以利用 VMDV 的局部选中功能，每次观察证明树的时候只观察选中的部分，相应的状态图中也只高亮显示相关的节点（见图5.9）。

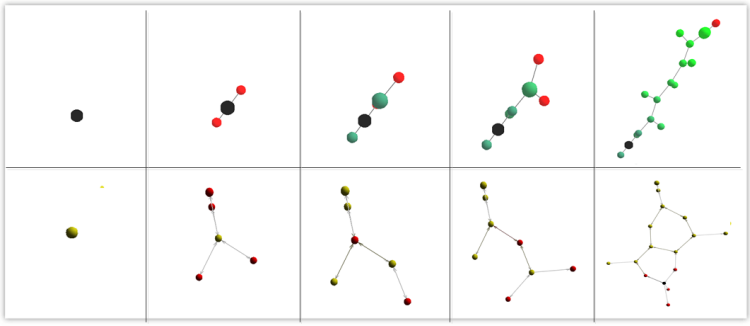


图 5.6 过河问题的验证过程

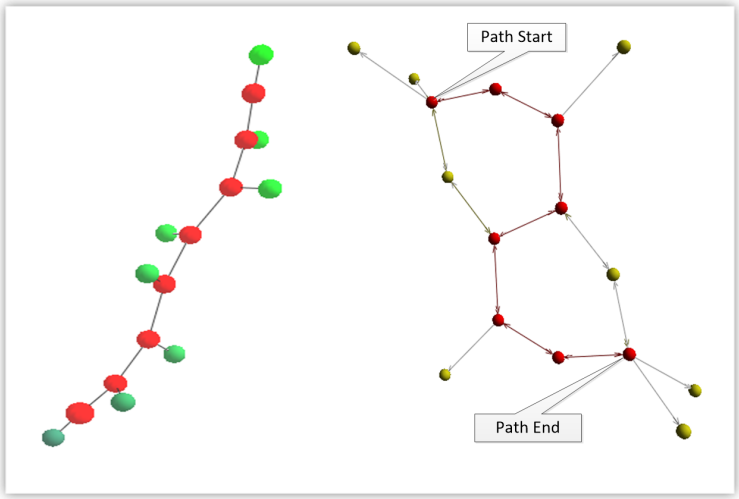


图 5.7 过河问题中证明树和状态图的高亮显示

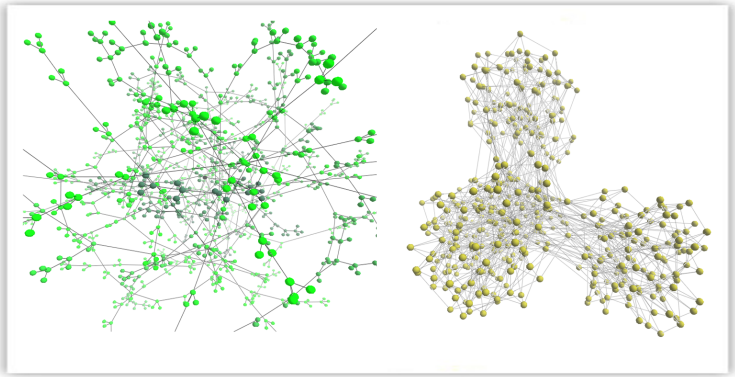


图 5.8 一个 *AG* 公式的证明树及其状态图

VMDV 与定理证明工具 Coq 的结合 与 SCTLProV 不同，Coq 中的证明树不是自动生成的，而是由编程人员通过输入证明脚本（Proof Script）来人工干预证明树的产生过程，其中每个证明脚本又包含若干条证明语句。在 Coq 中，每个需要证明的公式被称为一个目标，而该公式的所有前提则被称为该目标的子目标。一个目标的子目标是通过一条证明语句而生成的。因此，Coq 的证明树可表示为：

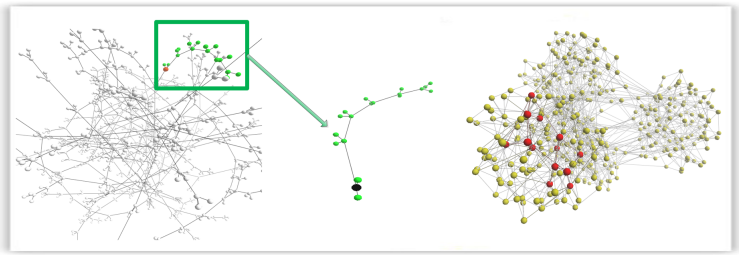


图 5.9 证明树的部分选中以及状态图的部分高亮

将每个公式（目标）都看成证明树中的节点，而且将该公式的前提（子目标）看成相应的节点的后继，如果该公式没有前提，则其对应的节点为叶节点。为了将 Coq 中的证明树在 VMDV 中可视化，我们开发了一个用于协调 VMDV 与 Coq 之间通信的中间件 Coqv[5]：Coqv 与 Coq 的通信遵循 Coq 自定义的 XML 通讯协议[6]；Coqv 与 VMDV 的通信遵循 VMDV 的 JSON 通讯协议。通过 Coqv，VMDV 可实现对 Coq 中生成的证明树的可视化。例如，公式

$$\forall ABC : Prop, (A \to (B \to C) \to ((A \to B) \to (A \to C)))$$

的证明脚本如下。

```
Proof.
    intros A B C.
    intros H1 H2 H3.
    apply H1. assumption.
    apply H2. assumption.
Qed.
```

该证明脚本所生成的证明树在 VMDV 中可视化如图5.10所示。图中证明树的生成分别由 6 步完成，分别对应于证明脚本的 6 个命令。目前，Coqv 的开发阶段已完成，目前正在根据 Coq 的不同应用场景进行测试，并逐步完善功能，这也是本论文的其中一项未来工作。

[5] https://github.com/terminatorlxj/coqv

[6] https://github.com/siegebell/vscoq/wiki/XML-protocol

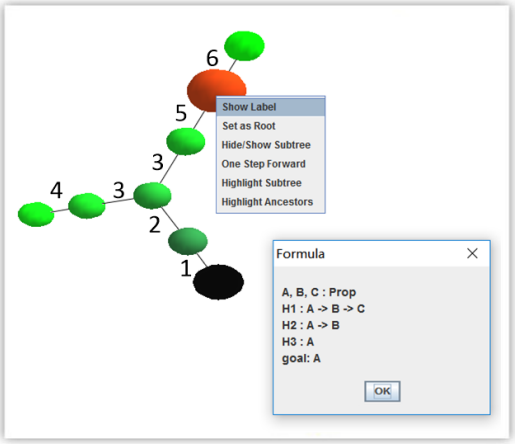


图 5.10 Coq 中的一棵证明树的可视化

5.4 本章总结

本章的内容介绍的是证明可视化工具 VMDV。起初，VMDV 的提出是为了解决上一章提出的定理证明工具 SCTLProV 的可视化问题：通过以 3D 图形的方式展示 SCTLProV 输出的证明树来达到理解输出结果并优化建模方式的目的。不过，为了一般性起见，我们将 VMDV 设计成了可以匹配不同的定理证明工具的可视化系统：定理证明工具（或其插件）通过实现 VMDV 预先定义的通讯协议来达到与 VMDV 进行交互并可视化的目的。不同于现有的证明可视化工具，VMDV 的图形显示效果更好，而且其动态布局算法能更好的适应交互式定理证明的可视化。

第 6 章　Coq 的证明可视化

本章工作是作为上一章工作的扩展。在上一章中我们提到，VMDV 提供了一个一般化的接口，用于与不同的定理证明工具进行通讯，进而可以实现不同定理证明工具的证明可视化。本章介绍的是对于一个成熟的，并且在工业界以及学术界得到广泛使用的定理证明工具 Coq 的证明可视化。

6.1　Coq 概述

Coq 是一个计算机辅助定理证明工具。研究人员和工程人员利用 Coq 可以开发程序，并表达程序的规范，然后利用定理证明的方式来形式化验证程序是否满足规范。因此，Coq 适合开发需要绝对可信的程序，这样的程序在许多行业中是非常重要的，比如航空航天、交通以及银行等。Coq 通常以交互的方式来构造证明，并尽可能利用自动化的证明搜索工具的辅助。Coq 不仅可以作为一个形式化验证系统，还可以作为一个逻辑框架来为新的逻辑提供公理、定义规则，并基于此来开发证明。

作为一个计算机工具，Coq 与编程人员交互时需要用到两种语言：Gallina 语言和 Vernacular 语言。Gallina 是一个面向表达式的语言，利用 Gallina 语言的表达式可以来描述项、类型、证明以及程序；Vernacular 是一个面向命令的语言，利用 Vernacular 命令可以控制 Coq 的行为（比如搜索、类型检查以及计算等。），配置 Coq 的参数（比如设置当前工作目录、是否输出警告等），以及最重要的：声明和定义 Gallina 表达式。Coq 与编程人员交互的方式是解析编程人员输入的 Vernacular 命令，并即时反馈解析的结果。

根据柯里-霍华德同构，程序、规范以及证明都可以用归纳构造演算（Calculus of Inductive Construction）[18] 中的项来表示。在归纳构造演算中，所有的逻辑判断（logical judgement）都是类型判断（type judgement），而 Coq 的核心功能即利用类型检查算法来检验证明的正确性，即检查一段程序是否符合其规范。除此之外，Coq 还提供了一种被称为证明策略（tactic）的机制来构造证明。

Coq 提供了两种构造证明的模式：交互模式和编译模式。

- 交互模式：用户通过在操作系统的命令行中输入“coqtop”来开启交互模式。在这种模式中，用户可以通过逐步输入命令的方式来构造理论和证明。

- 编译模式：用户通过在操作系统的命令行中输入“coqc”来调用 Coq 的编译模块。编译模块可被看作为证明检查工具，并来确保所输入的文件中包含的 Vernacular 命令是正确的。

6.1.1 Coq 的用户接口

Coq 的用户接口的作用是辅助用户编写 Vernacular 命令，并实时显示 Coq 对命令的解析状态，反过来帮助用户完成余下的命令的编写。目前最常用的 Coq 用户接口为 Proof General[1]和 CoqIDE[2]。Proof General 是一个基于 Emacs[3]的定理证明工具的前端（front end），并提供了一个统一的接口来适配不同的定理证明工具。不同于 Proof General，CoqIDE 是一个独立的，专门针对 Coq 的图形用户界面系统。在这两个用户接口程序中都可实现 Coq 的交互和编译两种模式的证明。

6.1.2 Coq 对 Vernacular 命令的解析

```
Definition decidable (P : Prop) := P \/ ~ P.

Theorem dec_False : decidable False.
Proof.
unfold decidable, not.
auto.
Qed.
```

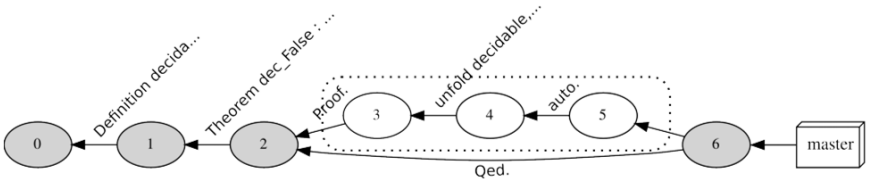


图 6.1 Vernacular 命令以及 Coq 对其的解析

无论是交互模式还是编译模式，Coq 都需要对一系列 Vernacular 命令进行逐个解析。在 Coq 内部有一个被称为系统状态（system state）的概念，Coq 将每对每条命令的解析看作为从当前的系统状态到下一个系统状态的迁移，当实现回退操作时，系统需要抛弃当前的状态，而重新从上一个状态开始解析命令。自从 8.5 版本开始，Coq 利用有向无环图（Directed Acyclic Graph，简称 DAG）来记录系统状态：DAG 中的点是系统状态，而点与点之间的有向边则标记为一条命令。Coq 中操作有向无环图的模块被称为状态处理机（State Transaction Machine，简称 STM），在 STM 中，每条命令的解析是自动的、原子的，即命令解析的中

[1] https://proofgeneral.github.io

[2] https://github.com/coq/coq/wiki/CoqIde

[3] https://www.gnu.org/software/emacs/

间状态是不可见的。例如，在图6.1中，STM 从系统状态 0 开始依次处理 6 条命令，每条命令对应着一条有向边，当处理完所有命令后，状态 6 变为当前的状态（master 指针所指向的状态）。其中，生成状态 1、2、6 的命令为全局命令（全局有效的定义或声明），而生成状态 3、4、5 的命令为局部命令（证明当前命题的策略）。

6.2 Coq 中证明树的构造与可视化

Coq 中的证明可以展现成证明树的形式：每个当前需要证明的命题被称为目标（goal），当应用一条规则，而且该命题匹配当前规则的结论，那么利用与命题和规则的结论同样的匹配方式去例化当前规则的假设，从而得到 0 个或多个命题，这些新生成的命题被成为子目标（subgoal），然后在新生成的命题上利用相同的证明方式去产生更多的子目标，直到无法生成更多的子目标为止。在这种形式的证明中，每个目标可看作证明树中的一个节点，而其子目标则可看作该节点的子节点。在 Coq 中对一个命题的证明过程则可看作一棵证明树的构造过程，在此过程中所应用的规则即证明脚本中的策略。

然而，无论在 Coq 中，还是在其用户接口 Proof General 和 CoqIDE 中，都不存在构造以及保存证明树的机制，因此这种状况给用户在证明过程中增添了一种无形的负担，即需要自行在脑海中想象当前证明树的完整形状。随着证明操作越来越复杂，比如需要回退操作以及在不同子目标之间相互跳转时等，用户所依赖的信息只有 Coq 给出的当前需要证明的命题的信息，而越来越难以建立起完整的证明树的概念。为了解决这个问题，我们在本章中研究 Coq 的证明可视化，因此我们设计并开发了一个工具 Coqv[4]来完整地记录 Coq 中证明树以及对证明树的操作。Coqv 通过与 VMDV 结合，就可实现 Coq 的证明树的可视化。

6.2.1 Coqv 的原理

如图6.2所示，Coqv 的作用是作为协调证明可视化工具 VMDV 与 Coqtop 通讯的中间件。Coqv 工作流程可总结为：

1. Coqv 接收用户输入的命令，然后将该命令发送到 Coqtop；
2. Coqtop 利用 STM 模块解析所接收到的命令，然后将反馈信息发送给 Coqv；
3. Coqv 解析所收到的来自 Coqtop 的反馈消息并在命令行中输出，除此之外，

[4]https://github.com/ProveVis/coqv

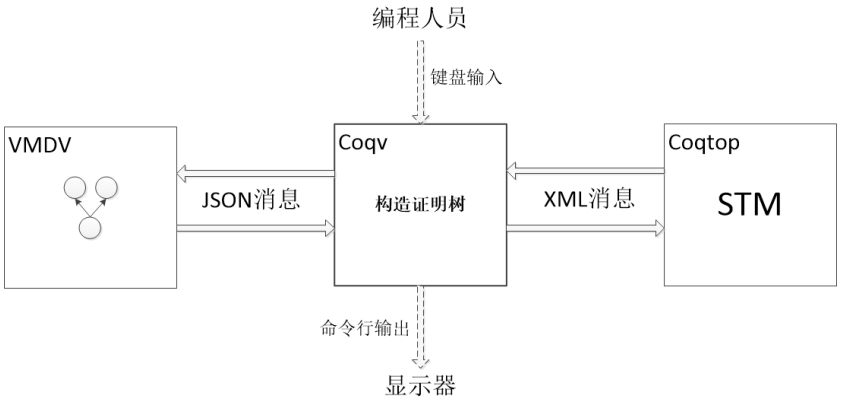


图 6.2 Coqv 作为 VMDV 和 Coqtop 进行通讯的中间件

Coqv 还根据解析的消息内容对证明树进行更新，同时将对于证明树的更新操作发送给 VMDV；

4. VMDV 接收到来自 Coqv 的消息，并根据消息内容在 3D 空间内更新证明树的显示效果，除此之外，用户通过 VMDV 对证明树的操作（比如删除某节点）还可反馈到 Coqv，然后 Coqv 将相应的命令（比如丢弃某个命题的证明）发送给 Coqtop。

在第 1 步和第 2 步中，Coqv 与 Coqtop 的是通过 Coqtop 内置的 XML 协议[5]来通讯的；在第 3 步和第 4 步中，Coqv 与 VMDV 是通过 VMDV 内置的 JSON 协议（见附录B）来通讯的。

需要注意的是，如果在图6.2中去掉 VMDV 相关的部分，则 Coqv 的功能与 Proof General 和 CoqIDE 一致，即都是作为 Coq 的用户接口。因此，相比于其他针对 Coq 的用户接口，Coqv 增加了构造证明树，以及将证明树的构造信息发送到 VMDV 以进行可视化的功能。

[5] https://github.com/siegebell/vscoq/wiki/XML-protocol

第 7 章　论文总结与对未来工作的展望

本论文的内容是从定理证明的角度来研究模型检测问题。具体方式为：首先，在逻辑层面对模型检测中常用的计算树逻辑 CTL 进行扩展，我们称扩展后的时序逻辑为 CTL_P。相比于 CTL，CTL_P 可表达多元谓词。多元谓词的存在使得扩展后的逻辑不仅可以表达状态的时序性质，同时可以表达状态之间的“空间”性质，即多个状态之间的关系。然后，我们提出了一种针对 CTL_P 逻辑的证明系统 SCTL。SCTL 以一个 Kripke 模型为参数，一个 CTL_P 公式在 $\mathrm{SCTL}(\mathcal{M})$ 是可证的当且仅当该公式在当前的 Kripke 模型 $\mathcal{M}$ 中是成立的。然后，我们对于 SCTL 证明系统设计了一种高效的证明搜索算法，并基于此算法开发了一个定理证明工具 SCTLProV。SCTLProV 既可以看作为定理证明工具，也可看作为模型检测工具。相比于传统的定理证明工具，SCTLProV 可利用一些模型检测中的常用策略来提高验证的效率；相比于传统的模型检测工具，SCTLProV 的输出结果更加丰富。最后，为了更好的理解 SCTLProV 的验证结果，我们开发了一个定理证明可视化工具 VMDV，同时，为了适配更多的定理证明工具，VMDV 提供了一个一般化的接口用来与不同的定理证明工具通讯，从而可视化不同的定理证明工具的输出结果。

基于本论文的工作，未来我们还可在以下几个方面对已有的工作进行扩展：第一，我们可将 CTL_P 逻辑以及 SCTL 证明系统扩展到基于符号迁移系统的验证中，而不仅仅是针对 Kripke 模型的验证。第二，我们可将 SCTLProV 扩展到多线程版本，并利用现代计算机具有多个核心的特性来进一步提高验证效率。第三，我们还可继续在 VMDV 方面的工作，实现针对更多更复杂的定理证明工具的证明结果以及证明过程的可视化。

附录 A　SCTLProV 的输入语言描述

A.1　词法标记与关键字

输入文件中包含的是有限个依次排列的字符，这些字符通过词法解析器的解析生成有限个词法标记。在词法标记中，

- 一个非负整数表示为有限个连续数字，而一个负整数则表示为在一个非负整数前添加符号“-”；

```
integer ::= [-]{0...9}+
```

- 一个非负浮点数表示为以符号“.”分隔的两个非负整数，其中当“.”后边的非负整数为 0 时可省略，而一个负浮点数则表示为在一个非负浮点数前添加符号“-”；

```
float ::= [-]{0...9}+.{0...9}*
```

- 一个标识符表示为以字母开头，并且有限个字母、数字、下划线以及符号“-”的组合，而且，在输入语言中区分以大写字母开头的标识符和以小写字母开头的标识符，不同标识符的详细用法可参考下面具体的语法描述。

```
iden     ::=  letter {letter|uletter|0...9|_|-}*
uiden    ::=  uletter {letter|uletter|0...9|_|-}*
letter   ::=  a...z
uletter  ::=  A...Z
```

- 空白符和注释被用来增强程序的可读性，且在输入文件的语法分析阶段被忽略掉。其中，空格、制表符、回车符、换行符均为空白符。注释分为单行注释与多行注释，其中单行注释以“//”开头且以换行符结尾，而多行注释分为两种：以“/*”开头且以“*/”结尾，或以“(*”开头且以“*)”结尾。

除了以上语法标记外，输入语言中还包括一系列关键字。关键字是一种特殊的词法标记，也叫做保留字，用于在输入文件中表示特殊的含义。输入语言中的关键字如下：

Model	Var	Define	Init	Transition	Atomic	Fairness	Spec
int	bool	list	array	true	false	TRUE	FALSE
not	AX	EX	AF	AF	EG	AR	EU
datatype	value	function	let	match	with	if	then
else	(	)	[	]	{	}	=
!=	<	<=	>	>=	+	+.	-
-.	*	*.	/	/.	!	|	||
&&	->	:	::	,	;	.	

在输入文件中，关键字不能用来定义值或者函数的名字。关键字的用法在接下来的语法描述中有具体描述。

A.2 值

在输入语言中，值通常被用来定义 Kripke 结构的状态。一个值通常具有以下几种形式：

- 单位值：单位值 `()` 通常可看作一个空元组，是最简单的值。
- 布尔值：布尔值有两种：`true` 和 `false`。
- 数值：数值分为两种：整数值与浮点数值。数值的取值范围与 OCaml 编译器所能表示的相应值的取值范围一致，即：整数值的取值范围为 -2^{30} 到 $2^{30}-1$（32 位）或 -2^{62} 到 $2^{62}-1$（64 位）；浮点数值的取值范围符合 IEEE 754[1]标准。
- 标量值：标量值也被称为枚举值，标量值通常由编程人员自定义且具有 `#iden` 形式，即“`#`”符号紧接着一个以小写字母为开头的标识符。
- 数组和列表：与 OCaml 语言类似，数组在这里具有 `[|v1,...,vn|]` 形式，其中 `v1,...,vn` 为 $n \geq 0$ 个值；列表在这里具有 `[v1,...,vn]` 形式，其中 `v1,...,vn` 为 $n \geq 0$ 个值。
- 元组：元组具有 `v1,...,vn`，其中 `v1,...,vn` 为 $n \geq 1$ 个值。
- 记录：记录可看作带标签的元组，记录具有 `{l1 = v1 ; ... ; ln = vn ;}` 形式，其中值 `vi` 的标签是 `li`。

[1] https://en.wikipedia.org/wiki/IEEE_754

- 变体：变体由变体构造子及若干个值构造而成。变体分两种：只包含构造子的变体（构造子的元数为 0）以及构造子和元组构造而成的变体（构造子的元数为元组中值的个数）。

值的语法描述如下：

```
value ::=
     ()
   | true | false
   | integer | float
   | #iden
   | [| value ; ... ; value |] | [ value ; ... ; value ]
   | ( value,...,value )
   | { iden = value ; ... ; iden = value ; }
   | uiden | uiden ( value , ... , value )
```

A.3 表达式与模式

在 SCTLProV 的输入语言中，我们通过表达式的形式来定义复杂的值，即表达式通过有限步规约得到值。为了使表达式更容易理解，我们令表达式的规约顺序与 OCaml 语言中表达式的规约顺序一致。另外，在模式匹配表达式中，模式的作用是为了匹配不同形式的表达式，其中所定义的模式的匹配形式也与 OCaml 中一致。

```
expr ::=
      value                          (*value*)
    | iden                           (*variable or name of a value*)
    | uiden [(expr,...,expr)]        (*variant expression*)
    | expr . iden                    (*select one field of a record*)
    | expr [ expr ]                  (*select one field of a array*)
    | ! expr                         (*logical negation*)
    | expr && expr                   (*logical and*)
    | expr || expr                   (*logical or*)
    | - expr                         (*integer negation*)
    | expr + expr                    (*integer addition*)
```

```
    | expr - expr                 (*integer subtraction*)
    | expr * expr                 (*integer multiplication*)
    | -. expr                     (*float negation*)
    | expr +. expr                (*float addition*)
    | expr -. expr                (*float subtraction*)
    | expr *. expr                (*float multiplication*)
    | expr = expr                 (*expression equivalence*)
    | expr != expr                (*expression non-equivalence*)
    | expr < expr                 (*less than*)
    | expr <= expr                (*less than or equal*)
    | expr > expr                 (*larger than*)
    | expr >= expr                (*larger than or equal*)
    | ( expr )                    (*expression grouping*)
    | let pattern = expr          (*declare local variables*)
    | if expr then expr           (*if-then expression*)
    | if expr then expr else expr (*if-then-else expression *)
    | expr ; expr                 (*sequence of expressions*)
    | expr <- expr                (*assignment*)
    | match_expr                  (*pattern matching*)
    | expr with {iden = expr ; ... ; iden = expr [;]}
                           (*a record with changed bindings*)
    | iden (expr , ..., expr)     (*a function call*)

match_expr ::= match expr with {| pattern -> expr}+

pattern ::=
      iden
    | constant
    | pattern :: pattern                    (*list*)
    | ( pattern ,..., pattern )             (*tuple*)
    | _                                     (*match any case*)
```

A.4 类型

与传统的编程语言类似，SCTLProV 的输入语言同样包含类型的定义。类型分为基本类型与复合类型。

基本类型

- 单元类型：单元类型是最简单的类型，记作 `unit`。这种类型的值只有一个，即 `()`。
- 布尔类型：布尔类型记作 `bool`，该类型的值有两个：`true` 和 `false`。
- 整数类型：整数类型可记作 `int` 或者 `(min .. max)`，当记作 `int` 时，该整数类型的取值范围与 OCaml 的整数取值范围一致；当记作 `(min .. max)` 时，在不超过 OCaml 的整数类型取值范围的基础上，该整数类型的取值范围为 $[min, max]$，其中 min 和 max 均为整数。
- 浮点数类型：浮点数类型记作 `float`，与 OCaml 的浮点数类型的取值范围一致。
- 标量类型：标量类型用来表示对有穷个符号的枚举，记作 `#iden`$_1$`, ... ,` `#iden`$_n$，其中 `iden`$_1$`, ... ,` `iden`$_n$ 为任意 n 个符号，即以小写字母开头的标识符。

复合类型

- 数组类型：数组类型记作 `array t`，其中 `t` 是一个类型。
- 列表类型：列表类型记作 `list t`，其中 `t` 是一个类型。
- 元组类型：远足类型记作 `(t`$_1$`, ... , t`$_n$`)`，其中 `t`$_1$`, ... , t`$_n$ 为 $n > 1$ 个类型。
- 记录类型：记录类型可看作带标签的元组类型，记作 `iden`$_1$`: t`$_1$`; ... ; iden`$_n$`: t`$_n$`;`,其中 `t`$_1$`, ... , t`$_n$ 为 $n \geq 1$ 个类型,而 `iden`$_1$`, ... , iden`$_n$ 为与其对应的 $n \geq 1$ 个标签。

- 变体类型：变体类型由若干个变体构造子组成，记作 $\texttt{constr}_1$ | ... | $\texttt{constr}_n$，其中 $\texttt{constr}_1$, ... , $\texttt{constr}_n$ 为 $n \geq 1$ 个变体构造子，每个构造子具有两种形式：uiden 和 uiden t，其中 uiden 为以大写字母开头的标识符，而 t 为一个类型。
- 函数类型：函数类型记作 $\texttt{t}_1$ -> $\texttt{t}_2$，其中 $\texttt{t}_1$ 为参数类型，而 $\texttt{t}_2$ 为返回值类型。

类型的语法定义如下：

```
type ::=
    unit | int | float | bool          (*base types*)
  | (min .. max)                       (*integer type with a range*)
  | {#iden, #iden...,#iden}            (*scalar type*)
  | (type1 , ... , typen)              (*tuple type*)
  | array type                         (*array type*)
  | list type                          (*list type*)
  | record_type                        (*record type*)
  | variant_type                       (*variant type*)
  | type -> type                       (*function type*)

variant_type ::= constr | ... | constr
constr  ::= uiden | uiden type
record_type  ::= { iden : type ; ... ; iden : type ;}
```

A.5 值、类型以及函数的声明

在 SCTLProV 的输入文件中，为了程序的可读性以及方便书写，我们可用标识符来代指先前所定义的值与类型以及声明函数。

- 值与类型的声明：值的声明语法如下。

  ```
  value_def ::= value iden = expr (*value definition*)
  ```

  类型的声明语法如下。

  ```
  type_decl ::=  datatype iden = type (*to define new types*)
  ```

- 函数的声明：函数的声明语法如下。

```
fun_decl   ::=
     function iden ( parameter,...,parameter ) : type = expr
parameter  ::=  iden | ( parameter,...,parameter )
```

A.6 Kripke 模型的声明

以上介绍的所有语法结构的目的是为了定义 Kripke 模型。在 SCTLProV 的输入语言中，一个 Kripke 模型的定义分为 6 个部分：状态变量声明、初始状态、迁移规则、原子公式、公平性约束以及性质描述。

1. 状态变量声明：在传统的模型检测工具的输入语言中，状态被定义为对若干个状态变量的赋值，对状态变量的不同赋值代表了不同的状态。在 SCTLProV 的输入语言中同样保留了这种定义状态的方式。然而，一个更一般化的定义状态的方式是将状态定义为值，即用一个值来表示一个状态。这样定义的状态就可以具有更复杂的类型，从而在表示复杂系统的建模中更方便。因此，状态变量声明在输入文件中被作为一个可选项。

2. 初始状态：当输入文件中存在状态变量声明，那么 Kripke 模型中必须有初始状态的定义。此时，初始状态被定义为对所声明的所有状态变量的一个赋值。当输入文件中不存在状态变量的声明时，初始状态就可用值的声明来定义。

3. 迁移规则：迁移规则的定义分为两种，第一种类似于传统的模型检测工具的输入语言中的定义，即列出若干个布尔表达式及其对应的状态变换方式，依次规约布尔表达式，并且当布尔表达式规约到值 `true` 时依据其对应的状态变换方式进行状态的迁移；第二种定义迁移规则的方式则更一般化：将要迁移的状态作为参数传递给一个函数，该函数会返回一个状态的列表，那么该列表中的所有状态则为当前要迁移的状态的后继。

4. 原子公式：原子公式被定义为布尔表达式，判断原子公式的满足性时，需将若干个状态作为参数带入到相应的布尔表达式中，如果代入后的布尔表达式可规约到值 `true` 则该原子公式为真，否则为假。

5. 公平性约束：公平性约束指的是若干个 CTL_P 公式的定义。

6. 性质描述：Kripke 模型的性质同样被定义为若干个 CTL_P 公式，不同于公平性约束，在性质描述种，为了可读性与叙述验证结果方便，通常将每个性质绑定到一个标识符上。

Kripke 模型声明的语法如下。

```
kripke_decl ::=
   Model {
     (*Define states as lists of state variable bindings (a record),
     this is optional.*)
     Vars { iden : type ; ... ; iden : type ; }
     (*Define the initial state (refered to as either "ini" or
     "init"), also optional.*)
     Init { iden := expr ; ... ; iden := expr ; }
     (*Define the transition relation.*)
     Transition { next iden := transitions }
     (*Define atomic formulae.*)
     Atomic { {iden ( iden , ... , iden ) := expr ;}* }
     (*Define fairness constraints, optional.*)
     Fairness { formula ; ... ; formula }
     (*Define the specification.*)
     Spec { iden := formula ; ... ; iden := formula ;}
   }
trasitions ::= expr | expr : expr ; ... ; expr : expr ;
formula ::=
     iden ( iden , ... , iden )
   | not formula
   | formula /\ formula
   | formula \/ formula
   | EX ( iden , formula , expr )
   | AX ( iden , formula , expr )
   | EG ( iden , formula , expr )
   | AF ( iden , formula , expr )
   | EU ( iden , iden , formula , formula , expr )
```

```
  | AR ( iden , iden , formula , formula , expr )
```

A.7 输入文件的结构

SCTLProV 在每次验证过程中可接受一个或多个输入文件。当只有一个输入文件时，那么该输入文件被作为主文件输入到 SCTLProV 中。主文件中包含若干个值、类型以及函数的声明，以及一个 Kripke 模型的声明。当有多个输入文件中，除了一个主文件以外，还有若干个模块文件，模块文件中只可包含值、类型或者函数的定义，而不包含 Kripke 模型的定义。多个输入文件利用 `import` 语句来确定依赖关系，即一个模块文件的内容可通过 `import` 引入到别的输入文件中。`import` 语句的参数为以大写字母开头的标识符，这个标识符代表的即为要引入的模块文件的文件名（如果文件名以小写字母开头，则将其起始字母变为大写字母）。

```
module ::=
    import uiden ... import uiden
    decl ... decl
decl   ::= type_decl | value_decl | fun_decl
main_file ::= kripke_decl | module kripke_decl
```

附录 B　VMDV 与定理证明工具的交互协议

VMDV 与不同的定理证明工具的交互是通过互相发送与解析 JSON 形式的消息来实现的，每个 JSON 对象代表一个消息，消息的详细说明如下：

- **初始化显示窗口**（定理证明器 ⟶ VMDV）：在 VMDV 中，我们将每个显示窗口成为一个会话（Session）。当定理证明器需要可视化一个数据结构（比如证明树）的时候，定理证明器发送一个“create_session”消息，指定需要可视化的数据结构的名称（“session_id”）、简介（“session_descr”）及类型（“graph_type”）。其中，定理证明器指定的可视化的数据结构的类型分为两种：树（“Tree”）和有向图（“DiGraph”），前者通常用来表示证明树，后者通常用来表示可能有环的有向图。VMDV 接收到该消息之后立即初始化并显示一个窗口，并等待定理证明器的进一步消息。

```
{
    "type": "create_session",
    "session_id": string,
    "session_descr": string,
    "graph_type": string
}
```

- **添加节点**（定理证明器 ⟶ VMDV）：向 VMDV 相应的会话中加入一个节点，并指定节点的标识（“id”）、要显示的字符串（“label”）、状态（“state”）。其中，节点的状态分为两种：已证明（“Proved”）和未证明（“Not_Proved”），不同状态的节点通常用不同的颜色高亮显示。

```
{
    "type": "add_node",
    "session_id": string,
    "node":
    {
        "id": string,
        "label": string,
```

```
        "state": string
    }
}
```

- **删除节点**（定理证明器 ⟶ VMDV）：在指定的会话中删除一个指定的节点。

```
{
    "type": "remove_node",
    "session_id": string,
    "node_id": string
}
```

- **添加边**（定理证明器 ⟶ VMDV）：在指定的会话中添加一条由“from_id”节点指向“to_id”节点的边，同时指定该条边上要显示的字符串（“label”）。

```
{
    "type": "add_edge",
    "session_id": string,
    "from_id": string,
    "to_id": string,
    "label": string
}
```

- **删除边**（定理证明器 ⟶ VMDV）：在指定的会话中删除由“from_id”节点指向“to_id”节点的边。

```
{
    "type": "remove_node",
    "session_id": string,
    "node_id": string
}
```

- **高亮节点**（定理证明器 ⟷ VMDV）：高亮节点消息可由定理证明器和 VMDV 互相发送给对方。当定理证明器向 VMDV 发送该消息时，VMDV

在相应的会话中将指定的节点用不同于节点当前的颜色高亮显示。当用户在 VMDV 窗口中执行鼠标选中或者搜索操作时，被选中或者搜索到的节点在窗口中被高亮显示的同时，将所有节点的高亮消息发送到定理证明器。当定理证明器同时在 VMDV 中可视化多个数据结构时，高亮节点消息的传递过程可用来实现多个数据结构的交互显示。比如，在可视化 SCTLProV 的证明结果的时候，VMDV 通常会初始化两个窗口，分别可视化当前公式的证明树，以及在搜索当前证明树的过程中所访问到的 Kripke 模型的状态。当用鼠标选中或搜索到证明树的某个或某些节点时，VMDV 将这些证明树节点的高亮信息发送给 SCTLProV，同时 SCTLProV 识别相应的证明树节点，并计算出与这些证明树节点相关的状态，然后将这些状态节点的高亮信息发送给 VMDV，于是 VMDV 在状态图窗口中高亮显示相应的状态。

```
{
    "type": "remove_node",
    "session_id": string,
    "node_id": string
}
```

- **取消高亮节点**（定理证明器 ⟷ VMDV）：同高亮节点消息一样，取消高亮节点的消息同样可由定理证明器和 VMDV 互相发送给对方。

```
{
    "type": "unhighlight_node",
    "session_id": string,
    "node_id": string
}
```

- **取消所有高亮**（定理证明器 ⟷ VMDV）：去掉所有节点的高亮颜色。

```
{
    "type": "clear_color",
    "session_id": string
}
```

- **删除会话**（定理证明器 ⟶ VMDV）：VMDV 收到删除会话消息后，销毁对应的窗口，结束一个数据结构的可视化过程。

```
{
    "type": "remove_session",
    "session_id": string
}
```

当定理证明器或 VMDV 收到对方发来的控制消息后，会发送回对方一个反馈消息，用来表明该控制消息是否被成功解析：

- **成功解析**（定理证明器 ⟷ VMDV）：

```
{
    "type": "feedback",
    "session_id": string,
    "status": "OK"
}
```

- **解析失败**（定理证明器 ⟷ VMDV）：

```
{
    "type": "feedback",
    "session_id": string,
    "status": "Fail",
    "error_msg": string
}
```

附录 C　部分实验结果的详细数据

性质	进程数	互斥算法							
		Verds		NuSMV		NuXMV		SCTLProV	
		sec	MB	sec	MB	sec	MB	sec	MB
P_1	6	0.286	321.99	0.153	9.07	0.270	21.18	0.005	2.25
	12	1.278	322.08	19.506	76.98	21.848	89.25	0.016	3.70
	18	4.719	426.45	-	-	-	-	0.037	5.44
	24	11.989	601.55	-	-	-	-	0.091	9.36
	30	26.511	926.25	-	-	-	-	0.200	16.49
	36	52.473	1287.57	-	-	-	-	0.418	27.46
	42	100.071	1944.95	-	-	-	-	0.682	48.28
	48	-	-	-	-	-	-	1.119	66.63
	51	-	-	-	-	-	-	1.392	82.32
P_2	6	0.375	322.07	0.054	9.07	0.048	21.31	0.012	3.07
	12	2.011	322.02	22.774	76.96	21.733	89.24	0.035	4.44
	18	7.958	446.71	-	-	-	-	0.101	8.09
	24	23.448	692.30	-	-	-	-	0.252	14.57
	30	48.800	1026.48	-	-	-	-	0.509	23.61
	36	105.183	1619.01	-	-	-	-	1.005	50.49
	42	-	-	-	-	-	-	1.791	57.93
	48	-	-	-	-	-	-	2.679	86.67
	51	-	-	-	-	-	-	3.453	129.83
P_3	6	0.331	322.02	0.089	9.04	0.033	21.27	0.012	3.03
	12	2.059	322.07	22.749	76.91	21.897	89.22	0.035	4.93
	18	7.995	449.13	-	-	-	-	0.110	9.59
	24	23.578	696.74	-	-	-	-	0.286	21.04
	30	51.774	1138.27	-	-	-	-	0.643	30.09
	36	106.027	1628.84	-	-	-	-	1.287	66.14
	42	-	-	-	-	-	-	2.138	86.29
	48	-	-	-	-	-	-	3.369	170.94
	51	-	-	-	-	-	-	4.333	149.03
P_4	6	0.446	321.97	0.089	9.04	0.033	21.27	0.039	3.38
	12	8.289	552.62	22.749	76.91	21.897	89.22	150.115	986.64
	18	-	-	-	-	-	-	-	-
	24	-	-	-	-	-	-	-	-
	30	-	-	-	-	-	-	-	-
	36	-	-	-	-	-	-	-	-
	42	-	-	-	-	-	-	-	-
	48	-	-	-	-	-	-	-	-
	51	-	-	-	-	-	-	-	-
P_5	6	0.430	322.03	0.031	9.09	0.047	21.19	0.011	3.10
	12	3.398	363.78	22.747	77.01	22.029	89.17	0.040	4.81
	18	18.176	783.24	-	-	-	-	0.115	10.99
	24	87.432	2382.82	-	-	-	-	0.322	18.68
	30	-	-	-	-	-	-	1.414	47.68
	36	-	-	-	-	-	-	1.287	66.35
	42	-	-	-	-	-	-	2.405	142.86
	48	-	-	-	-	-	-	4.848	225.55
	51	-	-	-	-	-	-	5.177	225.66

表 C.1 测试集三中互斥算法测试用例的实验数据

性质	进程数	环算法							
		Verds		NuSMV		NuXMV		SCTLProV	
		sec	MB	sec	MB	sec	MB	sec	MB
P_1	3	0.168	322.09	0.040	10.02	0.045	22.08	4.622	62.22
	4	0.216	322.12	0.299	22.46	0.255	34.96	-	-
	5	0.301	322.07	2.421	59.31	1.195	71.53	-	-
	6	0.449	322.13	22.127	80.49	17.967	92.82	-	-
	7	0.740	322.19	147.895	224.17	131.735	236.50	-	-
	8	1.115	322.09	1135.882	865.04	1083.48	877.36	-	-
	9	1.646	322.07	-	-	-	-	-	-
	10	2.232	321.96	-	-	-	-	-	-
P_2	3	-	-	0.058	10.74	0.068	22.73	0.031	3.22
	4	-	-	0.583	40.29	0.562	52.61	0.125	3.73
	5	-	-	5.164	62.29	5.295	74.62	0.444	4.05
	6	-	-	39.085	81.85	37.969	93.96	1.373	4.71
	7	-	-	246.123	229.07	241.375	241.15	3.745	6.03
	8	-	-	-	-	-	-	9.154	7.61
	9	-	-	-	-	-	-	19.997	10.07
	10	-	-	-	-	-	-	40.331	13.05
P_3	3	-	-	0.045	10.03	0.071	22.32	0.022	3.20
	4	-	-	0.296	22.46	0.299	34.96	0.820	13.11
	5	-	-	2.357	59.31	2.526	71.63	111.96	676.29
	6	-	-	22.147	80.49	21.304	92.93	-	-
	7	-	-	147.567	224.17	141.134	236.74	-	-
	8	-	-	-	-	-	-	-	-
	9	-	-	-	-	-	-	-	-
	10	-	-	-	-	-	-	-	-
P_4	3	0.158	322.09	0.066	10.00	0.171	22.32	0.024	3.24
	4	0.190	322.05	0.356	22.46	0.367	34.95	0.104	3.82
	5	0.263	322.04	2.726	59.31	2.781	71.63	0.385	3.99
	6	0.385	322.07	27.013	80.48	24.794	94.95	1.289	4.57
	7	0.528	322.07	181.007	224.16	166.725	236.61	3.727	5.29
	8	0.815	322.14	-	-	-	-	9.525	7.14
	9	1.138	322.19	-	-	-	-	21.568	9.31
	10	1.574	321.98	-	-	-	-	45.097	12.95

表 C.2 测试集三中环算法测试用例的实验数据

测试用例	有死锁	SCTLProV		CADP	
		sec	MB	sec	MB
vasy_0_1	No	0.13	27.16	0.40	10.95
cwi_1_2	No	0.13	27.67	0.39	10.80
vasy_1_4	No	0.14	27.75	0.39	10.71
cwi_3_14	Yes	0.14	25.70	0.40	10.82
vasy_5_9	Yes	0.14	25.70	0.40	10.82
vasy_8_24	No	0.17	28.90	0.43	10.79
vasy_8_38	Yes	0.14	25.76	0.39	10.74
vasy_10_56	No	0.20	29.90	0.43	10.86
vasy_18_73	No	0.24	31.68	0.47	11.81
vasy_25_25	Yes	0.97	33.52	2.18	23.26
vasy_40_60	No	0.21	29.42	0.46	15.08
vasy_52_318	No	0.59	41.09	0.65	16.69
vasy_65_2621	No	1.41	77.02	2.09	109.03
vasy_66_1302	No	0.89	34.92	1.25	14.13
vasy_69_520	Yes	0.23	27.47	0.51	11.84
vasy_83_325	Yes	0.21	27.96	0.48	11.32
vasy_116_368	No	0.67	35.27	0.77	14.40
cwi_142_925	Yes	0.28	28.33	0.57	12.72
vasy_157_297	Yes	0.18	27.14	0.45	11.48
vasy_164_1619	No	2.53	48.39	1.53	22.90
vasy_166_651	Yes	0.29	31.19	0.55	13.30
cwi_214_684	Yes	0.39	34.39	0.63	22.94
cwi_371_641	No	1.36	40.41	1.24	42.92
vasy_386_1171	No	2.14	74.11	1.66	45.12
cwi_566_3984	Yes	0.78	38.53	1.11	21.92
vasy_574_13561	No	18.23	246.97	9.72	188.21
vasy_720_390	Yes	0.23	28.49	0.48	12.89
vasy_1112_5290	No	10.2	89.81	6.54	97.47
cwi_2165_8723	No	16.51	166.74	14.55	185.58
cwi_2416_17605	Yes	3.19	87.61	3.38	71.80
vasy_2581_11442	Yes	2.40	74.11	2.68	58.43
vasy_4220_13944	Yes	2.85	89.50	3.20	73.82
vasy_4338_15666	Yes	3.41	96.21	3.83	80.59
vasy_6020_19353	No	37.19	456.34	74.24	649.41
vasy_6120_11031	Yes	2.35	82.57	2.60	67.01
cwi_7838_59101	No	72.76	1013.67	140.21	1019.55
vasy_8082_42933	No	7.85	309.74	7.82	240.69
vasy_11026_24660	Yes	4.82	149.80	5.15	134.17
vasy_12323_27667	Yes	5.40	164.73	5.67	149.09
cwi_33949_165318	No	366.51	2368.22	636.39	2972.61

表 C.3 SCTLProV 与 CADP 分别验证测试集四中测试用例的死锁性质的实验数据

测试用例	有活锁	SCTLProV		CADP	
		sec	MB	sec	MB
vasy_0_1	No	0.13	27.45	0.48	14.57
cwi_1_2	No	0.14	27.74	0.50	14.61
vasy_1_4	No	0.14	27.70	0.48	14.66
cwi_3_14	No	0.16	28.18	0.50	14.57
vasy_5_9	No	0.16	28.08	0.51	14.68
vasy_8_24	No	0.18	29.09	0.53	14.75
vasy_8_38	No	0.20	28.86	0.52	14.73
vasy_10_56	No	0.23	30.33	0.55	15.34
vasy_18_73	No	0.28	33.07	0.56	16.25
vasy_25_25	No	0.96	33.54	2.25	27.84
vasy_40_60	No	0.26	30.23	0.57	19.65
vasy_52_318	Yes	0.21	27.66	0.55	15.45
vasy_65_2621	No	5.31	280.57	1.98	113.40
vasy_66_1302	No	2.40	38.33	1.30	18.50
vasy_69_520	No	1.04	33.41	0.85	16.39
vasy_83_325	No	0.83	39.27	0.76	19.40
vasy_116_368	No	1.19	42.14	0.86	15.74
cwi_142_925	No	2.67	46.30	1.22	17.89
vasy_157_297	No	0.80	33.99	0.78	17.91
vasy_164_1619	No	3.69	53.30	1.51	23.44
vasy_166_651	No	1.60	49.98	1.02	26.02
cwi_214_684	Yes	0.26	29.93	0.63	16.81
cwi_371_641	Yes	0.26	30.63	0.62	17.41
vasy_386_1171	No	2.91	80.16	1.55	41.75
cwi_566_3984	No	13.32	106.25	3.95	54.08
vasy_574_13561	No	27.02	272.11	8.17	188.69
vasy_720_390	No	0.86	31.45	0.76	17.45
vasy_1112_5290	No	10.49	89.86	5.24	97.93
cwi_2165_8723	Yes	1.87	61.94	2.15	48.80
cwi_2416_17605	Yes	3.10	87.61	3.44	76.30
vasy_2581_11442	No	32.70	326.38	14.78	214.93
vasy_4220_13944	No	43.93	423.03	24.71	330.85
vasy_4338_15666	No	47.55	479.15	28.06	344.64
vasy_6020_19353	Yes	3.23	100.24	3.57	106.43
vasy_6120_11031	No	30.59	425.64	38.37	437.71
cwi_7838_59101	Yes	11.34	250.68	11.58	236.09
vasy_8082_42933	No	119.77	1123.85	106.49	908.29
vasy_11026_24660	No	60.86	698.85	108.97	804.34
vasy_12323_27667	No	68.50	793.83	134.44	898.61
cwi_33949_165318	Yes	33.89	732.05	34.60	738.78

表 C.4 SCTLProV 与 CADP 分别验证测试集四中测试用例的活锁性质的实验数据

参考文献

[1] CLARKE E M, GRUMBERG O, PELED D. Model checking[M]. Cambridge, MA, USA: MIT Press, 2001.

[2] BOUAJJANI A, JONSSON B, NILSSON M, et al. Regular model checking[C]//Proceedings of Computer Aided Verification, 12th International Conference, CAV 2000. Chicago, IL, USA: Springer-Verlag, Berlin, 2000: 403–418.

[3] BAIER C, KATOEN J. Principles of model checking[M]. USA: MIT Press, 2008.

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

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

[6] HOLZMANN G J. The model checker SPIN[J]. IEEE Trans. Software Eng., 1997, 23(5): 279–295.

[7] GARAVEL H, LANG F, MATEESCU R, et al. CADP 2011: a toolbox for the construction and analysis of distributed processes[J]. STTT, 2013, 15(2): 89–107.

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

[9] BRYANT R E. Graph-based algorithms for boolean function manipulation[J]. IEEE Trans. Computers, 1986, 35(8): 677–691.

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

[11] FITTING M. Graduate texts in computer science: First-order logic and automated theorem proving, second edition[M]. New York: Springer-Verlag, 1996.

[12] LOVELAND D W. Automated theorem proving: A logical basis (fundamental studies in computer science)[M]. Amsterdam: Elsevier, 1978.

[13] BUREL G. Automating theories in intuitionistic logic[C]//Proceedings of Frontiers of Combining Systems, 7th International Symposium, FroCoS 2009. Trento, Italy: Springer-Verlag, Berlin, 2009: 181–197.

[14] GORDON M. From LCF to HOL: a short history[C]//Proof, Language, and Interaction, Essays in Honour of Robin Milner. [S.l.: s.n.], 2000: 169–186.

[15] NIPKOW T. Interactive proof: Introduction to isabelle/hol[M]//Software Safety and Security - Tools for Analysis and Verification. [S.l.: s.n.], 2012: 254–285.

[16] OWRE S, RUSHBY J M, SHANKAR N. PVS: A prototype verification system[C]// Automated Deduction - CADE-11, 11th International Conference on Automated Deduction, Saratoga Springs, NY, USA, June 15-18, 1992, Proceedings. [S.l.: s.n.], 1992: 748–752.

[17] POERNOMO I H, WIRSING M, CROSSLEY J N. Monographs in computer science: Adapting proofs-as-programs - the curry-howard protocol[M]. [S.l.]: Springer, 2005.

[18] BERTOT Y, CASTÉRAN P. Texts in theoretical computer science. an EATCS series: Interactive theorem proving and program development - coq'art: The calculus of inductive constructions[M]. [S.l.]: Springer, 2004.

[19] ALLEN S F, CONSTABLE R L, EATON R, et al. The nuprl open logical environment[C]// Automated Deduction - CADE-17, 17th International Conference on Automated Deduction, Pittsburgh, PA, USA, June 17-20, 2000, Proceedings. [S.l.: s.n.], 2000: 170–176.

[20] KAUFMANN M, MOORE J S. An ACL2 tutorial[C]//Theorem Proving in Higher Order Logics, 21st International Conference, TPHOLs 2008, Montreal, Canada, August 18-21, 2008. Proceedings. [S.l.: s.n.], 2008: 17–21.

[21] PAULSON L C. Isabelle: The next 700 theorem provers[J]. CoRR, 1993, cs.LO/9301106.

[22] RAJAN S, SHANKAR N, SRIVAS M K. An integration of model checking with automated proof checking[C]//Computer Aided Verification, 7th International Conference, Liège, Belgium, July, 3-5, 1995, Proceedings. [S.l.: s.n.], 1995: 84–97.

[23] CAVALLI A R, DEL CERRO L F. A decision method for linear temporal logic[C]//7th International Conference on Automated Deduction, Napa, California, USA, May 14-16, 1984, Proceedings. [S.l.: s.n.], 1984: 113–127.

[24] VENKATESH G. A decision method for temporal logic based on resolution[C]//Foundations of Software Technology and Theoretical Computer Science, Fifth Conference, New Delhi, India, December 16-18, 1985, Proceedings. [S.l.: s.n.], 1985: 272–289.

[25] FISHER M, DIXON C, PEIM M. Clausal temporal resolution[J]. ACM Trans. Comput. Log., 2001, 2(1): 12–56.

[26] ZHANG L, HUSTADT U, DIXON C. A resolution calculus for the branching-time temporal logic CTL[J]. ACM Trans. Comput. Log., 2014, 15(1): 10:1–10:38.

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

[28] DOWEK G, HARDIN T, KIRCHNER C. Theorem proving modulo[J]. J. Autom. Reasoning, 2003, 31(1): 33–72.

[29] DOWEK G. Polarized resolution modulo[C]//Theoretical Computer Science - 6th IFIP TC 1/WG 2.2 International Conference, TCS 2010, Held as Part of WCC 2010, Brisbane, Australia, September 20-23, 2010. Proceedings. [S.l.: s.n.], 2010: 182–196.

[30] DOWEK G, JIANG Y. A logical approach to CTL[EB/OL]. 2013. http://www.lsv.ens-cachan.fr/~dowek/Publi/ctl.pdf.

[31] PNUELI A. The temporal logic of programs[C]//18th Annual Symposium on Foundations of Computer Science, Providence, Rhode Island, USA, 31 October - 1 November 1977. [S.l.: s.n.], 1977: 46–57.

[32] CLARKE E M, EMERSON E A. Design and synthesis of synchronization skeletons using branching time temporal logic[C]//25 Years of Model Checking - History, Achievements, Perspectives. [S.l.: s.n.], 2008: 196–215.

[33] EMERSON E A, HALPERN J Y. ”sometimes” and ”not never” revisited: on branching versus linear time temporal logic[J]. J. ACM, 1986, 33(1): 151–178.

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

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

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

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

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

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

[40] DERSHOWITZ N. Termination of rewriting[J]. J. Symb. Comput., 1987, 3(1/2): 69–116.

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

[42] SILBERSCHATZ A, GALVIN P B, GAGNE G. Operating system concepts - international student version, 9th edition[M]. [S.l.]: Wiley, 2014.

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

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

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

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

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

[48] MUÑOZ C A, DOWEK G, CARREÑO V. 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, 2004: 175–182.

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

[50] BYRNES J, BUCHANAN M, ERNST M, et al. Visualizing proof search for theorem prover development[J]. Electronic Notes in Theoretical Computer Science, 2009, 226: 23–38.

[51] LIBAL T, RIENER M, RUKHAIA M. Advanced proof viewing in prooftool[C]//Proceedings Eleventh Workshop on User Interfaces for Theorem Provers, UITP 2014, Vienna, Austria, 17th July 2014. [S.l.: s.n.], 2014: 35–47.

[52] SAKURAI K, ASAI K. Mikibeta: A general GUI library for visualizing proof trees[M]// Logic-Based Program Synthesis and Transformation. [S.l.]: Springer, 2011: 84–98.

[53] STEEL G. Visualising first-order proof search[C]//Proceedings of User Interfaces for Theorem Provers: volume 2005. [S.l.: s.n.], 2005: 179–189.

[54] FARMER W M, GRIGOROV O G. Panoptes: An exploration tool for formal proofs[J]. Electr. Notes Theor. Comput. Sci., 2009, 226: 39–48.

[55] BAJAJ C, KHANDELWAL S, MOORE J, et al. Interactive symbolic visualization of semi-automatic theorem proving[M]. [S.l.]: Computer Science Department, University of Texas at Austin, 2003.

[56] 张敏, 霍朝光, 霍帆帆, et al. 国际信息可视化知识族群：演化、聚类及迁徙研究[J]. 情报科学, 2016, 4: 13–17.

[57] 冯艳林. 城市道路交通信息可视化及其应用研究[J]. 黑龙江科技信息, 2015, 17: 180.

[58] MATHIEU J, TOMMASO V, SEBASTIEN H, et al. Forceatlas2, a continuous graph layout algorithm for handy network visualization designed for the gephi software[J]. PLOS ONE, 2014, 9(6): e98679.

[59] MARTIN S, BROWN W M, KLAVANS R, et al. Openord: an open-source toolbox for large graph layout[J]. SPIE Proceedings, 2011, 7868.

[60] HU Y. Efficient and high quality force-directed graph drawing[J]. The Matematical Journal, 2005, 10: 149–160.

[61] BIERE A, CIMATTI A, CLARKE E M, et al. Bounded Model Checking[J]. Advances in Computers, 2003, 58: 117–148.

[62] BOUAJJANI A, ESPARZA J, MALER O. Reachability analysis of pushdown automata: Application to model-checking[C]//Proceedings of CONCUR'97: Concurrency Theory. Warsaw, Poland: Springer-Verlag, Berlin, 1997: 135–150.

[63] GIRARD J Y. Proofs and types[M]. [S.l.]: Cambridge University Press, UK, 1989.

[64] KLEENE S C. Introduction to metamathematics[M]. [S.l.]: Ishi Press, California, 2009.

[65] BOLLIG B. On the OBDD complexity of the most significant bit of integer multiplication[J]. Theoretical Computer Science, 2011, 412(18): 1686 – 1695.

[66] GABBAY D M, PNUELI A. A sound and complete deductive system for ctl* verification[J]. Logic JOURNAL of the IGPL, 2008, 16(6): 499–536.

[67] REYNOLDS M. An axiomatization of full computation tree logic[J]. J. Symb. Log., 2001, 66 (3): 1011–1057.

[68] FRIEDMANN O. A proof system for *ctl**[D]. [S.l.]: University of Munich, 2008.

[69] HOLZMANN G J. The SPIN model checker - primer and reference manual[M]. [S.l.]: Addison-Wesley, 2004.

[70] BUREL G. Experimenting with deduction modulo[C]//SOFRONIE-STOKKERMANS V, BJØRNER N. Proceedings of CADE 2011. Wroclaw, Poland: Springer-Verlag, Berlin, 2011: 162–176.

[71] DAVIS M, PUTNAM H. A computing procedure for quantification theory[J]. J. ACM, 1960, 7(3): 201–215.

[72] PNUELI A, KESTEN Y. A deductive proof system for CTL[C]//Proceedings of CONCUR 2002. Brno, Czech Republic: Springer-Verlag, Berlin, 2002: 24–40.

[73] BIERE A, CIMATTI A, CLARKE E M, et al. Bounded model checking[J]. Advances in Computers, 2003, 58: 117–148.

[74] HE F, SONG X, HUNG W N N, et al. Integrating evolutionary computation with abstraction refinement for model checking[J]. IEEE Trans. Computers, 2010, 59(1): 116–126.

[75] CRESPI V, GALSTYAN A, LERMAN K. Top-down vs bottom-up methodologies in multi-agent system design[J]. Auton. Robots, 2008, 24(3): 303–313.

[76] LANGE M, STIRLING C. Model checking games for branching time logics[J]. J. Log. Comput., 2002, 12(4): 623–639.

[77] BRÜNNLER K, LANGE M. Cut-free sequent systems for temporal logic[J]. J. Log. Algebr. Program., 2008, 76(2): 216–225.

[78] DERSHOWITZ N, MANNA Z. Proving termination with multiset orderings[J]. Commun. ACM, 1979, 22(8): 465–476.

[79] JOUANNAUD J, LESCANNE P. On multiset orderings[J]. Inf. Process. Lett., 1982, 15(2): 57–63.

[80] DERSHOWITZ N. Orderings for term-rewriting systems[J]. Theor. Comput. Sci., 1982, 17: 279–301.

[81] TUFTE E R, GRAVES-MORRIS P. The visual display of quantitative information: volume 2 [M]. [S.l.]: Graphics press Cheshire, CT, 1983.

[82] DOWEK G, JIANG Y. Cut-elimination and the decidability of reachability in alternating push-down systems[J/OL]. arXiv preprint arXiv:1410.8470, 2014. https://who.rocq.inria.fr/Gilles.Dowek/Publi/reachability.pdf.

[83] BOUAJJANI A, ESPARZA J, MALER O. Reachability analysis of pushdown automata: Application to model-checking[M]//CONCUR'97: Concurrency Theory. [S.l.]: Springer, 1997: 135–150.

[84] BERTOT Y, CASTÉRAN P. Interactive theorem proving and program development: Coq'art: The calculus of inductive constructions[M]. [S.l.]: Springer Science & Business Media, 2013.

[85] FREEMAN E, ROBSON E, BATES B, et al. Head first design patterns[M]. [S.l.]: "O'Reilly Media, Inc.", 2004.

[86] CLARKE E M, GRUMBERG O, PELED D. Model checking[M]. [S.l.: s.n.], 1999.

[87] BEDERSON B B, SHNEIDERMAN B. The craft of information visualization: readings and reflections[M]. [S.l.]: Morgan Kaufmann, 2003.

[88] TRAC S, PUZIS Y, SUTCLIFFE G. An interactive derivation viewer[J]. Electronic Notes in Theoretical Computer Science, 2007, 174(2): 109–123.

[89] CLARKE E M, GRUMBERG O, MCMILLAN K L, et al. Efficient generation of counterexamples and witnesses in symbolic model checking[C]//DAC. [S.l.: s.n.], 1995: 427–432.

[90] BAIER C, KATOEN J. Principles of model checking[M]. [S.l.]: MIT Press, 2008.

[91] MAZZA R. Introduction to information visualization[M]. [S.l.]: Springer, 2009.

[92] SPENCE R. Information visualization: Design for interaction (2nd edition)[M]. [S.l.]: Prentice Hall, 2007.

研究成果及项目资助

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. Towards Combining Model Checking and Proof Checking. submitted.

公开的工具

[1] SCTLProV: https://github.com/sctlprov/sctlprov_code

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

项目资助情况

1. 中法合作项目 VIP（项目编号 GJHZ1844）

2. 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: liujian@ios.ac.cn

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！