arXiv:2301.11521v1 [cs.LO] 27 Jan 2023

栈感知超性质*

Ali Bajwa[2], Minjian Zhang[1], Rohit Chadha[2], 和 Mahesh Viswanathan[1]

[1] 伊利诺伊大学厄巴纳-香槟分校，美国[2] 密苏里大学哥伦比亚分校，美国

摘要。超性质关联程序的执行，可用于形式化机密性、非干扰、隐私和匿名性等安全目标。形式上，超性质是允许的执行集合构成的集合。若程序的执行集合不属于超性质所指定的集合，则该程序违反该超性质。文献中提出了逻辑 HyperCTL*，用于形式化地规范和验证超性质。已知，判断有限状态程序是否满足 HyperCTL*公式是可判定的。然而，对于过程式（递归）程序，这一问题却是不可判定的。令人惊讶的是，我们证明，如果考虑受限的超性质类别，即仅关联调用栈访问模式相同的程序执行，则可恢复其可判定性。我们将这类超性质称为栈感知超性质。我们的判定过程可用作证明方法，以确立递归程序的非推断等安全目标，也可用于反驳观察确定性等安全目标。此外，如果攻击者能够观察调用栈大小，该判定过程便可提供精确验证。

关键词： 超属性 · 时序逻辑 · 递归程序 · 模型检测 · 下推系统· 可见下推自动机。

1 引言

时序逻辑 HyperLTL 和 HyperCTL* [5] 旨在表达并推理安全保证，这些保证属于超属性 [6]。超性质 [6] 是一种安全保证，并非仅依赖于单个执行。相反，超性质关联多个执行。例如，非干扰是一种机密性性质，它指出：对于程序的任意两次 执行，如果它们仅在高安全级输入上有所不同，那么它们必须具有相同的低安全级观察结果。正如 [6], 所指出的，若干安全保证都是超属性。逻辑 HyperCTL* 包含 HyperLTL，而针对 HyperCTL*公式检查有限状态系统的问题是可判定的 [5]。

* Ali Bajwa 的部分工作获得了美国国家科学基金会 CNS 1553548 项目的资助。Rohit Chadha 的部分工作获得了美国国家科学基金会 CNS 1553548 和 NSFSHF1900924 项目的资助。Mahesh Viswanathan 和 Minjian Zhang 的部分工作获得了美国国家科学基金会 SHF 1901069 和 NSF SHF 2007428 项目的资助。

本文考虑针对安全超属性对过程式（递归）程序进行模型检测的问题。回顾一下，递归程序通常可建模为下推系统。与有限状态迁移系统的情形不同，检查下推系统是否满足 HyperCTL*公式的问题是不可判定的 [16]。相比之下，下推系统上的 CTL* 模型检测是可判定的 [3,18]。

我们的贡献。 我们考虑递归程序中受限的超属性类，即只关联具有相同调用栈访问模式的执行。直观地说，如果两次执行在每一步都以相同方式访问调用栈，那么它们就具有相同的栈访问模式；也就是说，如果一次执行在某个位置进行压栈（出栈），另一次执行也会在同一位置压栈（出栈）。请注意，如果两次执行具有相同的栈访问模式，那么它们在任何时刻的栈大小都相同。我们将此类超属性称为栈感知超性质。

为了描述栈感知超性质，我们将 HyperCTL* 扩展为 sHCTL*。逻辑 sHCTL* 采用双层语法。在第一层，语法与 HyperCTL* 公式相同，并对具有相同栈访问模式的下推系统执行进行解释。在顶层，我们对不同的栈访问模式进行量化。公式 $E\psi$ 为真，当且仅当系统存在某种栈访问模式 ρ，且将下推系统限制为具有栈访问模式 ρ 的执行后，该系统满足 HyperCTL* 公式 ψ.。公式 $A\psi$ 为真，当且仅当对于系统的每种栈访问模式 ρ，将下推系统限制为具有栈访问模式 ρ 的执行后，该系统都满足 Hyper-CTL* 公式 ψ.。图 1（第8页）并列比较了 HyperCTL* 与 sHCTL* 的语法。我们以类似的方式将 HyperLTL 扩展为 sHLTL。请注意，sHCTL* 包含 sHLTL；对于有限状态系统，sHCTL*（sHLTL）分别与 HyperCTL*（HyperLTL）等价，因为有限状态系统的所有执行都具有相同的栈访问模式。

我们证明，sHCTL* 的模型检测问题是可判定的。我们展示了这一结果有助于验证递归程序的三种不同方式。首先，对于非推断 [14], 等可用 HyperLTL 的 $\forall\exists^*$ 片段表达的安全保证，我们可以使用模型检测算法来确定递归程序是否满足 HyperLTL 属性。其次，对于 HyperLTL 的 $\forall^*$ 片段，可以使用模型检测算法，通过确定递归程序不满足安全保证来检测安全缺陷。观察确定性 [13,19] 就是此类属性的一个例子。最后，当攻击者能够观察栈访问模式（或等价地，栈大小）时，我们可以对若干属性进行精确验证。例如，在程序计数器安全模型 [15] 中，攻击者在每一步都能访问程序计数器，因此该模型满足攻击者观察栈访问模式这一假设。正如 [15], 中所论证的，程序安全模型适合用于刻画控制流侧信道，例如由时序行为和／或错误信息泄露引起的侧信道。

该判定过程采用了一种自动机理论方法，其灵感来自有限状态系统和 HyperCTL* 的模型检测算法，该算法发表于

[10]。由于栈感知超性质只关联栈访问模式相同的执行，因此，具有相同栈访问模式的一组执行可以编码为一个基于下推字母表的字，其中 [3] ，而 sHCTL*公式的模型检测问题可以归约为检查一个在无限字上运行的非确定性可见下推自动机（*NVPA*）的空性问题 [1]。将模型检测问题归约为空性问题，是基于对每个子公式进行组合式构造：为每个子公式构造一个自动机，使其恰好接受满足该子公式的路径变量赋值集合。为了使这一构造达到最优，我们充分利用了无限字上两个表达能力等价的自动机类，即 NVPA 和单向交替跳转自动机（*1-AJA*） [4]。针对过程式程序进行 sHCTL* 模型检测的算法，其复杂度与针对有限状态系统进行 HyperCTL* 模型检测的复杂度非常接近。若$g(k,n)$表示高度为 k的指数塔，其中最高层指数为 poly(n)，那么，对于公式复杂度为 r的公式 [4] ，以及大小受 n限制的系统和公式，我们的算法属于确定性时间复杂度 DTIME($g(\lceil \frac{r}{2} \rceil, n)$)。相比之下，针对有限状态系统进行 HyperCTL* 模型检测属于非确定性空间复杂度 NSPACE($g(\lceil \frac{r}{2} \rceil - 1, n)$)。这种轻微的复杂度差异，与检查有限状态系统（非确定性对数空间）和过程式程序（多项式时间）的其他性质（如不变式）时的情况一致。

我们还通过证明一个匹配的下界，证明了我们的模型检测算法是最优的。为了证明公式复杂度为 r 的公式的模型检测问题具有 DTIME($g(\lceil \frac{r}{2} \rceil, n)$-困难性，我们将 $g(\lceil \frac{r}{2} \rceil - 1, n)$ 空间受限交替图灵机 (ATM) 的成员资格问题归约到模型检测问题。该归约需要找出一种编码，将 ATM 的计算（这些计算是树）表示为可由下推系统猜测并生成的字符串。我们为模型检测问题构造的下推系统会猜测 ATM 可能的计算，而我们构造的 sHCTL* 公式则检查所猜测的计算是否为有效的接受计算。

相关工作。 Clarkson 和 Schneider 提出了超属性 [6] ，并说明了它们对于刻画复杂安全属性的必要性。用于描述超属性的时序逻辑 HyperLTL 和 HyperCTL* 由 Clarkson 等人提出 [5]。他们还通过将问题归约到 QPTL 的可满足性问题，刻画了针对 HyperCTL* 规约对有限状态转移系统进行模型检测的复杂度 [17]。随后，研究者提出了其他用于验证有限状态系统是否满足 HyperCTL* 属性的模型检测算法 [10,7]。此外，还开发了用于检查 HyperLTL公式的可满足性 [8] 和运行时验证 [9] 的工具。Finkbeiner 等人将自动机理论方法引入有限状态系统的 HyperCTL* 模型检测 [10]。

[3] 下推字母表是划分为三个集合的字母表：调用符号集合、内部符号集合和返回符号集合。参见第 4.1 节。[4] 我们对公式复杂度的定义大致是通常所说的量词交替次数的两倍。精确定义请参见定义 4。

文献 [16]证明，HyperLTL 的模型检测问题，以及由此推得的 Hyper-CTL* 模型检测问题，在下推系统上是不可判定的。对于 HyperLTL 的受限片段，Pommellet 和 Tayssir [16] 引入了过近似和欠近似，以判定或否定下推系统满足这些片段中某个 HyperLTL公式。Gutsfeld 等人提出了文献 [12] 中的 stuttering，这是一种用于检查递归程序异步超属性的线性 时序逻辑。作者在假设公平性并限制良好对齐的条件下，给出了模型检测问题的复杂度结果。尽管将路径限制为具有相同栈访问模式与良好对齐限制相似，但我们不假设任何公平性条件来建立可判定性。不过，由于 sHCTL* 是分支时序逻辑，且只考虑同步超属性，因此这两种逻辑无法直接比较。还值得一提的是，sHCTL* 的分支特性要求我们在为“当前”量化路径变量赋值时，从最近量化的路径变量复制一个可能无界的栈。相比之下，文献 [12] 中的所有路径赋值都从空栈开始。

本文的扩展摘要发表于第29届国际系统构造与分析工具和算法会议（ TACAS） [2]。

2 动机

Clarkson 和 Schneider [6] 指出，许多重要的安全保证只能表示为超属性。我们讨论两个安全超属性的示例，以及用于描述它们的逻辑 HyperLTL 和 HyperCTL*。

超属性与时序逻辑。 我们讨论非**干扰**的两个变体，这些变体用于建模机密性要求。在非干扰中，系统的输入被划分为低级输入安全变量和高级输入安全变量。假设攻击者知道低级安全输入的值。在执行期间，攻击者可以观察系统配置的某些部分，例如系统输出或内存使用量。如果攻击者无法从低安全级观测中推断出高级输入的值，则系统满足非干扰。为了形式化地描述这些变体，我们使用逻辑 HyperLTL [5], ，它是逻辑 HyperCTL* [5]的一个片段。HyperLTL 和 HyperCTL* 的精确语法如图 1 所示。在该语法中， π 是一个路径变量；如果命题 a 沿路径“π”为真，则公式 a_π 为真。直观地说，公式 $\exists \pi.\, \psi$ 表示对路径进行存在量化；如果存在一条可以赋值给 π 的路径，使得 ψ 为真，则该公式为真。全称量化（$\forall \pi.\, \psi$）以及其他逻辑联结词，例如合取（$\wedge$）、蕴含（$\rightarrow$）、等价（$\leftrightarrow$）和时序算子 G、F，都可以按标准方式定义。通过引入显式的路径变量，HyperLTL 和 HyperCTL* 允许同时对多条路径进行量化。

示例 *1.* 第一个变体“非推断” [14], 指出，对于程序的一次执行 σ，存在另一次执行 σ' ，使得 (a) σ' 是通过以下方式从 σ 获得的：

将高级安全输入替换为虚拟输入，且 (b) σ 和 σ' 具有相同的低安全级观测。非推断是一种超活性属性 [5,6]。

假设配置的低级观察由命题的取值决定，如 $L = \{\ell_1, \cdots \ell_m\}$. 所示。在 [5], 中，非干涉可用以下 HyperLTL公式表示：$\mathrm{NI}_{\mathrm{def}} = \forall\pi.\exists\pi'.(\mathsf{G}\,\lambda_{\pi'}) \wedge \pi \equiv_L \pi'.$ 这里，$\mathsf{G}\,\lambda_{\pi'}$ 表示全局（或在执行的每个配置中），π' 的高安全级输入都是虚拟输入 λ，而 $\pi \equiv_L \pi' \stackrel{\mathrm{def}}{=} \mathsf{G}(\wedge_{\ell\in L}(\ell_\pi \leftrightarrow \ell_{\pi'}))$表示 π 和 π' 具有相同的低级观察。

例 *2.* 第二种变体——观察确定性 [13,19], 指出，任何两个具有相同低安全级初始输入的执行，都必须具有相同的低安全级输出观测。观察确定性是一种超安全

性质 [5,6], ，也可以使用公式 [5]通过 HyperLTL 表示：$\mathrm{OD}_{\mathrm{def}} = \forall\pi.\forall\pi'.(\pi[0] \equiv_{L,\ in} \pi'[0]) \rightarrow \pi \equiv_{L,\ out} \pi'.$ 这里，$\equiv_{L,\ in}$和$\equiv_{L,\ out}$分别表示 π 和 π' 具有相同的低安全级输入和输出。

过程式（递归）程序与栈感知超性质。下推系统用于对不动态分配内存、且程序变量取值属于有限域的过程式程序建模。与有限状态转移系统不同，判定下推系统是否满足 HyperCTL*公式是不可判定的 [16]。然而，我们找到了一个自然的超属性类，对于该类，模型检测问题是可判定的。正如我们接下来将看到的，这类超属性不仅具有可判定性，而且有助于推理非推断和观察确定性等安全超属性。

我们考虑递归程序的一类受限超属性，它们只关联以相同方式访问调用栈的执行，即在同一时刻进行压栈或出栈。下推系统 $\mathcal{P}$ 的一次执行是配置（控制状态 $+$ 栈） $\sigma = c_1c_2\cdots$ 组成的序列，其中连续配置 c_i 和 c_{i+1} 的栈只因栈顶可能多出一个元素而有所不同，这个元素可能出现在 c_i 或 c_{i+1}的栈顶。对于这样的序列，我们可以关联一个序列 $\mathsf{pr}(\sigma) = \mathsf{o}_1\mathsf{o}_2\cdots$，其中 $\mathsf{o}_i \in \{$ call,int,ret$\}$ ，且当且仅当 c_{i+1} 中的栈比 c_i. 多（分别少）一个元素时，$\mathsf{o}_i =$ 才为 call（分别为 ret）。序列 $\mathsf{pr}(\sigma)$ 称为 σ 的栈访问模式。请注意，具有相同栈访问模式的两次执行，其栈大小的变化方式相似。因此，等价地，我们可以将这类超属性视为关联内存使用量相同的执行的超属性。

为描述这些超属性，我们提出了逻辑 sHCTL*，它扩展了 HyperCTL*。sHCTL*采用双层语法。在最内层，语法与 HyperCTL*公式的语法相同，但其解释对象是具有相同栈访问模式的下推系统执行。在外层，我们对不同的栈访问模式进行量化。直观而言，公式 $E\psi$ 为真，当且仅当系统所呈现的某个栈访问模式 ρ 存在，且具有访问模式 ρ 的执行集合满足超性质 ψ.。对偶公式 $A\psi$ 定义为 $\neg E\neg\psi$；当且仅当对于每个栈访问模式

ρ （由系统呈现），所有具有栈访问模式 ρ 的执行集合都满足 ψ. 。sHLTL 的语法以类似方式从 HyperLTL 得到。请参见第8页的图1，其中并列展示了 HyperCTL*（HyperLTL）与 sHCTL*（sHLTL）的语法。与 HyperCTL* 不同，我们证明了对于下推系统，判定 sHCTL* 的问题是可判定的（定理 3）。关于 sHCTL* 的栈访问模式、语法和语义的形式化定义见第3节。

本文余下部分中，可用 sHCTL* 表达的超属性称为栈感知超性质。如下文所述，将范围限制在栈感知超性质有助于验证递归程序的安全保证。

证明 $\forall\exists^*$ **超属性**。 属性非推断 （示例 1）可用 HyperLTL 表达为 NI def $= \forall\pi.\exists\pi.'$(G $\lambda_{\pi'}$) $\wedge\pi \equiv_L \pi'$。考虑在 NI 前加上 A 而得到的 sHLTL 公式 A(NI)。下推系统满足 A(NI) 的前提是：对于系统的每次执行 σ，都存在另一次执行 σ'，其栈访问模式与 σ 相同，并且 σ、σ' 一起满足 (G $\lambda_{\sigma'}$) $\wedge\sigma \equiv_L \sigma'$. 。因此，若下推系统满足 sHLTL 公式 A(NI)，则它也满足非推断。因此，可使用 sHLTL 的判定过程证明递归程序满足非推断。

上述观察可推广至形如 $\psi = \forall\pi.\exists\pi_1.\ldots.\exists\pi_k.\psi'$ 的 HyperLTL 公式——如果系统满足 sHLTL 公式 $A\psi$，那么它也必须满足 HyperLTL 公式 ψ. 。尽管即使将模型检测问题限制在此类 HyperLTL 公式上，对于下推系统仍然是不可判定的，但通过限制对 π、π_1、.....、π_k 的搜索空间，我们可以获得可判定性。

反驳 $\forall^*$ **超属性**。观察确定性（例 2）可用 HyperLTL 表示为 OD def $= \forall\pi.\forall\pi'.(\pi[0] \equiv_{L,\ in} \pi'[0]) \rightarrow \pi \equiv_{L,\ out} \pi'$.考虑 sHLTL 公式 A(OD)。只有当存在栈访问模式 ρ 以及执行 σ_1 和 σ_2，其栈访问模式为 ρ，且该下推系统不满足 $(\sigma[0] \equiv_{L,\ in} \sigma'[0]) \rightarrow \sigma \equiv_{L,\ out} \sigma'$. 时，下推系统才不满足 sHLTL 公式 A(OD)。

这一观察可以推广到形如 $\psi = \forall\pi_1.\ldots.\forall\pi_k.\psi'$ 的 HyperLTL公式：如果下推系统不满足 sHLTL公式$A\psi$，那么它就不满足 ψ. 。尽管针对这类受限规约对下推系统进行模型检测是不可判定的，但我们的判定过程可用于证明递归程序不满足此类性质。

当栈访问模式可观察时进行精确验证。通常，假设攻击者能够观察栈访问模式是合理的。例如，在程序计数器安全模型中， [15], 攻击者可以访问程序计数器轨迹，即一次执行期间程序计数器的序列。能够访问程序计数器轨迹意味着攻击者可以观察栈访问模式。假设程序计数器轨迹可观察，有助于对控制流侧信道攻击进行建模，其中包括时序攻击和错误泄露攻击 [15]。sHCTL* 可用于验证此安全模型中的安全保证。例如，sHCTL*公式A(NI)通过为每个控制状态引入一个唯一命题，忠实地对非推断进行建模

。通过适当转换下推自动机，也可以在此模型中验证观察确定性。

另一种可以观察到栈访问模式的情形是，攻击者能够根据栈大小观察程序的内存使用量。由于观察栈大小可能导致信息泄漏，因此应将栈大小视为低安全级观测。由于栈大小可能没有上界，因此无法将其建模为命题。不过，在这种情况下，仍可使用 sHCTL* 验证安全保证。例如， $A(\text{NI}) = A(\forall\pi.\exists\pi'.(\text{G}\ \lambda_{\pi'}) \wedge \pi \equiv_L \pi')$ 忠实地刻画了非干扰，因为 sHCTL* 的语义要求 π 和 π' 除其他低安全级观测外，还具有相同的调用栈大小。同样，也可以通过适当转换下推自动机，在该模型中验证观察确定性。

3 栈感知超计算树逻辑（sHCTL*）

本文正式介绍栈感知超计算树逻辑（sHCTL*）及其子逻辑——栈感知超线性时序逻辑（sHLTL）。我们首先确立关于字符串的一些约定。

字符串。有限字母表 Σ 上的字符串/词 w 是由 Σ 中有限或无限多个符号组成的序列 $w = a_0a_1\cdots$，即 $a_i \in \Sigma$ 对所有 i 都成立。字符串 w 的长度记作 $|w|$，表示其中出现的符号数量——如果 $w = a_0a_1\cdots a_{n-1}$ 是有限的，则 $|w| = n$；如果 $w = a_0a_1\cdots$ 是无限的，则 $|w| = \omega$。长度为 0 的唯一字符串，即空字符串，记作 ε。对于字符串 $w = a_0a_1\cdots a_i\cdots$，$w(i) = a_i$ 表示第 i 个符号，$w[:i] = a_0a_1\cdots a_{i-1}$ 表示长度为 i 的前缀，$w[i:] = a_ia_{i+1}\cdots$ 表示从位置 i 开始的后缀 w，而 $w[i:j] = a_ia_{i+1}\cdots a_{j-1}$ 表示从位置 i（含）到位置 j（不含）的子串。因此，$w[0:] = w$。按照约定，当 $i \le 0$ 时，我们取 $w[:i] = \varepsilon$。在 Σ 上，所有有限字符串构成的集合记作 Σ^*，所有无限字符串构成的集合记作 Σ^ω。对于有限字符串 u 和（有限或无限）字符串 v，uv 表示 u 与 v 的连接。

3.1 下推系统

下推系统可自然地为顺序递归程序建模。形式上，AP 标记下推系统是一个元组 $\mathcal{P} = (S, \Gamma, s_{\mathsf{in}}, \Delta, L)$，其中 S 是有限的控制状态集合，Γ 是有限的栈符号集合，$s_{\mathsf{in}} \in S$ 是初始控制状态，$L : S \to 2^{\mathsf{AP}}$ 是标记函数，而 Δ 是转移关系。转移关系 $\Delta = \Delta_{\mathsf{int}} \cup \Delta_{\mathsf{call}} \cup \Delta_{\mathsf{ret}}$ 是内部转移 $\Delta_{\mathsf{int}} \subseteq S \times S$（其间栈保持不变）、调用转移 $\Delta_{\mathsf{call}} \subseteq S \times (S \times \Gamma)$（将单个符号压入栈中）和返回转移 $\Delta_{\mathsf{ret}} \subseteq (S \times \Gamma) \times S$（从栈中弹出单个符号）的不相交并运算。当 AP 的含义可从上下文中明确得知时，我们简称其为下推系统。转移系统语义。我们回顾下推系统作为转移系统的标准语义。固定一个下推系统 $\mathcal{P} = (S, \Gamma, s_{\mathsf{in}}, \Delta, L)$。$\mathcal{P}$ 的一个配置 c 是一个有序对 (s, α)，其中 $s \in S$ 且 $\alpha \in \Gamma^*$。

$^{a \in \text{原子命题}}, \pi \in \mathcal{V}$

$\psi ::= a_\pi \mid \neg\psi \mid \psi \vee \psi \mid \mathrm{X}\psi \mid \psi \,\mathrm{U}\, \psi \mid \exists \pi.\, \psi$

(a) HyperCTL*

$\theta ::= E\psi \mid \neg\theta \mid \theta \vee \theta$

$\psi ::= a_\pi \mid \neg\psi \mid \psi \vee \psi \mid \mathrm{X}\psi \mid \psi \,\mathrm{U}\, \psi \mid \exists \pi.\, \psi$

(b) sHCTL*

图1: HyperCTL* 和 sHCTL* 的巴科斯-诺尔范式。令 $\forall$ 表示 $\neg\exists\neg$，令 A 表示 $\neg E\neg\psi$. HyperLTL 是 HyperCTL* 公式的集合 $Q_1\pi_1.\cdots Q_r\pi_r.\psi$，其中 $Q_i \in \{\exists, \forall\}$ 且 ψ 为无量词公式。sHLTL 是 sHCTL*公式 $\mathrm{q}\varphi$ 的集合，其中 $\mathrm{q} \in \{A, E\}$ 且 φ 属于 HyperLTL。

$\mathcal{P}$ 的所有配置构成的集合记为 $\mathsf{Conf}_\mathcal{P} = S \times \Gamma^*$。与 $\mathcal{P}$ 相关联的标记迁移系统为 $[\![\mathcal{P}]\!] := (\mathsf{Conf}_\mathcal{P}, c_{\mathsf{in}}, \longrightarrow, \mathrm{AP}\ L)$，其中$c_{\mathsf{in}} = (s_{\mathsf{in}}, \varepsilon)$是初始配置，$\longrightarrow \subseteq \mathsf{Conf}_\mathcal{P} \times (\{$调用、返回、内部操作$\} \times S \times (\Gamma \cup \{\varepsilon\}) \times S) \times \mathsf{Conf}_\mathcal{P}$ 是转移关系，而 L 是将 $\mathcal{P}$ 的标记函数扩展到配置上的标记函数，具体如下：$L(s, \alpha) = L(s)$。转移关系 $\longrightarrow$ 的定义旨在刻画内部、调用和返回转移的非形式化语义——对于任意 $\alpha \in \Gamma^*$，(内部操作) (s, α) (内部操作 $\xrightarrow{s,\varepsilon,s'}$) (s', α) 当且仅当 $(s, s') \in \Delta_{\mathsf{int}}$；(调用) (s, α) (调用 $\xrightarrow{s,a,s'}$) $(s', a\alpha)$ 当且仅当 $(s, (s', a)) \in \Delta_{\mathsf{call}}$；而 (返回) $(s, a\alpha)$ (返回 $\xrightarrow{s,a,s'}$) (s', α) 当且仅当 $((s, a)\ s') \in \Delta_{\mathsf{ret}}$。

一个 $[\![\mathcal{P}]\!]$ 的路径 是配置 $\sigma = c_0$、c_1、.....组成的无限序列，并且对于每个 i，都有 $c_i \xrightarrow{\mathsf{o}\ s,\ a,\ s'} c_{i+1}$，其中 $\mathsf{o} \in \{$内部操作、调用、返回$\}$、s、$s' \in S$ 以及 $a \in \Gamma \cup \{\varepsilon\}$。路径 σ 被称为从配置 c_0 （序列中的第一个配置）开始 。我们使用 $\mathsf{Paths}([\![\mathcal{P}]\!], c)$ 表示从配置 c 开始的 $[\![\mathcal{P}]\!]$ 的路径集，并使用 $\mathsf{Paths}([\![\mathcal{P}]\!])$ 表示 $[\![\mathcal{P}]\!]$的所有路径。

我们通过引入一些关于配置的记号来结束本节。对于$c = (s, \alpha)$，其 栈高 为 $|\alpha|$，其 控制状态 为 $\mathsf{state}(c) = s$，其 栈顶符号 为 $\mathsf{top}(c) = a \in \Gamma$（如果 $\alpha = a\alpha'$），而在 $\alpha = \varepsilon$ 时未定义。

3.2 sHCTL*语法

设原子命题集合为 AP，路径变量集合为 $\mathcal{V}$。sHCTL* 公式的 BNF 文法见图 1(b)。在该 BNF 文法中，$a \in$ AP 是一个原子命题，π 是一个路径变量，ψ 是一个同源公式，而 θ是一个 sHCTL* 公式。语法分为两个层次：内层与 HyperCTL* 公式相同，外层则允许对不同的栈访问模式进行量化（见第 3.3 节）。此外，按照 [5,10], ，我们假设直到算子 U 仅出现在路径量词的作用域内。

注 *1.* 为使本文的阐述更简洁，我们没有将 A（ E 的对偶）和合取作为显式逻辑算子。这一选择确实会降低本文所述自动机构造在处理涉及

合取的公式时的效率。将它们显式加入我们的框架不会带来技术上的困难，我们的自动机构造也可以扩展为显式处理它们。此外，我们有时会使用其他量词和逻辑算子来书写公式。一些标准示例包括： $\theta_1 \wedge \theta_2 = \neg(\neg\theta_1 \vee \neg\theta_2)$，其中 θ_i ($i \in \{1,2\}$) 是 sHCTL* 或同源公式； $\forall\pi.\psi = \neg\exists\pi.\neg\psi$； $\mathsf{F}\ \psi = \mathsf{true}\ \mathsf{U}\ \psi$，其中 $\mathsf{true} = a_\pi \vee \neg a_\pi$； $\mathsf{G}\,\psi = \neg\,\mathsf{F}\ \neg\psi$。

我们将形如 $\mathsf{q}\psi$ （其中 $\mathsf{q} \in \{A, E\}$ 且 ψ 是同源公式）的公式称为基本公式。注意，任何 sHCTL* 公式都是基本公式的布尔组合。若 sHCTL* 公式 θ 的每个基本子公式 $\mathsf{q}\psi$中， ψ 都是句子，也就是说， ψ 中出现的每个路径变量都已量化，则称该公式为句子 。不失一般性，我们假设在任何同源公式 ψ中， ψ 里的所有约束变量都已重命名，以确保每个路径变量最多只被量化一次。本文只讨论 sHCTL* 句子。sHLTL 逻辑是 sHCTL* 的子逻辑，由所有形如 q $Q_1\pi_1.\cdots Q_r\pi_r.\psi$的公式构成，其中 $\mathsf{q} \in \{A, E\}$， $Q_i \in \{\exists, \forall\}$ 且 ψ无量词。

3.3 sHCTL* 的语义

同源公式的语法与 HyperCTL* 公式的语法相同。它们的语义也将以类似的方式描述：在此语境中，公式里的自由路径变量被解释为系统的执行。不过，我们要求每个路径变量的解释共享一种共同的栈访问模式——这也是同源一词的由来。因此，在定义语义之前，我们将先定义路径的栈访问模式，以及为路径变量赋予解释的路径环境。

在本节余下部分中，固定一个下推系统 $\mathcal{P} = (S, \Gamma, s_{\mathsf{in}}, \Delta, L)$。如果字符串 $w \in \{\mathsf{call}, \mathsf{int}, \mathsf{ret}\}^*$ 满足以下条件之一，则称其为匹配良好： $w = \varepsilon$ ； 或者 $w = \mathsf{int}$; 或者 $w = \mathsf{call}\ u\,\mathsf{ret}$ $w = uv$，其中 $u, v \in \{\mathsf{call}, \mathsf{int}, \mathsf{ret}\}^*$ （递归地）匹配良好。在字符串 $\rho \in \{\mathsf{call}, \mathsf{int}, \mathsf{ret}\}^\omega$ 中， $\rho(i)$ 是一个未匹配返回，当且仅当$\rho[: i+1] = w\,\mathsf{ret}$，且 w 匹配良好。现在我们可以给出栈访问模式的定义了。

定义 1（栈访问模式）。如果字符串 $\rho \in \{$调用、内部操作、返回$\}^\omega$ 满足集合 $\{i \in \mathbb{N} \mid \rho(i)$ 有限，则称其为栈访问模式。

如果路径 $\sigma = c_0c_1c_2\cdots \in \mathit{Paths}(\llbracket\mathcal{P}\rrbracket)$ 具有栈访问模式 $\rho = o_0o_1\cdots$ （记作 $\mathit{pr}(\sigma) = \rho$），则称该路径具有栈访问模式，当且仅当对每个 i，均满足：**(a)** $o_i =$ 调用当且仅当 $\mathit{stack}(c_{i+1}) = \mathit{top}(c_{i+1})\,\mathit{stack}(c_i)$；**(b)** $o_i =$ 内部操作当且仅当 $\mathit{stack}(c_{i+1}) = \mathit{stack}(c_i)$；**(c)** $o_i =$ 返回当且仅当 $\mathit{stack}(c_i) = \mathit{top}(c_i)\,\mathit{stack}(c_{i+1})$。

现在我们给出路径环境的定义，它将同源公式中的自由路径变量解释为 $\llbracket\mathcal{P}\rrbracket$ 的路径，使它们共享同一个公共栈访问模式。这对于定义 sHCTL* 的语义至关重要。给定一组路径变量 $\mathcal{V}$，令 $\mathcal{V}^\dagger$ 定义为集合 $\mathcal{V} \cup \{\dagger\}$。

定义 2（路径环境）。针对变量 $\mathcal{V}$ 上的下推系 $\mathcal{P}$，路径环境是一个函数 Π：$\mathcal{V}^\dagger \to$ 路径集($\llbracket\mathcal{P}\rrbracket$) $\cup \{$调用、内部操作、返回$\}^\omega$，使得

对于每个 $\pi \in \mathcal{V}$， $\Pi(\dagger)$ 都是栈访问模式，并且 $\Pi(\pi) \in$路径集($[\![\mathcal{P}]\!]$)满足 $\mathsf{pr}(\Pi(\pi)) = \Pi(\dagger)$。当上下文中的下推系统明确时，我们将简称其为 $\mathcal{V}$ 上的路径环境。

当 $\mathcal{V} = \emptyset$ 时，我们还要求存在一条路径 $\sigma \in$ 路径集($[\![\mathcal{P}]\!]$, c_{in})（其中 c_{in} 是 $[\![\mathcal{P}]\!]$ 的初始配置），使得 $\mathsf{pr}(\sigma) = \Pi(\dagger)$。

我们引入一些与路径环境相关的记号。固定一个定义在变量 $\mathcal{V}$ 上的路径环境 Π。给定一条路径 $\sigma \in \mathrm{Paths}([\![\mathcal{P}]\!])$， $\Pi[\pi \mapsto \sigma]$ 表示定义在 $\mathcal{V} \cup \{\pi\}$ 上的路径环境，使得 $\Pi[\pi \mapsto \sigma](\pi) = \sigma$，且对任意满足 $\pi' \neq \pi$ 的 $\pi' \in \mathcal{V}^\dagger$， $\Pi[\pi \mapsto \sigma](\pi') = \Pi(\pi')$。最后，对于 $i \in \mathbb{N}$, $\Pi[i:\ ]$ 表示该后缀路径环境，其中每个变量都映射到从位置 i 开始的路径后缀。更形式化地，对每个 $\pi' \in \mathcal{V}^\dagger$, $\Pi[i:\ ](\pi') = \Pi(\pi')[i:\ ]$。

现在，我们定义下推系统 $\mathcal{P}$ 何时满足 sHCTL* 句子 θ，记作 $\mathcal{P} \models \theta$。 θ 的满足关系定义依赖于同源公式的满足关系定义。为了归纳地定义同源公式的语义，我们将使用路径环境来解释自由路径变量。最后，与 HyperCTL* 中一样，追踪最近量化的路径变量十分重要，因为它会影响 $\exists\pi(\cdot)$ 的语义。因此，同源公式的满足关系形式为 $\mathcal{P}, \Pi, \pi' \models \psi$，其中 π' 是最近量化的路径变量，而 Π 是定义在 ψ自由变量上的路径环境。最后，按照约定，我们将 $\mathrm{Paths}([\![\mathcal{P}]\!], \Pi(\dagger)(0))$ 视为 $\mathrm{Paths}([\![\mathcal{P}]\!], c_{\mathsf{in}})$，其中 c_{in} 是 $[\![\mathcal{P}]\!]$ [5]的初始配置。下文中， θ, θ_1和 θ_2 是 sHCTL* 句子，而 ψ, ψ_1, ψ_2 是同源公式。

$\mathcal{P} \models \neg\theta$ 当且仅当 $\mathcal{P} \not\models \theta\mathcal{P} \models \theta_1 \vee \theta_2$ 当且仅当 $\mathcal{P} \models \theta_1$ 或 $\mathcal{P} \models \theta_2$ $\mathcal{P} \models E\psi$ 当且仅当对于某个路径环境 Π，其定义在 $\emptyset, \mathcal{P}, \Pi$, $\dagger \models \psi\mathcal{P}, \Pi, \pi' \models a_\pi$ 当且仅当 $a \in L(\Pi(\pi)(0))\mathcal{P}, \Pi, \pi' \models \neg\psi$ 当且仅当 $\mathcal{P}, \Pi, \pi' \not\models \psi\mathcal{P}, \Pi, \pi' \models \psi_1 \vee \psi_2$ 当且仅当 $\mathcal{P}, \Pi, \pi' \models \psi_1$ 或 $\mathcal{P}$, $\Pi, \pi' \models \psi_2\mathcal{P}, \Pi, \pi' \models \mathrm{X}\psi$ 当且仅当 $\mathcal{P}, \Pi[1:\ ], \pi' \models \psi\mathcal{P}, \Pi$, $\pi' \models \psi_1 \,\mathrm{U}\, \psi_2$ 当且仅当 $\exists i \geq 0:\ \mathcal{P}, \Pi[i:\ ], \pi' \models \psi_2$ 以及 $\forall j$, $0 \leq j < i$,$\mathcal{P}, \Pi[j:\ ], \pi' \models \psi_1\mathcal{P}, \Pi, \pi' \models \exists\pi.\, \psi$ 当且仅当 $\exists\sigma \in$路径集($[\![\mathcal{P}]\!], \Pi(\pi')(0)$)，其中 $\mathsf{pr}(\sigma) = \Pi(\dagger)$，使得 $\mathcal{P}, \Pi[\pi \mapsto \sigma], \pi \models \psi$

4 sHCTL*的判定过程

给定一个下推系统 $\mathcal{P}$ 和一个 sHCTL* 句子 θ，我们提出一种算法，用于判定 $\mathcal{P} \models \theta$是否成立。我们的方法与 [10]中的方法类似。给定一个有限状态转移系统 $\mathcal{K}$ 和一个 HyperCTL*公式 φ, Finkbeiner 等人 [10],通过对φ进行归纳，构造一个交替（有限状态）Büchi 自动机 $\mathcal{A}_{\mathcal{K},\ \varphi}$, 使得输入词 σ 被$\mathcal{A}_{\mathcal{K},\ \varphi}$ 接受，当且仅当 σ 是某个路径环境的编码

[5] 之所以需要这一约定，是因为 $\Pi(\dagger)(0)$ 不是一个配置，而是集合 {调用、内部操作、返回} 的一个元素。

Π，满足 $\mathcal{K}, \Pi \models \varphi$。因此，判定 $\mathcal{K} \models \varphi$ 是否成立可归结为检查 $\mathcal{A}_{\mathcal{K},\ \varphi}$ 是否接受任意字符串。

将这些思想扩展到 sHCTL* 和下推系统，需要回答两个问题：(a) 对于路径变量映射到配置序列（控制状态 $+$ 栈）的同源公式，路径环境应如何编码？；(b) 哪些自动机模型能够刻画相对于某个下推系统、满足同源公式的路径环境集合？我们使用下推字母表上的字符串来编码同源公式的路径环境——符号上的下推标签增添了一种结构，有助于编码配置序列。至于自动机，我们考虑处理此类字符串并接受可见下推语言的自动机。对 [10] 中概述的方法进行自然推广，可能会让人想到在无限字符串 [4] 上使用交替可见下推自动机 (AVPA)。然而，使用 AVPA 会导致算法效率低下。为获得更高效的算法，我们转而谨慎地结合使用非确定性可见下推自动机 (NVPA) [1] 和单向交替跳转自动机 (1-AJA) [4]。使用 NVPA 和 1-AJA 的优势可从存在量化 ($\exists \pi$.) 的情形中看出；存在量化需要将交替自动机转换为非确定性自动机 [10]：从 1-AJA 转换为 NVPA 会导致指数级膨胀，而从 AVPA 转换为 NVPA 则会导致双指数级膨胀 [4]。

本节其余部分安排如下。我们首先介绍下推字母表上的自动机模型（第 4.1 节）。接下来，我们介绍路径环境的编码，最后介绍用于确立可判定性结果的自动机构造（第 4.2 节）。

4.1 下推字母表上的自动机

一个下推字母表是一个有限集合 Σ，它被划分为三个集合$\Sigma_{\mathsf{call}} \uplus \Sigma_{\mathsf{int}} \uplus \Sigma_{\mathsf{ret}}$，其中 Σ_{call} 是调用符号的集合，Σ_{int} 是内部操作符号的集合，而 Σ_{ret} 是返回符号的集合。在下推字母表上处理字符串的自动机模型，会根据读取的符号是调用符号、内部操作符号还是返回符号，限制其执行特定类型的转移。接下来，我们将非正式地介绍两种此类自动机模型。其精确定义及语义见附录 B 和附录 C。

非确定性可见下推 Büchi 自动机。A非确定性可见下推自动机（*NVPA*） [1] 类似于下推系统。它具有有限个控制状态，并使用一个无界栈进行存储。然而，与下推系统不同，它是一种自动机，处理来自下推字母表 $\Sigma = \Sigma_{\mathsf{call}} \uplus \Sigma_{\mathsf{int}} \uplus \Sigma_{\mathsf{ret}}$ 的无限输入符号序列。转移受到约束，必须符合下推字母表的规定——每当读取 Σ_{call} 符号时，就将一个符号压入栈中；每当读取 Σ_{ret} 符号时，就弹出栈顶符号；每当读取 Σ_{int} 符号时，栈保持不变。

单向交替跳转自动机。 我们的第二种自动机模型是 单向交替优先级跳转自动机（*1-AJA*） [4]。1-AJA 在计算能力上等价于 NVPA（即接受同一类语言），但能

更灵活地描述算法。1-AJA 是交替自动机，这意味着它们可以根据自动机在输入词上的多次运行来定义接受条件。虽然它们是没有辅助存储的有限状态机，但它们读取一个符号时能够生成一条计算线程，并使其跳转到输入字符串中后续的位置，因此拥有与更传统的带存储机器（如 NVPA）相同的计算能力。

我们介绍了非确定性可见下推自动机和 1-AJA 的一些有用性质。这两种模型具有相同的表达能力，且已知转换所构造自动机的大小。

定理 1 ([4])。对于任意大小为 n 的非确定性可见下推自动机 N，都存在一个大小为 $O(n^2)$ 的 *1-AJA* $\mathcal{A}_N$，使得 $\mathcal{L}(\mathcal{A}_N) = \mathcal{L}(N)$。反过来，对于任意大小为 n 的 *1-AJA* $\mathcal{A}$，都存在一个大小为 $2^{O(n)}$ 的非确定性可见下推自动机 $N_{\mathcal{A}}$，使得 $\mathcal{L}(N_{\mathcal{A}}) = \mathcal{L}(\mathcal{A})$。构造所需时间与所得自动机的大小成正比。

1-AJA 和非确定性可见下推自动机在语言运算下均封闭，例如补运算、并运算和前缀运算。我们还回顾以下结果。

定理 2。*([1])* 对于非确定性可见下推自动机，空性问题是多项式时间完全的。

4.2 sHCTL* 的算法

固定一个下推系统 $\mathcal{P} = (S, \Gamma, s_{\text{in}}, \Delta, L)$ 和一个 sHCTL* 句子 θ。我们的目标是判定 $\mathcal{P} \models \theta$ 是否成立。我们将把这个问题归约为检查多个非确定性可见下推自动机的空性（定理 2）。我们的方法与 [10] 类似——对于 θ 的每个相关子公式 ψ（不一定是句子），我们将组合地构造一个自动机，使其接受满足 ψ 的路径环境。路径环境将按如下方式编码为下推字母表上的字符串。

对于 $[\![\mathcal{P}]\!]$ 的一条路径 $\sigma = c_0 c_1 c_2 \cdots$，$\sigma$ 的轨迹 记为 tr(σ)，它是如下（唯一的）序列：$(\mathsf{o}_0, q_0, a_0, q_1)(\mathsf{o}_1, q_1, a_1, q_2)\cdots$，满足对每个 $i \in \mathbb{N}$，$c_i \xrightarrow{(\mathsf{o}_i, q_i, a_i, q_{i+1})} c_{i+1}$，其中 $\mathsf{o}_i \in$ {调用、内部操作、返回}，$q_i, q_{i+1} \in Q$，且 $a_i \in \Gamma \cup \{\varepsilon\}$ [6]。

虽然 tr(σ) 可由路径 σ 唯一确定，但反过来并不成立——不同路径可能具有相同的轨迹。为说明这一点，请看下面的例子。对于配置 c 和 $\gamma \in \Gamma^*$，令 $\gamma(c)$ 表示配置 (state(c),stack(c)γ)，即控制状态相同，但栈底包含 γ 中符号的配置。注意，对于任意 $\gamma \in \Gamma^*$，如果 $\sigma = c_0 c_1 c_2 \cdot$ 是一条路径，那么 $\gamma(\sigma) = \gamma(c_0)\gamma(c_1)\gamma(c_2)\cdots$ 也是一条路径。此外，tr(σ) =tr($\gamma(\sigma)$)。如果 $[\![\mathcal{P}]\!]$ 中的两条路径 σ_1 和 σ_2 满足 tr(σ_1) =tr(σ_2)，则称它们等价，记作 $\sigma_1 \simeq \sigma_2$。注意，等价路径具有相同的栈访问模式，即若 $\sigma_1 \simeq \sigma_2$，则 pr(σ_1) = pr(σ_2)。sHCTL* 的语义无法区分等价路径。

[6]请注意，即使 σ 不是 $[\![\mathcal{P}]\!]$ 中的一条路径（即不对应于 $\mathcal{P}$的一串实际转移），只要能够通过保持栈不变，或压入/弹出一个符号，来得到 σ 中连续配置的栈，σ 的轨迹就能被唯一确定。

命题1。设 φ 是一个同源公式，其自由路径变量集合为 $\mathcal{V}$。设 Π_1 和 Π_2 是两个路径环境，并且对于每个 $\pi \in \mathcal{V}$，$\Pi_1(\pi) \simeq \Pi_2(\pi)$。那么，$\mathcal{P}, \Pi_1, \pi \models \varphi$ 当且仅当 $\mathcal{P}, \Pi_2, \pi \models \varphi$。

命题1的证明通过对同源公式进行归纳得出。命题1说明，满足某个同源公式的路径环境集合，是关于路径等价的等价类的并运算。因此，我们不再构造接受路径环境的自动机，而是构造接受从路径变量到路径轨迹的映射的自动机。对于 $m \in \mathbb{N}$，令 $\Sigma[m] = \Sigma[m]_{\mathsf{call}} \uplus \Sigma[m]_{\mathsf{int}} \uplus \Sigma[m]_{\mathsf{ret}}$ 为下推字母表，其中 $\Sigma[m]_{\mathsf{call}} = \{\mathsf{call}\} \times S^m \times \Gamma^m$、$\Sigma[m]_{\mathsf{int}} = \{\text{内部操作}\} \times S^m \times \{\varepsilon\}^m$ 和 $\Sigma[m]_{\mathsf{ret}} = \{\text{返回}\} \times S^m \times \Gamma^m$。注意，$\Sigma[0]$（本质上）是集合 {内部操作、调用、返回}。

定义3（路径环境编码）。考虑路径变量集合 m ***path variables*** $\mathcal{V} = \{\pi_1, \pi_2, \ldots \pi_m\}$。字符串 $w \in \Sigma[m]^\omega$，其中对于任意 $j \in \mathbb{N}$, $w(j) = (\mathsf{o}j, (s_1^j, s_2^j, \ldots s_m^j), (a_1^j, a_2^j, \ldots a_m^j)$ 编码了所有满足以下条件的路径环境 Π

$$\Pi(\dagger) = \mathsf{o}_0\mathsf{o}_1\mathsf{o}_2 \cdots \mathsf{o}_j \cdots$$
$$\mathsf{tr}(\Pi(\pi_i)) = (\mathsf{o}_0, s_i^0, a_i^0, s_i^1)(\mathsf{o}_1, s_i^1, a_i^1, s_i^2) \cdots$$

对于任意 $i \in \{1, 2, \ldots m\}$。路径环境 Π 的编码字符串记为$\mathsf{enc}(\Pi)$（本例中为 $= w$）。

根据这些定义，可以得出关于轨迹和编码的以下观察结果。

命题 2. 对于任意路径 $\sigma \in$ 路径集($[\![\mathcal{P}]\!]$) 和 $i \in \mathbb{N}$，$\mathsf{tr}(\sigma[i:\ ]) = \mathsf{tr}(\sigma)[i:\ ]$。对于任意路径环境 Π 和 $i \in \mathbb{N}$，$\mathsf{enc}(\Pi[i:\ ]) = \mathsf{enc}(\Pi)[i:\ ]$。

将路径环境编码为以 $\Sigma[m]$ 上的字符串（取适当的 m）后，可用于我们的判定过程；该过程以组合方式构造自动机，使其接受满足每个同源公式的路径环境。与 [10], 中的情况类似，我们所构造的自动机的大小将是一个依赖于同源公式公式复杂度 的指数塔 φ。

定义 *4*（公式复杂度）。***sHCTL****公式 φ 的公式复杂度记为 ***fc(φ)***，其归纳定义如下。设 ***odd :*** $\mathbb{N} \to \mathbb{N}$ 为函数，它将一个数 n 映射为大于等于它的最小奇数 $\geq n$，即当 n 为奇数时，***odd(n)*** $= n$；当 n 为偶数时，***odd(n)*** $= n + 1$。类似地，***even :*** $\mathbb{N} \to \mathbb{N}$ 将 n 映射为大于等于它的最小偶数 $\geq n$，即 ***even(n)*** $=$ ***odd(***$n + 1$***)*** $- 1$。下文中，ψ_1、ψ_2 表示同源公式，θ_1、θ_2 表示 ***sHCTL**** 句子。

$$\mathsf{fc}(a_\pi) = 0 \qquad \mathsf{fc}(\neg\psi_1) = \mathsf{even}(\mathsf{fc}(\psi_1)) \qquad \mathsf{fc}(\mathsf{X}\psi_1) = \mathsf{fc}(\psi_1)$$
$$\mathsf{fc}(\psi_1 \vee \psi_2) = \max(\mathsf{fc}(\psi_1), \mathsf{fc}(\psi_2)) \qquad \mathsf{fc}(\psi_1 \,\mathsf{U}\, \psi_2) = \mathsf{even}(\max(\mathsf{fc}(\psi_1), \mathsf{fc}(\psi_2)))$$
$$\mathsf{fc}(\exists\pi.\, \psi_1) = \mathsf{odd}(\mathsf{fc}(\psi_1)) \qquad \mathsf{fc}(E\psi_1) = \mathsf{odd}(\mathsf{fc}(\psi_1))$$
$$\mathsf{fc}(\neg\theta_1) = \mathsf{fc}(\theta_1) \qquad \mathsf{fc}(\theta_1 \vee \theta_2) = \max(\mathsf{fc}(\theta_1), \mathsf{fc}(\theta_2))$$

注意 $\mathsf{fc}(\neg\theta_1)$ 和 $\mathsf{fc}(\neg\psi_1)$ 的定义差异；对于 $\neg\theta_1$，公式复杂度没有变化，而对于 $\neg\psi_1$，我们会转到下一个偶数层级。

我们的主要技术引理是为同源公式 ψ 构造自动机的一种组合式方法。根据 fc(ψ) 的优先级，该自动机要么是 1-AJA，要么是 NVPA。在陈述这个引理之前，我们先定义一个取值为指数塔的函数。对于 c、k、$n \in \mathbb{N}$，值 $g_c(k,n)$ 按照 k 归纳定义如下：$g_c(0,n) = cn\log n$，且 $g_c(k+1,n) = 2^{g_c(k,\ n)}$。我们用 $g_{O(1)}(k,n)$ 表示函数族 $\{g_c(k,n) \mid c \in \mathbb{N}\}$。

引理 1. *考虑下推系统 $\mathcal{P} = (S, \Gamma, s_{\mathsf{in}}, \Delta, L)$ 以及 sHCTL* 句子 θ。设 ψ 是 θ 的一个同源子公式，其自由路径变量属于集合 $\mathcal{V} = \{\pi_1, \dots \pi_m\}$，对应于 $m \in \mathbb{N}$。不失一般性地，我们假设变量 $\pi_1, \dots \pi_m$ 按照它们在 θ 中被量化的顺序排列，其中 π_m 是 ψ 中第一个将在上下文 θ 中被量化的自由变量。此外，我们假设 ψ 和 $\mathcal{P}$ 的大小均有 n 上界。存在一个在下推字母表 $\Sigma[m]$ 上的自动机 $\mathcal{A}_\psi$，使得对于 $\mathcal{V}$ 上的任意路径环境 Π，*

$$\mathcal{P}, \Pi, \pi_m \models \psi \ \textit{if and only if}\ \mathsf{enc}(\Pi) \in \mathcal{L}(\mathcal{A}_\psi).\ ^{7}$$

*若 $\mathsf{fc}(\psi)$ 为奇数，则自动机 $\mathcal{A}_\psi$ 是 **NVPA**；若 $\mathsf{fc}(\psi)$ 为偶数，则它是 **1-AJA**。$\mathcal{A}_\psi$ 的大小至多为 $g_{O(1)}(\lceil \frac{\mathsf{fc}(\psi)}{2} \rceil, n)$[8]。*

在给出引理 1 的证明之前，我们想指出其陈述中的一个微妙之处。该结果保证，对于有效的路径环境 Π，当且仅当 Π 满足 ψ 时，$\mathcal{A}_\psi$ 会接受编码 enc(Π)。它并未说明无效路径环境的情况。特别是，某些将路径变量映射到轨迹的函数，所映射的轨迹并不对应 $[\![\mathcal{P}]\!]$ 的实际路径，但仍会被 $\mathcal{A}_\psi$ 接受。不过请注意，当 $\psi = \exists\pi.\,\psi_1$ 是一个同源句子时，根据引理 1 所保证的条件，{call,int,ret} 上的字符串会被接受，当且仅当它对应于一条从初始状态出发且满足 $\exists\pi.\,\psi_1$ 的路径的栈访问模式。

*证明（引理 1 的概要）。*我们将以归纳方式构造 $\mathcal{A}_\psi$。所构造的自动机类型将与 fc(ψ) 的奇偶性一致：如果 fc(φ) 为奇数，则构造 NVPA；如果 fc(ψ) 为偶数，则构造 1-AJA。这里概述主要思路，完整证明见附录 D。

对于 a_π、$\neg\psi_1$、$\psi_1 \vee \psi_2$ 和 Xψ_1，该构造的基本做法是：必要时，利用定理 1 将 $\mathcal{A}_{\psi_i}$ ($i \in \{1,2\}$) 转换为目标自动机的类型（NVPA 或 1-AJA），然后利用标准闭包性质将它们组合起来，得到所需的自动机。对于 $\psi = \psi_1 \,\mathsf{U}\, \psi_2$，我们首先在必要时将 $\mathcal{A}_{\psi_i}$ ($i \in \{1,2\}$) 转换为 1-AJA。在每一步中，ψ 的自动机都会选择运行 $\mathcal{A}_{\psi_2}$，或运行 $\mathcal{A}_{\psi_1}$ 并重新启动自身。正确性依赖于这样一个事实：我们对路径环境的编码满足命题 2。

最有趣的情形是 $\psi = \exists\pi.\,\psi_1$。我们首先将 ψ_1 的自动机（如有必要）转换为 NVPA $\mathcal{A}_1$。ψ 的自动机基本上会猜测一条路径的编码，该路径与以下对象的转移保持一致：

[7] 当 $m = 0$ 时，我们令 π_m 为 †。[8] 当规约 ψ 的大小被视为常数时，$\mathcal{A}_\psi$ 的大小至多为$g_{O(1)}(\lceil \frac{\mathsf{fc}(\psi)}{2} \rceil - 1, n)$

$\mathcal{P}$，并检查将猜测的路径赋给变量 π 是否满足 ψ_1，方法是运行自动机 $\mathcal{A}_1$。我们还有一个额外要求：猜测的路径必须从与当前配置相同的配置开始，而该当前配置属于赋给变量 π_m 的路径。这带来了一些微妙的挑战。为了能够猜测一条路径，$\mathcal{A}_\psi$ 会在自己的控制状态中跟踪 $\mathcal{P}$ 的控制状态，并使用自己的栈跟踪猜测路径上 $\mathcal{P}$ 的栈操作。由于所有路径的栈都是同步的，因此 $\mathcal{A}_\psi$ 可以用它的（单个）栈同时跟踪 $\mathcal{P}$ 的栈和 $\mathcal{A}_1$ 的栈。□

利用引理 1，我们可以得出本节的主要结果。

定理 3. 给定一个 $\mathcal{P} = (S, \Gamma, s_{\text{in}}, \Delta, L)$ 和一个sHCTL* 句子 θ，判断 $\mathcal{P} \models \theta$ 是否属于 $\cup_c$确定性时间复杂度$(g_c(\lceil \frac{\text{fc}(\theta)}{2} \rceil, n))$的问题，其中 n 是 $\mathcal{P}$ 和 θ大小的上界。

证明。回顾一下，sHCTL* 句子是由形如 $E\psi$ 的公式进行布尔组合得到的，其中 ψ 是一个同源句子。可以结合关于每个此类子公式是否满足 $\mathcal{P} \models E\psi$ 的结果，以确定 $\mathcal{P} \models \theta$ 是否成立。由此，判断 $\mathcal{P} \models \theta$ 是否成立所需的时间，至多为判定 $\mathcal{P}$ 是否满足每个形如 $E\psi$ 的子公式所需的时间，加上 $O(n)$（用于计算这些结果的布尔组合）。接下来回顾一下，引理 1 中的构造确保，对于形如 $\exists \pi.\, \psi$ 的同源句子，$\mathcal{L}(\mathcal{A}_{\exists \pi.\, \psi})$ 恰好由{call、int、ret}$^\omega$ 中的字符串组成，这些字符串编码了满足 $\exists \pi.\, \psi$ 的、关于 $\emptyset$ 的路径环境。

考虑一个 sHCTL* 句子 $E\psi$。设 π 是一个不出现在句子 ψ 中的路径变量。根据 sHCTL* 的语义，有如下观察结果：$\mathcal{P} \models E\psi$ 当且仅当存在某个路径环境 Π，其定义在 $\emptyset$、$\mathcal{P}$、Π、$\dagger \models \exists \pi.\, \psi$ 上。也就是说，$\mathcal{P} \models E\psi$ 当且仅当 $\mathcal{L}(\mathcal{A}_{\exists \pi.\, \psi}) \neq \emptyset$。由于 $\text{fc}(E\psi) = \text{fc}(\exists \pi.\, \psi)$，且 NVPA 的空性问题可以在多项式时间内判定（定理 2），因此定理得证。□

5 下界

本节将建立针对下推系统上 sHCTL* 句子模型检测问题的下界。我们的证明将为 sHCTL* 的 sHLTL 子片段建立一个困难性结果。在给出该下界之前，我们先介绍函数 $h_c(\cdot, \cdot)$。该函数是另一个指数塔，其归纳定义如下：$h_c(0, n) = n$，且 $h_c(k+1, n) = h_c(k, n) \cdot c^{h_c(k,\ n)}$。

定理 4. 设 $\mathcal{P}$ 是一个下推系统，θ 是一个 sHLTL 句子，且 $\mathcal{P}$ 和 θ 的大小均受 n 和 $\text{fc}(\theta) = 2k - 1$ 的约束，其中 $k \in \mathbb{N}$为某个值。对于每个 $c \in \mathbb{N}$，检查 $\mathcal{P} \models \theta$ 是否为确定性时间复杂度$(h_c(k, n))$-困难的问题。

证明（概要）。我们概述证明背后的主要思路。为了突出该证明的新颖之处，回顾一下如何证明 HyperLTL 模型检测的 NSPACE($h_c(k-1, n)$) 困难性 [5]会有所帮助。其思路是将非确定性 $h_c(k-1, n)$ 空间有界机器 M的语言归约到模型检测

问题：构造一个猜测M运行的有限状态转移系统，以及一个 HyperLTL公式，用来检查该路径是否为有效的接受运行。

为了得到更严格的确定性时间复杂度界 DTIME($h_c(k,n)$)，我们利用正在检查下推系统这一事实。下推系统的栈可用于猜测一棵树，而不只是简单的轨迹。因此，我们将归约一个空间受限于 $h_c(k-1,n)$ 的交替图灵机，而非非确定性机器。由于对于 $f(n) \geq \log n$，ASPACE($f(n)$) = DTIME($2^{O(f(n))}$)，只要归约成功，定理便得证。

回想一下，交替图灵机 M 的运行是一棵有根标记树，其中顶点按照与 M 的转移函数一致的方式标记为 M 的配置。为了忠实地将树编码为符号序列，我们记录这棵树的深度优先搜索遍历，并明确写出遍历过程中执行的栈操作。考虑一棵有根标记树 T，其根为 r，标签为 $\ell(r)$，T_1 是左子树，T_2 是右子树。对 T 进行深度优先搜索遍历时，将压入 $\ell(r)$，递归遍历 T_1，弹出 $\ell(r)$，压入 $\ell(r)$，遍历 T_2，然后弹出 $\ell(r)$。我们将利用这种深度优先搜索遍历来猜测并编码 M 的运行。在遍历 T_1 和 T_2 之间弹出并压入 $\ell(r)$ 似乎是多余的。为什么不在遍历 T_1 和 T_2 之间什么也不做呢？要使 T 成为 M 的有效运行，T_2 根的配置标签必须是对 $\ell(r)$ 执行一步转移后的结果。我们会在 sHLTL句子中编码这些检查；要做到这一点，M 的相邻配置必须在字符串编码中相邻出现。

为了突出一些额外的一致性检查，我们继续使用上一段中的示例树 T。要使一个字符串成为 T 的正确编码，必须满足：在遍历 $T_i (i \in \{1,2\})$ 之前压入的字符串，与遍历之后弹出的字符串相同。下推系统可以通过实际压入和弹出这些符号来确保这一点。此外，在遍历 T_1 之后弹出的字符串，必须与遍历 T_2 之前压入的字符串相同。下推系统的栈和有限控制都无法用于确保这一点。因此，必须由我们构造的 sHLTL 句子来检查这一点。不过，弹出 $\ell(r)$ 时的符号顺序，与压入时的符号顺序相反，因此很难在公式中执行这项检查。为了解决这个问题，我们同时压入/弹出标签 及其反向形式 。这样，如果我们想检查压入的字符串是否与刚刚弹出的字符串相同，就可以检查字符串 是否相等；在 sHLTL 公式中进行这项检查更容易。sHLTL 句子还会利用 [17]中的思路，执行额外检查以确保该树编码了一个有效的接受运行。完整细节见附录E。 □

6 结论

本文介绍了一种分支时间时序逻辑 sHCTL*，可用于规约建模为下推系统的递归程序的同步超属性。它与用于描述同步超属性的标准分支时序逻辑 HyperCTL* 的主要区别在于，sHCTL* 考虑

一类受限的超属性，即只关联具有相同栈访问模式的执行。我们称这类超属性为栈感知超属性。我们证明了下推系统的 sHCTL* 规范模型检测问题是可判定的，并刻画了其复杂度。我们还展示了这一结果如何有望用于辅助安全验证。

参考文献

1. Alur, R., Madhusudan, P.：可见下推语言。载于：第36届 ACM 计算理论年度研讨会论文集。第202–211页。ACM（2004年） 2. Bajwa, A., Zhang, M., Chadha, R., Viswanathan, M.：栈感知超性质。载于：第29届系统构造与分析工具和算法国际会议（TACAS）（2023年），即将发表 3. Bouajjani, A., Esparza, J., Maler, O.：下推自动机的可达性分析：在模型检测中的应用。载于：并发理论，第八届国际会议。第135–150页。施普林格（1997年） 4. Bozzelli, L.：用于可见下推语言的交替自动机与时序不动点演算。载于：并发理论，第18届国际会议。第476–491页。施普林格（2007年） 5. Clarkson, M.R., Finkbeiner, B., Koleini, M., Micinski, K.K., Rabe, M.N., Sánchez, C.：超属性的时序逻辑。载于：安全与信任原理——第三届国际会议。第265–284页。施普林格（2014年） 6. Clarkson, M.R., Schneider, F.B.：超属性。载于：第21届 IEEE 计算机安全基础研讨会论文集。第51–65页。IEEE 计算机学会（2008年） 7. Coenen, N., Finkbeiner, B., Sánchez, C., Tentrup, L.：超活性验证。载于：计算机辅助验证——第31届国际会议。第121–139页。施普林格（2019年） 8. Finkbeiner, B., Hahn, C., Stenger, M.：EAHyper：超属性的可满足性、蕴含与等价性检查。载于：计算机辅助验证——第29届国际会议。第564–570页。施普林格（2017年） 9. Finkbeiner, B., Hahn, C., Stenger, M., Tentrup, L.：RVHyper：用于时序超属性的运行时验证工具。载于：系统构造与分析工具和算法——第24届国际会议。第194–200页。施普林格（2018年） 10. Finkbeiner, B., Rabe, M.N., Sánchez, C.：HyperLTL和HyperCTL*模型检测算法。载于：计算机辅助验证——第27届国际会议。第30–48页。施普林格（2015年） 11. Goguen, J.A., Meseguer, J.：安全策略与安全模型。载于：IEEE 安全与隐私研讨会。第11–20页。IEEE 计算机学会（1982年） 12. Gutsfeld, J.O., Müller-Olm, M., Ohrem, C.：递归程序异步超属性的判定。CoRR **abs/2201.12859**（2022年） 13. McLean, J.：使用轨迹证明非干扰性与功能正确性。《计算机安全期刊》**1**(1)，37–58（1992年） 14. McLean, J.：关于在选择性交错函数下封闭的轨迹集的组合一般理论。载于：IEEE 计算机学会安全与隐私研究研讨会。第79–93页。IEEE 计算机学会（1994年）

15. Molnar, D., Piotrowski, M., Schultz, D., Wagner, D.：程序计数器安全模型：控制流侧信道攻击的自动检测与消除。载于：第八届国际信息安全与密码学会议论文集。第156–168页。施普林格出版社（2005年） 16. Pommellet, A., Touili, T.：下推系统的HyperLTL模型检测。载于：软件模型检测——第二十五届国际研讨会。第133–152页。施普林格（2018年） 17. Sistla, A.P., Vardi, M.Y., Wolper, P.：Büchi自动机的补运算问题及其在时序逻辑中的应用。《理论计算机科学》**49**，217–237（1987年） 18. Walukiewicz, I.：下推过程：博弈与模型检测。载于：计算机辅助验证，第八届国际会议。第62–74页。施普林格（1996年） 19. Zdancewic, S., Myers, A.C.：并发程序安全的观察确定性。载于：第16届 IEEE 计算机安全基础研讨会论文集。第29页。IEEE 计算机学会（2003年）

A 当调用栈大小对攻击者可见时的观察确定性

观察确定性 [13,19] 指出，任何两个具有相同低安全级初始输入的执行，都必须具有相同的低安全级输出观测。观察确定性是一种超安全属性 [5,6]。正如 [5] 所示，观察确定性也可以用 HyperLTL 公式表示：$\mathrm{OD}\ \mathrm{def}\ =\ \forall\pi.\forall\pi'.(\pi[0] \equiv_{L,in} \pi'[0]) \rightarrow \pi \equiv_{L,out} \pi'.$ 这里 $\equiv_{L,in}$ 和 $\equiv_{L,out}$ 分别表示 π 和 π' 具有相同的低安全级输入和输出。

为了了解当调用栈大小可观察时如何验证观察确定性，我们来考虑一个简单情形：调用栈大小是唯一的低安全级别观察。设 $\mathcal{P}$ 是与某个程序对应的下推自动机。不失一般性，我们可以假设 $\mathcal{P}$ 的状态编码了栈是否为空，并且出栈转移只会发生在栈为空的状态中。[9] 设 $\Delta\mathcal{P}$ 是从 $\mathcal{P}$ 得到的下推自动机，其行为与 $\mathcal{P}$ 相同，只是额外加入了如下非确定性转移。ΔP 有三个新状态：q_{call}、q_{int} 和 q_{ret}. 。我们还假设有四个新命题：new、call、int 和 ret。命题 new 仅在 q_{call}、q_{int} 和 q_{ret} 中为真。新状态中的唯一转移是不改变栈大小的自循环。每当 $\mathcal{P}$ 在 $\mathcal{P}$ 中存在从状态 q 到 q' 的压栈/内部操作/弹栈转移时，我们就添加一条从 q 到 $q_{\mathsf{call}}/q_{\mathsf{int}}/q_{\mathsf{ret}}$ 的内部非确定性转移，且该转移不改变栈大小。此外，在这种情况下，命题 call/int/ret 为

[9] 本质上，初始时栈被视为空。当一个符号被压入空栈时，它会被“标注”为栈底，而状态会记住栈非空。当一个“标注”符号被弹出时，栈便记住栈已为空。

在 q 中为真。考虑公式

$$
\begin{aligned}
A\,\forall\pi.\forall\pi.'(\pi[0] \equiv_{L,in} \pi'[0]) \rightarrow \\
((\mathsf{new}_{\pi} \vee \mathsf{new}_{\pi'})\,\mathsf{R} \\
((\mathsf{call}_{\pi} \Rightarrow \mathsf{call}_{\pi'} \wedge \neg\mathsf{int}_{\pi'} \wedge \neg\mathsf{ret}_{\pi'}) \wedge \\
(\mathsf{int}_{\pi} \Rightarrow \mathsf{int}_{\pi'} \wedge \neg\mathsf{call}_{\pi'} \wedge \neg\mathsf{ret}_{\pi'}) \wedge \\
(\mathsf{ret}_{\pi} \Rightarrow \mathsf{ret}_{\pi'} \wedge \neg\mathsf{call}_{\pi'} \wedge \neg\mathsf{int}_{\pi'})))
\end{aligned}
$$

其中，R 是 U 的对偶。更确切地说，$\psi\,\mathrm{R}\,\psi'$ 等价于 $\neg(\neg\psi')\,\mathrm{U}(\neg\psi)$。

注意，只有当存在 $\mathcal{P}$ 的两次有限执行 σ_1 和 σ_2，分别到达状态q_1 和 q_2，并且满足以下条件时，上述公式才不被下推系统 $\Delta\mathcal{P}$满足：(a) σ_1 和 σ_2 中的低级输入相同；(b) σ_1和 σ_2 具有相同的栈访问模式；(c) 可以从状态 q_1 和 q_2中的一个执行压栈/内部操作/弹栈，但不能从另一个执行。

B 非确定性可见下推自动机（NVPA）

一个非确定性可见下推自动机（*NVPA*） [1] 与下推系统类似，具有有限多个控制状态，并使用无界栈进行存储。然而，与下推系统不同，它是一种自动机，能够处理来自下推字母表 $\Sigma = \Sigma_{\mathsf{call}} \uplus \Sigma_{\mathsf{int}} \uplus \Sigma_{\mathsf{ret}}$ 的无限输入符号序列。此外，转移受到下推字母表的约束——每当自动机读取 Σ_{call} 中的符号时，就会将一个符号压入栈中；每当读取 Σ_{ret} 中的符号时，就会弹出栈顶符号；每当读取 Σ_{int} 中的符号时，栈保持不变。

定义 *5*（*NVPA*）。非确定性可见下推 *Büchi* 自动机（*NVPA*）是一个元组 $N = (Q, q_{\mathsf{in}}, \Sigma, \Gamma, \bot, \delta, Q_F)$，其中，$Q$ 是有限控制状态集，$q_{\mathsf{in}} \in Q$ 是初始控制状态，$\Sigma = \Sigma_{\mathsf{call}} \uplus \Sigma_{\mathsf{int}} \uplus \Sigma_{\mathsf{ret}}$ 是用于编码输入的下推字母表，Γ 是一个称为栈字母表的有限集，$\bot \notin \Gamma$ 是栈底符号，$Q_F \subseteq Q$ 是接受状态集，δ 是转移关系。δ 将被假定划分为三个部分，即 $\delta_{\mathsf{call}} \uplus \delta_{\mathsf{ret}} \uplus \delta_{\mathsf{int}}$，其中 $\delta_{\mathsf{call}} \subseteq (Q \times \Sigma_{\mathsf{call}} \times Q \times \Gamma)$ 是调用符号上的转移，$\delta_{\mathsf{int}} \subseteq (Q \times \Sigma_{\mathsf{int}} \times Q)$ 是内部符号上的转移，而$\delta_{\mathsf{ret}} \subseteq (Q \times \Sigma_{\mathsf{ret}} \times (\Gamma \cup \bot) \times Q)$ 是返回符号上的转移。

设定一个 NVPA $N = (Q, q_{\mathsf{in}}, \Sigma, \Gamma, \bot, \delta, Q_F)$。与下推系统的情形类似，$N$的配置是形如 $(q, \alpha\bot)$ 的一对，其中 $q \in Q$且 $\alpha \in \Gamma^*$。函数 state(·)、stack(·) 和 top(·) 的定义方式与下推系统配置相同。N 在输入字符串$w \in \Sigma^{\omega}$ 上的运行是一个无限配置序列 $c_0, c_1, \ldots$，满足 $c_0 = (q_{\mathsf{in}}, \bot)$ 是 N的初始配置，且相邻配置与读入的输入符号以及 N的转移关系一致。更具体地，对于每个 $i \in \mathbb{N}$，(a) 如果 $w(i) \in \Sigma_{\mathsf{call}}$，那么 $(\mathrm{state}(c_i)\ w(i), \mathrm{state}(c_{i+1}), \mathrm{top}(c_{i+1})) \in \delta_{\mathsf{call}}$，且 $\mathrm{stack}(c_{i+1}) = \mathrm{top}(c_{i+1})\mathrm{stack}(c_i)$；(b) 如果 $w(i) \in \Sigma_{\mathsf{int}}$，那么

(state(c_i) $w(i)$, state(c_{i+1})) $\in$ δ_{int}，且 stack(c_{i+1}) $=$stack(c_i)；并且 (c) 如果$w(i)$ $\in$ Σ_{ret}，那么 (state(c_i) $w(i)$, top(c_i), state(c_{i+1})) $\in$ δ_{ret}，并且 stack(c_{i+1}) $=$ stack(c_i) $= \perp$，当且仅当 top(c_i) $= \perp$ 且 stack(c_i) $=$ top(c_i)stack(c_{i+1})，如果 top(c_i) $\neq \perp$。值得注意的是，调用符号上的转移会将一个符号压入栈中，内部符号上的转移保持栈不变，而返回符号上的转移通常会弹出栈顶符号，除非栈为空（即 $= \perp$），此时栈保持不变。

若运行 $c_0, c_1, \cdots$ 在输入 w 上满足 Büchi 接受条件，则称其为接受的，也就是说，对于某个 $q \in Q_F$，存在无穷多个 i，使得 state(c_i) $= q$。最后，NVPA N 接受的语言记为 $\mathcal{L}$(N)，它是

$$\mathcal{L}(N) = \{w \in \Sigma^{\omega} \mid N \text{ has an accepting run on } w\}.$$

C 单向交替跳转自动机（1-AJA）

我们介绍的第二种处理下推字母表上的字符串的自动机模型是单向交替奇偶跳跃自动机（*1-AJA*）[4]。1-AJA 在计算能力上与 NVPA 等价（即接受相同类别的语言），但在描述算法时提供了更大的灵活性。1-AJA 是交替自动机，这意味着它们可以根据机器在输入词上的多次运行来定义接受条件。尽管它们是没有辅助存储的有限状态机，但它们在读取一个符号时能够生成一条计算线程并跳转到输入字符串的后续部分，因此具有与更传统的带存储机器（如 NVPA）相同的计算能力。我们先给出一些必要的定义，然后再介绍这一模型。

交替自动机的转移标识了下一步中必须全部接受的子集，机器才能接受输入。我们用布尔函数描述这些下一步子集。对于集合 X，令 $\mathcal{B}^+(X)$ 表示由 X 中的元素作为命题构成的正布尔表达式 集合，即由 true 、 false 、 X 、 $\wedge$ 和 $\vee$ 构成的命题逻辑公式。给定 $\varphi \in \mathcal{B}^+(X)$ 和 $A \subseteq X$，如果在一种赋值下，φ 的求值结果为 true，我们就称 $A \models \varphi$；该赋值将 A 中的元素赋为 true，并将其他所有元素赋为 false。公式 $\varphi \in \mathcal{B}^+(X)$ 的对偶 记作 dual(φ)，它由 φ 替换得到：将 true 替换为 false，将 false 替换为 true，将 $\vee$ 替换为 $\wedge$，并将 $\wedge$ 替换为 $\vee$。更准确地说，dual($\cdot$) 可按如下方式归纳定义，其中 $x \in X$。

$$\begin{array}{lll} \mathsf{dual}(\mathsf{true}) = \mathsf{false} & \mathsf{dual}(\mathsf{false}) = \mathsf{true} & \mathsf{dual}(x) = x \\ \mathsf{dual}(\varphi \wedge \psi) = \varphi \vee \psi & \mathsf{dual}(\varphi \vee \psi) = \varphi \wedge \psi & \end{array}$$

1-AJA 是一种有限状态自动机，它读取下推字母表上的输入。换言之，读取一个输入符号时，计算线程会根据当前状态和所读符号改变其控制状态。然而，1-AJA 的一项新颖特性是，它在读取调用符号时能够跳转 。因此，1-AJA 的一次转移不仅指定下一个状态是什么，还指定下一个要读取的符号是什么。当所读符号是内部符号或返回符号时，下一个要读取的符号总是紧随其后的那个符号；

这用 $\to$ 表示。然而，当所读符号是调用符号时，自动机可以选择读取下一个符号，也可以选择接着读取匹配的返回符号； $\curvearrowright$ 用于表示自动机应接着读取匹配的返回符号。现在，我们已经具备了正式定义 1-AJA 所需的全部记号。

定义 6（1-AJA）。*A* 单向交替奇偶跳转自动机（1-AJA）是一个元组 $\mathcal{A} = (Q, q_{\mathsf{in}}, \Sigma, \delta, \text{优先级})$，其中，$Q$ 是一个有限的控制状态集，$q_{\mathsf{in}} \in Q$是初始控制状态，$\Sigma = \Sigma_{\mathsf{call}} \cup \Sigma_{\mathsf{int}} \cup \Sigma_{\mathsf{ret}}$ 是用于编码输入的下推字母表，$\delta : (Q \times \Sigma) \to \mathcal{B}^+(Q \times \{\to, \curvearrowright\})$ 是转移关系，并满足以下限制：对于任意的 $q \in Q$ 和 $a \in \Sigma_{\mathsf{int}} \cup \Sigma_{\mathsf{ret}}$，$\delta(q, a) \in \mathcal{B}^+(Q \times \{\to\})$，且 优先级 : $Q \to \mathbb{N}$ 是奇偶函数。

为了定义 1-AJA 在输入字符串上的运行，我们需要引入一些记号。固定一个下推字母表 $\Sigma = \Sigma_{\mathsf{call}} \cup \Sigma_{\mathsf{int}} \cup \Sigma_{\mathsf{ret}}$。我们可以将良匹配字符串的概念扩展到 Σ^*，如下所示。若字符串 $z \in \Sigma^*$ 满足以下任一条件： $z = \varepsilon$、 $z \in \Sigma_{\mathsf{int}}$、 $z = c\,u\,r$ 或 $z = uv$，则称其为匹配良好，其中 $c \in \Sigma_{\mathsf{call}}$、 $r \in \Sigma_{\mathsf{ret}}$ 和 u、$v \in \Sigma^*$ 均为（递归地）匹配良好的。固定一个输入字符串 $w \in \Sigma^\omega$。w 中位置 i 的抽象后继记为 $\curvearrowright(i)$，定义如下。

$$\curvearrowright(i) = \begin{cases} j & \text{if } w[i : j+1] = c\,u\,r \\ & \text{where } u \text{ is well matched,} \\ & c \in \Sigma_{\mathsf{call}}, r \in \Sigma_{\mathsf{ret}} \\ \mathsf{undefined} & \text{otherwise} \end{cases}$$

请注意，只有当 $w(i) \in \Sigma_{\mathsf{call}}$ 时，才定义 $\curvearrowright(i)$；若其有定义，其值就是其匹配的返回符号所在的位置。抽象后继确定了当 1-AJA 决定在调用处跳转 时将读取的下一个符号所在的位置。接下来，位置 i 在 w处的局部后继 （记作 $\to(i)$）始终定义为 $i + 1$。

固定一个 1-AJA $\mathcal{A} = (Q, q_{\mathsf{in}}, \Sigma, \delta, \mathsf{parity})$ 以及输入字符串 $w \in \Sigma^\omega$。$\mathcal{A}$在 w 上的一次运行是一个有根标记树 $R = (V, E, r, L)$，其中 V是顶点集，E 是边集，$r \in V$是根，$L : V \to (Q \times \mathbb{N})$。顶点的标签表示 $\mathcal{A}$ 的状态以及正在读取的 w 中的位置。我们要求 $L(r) = (q_{\mathsf{in}}, 0)$，即自动机处于初始状态，且正在读取第 0 个符号。令 $v \in V$ 为任意一个满足 $L(v) = (q, i)$ 的顶点，并令 $C \subseteq V$ 为 v 在 R中的子节点集合。我们要求 v 的标签与 C 中各顶点的标签符合 δ ——存在一个集合 $A \subseteq (Q \times \{\to, \curvearrowright\})$，使得：(a) $A \models \delta(q, w(i))$；(b) 对于每个满足 $d(i)$有定义的 $(q', d) \in A$，都存在一个顶点 $n \in C$，使得 $L(n) = (q', d(i))$；以及 (c) 对于每个顶点 $n \in C$，都存在 $(q', d) \in A$，使得 $d(i)$ 有定义，且$L(n) = (q', d(i))$。如果 $R = (V, E, r, L)$中的每条无限路径都满足由优先级定义的奇偶接受条件，那么这次运行就是接受的。也就是说，对于 R中的每条无限路径 σ，$\mathsf{mp}(\sigma)$ 都是偶数，其中

$$\mathsf{mp}(\sigma) = \min\{\mathsf{parity}(q) \mid \text{for infinitely many } i,\ L(\sigma(i)) = (q, k) \text{ for some } k\}.$$

最后，一如既往，$\mathcal{A}$ 所识别的语言定义为

$$\mathcal{L}(\mathcal{A}) = \{w \in \Sigma^\omega \mid \mathcal{A} \text{ has an accepting run on } w\}.$$

D 引理 1 的证明

在开始证明引理 1 之前，我们先回顾一下为什么 NVPA 和 1-AJA 对语言的标准运算封闭。我们重点关注所得自动机的大小，并陈述以下观察。对于语言 $A \subseteq \Sigma^\omega$ 和子集 $\Gamma \subseteq \Sigma$，我们将 ΓA 取为语言 $\{aw \mid a \in \Gamma \text{和 } w \in A\}$。

命题 3. 设 $\mathcal{A}_i$ 是一个识别大小为 n_i的 $L_i \subseteq \Sigma^\omega$ 的 *1-AJA*（*NVPA*），其中 $i \in \{1, 2\}$。设 $\Gamma \subseteq \Sigma$。那么可以在多项式时间内构造以下自动机。

1. 存在一个识别 $L_1 \cup L_2$ 的 *1-AJA*（*NVPA*），其大小为 $O(n_1 + n_2)$。*2.* 存在一个识别 ΓL_1 的 *1-AJA*（*NVPA*），其大小为 $O(n_1)$。*3.* 存在一个识别$(\Sigma^\omega \setminus L_1)$ 的 *1-AJA*，其大小为 $O(n_1)$。

证明（概要）。为证明 (1)，所构造的自动机非确定性地选择运行 $\mathcal{A}_1$ 或 $\mathcal{A}_2$，以识别 $L_1 \cup L_2$。通过检查第一个符号是否属于 Γ（对于 NVPA，还要执行第一个符号所要求的必要栈操作），然后运行 L_1 的自动机，即可证明 (2)。最后，对于 (3)，识别 L_1 的补集的 1-AJA 与原自动机具有相同的状态集，每个状态的优先级都增加 1；对于任意状态 q 和符号 a，$\delta(q, a)$ 为 $= \mathsf{dual}(\delta_1(q, a))$，其中 δ_1 和 δ 分别是 $\mathcal{A}_1$ 以及识别 $(\Sigma^\omega \setminus L_1)$ 的自动机的转移函数。

我们将根据 ψ 的结构，归纳地构造 $\mathcal{A}_\psi$。所构造的自动机类型将与 $\mathsf{fc}(\psi)$ 的奇偶性一致：若 $\mathsf{fc}(\varphi)$ 为奇数，则构造 NVPA；若 $\mathsf{fc}(\psi)$ 为偶数，则构造 1-AJA。

情形 $\psi = a_\pi$**：** 令 $\Gamma \subseteq \Sigma[m]$ 为符号集合 ζ，其中 a 属于符号 ζ中第 π个控制状态的标签集。我们将使用命题 3 (2) 构造一个接受$\Gamma(\Sigma[m])^\omega$ 的 1-AJA。

情形 $\psi = \neg\psi_1$**：** 我们将使用定理 1 将 $\mathcal{A}_{\psi_1}$ 转换为 1-AJA（如有必要），然后使用命题 3 (3) 构造一个接受语言$\mathcal{L}(\mathcal{A}_{\psi_1})$的 1-AJA。

情形 $\psi = \psi_1 \vee \psi_2$**：** 不失一般性，假设 $\mathsf{fc}(\psi_1) \geq \mathsf{fc}(\psi_2)$。我们使用定理 1 将 $\mathcal{A}_{\psi_2}$ 转换为与 $\mathcal{A}_{\psi_1}$ 类型相同的自动机（如有必要），然后使用命题 3 (1) 构造一个识别$\mathcal{L}(\mathcal{A}_{\psi_1}) \cup \mathcal{L}(\mathcal{A}_{\psi_2})$的自动机。

情形 $\psi = \mathsf{X}\psi_1$**：** 我们使用命题 3 (2) 构造一个用于$\Sigma[m]\mathcal{L}(\mathcal{A}_{\psi_1})$的自动机。

情形 $\psi = \psi_1 \,\mathsf{U}\, \psi_2$**：** 利用定理 1，我们将构造（如有需要）一个与 $\mathcal{A}_{\psi_i}$ 等价的 1-AJA，并记为 $\mathcal{A}_i = (Q_i, q_{\mathsf{in}_i}, \Sigma[m], \delta_i, \text{优先级}_i)$，其中 $i \in \{1, 2\}$。

ψ的 1-AJA 定义为 $\mathcal{A}_\psi = (Q_1 \cup Q_2 \cup \{q_{\mathsf{in}}\}, q_{\mathsf{in}}, \Sigma[m], \delta,$ 优先级$)$，其中$\delta(q_{\mathsf{in}}, \zeta) = \delta_2(q_{\mathsf{in}_2}, \zeta) \vee ((\rightarrow, q_{\mathsf{in}}) \wedge \delta_1(q_{\mathsf{in}_1}, \zeta))$，优先级 $(q_{\mathsf{in}}) = 1$，并且对于 $q \in Q_i (i \in \{1, 2\})$，有 $\delta(q, \zeta) = \delta_i(q, \zeta)$，且优先级$(q)$ =优先级$_i(q)$。

情形 $\psi = \exists \pi.\, \psi_1$：回顾一下，我们的下推系统 $\mathcal{P}$ 具有控制状态 S、栈字母表 Γ 和转移关系 $\Delta = \Delta_{\mathsf{int}} \cup \Delta_{\mathsf{call}} \cup \Delta_{\mathsf{ret}}$。利用定理 1，我们将构造一个（如有必要）与 $\mathcal{A}_{\psi_1}$ 等价的 NVPA，并将其记为 $\mathcal{A}_1 = (Q_1, q_{\mathsf{in}_1}, \Sigma[m+1], \Gamma_1, \bot, \delta_1, Q_{F_1})$。注意，$\mathcal{A}_1$ 的输入字母表为 $\Sigma[m+1],$，其中第 $m+1$ 个分量是分配给变量 π的路径编码。ψ 的自动机基本上会猜测一条路径的编码，该路径与 $\mathcal{P}$的转移一致，并通过运行自动机 $\mathcal{A}_1$来检查将猜测的路径分配给变量 π 是否满足 ψ_1。我们还有一个额外要求：猜测的路径必须从与分配给变量 π_m的路径当前配置相同的配置开始。为了能够猜测路径，$\mathcal{A}_\psi$ 会在其控制状态中记录 $\mathcal{P}$的控制状态，并使用其栈跟踪猜测路径上 $\mathcal{P}$的栈操作。在正式定义 $\mathcal{A}_\psi$ 之前，我们需要引入一些方便的记号。元素 $\zeta \in \Sigma[m]$ 的形式为 $(\mathsf{o}, (s_1, s_2, \ldots s_m), (a_1, a_2, \ldots a_m))$。我们用 $\mathsf{op}(\zeta)$ 表示 o，用 $\mathsf{state}(\zeta)|_i$ 表示 s_i，用 $\mathsf{stack}(\zeta)|_i$ 表示 a_i，其中 $i \in \{1, 2, \ldots m\}$。按照约定，我们规定 $\mathsf{state}(\zeta)|_0 = s_{\mathsf{in}}$和 $\mathsf{stack}(\zeta)|_0$ 未定义。最后，对于 $s \in S$和 $a \in \Gamma$，$\zeta + (s, a)$ 是字母表 $\Sigma[m+1]$ 中的符号，由 $(\mathsf{o}, (s_1, \ldots s_m, s), (a_1, \ldots a_m, a))$ 给出。

NVPA $\mathcal{A}_\psi = ((Q_1 \times S) \cup \{q_{\mathsf{in}}\}, q_{\mathsf{in}}, \Sigma[m], (\Gamma_1 \times \Gamma)\ \bot, \delta, (Q_{F_1} \times S))$；注意，栈底符号（$\bot$）与 $\mathcal{A}_1$ 中的相同。转移关系 $\delta = \delta_{\mathsf{call}} \cup \delta_{\mathsf{ret}} \cup \delta_{\mathsf{int}}$ 定义如下。

- When $\mathsf{op}(\zeta) = \mathsf{call}$,

$$\begin{aligned}\delta_{\mathsf{call}} = \{(q_{\mathsf{in}}, \zeta, (q, s), (b, a)) \mid (q_{\mathsf{in}_1}, \zeta + (\mathsf{state}(\zeta)|_m, a), q, b) \in \delta_1 \text{ and} \\ (\mathsf{state}(\zeta)|_m, (s, a)) \in \Delta_{\mathsf{call}}\} \cup \\ \{((q, s), \zeta, (q', s'), (b, a)) \mid (q, \zeta + (s, a), q', b) \in \delta_1,\ (s, (s', a)) \in \Delta_{\mathsf{call}}\}.\end{aligned}$$

- When $\mathsf{op}(\zeta) = \mathsf{int}$,

$$\begin{aligned}\delta_{\mathsf{int}} = \{(q_{\mathsf{in}}, \zeta, (q, s)) \mid (q_{\mathsf{in}_1}, \zeta + (\mathsf{state}(\zeta)|_m, \varepsilon), q) \in \delta_1,\ (\mathsf{state}(\zeta)|_m, s) \in \Delta_{\mathsf{int}}\} \\ \cup \{((q, s), \zeta, (q', s')) \mid (q, \zeta + (s, \varepsilon), q') \in \delta_1,\ \text{and } (s, s') \in \Delta_{\mathsf{int}}\}.\end{aligned}$$

- When $\mathsf{op}(\zeta) = \mathsf{ret}$,

$$\begin{aligned}\delta_{\mathsf{ret}} = \{(q_{\mathsf{in}}, \zeta, \bot, (q, s)) \mid (q_{\mathsf{in}_1}, \zeta + (\mathsf{state}(\zeta)|_m, \mathsf{stack}(\zeta)|_m), \bot, q) \in \delta_1 \text{ and} \\ ((\mathsf{state}(\zeta)|_m, \mathsf{stack}(\zeta)|_m), s) \in \Delta_{\mathsf{ret}}\} \cup \\ \{((q, s), \zeta, \bot, (q', s')) \mid (q, \zeta + (s, \mathsf{stack}(\zeta)|_m), \bot, q') \in \delta_1,\ \text{and} \\ ((s, \mathsf{stack}(\zeta)|_m), s') \in \Delta_{\mathsf{ret}}\} \cup \\ \{((q, s), \zeta, (b, a), (q', s')) \mid (q, \zeta + (s, a), b, q') \in \delta_1,\ ((s, a), s') \in \Delta_{\mathsf{ret}}\}\end{aligned}$$

要求 π 的猜测路径从与分配给 π_m 的路径当前配置相同的配置开始，这使得 δ 的定义中有几点值得强调。来自 q_{in} 的转移，无论

是调用、内部操作还是返回，都会在 $\mathcal{P}$ 中选取一个步骤，该步骤的起始状态与分配给 π_m 的路径中的状态相同。$\mathcal{P}$ 沿猜测路径压入的栈符号，会由 $\mathcal{A}_\psi$ 压入其栈中（见 δ_{call}）。如果在一次返回转换处，$\mathcal{A}_\varphi$ 的栈为 $\perp$，这意味着在猜测路径上弹出的符号必须来自路径起始时栈上的内容。由于这与映射到 π_m 的路径上的配置相匹配，因此弹出的符号必须与为 π_m 弹出的符号相同。对于包含 $\perp$ 的情形，这一点反映在 δ_{ret} 中。

我们将利用一个重要观察：如果 $\psi = \exists \pi.\, \psi_1$ 是一个句子，那么以下更强的正确性保证成立：对于任意 $\rho \in${调用、内部操作、返回}$^\omega$，$\rho \in \mathcal{L}(\mathcal{A}_\varphi)$ 当且仅当 ρ 是一个栈访问模式，并且 $\mathcal{P}$、$[\dagger \mapsto \rho]$、$\dagger \models \psi$。在这种情况下，$\mathcal{A}_\psi$ 的语言恰好由满足 ψ 的路径环境集合构成。这个更强的陈述由 $\mathcal{A}_\psi$ 的构造得出。

为完成证明，我们将通过对 fc(ψ) 进行归纳来界定 $\mathcal{A}_\psi$ 的大小。回忆一下，n 是 $\mathcal{P}$ 和 ψ 大小的界。

基础情形： 考虑满足 fc(ψ) $= 0$ 的 ψ。根据 fc(·) 的定义（定义 4），这意味着 ψ 由命题、$\neg$、$\vee$、X 和 U 构成；特别地，在这种情况下，ψ 中不存在路径量词。注意，a_π 的构造是一个常数大小的自动机。此外，$\neg$、$\vee$、X 和 U 的构造至多使自动机的大小增加一个常数因子。根据这些观察，在这种情况下，$\mathcal{A}_\psi$ 的大小由 $O(n)$ 界定，而该值又由 $g_{O(1)}(0, n)$ 界定。因此基础情形成立。

归纳步骤：我们根据 fc(ψ) 的优先级，将归纳步骤分为两种情形。当 fc(ψ) $= 2k$（对某个 $k \geq 1$）时，ψ 由（公式）复杂度至多为 $2k$ 的子公式构成，并使用布尔运算符（$\neg$、$\vee$）、X 和 U。假设我们首先为每个（公式）复杂度为 $< 2k$ 的子公式构造 1-AJA。这样会导致至多二次膨胀（定理 1），因此每个此类子公式对应的自动机大小至多为 $(g_c(k, n))^2$，其中 c 为某个值。$\neg$、$\vee$、X 和 U 的构造所生成的自动机大小至多为组成自动机大小之和的一个常数倍。因此，$\mathcal{A}_\psi$ 的大小至多为 $dn(g_c(k, n))^2 \leq g_{c'}(k, n)$，其中 c' 为某个值，从而证明了这一情形下的结论。现在考虑 fc(ψ) $= 2k+1$（对某个 $k \geq 0$）的情形。在这种情形下，ψ 由 $\vee$、X 和 $\exists$ 构成，用以组合（公式）复杂度为 $\leq 2k+1$ 的子公式。同样，我们可以将每个（公式）复杂度为 $< 2k+1$ 的子公式对应的自动机转换为非确定性可见下推自动机，代价为指数级（定理 1）。因此，可以假设所有这些自动机的大小均由 $g_c(k+1, n)$ 界定，其中 c 为某个值。根据命题 3 可知，析取和 X 生成的自动机，其大小增长至多为一个常数倍。存在量化是唯一会导致非平凡膨胀的算子。根据本证明中概述的构造，若 φ 具有一个大小为 ℓ 的 NVPA，则 $\exists \pi.\, \varphi$ 的自动机大小为 $O(n\ell)$，因为 n 界定了 $\mathcal{P}$的大小。由于量词至多有 n 个，因此 $\mathcal{A}_\psi$的大小可界定为 $dn^n(g_c(k+1, n)) \leq 2^{d'n \log n + g_c(k,n)} \leq g_{c'}(k+1, n)$；最后一步是因为$g_c(k, n) \geq cn \log n$。由此证明了归纳步骤。

E 定理 4 的证明

我们将证明，每个语言 $L \in \mathsf{ASPACE}(h_c(k-1,n))$ 都可归约到公式复杂度为 $2k-1$ 的 sHLTL句子的模型检测问题。由于对于 $f(n) \geq \log n$，$\mathsf{ASPACE}(f(n)) = $ 确定性时间复杂度 $(2^{O(f(n))})$，定理即得证。

考虑任意一台 $h_c(k-1,n)$-空间有界交替图灵机（ATM） M。由于 $h_c(k-1,n) \geq n$，我们可以假设 M 是一台单带图灵机。令 $M = (Q_\exists, Q_\forall, \Sigma, \Gamma, \sqcup, q_{\mathsf{in}}, \delta, q_a)$，其中 $Q_\exists$ 是存在控制状态集合，$Q_\forall$ 是全称控制状态集合，Σ 是输入字母表；$\Gamma \supseteq \Sigma$ 是带字母表，$\sqcup \in \Gamma \setminus \Sigma$ 是空白符号，$q_{\mathsf{in}} \in Q_\exists \cup Q_\forall$ 和 $q_a \in Q_\exists$ 分别是初始状态和接受状态，而 δ 是该图灵机的转移函数。我们用 $Q = Q_\exists \cup Q_\forall$ 表示 M 的所有状态构成的集合。我们假设 q_a 是一个停机状态（没有可执行的转移），因此 $\delta\colon (Q \setminus \{q_a\}) \times \Gamma \to 2^{(Q \times \Gamma \times \{\rightarrow, \leftarrow\})}$；给定一个状态和当前读取的符号后，转移函数会确定下一状态、要写入的符号以及读写头的移动方向（$\rightarrow$ 表示向右，$\leftarrow$ 表示向左）。不失一般性地，我们假设对于每一对 $(q,b) \in Q \times \Gamma$，$|\delta(q,b)| \in \{0,2\}$，也就是说，每一步要么没有选择，要么有两个选择。此外，我们假设这些选择按某种方式排序，因此常常会将 $i \in \{1,2\}$ 的“选择 i”作为讨论对象。

c的配置是 $\Gamma^*(Q \times \Gamma)\Gamma^*$中的一个字符串，其中 $c = u(q,b)v \,\mathsf{with}\, u, v \in \Gamma^*$，$b \in \Gamma$和 $q \in Q$表示 M的纸带内容为字符串 ubv，控制状态为 q，且读头正在读取包含 b的单元格。由于 M 是$h_c(k-1,n)$-空间有界的，我们可以假设 M 的任意配置 c都是长度恰为 $h_c(k-1,n)$的字符串。 M 在长度为 n的输入 bw 上的初始配置是$(q_{\mathsf{in}}, b)w\sqcup^{h_c(k-1,n)-n}$。配置 $c = u(q,b)v$若满足 $q \in Q_\exists$，则为存在配置；若满足 $q \in Q_\forall$，则为全称配置；若满足 $q = q_a$，则为接受配置。对于一对配置 c 和 c'，若 M的配置从配置 c出发，按照选择 $i \in \{1,2\}$执行一步即可到达，我们称 $c \vdash_i c'$。 M 在输入 w 上的一次运行是一个有限的有根二叉树$T = (V, E, r, \ell)$，其中 V是顶点集，E是从根向外定向的边集，$r \in V$ 是根，ℓ 是一个将每个顶点映射到 M的某个配置的函数。此外，ℓ 还须满足以下条件：根 r 标记为初始配置。对于任意内部顶点 $v \in V$，若 $\ell(v)$是存在配置，则 v 有一个子节点 c，使得对于某个 $i \in \{1,2\}$，$\ell(v) \vdash_i \ell(c)$；若 $\ell(v)$是全称配置，则v 有两个子节点 c_1, c_2，使得对于 $i \in \{1,2\}$，$\ell(v) \vdash_i \ell(c_i)$。最后，若 T 的每个叶节点都标记为接受配置，则运行 T为接受运行。若 M在 w上有接受运行，则输入 w被 M接受。

我们将构造一个从 $\mathcal{L}(M)$（根据定义，它属于 $\mathsf{ASPACE}(h_c(k-1,n))$）到 sHLTL 模型检测问题的归约。也就是说，给定输入 w，我们将构造一个下推系统 $\mathcal{P}_w$ 和 sHLTL句子 θ_w，使得 $\mathcal{P}_w \models \theta_w$ 当且仅当 $w \in \mathcal{L}(M)$。该归约的核心思路是构造一个下推系统 $\mathcal{P}_w$，使得从 $[\![\mathcal{P}_w]\!]$ 中的初始配置出发的路径集的标签能够编码 M 在 w上的可能计算。并且 θ_w

其构造目的是检查一条路径是否编码了 M 在 w上的有效接受运行。为了将这一思路形式化，我们首先需要找到一种方法，将 M的运行（它们是二叉树）编码为标签字符串。

交替图灵机运行编码。回想一下， M 的一次运行是一棵二叉树，我们需要找到一种将这棵树编码为字符串的方法。一种忠实实现这一目标的方法，是让编码记录对树进行深度优先搜索（DFS）遍历期间的栈操作。例如，假设有一棵树 T，其根为 r，左子节点为 T_1，右子节点为 T_2，那么在 DFS 过程中，算法会先将 r 压入栈中，对 T_1 进行 DFS 遍历（递归地），再将 r从栈中弹出，将 r 重新压入栈中，对 T_2 进行 DFS 遍历（递归地），最后弹出 r。对运行进行编码时，需要压入或弹出的不是节点 r，而是它的标签。请注意，要使一系列栈操作符合对一棵树的 DFS 遍历，压入和弹出的符号必须是同一个节点的标签。例如，在前面的树中，遍历完 T_1后，需要弹出并压入同一个节点 r。由于弹出时符号的顺序与压入时相反，对于较长的标签（例如配置），这一检查颇具挑战。为了解决这个问题，我们同时压入或弹出标签及其逆序形式。这样，如果要检查刚刚压入的字符串是否与刚刚弹出的字符串相同，就可以检查字符串相等，而不必检查一个字符串是否为另一个的逆序；使用 sHLTL 中的公式更容易完成这项检查。

我们将上述讨论形式化，以精确定义如何将二叉树编码为字符串。我们会滥用记号，并让 enc(·) 表示多个函数——上下文会明确我们指的是哪个 enc(·)。令 $\Lambda = \Gamma \cup (Q \times \Gamma)$ 为用于编码 M 的配置的字母表。对于$i \in \{1,2\}$，令 $[\Lambda]_i = \{[a]_i \mid a \in \Lambda\}$ 为字母表 Λ 的一个“副本”。对于长度为 m 的字符串 $c \in \Lambda^*$，我们定义

$$\begin{aligned}\mathsf{enc}(\mathsf{push}, c) = {} & (\mathsf{call}, [c(0)]_1, [c(m-1)]_2)(\mathsf{call}, [c(1)]_1, [c(m-2)]_2)\cdots \\ & (\mathsf{call}, [c(i)]_1, [c(m-i-1)]_2)\cdots(\mathsf{call}, [c(m-1)]_1, [c(0)]_2) \\ \mathsf{enc}(\mathsf{pop}, c) = {} & (\mathsf{ret}, [c(m-1)]_1, [c(0)]_2)(\mathsf{ret}, [c(m-2)]_1, [c(1)]_2)\cdots \\ & (\mathsf{ret}, [c(i)]_1, [c(m-i-1)]_2)\cdots(\mathsf{ret}, [c(0)]_1, [c(m-1)]_2)\end{aligned}$$

本质上，字符串的 enc(·) 同时编码了该字符串及其逆序形式，并带有一个标记，用于指示该字符串是被压入还是弹出；如果是弹出操作，则符号的顺序会被反转。考虑一棵有根的、带标签的二叉树 $T = (V, E, r, \ell)$。其编码按归纳方式定义如下。如果$V = \{r\}$ 和 $E = \emptyset$ （即， T 是一棵仅包含一个顶点的树），则 enc(T) = enc(push $\ell(r)$)enc(pop $\ell(r)$)。如果 r 只有一个子节点，且该子节点是子树 T_1，则 enc(T) =enc(push $\ell(r)$)enc(T_1)enc(pop $\ell(r)$)，其中 enc(T_1) 按递归方式给出。最后，如果 r 分别以 T_1 和 T_2 作为左、右子树，则 enc(T) =enc(push $\ell(r)$)enc(T_1) enc(pop $\ell(r)$)enc(push $\ell(r)$)enc(T_2)enc(pop $\ell(r)$)。在继续之前，让我们强调一下这种编码的一个微妙之处。对于一棵其中 r 具有两个子树 T_1 和 T_2的树，我们会在遍历 T_1 和 T_2之间“弹出” $\ell(r)$并“压入” $\ell(r)$。初读时这似乎是不必要的。注意，要使 T 成为有效的运行， T_2 的根的标签必须是执行一步得到的结果

从 $\ell(r)$ 出发执行一步。此类检查将被编码到我们的句子中，而要实现这一点，我们需要字符串编码中的连续 ATM 配置彼此相邻。

下推系统。 我们所构造的下推系统中的路径标签将对 M 在输入 w 上的可能运行进行编码。在每一步，下推系统都会猜测可能运行的下一个符号，并转移到标签与该符号对应的控制状态。栈用于确保在编码中遇到“弹出”符号时，它们与先前猜测中“压入”的符号相匹配。我们可以如下对其进行精确定义。回顾一下，对于 $\Lambda = \Gamma \cup (Q \times \Gamma)$， $[\Lambda]_i (i \in \{1,2\})$ 指的是 Λ 的“第 i 个副本”。令原子命题集 $\mathsf{AP} = \{\mathsf{call}, \mathsf{ret}, \mathsf{end}\} \cup [\Lambda]_1 \cup [\Lambda]_2$ 固定，并令 $S = \{\mathsf{call}, \mathsf{ret}\} \times [\Lambda]_1 \times [\Lambda]_2$。令 $\mathcal{P} = (S \cup \{s_{\mathsf{in}}, s_e\}, ([\Lambda]_1 \times [\Lambda]_2) \cup \{\bot\}, s_{\mathsf{in}}, \Delta, L)$，其中 $L(s_{\mathsf{in}}) = \emptyset$, $L(s_e) = \{\mathsf{end}\}$，且 $L((\mathsf{o}\ [a]_1, [b]_2)) = \{\mathsf{o}\ [a]_1, [b]_2\}$。转移关系 $\Delta = \Delta_{\mathsf{int}} \uplus \Delta_{\mathsf{call}} \uplus \Delta_{\mathsf{ret}}$ 定义如下： $\Delta_{\mathsf{int}} = \{(s_e, s_e)\}$并且

$$\begin{aligned}\Delta_{\mathsf{call}} &= \{(s_{\mathsf{in}}, (s, \bot)) \mid s \in S\} \cup \\ &\quad \{((\mathsf{call}, [a]_1, [b]_2), (s, ([a]_1, [b]_2))) \mid s \in S \text{ and } a, b \in \Lambda\} \\ \Delta_{\mathsf{ret}} &= \{(((\mathsf{ret}, [a]_1, [b]_2), ([a]_1, [b]_2)), s) \mid s \in S \text{ and } a, b \in \Lambda\} \cup \\ &\quad \{((s, \bot), s_e) \mid s \in S\}\end{aligned}$$

$[\![\mathcal{P}]\!]$ 的路径可能并不对应于 M 在 w 上的实际计算，因为 $\mathcal{P}$不会检查这类字符串必须满足的许多性质。另一方面， M 在 w 上的正确运行确实对应于 $\mathcal{P}$ 中以$(\mathsf{end})^\omega$结尾的路径。最终的下推系统将是对 $\mathcal{P}$的轻微修改，体现在两个方面。首先，我们会在状态和栈符号中添加一些额外的簿记，以确保每当猜测出一个全称配置时，计算都会包含对应于两种选择的转移。其次，我们需要修改该系统以考虑规约，正如我们将在本证明末尾看到的那样。

如前所述，要使 $\mathcal{P}$ 的一条路径的标签对应于 M 在 w上的接受计算编码，我们需要确保这些标签满足若干性质。这些性质将由我们的 sHLTL 句子编码。不过，我们不会直接用 sHLTL 编码这些条件，而是先用 QPTL（一种在 [17]中引入的逻辑）来表述它们，这样会更方便。我们先介绍这种逻辑，说明如何对接受运行的性质进行编码，然后描述如何将其转换回 sHLTL。

量化命题时序逻辑。 量化命题时序逻辑（QPTL） [17] 通过对命题进行量化扩展了线性时序逻辑。固定一个原子命题集合 AP，该逻辑中的公式由以下 BNF 文法给出；下文中， a 是 AP 的一个元素。

$$\varphi ::= a \mid \neg\varphi \mid \varphi \vee \varphi \mid \mathsf{X}\varphi \mid \mathsf{F}\,\varphi \mid \exists a.\, \varphi$$

QPTL 的模型与 LTL 的相同，即 $(2^{\mathsf{AP}})^\omega$ 的元素；大多数构造的语义也类似。对于 $w \in (2^{\mathsf{AP}})^\omega$，当且仅当 $a \in w(0)$ 时， a 成立；当 φ 不成立时， $\neg\varphi$ 成立；当 φ_1 或 φ_2 成立时， $\varphi_1 \vee \varphi_2$ 成立；

当 φ 在后缀 $w[1:]$上成立时，Xφ 成立；当 φ 在某个后缀$w[i:]$上成立时，Fφ 成立。唯一新增的算子是 $\exists a.\varphi$：如果在某个词 w' 中 φ 成立，且该词在所有命题（除了可能的 a）上的赋值都与 w 一致，那么 $\exists a.\varphi$ 就在 w中成立。回想一下，F φ等价于 true Uφ，G φ是 $\neg$F($\neg\varphi$) 的简写，而 $\forall a.\varphi$是 $\neg\exists a.(\neg\varphi)$。我们可以将 fc(·) 扩展到 QPTL 公式，定义保持不变；在此定义中，对于同源公式，$\neg$ 的作用类似于否定，而不是对 sHCTL* 句子的否定。最后，已有研究表明，每个 QPTL 公式都等价于一个前束范式公式，其中所有量词都被移到了公式的开头。

我们的量化命题时序逻辑公式 φ_w 用于描述一个词何时编码了 M 在 w上的接受运行，它将依赖于 [17]中构造的公式。第一个是公式$\varphi_{c,\ k,\ n}(p_1, p_2)$（引理 4.4.，见 [17]），其大小为 $O(k+n)$，并满足：当且仅当命题 p_1 和 p_2 在 w中恰好为真一次，p_2 在 p_1之后为真，且它们之间恰好相隔 $h_c(k,n)$个位置时，$w \models \varphi_{c,\ k,\ n}(p_1, p_2)$成立。此外，fc($\varphi_{c,\ k,\ n}$) $= 2k-1$，并且可以在 $O(\log n)$空间内构造。我们将在介绍所需条件时引入其他公式。

量化命题时序逻辑公式 φ_w。现在我们描述 φ_w，它具有这样的性质：$[\![\mathcal{P}]\!]$ 的一条路径满足 φ_w，当且仅当 M接受 w；此处，若路径 $c_0c_1\cdots$满足 $[\![\mathcal{P}]\!]$，则称其满足 φ_w，当且仅当 $L(c_0)L(c_1)\cdots \models \varphi_w$。注意，$\mathcal{P}$的构造保证了“弹出”的符号与“压入”的符号相同，并且每个全称配置都有两个后继，分别对应于每种选择所对应的转移。因此，要检查一条路径是否编码了 M 在 w 上的接受计算，还需满足以下条件。

1. 每个配置的长度恰好为 $h_c(k-1,n)$。2. 编码单个配置的所有符号具有相同的标记，即要么全部为调用，要么全部为返回。3. 第一个配置是 M 在 w上的初始配置。4. 相继的“压入”配置对应于与其转移函数一致的 M 的一步计算。5. 运行的“叶节点”是接受配置。换言之，如果一个压入配置后面紧接着一个弹出配置（由于 $\mathcal{P}$，它们将完全相同），那么控制状态必须为 q_a。6. 如果一个弹出配置后面紧接着一个压入配置（即对运行进行深度优先搜索遍历时正在探索右子节点），那么它们必须是同一个配置。7. 运行结束时，所有配置都已弹出。

如果 φ 表示用量化命题时序逻辑写出的上述所有条件的合取，那么我们所需的公式 φ_w 是 Xφ，因为 $\mathcal{P}$ 的初始状态 s_{in} 不属于对计算编码的猜测。

我们将介绍如何在量化命题时序逻辑中编码这些条件，这通常会用到 [17]中的公式。

1. 为了表述性质（1），我们将使用一个命题 $\triangleright$（它将被存在量化），标记每个配置的开始。$\triangleright$ 标记每个配置开始这一事实将由

我们写下的其他性质来保证。我们要求 $\triangleright$ 恰好每隔$h_c(k-1,n)$ 个位置为真，这可以用之前引入的公式 $\varphi_{c,\ k-1,\ n}(\cdot,\cdot)$ 表示。根据 [17] 中的公式（9），将 r 替换为 $\triangleright$ 即可得到。

2. 由于 $\triangleright$ 标记了每个配置的开头，因此，说一个配置中的所有符号都具有相同的标记，等价于说，如果标记发生变化，那么在标记变化处，$\triangleright$ 必须为真。换言之，性质（2）可以写为

$$\mathsf{G}((\mathsf{call} \wedge \mathsf{Xret}) \rightarrow \mathsf{X}\triangleright) \wedge \mathsf{G}((\mathsf{ret} \wedge \mathsf{Xcall}) \rightarrow \mathsf{X}\triangleright)$$

3. [17] 中的公式（10）陈述了性质（3）。需要稍作修改，以同时纳入以下条件：来自 $[\Lambda]_2$ 的命题对 M的初始配置进行反向编码。

4. [17] 中的公式（11）陈述了性质（4）。需要稍作修改，以确保使用 $[\Lambda]_2$ 进行的反向编码与 M的转移一致；且此要求仅针对带有调用标签的连续配置。

5. 请记住，连续配置中的对应符号相距 $h_c(k-1,n)$。性质（5）可以表述为：如果存在两个位置p和 q，它们相距 $h_c(k-1,n)$，且标记相反；并且此外，p 对应于读写头扫描的位置，那么 p 处的控制状态必须是接受状态 q_a。

$$\forall p_1 \forall p_2.\ (\varphi_{c,k-1,n}(p_1,p_2) \wedge \mathsf{F}(p_1 \wedge \mathsf{call} \wedge \bigvee_{a \in Q \times \Gamma} [a]_1) \wedge \mathsf{F}(p_2 \wedge \mathsf{ret})) \rightarrow \mathsf{F}(p_1 \wedge \bigvee_{a \in \{q_a\} \times \Gamma} [a]_1)$$

6. 利用性质 (5) 中的观察结果，性质 (6) 可写为

$$\forall p_1 \forall p_2.\ ((\varphi_{c,k-1,n}(p_1,p_2) \wedge \mathsf{F}(p_1 \wedge \mathsf{ret}) \wedge \mathsf{F}(p_2 \wedge \mathsf{call})) \rightarrow \bigvee_{a,b \in \Lambda} (\mathsf{F}(p_1 \wedge [a]_1 \wedge [b]_2) \wedge \mathsf{F}(p_2 \wedge [b]_1 \wedge [a]_2))$$

7. 只需要求路径以状态 s_2 结束，即可确保该性质。我们将其记作 Fend。

由于量化命题时序逻辑中的公式可以写成前束范式，我们可以将上面列出的各个公式中的所有全称量词提到前面，从而得到一个形如 φ_w 的公式 $\mathbf{X}\exists \triangleright \forall p_1 \forall p_2.\ \varphi$，其中 φ 是 $\neg\varphi_{c,k-1,n}(\cdot,\cdot)$ 与时序算子公式复杂度为 0 的简单公式的布尔组合。因此，$\mathrm{fc}(\varphi_w) = 2k-1$。

转换为 sHLTL。 我们的 QPTL 公式 φ_w具有 $\mathbf{X}\exists \triangleright \varphi'$ 的形式。我们将说明如何构造一个在 sHLTL 中“等价”的公式；这还需要修改我们的下推系统 $\mathcal{P}$，以得到最终的下推系统$\mathcal{P}_w$。按如下方式构造一个同源句子 ψ'。令 a 为一个新的命题 不

出现在 φ_w 中。对于 φ_w 中的每个量化的 命题 p，将 p 的每个约束出现替换为 a_{π_p}，并将量化 $\exists p$ 替换为 $\exists \pi_p$。令 π_* 为一个新的路径变量。将 φ_w 中的每个自由命题 p 替换为 p_{π_*}。令 ψ 为所得的同源公式。最后令 $\theta_w = E\exists \pi_*.\, \psi$。

为完成归约，我们需要修改 $\mathcal{P}$，以考虑在 θ_w 中引入的新命题 a。最终的下推系统 $\mathcal{P}_w$ 构造如下。回顾一下，$S =$（{调用，返回} $\times [\Lambda]_1 \times [\Lambda]_2$），且 $\mathcal{P}$ 的状态集合为 $S \cup \{s_{\mathsf{in}}, s_e\}$。令 $[S]_1$ 和 $[S]_2$ 为 S 的两个副本。$\mathcal{P}_w$ 的状态集合为 $[S]_1 \cup [S]_2 \cup \{s_{\mathsf{in}}, s_e\}$。状态 $s \in S$ 的两个副本具有相同的入边和出边转移，且对所有旧命题都具有相同的标签。唯一的区别是 $a \in L([s]_1)$，而 $a \notin L([s]_2)$。

容易得出以下结论：$\mathsf{fc}(\theta_w) = \mathsf{fc}(\varphi_w)$，并且 $[\![\mathcal{P}]\!]$ 的一条路径满足 φ_w 当且仅当 $\mathcal{P}_w \models \theta_w$。由此完成了困难性结果的证明。