南京大学

本科毕业设计

院　　系　软件学院

专　　业　软件工程

题　　目　面向 TLA$^+$ 规约的归纳不变式自动生成技术

年　　级　2020　学　　号　201250040

学生姓名　张继华

指导教师　魏恒峰　职　　称　助理研究员

提交日期　2024 年 5 月 27 日

南京大学本科毕业论文（设计）
诚信承诺书

本人郑重承诺：所呈交的毕业论文（设计）（题目：面向 TLA^+ 规约的归纳不变式自动生成技术）是在指导教师的指导下严格按照学校和院系有关规定由本人独立完成的。本毕业论文（设计）中引用他人观点及参考资源的内容均已标注引用，如出现侵犯他人知识产权的行为，由本人承担相应法律责任。本人承诺不存在抄袭、伪造、篡改、代写、买卖毕业论文（设计）等违纪行为。

作者签名：

学号：

日期：

南京大学本科生毕业论文（设计、作品）中文摘要

题目：面向 TLA^+ 规约的归纳不变式自动生成技术

院系：软件学院

专业：软件工程

本科生姓名：张继华

指导教师（姓名、职称）：魏恒峰　助理研究员

摘要：

分布式协议以及分布式系统，在当今的计算机世界不可或缺。自动化地对分布式协议验证其正确性是一个重要且困难的挑战。分布式协议的正确性表达为其定义的安全属性（safety property）在每个状态下都成立。对于复杂系统，我们无法像验证简单系统，通过遍历所有状态的方式来验证安全属性。已有的研究常常通过生成一个蕴含安全属性的归纳不变式的正确性的方式来验证安全属性的正确性，继而验证规约的正确性。

自动化地生成分布式协议的归纳不变式是自动化验证分布式协议正确性的关键步骤。但是，自动化寻找归纳不变式是一个困难的问题，并且已有的研究主要基于的 IVy 进行实现，TLA^+ 语言领域的相关研究较少。本文将实现一个面向 TLA^+ 规约的归纳不变式生成工具 RL-TLA，实现对以 TLA^+ 语言描述的分布式协议规约的归纳不变式的自动化生成。与此同时，不同于以往通过优化遍历顺序的方式，本文将使用强化学习的方法，加速归纳不变式的推导过程。通过引入 TLC 或 Apalache 模型检查器，实现对候选不变式和归纳不变式的验证。

通过面向部分规约的实验，我们可以验证 RL-TLA 的有效性，证明强化学习在归纳不变式生成中应用的可行性。

关键词：分布式协议；形式化验证；归纳不变式；TLA^+；强化学习

南京大学本科生毕业论文（设计、作品）英文摘要

THESIS: Automatic Inductive Invariants Inference for Specifications in TLA$^+$

DEPARTMENT:

SPECIALIZATION: Software Engineering

UNDERGRADUATE: Jihua Zhang

MENTOR: Hengfeng Wei, Research Assistant

ABSTRACT:

Distributed protocols and distributed systems are indispensable in contemporary computer world. Automatical inference for the correctness of distributed protocols is an important and challenging task. The correctness of distributed protocols is expressed as the safety property defined by the protocol holds in every state. For complex systems, we cannot verify the safety property by traversing all states as for simple protocols. Existing research often generates the correctness of an inductive invariant that implies the safety property to infer the correctness of the safety property, and then infer the correctness of the protocol.

Automatic generation of inductive invariants for distributed protocols is a key step in automatic verifying the correctness of distributed protocols. However, automatically finding inductive invariants is a difficult problem, and existing research is mainly based on IVy, with less research in the TLA$^+$language domain. This paper will implement an inductive invariant generation tool RL-TLAfor TLA$^+$specifications, to automatically generate inductive invariants for distributed protocol specifications in TLA$^+$language. Meanwhile, different from the previous way of optimizing the traversal order, this paper will use reinforcement learning to accelerate the derivation process of inductive invariants. By introducing TLC or Apalache, model checkers, the verification of candidate invariants and inductive invariants is realized.

Through experiments, we can verify the effectiveness of RL-TLAand prove the feasibility of application of reinforcement learning in inductive invariant generation.

KEYWORDS: Distributed Protocols; Formal Verification; Inductive Invariant; TLA$^+$; Reinforcement Learning

目　录

第一章　绪论

1.1　研究背景和意义

分布式协议，如 Paxos[1]和 Raft[2] 等，是现代分布式系统的基石。验证分布式协议的正确性，对保障大规模的数据库系统，云计算系统以及其他分布式系统运行的可靠性和稳定性至关重要。然而，目前分布式协议愈发复杂，验证分布式协议的正确性并不是一件容易的事情。在分布式协议验证中，安全属性（safety property）[3]具有至关重要的地位。如果在运行过程中，系统的每个状态都不与安全属性产生冲突，那么我们可以认为这个系统是安全的。因此，验证分布式协议的正确性，可以转化为验证系统的每个状态都满足安全属性的问题。对于复杂系统，我们无法简单地采用遍历的方式来验证安全属性是否在每个状态下都成立。目前的研究方式，大多是寻找一个归纳不变式（Inductive Invariant）[4]，这个不变式在所有的系统正确运行所能到达的状态上都成立。另外，这个不变式蕴含了安全属性，即如果这个不变式成立，那么安全属性也成立，且它具有归纳性，即如果这个不变式在一个状态成立，那么它在转移后的状态也成立。寻找到归纳不变式就意味着验证了分布式协议的正确性。[5] 自动化地生成分布式协议的归纳不变式是验证自动化分布式协议正确性的关键步骤。

TLA^+[6]是一个对程序和系统，尤其对高并发的程序和分布式的系统进行规约建模的高级语言。在高并发和分布式系统设计和开发过程中，非常容易发生基础性的设计问题，这些问题往往难以被发现。而 TLA^+ 以及其工具，利用集合论和时态逻辑精确地表达系统的状态和行为，可以帮助开发人员在设计阶段避免这些问题，以及在开发阶段定位问题。

目前，归纳不变式的自动生成技术，大多基于 IVy[7] 实现。然而，IVy 的功能相比较 TLA^+ 比较局限，且在工业界的应用也不及 TLA^+ 应用更加广泛。但是，当前针对 TLA^+ 规约的自动化归纳不变式生成工具较少，且实现方式比较初级，所能面向的规约比较简单。我们希望能 TLA^+ 语言上开发出一种新的自动化归

纳不变式生成方法，借助机器学习的技术以提高生成效率，降低生成难度，为分布式协议的设计和验证提供帮助。

1.2 研究问题及挑战

本文主要研究面向 TLA^+ 定义的规约的自动化归纳不变式生成技术，并探究机器学习在归纳不变式生成中的应用。

自动化生成归纳不变式是一个相对复杂的问题，需要通过 try-and-error 的方式不断地进行尝试，以获取到最终的归纳不变式。在这个过程中，需要不断对生成的结果进行验证，并基于验证结果不断调整，给出新的尝试。另外，目前基于 TLA^+ 的归纳不变式生成工具较少，且实现方法比较单一，提供的参考内容较少，也没有充足且全面的测试集合用以参考。

1.3 国内外研究现状

目前的归纳不变式生成技术主要是基于 IVy 实现的，科研人员基于 Ivy 的平台设计了诸多归纳不变式自动生成的算法和工具。

从实现理念和思路上，这些工具的大致可以分为两类，一种是基于程序语义（syntax-guided）[8]的白盒技术，另一种是基于程序行为的黑盒技术。近年来，随着 AI-for-SE 的发展，一种叫做 ICE[9]（implication counterexamples）[10]的学习框架流行起来，它将不变式的证明工作分为了两个部分：学习者和教育者。依赖随机搜索、决策树[11]、强化学习[12]等技术，许多工作推进了学习者模块的发展。此外，也有人将新颖的语言大模型引入了不变式生成的工作中[13]。白盒技术和黑盒技术的界限并不明确，一些工具其实兼而有之地采取两种技术的优势。

DistAI[14]以及 DuoAI[15]来自同一个研究团队，使用枚举候选不变式的算法进行自动不变式生成。他们基于已有的小体积的运行数据，在削减过的空间上，在有限的句法空间中通过工具裁剪谓词来生成候选不变式，他们首先要基于协议的定义，获得一部分的运行数据，即一些协议允许到达的状态。与此同时，他们还将量词模板进行划分，以减少意义上重复的候选不变式的出现。然后将获得的状态分配到对应的量词模板中，并考察每个状态下的谓词是否成立。基于运行数据，他们初步筛选出了一些可能的候选不变式，然后交给工具进行验证，最终

得到归纳不变式。如果运行数据不足以枚举出归纳不变式，他们也会生成更多的运行数据。总体而言，他们通过小部分的运行数据，削减了搜索空间，减少了验证器的调用次数，提高了不变式生成的效率。

I4[16]基于有限实例推广进行自动不变式生成。它首先会根据初始参数，创建一个有限的实例。然后使用 Averroes model checker[17]生成一个基于小实例的归纳不变式。如果实例过于复杂，I4 会将实例简化。之后，I4 会基于已经得到的，小规模实例上的归纳不变式，泛化到一般的归纳不变式。

LIPuS 则在基于语义的基础上，使用了强化学习的框架对搜索空间进行剪枝，并在修剪过后的空间上进行 SMT 求解。它首先将程序输入给强化学习框架，让其对总体的不变式模板进行修剪。然后将修剪之后的模板交给 SMT solver 进行求解。如果无法求解出来，出现了反例（counterexamples），则将反例交给强化学习框架，让其再次对模板进行修剪。直到 SMT solver 求解出了不变式，或者强化学习框架无法再修剪出新的模板为止。使用这种方式可以有效地减少对 SMT solver 的调用，从而提高生成归纳不变式的效率。

以上的工作，均是基于 IVy 实现，接受 IVy 描述的规约。目前基于 TLA^+ 的归纳不变式生成工具较少。

IronFleet[18]和 Verdi[19] 是比较早在 TLA^+ 上实现的分布式系统验证工具。IronFleet 结合使用细化和简化的方式来加速分布式协议的验证。而 Verdi 则是使用了一系列的系统转换器。它先证明比较强约束的模型的正确性，然后通过转换器，将这个模型转换为更弱的模型，再证明这个弱模型的正确性。事实上，IronFleet 和 Verdi 都离不开人工的加入来验证归纳不变式的正确性，并不是一个完全自动化的归纳不变式生成工具。

endive[20]是一份基于 TLA^+ 的自动化归纳不变式生成工具。endive 的实现基于用户提供的原子谓词作为种子，算法将种子按照一定顺序组合成不变式，并按照从短到长的顺序交给模型检查器检查，排除掉那些违反安全属性的不变式。之后，endive 会选择可以杀死反例最多的不变式，并重复这一过程，直到杀死所有反例，组合出最终的归纳不变式。endive 是一种基于 IC3 思想，基于增量搜索实现寻找归纳不变式的工具。

1.4 本文主要工作

本文使用强化学习的方法，使用 python 语言和 pytorch，基于 TLA^+ 语言平台，设计了一个自动化归纳不变式生成工具 RL-TLA。本文中的工具对归纳不变式的验证模块使用的 TLC 或 Apalache 工具进行验证。此外，我们基于 endive 所提供的测试数据集合，对 RL-TLA 功能和性能进行了验证和测试。实验验证了强化学习在寻找归纳不变式领域应用的有效性和可行性。

本文的主要工作包括:

- **预处理：**对 TLA^+ 源文件和配置文件的预处理，识别和存储必要数据，为自动生成归纳不变式模块做准备。
- **系统设计和实现：**设计了一个基于强化学习的归纳不变式生成工具 RL-TLA。其中包括归纳不变式生成模块和归纳不变式验证模块。其中，生成模块引入了强化学习，借助强化学习的优势，优化了归纳不变式的生成效率。检验模块主要调用模型检测器，对模型检测器返回的结果进行分析，判断归纳不变式的正确性和归纳性质和生成归纳反例。
- **实验和验证：**使用 endive 提供的测试数据集合，对 RL-TLA 的功能和性能进行验证和测试，证明 RL-TLA 的可行性。

本文的组织结构如下：

第一章主要介绍了项目的研究背景，研究问题，当前国内外在归纳不变式自动生成领域的研究现状，本文的工作内容和组织结构；

第二章介绍预备知识和相关技术；

第三章介绍系统的体系结构设计；

第四章介绍系统各个模块功能和模块间交互的具体实现；

第五章对工具的功能和性能进行验证，并与已有的工具进行对比。

第六章对本文工作进行总结与展望。

第二章　预备知识

本章节将以规约 *Client_Server* （图2-1）为例，介绍 TLA$^+$ 规约的基本结构，以及在寻找归纳不变式过程中的其他预备知识。

```
MODULE Client_Server

CONSTANTS Server, Client
VARIABLE locked, held

Init ≜
    ∧ locked = [i ∈ Server ↦ TRUE]
    ∧ held = [i ∈ Client ↦ {}]

Connect(client, server) ≜
    ∧   locked[server] = TRUE
    ∧   held' = [held EXCEPT ![client] = held[client] ∪ {server}]
    ∧   locked' = [locked EXCEPT ![server] = FALSE]

Disconnect(client, server) ≜
    ∧ server ∈ held[client]
    ∧ held' = [held EXCEPT ![client] = held[client] \ {server}]
    ∧ locked' = [locked EXCEPT ![server] = TRUE]

Next ≜
    ∨ ∃ client ∈ Client, server ∈ Server : Connect(client, server)
    ∨ ∃ client ∈ Client, server ∈ Server : Disconnect(client, server)

Spec ≜ Init ∧ □[Next]⟨locked, held⟩

Safe ≜
    ∀ client_i, client_j ∈ Client :
    (held[client_i] ∩ held[client_j] = {}) ∨ (client_i = client_j)
```

图 2-1　Client_Server 规约

2.1　TLA$^+$ 与 TLC

2.1.1　TLA$^+$

TLA$^+$[6]是由计算机科学家 Leslie Lamport 主导开发的，基于时许逻辑 TLA（temporal logic of actions）[21]的，在对计算机程序和系统建模，尤其是对并行系统和分布式系统建模具有广泛应用[22]的一种高级语言。它是基于使用简单的数学语言来精确描述系统行为的理念开发的。因此，TLA$^+$ 的表达方式和一般的编程

语言有很大的不同，反而和数学语言更为接近。TLA$^+$ 并不是一种编程语言，而是一种规约语言，它不关注协议或者系统的具体实现，从而能更高层次看到程序整体的设计。因此，TLA$^+$ 及其工具对于消除代码中很难发现和纠错成本高昂的错误非常有用。

需要注意到的是，TLA$^+$ 是一个对程序或者系统建模的语言，为了让规约开发人员能更好地表达一个协议或系统而设计的，并不是为了寻找归纳不变式而设计的。尽管如此，它的语法更加丰富，以更加直观的方式表达一个协议或系统，而且在工业界应用更加广泛，使得它在寻找归纳不变式的研究中有广阔的应用。

开发者使用 TLA$^+$ 或者其他工具来对分布式协议进行建模的代码，我们将其称之为规约（specification，简称 spec）。图 2-1 展示了一个简单的 TLA$^+$ 规约，其中包含了一个简单的客户端和服务器的通信协议。其中两个重要的谓词是 $Init$ 和 $Next$。$Init$ 表示系统的初始状态，描述系统最开始时的状态；而 $Next$ 则是表示系统的状态是如何转移，也就是系统的状态在每个时间片后会发生怎样的变化。谓词 $Safe$ 是安全属性（safety property），一个正确定义的分布式协议规约，应当在每个可达的状态下都满足安全属性。这个变量在自动化生成归纳不变式的研究中非常关键。在这个规约中，还有 $Connect$, $Disconnect$ 等这样的动作定义，使用这些定义，就像在一般的编程语言中使用函数一样，方便阅读和重复使用。除此以外，一些规约中还有谓词 $TypeOK$，用于约束变量的类型。另一种在自动归纳不变式生成研究中常常使用的工具，IVy，也有相似的语法和结构。可以看到的是，TLA$^+$ 更关注系统的状态和系统状态是发生怎样的转移，对于系统状态转移的具体实现，TLA$^+$ 并不关心。这样的描述方式和状态机非常相似。

TLC 是 TLA$^+$ 集成的模型检测工具。除了 TLC 以外，TLA$^+$toolbox[23] 还集成有 PlusCal[24]和 TLAPS 用于命题证明工具，sany 用于语法检查工具，tex 用于将 TLA$^+$ 美化打印的工具等，这些工具与本文所讨论的问题相关性不高，不展开讨论。

本文所述工具只接受 TLA$^+$ 的规约。

2.1.2 TLC

TLC 既是对 TLA$^+$ 规约的模型检查工具，也是一个面向规约的模拟器。它是一个显式状态模型检查器，依照用户给出的规约和设置，搜索所有满足约束的状

态和状态转移，并在这个过程中检查安全属性和其他用户定义的谓词逻辑时时是否成立。如果遇到错误，TLC 会将错误的状态和状态转移过程输出，以便用户进行分析。

TLC 可以通过使用超过 32 个计算机线程以获得近乎线性的加速。它可以通过在分布式部署的计算机网络上运行来进一步加速模型检查，并提供在云系统上的轻松部署。

Apalache[25-26]是另一由社区开发的模型检测器，和 TLC 不同的是，Apalache 并不是通过遍历所有可能的状态来检验安全属性是否成立，而是通过 SMT solver 来检验。它是将 TLA^+ 规约转换为 SMT 问题，然后使用 SMT solver（如 Z3[27]）求解来检验安全属性是否成立。Apalaches 是一种符号检查器，它和 SMT solver 一样基于逻辑推理和公式求解实现的。Apalache 对 TLA^+ 源文件的语法中引入了一些限制，尽管没有完全支持 TLA^+ 的所有语法，但是这方便使用 SMT 求解器进行求解。

TLA^+ 是一个“弱类型”的编程语言，它对变量没有严格的类型注明。但是，Apalache 需要了解 TLA^+ 规约中变量的类型才能工作。尽管 Apalache 有一套自己的类型推断系统，但是，它并不能完全解决所有的类型推断问题。这使得用户，对于某些协议，需要以注释的形式来提供变量的类型，才能交给 Apalache 进行处理。

2.2 归纳不变式与归纳反例

验证分布式协议的正确性，就是验证协议定义的安全属性（safety property）是否在每个可达的状态下都成立。在 *Client_Server* 规约中，我们可以看到 $Safe$ 是一个安全属性，它表达的是，在任何状态下，如果两个客户端同时连接有同一个服务器，那么这两个客户端是同一个客户端。换言之，两个不同的客户端不能连接到同一个服务器。

对于简单的系统，即变量和状态不多的系统，我们可以通过遍历每一个可能的状态来验证。但是对于稍微复杂一些的系统，尤其是越来越多的分布式系统，规模越来越大，状态也越来越复杂。通过简单的遍历的方式来验证系统的正确性，是不现实的。寻找一个能够蕴含安全属性的不变式，并且能够在所有可能的

状态转移后保持其自身的正确性，这个不变式被称为归纳不变式。以数学的语言表示为：

$$Init \vDash Ind \tag{2-1}$$

$$Ind \wedge Next \vDash Ind' \tag{2-2}$$

$$Ind \vDash Safe \tag{2-3}$$

其中 $Init$ 表示初始状态，$Next$ 表示状态转移，$Safe$ 是安全属性，Ind 表达的是归纳不变式，而 Ind' 表达谓词 Ind 经过状态转移后的变量的状态。定理2-1表明归纳不变式在初始状态下成立；定理2-2表明归纳不变式在状态转移后依然成立，具有归纳性质。比如说，如果 Ind 在状态 s 下成立，那么在 s 的后继状态下，Ind 依然成立；定理2-3表明归纳不变式蕴含安全属性，因此，如果某个运行时可达状态满足 Ind, 那么也必然满足 $Safe$。这是我们寻找一个这样的归纳不变式的目的，通过归纳不变式的正确性验证安全属性的正确性。这是归纳不变式所必须满足的三个条件。

$$A_1 \triangleq \forall s \in Server : \forall c \in Client : locked[s] \Rightarrow (s \notin held[c])\big) \tag{2-4}$$

$$Ind \triangleq Safe \wedge A_1 \tag{2-5}$$

对于 *Client_Server* 规约，表达式2-5是一个可能的归纳不变式。可以看到的是，Ind 是由 $Safe$ 和 A_1 两个谓词逻辑表达式合取组成而来。事实上，大部分规约的归纳不变式都可以表达为 $Ind \triangleq Safe \wedge A_1 \wedge A_2 \wedge ... \wedge A_n$ 的形式。其中合取子式 A_k 是约束状态的谓词，我们将之称为引理不变式（Lemma Invariant）。因为归纳不变式 Ind 是由这些引理不变式 A_k 组合而成的，也就是说，归纳不变式强于每一个引理不变式。因此，引理不变式需要满足不变性，也就是在系统运行的每个状态下都成立，才能成为一个合适的引理不变式。但是，引理不变式本身不需要满足归纳性，只需要它们和安全属性的合取结果能够满足归纳性。

对于一个谓词表达式 P，如果一个状态 s 满足 $s \vDash P$，但是 s 的后继状态 $s_n \vDash \neg P$，那便可以称 s 为 P 的归纳反例（counterexample），揭示了 P 不是归纳

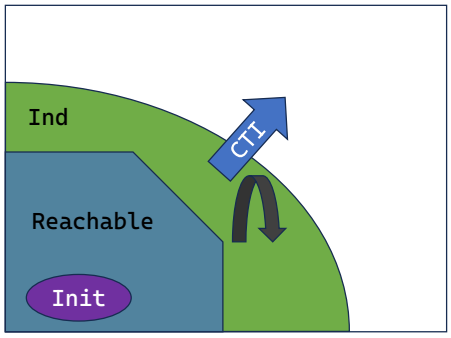


图 2-2　归纳不变式和归纳反例

不变式。一个归纳反例往往包括两个状态，前一个状态满足谓词 P，而后一个状态不满足谓词 P。

图2-2形象地介绍了归纳反例以及归纳不变式、运行时可达空间和初始化空间之间的关系。寻找归纳不变式的过程，也可以理解为通过添加新的引理不变式作为约束修剪空间，来排除归纳反例的过程。但是，尤其是对于越复杂的系统而言，寻找归纳不变式并不是一个简单的任务。实现归纳不变式的自动生成是形式化验证领域一个重要的研究目标，这也是本文研究的内容。

2.3　强化学习

强化学习（Reinforcement learning, RL）[28]是机器学习的一个领域，强调如何基于外部环境做出决策，以获得最大化的预期累积奖励。是区别于监督学习和非监督学习的另外一种基本的机器学习方法。强化学习的关注点在于寻找对未知领域的探索和对已有知识的利用之间的平衡。它的目标是通过奖惩来控制智能体完成任务，以获得最大化的预期累积奖励，但程序无需明确告诉智能体如何完成任务。

在机器学习问题中，环境通常被抽象为马尔可夫决策过程（Markov decision processes，MDP）[29]，因为很多强化学习算法在这种假设下才能使用动态规划的方法。但是，强化学习相较于动态规划，并不一定需要了解 MDP 的具体信息和全局信息，只需要通过与环境的交互，不断试错来学习。

图 2-3 展示了强化学习的框架。在强化学习中，核心在于智能体（agent）与环境（environment）之间的交互。环境是智能体所在的背景，它会根据智能体的动作给予奖励或惩罚，并做出状态的转移。智能体能够感知环境的状态（State），并

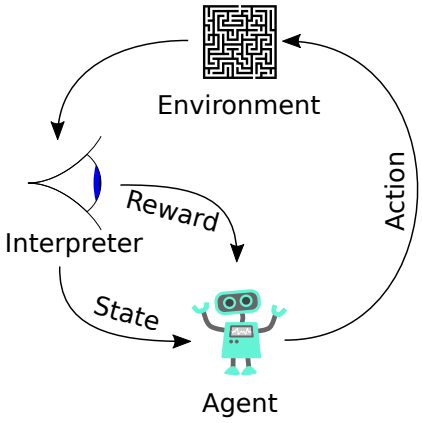


图 2-3　典型强化学习框架

根据反馈的奖励（Reward）或惩罚（Reward 为负值），来调整自身策略（Policy），学习选择适当的动作（Action），以最大化长期总收益。通过不断与环境互动，智能体根据环境的反馈不断调整策略，以期获得最大化的预期累积奖励。实现强化学习的策略算法有很多，其中最著名的有 Deep Q Network（DQN）[30]，DDPG[31]等。

在本文的项目中，我们使用强化学习的方法来加速归纳不变式的生成。我们使智能体理解 TLA$^+$ 原文件的内容，让智能体合理选择生成归纳不变式的种子（seed），并将每一次智能体选择的种子所生成的不变式的检验结果反馈给智能体，包括反例的数量，内容和生成时间等。智能体根据这些反馈信息，调整自己的策略，以便更快地找到一个满足安全属性的归纳不变式。

第三章　归纳不变式自动生成工具的设计

本章节将介绍 RL-TLA 的具体设计，每个模块的职责和功能。总体上，工具需要完成一下工作:

- 解析用户输入，以合适形式的存储并预备后续使用；
- 对生成模块所生成的候选不变式检验其正确性、独立性和与候选的归纳不变式的递归性，在检查递归性时还需要生成归纳反例，回传给生成模块；
- 生成模块，基于用户输入和检验模块的结果，生成合适的不变式;
- 其他非功能模块，包括日志模块，计时模块等；

系统的输出是对一系列引理不变式的合取范式，对于规约而言，是一个包含 $Safe$ 属性的归纳不变式。

3.1　总体设计

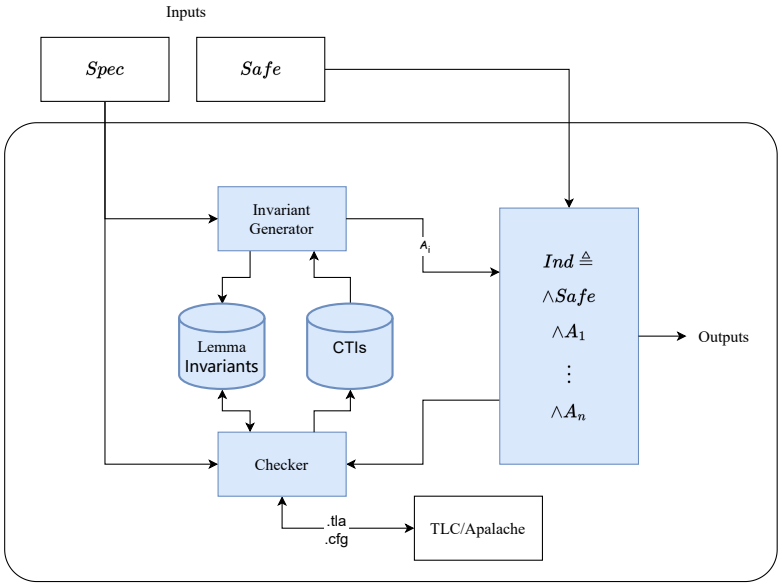


图 3-1　RL-TLA 工作流图

RL-TLA 的工作流如图3-1所示，功能模块主要分为候选不变式生成模块（Invariant Generator），候选不变式检验模块（TLC/Apalache）两个部分。其中候选

不变式生成模块接入了强化学习，训练强化模型智能体，以提高候选不变式的生成效率和准确率。候选不变式检验模块则接入了 TLC 或 Apalache，对生成的候选不变式进行验证，检验其正确性，独立性和候选归纳不变式的递归性，并将结果返回给生成模块。在检验不变式的正确性和归纳不变式的递归性出现错误时，应当返回对应的不变式反例和归纳反例。目前系统接受 endive 提供的数据源，使用 endive 中的对规约人工标记的谓词，作为生成候选不变式的种子（seed，图5-1中 preds 字段）。

Algorithm 1: RL-TLA 的工作流

Input: M: Finite instance of parameterized system
Input: $Safe$: Safety property
Output: Ind: Inductive Invariant Candidiate

```
1  Ind ←Safe;
2  CTIs ←GenIndCTIs(M, Ind) ;
3  CEs ←∅ ;
4  while CTIs ≠∅ do
5      Inv ←GenNextInv(M, Ind, CTIs, CEs) ;
6      CEs ←GenInvCEs(M, Ind) ;
7      if CEs =∅ then
8          if EliminateCTIs(M, Inv, CTIs) then
9              Ind ←Ind ∧ Inv;
10             CTIs ← GenIndCTIs(M, Ind) ;
11 return Ind
```

基本的逻辑结构如算法1展示。初始化时候，候选的归纳不变式首先合取 $Safe$ 属性，然后基于此生成第一轮的归纳反例（Counterexample to Induction）。在每一轮的迭代中，强化学习系统都会不断地生成一个个候选的引理不变式。但只有满足两个条件的，才能被添加到候选归纳不变式中。首先是，引理不变式必须，满足在运行的状态空间上是不变式，也就是没有不变式反例 CE 生成；其次，新的引理不变式必须不能被已有的不变式包含，因为加入这样的引理不变式不会提高候选归纳不变式的约束能力。这样的不变式便会加入到候选的归纳不变式中做合取，直到对候选归纳不变式生成的 CTI 为空，即不再有归纳反例产生。这说明现在系统中所有生成不变式的合取结果是一个包含 $Safe$ 属性归纳不变式，也就是得到了归纳不变式，系统选择将这个结果输出给用户。

关于不变式反例 CE 的定义大致如下:

$$Spec \triangleq Init \wedge [Next] \tag{3-1}$$

$$Spec \vDash (s, s') \tag{3-2}$$

$$s \vDash Inv \tag{3-3}$$

$$s' \nvDash Inv \tag{3-4}$$

和归纳反例不同的是，状态 s 和 s' 都来自规约允许的允许状态，也就是图2-2中可到达的允许空间里的状态。针对一个谓词表达式，只有不存在这样的 CE，才能被称为不变式。

因为逻辑复杂，运行时间长，为了开发人员和用户对系统信息可见，需要设计有日志模块，以不同的日志等级记录系统的运行状态。日志模块应当在系统运行的每个重要步骤打印具体的运行信息，同时在系统出现运行错误时，打印错误信息，以便开发人员定位错误进行调试。

为了研究和评估系统的性能，需要设计计时模块，记录系统运行的时间，以便于后续的性能分析。同时，需要在循环中记录强化学习模块的特征信息。

3.2 候选不变式检验模块

候选不变式检验模块需要调用模型检查器，并且需要将输出解析，并将结果返回给生成模块。检验模块的工作，可以分为三个部分，分别是检验新生成候选不变式正确性、独立性和多个不变式合取结果的递归性。

引理3-5表达了候选不变式的正确性，即候选不变式在规约有限实例运行的每个状态下都成立。引理3-6表达了候选不变式的独立性，即新生成的候选不变式不能被已有的不变式的合取结果包含。

$$Spec \vDash Inv \tag{3-5}$$

$$IndCand \wedge Next \nvDash Inv \tag{3-6}$$

检验候选不变式的正确性，就是检验每个候选不变式是否在规约的每个状态下，布尔值都为真。这个问题虽然简单，但是直觉上可能需要遍历许多状态和

多个状态转移轨迹，可能需要很长的时间。但是实际上，很多的不正确的候选不变式在规约的某个比较容易到达的状态被验证器检验出来，便可以退出了。一个正确的不变式，尽管现在还不能成为归纳不变式，但是它确实我们需要归纳不变式的开始。

检验不变式的独立性时，我们需要验证新生成的候选不变式是否能被已有的不变式的合取结果包含。检验不变式的独立性十分重要，尤其是对于指导强化学习生成候选不变式，因为允许这样，那么生成模块为了得到更高的奖励，会偏向于生成这样的候选不变式，这样会导致不变式的重复，合取结果的约束能力也不能加强，对状态空间不能做出有价值的修剪，这样的不变式便是没有意义的。系统也就无法找到一个合适的归纳不变式。一个不被已有不变式包含的候选不变式，便可以成为一个合适的引理不变式。

检验归纳不变式的递归性，是我们工作的终点。如果多个不变式的合取结果具有递归性，那么我们可以将这个合取结果作为归纳不变式，我们可以将这个结果输出，并退出。对于不变式的递归性质，在引理2-1和2-2中已经提及。归纳不变式需要包含所有的初始状态，并且，从归纳不变式约束的状态出发，进行状态转移，新生成的状态也需要满足归纳不变式。另外，我们最根本的目的是证明安全属性，所有归纳不变式需要蕴含安全属性。

如果给出的候选不变式不正确，检验模块应当给出不变式反例，以帮助生成模块调整策略，生成正确的候选不变式。同样的，如果当前给出的引理不变式的合取结果不是归纳不变式，检验模块还应该返回归纳反例。不变式反例和归纳反例有着相似的内容，由两个状态组成，前一个状态满足现有的候选（归纳）不变式，而后一个状态则不满足。

3.3 候选不变式生成模块

生成模块可以分候选不变式的语法生成的逻辑，以及强化学习框架在生成模块中的应用，两个部分。

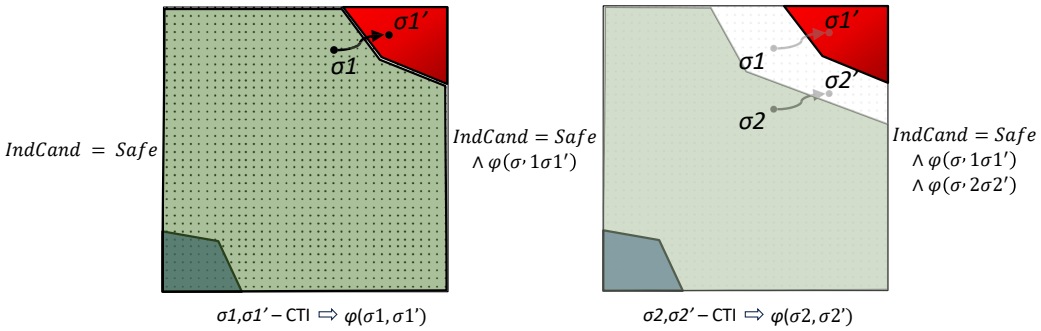


图 3-2　杀死归纳反例以获得归纳不变式

3.3.1　候选不变式的语法与生成逻辑

图3-2是一个典型的杀死归纳反例以获得归纳不变式的过程。候选的归纳不变式初始化为 $Safe$ 属性，然后基于此生成第一轮的归纳反例（Counterexample to Induction）。以杀死归纳反例为目的，生成模块不断地生成一个个候选的引理不变子式。以此迭代，直到没有归纳反例产生，也就是获得了最后的归纳不变式。因此，归纳反例对生成归纳不变式有重要的指导作用。

尽管接入了强化学习，生成模块的工作形式和传统的归纳不变式生成工作流程类似。引理不变式以一阶逻辑谓词的形式存在，首先需要满足在规约的有限实例上保持布尔值为真，其次便是需要杀死一些归纳反例。为了找到这些引理不变式，我们使用一种基于语法合成指导的归纳不变式生成技术。我们从输入的种子（seed）谓词中，按照强化学习模块的意愿选择一部分种子谓词，并按照语法组合成一个可能的不变式。每个种子谓词都是针对系统状态变量的布尔表达式。当然，我们也有可能采取给出种子谓词的否定，根据 TLA$^+$ 的语法，我们只需要在给出的谓词前面加上否定符号“~”即可。这部分的决定权交给强化学习模块，强化学习模块会根据当前的状态，选择一个合适的种子谓词，或者它的否定。

一个不变式的语法大致可以表达为：

$$< lemma > :=< quans >:< expr > \tag{3-7}$$

$$< quant > := \forall x \in SetA \mid \exists y \in SetB \tag{3-8}$$

$$< expr > :=< seed > \mid \sim< seed > \mid < seed > \vee < seed > \tag{3-9}$$

注意到，我们对于引理不变式内部每个谓词之间的连接方式统一选择了析取符

号“∨”。这是因为，一方面这种语法在表达归纳不变已经足够完备，在每个引理不变式内部使用析取计算，对引理不变式之间选择合取计算足够我们表达对于绝大多数规约的归纳不变式；另一方面，由于简化了种子谓词之间的逻辑计算方式，不再需要对此分别研究，也可以帮助了我们简化问题，缩小了搜索不变式的空间。另外，对于每个引理不变式的量词部分，我们统一使用了直接的输入，不做其他更改。这部分量词，往往和后面跟随的谓词逻辑表达式有关系，而谓词逻辑表达式的种子谓词又来自于用户输入。因此，直接使用和种子谓词匹配好的量词表达式，避免了许多无意义的候选不变式，方便了对候选不变式的生成。尽管有些时候，可能量词表达式中定义的变量在后面的谓词逻辑表达式中并没有使用，但这不影响生成的结果的布尔值，对于这部分多余的量词定义，可以不做处理。

3.3.2 强化学习在生成模块中的应用

本文的目的是希望研究强化学习在自动的归纳不变式生成过程中的应用。强化学习是一种通过智能体和环境的交互，智能体通过感知环境的状态的转移，采取行动，获得奖励，来学习如何在环境中获取最大的奖励。对于归纳不变式生成问题而言，智能体可以根据环境的变化，连续地进行动作选择的，并根据环境返回的奖励值，来调整策略的特点，可以加速归纳不变式的生成过程。

强化学习模块接受 TLA^+ 协议和检验模块对于本身生成的候选不变式的检验结果，修改策略，生成更加合理的候选不变式。强化学习模块的输入是一个状态，输出是一个动作，动作是对于种子谓词的选择与否，和是否选择它的否定。

强化学习模块的目标是生成一个合适的引理不变式，这个引理不变式在规约的每个状态下都成立，且不会被已有的不变式的合取结果包含。这个过程是自动化的，但人可以通过调整对强化学习智能体在每个状态下的每个动作的选择给出合适的奖励或者惩罚，以指导智能体学习到一个合适的策略并不断调整，以应用到后续的候选不变式生成中。

环境对智能体的动作给出反馈，包括奖惩值和自身的状态信息。

最高的奖励应当给予给出最终的归纳不变式的行为，也就是给出了一个不变式，使得它和前面所有不变式的合取结果是归纳不变式的行为。当强化学习模块给出一个不变式的时候，也应当给出一定的奖励。给出一个不变式，尤其是

给出第一个不变式，往往是一个十分困难的过程。而且，这一步也是实现最终目标的关键。因此，我们应当给予一定的奖励，以鼓励智能体继续学习。生成一个已经被已有不变式包含的不变式，尽管没有意义，但也体现出智能体如何寻找不变式的能力，因此也应当给予一定的奖励。这个奖励的值很小，但是也是必要的。因为这个过程是一个逐步的过程，智能体需要不断地尝试，才能找到一个合适的不变式。但是，为了防止智能体的惰性，我们不允许智能体在已有的不变式上简单的析取上一个谓词，然后给出这个谓词作为候选不变式，同样的，不允许在一个错误的不变式上去除掉一些谓词，以得到候选不变式。这是简单的重复，是没有意义的，我们对智能体的这种行为应当给予惩罚。

环境的状态信息包括候选归纳不变式中已有的引理不变式，归纳反例，种子谓词，以及规约的状态信息，在强化学习给出的候选不变式不正确时，也应当给出在哪些状态下，这个候选不变式的布尔值为假，以便于智能体基于错误的候选不变式做出调整。

第四章　归纳不变式自动生成工具的实现

本章节将基于设计方案，详细介绍了归纳不变式自动生成工具的实现细节，包括模块之间的交互以及模块的具体实现。

4.1　总体实现

主要包括输入模块、候选不变式检验模块、候选不变式生成模块和非功能模块。

输入模块主要负责接收用户传入的 TLA^+ 规约文件，配置信息等，将这些信息存储以供后续使用。对 TLA^+ 规约文件，利用 tla2sany 工具将其转换为抽象语法树，并通过遍历抽象语法树，获取规约文件中常量、变量和谓词的定义。

非功能模块包括日志功能，计时功能，报错信息等，这些功能伴随着系统每个行为，为开发人员和用户提供更多信息，方便调试和使用。日志模块需要以不同的记录级别记录系统运行的行为和结果，尤其是对于 TLC 和 Apalache 的调用的结果，因为是外部调用，需要格外关注。计时模块需要记录系统运行的总时间和每个部分分别运用的时间。

4.2　候选不变式检验模块

候选不变式检验模块主要的职责是对生成模块的生成的候选不变式的正确性，给出的候选归纳不变式的递归性，以及对新生成不变式的独立性进行判断。

候选不变式检验模块接入了 TLC 和 Apalache，用户可以选择其一对生成的候选不变式进行验证。TLC 和 Apalache 是两个常见的面向 TLA^+ 规约的模型检查工具 (model checker)，可以使用相似的配置文件对规约进行验证，但是，两者的结果输出格式不同，需要做分别处理。

由于 Apalache 需要用户对协议中的变量和常量做出类型的注释，因此，目前能够提供的测试集中大多数的规约都无法使用 Apalache 进行验证。在系统实

```
INIT Init
NEXT Next

INVARIANTS Inv_0 Inv_1 Inv_2

CONSTANTS
Node = {n1,n2,n3}
Request = {r1,r2}\nResponse={p1,p2}
n1 = n1
n2 = n2
n3 = n3
r1 = r1
r2 = r2
p1 = p1
p2 = p2
```

图 4-1　TLC 配置文件

现时，我们默认状态下使用 TLC 作为系统的模型检查器。当然，在条件允许时，用户可以设置系统的参数来使用 Apalache 作为系统的模型检查器。

在检验候选的不变式和归纳不变式时，如果出现反例，无论是对候选不变式的反例，还是对归纳不变式的归纳反例，都应该提取出来，返回给生成模块。

4.2.1　model checker 的配置文件和运行选项

对于图2-1中的规约，TLC 和 Apalache 会使用默认的配置文件进行验证，即以 INIT 为初始状态，NEXT 为状态转移关系，在状态变化的过程中验证 Safe 安全属性的正确性。用户也可以指定使用其他配置文件，以验证从不同状态出发和不同状态转移条件下的用户定义的不变式的成立与否。比如说需要验证的不变式，可以放在 INVARIANT 字段下。对于一些常量，用户也可以通过配置文件 CONSTANTS 字段进行定义。一个典型的 TLC 配置文件如图4-1所示。我们可以简单的理解为，模型检测器可以判断 $\text{INIT} \wedge \text{NEXT} \vDash \text{INVARIANT}$ 是否成立。在不成立时，模型检测器会给出一个状态的链接，展示系统状态如何从初始状态转移到不满足不变式的状态。

我们希望 TLC 和 Apalache 为我们验证生成模块生成的候选不变式的正确性，独立性和与已有不变式合取结果的递归性。在验证过程中，我们希望模型检查器能够输出验证结果，以及验证过程中的反例（Counterexample）。

验证不变式的正确性是验证这三种性质中最为简单的。只需要将需要验证的候选不变式放入 INVARIANT 字段中，然后运行模型检查器即可。如果没有报

错，说明候选不变式在规约的有限实例上保持布尔值为真。

由于候选的归纳不变式是由一系列引理不变式和安全属性合取而来，且每一个合取子式都满足 Init ∧ Next ⊨ Lemma，所以候选归纳不变式自然满足引理2-1和2-3。验证候选的归纳不变式的递归性质时，我们只需要验证引理2-2的正确性，我们需要验证归纳不变式在状态转移后依然成立。在此过程中，模型检查器弹出的报错就是归纳反例。

验证不变式的独立性是验证这三种性质中最为困难的。检查不变式的独立性就是检查新生成的引理不变式是否能杀死 (eliminate) 候选的归纳不变式的归纳反例。借助 TLC，我们将系统的初始状态设置为这些归纳反例的状态之一，也就是将 INIT 设置为这些归纳反例状态的析取。然后看在这些状态下，哪些新生成的引理不变式不成立。不成立便能代表它们能够杀死对应的归纳反例。在操作中，我们需要用变量记录下这些引理不变式的值。使用 TLC 的**"-dump"** 选项将每个状态下的变量值输出到文件中，然后解析这些文件，找到能够杀死归纳反例的引理不变式。如果归纳反例太多，会导致 TLC 计算状态转移关系的时间过长，状态转移图过于复杂。尽管可以通过**"-workers"** 选项添加 TLC 调用的线程数量，但是也收效甚微。因此将归纳反例分组，分别使用一个线程对每一组归纳反例进行验证，可以有效减少验证时间。

但是这种做法并不充分，还需要通过 IndCand ∧ Next ⊭ Lemma 的结果来确认新引理不变式的独立性。

我们需要将新生成的不变式放到一个新的文件中，并使用关键字 **EXTENDS** 将原有规约中的定义引入。这样，我们就可以在新的文件中使用原有规约中的定义，和引入生成模块生成的候选不变式。由于新的 TLA^+ 文件中有着相似的结构，在验证同一个性质时，配置文件是可以复用的。所以我们在系统运行之初就定义好配置文件中的内容，并写入硬盘供 TLC/Apalache 使用。

在使用 TLC 验证时，我们还需要关注诸多选项。**"-config"** 是我多样化使用 TLC 和 apalache 的关键，通过这个选项，我们可以指定 TLC 和 Apalache 的配置文件，以检验不变式的不同性质。**"-deadlock"** 选项用于检查是否存在死锁状态，如果选择了这个选项，那么 TLC 就不会检验死锁。由于我们的目的是检验不变式的一些性质，所以我们不需要检验死锁，并选择了这一选项。**"-continue"** 选项揭示了 TLC 在检测出错误后是否继续运行，为了得到更多的反例，我们需要

使用这个选项。

4.2.2 model checker 的调用和结果解析

本项目的代码主要基于 Python 实现，然而不论是 TLC 还是 Apalache，都是 Java 实现的模型检查器，且没有可以直接调用的 Python 接口。因此，我们需要通过 Python 的 subprocess 库来调用 Java 程序，并通过解析 Java 程序的命令行输出结果来获取验证结果。在验证不同性质的时候指定好不同的配置文件并调整好不用的运行参数。

对于结果的解析，主要是将 TLC 或 Apalache 的输出结果进行解析，去除无用的信息，将有用的信息交给生成模块，以便强化学习模块调整策略，提高生成的候选不变式的正确性。TLC 和 Apalache 尽管两者有着不同的输出格式，但是它们的功能其实是一致的，都是将出现不变式错误时的状态，以及前序状态，也就是错误轨迹 (error trace)。错误轨迹的每一个节点都是一个状态，表达的是在这个状态下，各个变量的值。TLC 会以析取范式的形式将各个变量的值表达出来，而 Apalache 默认使用的 json 文件格式，将各个变量的值以键值对的形式表达出来。我们需要将这些信息解析出来，以便强化学习模块能够理解这些信息，调整生成的候选不变式。

不变式反例和归纳反例有着相似的作用，都是用于提示用户或者生成模块，哪些状态下，不变式或者归纳不变式不成立。

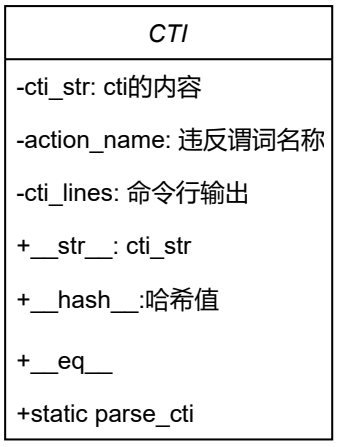


图 4-2　CTI 类信息

类 CTI 的信息如图4-2。在模型检测器检测去不变式或者归纳不变式的错误时，通过类 CTI 中的静态方法 **parse_cti**，可以将错误轨迹提取出来，生成多个

```
Simulation using seed 4257 and aril 0
Error: Invariant InvStrengthened is violated.
Error: The behavior up to this point is:
State 1: <Initial predicate>
/\ request_sent = {<<n2, r1>>, <<n3, r2>>}
/\ match = {<<r1, p1>>, <<r1, p2>>, <<r2, p1>>, <<r2, p2>>}
/\ response_received = {<<n2, p2>>}
/\ response_sent = {<<n1, p1>>, <<n1, p2>>, <<n2, p1>>, <<n3, p1>>}

State 2: <ReceiveResponse line 31, col 5 to line 33, col 53 of module client_server_ae>
/\ request_sent = {<<n2, r1>>, <<n3, r2>>}
/\ match = {<<r1, p1>>, <<r1, p2>>, <<r2, p1>>, <<r2, p2>>}
/\ response_received = {<<n1, p2>>, <<n2, p2>>}
/\ response_sent = {<<n1, p1>>, <<n1, p2>>, <<n2, p1>>, <<n3, p1>>}
```

图 4-3　反例的命令行输出

不同的 cti 对象。

CTI 类反应的是一个状态，在这个状态下，所有变量的值存储在 *cti_str* 字段中。从这个状态出发，系统可以运行一个状态，这个状态下，给出的候选归纳不变式不成立。一个 TLC 输出的错误轨迹如图4-3。其中 (State 1, State 2) 就是图2-2的一个归纳反例，CTI 类存储的是 State 1 的状态信息，cti.action_name 对应于 ReceiveResponse, cti.cti_str 存储的是 State 1 的状态信息。

CE 类有着和 CTI 类相似的结构，不同的是，CE 类更关注那个使得不变式不成立的状态。CE 类对应的状态都是系统正常运行时可达的状态，而 CTI 类对应的状态是则全部不是系统正常运行时可达的状态。

4.3　候选不变式生成模块

生成模块是本项目的关键，它负责生成候选不变式，检验模块是为生成模块服务的。不同于以往的归纳不变式生成工具，使用随机枚举的方式生成候选不变式，我们引入强化学习来提高我们枚举的效率和成功率。强化学习是一种通过智能体和环境的交互，智能体通过观察环境的状态，采取行动，获得奖励，来学习如何在环境中获取最大的奖励。强化学习的智能体通过观察检验模块对已经生成的候选不变式和候选归纳不变式的验证结果，调整生成候选不变式的策略，提高生成候选不变式的效率和准确率。

强化学习框架是本项目的核心，它负责生成候选不变式，检验模块是为生成模块服务的。RL-TLA 核心逻辑是通过提供必要的信息和反馈，让强化学习智能

体选择合适的种子，将它们组合起来，生成引理不变式的候选。一个强化学习框架可以分为环境和智能体两个部分，分别是环境和智能体。图4-4简单地介绍了本文中的强化学习框架，描述了智能体和环境交互，不断生成候选引理不变式的大致过程。

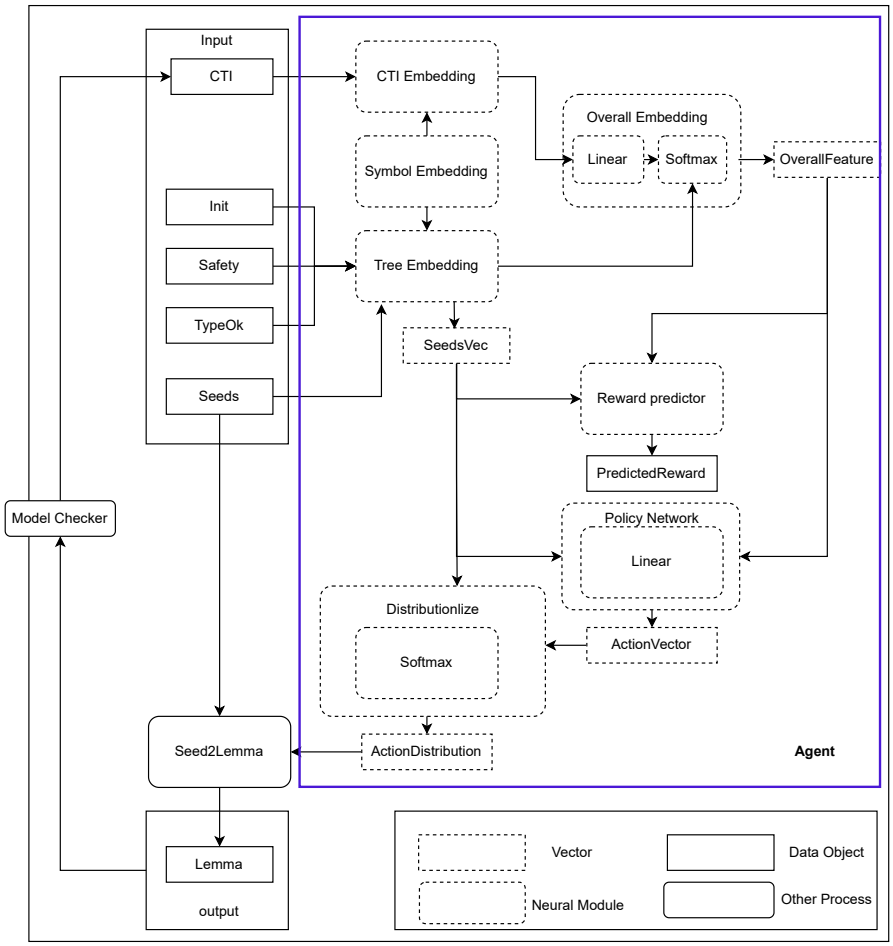


图 4-4　RL-TLA 强化学习框架

4.3.1　强化学习环境

环境模型是环境在强化学习中的数学表示，包含了环境中的重要信息，和对智能体的动作做出反馈。本文中，环境给智能体提供的内容包括输入部分，也就是 TLA^+ 规约本身和用于提供给智能体选择的种子、以及在每轮循环中的反馈，即反例和归纳反例。这些信息足够作为智能体生成候选不变式的提示。

对于反例和归纳反例的介绍，我们在候选不变式检验模块中已经有所介绍，这里不再赘述。

输入部分，主要目的是存储和必要的数据和约束强化学习智能体的行为，使得其所能采取的行为能生成的候选不变式都必须符合基本的语法。

输入部分包括 TLA^+ 规约文件，以及以 json 形式存储的对规约的配置文件。TLA^+ 规约文件是生成归纳不变式的基础和目标，规约文件和相关的配置使 RL-TLA 能够运行一个有限的实例。基于有限实例的运行数据，也就是各个可达的状态信息，RL-TLA 可以以此为依据生成引理不变式和组成归纳不变式。对图2-1中的规约，关于配置文件的具体信息，我们在第五有具体展示。配置文件中包括生成规约有限实例的参数，主要包括常量的范围和对常量的定义，类型限制 (typeok) 和安全属性 (Safety)。preds 字段下是生成候选不变式的种子 (seed)。quants 字段下存储的是默认的量词。生成的候选不变式就是按照语法规则，组合种子和量词，生成符合语法的候选不变式。

环境部分还包括对智能体动作的奖惩机制，奖惩机制是智能体在环境中学习的重要手段。基于智能体给出的不同的不变式，环境需要给出不同的奖惩。表4-1给出了具体的数值设置。

表 4-1　奖励与惩罚的情况设置

类型	程度	情况	取值
奖励	very	通过归纳不变式的检测	101
奖励	little	通过蕴含检测但没能通过归纳不变式检测	消除的 CTI 数量 (被归一化到区间 [1,100])
惩罚	little	没有通过蕴含检测	-1
惩罚	medium	不变式检测超时	-5
惩罚	very	没有通过不变式检测	违反的状态数量 (被归一化到区间 [-100,-6])

4.3.2　强化学习智能体

智能体是强化学习的核心组件，它是学习和决策的主体。通过与环境的交互，智能体不断调整自己的策略，以期获得最大的奖励。这个过程中有三个主要步骤：感知环境的状态，选择行动，获得奖励并更新策略。

智能体通过环境中状态的数据来感知环境，这些数据包括规约信息，种子信息和当前的 CTI 信息。在接受这些数据前，需要先对数据进行预处理，将数据从高维或稀疏数据转换为低维、密集向量。这个动作叫做数据嵌入，是强化学习

中的一个重要步骤，可以简化数据结构，使其在机器学习模型中更易于处理和分析。嵌入的目的在于捕捉数据的语义或结构信息，同时减少数据的维度和复杂度。本文在此选择的是 LSTM(Long Short-Term Memory) 的嵌入方法，LSTM 是一种特殊的递归神经网络。普通 RNN 在处理长序列数据时存在的梯度消失和梯度爆炸问题，而 LSTM 设计用来解决这样的问题，因此适合作为数据嵌入的方法。

选择行动则表示的是智能体在当前状态下，根据策略选择行动的过程。在本文中，智能体的行动是选择种子，将种子组合成候选不变式。智能体的策略表示智能体在不同的状态下，选择不同行动的概率，策略选择的结果直接指导了智能体的行为。值函数用于评估状态-动作对的“价值”，即从该状态-动作对开始，智能体在未来期望获得的累积奖励。简而言之，值函数是用于评估智能体在某一个状态下采取某一个行动的好坏。

PolicyNetwork 和 *RewardPredictor* 是具体实现策略和值函数的网络模型。*PolicyNetwork* 的输入是系统的状态，输出是动作的概率分布。它接受了环境的总体特征和 seeds 的嵌入，获得 action_vector，并在 *Distributionlize* 中进行归一化处理，得到概率分布 action_distribution，以此作为智能体选择种子的指导。*RewardPredictor* 在奖励稀疏时十分有用，它将输入的状态向量和整体特征向量进行拼接，经过线性层逐层地提取特征，并把预测的奖励值投影到奖励值范围中，输出 reward 值。在智能体采取行动后，奖励预测器还会根据实际奖励更新自己的参数，以提高奖励预测的准确性。本文的奖励值范围是 [-10, 10]。

action_vector 对应于对种子的选择。强化学习智能体选择好种子，然后交给 Seed2Lemma 模块，组合上量词，从而生成候选不变式。每个种子都会取其本身和取其否定并组成多个候选不变式。这里不需要智能体选择种子或其否定，主要为了降低智能体处理问题的难度。

$$\text{StrictLoss} = -\frac{1}{N}\sum_{i=1}^{N} log\_softmax(\text{action\_distribution}) \cdot \text{sd}_i \cdot \gamma \tag{4-1}$$

$$\text{r} = \left[\text{f} \cdot \text{discounter}^{n-1}, \cdots, \text{f} \cdot \text{discounter}^{1}, \text{f} \cdot \text{discounter}^{0}\right] \tag{4-2}$$

$$\text{CELoss}_i = -q_i \log(p_i) \tag{4-3}$$

$$\text{RewardLoss} = \frac{1}{N}\sum_{i=1}^{N}(\text{r}_i - \text{r}_{i-1}) \cdot \text{CELoss}_i \tag{4-4}$$

$$\text{MSELoss} = \frac{1}{N}\sum_{i=1}^{N}(\hat{r}_i - r_i)^2 \tag{4-5}$$

$$\text{Total Loss} = \text{Strict Loss} + \text{RewardLoss} + \text{MSELoss} \tag{4-6}$$

智能体在采取每个行动后，除了得到环境的状态信息，包括反例或者归纳反例和候选归纳不变式的状态，还会得到奖惩值和调整系数，这两个参数用于调整智能体的策略。智能体会计算每轮采取的行动的损失值，并通过 *backward* 方法向后传播给模型，更新模型的参数。损失值有三个度量维度，分别为严格性损失 (Strict Loss, 4-1)，预测损失 (Prediction Loss, 4-4) 和均方损失 (MSE Loss, 4-5)，其总和为总损失4-6。表达式中，N 表示每次选择的种子数量 sd_i 表示每个种子被选择的概率分布。环境的反馈中给出了调整系数 γ，调整系数是用于调整严格性损失在总体损失中的占比，f 则是环境给出的具体奖惩值。p 和 q 分别为 action_distribution 的输出概率分布和目标概率分布。三个维度的损失组成总体损失，总体损失反应了智能体做出的决策优劣，指导做出下一轮决策。

第五章　运行试验和结果分析

5.1　运行结果

以 Client_Server 规约为例，一个典型的配置文件如下：

```
{
    "preds" : [
        "<<VARR,VARP>> \\in match",
        "<<VARI,VARR>> \\in request_sent",
        "<<VARJ,VARR>> \\in request_sent",
        "<<VARI,VARP>> \\in response_sent",
        "<<VARJ,VARP>> \\in response_sent",
        "<<VARI,VARP>> \\in response_received",
        "<<VARJ,VARP>> \\in response_received",
        "VARI=VARJ /\\ match = match",
        "ResponseMatched(VARI,VARP)"
    ],
    "preds_alt" : [],
    "safety" : "Safety",
    "constants" : "CONSTANT\nNode = {n1,n2,n3}\nRequest =
     {r1,r2}\nResponse={p1,p2}\nn1 = n1\nn2 = n2\nn3 = n3\nr1 = r1\nr2 = r2\np1 = p1\np2
     = p2\n",
    "constraint" : "",
    "quant_inv" : "\\A VARI \\in Node : \\A VARJ \\in Node : \\A VARR \\in Request : \\A
     VARP \\in Response : ",
    "quant_inv_alt" : null,
    "quant_vars": [],
    "model_consts" : "CONSTANT n1,n2,n3,r1,r2,p1,p2",
    "symmetry" : true,
    "typeok" : "TypeOK",
    "simulate" : true
}
```

图 5-1　Client_Server 规约配置文件

RL-TLA 和 endive 使用相同的配置文件。

本文得到的运行数据，包括对 endive 的运行数据的机器配置如图5-1

- CPU: AMD Ryzen 9 5950X Processor 16C32T @4.6GHz
- 内存: 128GB DDR4
- 操作系统: Ubuntu 22.04
- 显卡: Tesla V100-32GB
- TLC 版本: 2.15

运行结果数据如表5-1展示，其中数据取三次运行时长中位数的结果，内存数据采用 Memray 工具的最大内存占用值。

表 5-1　RL-TLA 和 endive 运行结果对比

规约名称	RL-TLA			endive		
	耗时/sec	内存占用/GB	lemma 数量	耗时/sec	内存占用/MB	lemma 数量
TwoPhase	25.49	13.54	13	30.02	120.04	9
client_server_ae	100.38	12.61	2	45.13	85.49	1
simpele_election	65.68	12.39	6	19.26	61	3
learning_switch_i4	93.34	12.12	1	147.07	124	1
consensus_epr	900.22	14.46	7	513.83	196	7
sharded_kv	164.73	15.03	9	181.04	222.4	5

MODULE *TwoPhaseInd*

EXTENDS *TwoPhase*

$$
\begin{array}{l}
IndAuto \triangleq \\
\quad \land\ TCConsistent \\
\quad \land\ \forall\, rmi \in RM : \forall\, rmj \in RM : \\
\qquad \lor\ ([type \mapsto \text{“Commit”}] \in msgs) \\
\qquad \lor\ \neg(rmState[rmi] = \text{“committed”}) \\
\quad \land\ \forall\, rmi \in RM : \forall\, rmj \in RM : \\
\qquad \lor\ (tmPrepared = tmPrepared \cup \{rmi\}) \\
\qquad \lor\ \neg([type \mapsto \text{“Commit”}] \in msgs) \\
\quad \land\ \forall\, rmi \in RM : \forall\, rmj \in RM : \\
\qquad \lor\ \neg([type \mapsto \text{“Commit”}] \in msgs) \\
\qquad \lor\ \neg(rmState[rmj] = \text{“working”}) \\
\quad \land\ \forall\, rmi \in RM : \forall\, rmj \in RM : \\
\qquad \lor\ ([type \mapsto \text{“Prepared”}, rm \mapsto rmi] \in msgs) \\
\qquad \lor\ \neg([type \mapsto \text{“Commit”}] \in msgs) \\
\quad \land\ \forall\, rmi \in RM : \forall\, rmj \in RM : \\
\qquad \lor\ ([type \mapsto \text{“Prepared”}, rm \mapsto rmi] \in msgs) \\
\qquad \lor\ \neg(tmPrepared = tmPrepared \cup \{rmi\}) \\
\quad \land\ \forall\, rmi \in RM : \forall\, rmj \in RM : \\
\qquad \lor\ (rmState[rmj] = \text{“committed”}) \lor (rmState[rmj] = \text{“prepared”}) \\
\qquad \lor\ \neg([type \mapsto \text{“Commit”}] \in msgs) \\
\quad \land\ \forall\, rmi \in RM : \forall\, rmj \in RM : \\
\qquad \lor\ \neg([type \mapsto \text{“Abort”}] \in msgs) \\
\qquad \lor\ \neg([type \mapsto \text{“Commit”}] \in msgs) \\
\quad \land\ \forall\, rmi \in RM : \forall\, rmj \in RM : \\
\qquad \lor\ \neg([type \mapsto \text{“Abort”}] \in msgs) \\
\qquad \lor\ \neg(tmState = \text{“init”}) \\
\quad \land\ \forall\, rmi \in RM : \forall\, rmj \in RM : \\
\qquad \lor\ \neg([type \mapsto \text{“Commit”}] \in msgs) \\
\qquad \lor\ \neg(tmState = \text{“init”}) \\
\quad \land\ \forall\, rmi \in RM : \forall\, rmj \in RM : \\
\qquad \lor\ (rmState[rmj] = \text{“prepared”}) \\
\qquad \lor\ \neg([type \mapsto \text{“Prepared”}, rm \mapsto rmj] \in msgs) \\
\qquad \lor\ \neg(tmState = \text{“init”})
\end{array}
$$

图 5-2　TwoPhase 规约的一个归纳不变式

图5-2展示了 TwoPhase 规约的一个归纳不变式。可以看到归纳不变式是由多条引理不变式合取而来，而每个引力不变式又是由多个谓词析取而来。除此以外，第一条引理不变式是 TwoPhase 规约的安全属性。

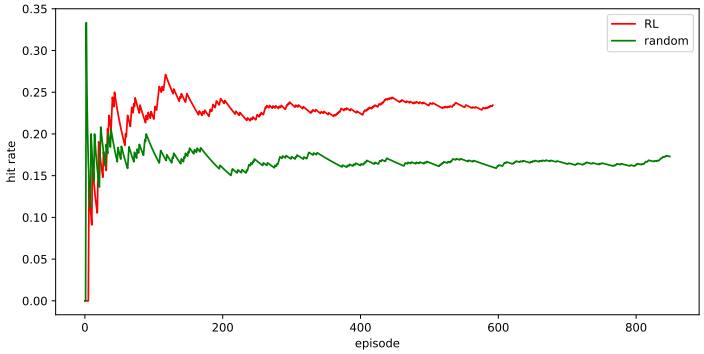
5.2 结果分析

对比 endive 的运行结果，在大多数规约上 RL-TLA 不具有优势。RL-TLA 的主要内存占用为强化学习模型和网络模型的内存占用，endive 没有相关的内容。相较而言，我们的工具生成归纳不变式寻找的引理不变式更多，效率较低。对于这一现象的现象的解释，我认为是强化学习智能体在一开始尝试时会更偏向于选择已有的不变式比较相似的不变式，但是在系统的提示下，相似的不变式往往不能得到很好的奖励值，于是系统便会在更加稀疏的区域寻找不变式。并且没有考虑每个不变式所能消除的归纳反例的数量，只要某个不变式能消除归纳反例，就会被加入。而 endive 则是通过基于候选不变式能杀死归纳反例的个数进行选择，事实上，它可能验证的不变式的个数更多。这样的结果导致了，我们的工具会生成更多的相近不变式，但是，这不妨碍我们的工具生成最终正确的归纳不变式。

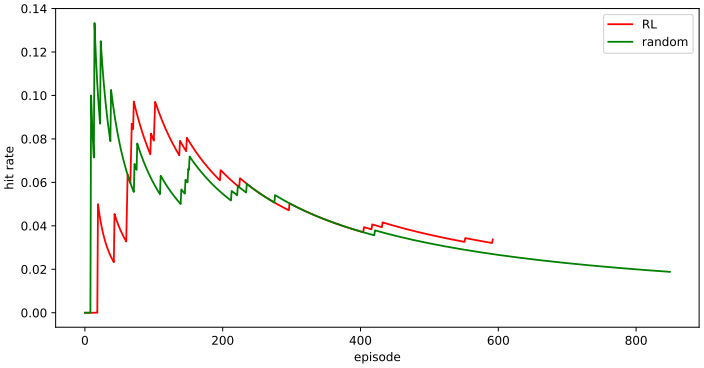
另外，系统运行的总体时间相较于 endive 偏长。一方面，RL-TLA 所需的轮次比 endive 更多。TLC/Apalache 还没有面向 python 的接口，这导致我们需要通过调用命令行的方式来调用 TLC/Apalache。一次调用和数据解析的时间随状态的多少和需要检查不变式的多少指数级增加。另一方面，强化学习模型的训练时间和网络的向前向后传播也是一个不可忽视的因素。但相较于随机遍历，以 *simple_election* 为例，拥有 12 个用户定义谓词，对于长度小于 4 的候选不变式，大概需要遍历接近万次。然而我们的工具，遍历的次数在千次左右，在效率上有显著的提升。

5.3 对强化学习的消融实验

为了验证强化学习模块在归纳不变式生成的过程中是否产生了积极的作用，我们进行了强化学习模块的消融实验。在 TwoPhase 规约上，我们使用完全随机选择 seed 的方式作为对照组，对比使用了强化学习模块的效果。图5-3展示我们的实验结果，子图5-3a展示了随机选择和强化学习选择的候选引理不变式通过不变式检查的频率，子图5-3b展示了最终加入到归纳不变式候选集合中的频率。可以看到，强化学习模块在引理不变式生成过程中起到了积极的作用，提高了候选引理不变式的质量。在将引理不变式加入到归纳不变式集合中的频率对比随机



(a) 引理不变式命中率



(b) 归纳不变式命中率

图 5-3　强化学习选择和随机选择的命中率

过程也有所提升。以上图表表明，强化学习模块在归纳不变式生成过程中起到了积极的作用，证明了强化学习在归纳不变式自动生成中的可用性。

第六章　总结与分析

本章对论文工作进行了总结，并展望了未来可能的优化和改进方向。

6.1　工作总结

本文主要目的是验证机器学习，尤其是强化学习在对分布式系统规约的归纳不变式生成领域中的可行性和有效性。

本文在 TLA^+ 的语言平台上，实现了一个基于强化学习的归纳不变式生成系统。首先，介绍了系统设计的预备知识和理论基础，然后介绍了 RL-TLA 的系统体系结构的设计和实现，最后展示了运行数据，并对结果进行分析。

TLA^+ 相较于 IVy 更加复杂，其中存在灵活多样的数据结构，同时也支持任意的嵌套来表达规约。这对开发人员在设计阶段表达系统的行为和状态转移关系提供了很大的便利，但这点对于归纳不变式生成工具的设计并不友好。

实现上，本文依靠于 TLA^+ 源文件和 endive 对于 TLA^+ 解析和人工识别的假设作为输入，基于 try-and-error 的生成思路，采用强化学习的方式生成候选不变式，通过模型检查器验证候选不变式的正确性，独立性以及当前所有候选不变式合取结果的递归性，最终生成归纳不变式。这一生成思路和大部分的归纳不变式生成工具类似，但是在实现上，本文寄希望于强化学习以提高生成效率。

本文基于 LIPuS[12] 对结构化程序寻找循环不变式的研究开发了一套面向 TLA^+ 的强化学习模型。

本文使用 TLC 和 Apalache 对生成的候选不变式进行验证，检验生成的候选不变式的正确性，独立性和与已有不变式合取结果的递归性，并将结果返回给强化学习模块。TLC 和 Apalache 没有提供 python 的接口，本文通过调用命令行的方式调用 TLC 和 Apalache，并通过对命令行结果的解析，获取验证结果。

在测试部分，本文使用 endive 提供的测试用例对系统进行测试，并和 endive 进行了比较，验证了强化学习在面向分布式系统规约的归纳不变式生成工作中

具有可行性和有效性。

6.2 未来展望

由于一些因素的限制，本文所实现的系统的性能和实现方式上有许多不足，存在大量的改进空间。

目前，和 endive 一样，RL-TLA 还以来于一些人工的输入，即一些人工识别的假设 (predicates)，这些假设的得到是依赖于人脑对于 TLA^+ 规约理解，尤其是对每个 Action 进行调用时参数类型的理解。如果要自动化地识别和生成这些假设，也是一个复杂的工作，目前系统还不具备这项能力。与此同时，人脑的参与可能带来效率的下降和不可预计的错处的出现。另一方面，如同 endive 的通过机械搜索的方式，可能获得比强化学习更快的生成效率，强化学习的优势难以发挥。未来需要实现一个功能更加丰富的静态分析工具，以自动提取出 TLA^+ 规约的语义信息，包括 Action 的" 函数签名" 等，以帮助强化学习模块更好地理解 TLA^+ 规约和生成合适的候选不变式。当前 RL-TLA 学习内容仅限于生成候选的引理不变式，还没有到学习如何生成归纳不变式的地步，这导致到运行过程的后期，系统经常生成已经被覆盖的候选不变式。

近些年，大语言模型的流行，使得自动归纳不变式生成有了更多可能。通过将协议交给大语言模型进行学习，可以更好地理解协议的行为。或者，大语言模型也许可以直接应用于归纳不变式的生成工作，这是一个值得尝试的方向。

其次，目前提供给系统可以选择的谓词对于系统来说都是平行的，并没有考虑谓词的子式以及谓词之间的关系。系统也无法考虑给出的这些谓词表达式之间，以及每个可能的候选不变式之间的关系。这有可能导致系统生成的候选不变式之间存在冗余，或者存在矛盾，导致系统的效率降低。另外，目前系统设计上，一次只生成一个可能的候选不变式，这两方面因素，导致了对于每一次的生成的候选不变式的检查，都需要调用 1-3 次模型检查器进行检查，这往往很花时间，导致系统的效率较低。这也是目前在效率上较 endive 等工作有所不足的地方。

基于 TLA^+ 的归纳不变式的生成是一个复杂的问题，在这一领域的研究不是十分充足，可以参考和对比的工作较少。目前，对于基于 TLA^+ 的归纳不变式生

成工具还没有统一的测试集合，也没有十分充足的测试用例。这导致我们一方面很难评估系统的效率，另一方面，也很难提供给强化学习模块足够的训练数据。在 endive 的测试集合下，一个归纳不变式常常只需要不超过 10 个子式合取而来。在这一背景下，强化学习的效率并不理想，常常带来相较于 endive 等工作提供的搜索算法更高的开支和更低的效率。

参考文献

[1] LAMPORT L. Paxos made simple[J]. ACM SIGACT News (Distributed Computing Column) 32, 4 (Whole Number 121, December 2001), 2001: 51-58.

[2] ONGARO D, OUSTERHOUT J. In search of an understandable consensus algorithm[C]//2014 USENIX annual technical conference (USENIX ATC 14). 2014: 305-319.

[3] KUPFERMAN O, VARDI M. Model checking of safety properties[C]// International Conference on Computer Aided Verification. 1999: 172-183.

[4] MANNA Z, PNUELI A. Temporal verification of reactive systems: safety[M]. Springer Science & Business Media, 2012.

[5] MA H, GOEL A, JEANNIN J B, et al. Towards automatic inference of inductive invariants[C]//Proceedings of the Workshop on Hot Topics in Operating Systems. 2019: 30-36.

[6] LAMPORT L, MATTHEWS J, TUTTLE M, et al. Specifying and verifying systems with TLA+[C/OL]//EW 10: Proceedings of the 10th Workshop on ACM SIGOPS European Workshop. Saint-Emilion, France: Association for Computing Machinery, 2002: 45-48. https://doi.org/10.1145/1133373.1133382. DOI: 10.1145/1133373.1133382.

[7] PADON O, MCMILLAN K L, PANDA A, et al. Ivy: safety verification by interactive generalization[C]//Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation. 2016: 614-630.

[8] FEDYUKOVICH G, PRABHU S, MADHUKAR K, et al. Quantified invariants via syntax-guided synthesis[C]//Computer Aided Verification: 31st International Conference, CAV 2019, New York City, NY, USA, July 15-18, 2019,

Proceedings, Part I 31. 2019: 259-277.

[9] GARG P, LÖDING C, MADHUSUDAN P, et al. ICE: A robust framework for learning invariants[C]//Computer Aided Verification: 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings 26. 2014: 69-87.

[10] SHEERAN M, SINGH S, STÅLMARCK G. Checking Safety Properties Using Induction and a SAT-Solver[C]//HUNT W A, JOHNSON S D. Formal Methods in Computer-Aided Design. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000: 127-144.

[11] GARG P, NEIDER D, MADHUSUDAN P, et al. Learning invariants using decision trees and implication counterexamples[J]. ACM Sigplan Notices, 2016, 51(1): 499-512.

[12] YU S, WANG T, WANG J. Loop Invariant Inference through SMT Solving Enhanced Reinforcement Learning[C]//Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis. 2023: 175-187.

[13] PEI K, BIEBER D, SHI K, et al. Can large language models reason about program invariants?[C]//International Conference on Machine Learning. 2023: 27496-27520.

[14] YAO J, TAO R, GU R, et al. DistAI: Data-Driven Automated Invariant Learning for Distributed Protocols[C/OL]//15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21). USENIX Association, 2021: 405-421. https://www.usenix.org/conference/osdi21/presentation/yao.

[15] YAO J, TAO R, GU R, et al. DuoAI: Fast, Automated Inference of Inductive Invariants for Verifying Distributed Protocols[C/OL]//16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22). Carlsbad, CA: USENIX Association, 2022: 485-501. https://www.usenix.org/conference/osdi22/presentation/yao.

[16] MA H, GOEL A, JEANNIN J B, et al. I4: incremental inference of inductive

invariants for verification of distributed protocols[C]//Proceedings of the 27th ACM Symposium on Operating Systems Principles. 2019: 370-384.

[17] GOEL A, SAKALLAH K. Model checking of verilog rtl using ic3 with syntax-guided abstraction[C]//NASA Formal Methods: 11th International Symposium, NFM 2019, Houston, TX, USA, May 7–9, 2019, Proceedings 11. 2019: 166-185.

[18] HAWBLITZEL C, HOWELL J, KAPRITSOS M, et al. IronFleet: proving practical distributed systems correct[C]//Proceedings of the 25th Symposium on Operating Systems Principles. 2015: 1-17.

[19] WILCOX J R, WOOS D, PANCHEKHA P, et al. Verdi: a framework for implementing and formally verifying distributed systems[C]//Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation. 2015: 357-368.

[20] SCHULTZ W, DARDIK I, TRIPAKIS S. Plain and simple inductive invariant inference for distributed protocols in tla+[C]//2022 Formal Methods in Computer-Aided Design (FMCAD). 2022: 273-283.

[21] LAMPORT L. The temporal logic of actions[J]. ACM Transactions on Programming Languages and Systems (TOPLAS), 1994, 16(3): 872-923.

[22] 易星辰, 魏恒峰, 黄宇, 等. PaxosStore 中共识协议 TPaxos 的推导、规约与精化[J]. 软件学报, 2020, 31(8): 2336. DOI: 10.13328/j.cnki.jos.005964.

[23] KUPPE M A, LAMPORT L, RICKETTS D. The TLA+ toolbox[J]. arXiv preprint arXiv:1912.10633, 2019.

[24] LAMPORT L. The PlusCal algorithm language[C]//International Colloquium on Theoretical Aspects of Computing. 2009: 36-60.

[25] KONNOV I, KUKOVEC J, TRAN T H. TLA+ model checking made symbolic [J]. Proceedings of the ACM on Programming Languages, 2019, 3(OOPSLA): 1-30.

[26] OTONI R, KONNOV I, KUKOVEC J, et al. Symbolic Model Checking for TLA+ Made Faster[C]//International Conference on Tools and Algorithms for

the Construction and Analysis of Systems. 2023: 126-144.

[27] DE MOURA L, BJØRNER N. Z3: An efficient SMT solver[C]//International conference on Tools and Algorithms for the Construction and Analysis of Systems. 2008: 337-340.

[28] KAELBLING L P, LITTMAN M L, MOORE A W. Reinforcement learning: A survey[J]. Journal of artificial intelligence research, 1996, 4: 237-285.

[29] PUTERMAN M L. Markov decision processes[J]. Handbooks in operations research and management science, 1990, 2: 331-434.

[30] MNIH V, KAVUKCUOGLU K, SILVER D, et al. Human-level control through deep reinforcement learning[J]. nature, 2015, 518(7540): 529-533.

[31] WITANTO J N, LIM H. Software-Defined Networking Application with Deep Deterministic Policy Gradient[C/OL]//ICCMS '19: Proceedings of the 11th International Conference on Computer Modeling and Simulation. North Rockhampton, QLD, Australia: Association for Computing Machinery, 2019: 176-179. https://doi.org/10.1145/3307363.3307404. DOI: 10.1145/3307363.3307404.

致　　谢

时光荏苒，四年的时间很快过去了。2020 年 9 月 3 日大清早，拖着行李走进南大校门，穿过教学楼和一组团主干道的场景还历历在目，仿佛就在昨天，如今却快要到达四年本科学习的终点，走向下一段旅程。本科四年，求学生涯，乃至人生到此的所有阶段，都离不开许多人的帮助和支持。

感谢父母，祖父母对我的养育之恩，对我生活和求学的支持。是他们的付出，让我有成长学习的环境。

感谢魏恒峰老师在学习生活上的指导。是他带领我进入科研的大门，让我了解分布式系统验证和形式化方法。同时感谢邓楚宸同学在本项目实现上做出的大量工作。

感谢诸位朋友的陪伴，开导，帮助和建议，人生因为你们而丰富多彩。

感谢党和国家对教育事业的支持。感谢南京大学，以及诸位老师，工作人员为我们提供了一个良好的学习环境。