密级:

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

硕士学位论文

基于启发式方法的带参系统形式化验证

作者姓名: 段凯强

指导教师: 李勇坚 副研究员

中国科学院软件研究所

学位类别: 工学硕士

学科专业: 计算机软件与理论

研究所: 中国科学院软件研究所

2016 年 4 月

Formal Verification of Parameterized Systems based on Heuristics

By

Kai-Qiang Duan

A Dissertation Submitted to
The University of Chinese Academy of Sciences
In partial fulfillment of the requirements
For the degree of
Master of Computer Software and Theory

Institute of Software, Chinese Academy of Sciences

April, 2016

学位论文独创性声明

本人郑重声明：我所呈交的学位论文是本人在导师指导下进行的研究工作及所取得的研究成果。尽我所知，除了文中已经标注引用的内容外，本论文中不含其他个人或集体已经发表或撰写过的研究成果。对本文的研究做出贡献的个人和集体，均已在文中作了明确的说明或致谢。本人知道本声明的法律结果由自己承担。

学位论文作者签名：_____日期：_____

关于学位论文使用授权的说明

本人完全了解中国科学院软件研究所有关保留、使用学位论文的规定，即：中国科学院软件研究所有权保留送交论文的复印件，允许论文被查阅和借阅；中国科学院软件研究所可以公布论文的全部或部分内容，可以采用影印、缩印或其它复制手段保存论文。

学生签名：_____导师签名：_____日期：_____

基于启发式方法的带参系统形式化验证

摘　　要

在很多领域中都有带参系统的实际应用，如缓存一致性协议、安全系统和网络通信协议等。一般而言，带参系统中存在多个完全相同的并发进程，这些进程的个数即带参系统的参数。对带参系统进行形式化验证是一个颇具挑战性的困难问题，因为需要证明其安全性质在任意参数下都能够成立。针对这一问题，本文进行了以下方面的工作：

1. 设计了调用第三方工具的客户端/服务器架构，以达到提高查询效率、分解大型任务、实现结果复用等目的；

2. 提出了一个形式化语义模型并构建了相关工具，设计并实现了验证工具 ParaVerifer，应用一系列启发式策略来自动生成因果关系和辅助不变式，并最终生成 Isabelle 证明脚本；

3. 能够进行简单的流图分析，对一些带参系统状态的典型迁移过程给出直观的表示，有助于加深对带参系统的理解。

ParaVerifer 验证工具能够完成对 MESI、German 和 Flash 等一系列基准测试用例的完整形式验证工作，且能够对这些带参系统中的典型状态迁移流进行分析并给出直观的图形表示。

关键词： 带参系统，形式验证，启发式方法，流图

Formal Verification of Parameterized Systems based on Heuristics

Abstract

Parameterized systems exist in many important application areas, such as cache coherence protocols, security systems and network communication protocols. In general, there exist many replicated processes in a parameterized system, and number of the processes is the so-called parameter. Formal verification of parameterized systems is challenging and hard, because we need to verify that safe properties of parameterized systems must hold with any parameter. In order to solve this problem, this paper presents the following works:

1. Design a client/server architecture for calling third-party tools to improve performance, balance load and reuse results.

2. Present a formal semantic model and construct some corresponding tools; design and implement verification tool ParaVerifer, which apply a series of heuristic strategies to generate casual relations and auxiliary invariants automatically, and finally generate proof scripts in Isabelle.

3. Do simple flow chart analysis which can give intuitive representation of typical transitions in some parameterized systems. Flow chart analysis is useful to understand a parameterized system.

Our verification tool ParaVerifer performs well in formal verification of many parameterized benchmarks, such as MESI, German and Flash. Besides, ParaVerifier can generate intuitive flow chart for typical flows in the benchmarks.

KEYWORDS: Parameterized systems, Formal Verification, Heuristics, Flow chart

目　录

表　格

插　图

第一章　引言

带参系统的形式化验证是一个有趣而颇具挑战性的问题，因为其具有重要的实践意义。很多领域中的实际应用，如缓存一致性协议、安全系统和网络通信协议等，都可以用带参系统描述。这些带参系统给常规的模型检测技术带来了严峻的挑战。对带参系统进行形式化验证的困难之处，在于需要保证其安全性质在任意规模的系统中都得到满足。安全性质就是在一个系统必须始终得到满足的性质，也就是说，在这个系统所有能够到达的状态中，安全性质永远保持成立，因而安全性质也称为不变式。为了解决带参系统的验证问题，人们已经提出了一系列的方法[1–6]，包括基于模型检测技术的和基于定理证明技术的，前者针对带参系统的某个足够小的实例能够给出很好的解决方案，而后者在简单系统的完整验证上具有不错的表现。然而，对比较复杂的带参系统进行完整的形式化验证仍然存在很大的困难。

1.1　带参系统验证简述

带参系统广泛存在于计算机体系的核心模块中，一般包含参数个具有相同结构的并发执行的主体，此外也包含有限个结构不同的主体。带参系统 $P(N)$ 通常使用一个参数 N 来表示具有相同结构的主体的数目，这里的参数 N 是一个任意的自然数。如果确定参数 x，那么就能够得到带参系统的一个实例。一个带参系统具有无穷多个实例。带参系统中的主体通过一系列的规则进行交互，这些规则描述了带参系统何时以及如何发生状态迁移，从而实现带参系统预定的功能，因此这些规则构成的集合 R 能够刻画带参系统的运行过程。

在实际应用中，很多系统都可以使用带参系统描述，如缓存一致性协议中的 German 协议[7]。German 协议是一种基于目录的带参缓存一致性协议，具有一个维护目录的主节点 Home 和多个执行功能的节点 Client 两种主体，其中 Client 节点就是具有相同结构的并发执行的主体，也就是参数化的。Client 节点向 Home 节点发出申请数据缓存副本的命令，根据读写命令的不同，数据缓存的方式有共享（Shared）、独占（Exclusive）等方式，Home 节点要根据系统所处的状态进行处理，进行必要的写回/写穿等操作，对主存的数据进行读写，并在必要的情况下使其他的节点缓存副本变为无效状态（Invalid），所有这些活动都是用规则描述的。

带参系统的验证目标一般是安全性质，要求在带参系统运行过程中所达到的任何状态都得到满足，而对带参系统进行完整验证是非常困难的[8]。一个带参系统 $P(N)$ 对应着无穷多个实例 $P(x)$，当实参 x 分别为具体的自然数 $1, 2, 3, \cdots$ 时，可以通过穷尽系统可达状态的方法来验证安全性质是否满足，但是无论实参 x 多大，都无法确保带参系统在任意规模下的正确性，况且随着实参 x 取值的增大，系统状态空间也呈指数

级增长，这种模型检测的方法也不再可行。另一种验证策略使用定理证明技术，基于某种逻辑的形式化系统，将系统的结构以及功能用该逻辑系统的公式表示出来，然后基于该逻辑的推理系统，去证明描述功能定义的公式是结构定义公式的逻辑推论。由于高阶逻辑系统有非常强的表达能力，使用它可以描述带参系统的规则以及性质，把系统参数 N 看成符号，可以使用诸如归纳法等强有力的数学工具来证明性质，所以从理论上来讲，定理证明技术非常适合来验证带参协议的性质。然而在使用归纳法时，必须构造出足够的辅助归纳不变式集才能使得归纳证明能够完成。对于稍微复杂的一些带参协议，这种构造是非常困难的，正是这一点，极大地阻碍了定理证明技术在带参协议验证上的应用。

在这篇论文中提出了一种基于启发式方法的带参系统形式化验证方案并实现了对应的验证工具 ParaVerifier。首先建立合适的形式化语义模型，以用于描述带参系统的语义行为与协议性质；基于协议规则与性质之间存在的保证性质在规则执行前后都能成立的因果关系，提出针对带参系统的归纳证明方法；然后基于一系列的启发式策略，构造归纳证明需要的不变式；最后使用一系列的自动化策略，融合归纳不变式集的计算，形式化证明的自动构造，以及证明的自动检测，完成对带参系统的验证工作。

1.2 论文大纲

本论文接下来的内容结构如下：第二章介绍了一些常用的带参系统验证方法和验证工具 ParaVerifier 的理论基础；第三章介绍了 ParaVerifier 的形式化语义模型和基于已有模型自动化建模的方法；第四章介绍了 ParaVerifier 使用第三方工具进行符号模型检测和可满足性检查的技术方案；第五章介绍了 ParaVerifier 如何应用启发式策略进行设计和实现；第六章以一些带参缓存一致性协议为例介绍了使用 ParaVerifier 完成验证的过程；第七章给出了结论和下一步工作。

第二章　常用验证方法和 ParaVerifier 的理论基础

对带参系统进行验证一直都是形式化领域中的热点和难点。要证明安全性质即不变式在带参系统的所有可达状态下都得到满足，常用的策略有两种，即基于模型检测的技术[9–11] 和基于定理证明的技术[12–15] 。模型检测技术基于有限状态空间的自动搜索，它完全不需要人的干预，而且随着对称、抽象等技术的引进，模型检测技术的能力在不断提高，但是模型检测只能检测带参系统的一个实例，无法对其所有实例进行检测。定理证明技术能够严格证明带参系统的正确性，然而在使用归纳法时，必须构造出足够的辅助归纳不变式集才能完成归纳证明，对于稍微复杂一些的带参系统，这种构造是非常困难的。验证工具 ParaVerifier 正是基于启发式策略，借助模型检测技术完成辅助不变式的自动生成，最终应用定理证明技术完成对带参系统的归纳证明。

2.1　常用验证方法简述

带参系统的性质验证问题在一般情况下是不可判定的，所以到目前为止，在带参系统验证领域，往往针对某一类具有特别重要价值的协议而展开研究：如缓存一致性协议，安全协议等。到目前为止，验证方法大体上可分为四类。

2.1.1　截止方法

截止（cutoff）方法使用一个阈值 C ，只要参数 N 小于或者等于 C 的所有带参系统实例都满足性质 F ，那么就认为对参数 N 的任意取值，相应实例都满足 F 即 $P(N) \models F$。这样，问题就从对任意规模的带参系统实例进行安全性质验证，规约为对有限多个小规模实例的验证，由此给出一个确定的判定过程。所以只需对所有规模小于 C 的实例进行性质 F 的模型检测，即可完成对带参系统的性质 F 的验证。Emerson 最早提出了这种将无穷实例转化到有限个实例的规约思想，并对简单广播通讯协议做了实例研究[6] 。基于截止的思想，针对不同的应用，许多学者都对截止方法进行了扩展与优化。Bouajjani 对先进先出队列（FIFO）[16] ，Alan hu 对简单的目录协议（如 German 协议）[17] ，Daniel Kroening 对多线程的动态性质[18] 都进行了形式化验证。

截止方法的主要问题是阈值 C 的确定以及规约到 C -实例的正确性证明。阈值 C 的确定，往往依赖于人们对某类特定协议的知识，手工构造，而且规约的正确性往往是非形式化的纸面证明（paper-proof)。另外，对某些协议来说，人工构造出来的阈值 C 仍然太大，规模等于 C 的协议实例的复杂度仍然超过现代模型检测工具能够处理的能力。

2.1.2 抽象方法

抽象（abstraction）方法就是给定带参系统 $P(N)$，给定所要验证的性质 F，构造出一个带参系统的抽象实例 $P(M)$，$P(M)$ 的主体个数通常很少（3 到 4 个），如果性质 F 在 $P(M)$ 上成立，则 F 对该带参系统的的任何实例 $P(x)$ 都成立。在这里，F 通常是针对所有主体而言的安全性质。$P(M)$ 之所以具有这种代表性，是因为 $P(M)$ 可以模拟 $P(x)$ 中与验证性质相关的状态与迁移信息，并略去 $P(x)$ 中与验证性质无关的信息。更形式化地讲，从 $P(x)$ 的状态空间到 $P(M)$ 的状态空间，存在一个抽象函数，该函数不但保证两个空间状态和迁移之间的对应，而且在映射后安全性质仍然能够得到保持。基于抽象的代表性工作有基于参数抽象与卫式加强的方法和基于谓词抽象的方法。

- 基于参数抽象与卫式加强（parameter abstraction and guard strengthening）的方法[19]。该方法所构造的抽象模型包括所选取的有限个代表主体，以及另外一个附加主体，它代表对所有未选主体的抽象。该抽象模型的构造过程遵循一种反例引导逐步精化（counter-example guided refinement）的模式，当抽象模型不满足所验证的性质时，需要人工分析反例，找出一个适当的辅助不变式，去加强某些规则的卫式以约束该模型。然后一直重复这个过程，直至原验证性质以及所有辅助不变式在抽象模型中能够被满足。Xiaofang Chen 等应用该方法分析了复杂的多核缓存一致性协议，多层次缓存协议[20,21]。根据类似的思想，Talupur 和 Clarke 提出了环境抽象的概念[22,23]，以抽象节点来抽象非代表主体的行为。上述两种方法的缺点在于不变式必需由人手工提供，这需要对协议语义要有深刻的理解，求精过程的反例分析要有很好的掌握。吕毅等试图从一个小的协议的实例出发，用该实例的可达集的 BDD 表示导出的布尔公式作为不变式整体，用于加强协议规则的卫式条件，从而自动实现求精与卫式加强[24]，此方法适用于 German 协议，但不适合 Flash[25] 协议。Talupur 等提出了流（Flow）的概念，对不变式的生成起到辅助作用，但依然需要手工分析[26,27]。

- 基于谓词抽象（predicate abstraction）的方法[28]，要求给出一组关于协议变量的谓词作为抽象模型的抽象状态集合。抽象状态 p 与 p' 之间存在迁移关系，当且仅当存在协议的具体状态 s 与 s' 使得有 $p(s)$，$p(s')$，并且存在从 s 到 s' 的迁移。谓词发现和谓词抽象技术的结合，为非有穷状态系统的安全性性质给出了一种全自动验证的方法。这里的谓词发现技术指的是根据初始谓词集、所验证的性质和系统迁移规则，通过经验性的启发式算法自动找到有用的谓词集合。应用这种方法，Baukus 等分析了 German 协议[29]，Das 等分析了简化版本的 Flash 协议[30]。应用谓词抽象技术的关键难点在于如何找到一组合适的谓词用于协议抽象。这些谓词并不是很容易就能被发现的，同样需要人们根据领域知识手工构造。找到这些合适的谓词实际上等价于找到合适的归纳不变式。

2.1.3　组合式验证技术

组合式验证技术的基本思想是分而治之 (divide-and-conquer)，根据被验证模型的特点，将一个无穷的带参验证问题自动或人工地分解为若干有穷的小问题，并且保证这些小问题能够被模型检测工具自动检测。Ken McMillan 使用 Cadence SMV 工具实现了相应的组合式验证技术，并分析了 Flash 协议的安全性质与活性性质[31] ，该技术同时结合了时态切分、数据归约、对称规约等方法。组合式验证的理论基础是针对时序的超限归纳法，但是用户必须提供辅助不变式来分解问题，因而使用者不但要对 SMV 相应的归约证明机制有很好的理解，同时又要对所要验证的带参协议的语义非常清楚。值得一提的是，McMillan 的工作验证了 Flash 协议的活性性质，这是其他方法不能做到的。

2.1.4　归纳证明方法

基于归纳不变式 (inductive invariant) 的归纳证明方法来源于定理证明中的归纳法思想。归纳定义以及归纳证明的具体实现依赖于人们选取的定理证明工具。Park 等人使用 PVS 定理证明器，基于对 Flash 协议本身的深刻理解，构造出相应的辅助不变量，并验证了完整的 Flash 协议的一致性[25] ，值得一提的是，这是人们首次完成对 Flash 协议的安全性验证。A. Pnueli 等提出了不可见不变量 (invisible invariants)，基于小的协议实例的可达集的 BDD 符号化表示，自动计算出相应的辅助不变式，并基于 TLV 工具实现了归纳证明[32] 。Pandav 等提出了一种启发式的方法，给出了分析反例并构造不变式的几种模式，人工构造所需归纳不变式，并基于 UCLID 工具实现了归纳证明，手工完成了 German 与简化版的 Flash 的验证[33] 。Sylvain 等实现了 CUBICLE 工具，采用了后向搜索算法，也根据小实例的可达集，自动地计算出辅助不变式集合，并能够生成相应的 Why3 证书 (certificate)，来作为该带参协议的形式化证明，Coq 证明器能够支持 Why3 证书对应的证明。CUBICLE 实现基于两个关键技术，一个是基于并行 SMT 求解器的一阶逻辑判定工具，另一个是后向搜索算法。前者用于表示带参协议的语义模型以及性质的形式化表示，后者用于不变式的生成。这两项技术使得 CUBICLE 成为目前为止此最好的带参协议验证工具。除了还未能给出完整版的 Flash 协议的 Why3 证书，一般带参协议的基准测试集 (benchmark) 都能得到自动验证[34,35] 。

2.2　ParaVerifier 的理论基础

验证工具 ParaVerifier 使用一个语义模型形式化描述带参系统，包括带参系统的状态变量、迁移规则和不变式。前边已经提到，构造出足够的辅助归纳不变式，有助于完成对带参系统安全性质的归纳证明，而我们的工作根据不变式和规则之间的因果关系，解决了自动生成足够辅助不变式的问题，并能够在此基础上给出归纳证明脚本。

2.2.1 带参系统模型

在 ParaVerifier 中，带参系统用一个六元组 $< N, T, V, I, R, F >$ 进行建模，其中

1. N 是一个字符串，代表带参系统名称；
2. T 是一个集合，代表自定义类型；
3. V 是一个集合，代表带参系统的状态变量，这个变量集可以描述系统所可能到达的每个状态，决定状态空间的上界；
4. I 是一个语句，代表系统初始状态，是一个赋值集合 $\{v_i := e_i | 1 \leq i \leq M\}$ ，表示将第 i 个状态变量 v_i 的值更新为表达式 e_i 的值，其中 M 是赋值数目；
5. R 是一个集合，代表迁移规则，每个迁移规则都由卫式和语句两部分组成，卫式是一个指定何时触发该规则的谓词公式，语句则描述该规则触发后如何更新状态变量发生迁移；
6. F 是一个谓词公式集合，代表要验证的安全性质，安全性质必须在系统的可达状态集中都得到满足。

在描述带参系统的这六种元素中，V 、I 和 R 这三个属于本征性元素，用于描述带参系统的本征活动和生命周期；N 和 T 属于辅助性元素，分别用于标志带参系统和提高模型表达能力；而 F 是目标性元素，刻画验证目标。

2.2.2 因果关系和一致性引理

我们在之前的工作中已经提出了因果关系和一致性引理[36,37] 。在 ParaVerifier 中，安全性质使用谓词公式描述，这个谓词公式中包含一系列的常量或者状态变量，用来说明状态变量之间需要满足的关系。对于某个固定的参数，带参系统的安全性质需要在可达集的每个状态中都得到满足，而其可达集 $R(P)$ 递归定义如定义 2.1 所示。

定义 2.1.
对固定参数的带参系统 P ，其可达集 $R(P)$ 是一个状态集合：若状态 s_0 是一个初始状态，那么 $s_0 \in R(P)$ ；若状态 $s \in R(P)$ 且 s' 是 s 在触发某个规则后迁移到的状态，那么 $s' \in R(P)$ 。

某条安全性质 $f \in F$ 关于规则 $r \in R$ 的前置条件如定义 2.2 所示。

定义 2.2.
若语句 $\alpha = \{v_i := e_i | 1 \leq i \leq M\}$ 是规则 r 的赋值序列，其中 M 是赋值语句的数目，那么公式 f 关于规则 r 的前置条件 $preCond(f, \alpha) = f[\vec{e}/\vec{v}]$ ，即使用 e_i 替换 v_i 在 f 中的每次出现后得到的公式。

如果状态 s' 可以由状态 s 通过执行规则 r 的赋值序列 α 得到，那么根据 Hoare 逻辑，若有 $s' \models f$ 则有 $s \models preCond(f, \alpha)$ ，也就是说，前置条件是执行一条规则前系统状态所需要满足的最弱的条件。

现在，给出关于规则 r 和公式 f 的因果关系的归纳定义，如定义 2.3 所示。

定义 2.3.
考虑规则 r ，公式 f ，公式集合 F ，以及规则的卫式 g 和赋值序列 α ，可以定义一系列的因果关系：

1. ***invHoldForRule1***：$g \rightarrow preCond(f, \alpha)$ 成立，也就是说，执行规则 r 后，f 立即得到满足；
2. ***invHoldForRule2***：α 不改变 f 中的任何变量即 $f \leftrightarrow preCond(f, \alpha)$ ，也就是说，执行规则 r 后 f 自然满足；
3. ***invHoldForRule3***：存在公式 $f' \in F$ ，使得 $(f' \wedge g) \rightarrow preCond(f, \alpha)$ ；
4. ***invHoldForRule***：以上三种关系至少成立其一，即

$$invHoldForRule1 \vee invHoldForRule2 \vee invHoldForRule3$$

在 ParaVerifier 中，带参系统的初始状态用一系列的赋值 $I = \{v_i := e_i | i \geq 1\}$ 描述，这些赋值实际上指定了系统在一开始时每个状态变量需要满足的条件，因此也可以用一个公式集合 $inis$ 来描述，再结合带参系统的不变式集 $invs$ 和规则集 rs ，给出定义 2.4 。

定义 2.4.
对不变式集 $invs$ ，初始状态所满足的条件集 $inis$ 和规则集 rs ，若满足以下条件，则称它们之间的关系为一致性关系：

1. 对任意不变式 $inv \in invs$ ，任意初始条件 $ini \in inis$ ，任意可达系统状态 s ，若 $s \models ini$ 则 $s \models inv$ ；
2. 对任意不变式 $inv \in invs$ ，任意规则 $r \in rs$ ，任意可达系统状态 s ，关系 $invHoldForRule$ 成立。

在一致性关系的基础上，给出一致性引理 2.1 。一致性引理为使用归纳法完成对带参系统的形式验证提供了理论支持。

引理 2.1.
对一个带参系统的不变式集 $invs$ ，初始状态所满足的条件集 $inis$ 和规则集 rs ，以及某个可达系统状态 s ，如果满足一致性关系，那么对任意 $inv \in invs$ ，有 $s \models inv$ 。

证明. 使用归纳法证明如下:

1. 固定初始条件 ini 和可达状态 s ，假设 $s \models ini$ ，根据定义 2.4 的第一部分，对任意不变式 $inv \in invs$ 有 $s \models inv$;

2. 固定规则 $r =< g, \alpha >$ 和可达状态 s ，假设

   (a) 对任意 $inv \in invs$ 有 $s \models inv$;

   (b) $s \models g$ 且执行规则 r 的赋值序列 α 后，状态 s 迁移为状态 s'。

   对任意不变式 $inv \in invs$ ，现在要证明 $s' \models inv$ ，根据 Hoare 逻辑，即要证明 $s' \models preCond(inv, \alpha)$ 。根据定义 2.4 的第二部分，有三种情况:

   - **invHoldForRule1** : 即有 $g \rightarrow preCond(f, \alpha)$ ，结合假设 2b ，显然成立;
   - **invHoldForRule2** : 即有 $preCond(inv, \alpha) \leftrightarrow inv$ ，结合假设 2a ，亦成立;
   - **invHoldForRule3** : 即有 $inv' \in invs$ 使 $(inv' \wedge g) \rightarrow preCond(inv, \alpha)$ ，根据假设 2a 有 $s \models inv'$ ，再根据假设 2b 可得结论 $s' \models preCond(inv, \alpha)$ 。

3. 综上，引理得证。

□

2.2.3 一致性关系的必然满足性

一个带参系统如果满足一致性关系，那么根据一致性引理，该带参系统必然能够保证其不变式成立。实际上，正确运行的带参系统必然满足一致性关系，即引理 2.2 。

引理 2.2.
对一个带参系统的不变式集 $invs$ 和某个可达系统状态 s ，如果对任意 $inv \in invs$ 有 $s \models inv$ ，那么该带参系统满足一致性关系。

证明. 对一个带参系统的不变式集 $invs$ ，初始状态所满足的条件集 $inis$ 和规则集 rs ，以及某个可达系统状态 s ，使用反证法。

1. 若带参系统不满足一致性关系的第一部分，即存在 $ini \in inis$ 和 $inv \in invs$ ，使得 $s \models ini$ 成立时 $s \models inv$ 不成立，即一个状态不满足某个不变式，与前提条件矛盾;

2. 若带参系统不满足一致性关系的第二部分，即 *invHoldForRule1-3* 同时不成立，也就是说，在 *invHoldForRule1-2* 不成立时，*invHoldForRule3* 亦不能成立，也就是说，存在不变式 $inv \in invs$ 和规则 $r \in rs$ ，使得对任意 $inv' \in invs$ ，都无法满足 $(inv' \wedge g) \rightarrow preCond(inv, \alpha)$ ，其中 g 为规则 r 的卫式，α 为规则 r 的赋值序

列。因此，不变式 inv 的前置条件 $preCond(inv, \alpha)$ 无法得到满足，故该不变式在执行这条规则后无法成立，带参系统产生反例，与前提条件矛盾。

□

引理 2.2 为形式化验证带参系统提供了一种有效的思路，即可以先假设系统能够正确运行，构造出其因果关系和归纳不变式，然后基于归纳法和一致性引理完成证明。如果在构造过程中违背了一致性关系，那么说明带参系统的运行过程存在反例。

2.2.4　启发式方法

生成因果关系和辅助不变式的过程本质上是一个搜索过程，而带参系统的搜索空间是无穷大的。基于先验领域知识提出一系列的启发式策略，可以显著减小搜索空间，使得验证过程成为可能。这些启发式策略的基础在于：

1. 带参系统的有多个完全相同的主体参与，这些主体是对称轮换的；
2. 错误的候选不变式实例一般可以快速判定。

2.3　本章小结

本章首先介绍了几种常用的验证方法，并指出了每种方法的特点和优劣。然后介绍了验证工具 ParaVerifier 的理论基础：

- ParaVerifier 的带参系统模型，即对一个带参系统进行建模的宏观组件；
- 提出一致性关系和一致性引理，是对带参系统进行证明的基础；
- 论证了一致性关系的必然满足性，为生成归纳辅助不变式提供了理论支持；
- 简述了生成因果关系和归纳辅助不变式的启发式策略。

第三章　ParaVerifier 的语义模型和自动建模

使用定理证明技术完成对带参系统的形式验证，需要找到足够的辅助不变式。基于定义 2.3 中的第三种因果关系，可以得到最弱的辅助不变式，然后通过符号模型检测、可满足行检查等技术逐步将其强化，从而得到真正的辅助不变式。但是模型检测和可满足性检查需要使用相应的工具如 NuSMV[38] 和 Z3[39] ，这两者都需要使用相应的语言建立对应的模型。然而，一方面，对每个需要验证的带参系统都建立不同语言版本的模型，会造成维护上的困难，且容易出错；另一方面，对复杂系统建立这些模型，工作量庞大，以至于无法真正实施。而 ParaVerifier 先将已有 Murphi[40] 模型提升到一个形式化语义模型，再自动建立其他所需要的模型，从而一举解决这些问题。这个工作的一般过程如图 3.1 所示

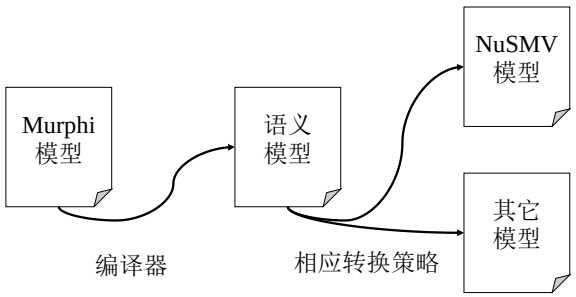

图 3.1: 自动建模的一般过程

3.1　形式化语义模型

ParaVerifier 内部的形式化语义模型使用 OCaml 语言[41] 描述。OCaml 语言是一种多范式语言，具有函数式、模块化、面向对象、面向过程等多种特性，并且有较好的性能。ParaVerifier 的语义模型完整定义如表 3.1 所示。

3.1.1　基本定义

表 3.1 中的第一部分为 ParaVerifier 形式化语义模型的基本定义，这些定义构成带参系统语义模型的底层组件，决定了模型的基本表达能力。

常量 const 包括三种类型的常量，分别是数值类型（构造器 `Intc`）、字符串类型（构造器 `Strc`）和布尔类型（构造器 `Boolc`）。

类型定义 typedef 允许使用构造器 `Enum` 自定义枚举类型，由类型名称和枚举值列表构成，用来表示一组有限范围内的常量。

```
type const =
    | Intc of int
    | Strc of string
    | Boolc of bool
type typedef =
    | Enum of string * const list
type paramdef =
    | Paramdef of string * string
type paramref =
    | Paramref of string
    | Paramfix of string * string * const
type vardef =
    | Arrdef of (string * paramdef list) list * string
type var =
    | Arr of (string * paramref list) list
```

```
type exp =
    | Const of const
    | Var of var
    | Param of paramref
    | Ite of formula * exp * exp
and formula =
    | Chaos
    | Miracle
    | Eqn of exp * exp
    | Neg of formula
    | AndList of formula list
    | OrList of formula list
    | Imply of formula * formula
    | ForallFormula of paramdef list * formula
    | ExistFormula of paramdef list * formula
type statement =
    | Assign of var * exp
    | Parallel of statement list
    | IfStatement of formula * statement
    | IfelseStatement of formula * statement * statement
    | ForStatement of statement * paramdef list
type prop =
    | Prop of string * paramdef list * formula
type rule =
    | Rule of string * paramdef list * formula * statement
```

```
type protocol = {
    name: string; types: typedef list; vardefs: vardef list;
    init: statement; rules: rule list; properties: prop list;
}
```

表 3.1: 形式化语义模型的定义

参数定义 paramdef 允许使用构造器 `Paramdef` 定义参数，由参数名称和参数类型名称构成。

参数引用 paramref 允许引用已有定义的参数，这里会出现两种情况：

1. 形式参数引用构造器 `Paramref` ，通过引用参数名表示引用了形式参数；
2. 实例参数引用构造器 `Paramfix` ，由参数名、参数类型名和实参值构成，带参系统实例化后会出现这种情况。

变量定义 vardef 允许使用构造器 `Arrdef` 定义变量，由变量成员列表和变量类型构成，每个变量成员都需要指定变量成员的名称和变量成员的参数列表，当变量成员列表超过 1 个时即定义了一个复合变量，关于变量定义，更详细的内容请参考附录 A 。

变量引用 var 允许使用构造器 `Arr` 引用变量，是一个变量成员列表，每个变量成员都需要指定成员名称和其参数列表，更详细的内容请参考附录 A 。

3.1.2 高级定义

表 3.1 中的第二部分为 ParaVerifier 形式化语义模型的高级定义，这些定义构成带参系统语义模型的高层组件，如表达式和公式等，决定了模型支持哪些高级语法概念。

表达式 exp 允许对简单表达式进行建模，跟类型 **formula** 递归定义，有以下几种类型：

1. 常量构造器 `Const` ，需要一个常量值；
2. 变量构造器 `Var` ，需要引用一个变量；
3. 参数构造器 `Param` ，需要引用一个参数，可以是形参或者实参；
4. 条件构造器 `Ite` ，由条件、条件真时的值、条件假时的值构成，可以对条件表达式建模。

公式 formula 允许对谓词公式进行建模，跟类型 **exp** 递归定义，有以下几种类型：

1. 真值构造器 `Chaos` ，代表公式值为真；
2. 假值构造器 `Miracle` ，代表公式值为假；
3. 等值构造器 `Eqn` ，需要指定两个表达式，表示断言这两个表达式相等；
4. 否定构造器 `Neg` ，表示否定一个公式；
5. 连续合取构造器 `AndList` ，需要指定一个公式列表，表示这些公式之间是合取的关系；

6. 连续析取构造器 `OrList` ，需要指定一个公式列表，表示这些公式之间是析取的关系；
7. 蕴含构造器 `Imply` ，需要指定两个公式，表示前者蕴含后者；
8. 全称量词构造器 `ForallFormula` ，需要指定参数列表和一个带参公式，表示该带参公式的实例之间是合取关系；
9. 存在量词构造器 `ExistFormula` ，需要指定参数列表和一个带参公式，表示该带参公式的实例之间是析取关系。

语句 statement 允许对语句建模，有以下形式：

1. 简单赋值构造器 `Assign` ，需要指定一个变量和一个表达式，表示将该表达式的值赋给这个变量；
2. 并行赋值构造器 `Parallel` ，需要指定一个语句列表，这些语句表示的赋值是并行执行的；
3. 条件赋值构造器 `IfStatement` ，需要指定一个表示条件的公式和满足条件时的动作语句；
4. 条件赋值构造器 `IfelseStatement` ，与上类似，只是还需要一个不满足条件时的动作语句；
5. 带参赋值构造器 `ForStatement` ，需要指定一个带参语句和一个参数列表，这个带参语句的实例是并行执行的。

性质 prop 允许使用构造器 `Prop` 对带参安全性质建模，由性质名称、参数列表、带参公式构成；

规则 rule 允许使用构造器 `Rule` 对带参规则建模，由规则名称、参数列表、卫式和语句构成。

3.1.3 顶级定义

表 3.1 中的第三部分为 ParaVerifier 形式化语义模型的顶级定义，这个定义用来对带参系统本身进行建模，即六元组 $< name, types, vardefs, init, rules, properties >$ ，需要指定描述带参系统的六大组件。

3.1.4 语义模型的建立

ParaVerifier 的形式化语义模型的表达能力与 Murphi 语言相当，我们已经实现了一个简单编译器 *murphi2ocaml* ，来完成从 Murphi 模型提升到语义模型的工作，这个工作大部分都可以使用普通的编译知识解决。但是需要注意的是 Murphi 的建模语言支

持函数（function）和过程（procedure），而 ParaVerifier 并没有显式支持这些结构。利用 OCaml 语言的函数能够支持类似的表达，需要解决的问题有三个，一是如何完成参数传递，二是如何调用函数或者过程，三是如何完成函数返回值的回传。

1. **参数传递** 不带参的简单变量、复合变量前缀均可直接传递（当传递的参数为复合类型时可能会传递复合变量前缀，如传递 ex 而使用 ex.com1 和 ex.com2 ）；带参变量也可直接传递，但是最好在变量命名中暗示参数的名称，如 cache_i 暗示参数名称为 i 。

2. **调用** 不带参的简单变量、复合变量前缀需要以列表的形式直接传递，如 [global "ex"] ，列表形式的目的是方便在函数或过程中使用完整的复合变量；带参变量传递时需要引用其参数，如 [arr [("cache", [paramref "i"])]] ，这个参数实际上是定义在函数或者过程内部的。

3. **返回值回传** 过程返回的是一系列语句，直接使用即可；函数返回的是一个表达式，也可以直接使用，生成这个表达式的策略是：先消去带参语句，再消去量词，然后对函数内的语句进行符号执行，将变量与值对应，最后在这些变量中查找需要返回的变量，返回其对应的值即可。

其他需要注意的是对定义和引用的处理、对非参量的处理和对自定义数据结构的处理。

对定义和引用的处理会出现在两个方面，一个是变量，一个是参数，但是处理方法是类似的。因为定义和引用一般会在语法树的不同位置出现，因此只需要区别处理，也就是如果是在定义的语法位置出现则转换为定义，否则就是引用。

一般地，在 Murphi 中，刻画带参系统的参量会出现于规则、不变式、forall / exists 量词和 for 语句中，但是偶尔这里也会出现实际上没有刻画带参特性但是具有参数形式的量。这时只需要先将其按照正常的参量做处理，然后再将非参数类型展开，最后由于展开后依然保持参量形式，因此还需要将其转化为普通的常量。

处理自定义数据结构有两种方案：一种是在语法分析阶段就将复合数据结构展开为多个变量，在转换到内部模型中；另一种是先保留复合数据结构的形式，转换完毕后再在内部模型中将复合数据结构展开。两种方案的复杂度相近，但由于在实现上，我们使用函数式语言处理内部模型，进行复合变量展开有一定优势，因此采用第二种方案。

3.2 NuSMV 自动建模策略

因为内部的形式化语义模型是从语义角度描述带参系统的，因此实际上它可以转换到任何具有相同语义的语言的模型，当然也包括 NuSMV 语言，需要做的就是提出相应的转换策略。一般化的转换策略就是根据目标建模语言的语法，直接根据形式化语义模

型的语义节点生成相应的语法字符串。然而每种具体的建模语言都自有其特点，还需要根据这些特点具体提供转换策略。下面提供针对 NuSMV 建模语言的特殊策略。

3.2.1 分支语句

在 Murphi 中，存在与 C 语言等高级语言含义类似的分支语句，但是在 NuSMV 中并不支持分支语句，而是用条件表达式取而代之。图 3.2 给出了两者实现相同功能的一个例子。

Murphi	NuSMV

`if a < 10 then`
`  x := 1; y:= 2;`
`elif a < 20 then`
`  x := 2;`
`else`
`  x := 3;`
`endif;`
	`next(x) := case`
	`  a < 10: 1;`
	`  a < 20: 2;`
	`  TRUE: 3;`
	`esac;`
	`next(y) := case`
	`  a < 10: 2;`
	`  TRUE: y;`
	`esac;`

图 3.2: Murphi 和 NuSMV 对分支的不同支持方式

ParaVerifier 的形式化语义模型也支持分支，因此 Murphi 的分支语句可以直接提升到语义模型中。但是要再转换到 NuSMV，必须对分支语句做出处理。具体地，首先拆分分支语句的语句体，使每个分支语句仅包含对一个变量的赋值；然后合并嵌套的分支语句，使其成为简单的形式；最后为不完整的分支添加 `else` 分支，成为完整的分支语句。这样就可以直接将处理后的分支语句转换为 NuSMV 中的条件表达式。

3.2.2 规则

NuSMV 中不能直接支持规则，但是可以使用 `process` 变量结合模块来模拟规则。具体地，为每个规则定义一个模块，在这个模块中对该规则所涉及的迁移进行说明。然后定义对应的 `process` 变量来模拟规则实例，该变量在 NuSMV 中的行为就会类似于 Murphi 中的规则。存在的两个问题是对规则卫式的支持，以及如何提供规则所涉及到的所有变量。

第一个问题比较简单，可以使用 NuSMV 的条件表达式来模拟前置条件；规则卫式即条件，目标值为条件表达式的真值，而变量本身为条件表达式的假值，在所定义的模块中以条件表达式取代目标值即可。

这里重点解决第二个问题，即提取规则内会用到的所有的变量。从最细的粒度观察，可以发现变量都是存在于表达式中的，而表达式的存在形式中，常量和参数引用都不是变量，变量表达式可以直接提取出一个变量，条件表达式可以递归地从其条件和两

个值中分别提取出变量并给出并集。更粗粒度的结构只需要将其拆分为细粒度的结构，直到拆分到表达式级别上分别进行提取，最后取各个不同结构所得结果的并集，这一过程可以使用递归方便地实现。

3.2.3　其他

与大部分编程语言不同的是，NuSMV 的状态变量可以包含部分特殊符号，包括半角的中括号和点号，因此可以方便地表示展开后的自定义数据结构。如 `cache[1].state` ，在大部分的语言中是不合法的，但是在 NuSMV 中是合法的。

另外，NuSMV 本身是不支持带参的，因此在转换之前需要实例化参数。因为 NuSMV 是自动生成的，因此只需要修改对应的 Murphi 模型，就可以方便地将维护工作同步到 NuSMV 模型中，从而也使 NuSMV 的建模间接参数化。

3.2.4　转换结果分析

如表 3.2 所示，我们对一系列的带参系统进行了实验，通过其 Murphi 模型自动建立其对应的 NuSMV 模型，且得到的 NuSMV 模型能够满足后边进行符号模型检测工作的需要的需要。

	Murphi 模型大小/KB	数据类型数目	规则数目	NuSMV 模型大小/KB
MutualEx	0.6	2	4	2.5
MESI	0.9	2	4	5.2
MOESI	1.0	2	5	4.9
Germanish	2.1	3	6	14.6
German	4.1	6	13	26.1
Flash	18.6	19	62	546.2

表 3.2: 一些带参系统的 NuSMV 自动建模实验结果

我们已经把表 3.2 中所有实例的 Murphi 模型和自动生成的 NuSMV 的模型放在网上[42] 。不难发现，NuSMV 模型的规模与原始 Murphi 模型的规模、自定义数据类型的多少和规则数目都是正相关的，表明随着带参系统复杂度的提高，NuSMV 模型的规模也急剧增大；另外，NuSMV 模型规模也都显著大于 Murphi 模型，对这样的模型手工进行维护，几乎是不可能的，但是这里可以通过维护其对应的 Murphi 模型后进行转换，间接维护 NuSMV 模型，也使得 NuSMV 模型有了间接参数化的能力。

3.3　本章小结

本章首先提出了一个形式化语义模型，用来支持对带参系统的语义建模。在已有 Murphi 模型的情况下，可以使用编译器 *murphi2ocaml* 自动将 Murphi 模型提升到语义模型。通过提供针对性的转换策略，可以方便地自动将语义模型转换为其他建模语言所

描述的模型，如 NuSMV。这项工作方便了对复杂带参系统的多工具建模和维护，也使不支持参数化的 NuSMV 工具间接参数化。通过类似的方法，也可以将形式化语义模型转换为其他建模工具所要求的版本。

第四章　ParaVefier 中的符号模型检测和可满足性检查

ParaVerifier 在生成辅助不变式时，一开始得到的是一个很弱的不变式，然后逐步强化，强化过程中需要不断检查不确定参数的不变式候选是否为真正的不变式，因此需要借助模型检测工具 NuSMV 或者 Murphi 结合符号化方法进行符号模型检测。另外，在验证过程中，需要不断对各种谓词公式进行可满足性检查，可以使用可满足性求解工具 Z3。在 ParaVerifier 中，通过使用一个如图 4.1 所示的客户端/服务器架构，来调用这些第三方工具，以达到提高查询效率、分解大型任务、实现结果复用等目的。

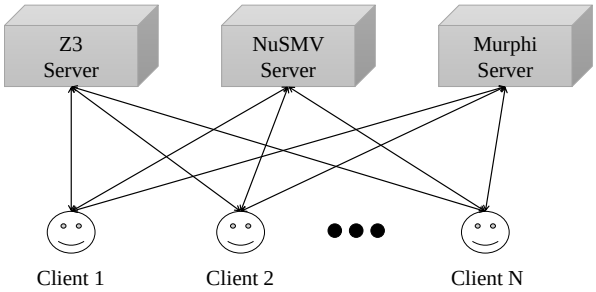

图 4.1: ParaVerifier 中的客户端/服务器架构

4.1　第三方工具的调用接口

一般地，第三方工具提供的调用接口有应用程序编程接口（API）调用和进程调用两种形式，其中后者又包括在线调用和离线调用。而在 ParaVerifier 需要使用的工具中，Z3 提供 API 调用和离线调用两种方式，NuSMV 需要在线调用，而 Murphi 只能离线调用。

4.1.1　API 调用

接口名称	接口参数	返回值说明
parse_smt2_string	SMT2 字符串	SMT2 对象
Solver	-	Solver 对象
Solver.add	SMT2 对象	-
Solver.check	-	可满足 `"sat"`；不可满足 `"unsat"`

表 4.1: Z3 工具的部分 Python 接口

虽然 Z3 工具提供两种调用方式，但是优先选用 API 调用方式，因为这种方式避免了多进程带来的额外开销，拥有更好的性能。Z3 的 API 提供 C/C++/Python/Java 等

多种语言的版本，ParaVerifier 采用 Python 版本的 Z3 工具 API，所需要使用的接口如表 4.1 所示。其中，SMT2 字符串使用类 Lisp 语言描述，是 Z3 工具的输入，由两部分组成，分别是上下文环境和公式检查。

1. 上下文环境

   - 通过 `declare-datatypes` 函数定义枚举类型；
   - 通过 `declare-fun` 函数定义带参或者不带参的状态变量。

2. 公式检查

   - 通过预定义的内部函数来描述一个谓词公式，如条件表达式 `ite` 、真值 `true` 、假值 `false` 、等值 `=` 、非 `not` 、合取 `and` 、析取 `or` 和蕴含 `=>` ；
   - 通过 `assert` 函数完成对一个谓词公式的断言；
   - 通过 `check-sat` 来检查一个断言的可满足性。

对于某个特定的带参系统，可能需要多次进行可满足性检查，每次检查的时候都需要生成相应的 SMT2 字符串，包括其中的上下文环境部分和公式检查部分。但是对于这些检查来说，上下文环境部分是完全一致的，只有公式检查部分是不同的。

4.1.2 在线调用

NuSMV 工具通过在线调用的方式提供了一种快速检查一个关于某个带参协议的实例公式是否为不变式的方法，以这种模式启动 NuSMV 的时候需要使用 `-dcx -int -old` 这些开关。这种调用方式的特点是，首先需要进入一个交互式终端，在此终端内使用 NuSMV 的内置命令来完成所期望的工作，对每个命令，NuSMV 会给出相应的输出结果，在交互工作完成后，也可以使用一条内置命令退出交互式终端。ParaVerifier 中会使用到的内置命令如表 4.2 所示。

命令名称	命令含义	命令参数	命令输出
go	检查文件语法	-	-
compute_reachable	计算可达集	-	可达集半径
check_invar	检查不变式	-p < 字符串形式的公式 >	是否为不变式
quit	退出	-	-

表 4.2: NuSMV 工具交互式模式下的部分内置命令

在使用交互式模式下的 NuSMV 时，一般步骤是：

1. 以交互式模式启动 NuSMV，即在终端输入 `NuSMV -dcx -int -old <smv_file>` ；

2. 输入命令 `go` 检查文件是否合法；

3. 输入命令 `compute_reachable` 计算可达集，对于较复杂的系统，这一步可能耗时较长，甚至无法计算；

4. 使用命令 `check_invar` 检查不变式，每个需要检查的不变式都需要执行一次这条命令，在执行这条命令时，NuSMV 会使用 BDD 快速计算，在极短的时间内就可以判定该公式是否为真的不变式；

5. 检查完毕后使用命令 `quit` 退出 NuSMV。

显然，对于某个特定的带参系统，使用 NuSMV 多次检查不变式时，也有一个固定的上下文环境，即启动 NuSMV 时所需要的 SMV 文件。

4.1.3 离线调用

Murphi 工具仅提供一种离线调用的方式来检查不变式。离线调用的特点是，每次检查都需要准备输入文件，启动 Murphi 工具相应的一系列进程，等待运行，获取运行结果且进程立即退出，如图 4.2 所示。这种调用方式速度慢，开销大，但是在带参系统的实例过大时可以发挥作用。

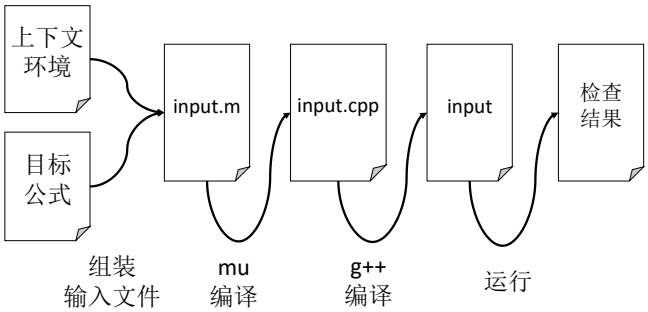

图 4.2: Murphi 的离线调用过程

从图 4.2 中可以看出，对于某个特定的带参系统，使用 Murphi 多次检查不变式，同样有一个固定的上下文环境。另外，调用 Murphi 工具进行一次不变式检查，需要三次离线调用，分别是调用 Murphi 编译器 `mu` 编译 `.m` 文件生成 `.cpp` 文件，然后调用 C++ 编译器 `g++` 编译相应的 `.cpp` 文件生成可执行文件，最后运行可执行文件获取结果。这三次调用涉及了三次进程的创建和销毁，开销比较大。另外，Murphi 每次进行不变式检查都需要遍历整个状态空间，对于复杂的系统需要申请较大的内存，进一步降低了性能。

4.2 第三方工具服务的创建

ParaVerifier 是将 NuSMV、Murphi 和 Z3 这些第三方工具作为网络服务调用的，因此首先需要将这些工具的调用做成网络服务的形式。这是通过 Python 语言的 `socket` 包实现的。

4.2.1 创建服务的一般过程

使用 Python 的 `socket` 创建网络服务，首先需要创建一个 socket 对象，一般需要指定两个参数，即网络地址的类型和 socket 的类型。这两个参数的常用取值与含义分别如表 4.3 和表 4.4 所示[43] 。

名称	含义
socket.AF_INET	默认，地址由主机（域名或者 IP 地址）和端口（数字）组成
socket.AF_INET6	地址由 IPv6 地址主机和数字端口组成
socket.AF_UNIX	地址是 UNIX 风格的文件系统节点

表 4.3: 网络地址的类型

名称	含义
socket.SOCK_STREAM	默认，传输层使用 TCP 协议，有序、可靠、双向、面向连接
socket.SOCK_DGRAM	传输层使用 UDP 协议，固定长度、无连接、不可靠
socket.SOCK_RAW	IP 协议的数据报接口

表 4.4: socket 的类型

之后需要设置 socket 选项，将 socket 对象绑定到相应的地址和端口，然后开始监听客户端网络的连接请求。对于客户端的每个连接请求，服务器端都可以使用 `accept` 函数取得连接对象和客户端地址，这个连接对象可以从客户端接受数据以及向客户端回送数据。

4.2.2 第三方工具服务的具体创建

因为用到的三个第三方工具，调用方式俱不相同，因此需要针对每种调用方式的特点，分别制定其服务的创建策略。

1. **Z3 服务** Z3 工具提供 Python 版本的 API，不需要创建相关的进程，因此只需要直接在服务端调用 API 进行判断即可，也拥有最高的性能；

2. **NuSMV 服务** NuSMV 使用在线调用的方式，在其交互式终端，如果有无穷次输入，那么就会产生无穷多的输出，因此其输出类似于一个无限大的文件，与无限大的文件交互需要使用伪终端相关技术，Python 的第三方库 `pexpect` 提供了相应的功能。

3. **Murphi 服务** Murphi 使用离线调用的方式，且每次检查都需要三次离线调用，因为这种调用的输出类似于普通文件，因此只需要 Python 的子进程接口即可实现检查并读取输出的功能。

4.2.3　服务接口

在 ParaVerifier 中，如果客户端连接服务器成功，那么客户端可以向服务器发送以字符串表示的命令。每个命令的模式都为

```
<length>, <command>, <command_id>, [options]...
```

其中 `length` 为命令长度，`command` 为命令名称，`command_id` 为客户端的命令标志，`options` 对每个命令都不同，个数和含义分别由每个命令决定。注意命令的每个部分都是用半角逗号分隔的。

对客户端发出的每条命令，如果合法，那么服务器就会返回一个状态码表示执行是否成功，如果需要回送数据，数据会附加在后边，同样以英文半角逗号分隔。现在可用的命令以及服务器返回的状态码，如表 4.5 所示。

代码	含义	说明
4	设置 Z3 上下文	需要提供系统名称和 Z3 上下文代码
5	使用 Z3 检查可满足性	需要提供系统名称和 SMT2 公式
1	计算可达集	设置 NuSMV 服务上下文并计算可达集，需要提供系统名称和 NuSMV 模型
2	查询可达集	查询 NuSMV 服务是否已经算得可达集，需要提供系统名称
3	使用 NuSMV 检查不变式	需要提供系统名称和不变式
7	退出	结束某系统的 NuSMV 服务，需要提供系统名称
8	设置 Murphi 上下文	需要提供系统名称和 Murphi 模型
9	使用 Murphi 检查不变式	需要提供系统名称和不变式
-2	错误	命令有误
-1	等待	服务器忙
0	正常	命令执行成功

表 4.5: 可用的命令及状态码

启动服务器的命令是

```
python server.py [-v]
```

其中 `-v` 开关是可选的，如果启用，那么会在终端实时回显所接收到的命令以及命令的执行结果，以方便调试，但因为使用了系统 IO，会一定程度上降低性能。

另外，每个被启动的服务都可以同时充当 Z3 服务器、NuSMV 服务器或者 Murphi 服务器，具体角色由客户端的要求决定。这在分解服务器任务上提供了较高的灵活性，如果客户端执行的是小任务，那么可以只使用一台服务器，如果客户端执行的是复杂的

大型任务，那么可以把第三方工具检查的工作指派给多台不同的服务器，从而实现一定程度上的负载均衡。

4.3 第三方工具服务的使用

调用第三方工具服务的是 ParaVerifier 中的一个子模块，也使用 OCaml 语言编写。对每个需要服务器执行的命令，都通过一次网络请求发送给服务器。首先，与服务器的创建类似，同样需要一个 socket 对象，然后指定服务器的地址和端口进行连接，连接成功后将需要服务器执行的命令发送给服务器并等待结果。

与用到的三个第三方工具的服务对应，也需要创建三种类型的客户端，其中 Z3 客户端提供设置上下文和检查公式可满足性的接口；NuSMV 客户端提供设置上下文并计算可达集、查询可达集和检查不变式的接口；Murphi 客户端提供设置上下文和检查不变式的接口。

对于一个带参系统的验证任务，ParaVerifier 首先根据指定的各个服务器的地址和端口分别设置上下文，然后开始各种检查任务。如果没有给定服务器的地址和端口，则默认使用本地主机。

4.4 本章小结

本章首先介绍了 ParaVerifier 进行符号模型检测和可满足性检查所需要的第三方工具的特点，然后说明了将这些第三方工具做成网络服务的策略，最后说明了使用这些网络服务的策略。本章是 ParaVerifier 高效完成形式化验证任务的基础。

第五章　ParaVerifier 的设计和实现

ParaVerifier 被设计为一个尽可能避免人工介入的高性能带参系统验证工具，以类 Murphi 建模语言作为输入，经过编译器处理后提升为内部的形式化语义模型。在此语义模型的基础上，确定一个小的系统实例，然后进行预处理，使得模型的规则、不变式能够满足 ParaVerifier 进行后续工作的需要，另外，从语义模型也可以转换得到需要使用的其他语言版本的模型，如 NuSMV 和 Z3。然后基于引理 2.2 ，应用一系列的启发式策略，不断判断规则和不变式之间的因果关系，查找辅助不变式候选并使用第三方工具判定，直到这一过程收敛。最后在因果关系和不变式的基础上，基于一致性引理 2.1 生成 Isabelle 证明脚本，完成证明。整个过程如图 5.1 所示。

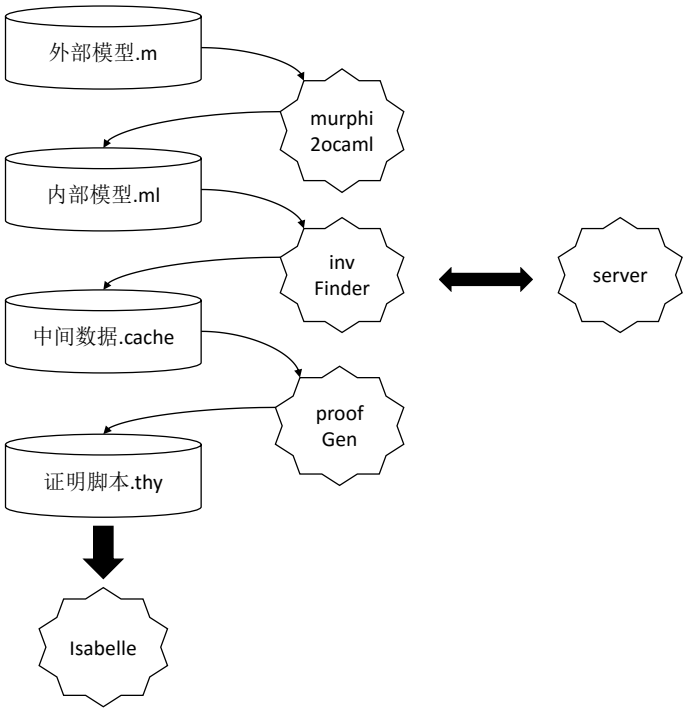

图 5.1: ParaVerifier 的验证过程

其中，子工具 *murphi2ocaml* 已经在第三章介绍过，子工具 *server* 也在第四章介绍过；子工具 *invFinder* 有两个作用，一个是对语义模型进行预处理，另一个是生成因果关系和辅助不变式，这一过程需要 *server* 的帮助；子工具 *proofGen* 会基于语义模型、因果关系和不变式，生成定理证明脚本；工具 *Isabelle* 是第三方工具，用于最终执行定理证明脚本，完成验证过程。

5.1 预处理

从 Murphi 建模语言提升而来的语义模型并不能被 ParaVerifier 直接处理，因为其规则、不变式一般具有复杂的形式。因此要先对语义模型进行预处理，使其成为更简单的形式。

5.1.1 规则的预处理

带参规则由四部分构成，分别是规则名称、规则参数列表、卫式和语句。实例化后，规则的参数列表部分将会消去，带参规则本身也会变成若干个规则实例，原先可能会包含参数的卫式和语句部分，其参数也将会具体化。但此时的规则仍然不一定是简单规则实例，简单规则实例的定义如定义 5.1 所示。

定义 5.1.
如果一个规则实例的卫式部分是合取的形式，且每个合取分量要么是等值表达式，要么是等值表达式的否定形式，那么这样的规则实例就是简单规则实例。

简单规则实例的意义在于，一条普通的规则实例描述了在哪些具体条件下将会发生何种具体的迁移，而简单规则实例则仅描述了在某一种具体条件下会发生何种具体的迁移，也就是说，相对于普通规则实例，每个简单规则实例都有更强的卫式。将一个普通规则实例预处理为简单规则实例，需要以下步骤：

1. 将规则实例的卫式部分规范化为析取范式；

2. 针对析取范式的每一个合取子句，构造一个简单规则实例，其卫式是该合取子句，语句是原规则实例的语句；

3. 用新构造的这些简单规则实例替换原规则实例。

将普通规则实例变为简单规则实例集，也是为了后边在生成因果关系和辅助不变式时，规则实例的卫式与不变式有相同的形式，从而方便计算。

5.1.2 不变式的预处理

不变式，也就是带参系统的安全性质，一般也是带参的形式，一般由性质名称、参数列表和带参公式构成。带参不变式一个例子如下：

$$i \neq j \rightarrow (node_i = Exclusive \rightarrow node_j \neq Exclusive), 1 \leq i, j \leq N$$

同样地，实例化后，这个带参不变式也会变成一系列的不变式实例，以参数 N 取 2 为例，上述不变式会被实例化为：

$$1 \neq 1 \rightarrow (node_1 = Exclusive \rightarrow node_1 \neq Exclusive)$$
$$1 \neq 2 \rightarrow (node_1 = Exclusive \rightarrow node_2 \neq Exclusive)$$
$$2 \neq 1 \rightarrow (node_2 = Exclusive \rightarrow node_1 \neq Exclusive)$$
$$2 \neq 2 \rightarrow (node_2 = Exclusive \rightarrow node_2 \neq Exclusive)$$

显然，以上第一式和最后一式都是永真式，没有意义；中间两式可以化简如下

$$node_1 = Exclusive \rightarrow node_2 \neq Exclusive$$
$$node_2 = Exclusive \rightarrow node_1 \neq Exclusive$$

很明显，这两式是对称的，也就是说，它们的区别仅仅体现在编号上。实际上，在 ParaVerifier 中，公式实例对称的概念更广泛。首先如定义 5.2 所示，给出同构公式的定义；然后再如定义 5.3 所示，给出对称公式的定义。

定义 5.2.
如果两个公式的实参经过轮换后，这两个公式相同，那么这两个公式就是同构的。

定义 5.3.
对于公式实例 f 和公式实例 g，如果分别存在其同构公式 f' 和 g'，使得 $f' \leftrightarrow g'$，那么公式实例 f 和公式实例 g 就是对称的。

公式实例间的对称关系是一种等价关系。

在 ParaVerifier 中，一般使用的是不变式的否定形式，经过处理后，上述例子的不变式实例成为如下形式，也是对不变式进行预处理的最终形式：

$$node_1 = Exclusive \wedge node_2 = Exclusive$$

注意，这个是不变式的否定形式，也就是说，对其取否定才是真正的不变式。在这个例子中，一个带参不变式最终被处理为一个不变式实例，但实际上有可能最终仍然存在多个不变式实例。

综上所述，对带参不变式进行预处理的过程如下：

1. 将带参不变式实例化为不变式实例集；
2. 将每个不变式的实例改写为其否定形式；
3. 对不变式实例集中的每个实例，将其规范化为析取范式，然后用析取范式中的合取子句集代替原不变式实例；

4. 剔除永假式；

5. 以公式实例间的对称关系为等价关系，划分等价类，每个等价类仅保留一个公式实例，其余剔除。

对不变式进行预处理，一方面是为了让不变式有更简单的形式，便于计算；另一方面，是为了减少不变式实例的数目，降低复杂度。

5.2 因果关系的生成

对于一个带参系统实例的不变式集和规则集，存在定义 2.3 中所示的三种因果关系。考虑不变式集 $invs$ 的一个子集 $invs' \subset invs$ ，如果存在不变式 $f' \in invs'$ ，使得其与某条规则 r 不满足关系 $invHoldForRule1$ 与关系 $invHoldForRule2$ ，那么根据引理 2.2 ，它们必然满足关系 $invHoldForRule3$ ，也就是说，必存在不变式 $f \in invs$ ，使得 $(f \wedge g) \rightarrow preCond(f', \alpha)$ ，其中 g 和 α 分别为规则 r 的卫式和语句。如果 $f \notin invs'$ ，可以设法找出这个辅助不变式，将其置入 $invs'$ 中，完成关系 $invHoldForRule3$ 的生成。查找辅助不变式的具体方法见第5.3节。

5.2.1 选择不变式实例和规则实例的启发式方法

在考虑规则和不变式之间的因果关系时，所用到的规则和不变式分别是带参规则的实例和带参不变式的实例，因此可以充分利用参数的对称特性这一启发式信息，减少搜索空间，如启发式策略 5.1 所示。

启发式策略 5.1.

带参系统一般包含多个完全相同的主体，这些主体使用参数刻画。令规则实例 $r(m,n)$ 表示有两个主体参与的活动，令不变式 $f(i)$ 表示关于一个主体的性质，那么在考虑生成这样的不变式和规则的因果关系的时候：

1. 不变式实例只需要考虑 $f(1)$ ，因为其他的实例诸如 $f(2), f(3), \cdots$ 与 $f(1)$ 是对称轮换的。

2. 规则实例只需要考虑 $r(1,2), r(2,1), r(2,3)$ 这三种，它们分别是某一类规则实例的代表：

   - $r(1,2)$ 代表第一个参数与不变式实例 $f(1)$ 相同的所有规则实例；
   - $r(2,1)$ 代表第二个参数与不变式实例 $f(1)$ 相同的所有规则实例；
   - $r(2,3)$ 代表两个参数均与不变式实例 $f(1)$ 不同的所有规则实例。

这样，原本可能需要考虑无穷多个不变式和规则的实例，而现在只需要考虑 1 个不变式实例和 3 个规则实例。算法 1 描述了应用这种启发式策略给出的规则实例选择方法。

算法 1: 规则实例选择算法 SemiPerm

Input: 带参不变式的参数数目 N_f，带参规则的参数数目 N_r ，带参规则 r

Output: 规则实例集 R^{inst}

```
1  m ← N_f + N_r ;
2  n ← N_r ;
3  if m = 0 then
4      return empty set ∅;
5  else if n = 0 then
       // r is a 0-parameter rule
6      return {r};
7  else
8      l ← n-permutation of number set {i|1 ≤ i ≤ m};
9      foreach p in l do
10         remove all the other items equal to p in l;
       // r(p) means instantiate rule r with parameter(s) p
11     return {r(p)|p ∈ l};
```

5.2.2　因果关系的生成算法

因果关系是规则实例和不变式实例之间的多对多关系，其生成是一个图搜索过程，可以选用广度优先搜索或者深度优先搜索。ParaVerifier 使用广度优先搜索，如算法 2 所示。

在算法 2 中，*SemiPerm* 即算法 1 ，*preCond* 即定义 2.2 所给出的算子，而 *FindInv* 将在第5.3节中给出，另外所有的不变式都是否定形式的。这个算法会从初始不变式实例集出发，不断比对不变式实例与规则之间的因果关系，同时也查找新的辅助不变式，而新的辅助不变式又可以继续与规则之间产生新的因果关系，不断进行这一搜索过程，直到收敛，就此得到一个因果关系集和不变式实例集。

通过算法 2 不难发现，在搜索过程中，使用了多个层次的循环，因此所发现的因果关系也是有明显层次的，具体表现为：

1. 对每个不变式实例，均需判断其与带参规则之间的因果关系；
2. 对每个带参规则，均需要合适的实例化；
3. 对每个规则实例，要考虑其语句内的所有分支；
4. 对每个规则实例的一个分支，判断其与不变式实例的因果关系。

算法 2: 因果关系生成算法 RelationGen

Input: 带参规则集 R，初始不变式实例集 $invs_0^{inst}$

Output: 因果关系集合集合 C 和辅助不变式实例集 $invs^{inst}$

$C \leftarrow \emptyset$;
$invs^{inst} \leftarrow invs_0^{inst}$;
$q \leftarrow invs_0^{inst}$ as a queue;
while $q \neq \emptyset$ **do**
  $f^{inst} \leftarrow q.Dequeue()$;
  $N_f \leftarrow$ number of parameters in f^{inst} ;
  foreach $r \in R$ **do**
    $N_r \leftarrow$ number of parameters in r ;
    $R^{inst} \leftarrow SemiPerm(N_f, N_r, r)$;
    foreach $r^{inst} \in R^{inst}$ **do**
      $g \leftarrow$ guard of rule r^{inst} ;
      $\alpha \leftarrow$ statement of rule r^{inst} ;
      // there always are branches in α
      // so result of $preCond$ is a set B
      $B \leftarrow preCond(f^{inst}, \alpha)$;
      // b is a branch, c is the corresponding result
      foreach $(b, c) \in B$ **do**
        if c *is symmetric with* f^{inst} **then** // invHoldForRule2
          $C \leftarrow C \cap \{invHoldForRule2(r^{inst}, f^{inst}, b)\}$;
        else if $g \rightarrow \neg c$ **then** // invHoldForRule1
          $C \leftarrow C \cap \{invHoldForRule1(r^{inst}, f^{inst}, b)\}$;
        else // invHoldForRule3
          $\hat{f}^{inst} \leftarrow FindInv(g, b, c)$;
          if $\hat{f}^{inst} \leftrightarrow \top$ **then** // in fact invHoldForRule1
            $C \leftarrow C \cap \{invHoldForRule1(r^{inst}, f^{inst}, b)\}$;
          else if $\exists h^{inst} \in invs^{inst}.h^{inst} \rightarrow \hat{f}^{inst}$ **then** // old
            $C \leftarrow C \cap \{invHoldForRule3(r^{inst}, f^{inst}, b, h^{inst})\}$;
          else // new
            $C \leftarrow C \cap \{invHoldForRule3(r^{inst}, f^{inst}, b, \hat{f}^{inst})\}$;
            $invs^{inst} \leftarrow invs^{inst} \cap \{\hat{f}^{inst}\}$;
            $q.Enqueue(\hat{f}^{inst})$;
return $(C, invs^{inst})$;

5.3 不变式查找

考虑不变式集 $invs$ 的一个子集 $invs' \subset invs$ ，当不变式实例 $f' \in invs'$ 与规则实例 r 满足因果关系 *invHoldForRule3* 时，必存在不变式 $f \in invs$ ，使得 $(f \wedge g) \rightarrow preCond(f', \alpha)$ ，其中 g 和 α 分别为规则 r 的卫式和语句。如果 $f \notin invs'$ ，就需要找到这个辅助不变式。

对公式 $(f \wedge g) \rightarrow preCond(f', \alpha)$ 进行变形，得到

$$(g \wedge \neg preCond(f', \alpha)) \rightarrow \neg f$$

而 $preCond(f', \alpha)$ 只是对 f' 里的变量进行了替换，并没有改变其逻辑连接词，因此

$$\neg preCond(f', \alpha) \Leftrightarrow preCond(\neg f', \alpha)$$

从而

$$(g \wedge preCond(\neg f', \alpha)) \rightarrow \neg f$$

而经过第5.1节的预处理，$\neg f$ 和 $\neg f'$ 都是合取的形式，规则卫式 g 也是合取的形式，因此必然能够通过弱化 $g \wedge preCond(\neg f', \alpha)$ 得到辅助不变式的否定形式 $\neg f$ 。

5.3.1 判定不变式的启发式方法

在查找辅助不变式的时候，首先生成一系列不变式实例的候选，然后使用第三方工具 NuSMV 和 Murphi 的服务判定这些实例是否为真正的不变式。使用 NuSMV 判定不变式需要先计算出带参系统实例的可达集，在此基础上可以快速确定不变式的候选是否为真正的不变式，缺点是复杂系统的可达集不易算出；而使用 Murphi 可以完成对较大规模系统的不变式的判定，缺点是每次判定都需要遍历整个可达集，需要消耗很长的时间才能确定一个正确的不变式，不过一般可以相对较快地判定一个错误的不变式。ParaVerifier 对不变式候选的判定方法如启发式策略 5.2 所示。

启发式策略 5.2.

对于一个带参系统 $P(N)$ ，可以通过试探得到一个阈值参数 T_p ，使得对任意参数 x ，若 $1 \leq x \leq T_p$ ，那么带参系统实例 $P(x)$ 的 *NuSMV* 模型可以算得可达集。另外，记拥有 n 个参数的公式实例为 $f^{(n)}$ 。那么可以按照如下策略判定公式 $f^{(n)}$ 是否为不变式：

1. 如果 $n \leq T_p$ ，那么使用上下文环境为 $P(T_p)$ 的 *NuSMV* 服务进行判定，可以快速得到判定结果；

2. 如果 $n > T_p$ ，那么使用上下文环境为 $P(K)$ 的 *Murphi* 服务进行判定，其中 $K \geq n$ ，对于一个在 *Murphi* 服务中设置的时间阈值 T_t ，判定结果有以下情况：

   - 在时间阈值 T_t 内，得到判定结果，以判定结果为准；

- 超出时间阈值 T_t 后，中断判定过程，认为 $f^{(n)}$ 是不变式，如果时间阈值 T_t 足够大，这种判定的可信度是比较高的。

这种策略实际上是截止方法的一种应用，可以对原本不可判定的问题给出一种解决方案。算法 3 描述了使用这种启发式策略对不变式候选进行判定的方法。

算法 3: 不变式候选判定算法 JudgeInv

Input: 不变式实例候选的否定形式 f^{inst} ，参数阈值 T_p

Output: 布尔类型的判定结果，$\top$ 表示是不变式，$\bot$ 表示不是不变式

```
1 g ← ¬f^inst ;
2 n ← number of parameters in f^inst ;
3 if n ≤ T_p then
4     return JudgeByNuSMV(g);
5 else
6     return JudgeByMurphi(g);
```

其中 *JudgeByNuSMV* 和 *JudgeByMurphi* 分别调用 NuSMV 和 Murphi 的服务来对一个公式进行判定。Murphi 判定过程中需要的时间阈值 T_t 在其服务中设置，相关过程也在其服务中运行。

5.3.2 不变式查找算法

查找辅助不变式的基础就是弱化公式 $g \wedge preCond(\neg f', \alpha)$ ，从这个公式出发，不断尝试候选不变式，直到发现一个真正的不变式为止。因为 g 和 $preCond(\neg f', \alpha)$ 都具有合取的形式，所以公式 $g \wedge preCond(\neg f', \alpha)$ 总体上具有 $\bigwedge_i p_i$ 的形式，其中 p_i 是等值或者等值的否定。弱化这个形式的公式有两种策略。

1. **候选递减策略** 从 $\bigwedge_i p_i$ 作为初始不变式候选，若不为不变式，则不断轮换去除某一项或者某几项，直到发现一个不变式，这种策略如算法 4 所示；

2. **候选递增策略** 从 $\bigwedge_i p_i$ 的某一项 p_i 出发，作为不变式候选，若不为不变式，则不断轮换合并某几项，直到发现一个不变式，这种策略如算法 5 所示。

算法 4 会产生较少的候选不变式，从而会有更少的试探次数，但每次都会试探一个很复杂的不变式候选，可能每次都会调用 Murphi 服务进行判定，从而降低性能；而算法 5 从简单的候选开始试探，可能在较小规模时即得到正确的不变式候选，从而有可能避免使用资源消耗较大的 Murphi 服务，总体上提高性能，因此 ParaVerifier 使用算法 5 ，即 $FindInv = FindInvInc$ 。

算法 4: 不变式查找算法 FindInvDesc

Input: 规则实例的卫式 g ，规则分支 b ，不变式否定形式的前置条件 c

Output: 辅助不变式实例 $\hat{f}^{inst}$

$C \leftarrow \{p_i |$ rewrite $g \wedge b \wedge c$ as $\bigwedge_i p_i\}$;

$\hat{f}^{inst} \leftarrow \top$;

while $C \neq \emptyset$ **do**

  $p \leftarrow$ pick an element from C ;

  $C \leftarrow C - \{p\}$;

  $f \leftarrow \hat{f}^{inst} \wedge \bigwedge_i p_i$, where $\{p_i | i\} = C$;

  if $JudgeInv(f) \leftrightarrow \bot$ **then** `// p is necessary`

    $\hat{f}^{inst} \leftarrow \hat{f}^{inst} \wedge p$;

return $\hat{f}^{inst}$;

算法 5: 不变式查找算法 FindInvInc

Input: 规则实例的卫式 g ，规则分支 b ，不变式否定形式的前置条件 c

Output: 辅助不变式实例 $\hat{f}^{inst}$

$C \leftarrow \{p_i |$ rewrite $g \wedge b \wedge c$ as $\bigwedge_i p_i\}$;

$D \leftarrow 2^C - \{\emptyset\}$;

sort D by cardinality of its elements increasingly;

foreach $D^\sharp \in D$ **do**

  $\hat{f}^{inst} \leftarrow \bigwedge_i p_i$, where $\{p_i | i\} = D^\sharp$;

  if $JudgeInv(\hat{f}^{inst}) \leftrightarrow \top$ **then** `// an invariant found`

    return $\hat{f}^{inst}$;

return $\top$;

5.4 缓存技术的应用

ParaVerifer 在很多地方使用了缓存技术，一方面可以提高查找不变式和生成因果关系的性能，另一方面通过避免重新运行带来的反复搜索过程，缩短调试阶段所用时间。

1. 在进行可满足性检查的时候，因为某些规则实例及其分支可能存在相同的卫式条件，因此会遇到重复对同一公式进行检查的情况；ParaVerifier 使用一张哈希表来缓存已经检查过的公式的结果，避免额外开销；

2. 在进行候选不变式判定的时候，不论使用候选递增策略还是候选递减策略，都会大量遇到重复判定的情况；ParaVerifier 同样使用一张哈希表来缓存已有判定结果的候选不变式，因为不变式判定的潜在开销较大，因此可以得到较好的性能提升；

3. 根据启发式策略 5.1 ，生成一个不变式实例与带参规则之间的因果关系时，规则实例是动态确定的，为了避免规则的反复实例化，ParaVerifier 将已经得到的规则实例放入哈希表中缓存，避免不必要的开销；

4. 在对复杂的带参系统开展验证工作时，每次运行都需要重新开始搜索过程，从而大大延长调试阶段；ParaVerifier 在生成一个不变式实例与所有规则的因果关系后，将得到的因果关系集、辅助不变式集和当前的搜索进度序列化后缓存为本地文件，下次运行时先读取缓存并反序列化，避免重复搜索过程。

5.5 证明脚本自动生成

充分查找辅助不变式，生成不变式集和规则集之间的因果关系，是为了能够自动生成对带参系统正确性的归纳证明。ParaVerifier 生成的 Isabelle 证明脚本由以下部分组成：

1. 基础部分，包含语义模型的 Isabelle 形式，对归纳证明的基本讨论，以及带参系统模型的 Isabelle 版本；

2. 主定理，综合所有讨论，完成证明过程；

3. 不变式和规则之间因果关系的讨论；

4. 关于不变式与初始状态之间关系的讨论。

对复杂的带参系统，最后生成的证明脚本会非常庞大，以至于无法一次载入 Isabelle 中，可以采取切分成多个文件的策略，分别进行证明后再综合证明。

5.5.1　Isabelle 模型的建立

Isabelle 形式的语义模型是 ParaVerifier 形式化语义模型的等价版本，目的是令 Isabelle 拥有等价的模型表达能力并完成带参系统的分析和归纳证明。同转换到其他建模语言的模型类似，由 ParaVerifier 的语义模型转换到 Isabelle 形式也需要建立对应的转换策略。而因为 Isabelle 形式的语义模型与 ParaVerifier 形式化语义模型拥有完全相同的表达能力，这些转换策略是平凡的，所以不再详细列出。

5.5.2　主定理

主定理的意义在于，需要说明带参系统的初始状态满足其不变式集内的所有不变式，且其不变式集与规则集满足一致性关系，那么根据引理 2.1 ，带参系统的可达状态就满足其不变式集内的所有不变式，从而完成证明。因此，主定理就分为相应的两个大组件 `lemma_invs_on_rules` 和 `on_inis` ，如图 5.2 所示。

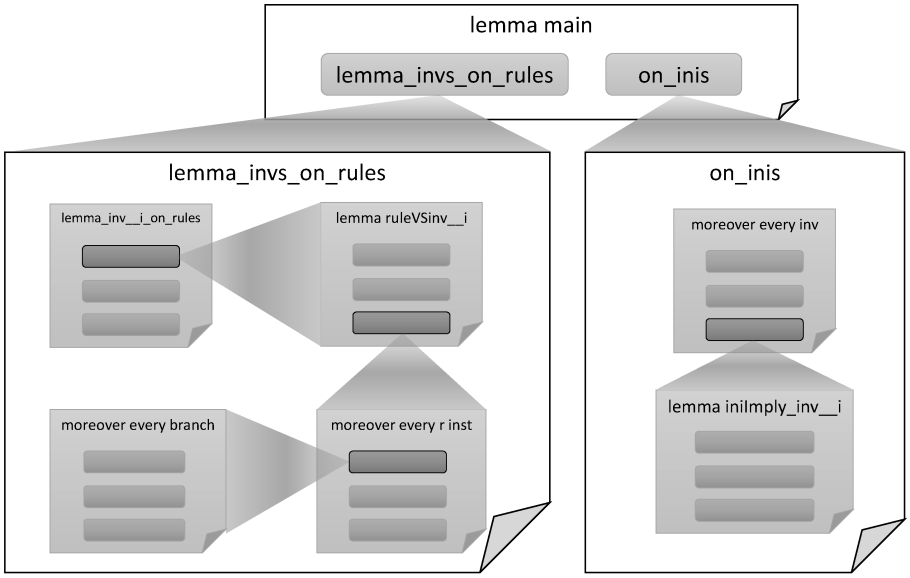

图 5.2: 因果关系的层次

主定理分别由两个大的组件构成，而每个组件又各有层次。具体到每个组件的层次，以及整个证明脚本的组织，将在下面详细阐述。

5.5.3　不变式和规则之间的因果关系

图 5.2 中的 `lemma_invs_on_rules` 示意不变式集和规则集之间的因果关系集，这个因果关系集的显著层次的形成有两个原因，其一是一条因果关系是发生在一条不变式的代表实例和一个规则代表实例的确定分支上的，其二是带参规则本身所具有的层次性。因果关系的层次直接导致了证明脚本也是分层的，自底向上体现在如下几点：

1. 一条不变式的代表实例和一个规则代表实例的确定分支，才能唯一确定一条因果关系，这是因果关系的基准情形；

2. 一个规则的代表实例，往往包含多个分支语句，由分支的条件决定规则的实际执行路径，只有在完成对所有分支下因果关系的讨论，才能完整地关于该规则实例的证明；

3. 在针对一条不变式的代表实例进行讨论时，一条带参规则往往需要分情况生成多条代表实例，证明脚本需要覆盖所有的情况；

4. 每条不变式都需要与整个带参规则集进行讨论，以生成关于该不变式的完整因果关系。

此外，`lemma_invs_on_rules` 本身还会讨论不变式集与规则集之间的所有因果关系，至此关于因果关系的完整证明脚本才算结束。

以上用到的不变式以及规则的实例称为代表实例，是因为他们分别代表了与其等价的一大类实例，从而令所有讨论不是针对几个特例的，而是更一般化的证明。这种一般化是针对参数进行的，因为参数间关系相同的带参对象是对称轮换的。这种对称轮换关系是一种等价关系。若与一条不变式 f 的代表实例等价的不变式实例集为 $\tilde{F}$ ，而与一个规则 r 的代表实例等价的规则实例集为 $\tilde{R}$ ，该不变式与规则之间的因果关系为 c ，即 $c(f,r)$ ，那么有

$$\forall f'.\forall r'.(f' \in \tilde{F} \wedge r' \in \tilde{R}) \rightarrow c(f',r')$$

算法 6 描述了如何确定一个带参对象的等价类的参数间需要满足什么关系，该算法又称泛化算法。

5.5.4 不变式与初始状态

从图 5.2 中可以看出，不变式与初始状态之间的讨论也是分层次的，这种分层是由不变式集本身的层次造成的。

1. `lemma iniImply_inv__i` 论证一条不变式在初始状态得到满足的情况，会针对初始状态条件集中的每个条件进行讨论；

2. 对不变式集中的每条不变式都做类似的讨论。

算法 6: 参数关系确定算法（泛化算法）ParamRel

Input: 不变式代表实例或规则代表实例的参数集 P

Output: 参数间需要满足的关系 h_P

```
1  h_P ← ⊤ ;
2  while P ≠ ∅ do
3      p ← pick an element from P ;
       // all elements in Q have same type with p
4      Q ← {q|q ∈ P, q ≠ p, type(q) = type(p)} ;
5      while Q ≠ ∅ do
6          v_p ← value of p ;
7          n_p ← name of p ;
8          foreach q ∈ Q do
9              v_q ← value of q ;
10             n_q ← name of q ;
11             if v_p = v_q then
12                 h_P = h_P ∧ n_p = n_q ;
13             else
14                 h_P = h_P ∧ n_p ≠ n_q ;
15         p ← pick an element from Q ;
16         Q ← Q − {p} ;
17     P ← P − Q ;
18 return h_P;
```

5.6 流图分析

流图分析与基于因果关系和不变式的形式验证是互相促进的，一方面，如果完成了形式验证，那么流图分析有助于分析带参系统的状态变迁过程，加深对带参系统的理解；另一方面，如果正在对一个带参系统开始验证工作，那么前期的流图分析有助于获得更多关于该带参系统的先验知识。因此流图分析的工作也是 ParaVerifier 的一个重要子系统。

例如，附录 B 所示的互斥协议的一个典型流如图 5.3 所示，其中每个节点代表一个状态，含义如表 5.1 所示，每条边代表状态间发生迁移所触发的规则，以及在规则中选择的分支。这个流图显示了在互斥协议中，一个节点的状态从无效（i）到独占（e）的迁移过程。

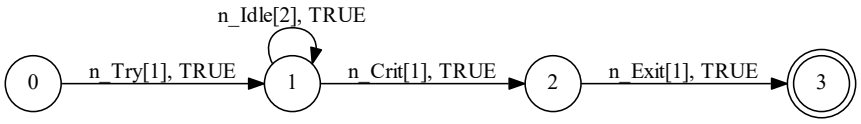

图 5.3: 互斥协议的一个典型流

编号	状态
0	$(n[1] = i)$
1	$((n[1] = t) \wedge (x = TRUE))$
2	$(n[1] = c)$
3	$(n[1] = e)$

表 5.1: 互斥协议典型流中的状态

使用 ParaVerifer 对一个带参系统进行流图分析，需要指定流图的起始状态和终止状态，另外还需要给定观察变量。流图分析的框架如算法 7 所示。

算法 7 只是生成流图的大体框架，是一个广度优先搜索的过程，用到了一些比较复杂的策略，在这里作几点补充说明：

- 生成新状态集时需要考虑规则的分支，因此会生成多个状态，且生成新状态的过程需要使用前置条件算子 $preCond$ ；
- 目前采用的等价关系 $\sim$ 要求两个状态中的观察变量必须完全一致；
- 搜索过程会找到大量可到达 f_{end} 的状态，但可能只有一部分状态是从 f_{start} 出发的，因此需要剔除。

虽然现在的流图分析还处于前期探索阶段，需要反复尝试合适的观察变量，但是对中型或者大型协议，ParaVerifier 也能够进行相当程度的分析工作，如附录 C 和附录 D 所示。

算法 7: 流图分析算法 FindFlow

Input: 起始状态 f_{start} ，终止状态 f_{end} ，观察变量集 O ，规则实例集 R^{inst}

Output: 顶点集 V 和边集 E

```
1  V ← {f_start, f_end} ;
2  E ← ∅ ;
3  q ← {f_end} as a queue;
4  while q ≠ ∅ do
5      f ← q.Dequeue() ;
6      foreach r^inst ∈ R^inst do
7          F̂ ← generate new states between f and r^inst ;
           // refine F̂
8          F̂ ← F̂ − {g|g ∼ h, g ∈ F̂, h ∈ V} , where ∼ is an equivalence relation
           about O ;
9          foreach f̂ ∈ F̂ do
10             E ← E ∪ { an edge from f̂ to f} ;
11             if ¬(f̂ → f_start) then
12                 q.Enqueue(f̂) ;
13 remove all vertices and edges that cannot be arrived from f_start ;
14 return < V, E >;
```

5.7 本章小结

本章详细介绍了 ParaVerifier 的设计和实现：首先介绍了 ParaVerifier 的设计思路和对一个带参系统进行验证的过程框架；然后详细介绍了 ParaVerifer 的实现，包括：预处理过程、生成因果关系的启发式策略和相关算法、不变式查找的启发式策略和相关算法、使用缓存技术提高性能和 Isabelle 定理证明脚本的生成；最后拓展了 ParaVerifer 的能力，使之能够进行流图分析，帮助加深对带参系统状态迁移过程的理解。

第六章　案例研究

本章将使用 ParaVerifer 对不同规模的几种带参系统的基准测试用例进行形式化验证。实验数据均在 4 核 Inter® Xeon® 2.40GHz 处理器，8G 内存，64 位 Linux 3.15.10 环境下测试获得。另外假设

1. 编译器 *murphi2ocaml* 所在目录为 `${moc}` ；
2. NuSMV 服务器为 `vserv` ，Murphi 服务器为 `mserv` ；
3. Isabelle 2015 在环境变量中已经进行设置;

6.1　MESI 协议的验证

MESI 协议是一个被广泛使用的小型缓存一致性协议，支持写回操作，属于总线监听协议，因其缓存块有修改（M)、独占（E)、共享（S）和无效（I）这几种状态而得名。

若 MESI 协议的 Murphi 模型为 `mesi.m` ，使用 ParaVerifier 对 MESI 协议进行验证，只需要执行以下命令。

```
python ${moc}/gen.py -m mesi.m > mesi.ml
corebuild mesi.byte -pkg re2 -I src
./mesi.byte -vh vserv
cd n_mesi
./run.sh
```

MESI 协议的验证结果如表 6.1 所示。

系统名称	规则数目	不变式数目	时间/s	内存/MB
MESI	4	3	3.33	148

表 6.1: MESI 协议的验证结果

6.2　Germam 协议的验证

German 协议是一个基于目录结构的中型缓存一致性协议，被用在共享内存的并发多处理器系统中以保证缓存一致性。German 协议中有主节点 Home 和用户节点 Client，其中 Client 节点是参数化的。

若 German 协议的 Murphi 模型为 `german.m` ，使用 ParaVerifier 对 German 协议进行验证，只需要执行以下命令。

```
python ${moc}/gen.py -m german.m > german.ml
corebuild german.byte -pkg re2 -I src
./german.byte -vh vserv
cd n_german
./run.sh
```

German 协议的验证结果如表 6.2 所示。与小型协议 MESI 的结果相比，所用时间明显增加，内存消耗则轻微增加。

系统名称	规则数目	不变式数目	时间/s	内存/MB
German	13	52	48.20	158

表 6.2: German 协议的验证结果

6.3 Flash 协议的验证

Flash 协议是一个基于目录结构大型缓存一致性协议，为上千规模的处理单元进行了针对性的复杂设计。在 Flash 协议中，每个处理器所在节点都有一个本地缓存，也包含全局内存的一部分。处理器所请求和缓存的内存块可能是本地（Home）的，也可能是远程（Remote）的。Flash 协议非常复杂，因此完成对 Flash 协议的验证，需要更多的额外工作。

Flash 协议是一个非常复杂的大型协议，经过测试，发现当对称节点即 Remote 节点的个数在 3 个以内，并且不考虑数据性质的情况下，从其 Murphi 模型转换得到的 NuSMV 模型可以计算得到可达集，但是更大规模的可达集就不可算得了；另外，Flash 协议的原始 Murphi 版本 `flash.m` 中，节点类型 `NODE` 包含不对称的 Home 节点，无法应用启发式策略 5.1 ，因此无法直接在 ParaVerifer 中完成验证工作。针对这两个问题，提出以下方案:

1. 限制 Remote 节点个数为 3，剔除数据相关状态变量和规则，并使用 NuSMV 服务计算得到可达集；如果要检查的不变式的参数范围在 3 以内并且不包含数据状态变量，那么使用 NuSMV 服务，否则使用 Murphi 服务。

2. 修改 `flash.m` ，从带参状态变量、带参规则中剥离 Home 节点，并在得到的内部模型表示中进行相应修改，从而使节点类型 `NODE` 成为一个仅包含对称节点的类型，得到的新 Murphi 模型为 `flash_global.m` 。

现在对 Flash 协议进行验证。

```
python ${moc}/gen.py -m flash_global.m > flash_global.ml
```

修改 `flash_global.ml` 以匹配所使用的可达集，然后继续验证过程。

```
corebuild flash_global.byte -pkg re2 -I src
./flash_global.byte -vh vserv -mh mserv
cd n_flash_global
./run.sh
```

Flash 协议的验证结果如表 6.3 所示。与小型协议 MESI 和中型协议 German 的结果相比，所用时间增长显著，内存消耗也有所增长。值得一提的是，实现完整版本的 Flash 协议的几乎全自动化形式验证并给出严格的证明，这项工作尚属首次。

系统名称	规则数目	不变式数目	时间/s	内存/MB
Flash	62	162	589.23	178

表 6.3: Flash 协议的验证结果

6.4 本章小结

本章分别各通过一个实例介绍了使用 ParaVerifer 对小型、中型和大型带参系统的验证过程，并给出了各自的验证结果和实验数据。每个案例的 Murphi 模型和最后生成的证明脚本可以在网上查阅[44] 。不难发现，一般带参系统越复杂，所需要的辅助归纳不变式和完成验证所消耗的资源就越多。但总而言之，使用 ParaVerifer 对带参系统进行验证，是相对便捷的。

第七章　结论和下一步工作

本文提出了一种对带参系统进行形式化验证的新方法并实现了相应的验证工具 ParaVerifer。ParaVerifer 以一致性引理等为理论基础，应用启发式方法，在已有 Murphi 模型的基础上可以自动完成辅助不变式的查找和因果关系的生成，并给出 Isabelle 证明脚本以证明其安全性质的正确性。本文的创新点体现在以下方面：

- 应用了一系列的启发式策略，大大缩减了生成因果关系时的搜索空间，并且能够在短时间内完成对不变式的检查，将原本不可判定的带参问题转化为能够快速判定的问题。
- 提出了能够全面描述带参系统的形式化语义模型并且构建了相关的模型语言版本转换工具，在此基础上自动分析初始带参不变式，大大提高了验证过程的自动化水平。
- 通过应用一个客户端/服务器的架构，提高了 ParaVerifier 调用第三方工具的性能，同时可以将不同工具的调用工作拆分到不同的处理机上，实现了检查结果的复用。
- 对一系列基准测试用例尤其是 Flash 缓存一致性协议进行了形式化验证方面的研究，均能够全自动化完成验证目标，所需要的时间和资源相对较少，且能够查找到最为全面的不变式。
- 对带参系统的流图分析能够直观地表示系统关键状态的迁移过程，一方面有助于对不熟悉的带参系统展开最初研究，另一方面也可以加深对熟悉的带参系统的理解。

当然，ParaVerifier 验证工具也还存在一些缺陷，需要进一步的改进。一方面，当前的形式化验证方法只适用于带参系统，如何将其推广到一般系统的验证，以及如何使其支持更复杂的算术操作，还需要进一步探索。另一方面，生成因果关系时，各个不变式实例和各个规则实例之间的比对是相互独立的，可以设法使用并行或者分布式相关技术，进一步提高性能。另外，工具的实现还需要进一步的优化，包括实现更完善的编译器来将 Murphi 模型提升到语义模型，构建形式化验证的集成开发调试环境，以及改进流图分析使其降低对手动选取观察变量的依赖等。

参考文献

[1] C Norris Ip and David L Dill. Efficient verification of symmetric concurrent systems. In *Computer Design: VLSI in Computers and Processors, 1993. ICCD'93. Proceedings., 1993 IEEE International Conference on*, 1993, 230–234.

[2] Javier Esparza, Alain Finkel, and Richard Mayr. On the verification of broadcast protocols. In *Logic in Computer Science, 1999. Proceedings. 14th Symposium on*, 1999, 352–359.

[3] Giorgio Delzanno. Automatic verification of parameterized cache coherence protocols. In *Computer Aided Verification*, 2000, 53–68.

[4] Tamarah Arons, Amir Pnueli, Sitvanit Ruah, Ying Xu, and Lenore Zuck. Parameterized verification with automatically computed inductive assertions? In *Computer Aided Verification*, 2001, 221–234.

[5] Giorgio Delzanno. Constraint-based verification of parameterized cache coherence protocols. *Formal Methods in System Design*, 2003, 23(3):257–301.

[6] E Allen Emerson and Vineet Kahlon. Exact and efficient verification of parameterized cache coherence protocols. In *Correct Hardware Design and Verification Methods*, Springer, 2003, 247–262.

[7] Steven German and Geert Janssen. Tutorial on verification of distributed cache memory protocols. In *Formal methods in computer aided design*, 2004.

[8] Krzysztof R Apt and Dexter C Kozen. Limits for automatic verification of finite-state concurrent systems. *Information Processing Letters*, 1986, 22(6):307–309.

[9] Edmund M Clarke, Orna Grumberg, and Doron Peled. *Model checking*, MIT press, 1999.

[10] Robert P Kurshan. *Computer-aided verification of coordinating processes: the automata-theoretic approach*, Princeton university press, 2014.

[11] Gerard J Holzmann. *The SPIN model checker: Primer and reference manual*, Addison-Wesley Reading, 2004, volume 1003.

[12] Michael JC Gordon. *HOL: A proof generating system for higher-order logic*, Springer, 1988.

[13] Tobias Nipkow, Lawrence C Paulson, and Markus Wenzel. *Isabelle/HOL: a proof assistant for higher-order logic*, Springer Science & Business Media, 2002, volume 2283.

[14] Yves Bertot and Pierre Castéran. *Interactive theorem proving and program development: Coq' Art: the calculus of inductive constructions*, Springer Science & Business Media, 2013.

[15] Sam Owre, John M Rushby, and Natarajan Shankar. Pvs: A prototype verification system. In *Automated Deduction—CADE-11*, Springer, 1992, 748–752.

[16] Ahmed Bouajjani, Peter Habermehl, and Tomáš Vojnar. Verification of parametric concurrent systems with prioritised fifo resource management. *Formal Methods in System Design*, 2008, 32(2):129–172.

[17] Jesse Bingham and Alan J Hu. Empirically efficient verification for a class of infinite-state systems. In *Tools and Algorithms for the Construction and Analysis of Systems*, Springer, 2005, 77–92.

[18] Alexander Kaiser, Daniel Kroening, and Thomas Wahl. Dynamic cutoff detection in parameterized concurrent programs. In *Computer Aided Verification*, 2010, 645–659.

[19] Ching-Tsun Chou, Phanindra K Mannava, and Seungjoon Park. A simple method for parameterized verification of cache coherence protocols. In *Formal Methods in Computer-Aided Design*, 2004, 382–398.

[20] Xiaofang Chen, Yu Yang, Ganesh Gopalakrishnan, and Ching-Tsun Chou. Reducing verification complexity of a multicore coherence protocol using assume/guarantee. In *Formal Methods in Computer Aided Design, 2006. FMCAD'06*, 2006, 81–88.

[21] Xiaofang Chen, Yu Yang, Michael Delisi, Ganesh Gopalakrishnan, and Ching-Tsun Chou. Hierarchical cache coherence protocol verification one level at a time through assume guarantee. In *High Level Design Validation and Test Workshop, 2007. HLVDT 2007. IEEE International*, 2007, 107–114.

[22] Edmund Clarke, Muralidhar Talupur, and Helmut Veith. Environment abstraction for parameterized verification. In *Verification, Model Checking, and Abstract Interpretation*, 2006, 126–141.

[23] Edmund Clarke, Murali Talupur, and Helmut Veith. Proving ptolemy right: The environment abstraction framework for model checking concurrent systems. In *Tools and Algorithms for the Construction and Analysis of Systems*, Springer, 2008, 33–47.

[24] Yi Lv, Huimin Lin, and Hong Pan. Computing invariants for parameter abstraction. In *Proceedings of the 5th IEEE/ACM International Conference on Formal Methods and Models for Codesign*, 2007, 29–38.

[25] Seungjoon Park and David L Dill. Verification of flash cache coherence protocol by aggregation of distributed transactions. In *Proceedings of the eighth annual ACM symposium on Parallel algorithms and architectures*, 1996, 288–296.

[26] MURALI TALUPUR SAVA KRSTI and JOHN O' LEARY MARK R TUTTLE. Parametric verification of industrial cache protocols. *Designing Correct Circuits*, 2008, 117.

[27] Murali Talupur and Mark R Tuttle. Going with the flow: Parameterized verification using message flows. In *Proceedings of the 2008 International Conference on Formal Methods in Computer-Aided Design*, 2008, 10.

[28] Cormac Flanagan and Shaz Qadeer. Predicate abstraction for software verification. In *ACM SIGPLAN Notices*, 2002, 191–202.

[29] Kai Baukus, Yassine Lakhnech, and Karsten Stahl. Parameterized verification of a cache coherence protocol: Safety and liveness. In *Verification, Model Checking, and Abstract Interpretation*, 2002, 317–330.

[30] Satyaki Das, David L Dill, and Seungjoon Park. Experience with predicate abstraction. In *Computer Aided Verification*, 1999, 160–171.

[31] Kenneth L McMillan. Parameterized verification of the flash cache coherence protocol by compositional model checking. In *Correct hardware design and verification methods*, Springer, 2001, 179–195.

[32] Amir Pnueli, Sitvanit Ruah, and Lenore Zuck. Automatic deductive verification with invisible invariants. In *Tools and Algorithms for the Construction and Analysis of Systems*, Springer, 2001, 82–97.

[33] Sudhindra Pandav, Konrad Slind, and Ganesh Gopalakrishnan. Counterexample guided invariant discovery for parameterized cache coherence verification. In *Correct Hardware Design and Verification Methods*, Springer, 2005, 317–331.

[34] Sylvain Conchon, Amit Goel, Sava Krstić, Alain Mebsout, and Fatiha Zaïdi. Cubicle: A parallel smt-based model checker for parameterized systems. In *Computer Aided Verification*, 2012, 718–724.

[35] Sylvain Conchon, Amit Goel, Sava Krstić, Alain Mebsout, and Fatiha Zaïdi. Invariants for finite instances and beyond. In *Formal Methods in Computer-Aided Design (FMCAD)*, 2013, 61–68.

[36] 曹燊和李勇坚. 基于不变量查找的 german 协议验证. *计算机系统应用*, 2015, 24 (11):173–178.

[37] Yongjian Li, Jun Pang, Yi Lv, Dongrui Fan, Shen Cao, and Kaiqiang Duan. Paraverifier: An automatic framework for proving parameterized cache coherence protocols. In *Automated Technology for Verification and Analysis*, Springer, 2015, 207–213.

[38] Alessandro Cimatti, Edmund Clarke, Enrico Giunchiglia, Fausto Giunchiglia, Marco Pistore, Marco Roveri, Roberto Sebastiani, and Armando Tacchella. Nusmv 2: An opensource tool for symbolic model checking. In *Computer Aided Verification*, 2002, 359–364.

[39] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, Springer, 2008, 337–340.

[40] David L Dill. The mur ϕ verification system. In *Computer Aided Verification*, 1996, 390–393.

[41] Xavier Leroy, Damien Doligez, Alain Frisch, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. The ocaml system release 4.02. *Institut National de Recherche en Informatique et en Automatique*, 2014.

[42] DUAN Kai-qiang and LI Yong-jian. Results of generated nusmv code of some protocols. `https://github.com/lyj238/murphismv`.

[43] W Richard Stevens and Stephen A Rago. *Advanced programming in the UNIX environment*, 2013. 589–628.

[44] I Yong-jian and DUAN Kai-qiang. paraverifier. `https://github.com/paraVerifier/paraVerifier`.

附录 A　复杂变量的定义和引用

A.1　复杂变量的定义

简单变量和复杂变量的定义例子如表 A.1 所示。其中上边定义了 boolean 类型的变量 ExGntd ，下面定义了两个变量，分别是CACHE_STATE 类型的复合带参变量 Cache[i].State 和DATA 类型的复合带参变量 Cache[i].Data ，且其参数 i 的类型为 NODE 。

简单变量	Arrdef([("ExGntd" , [])], "boolean")
复杂变量	Arrdef([("Cache", [Paramdef("i", "NODE")]); ("State", [])], "CACHE_STATE") Arrdef([("Cache", [Paramdef("i", "NODE")]); ("Data", [])], "DATA")

表 A.1: 简单变量和复杂变量的定义

显然这种写法不是特别方便，写起来过于繁琐。借助 OCaml 的函数可以实现大大简化这种写法并使得其有更直观的含义。

```
(** Array variable definition *)
let arrdef ls typename = Arrdef(ls, typename)

(** Record definition *)
let record_def name paramdefs vardefs =
  List.map vardefs ~f:(fun vardef ->
    let Arrdef(ls, t) = vardef in
    arrdef ((name, paramdefs)::ls) t
  )
```

应用这些函数，实现和表 A.1 相同的功能，如表 A.2 所示。

简单变量	arrdef [("ExGntd" , [])] "boolean"
复杂变量	let _CACHE = List.concat [[arrdef [("Cmd", [])] "MSG_CMD"]; [arrdef [("Data", [])] "DATA"]] record_def "Cache" [paramdef "i" "NODE"] _CACHE

表 A.2: 简单变量和复杂变量的简化定义

A.2 复杂变量的引用

简单变量和复杂变量的引用例子如表 A.3 所示。其中上边引用了变量 ExGntd，下面引用了两个变量，分别是复合带参变量 Cache[i].State 和复合带参变量 Cache[i].Data 。

简单变量	Arr([("ExGntd" , [])])
复杂变量	Arr([("Cache", [Paramdef("i", "NODE")]); ("State", [])]) Arr([("Cache", [Paramdef("i", "NODE")]); ("Data", [])])

表 A.3: 简单变量和复杂变量的引用

显然这种写法不是特别方便，写起来过于繁琐。借助 OCaml 的函数可以实现大大简化这种写法并使得其有更直观的含义。

```
(** Array variable *)
let arr ls = Arr(ls)

(** Global variable *)
let global name = arr [(name, [])]

(** Record *)
let record vars =
  arr (List.concat (List.map vars ~f:(fun (Arr(ls)) -> ls)))
```

应用这些函数，实现和表 A.3 相同的功能，如表 A.4 所示。

简单变量	global "ExGntd"
复杂变量	record [arr [("Cache", [paramref "i"])]; global "State"] record [arr [("Cache", [paramref "i"])]; global "Data"]

表 A.4: 简单变量和复杂变量的简化引用

附录 B　互斥协议的 Murphi 模型

```
const
  clientNUMS: 5;

type
  state: enum{I, T, C, E};
  client: 1..clientNUMS;

var
  n: array [client] of state;
  x: boolean;

ruleset i: client do
rule "Try"
  n[i] = I ==>
begin
  n[i] := T;
endrule;

rule "Crit"
  n[i] = T & x = true ==>
begin
  n[i] := C;
  x := false;
endrule;

rule "Exit"
  n[i] = C ==>
begin
  n[i] := E;
endrule;


rule "Idle"
  n[i] = E ==>
begin
  n[i] := I;
  x := true;
endrule;
endruleset;

startstate
begin
  for i: client do
    n[i] := I;
```

```
  endfor;
  x := true;
endstartstate;

ruleset i: client; j: client do
invariant "coherence"
  i != j -> (n[i] = C -> n[j] != C);
endruleset;
```

附录 C　German 协议的流图

起始状态 Cache[1].State = i

终止状态 Cache[1].State = e

观察变量集 { Cache[1].State, Cache[1].Cmd }

状态编号含义如下

0 : (Cache[1].State = i)

1 : ((CurCmd = empty) & (!(Chan1[1].Cmd = empty)) & (Chan2[1].Cmd = empty) & (Chan1[1].Cmd = reqe) & (ExGntd = FALSE) & (ShrSet[1] = FALSE) & (ShrSet[2] = FALSE))

2 : ((CurCmd = empty) & (!(Chan1[1].Cmd = empty)) & (Chan2[2].Cmd = empty) & (Chan1[1].Cmd = reqe) & (ShrSet[2] = TRUE) & (Chan3[2].Cmd = empty) & (ExGntd = TRUE) & (Chan2[1].Cmd = empty) & (ShrSet[1] = FALSE))

3 : ((CurCmd = empty) & (!(Chan1[1].Cmd = empty)) & (Chan2[2].Cmd = empty) & (Chan1[1].Cmd = reqe) & (ShrSet[2] = TRUE) & (Chan3[2].Cmd = empty) & (!(ExGntd = TRUE)) & (Chan2[1].Cmd = empty) & (ExGntd = FALSE) & (ShrSet[1] = FALSE))

4 : ((CurCmd = empty) & (!(Chan1[1].Cmd = empty)) & (Chan2[2].Cmd = gnte) & (Chan1[1].Cmd = reqe) & (ShrSet[2] = TRUE) & (Chan3[2].Cmd = empty) & (ExGntd = TRUE) & (Chan2[1].Cmd = empty) & (ShrSet[1] = FALSE))

5 : ((CurCmd = empty) & (!(Chan1[1].Cmd = empty)) & (Chan2[2].Cmd = gnts) & (Chan1[1].Cmd = reqe) & (ShrSet[2] = TRUE) & (Chan3[2].Cmd = empty) & (!(ExGntd = TRUE)) & (Chan2[1].Cmd = empty) & (ExGntd = FALSE) & (ShrSet[1] = FALSE))

6 : ((Chan2[1].Cmd = empty) & (CurCmd = reqe) & (CurPtr = 1) & (ExGntd = FALSE) & (ShrSet[1] = FALSE) & (ShrSet[2] = FALSE))

7 : ((Chan2[2].Cmd = empty) & (CurCmd = reqe) & (InvSet[2] = TRUE) & (Chan3[2].Cmd = empty) & (!(CurCmd = empty)) & (ExGntd = TRUE) & (Chan2[1].Cmd = empty) & (CurPtr = 1) & (ShrSet[1] = FALSE))

8 : ((Chan2[2].Cmd = empty) & (CurCmd = reqe) & (InvSet[2] = TRUE) & (Chan3[2].Cmd = empty) & (!(CurCmd = empty)) & (!(ExGntd = TRUE)) & (Chan2[1].Cmd = empty) & (CurPtr = 1) & (ExGntd = FALSE) & (ShrSet[1] = FALSE))

9 : ((Chan2[2].Cmd = gnte) & (CurCmd = reqe) & (InvSet[2] = TRUE) & (Chan3[2].Cmd = empty) & (!(CurCmd = empty)) & (ExGntd = TRUE) & (Chan2[1].Cmd = empty) & (CurPtr = 1) & (ShrSet[1] = FALSE))

10 : ((Chan2[2].Cmd = gnts) & (CurCmd = reqe) & (InvSet[2] = TRUE) & (Chan3[2].Cmd = empty) & (!(CurCmd = empty)) & (!(ExGntd = TRUE)) & (Chan2[1].Cmd = empty) & (CurPtr = 1) & (ExGntd = FALSE) & (ShrSet[1] = FALSE))

11 : (Chan2[1].Cmd = gnte)

12 : ((Chan2[2].Cmd = inv) & (Chan3[2].Cmd = empty) & (!(CurCmd = empty)) & (ExGntd = TRUE) & (Chan2[1].Cmd = empty) & (CurCmd = reqe) & (CurPtr = 1) & (ShrSet[1] = FALSE))

13 : ((Chan2[2].Cmd = inv) & (Chan3[2].Cmd = empty) & (!(CurCmd = empty)) & (!(ExGntd = TRUE)) & (Chan2[1].Cmd = empty) & (CurCmd = reqe) & (CurPtr = 1) & (ExGntd = FALSE) & (ShrSet[1] = FALSE))

14 : (Cache[1].State = e)

15 : ((Chan3[2].Cmd = invack) & (!(CurCmd = empty)) & (ExGntd = TRUE) & (Chan2[1].Cmd = empty) & (CurCmd = reqe) & (CurPtr = 1) & (ShrSet[1] = FALSE))

16 : ((Chan3[2].Cmd = invack) & (!(CurCmd = empty)) & (!(ExGntd = TRUE)) & (Chan2[1].Cmd = empty) & (CurCmd = reqe) & (CurPtr = 1) & (ExGntd = FALSE) & (ShrSet[1] = FALSE))

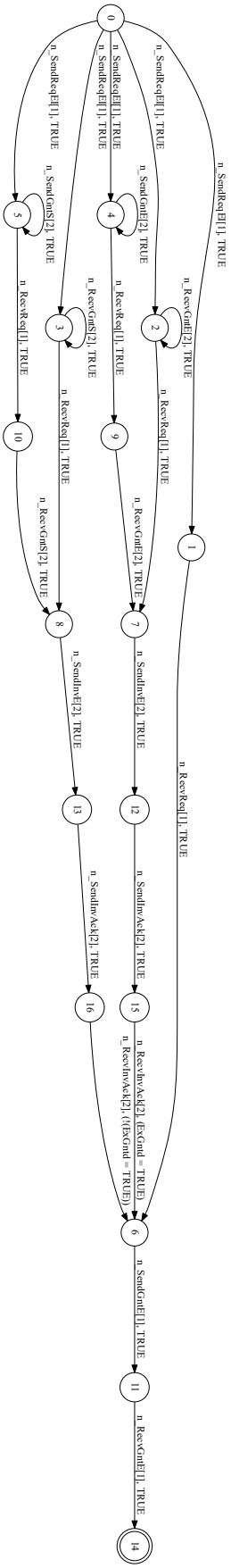

图 C.1: German 协议的一个典型流

附录 D　Flash 协议的流图

起始状态 Sta.Proc[1].CacheState = cache_i

终止状态 Sta.Proc[1].CacheState = cache_e

观察变量集 { Sta.Proc[1].CacheState, Sta.UniMsg[1].Cmd }

状态编号含义如下

0 : (Sta.Proc[1].CacheState = cache_i)

1 : ((Sta.Dir.Dirty = FALSE) & (Sta.Dir.HeadVld = FALSE) & (Sta.Dir.Local = TRUE) & (Sta.Dir.Pending = FALSE) & (Sta.UniMsg[1].Cmd = uni_getx) & (Sta.UniMsg[1].HomeProc = TRUE) & (!(Sta.HomeProc.ProcCmd = node_get)) & (Sta.Proc[1].ProcCmd = node_getx))

2 : ((Sta.Dir.Dirty = FALSE) & (Sta.Dir.HeadVld = FALSE) & (Sta.Dir.Local = FALSE) & (Sta.Dir.Pending = FALSE) & (Sta.UniMsg[1].Cmd = uni_getx) & (Sta.UniMsg[1].HomeProc = TRUE) & (Sta.Proc[1].ProcCmd = node_getx))

3 : ((Sta.Dir.Dirty = FALSE) & (Sta.Dir.HeadPtr = 1) & (Sta.Dir.HomeHeadPtr = FALSE) & (Sta.Dir.HomeShrSet = FALSE) & (Sta.Dir.Local = TRUE) & (Sta.Dir.Pending = FALSE) & (Sta.UniMsg[1].Cmd = uni_getx) & (Sta.UniMsg[1].HomeProc = TRUE) & (!(Sta.HomeProc.ProcCmd = node_get)) & (Sta.Dir.ShrSet[2] = FALSE) & (Sta.Proc[1].ProcCmd = node_getx))

4 : ((Sta.Dir.Dirty = FALSE) & (Sta.Dir.HeadPtr = 1) & (Sta.Dir.HomeHeadPtr = FALSE) & (Sta.Dir.HomeShrSet = FALSE) & (Sta.Dir.Local = FALSE) & (Sta.Dir.Pending = FALSE) & (Sta.UniMsg[1].Cmd = uni_getx) & (Sta.UniMsg[1].HomeProc = TRUE) & (Sta.Dir.ShrSet[2] = FALSE) & (Sta.Proc[1].ProcCmd = node_getx))

5 : ((Sta.Dir.Dirty = FALSE) & (Sta.Dir.HeadVld = TRUE) & (Sta.Dir.Local = TRUE) & (Sta.Dir.Pending = FALSE) & (Sta.UniMsg[1].Cmd = uni_getx) & (Sta.UniMsg[1].HomeProc = TRUE) & (!(Sta.Dir.HeadPtr = 1)) & (!(Sta.HomeProc.ProcCmd = node_get)) & (Sta.Proc[1].ProcCmd = node_getx))

6 : ((Sta.Dir.Dirty = FALSE) & (Sta.Dir.HeadPtr = 1) & (Sta.Dir.HeadVld = TRUE) & (Sta.Dir.HomeHeadPtr = FALSE) & (Sta.Dir.Local = TRUE) & (Sta.Dir.Pending = FALSE) & (Sta.Dir.ShrSet[1] = TRUE) & (Sta.UniMsg[1].Cmd = uni_getx) & (Sta.UniMsg[1].HomeProc = TRUE) & (!(Sta.HomeProc.ProcCmd = node_get)) & (Sta.Proc[1].ProcCmd = node_getx))

7 : ((Sta.Dir.Dirty = FALSE) & (Sta.Dir.HeadPtr = 1) & (Sta.Dir.HeadVld = TRUE) & (Sta.Dir.HomeHeadPtr = FALSE) & (Sta.Dir.Local = TRUE) & (Sta.Dir.Pending = FALSE) & (Sta.Dir.ShrSet[2] = TRUE) & (Sta.UniMsg[1].Cmd = uni_getx) & (Sta.UniMsg[1].HomeProc = TRUE) & (!(Sta.HomeProc.ProcCmd = node_get)) & (Sta.Proc[1].ProcCmd = node_getx))

8 : ((Sta.Dir.Dirty = FALSE) & (Sta.Dir.HeadVld = TRUE) & (Sta.Dir.Local = FALSE) & (Sta.Dir.Pending = FALSE) & (Sta.UniMsg[1].Cmd = uni_getx) & (Sta.UniMsg[1].HomeProc = TRUE) & (!(Sta.Dir.HeadPtr = 1)) & (Sta.Proc[1].ProcCmd = node_getx))

9 : ((Sta.Dir.Dirty = FALSE) & (Sta.Dir.HeadPtr = 1) & (Sta.Dir.HeadVld = TRUE) & (Sta.Dir.HomeHeadPtr = FALSE) & (Sta.Dir.Local = FALSE) & (Sta.Dir.Pending = FALSE) & (Sta.Dir.ShrSet[1] = TRUE) & (Sta.UniMsg[1].Cmd = uni_getx) & (Sta.UniMsg[1].HomeProc = TRUE) & (Sta.Proc[1].ProcCmd = node_getx))

10 : ((Sta.Dir.Dirty = FALSE) & (Sta.Dir.HeadPtr = 1) & (Sta.Dir.HeadVld = TRUE) & (Sta.Dir.HomeHeadPtr = FALSE) & (Sta.Dir.Local = FALSE) & (Sta.Dir.Pending = FALSE) & (Sta.Dir.ShrSet[2] = TRUE) & (Sta.UniMsg[1].Cmd = uni_getx) & (Sta.UniMsg[1].HomeProc = TRUE) & (Sta.Proc[1].ProcCmd = node_getx))

11 : ((Sta.Dir.Dirty = TRUE) & (Sta.Dir.Local = TRUE) & (Sta.Dir.Pending = FALSE) & (Sta.HomeProc.CacheState = cache_e) & (Sta.UniMsg[1].Cmd = uni_getx) & (Sta.UniMsg[1].HomeProc = TRUE) & (Sta.Proc[1].ProcCmd = node_getx))

12 : ((Sta.Proc[1].ProcCmd = node_getx) & (Sta.UniMsg[1].Cmd = uni_putx))

13 : (Sta.Proc[1].CacheState = cache_e)

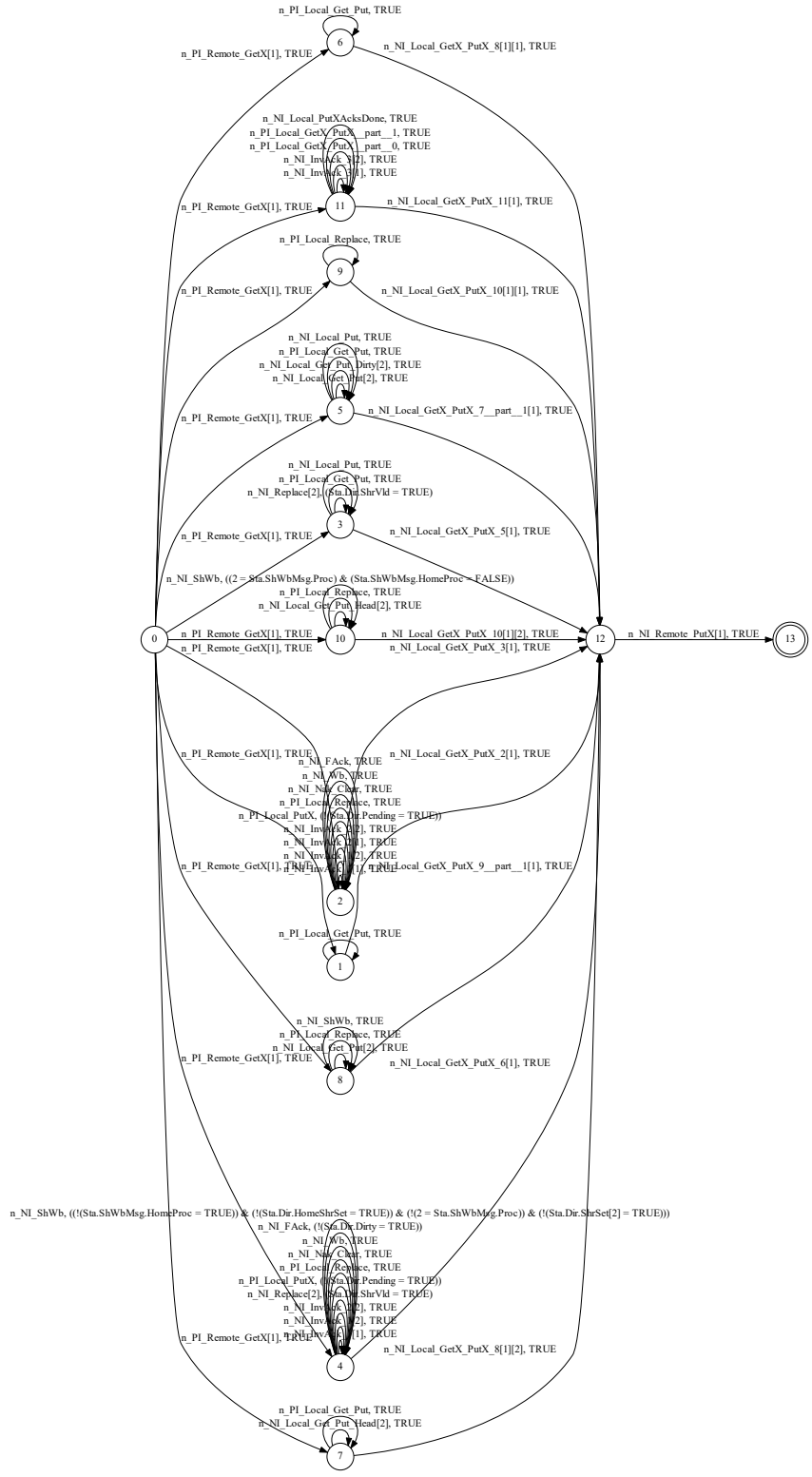

图 D.1: Flash 协议的一个典型流

作者简历及攻读学位期间发表的学术论文与科研成果

作者基本情况

段凯强，男，山西省洪洞县人，1991 年出生，中国科学院软件研究所硕士研究生。

联系方式

通讯地址：北京市海淀区中关村南四街 4 号中国科学院软件研究所

邮编：100190

E-mail: duankq@ios.ac.cn

发表的学术论文及科研成果

段凯强, 李勇坚. 基于带参系统 Murphi 模型的 SMV 自动建模. 计算机系统应用, 已录用.

致　谢

研究生的三年即将结束，这段时间我也收获了很多。首先感谢我的导师李勇坚老师，在他的悉心指导下，我具备了一定的科研能力并完成了毕业论文，也感谢实验室的吕毅老师、吴鹏老师等，与他们的讨论让我获益匪浅。另外还要感谢秘书费腾老师和张丽老师，以及系统管理员孙守云老师，还有研究生部的张欢老师和汤诗豪老师，他们的工作为我带来了很多方便。

此外还要感谢我的小伙伴们。同宿舍的小伙伴们给我带来了很多欢乐，他们是耿朋、李世强、梁超超、廖文琪、刘俊、吕荫润和宋小远；跟实验室的小伙伴的交流很开心，他们是熊浩军、陈明帅、胡蓝艺、景丽莎、周青、麻婧、李勇等。也感谢小伙伴朱自明和冯庆，跟他们养成的运动习惯会让我一直受益。还有小伙伴贺凯强、邓超和翟炀，在最艰难的时候给我支持，谢谢你们。

最后感谢我的家人，陪伴我渡过所有的难关，包容我的任性，给我快乐和鼓励，对我毫无所求，是我坚强和隐忍的理由。

还有很多帮助过我的人，不能一一具名，感谢你们。