摘 要

随着计算机系统的广泛应用，计算机软件的高可信性质受到了越来越多的关注。面向软件源程序的形式验证是保障软件高可信性质的一种有效方法，受到了大量的关注和广泛的研究。但是，由于软件程序逻辑复杂、规模庞大，如何提高软件源程序验证的精度特别是可扩展性仍是当前研究所面临的主要问题。

本文提出了一种面向 C 程序的时序安全性质验证方法，即切片执行方法。它本质上是一种轻量级的符号执行，能够自动抽象出顺序和并发 C 程序的有限状态模型，并基于模型检验方法进行验证。切片执行融合了程序切片、程序语义抽象、符号化状态表示、反例制导的抽象精化、偏序缩减等多种有效的状态空间缩减技术，大大缩减了抽象模型的状态空间，从而提高了软件源程序形式验证的可扩展性。切片执行的主要特点是面向时序安全性质的验证，能够以接近标准流敏感数据流分析的代价，达到路径敏感程序模拟的验证精度；同时切片执行支持包括循环在内的程序结构，支持全自动的模型抽象和性质验证。

切片执行构建在基于变量抽象的程序保守近似语义的基础上，它符号化地执行变量抽象下的相关语句，计算用于描述程序保守近似语义的部分最强后置条件和部分最弱前置条件，抽象出 C 程序的有限状态模型，并基于模型检验方法验证抽象模型是否满足给定的时序安全性质。基于本文提出的搜索复用框架，切片执行不断根据产生的伪反例路径精化变量抽象的抽象准则，直到性质被验证或找到真实反例路径。面向并发 C 程序验证，切片执行集成了本文提出的有状态动态偏序缩减方法，大大缩减搜索的状态空间。切片执行还集成了一种轻量级判定过程，用于对 C 程序验证过程中的验证公式进行高效判定。具体地，本文的创新点包括如下五个方面：

1、提出了切片执行的基本概念和方法。基于对程序验证基本规律的分析，我们提出了变量抽象方法，只考虑程序中与待验证性质相关的变量和语句。基于变量抽象，我们定义了部分最强后置条件，进而定义了程序的保守近似语义。基于这两个概念，我们提出了切片执行的概念，它是一种轻量级的符号执行过程。

2、提出了面向时序安全性质验证的搜索复用框架并将其应用于切片执行。搜索复用框架也是一种反例路径制导的抽象精化（CEGAR）框架，基于伪反例路径进行模型精化。相比传统的 CEGAR 框架，搜索复用框架的最大特点是能够在不同精度的抽象模型之间进行充分的信息复用，从而避免了大量不必要的重复搜索，有效降低了验证开销。

3、提出了部分最弱前置条件的概念并将其应用于切片执行。部分最弱前置条

件是程序保守近似语义的另一种表示方法，同样基于变量抽象定义。在切片执行过程中，可以用部分最弱前置条件部分地取代部分最强后置条件，以生成更弱的一阶逻辑公式来描述抽象模型的同一个状态，从而在不影响模型精度的前提下有效缩减生成模型的状态空间。

4、提出了面向有状态模型检验的有状态动态偏序缩减方法，并将其集成到切片执行过程中。我们将切片执行方法扩展到了对并发 C 程序的验证。为了进一步进行状态空间缩减，我们提出了有状态动态偏序缩减方法，并将其自然地集成到切片执行框架中，用于指导切片执行，使其避免搜索多条具有相同偏序关系的并发进/线程交迭执行路径。我们进行了多个实验，包括两个实用并发程序片段和一个并发 SSL 程序。实验结果表明，切片执行和有状态动态偏序缩减这两种正交的状态空间缩减方法的集成，大大缩减了并发程序抽象模型的状态空间，特别是降低了验证的空间开销。

5、提出了面向切片执行验证公式的轻量级判定过程。我们定义了一类整数线性一阶逻辑判定公式，此类判定公式支持 C 程序中常用的整数线性运算。我们优化了判定过程，并扩充了对整数除法、取余和位运算的支持。实验表明，扩充后的判定过程能够对面向 C 程序的切片执行所产生的绝大多数验证公式进行高效判定。在基于 SSL 程序的切片执行实验中我们发现，采用该判定方法后，验证效率比使用定理证明工具 Simplify 提高了 10.5 倍。

我们还基于开放源代码的 MAGIC 项目，实现了基于切片执行的 C 程序验证工具原型，并基于 *openssl-0.9.6c* 等实用程序针对上述每个创新点都进行了实验。我们在实验时采用了与 BLAST 和 MAGIC 相同的硬件平台，并针对相同的验证用例，验证了相同的性质集合。经过充分的实验数据对比，我们发现切片执行在验证效率上具有一定的优势。

主题词：切片执行，模型检验，时序安全性质，C 程序验证，变量抽象，部分最强后置条件，部分最弱前置条件，有状态动态偏序缩减

ABSTRACT

With the wide using of computer systems, the trustworthy properties of the software are receiving more and more attentions. The formal verification of the software program is one of the effective methods to promise trustworthy. Therefore, more and more researchers are attracted to this area. However, the software programs usually have complex logic and large scale, so how to increase the accuracy and the scalability of the program verification is the main research topic.

The dissertation presents a novel method, namely slicing execution, to verify C programs with respect to temporal safety properties. It is basically a lightweight symbolic execution technical, and may automatically extract finite models from sequential and concurrent C programs. Slicing execution integrates several state space reduction techniques, including program slicing, abstraction of program semantics, symbolical representation, counterexample guided abstraction refinement and partial-order reduction, etc. Therefore, the state spaces of the abstracted models can be greatly reduced, and the scalability of the verification can be improved correspondingly. The distinguished feature of slicing execution is that it may achieve the accuracy of path-sensitive simulation for verifying temporal safety properties, only with the cost close to standard flow-sensitive dataflow analysis. More than that, it supports arbitrary program structure including all kinds of program loops. It can also full-automatically abstract models and verify properties without any aid of people.

Slicing execution is founded on the over-approximated program semantics of C programs by variable abstraction. It only symbolically executes the relevant statements under variable abstraction, and calculates the partial strongest post-conditions and the partial weakest preconditions to construct finite abstract models, which may be model checked with respect to temporal safety properties. Based on the verification reusing framework presented in the dissertation, slicing execution keeps refining the variable abstraction criteria according to the spurious counterexample paths until the property is proven or a feasible counterexample path is found. To verify concurrent C programs, a novel stateful dynamic partial-order reduction method is integrated into slicing execution, which greatly reduces the searching state space. A lightweight decision procedure is also integrated to efficiently decide the formulas generated during the verification of C programs. More thoroughly, the innovations of the dissertation are summarized into following five aspects:

1. The basic definitions and techniques for slicing execution. Based on the analysis of practical program verification, we present variable abstraction to only consider the variables and statements that are relevant to the given property. We then present the partial strongest post-condition, which represents the over-approximated program semantics. Finally, we present slicing execution as a lightweight symbolic execution.

2. The temporal safety property oriented searching reusing framework and its application in slicing execution. Like counterexample guided abstraction refinement (CEGAR), the searching reusing framework also refines the abstracted model under the guidance of spurious counterexamples. However, it makes the searching information be reused among the models of different precision, and thus a great deal of redundant searching may be avoided.

3. The definition of partial weakest precondition and its application in slicing execution. The partial weakest precondition is another way to represent the over-approximated program semantics, and its definition is also based on variable abstraction. In slicing execution, we may use partial weakest preconditions to replace part of partial strongest post-conditions to generate much weaker state formulas. As a result, we may greatly reduce the state space of the abstracted model without loss of accuracy.

4. The stateful dynamic partial-order reduction technique and its application in slicing execution. After extending slicing execution to concurrent C programs, we present the stateful dynamic partial-order reduction technique to guide it not to search multiple interleaving transition sequences that has the same partial-order. We select several experiments including two practical program fragments and a concurrent SSL system. The experiments illustrate that the integration of slicing execution and stateful dynamic partial-order reduction leads to the reduction of the state space, as well as the reduction of the space cost of the verification.

5. The lightweight decision procedure for the verification formulas of slicing execution. We define an extended class of integer linear first-order logic formula, which supports most integer linear and bit-wise expressions in C programs including integer division, integer modular and bit-wise operation. Experiment results show that the extended decision procedure supports most verification formulas met in slicing execution. Based on the experiment over the practical program *openssl-0.9.6c*, we find that the decision cost is 10.5 times less than the theorem prover Simplify.

Our slicing execution tool prototype is implemented on the open-source project

MAGIC. Each innovation is demonstrated by an experiment on the practical program *openssl-0.9.6c*. We also compare our experiment results with BLAST and MAGIC. All experiments are performed on machines of same hardware, using same practical program and verifying same properties. The comparing results show that slicing execution is even a bit more effective than the other two.

Key Words: slicing execution, model checking, temporal safety property, C program verification, variable abstraction, partial strongest post-condition, partial weakest precondition, stateful dynamic partial-order reduction

表 目 录

图 目 录

独 创 性 声 明

本人声明所呈交的学位论文是我本人在导师指导下进行的研究工作及取得的研究成果。尽我所知，除了文中特别加以标注和致谢的地方外，论文中不包含其他人已经发表和撰写过的研究成果，也不包含为获得国防科学技术大学或其它教育机构的学位或证书而使用过的材料。与我一同工作的同志对本研究所做的任何贡献均已在论文中作了明确的说明并表示谢意。

学位论文题目：面向C程序验证的切片执行方法

学位论文作者签名：易晓东　　日期：2006年9月26日

学位论文版权使用授权书

本人完全了解国防科学技术大学有关保留、使用学位论文的规定。本人授权国防科学技术大学可以保留并向国家有关部门或机构送交论文的复印件和电子文档，允许论文被查阅和借阅；可以将学位论文的全部或部分内容编入有关数据库进行检索，可以采用影印、缩印或扫描等复制手段保存、汇编学位论文。

（保密学位论文在解密后适用本授权书。）

学位论文题目：面向C程序验证的切片执行方法

学位论文作者签名：易晓东　　日期：2006年9月26日

作者指导教师签名：　　日期：2006年9月26日

第一章 绪论

1.1 课题研究背景

1.1.1 软件的可信性质

随着科技的高速发展和计算机技术的广泛普及，人类社会的信息化、智能化程度越来越高，作为其核心的计算机软件已经融入到人们生活的方方面面，从飞机汽车到家用电器，都离不开软件的自动控制。另一方面，由软件控制的各种各样的机器和设备大量替代了以前必须由人工完成的低级劳动，不仅极大地减轻了人们工作量、减小了对工人可能造成的伤害，还大大地提高了工作的效率和质量。因此，如果说计算机技术的应用开创了一种新的生活和工作方式的话，那么软件就是这种新方式的灵魂和主导。

随着软件承担了越来越重要的功能和作用，软件失效所带来的代价越来越不能容忍。例如，海湾战争中由于软件失效曾导致美国爱国者导弹未能及时拦截伊拉克的飞毛腿导弹攻击；软件失效还导致 1996 年 6 月欧洲 Ariane 五型火箭首发失败，造成高达 25 亿美元的直接经济损失。

因此，作为信息化核心的软件，必须满足包括可靠性、安全性（包括可靠安全性和保密安全性）、可用性、可生存性等性质在内的高可信性质。其中，可靠性是指在规定的环境下和规定的时间内软件无失效运行的能力，可靠安全性是指软件运行不引起危险和灾难的能力，保密安全性是指软件系统对数据和信息提供保密性、完整性、可用性、真实性保障的能力，等等。

然而，软件作为人类连续的高度复杂的智力产品，其科学原理和工程规律远未得到充分的认识，缺乏有效地生产满足高可信软件的软件技术。导致的结果是：一方面，人们的生产生活越来越依赖于各种各样的软件；另一方面，随着软件功能的日益复杂和强大，软件的高可信性质越来越不能保证。例如，仅从军事方面来看，美国国防部认为嵌入式软件开发成为美武器项目研制的瓶颈；而在国内，根据当前总装备部的调查结果可以看到，软件失效已成为武器装备系统不能正常运转的主要原因，甚至已经超过硬件出现的问题。因此，如何有效的保证软件的安全性、提高可靠性成为软件研究领域亟需解决的重大课题，并受到各国政府、军方和软件公司的高度重视。

软件的高可信性质受到了国内外政府、军队和企业的广泛关注。作为 1999 年著名的 PITAC 报告《信息技术研究：为未来投资》(Information Technology Research: Investing in Our Future）认为美国“依赖于脆弱的软件”，“缺乏构造可靠安全软

件的技术”，明确指出将软件基础研究放在绝对优先的位置上。2001 年 1 月，美国 NITRD 信息技术研究与发展联合工作组（Interagency Working Group on Information Research and Development）发布了《高可信软件与系统研究需求》（High Confidence Software and Systems Research Needs）报告，提出了 HCSS（高可信软件与系统）的国家研究计划。2003 年 9 月，NITRD 在 2004 财年蓝皮书《美国创新的先进基础》（Advanced Foundations for American Innovation）中，设计高可靠软件的有效方法、复杂软件中高可信的科学和工程方法都成为了最优先研究的内容。欧盟的第五框架计划和第六框架计划都把高可信软件作为软件技术发展的重点，Information Society Technologies 支持了 SAFEAIR II（安全攸关系统的先进设计工具）、OMEGA（实时嵌入式系统的正确开发）、ARTIST（先进实时系统）等项目。为了应对软件安全性问题，我国也采取了一些针对性措施，如制定了 GJB/Z 102-97《软件可靠性和安全性设计准则》等软件开发标准，并进行了一些实践。

1.1.2 基于软件源代码的可靠安全性质保证

软件可信性质的保证是贯穿整个软件生命周期的，在软件的设计、实现、测试、维护等阶段进行的可信性质保证是相互依赖、缺一不可的。从保证方法上来看，形式化方法对软件可信性质的保证有着不可替代的作用。然而，形式化方法在实用软件的可信性质保证中仍不多见，基本处于实验室的试验阶段，并且，其使用者多是专家型用户。因此，将形式化方法以工程化的途径自动化、可扩展地实施软件安全性保证是一个重要趋势。

从当前软件的可信需求来看，可靠安全性（Safety）需求占据了重要的地位，它反应了人们对软件运行不会引起灾难和危害的根本要求。从对软件可靠安全性质的保证方法上来看，面向可靠安全性质的软件验证是一种重要且行之有效的手段，从而得到了长期和广泛的研究和应用。它通过检验软件是否满足所关心的可靠安全性质，来最终保证软件的可靠安全性。下文中，除非特别指明，否则我们讨论的安全性都是指可靠安全性。

本论文主要研究面向软件源代码验证的软件安全性质保证，它是软件安全性保证的一个重要组成部分，其意义主要表现在三个方面：首先，软件源代码包含了软件的所有细节，从而源代码级的验证能够准确地回答所关心的安全性质是否得到保证；其次，软件所关心的大量安全性质本身就是针对软件源代码的，例如不能有空指针、数组越界等安全性质，从而只能在源代码级进行验证和保证，同时软件编码又是最容易引入错误的阶段之一，因此源代码级的安全性质的验证能够确实提高软件可信性；最后，从目前软件工程方法的应用现状来看，软件安全性保证的主要矛盾尚处在软件代码层面上，在软件开发中，开发者把大量的时间

和精力花费在寻找软件错误的调试和测试中，而结果依然是不完全的。同时，目前使用的软件开发平台中对安全性保证的技术支持十分薄弱，主要通过软件过程控制进行保障，人为因素占了主导地位，从而带来了很大的不确定性。

当前面向源代码的安全性质保证主要依靠软件开发人员来手工发现软件中的错误，开发人员通过对软件执行步骤、变量和数据结构状态的变化、断言的违反等进行跟踪和判断，以确定程序中是否存在错误。一般情况下，这只能发现一些简单、明显的错误。当软件稍微复杂一些，由于庞大的可能运行路径集合，即使调试中不出现错误，也不能说明软件中就不存在错误。对于高可信软件，其失效并不仅仅是一些简单的计算错误，可能存在的进程同步时序错误、死锁、接口绑定错误、访问越界、栈溢出、空指针等，都会导致软件运行失效，而这些是仅仅通过传统调试手段无法完全发现的。已有断言方法主要对一些布尔表达式进行判断，也不能有效解决这些问题。例如，GJB/Z 102-97《软件可靠性和安全性设计准则》中所列出的大量安全性要求都是传统调试和编译环境难以发现的，因而在实际软件开发中出现了可操作性差等问题。

体现源代码级验证这种安全性保障技术的发展趋势的一个值得关注的动向是由图灵奖获得者 Tony Hoare（现任职微软剑桥研究院）于 2002 年提出的程序验证器（Verifying Compiler，后改称 Program Verifier）设想[1]。他希望程序验证器通过自动化的数学和逻辑推理方法检测其所编译的程序的正确性，而该正确性是通过程序中的类型、断言、规范或其它一些标注进行声明的。该设想正是我们的研究目标和研究内容。事实上，微软在其软件开发中广泛使用各类断言以提高程序可调试的能力，并开发出程序分析工具 PREfix 供其内部各开发小组在调试中使用，以检查程序中每条路径可能出现的空指针、访问越界等高可信软件所关注的错误中的一些较为简单的部分（但其面临的一个问题是找到的很多错误路径实际是不可行的，需要人工排除这些不可行路径）。工业界 2002 年统计结果认为，若程序验证器当时能够投入实用，美国当年因为程序错误引起的损失可以减少 220 – 600 亿美元。目前，验证编译器被国际计算机软件领域认为是甚至可以和人类登月、人类基因组图谱等相提并论的巨大挑战之一。

1.2 相关研究工作

面向软件源程序的形式化分析与验证方法可以分为动态与静态两大类方法，动态方法是指运行待检验的程序，并根据程序的动态运行情况判断待验证的性质

[1] 最近，该设想又被改名为 Verified Software，并包含了软件需求分析、设计、实现等不同软件开发阶段的安全性保障需求，但面向程序的验证仍是该设想的主要内容之一。

是否被满足，例如测试方法；而静态方法则是指通过对程序源代码进行静态分析来分析和验证其是否满足某种性质。两类方法各有优缺点：动态方法往往不能保证对程序进行完全的分析和验证，即不能保证性质一定被程序满足，而静态方法所花费的时间与空间的代价往往比动态方法大得多。本文的研究是围绕静态方法展开的，研究的重点集中在如何尽量减小静态分析和验证方法的时间和空间开销，从而提高静态方法的可扩展性。

根据特性的不同，我们将静态方法分为以下五类进行介绍，即静态分析方法、定理证明方法、抽象解释方法、模型检验方法和谓词抽象方法，下面我们结合实例介绍这五类方法的研究进展。

1.2.1 静态分析方法

静态分析（Static Analysis）是指直接对程序进行分析，以检查程序源代码是否满足某种性质。静态分析方法的思想比较直接，它往往较少涉及程序语义，因此一般来说不能保证可靠性和完备性。

使用静态分析方法的一个成功的例子来自于 IBM T.J. Waston 研究中心的 Xiaolan Zhang 和 Antony Edwards 的工作，他们使用一种基于类型的静态检查工具 CQUAL[1]对 Linux 操作系统的 LSM（Linux Security Models）框架在内核中钩子函数的放置进行了验证并找到了一个 Bug[2, 3]。Linux 的 LSM 框架是一个强制访问控制框架，每当 Linux 内核试图访问一个内核数据结构（如 Vnode、IPC、Socket 等）时，LSM 都会调用一个被称为钩子函数的函数对访问进行检查。Zhang 和 Edwards 试图验证 Linux 内核中的每次内核数据访问都确实被 LSM 检查过。他们的做法是使用 CQUAL 的类型检查系统，一个数据结构是否被 LSM 的钩子函数检查过对应不同的类型，通过类型冲突的检查可以验证是否有没有经过 LSM 钩子函数检查的内核对象访问。

Stanford 大学的 Dawson Engler 教授和他的研究小组研制了一个通用的静态分析工具来检查源程序中的 BUG，他们实现了 Meta-level Compilation（元编译器，MC），用于检验程序是否满足某种性质[4, 5]。MC 的实现方式是对 GUN g++扩展，得到 xg++编译器对软件编译，当 xg++把源程序编译为内部表示形式后，再根据用户要求对其进行流敏感和流不敏感的分析。但是，由于不考虑数据流，MC 既不能保证完备性（即 MC 找不出错误的程序不能保证确实不存在错误），也不能保证可靠性（即不能保证 MC 找出的每个错误都确实是一个错误）。因此 MC 是一个分析工具，而不能作为验证工具。另外，MC 也只能描述和检验一些比较直观的性质，如程序的加锁解锁等，但根据研究者的实际使用经验来看，很多现实程序中我们关心的性质，例如死锁、空指针以及一些优化问题都可以表示为这些简单直

观的形式。例如，MC 成功地应用到了四个大型的软件中，并找到了大约 500 多个错误，这四个软件是 Linux 2.3.99 内核源代码、OpenBSD 内核源代码、Xok 内核源代码[6]和 FLASH 的 Cache 一致性协议[7]。

此类静态分析工具中比较著名的还包括 PREfix[8]及 LCLint[9]等，其中 PREfix[8]是一个符号化的路径搜索工具，它通过将路径搜索进行局部化限制来提高可扩展性，该工具在实际程序中找出了大量的程序错误。LCLint[9]则将类型检查与数据流分析结合，基于提供的函数注释进行代码的局部分析。

ESP[10]是静态分析工具中比较有特色的一种，它基于“性质模拟（Property Simulation）”的技术，能够以多项式复杂度对时序安全性质进行验证，并且如果 ESP 报告性质不被违背，则该性质能够确保不被违背。但是，ESP 面向的是编写良好的代码，程序员需要隐式地维护程序状态和程序必须满足的性质的一致性，否则 ESP 将误报一些本不存在的错误。另外，对于程序循环，ESP 必须限定循环的执行次数，才能保证终止性。因此，在某些情况下，如果不提供额外的信息如循环不变式，也不采用额外的手段如 Widening，则 ESP 实际上不能保证验证的完备性。

1.2.2 定理证明方法

采用定理证明方法进行分析与验证时，需要有经验的用户提供大量的公理、前提条件及其它的系统相关信息，再由定理证明工具使用逻辑推理的方法对待验证的命题进行证明。基于定理证明的验证方法适用于一般的模型（这些模型通常都具有无限的状态空间），并可以验证涉及到程序的数据和控制流的命题[11]。常用的定理证明工具有 PVS[12]、Isabelle[13]、Nuprl[14]及 Simplify[15]等。

VFiasco 是德国 Dresden 大学研制的微内核操作系统，该项目使用了基于定理证明的验证方法[16]对其内核进行安全性验证，它使用了 Isabelle[13]定理证明工具，目标是验证其微内核的安全性达到欧洲 CC 标准的最高安全级 EAL7 级。验证的基本思路是设计一个逻辑编译器（Logical Compiler），把 VFiasco 内核的实现语言 C++转换为 Isabelle 的输入语言高阶逻辑（HOL）。验证者建立了 CPU 模型和类型安全（Type-safe）的内存模型，由于 Vfiasco 的内核很小，从而可以完成对所需性质的证明。

应用定理证明方法进行验证的难点在于需要用户输入推理策略以及大量的定理，因此定理证明方法不是一个自动化的方法[17]。另外，定理证明很难适用于大规模的项目。

1.2.3 抽象解释方法

抽象解释（Abstract Interpretation）理论[18]是 P. Cousot 和 R. Cousot 于 1977 年提出的程序静态分析时构造和逼近（Approxiamation）程序不动点语义的理论。作为一种逼近方法，抽象解释本质是为了在计算效率和计算精度之间取得均衡，以损失计算精度求得计算可行性，再通过迭代计算增强计算精度。基于抽象解释理论的形式化分析结果可靠但不完备，如果基于抽象解释理论的形式化分析结果表明抽象系统正确，则原系统正确；若分析结果表明抽象系统不正确，则原系统可能正确，也可能不正确（需要更精细的分析）。

在程序语义基于 Galois 连接的抽象解释理论框架基础上进行的程序验证工作，主要是 Patrick Cousot 等人以程序静态分析工具 ASTREE 为主要成果的一系列研究[19-22]，他们以嵌入式安全攸关实时软件系统作为研究对象，以检测和验证软件系统中的运行时错误（Runtime Error）作为主要研究内容。运行时错误包括：数组越界、浮点数除法运算时 0 作为除数、类型为短整型 Short 的变量在算术运算时溢出[-32768,32768]的范围等等。ASTREE 能够处理 C 程序中的指针、数组、整数和浮点数运算、条件测试、循环、函数调用、分枝（包括 goto、break、continue、switch），但不支持类型 Union、内存动态分配、C 函数库调用、无界的递归函数调用等。ASTREE 分析程序时直接处理源代码，以程序的操作语义作为标准语义，以程序的部分踪迹语义作为集合语义，以程序可达性语义精化的抽象作为抽象语义。ASTREE 中的程序分析方法建立在区间抽象（Interval Abstract）分析技术和一个简单的存储模型基础上，为区间抽象设计了多个数值抽象域（Numerical Abstract Domain）上的抽象分析方法，并且抽象域上的抽象分析过程使用了参数化方法，用于调整计算精度和计算效率。ASTREE 对程序进行分析具有自动、可靠、保证停机以及高效等特点。空客（Airbus）A340 有线飞行系统（Fly-By-Wire System）主要飞行控制软件是有 132000 行代码的 C 程序，2003 年 11 月，ASTREE 完全自动地验证了该软件中不存在运行时错误，在 2.8GHz、300MB 内存的 PC 机上，验证时间仅为 1 小时 20 分钟。2004 年 1 月起，ASTREE 被进一步扩展，对 A340 电动飞行控制代码（Electric Flight Control Codes）进行分析，并被进一步应用于空客 A380 系列飞行控制软件的开发和测试工作中。

以色列 Technion Haifa 大学 Orna Grumberge 等人提出了基于下界逼近、Widening 操作算子和有界模型检验（Bounded Model Checking）、针对多进程系统（Multi-Process System）的程序验证方法[23]。与基于抽象-精化的分析框架不同，Orna Grumberge 的方法采用了下界逼近-Widening 的分析框架，使用一系列下界逼近模型，避免了虚假反例的出现，对多进程系统的分析从多个进程的一次交叠运

行开始，逐步增加交叠的次数继续进行验证，直到分析过程找到不满足验证性质的反例，或者下界逼近模型中的所有踪迹都满足验证性质并且此时所有的踪迹都不依赖于下界逼近算子，从而表明原系统满足待验证性质。Orna Grumberge 的方法将下界逼近、Widening 操作算子和约束模型检验有机地结合在一起，是检测多进程系统中“bug”的一种高效方法。

C Global Surveyor（CGS）[24]也是一种基于抽象解释理论的 C 程序运行时错误检查工具，它由 NASA 开发，并成功地应用于 Mars Path-Finder（13.5 万行代码）、Deep Space One（28 万行代码）、Mars Exploration Rover（65 万行代码）等 NASA 的火星探测项目。CGS 重点检查 C 程序的三个运行时性质：访问未初始化的变量；访问空指针；数组越界访问。经过大量的调整，其误报率被控制在 10%左右。同时 CGS 能够在多处理器平台协作检查，从而进一步提高可扩展性。

1.2.4 模型检验方法

模型检验方法是近年来最受青睐的方法，在给定了系统的有限状态模型和使用合适的逻辑公式（如时序逻辑公式）描述的性质之后，模型检验工具将自动地检验所描述的性质是不是有效[25]。模型检验的基本思想是遍历系统的状态空间，以检验系统的所有可能路径是否满足给定的性质。状态空间的遍历一般采用两类方法进行，一类方法是通过显式的状态计算来遍历状态空间，另一类方法是通过隐式的不动点计算来遍历状态空间。

比较典型的模型检验工具有 Spin[26]、SMV[27]等，其中 SMV[27]主要用于对硬件的验证，而 Spin[26]则成功地应用于了多个实际的软件项目中。除了上述两个模型检验工具外，随着模型检验方法的广泛应用，出现了大量采用不同的模型检验思想的模型检验工具，下面我们结合具体的应用分类介绍。

1.2.4.1 基于通用模型检验工具的直接建模验证

要对软件项目进行验证，最直接的方法就是对软件建模，并使用已有的模型检验工具如 Spin[26]、SMV[27]等进行验证，NASA 的深空一号（Deep Space 1）火星探测器中的任务调度与控制系统的验证[28]以及 Utah 大学的 Fluke 操作系统的 IPC 子系统的验证[29]都是采用直接建模的方法进行的。

NASA 的深空一号火星探测器的任务调度与控制系统向外提供了一套编程语言 ESL（Executive Support Language），所有运行于飞行器上的任务都使用该语言编写。由于任务之间是并行的，为了确保资源的共享与互斥访问以及简化任务的编程，任务调度与控制系统把飞行器上所有设备的状态都存储到一个数据库中，再提供接口供运行于其上的任务感知和操作飞行器上设备的状态。为了互斥访问

设备，任务调度与控制系统使用锁机制来确保两个需要同一设备处于不同状态的任务不能并行执行。同时任务调度与控制系统使用了一个守护进程和一张锁表来监测锁和数据库资源，锁表中记录了一个锁对应的资源的状态，如果锁表与数据库中的同一个资源的状态出现了不一致，则该守护进程将中止所有通过该锁获得相应资源的任务。NASA 使用 Spin 对任务调度与控制系统进行了建模，并验证了两个性质：(1)所有任务在退出之间释放了所有的锁，(2)如果锁表中某锁对应的资源状态与数据库中的资源状态不一致，则所有使用该锁获得相应资源的任务都将被中止。使用 Spin 建模并在验证时找到了 5 个错误，其中 4 个是致命的[28]。

美国 Utah 大学的 Fluke 操作系统是基于面向对象的思想实现的微内核操作系统，其进程间通信机制(IPC)由于涉及到多线程而十分复杂，Utah 大学的科研人员的做法是使用手工的方法直接把 IPC 子系统的源代码转换到等价的 Promela 源代码给 Spin 做模型检验，也取得了满意的验证结果。

1.2.4.2 模型的自动抽取与验证

由于一般的模型检验工具都要求使用它们专用的语言来对系统建模，所以要想使用模型检验工具，必须用人工的方法对待验证的系统进行抽象和建模，这一点将会导致两个问题，一是建模工作量大，二是建模过程容易出错[30]。于是人们开始研究自动模型抽取工具，从高级语言源代码根据所要验证的性质自动提取模型并转换为模型检验工具如 Spin、SMV 等的输入语言描述形式。这一类的工具包括 Bandera 和 Modex 等。

Bandera[30]是 Kansas 州立大学的一个工具，其后续版本改名为 Bogor[31]。它接收系统实现的 Java 源程序和待检验的性质，自动构建可以直接被 Spin、SMV 等多种模型检验工具直接检验的模型。Bandera 借鉴了编译器的结构，分为前端和后端，前端负责对待检验性质和 Java 源程序进行处理，经过 Bandera 的模型抽取机制得到模型的中间代码表示形式，再用后端将之转化为适用于各个模型检验工具的语言。

Bandera 及 Bogor 使用了三种方法，即程序切片、数据抽象和模块限制从 Java 源程序中基于被检验的性质构建模型。程序切片是指把程序 P 中与被检验的性质 ϕ 无关的语句去掉，使得精简的程序 P' 满足 ϕ 当且仅当 P 满足 ϕ。数据抽象是指减小一个数据或数据结构的状态数，例如如果只关心一个整数是否大于 10，则就把一个整数抽象成了一个布尔值，从而大大缩减了系统状态数。模块限制是指在不影响正确性的前提下限制程序的规模，如在大多数情况下可以合理地假设系统中最多只有两个线程同时运行，从而进一步压缩状态空间。

与 Bandera 类似的 Modex[32]是 Spin 的设计者 Holzmann 开发的，用于从 ANSI-C 编写的程序中抽取模型。

1.2.4.3 基于下推自动机的模型检验

上述模型检验工具都是基于有限自动机（Finite State Automaton，FSA）或标记迁移系统（Labeled Transition System，LTS）来描述程序的，而这两类描述方式都不能描述递归程序，而下推自动机（Pushdown Automaton，PDA）可以用来描述递归程序。下推自动机由一个状态集合、一个符号栈、一组输入符号集合和一个状态迁移函数组成。对下推自动机的模型检验可以使用 Javier 等人提出的算法[33]，该算法可以在多项式时间内对线性时序逻辑（Linear Temporal Logic，LTL）公式进行模型检验。

MOPS（Model Checking Programs for Security Properties）[34, 35]是 Berkeley 大学的 Hao Chen 开发的一个基于下推自动机对程序的安全性质进行检验的工具，它使用 FSA 来描述安全性质，使用 PDA 描述程序模型，再使用上述算法进行检验。MOPS 忽略几乎所有的数据流而只关心程序的控制流，所以它很适合于检验与控制流相关的安全性质。

由于 MOPS 只考虑控制流，因此它的扩展性很强，例如可以应用于大规模的软件系统如 Linux 内核等，文献[34]指出，MOPS 可以应用于百万数量级代码行数的大规模软件中。但是，由于忽略了数据流，MOPS 是一个非常不精确的模型检验工具，会产生大量错误的警告，即 MOPS 找到的很多错误其实并非错误。但是 MOPS 可以保证验证的可靠性，也就是说，MOPS 不会漏掉一个真正的错误。

1.2.4.4 基于代码运行的模型检验

一般的模型检验方法都是静态的方法，要对程序进行检验，必须先对待检验的软件系统建模，但不幸的是，由于软件的复杂性，建模工作往往既费时间，同时又很难保证正确性，而且由于模型检验的状态爆炸问题，使得建模时往往要对软件系统进行较大程度的抽象，这将可能引起模型的失真而导致检验错误。另外，静态的模型检验方法往往无法精确地处理数据流以及指针和变量别名等问题。针对这些问题，研究者们提出一种动态的模型检验方法，即直接运行待检验的程序对待检验的性质进行验证，下面的 CMC 就是这样一个例子。

CMC（C Model Checker）[36, 37]是 Stanford 大学研制的一个直接对 C 语言和 C++语言进行模型检验的工具，以直接运行软件的动态方法对软件进行模型检验。CMC 是一个显式的模型检验工具，即它根据当前状态生成所有可能的后续状态。CMC 定义的状态很大，包括了程序当前时刻的栈、堆、CPU 寄存器和共享内存的内容。为了遍历软件的所有状态，CMC 提供了状态的恢复与保存机制，并通过 Hash 的方法保存遍历过的状态以节约时间和空间。

CMC 成功地用于对 Linux 的 TCP/IP 实现协议的验证[37]。为了对其进行验证，CMC 将整个 Linux 作为一个进程运行于用户空间，并将该 Linux 进程的所有栈、

堆、CPU 寄存器和共享内存作为模型检验的一个状态。CMC 利用两个进程运行了两个 Linux，并通过这两个进程的通信对其 TCP/IP 协议进行了验证。

CMC 采用了一种全新的模型检验思想，但是由于 CMC 的一个状态的数据量太大，导致 CMC 常常发生内存溢出的现象，于是 CMC 使用了一些启发式的方法在内存溢出前尽可能地遍历更多的代码与路径，所以从某种意义上说，CMC 只能是一个发现 BUG 的工具而不是一个验证工具，因为它往往不能遍历完程序的所有状态和路径。另外，CMC 目前只能验证一些简单的性质而不能验证复杂的、使用时序逻辑描述的性质。

此外，同类工具还包括 VeriSoft 以及 Java PathExplorer 等，它们的实现思想都与 CMC 比较类似。VeriSoft[38]面向可执行程序，能够验证由 C、C++、Tcl 等任何程序编写的可执行代码，它还支持对包含多个进程的并发程序的验证。与 CMC 不同的是，VeriSoft 采用了无状态的动态模型检验方法，即进行状态空间搜索时不保存任何状态。

Java PathExplorer[39, 40]则是美国 NASA 开发的面向 Java 程序的动态监测（Monitor）工具。它在 Java 程序中的适当位置插入代码，当程序执行这些代码时，这些内嵌代码就会向一个内嵌的事件监测器发送特定的事件信号，监测器通过对这些事件的汇总与分析，判断给定的时序性质是否能够被满足。

1.2.4.5 面向软件源程序的直接模型检验

模型检验方法的基本思想是遍历模型的所有可能执行路径，从而能够对所给性质进行彻底的验证。在数字硬件逻辑电路和系统中，由于其状态空间是有限的，因此模型检验方法最先在硬件系统的辅助设计与验证方面取得了巨大的成功。相比硬件，软件程序的状态空间往往大得多，甚至是无穷的，可能的原因包括：

- 软件程序的规模往往比硬件系统大得多，例如 Microsoft Word 程序的规模为 1,400,000 行数量级；
- 程序变量的数据类型所代表的数据域往往是无限的，如 int、float、string 等类型，因此要遍历一个变量的所有可能取值是不可能的；
- 程序中往往存在循环，根据输入参数的不同，循环被执行的次数是不同、甚至是无限的，这导致了程序的执行路径可能无穷长，以及程序可能拥有无穷多条执行路径；
- 并发程序中，往往需要考虑大量的并发进/线程，不同进/线程之间的交迭执行使得组合后的状态空间呈指数增长。

因此，面向软件源程序的直接模型检验的基础是模型抽象，即从程序中抽象出有限状态空间的模型。面向 C 程序的模型检验工具如 SLAM[41-43]、Zing[44-46]、BLAST[47]、MAGIC[48, 49]、ComFoRT[50-52]等，及面向 Java 程序的模型检验工具如

Java PathFinder[53, 54]等，他们都集成了模型抽象方法和工具。

1.2.5 谓词抽象方法

谓词抽象（Predicate Abstraction）最早是由 Graf 等提出来的[55]，它可被视为抽象解释的一种实例化，即可以基于抽象解释理论定义。谓词抽象本身只是一种面向程序的模型抽象方法，它把程序抽象为基于一组有限数量谓词表示的有限状态机模型，从而可以使用模型检验工具对其验证。结合反例制导的抽象精化 CEGAR（Counter-Example Guided Abstraction Refinement）方法[56]，基于谓词抽象的模型构建与检验方法可以对软件程序源代码进行全自动的验证，因此受到广大研究者的青睐，人们研制了数个验证工具原型，包括 Microsoft 的 SLAM[41-43]、Zing[44-46]、Berkeley California 大学的 BLAST[47]以及 Carnegie Mellon 大学的 MAGIC[48, 49]、ComFoRT[50, 52, 57]等。

设软件程序源代码为 C，对其使用基于谓词抽象方法所抽取的模型为 M，设它们所有执行路径的集合分别为 P_C 与 P_M。基于谓词抽象的模型抽取是一个保守的过程，可以保证 $P_C \subseteq P_M$。对待验证的安全性质而言，安全性是指程序中“某些坏事情永远不会发生[58]”，实际上描述了一组非法的程序执行路径，设这些路径的集合为 P_B，则验证的目标可以描述为验证 $P_B \cap P_C = \varnothing$。我们的验证基于模型 M 进行，验证 $P_B \cap P_M = \varnothing$ 是否成立，如果成立，则由于 $P_C \subseteq P_M$，我们可以保证 $P_B \cap P_C = \varnothing$ 成立，但如果 $P_B \cap P_M = \varnothing$ 不成立，设 $p \in P_B \cap P_M$，则我们要考查 $p \in P_C$ 是否成立，如果成立，则 p 为一条真实的反例路径，否则，我们基于伪反例路径 p 精化模型，加入新的谓词以排除路径 p，然后重新构建模型并验证。

设当前的谓词集合为 $\Theta = \{\varphi_1, \varphi_2, \cdots, \varphi_n\}$，基于该谓词集合抽象的模型每个状态定义为对谓词集合中所有谓词的一组赋值。我们定义向量 $\theta = \langle \theta_1, \theta_2, \cdots, \theta_n \rangle$ 为对谓词向量 $\varphi = \langle \varphi_1, \varphi_2, \cdots, \varphi_n \rangle$ 的一组赋值，其中 $\theta_i \in \{True, False\}$ 为对谓词 $\varphi_i \in \Theta$ 的赋值，那么 θ 就是模型的一个状态。可以看出，如果所给定的谓词集合包含 n 个谓词，那么所抽象出模型的状态空间大小为 2^n。为了构建出程序的模型，我们需要计算执行完赋值语句和 *if-then-else* 分支语句后模型的状态变迁。

设模型的当前状态为 $\theta = \langle \theta_1, \theta_2, \cdots, \theta_n \rangle$，我们需要计算通过赋值语句 $x := e$ 迁移到的下一个状态，也就是执行赋值语句 $x := e$ 后各个谓词的取值，这需要用到最弱前置条件的概念。对于一条赋值语句 t 和集合 Θ 中的某个谓词 φ_i，设 $WP(t)(\varphi_i)$ 为谓词 φ_i 相对语句 t 的最弱前置条件（*Weakest Precondition*）[59]，则 $WP(t)(\varphi_i)$ 是执行语句 t 之前的最弱条件，满足 $WP(t)(\varphi_i)$ 即可保证谓词 φ_i 在执行语句 t 结束后为真。我们定义 $WP(x := e)(\varphi_i)$ 为将 φ_i 中的所有变量 x 使用 e 替换后得到的新谓词，记为 $\varphi_i[e/x]$，例如：

$WP(x:=x+1)(x<2) = (x+1)<2 = x<1$。

为了确定φ_i在状态s'的取值，我们需要计算下面两公式是否成立：

$$(\varphi_1=\theta_1)\wedge(\varphi_2=\theta_2)\wedge\cdots\wedge(\varphi_n=\theta_n)\Rightarrow WP(t)(\varphi_i) \tag{1.1}$$

$$(\varphi_1=\theta_1)\wedge(\varphi_2=\theta_2)\wedge\cdots\wedge(\varphi_n=\theta_n)\Rightarrow WP(t)(\neg\varphi_i) \tag{1.2}$$

如果公式(1.1)成立，则我们可以保证执行语句 t 后谓词φ_i的取值为 *True*，如果公式(1.2)成立，则我们可以保证执行语句 t 后谓词φ_i的取值为 *False*，如果上述两个公式都不成立（注意它们不可能同时成立），则表示无法确定状态s'中谓词φ_i的取值，此时我们保守地假设状态s'中谓词φ_i的取值既可以为 *True* 也可以为 *False*，从而经过赋值语句 t 后，对φ_i而言状态s有两组后续状态，一组后续状态中φ_i取值为 *True*，另一组后续状态中φ_i取值为 *False*。我们可以看出，如果经过一个赋值语句t后无法确定k个谓词的取值，则一个状态经过p后将产生2^k个后续状态。

如果 t 为 *if-then-else* 分支语句，则执行分支语句的条件判断后，状态中谓词的取值并不会发生变化（因为没有改变谓词中变量的值），即$\theta'=\theta$，但由于分支语句有两个后续语句，因此我们的主要工作是确定分支语句的后续语句是否可行。设 t 为 "*if* (c) *then* t_1 *else* t_2" 形式，其中 c 为分支谓词，t_1 与 t_2 分别为其真假分支的后续语句。我们需要计算下列两公式是否成立：

$$(\varphi_1=\theta_1)\wedge(\varphi_2=\theta_2)\wedge\cdots\wedge(\varphi_n=\theta_n)\Rightarrow c \tag{1.3}$$

$$(\varphi_1=\theta_1)\wedge(\varphi_2=\theta_2)\wedge\cdots\wedge(\varphi_n=\theta_n)\Rightarrow \neg c \tag{1.4}$$

如果公式(1.3)为真，则只有分支条件为真的分支是可行的；如果公式(1.4)为真，那么只有分支条件为假的分支是可行的；如果公式(1.3)和公式(1.4)都不为真，那么我们保守地假设两个分支都是可行的，从而该分支语句具有两个后继状态。

1.3 课题研究思路

1.3.1 相关工作的小结与比较

静态分析方法的主要优点是简单实用、易于实现、可扩展性很强。它的主要缺点是它往往只考虑控制流、不考虑数据流，即较少涉及程序语义，因此分析的精度低。由于静态分析方法往往不能保证可靠性（Soundness）和完备性（Completeness），因此只能用于查找错误，而不能用于验证目的。ESP[10]虽然能够保证验证的完备性，但它的精确性是建立在代码编写良好的基础上的，否则会产生大量的错误误报。此外，ESP 对循环进行了近似处理，引入了额外的不精确。

定理证明方法与人类的逻辑推理思维非常接近，它的主要优点是可以适用于任意的程序，包括状态空间无限的程序，另外它可以高精度地验证混合数据流和控制流的命题。但它的主要缺点表现在它是一个非自动化的方法，需要人工提供

大量的公理、前提条件和与系统相关的信息如证明策略等，因此很难扩展到对大型实用软件的验证。

抽象解释方法本身是一个理论框架，提示了程序和抽象模型的语义之间的对应关系和一般规律，具体应用时需要对其进行实例化，即设计出合适的抽象和具体函数。基于抽象解释理论的代表性验证工具 ASTREE[19-22]和 C Global Surveyor（CGS）[24]目前只支持对 C 程序的运行时错误进行检查和验证，而不支持对更一般的时序安全性质的验证，从而不能满足对复杂软件高可信性质的验证需求。

模型检验方法是近年来发展迅速、应用广泛的方法，它的自动化程度很高，几乎无需人工交互，而且它支持对复杂的时序逻辑性质的验证，因此它可以验证复杂和隐藏得很深的错误，以及支持对丰富的性质[60, 61]的可靠验证。但状态空间爆炸问题一直是制约模型检验应用到大型实用程序的瓶颈，尤其对软件程序而言，状态空间爆炸问题更为严重。

基于程序切片的验证方法的核心是静态程序切片技术，静态程序切片只关注程序中的部分变量，从而能够大大降低验证的复杂度。但是，代价高昂的程序依赖关系图的计算是静态切片执行面临的主要问题。而且，如何确定尽量少的相关变量，以尽可能降低复杂度也是提高基于程序切片技术的验证方法的可扩展性的重要课题。

谓词抽象方法是一种面向程序的模型抽象方法，它与本文工作的目标和功能最相近，具有高度的相关性，因此我们对其优缺点进行详细的分析。谓词抽象方法的主要优点是能够满足当前模型检验方法的需求，包括：

- 谓词抽象使用符号化的隐式状态表示法，即生成模型的每个状态表示为一个逻辑公式，而不是显式地表示为每个程序变量的值。由于一个符号化的隐式状态可以表示大量、甚至无穷多个显式的状态，因此谓词抽象能够使用有限和少量的模型状态表示大量甚至无穷多的程序状态。例如，对于谓词 $x=y$，如果模型中的某个状态将该谓词赋值为 1，则代表了无穷多个 x 和 y 的取值，如 $x=y=1$、$x=y=2$ 等等；
- 给定谓词集合的前提下，谓词抽象从程序源代码中抽象出的模型的状态一定是有限的。因此，谓词抽象支持任意程序结构，如循环等，而不需要提供诸如循环不变式等外部辅助条件；
- 基于谓词抽象的模型生成是全自动进行的，完全不需要人工干预，甚至谓词集合也可以使用反例制导的抽象精化（CEGAR）方法，根据待验证的性质和验证过程中找到的伪反例路径自动生成，从而大大提高了易用性。
- 基于谓词抽象理论生成的程序模型的每个状态都表示为一个布尔值向量，因此用于存储模型状态的内存占用量很小，而内存资源是制约模型检验工

具应用于大型程序的另一个主要约束条件；

与此同时，谓词抽象理论也有它所固有的缺点，例如：

- 谓词抽象需要基于一组谓词的逻辑组合来保守地表示一个状态，这可能带来很大的不精确性。例如，给定谓词集合 $\Theta=\{x=0,x<5\}$，在状态 $\theta=\langle True,False\rangle$ 执行赋值语句 $x:=x+1$ 后得到的状态是 $\theta'=\langle False,True\rangle$。实际上，状态 θ 所描述的条件是 $x=0$，而状态 θ' 则使用条件 $x<5$ 来近似描述执行完赋值语句 $x:=x+1$ 后的最强后置条件 $x=1$，这显然带来了很大的误差，从而可能大大增加验证开销；
- 模型抽象过程中可能需要指数数量级的定理证明工具调用次数。我们知道，如果谓词集合中谓词的数量为 n 个，那么每个程序位置的状态数量可能多达 2^n 个。也就是说，对每一条赋值语句，我们最坏情况下需要调用定理证明工具 $2\times n\times 2^n$ 次，从而带来指数级的时间复杂度。

1.3.2 课题研究目标

本文研究的动机和目标，是面向较复杂的软件高可信性质验证，重点着眼于提高验证方法的可扩展性，同时保证验证过程的高度自动化，以提高验证方法和相应工具的实用性。进一步地，我们还要求验证方法具有尽可能高的精确性，尽量找出程序中的所有错误（即降低漏报率），以及尽量保证报告的所有错误都是程序中的真实错误（即降低误报率）。具体地，本文研究所面向的目标如下：

- 面向时序安全性质（Temporal Safety Property）的验证：从而能够验证复杂的高可信软件需求；
- 具有高度的精确性：如果所用定理证明工具能够保证可靠和完备，则理论上尽量不漏报和误报任何错误。
- 具有高度的可扩展性：要求其可扩展性与当前主流方法如谓词抽象相当甚至更好；
- 具有高度的自动化程度：要求验证之前不需要手动提供任何额外信息，验证过程中也完全无需人工交互；
- 对程序结构和特点没有任何要求：要求能够支持任意形式的循环，而不需要进行近似或引入额外的工作；

从技术层面上看，本文的工作属于面向软件源程序的直接模型检验方法，能够直接基于软件源程序验证给定的时序安全性质是否满足。由于模型检验方法只能对有穷状态空间的模型进行验证，因此将模型检验方法应用于软件程序验证的关键是抽象出有穷状态模型，并要大幅地缩减模型的状态空间，这正是本课题研究的主要难点。

1.3.3 课题研究内容

总的来说，切片执行（Slicing Execution）方法的目标是从C程序中自动抽象有限状态模型，并基于模型检验的方法对时序安全性质（Temporal Safety Property）进行验证。其体系结构如图1.1所示，它主要包括四大组成部分，这四大部分是一个有机的整体，集成到切片执行框架中，对顺序和并发C程序进行自动验证。

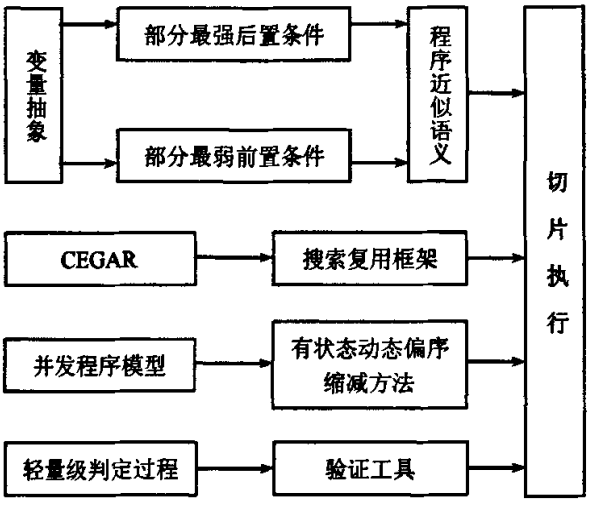

图1.1 切片执行的技术体系组成

我们首先提出了变量抽象（Variable Abstraction）方法，根据待验证的时序安全性质鉴别程序变量和程序语句的相关性。变量抽象与程序切片[62]的思想有相似之处，都是根据程序语句所涉及的程序变量来决定其相关性。但是，变量抽象只是简单地比较程序语句所涉及变量是否属于某个给定的变量集合，省去了代价高昂的依赖关系计算（参见[63, 64]），从而其时空开销都大大小于程序切片。

在变量抽象的基础上，我们提出了部分最强后置条件（Partial Strongest Post-Condition）和部分最弱前置条件（Partial Weakest Precondition）的概念，它们是对传统最强后置条件和最弱前置条件的保守近似，只考虑变量抽象下的相关变量和语句，从而对程序的行为进行了抽象和近似。因此，由部分最强后置条件和部分最弱前置条件描述的程序语义是对传统程序语义的保守近似，体现了程序抽象的思想。程序的保守近似语义是切片执行的基础，切片执行通过检验保守近似语义下的程序行为是否满足给定性质来判断程序自身是否满足给定的性质。

反例路径指导的抽象精化(CEGAR)方法是切片执行的总体框架，基于CEGAR框架的切片执行是一个迭代的过程，在每次迭代时对模型进行抽象并验证所给性质，如果找到了模型中违背性质但却在程序中不存在的反例路径（我们称其为伪

反例路径），则基于该路径精化变量抽象的抽象准则，以考查程序中更多的变量和语句，从而得到更加精确的程序语义。

在CEGAR框架的基础上，我们提出了搜索复用框架，该框架除了具有CEGAR框架的优点外，还能够复用每次迭代过程中的状态搜索信息，以避免对不会违背给定性质的状态和执行路径的重复搜索，从而进一步提高切片执行的效率。

为了对并发C程序进行验证，我们先提出了基于*P/V*同步原语的并发C程序模型，并提出了有状态动态偏序缩减（Stateful Dynamic Partial-Order Reduction）方法，以缩减切片执行需要搜索的并发C程序模型的状态空间。偏序缩减特别是动态偏序缩减是一种能够有效缩减并发程序模型状态空间的技术，我们提出的有状态动态偏序缩减方法解决了原有动态偏序缩减不支持有状态搜索的问题，可用于切片执行等有状态模型检验工具。

我们注意到切片执行会大量地调用定理证明工具对切片执行过程中的一阶逻辑公式进行判定。为了提高切片执行的效率，我们提出了一种轻量级的判定过程，该过程支持对包含整数线性和给定类型位运算的一阶逻辑公式的判定，从而能够对切片执行过程中出现的绝大部分判定公式进行高效的判定。

1.3.4 课题技术特色

从技术上看，本文工作的目标是充分汲取当前各种面向软件源程序的模型抽象方法的优点，同时试图（部分）克服现有方法的缺点，从而提高模型抽象的效率，以及有效缩减抽象模型的状态空间。从技术和原理上看，切片执行方法和工具具有如下的特点：

- 切片执行使用精确的符号化隐式状态表示方法，如果抽象过程能够终止，则所抽象出的模型的状态空间一定是有限的，而且模型的每个状态都用一个一阶逻辑公式精确描述。这样的表示方法能够大大缩减生成模型的状态空间，特别是能够克服状态描述不精确造成的模型状态空间的无意义膨胀。但是，与谓词抽象相比，由于每个状态都对应于一个一阶逻辑公式，其状态储存所需要的内存空间更大，这是切片执行的缺点；
- 面向时序安全性质，切片执行采用“边抽象、边检验”的验证方式，使得在切片执行过程中的任何时刻，都只需要保存抽象模型状态空间的很少一部分状态信息，从而能够大大减小切片执行的空间复杂度，较好地平衡状态存储的内存空间占用；
- 切片执行能够全自动地进行，无需人工交互。它与基于谓词抽象的模型检验理论一样，基于反例指导的抽象精化（CEGAR）思想自动从伪反例路径中提取必要信息细化抽象粒度，直到性质被证明、或找到一条真正的反

例路径为止；

- 切片执行支持任意的程序结构，包括循环，也不需要用户提供诸如循环不变式等外部辅助信息，从而保证了验证工具的全自动进行；
- 切片执行从一定程度上克服了指数数量级的定理证明工具调用次数，实现为一种轻量级的符号执行，能以接近数据流分析的代价对程序进行路径敏感的精确验证；
- 切片执行基于搜索复用框架实现，搜索复用框架是对CEGAR框架的改进，支持在不同精度的模型之间尽可能地共享搜索信息，从而既降低了模型生成的复杂度，又缩减了生成模型的状态空间；
- 切片执行不仅支持串行 C 程序，也支持对并发 C 程序的自动模型抽象和验证。同时，它还采用改进的动态偏序缩减方法来缩减并发程序模型的状态空间，效率显著。

1.4 课题创新点

我们首先介绍本文提出的切片执行方法的整体创新思路，再详细介绍每个创新点。

由于可扩展性是面向软件源程序的形式验证所面临的主要问题，因此切片执行方法研究的动机和目标是提高可验证程序的规模。我们注意到如下事实：软件源程序包含了软件的所有行为和动作，不同的软件行为和动作面向不同的功能需求，体现软件设计人员不同层次和角度的关注点；而待验证的性质往往站在某一特定的关注点上，描述某一特定的软件功能需求；因此在实际的程序验证过程中，待验证的性质通常只涉及到一小部分程序变量。基于上述事实，我们提出了变量抽象的概念，其基本思想在验证过程中只考查与性质相关的少部分变量，这样能够大大降低验证开销，从而提高可扩展性。

为了对程序进行精确的验证，我们需要考查程序的语义。传统的程序语义是基于所有的程序变量和语句定义的，描述了程序的精确行为。为了应用变量抽象方法，我们必须定义一种新的程序语义描述方法，以描述只考查部分变量和语句的程序近似行为。为此，我们提出了部分最强后置条件和部分最弱前置条件这两种程序近似语义描述方法。

由于变量抽象并不计算变量依赖关系，所以不能保证性质验证必需的所有变量都被考查。因此，切片执行构建在反例制导的抽象精化（CEGAR）框架之上，利用验证过程中出现的伪反例路径进行模型的精化，这样的验证模式能够在保证性质被验证的前提下，考查尽可能少的程序变量，从而尽量降低验证代价。我们提出的搜索复用框架进一步降低了切片执行的验证开销。

将切片执行方法拓展到对并发 C 程序验证后，为了降低并发程序模型的状态空间搜索代价，我们提出了适用于有状态模型检验的有状态动态偏序缩减方法，作为该方法的一个应用，我们将其集成到切片执行中。

为了对切片执行过程中产生的大量一阶逻辑判定公式进行判定，我们提出了一种轻量级判定方法，它能够大大提高切片执行的效率。

具体地，本论文面向顺序 C 程序和并发 C 程序的时序安全性质验证，提出了切片执行的概念，并围绕切片执行进行了比较系统和深入的研究，提出了比较完整的技术体系。主要工作和创新点体现在如下方面：

1. 提出了切片执行的基本概念和方法（第 2 章）。切片执行的最大特点是结合了抽象和符号化状态表示这两种策略，能够大量缩减抽象模型的状态空间。我们首先分析了程序验证的基本规律，提出了变量抽象方法，变量抽象只考虑程序源代码中与待验证的时序安全性质相关的程序变量和语句，其抽象准则能够根据验证过程中的伪反例路径自动精化。基于变量抽象，我们定义了部分最强后置条件，进而定义了程序的保守近似语义。基于这两个概念，我们提出了切片执行的概念，它是一种轻量级的符号执行过程。考虑到性质验证往往只涉及极少数的程序变量，切片执行方法能够以接近标准流敏感数据流分析的代价，达到路径敏感程序模拟的验证精度。基于 *openssl-0.9.6c* 实用程序的实验数据表明，切片执行的验证效率总体上来看优于谓词抽象。
2. 提出了面向时序安全性质验证的搜索复用框架并将其应用于切片执行（第 3 章）。搜索复用框架也是一种反例路径制导的抽象精化框架，基于伪反例路径进行模型精化。相比传统的 CEGAR 框架，其最大特点是能够在不同精度的抽象模型之间进行充分的信息复用，从而避免了大量不必要的重复搜索，有效降低了验证开销。基于 *openssl-0.9.6c* 的实验结果表明，将搜索复用框架应用于切片执行后能够较大程度地提高大部分程序和性质的验证效率。
3. 提出了部分最弱前置条件的概念并将其应用于切片执行（第 4 章）。部分最弱前置条件是程序保守近似语义的另一种表示方法，同样基于变量抽象定义。在切片执行过程中，可以用部分最弱前置条件部分地取代部分最强后置条件，以生成更弱的一阶逻辑公式来描述抽象模型的同一个状态，从而在不影响模型精度的前提下大大缩减生成模型的状态空间。基于 *openssl-0.9.6c* 的实验结果表明，应用部分最弱前置条件的切片执行效率平均提高 10 多倍。
4. 提出了面向有状态模型检验的有状态动态偏序缩减方法（第 5 章），并将

其集成到切片执行过程中（第 6 章）。我们将切片执行方法扩展到了对并发 C 程序的验证，为了进一步提高状态空间缩减的效果，我们提出了有状态动态偏序缩减方法，并将其自然地集成到切片执行框架中，用于指导切片执行，使其避免搜索多条具有相同偏序关系的并发进/线程交迭执行路径。实验结果证明，切片执行和有状态动态偏序缩减这两种正交的状态空间缩减方法的集成，大大缩减了并发程序抽象模型的状态空间，特别是降低了验证的空间开销。

5. 提出了面向切片执行的轻量级判定过程（第 7 章），实现了切片执行原型工具（贯穿于论文各章）。我们定义了一类整数线性一阶逻辑判定公式，此类判定公式支持 C 程序中常用的整数线性运算。在此基础上，我们扩充了对整数除法、取余和位运算的支持，并给出了相应的轻量级判定过程。实验表明，定义的判定公式覆盖了面向 C 程序的切片执行过程中所产生的绝大多数验证公式。在基于 *openssl-0.9.6c* 的实验中我们发现，提出的判定方法的判定效率相比定理证明工具 Simplify 提高了 10.5 倍。我们还基于开放源代码的 MAGIC 项目实现了切片执行工具原型，我们在切片执行工具原型中实现了对 C 程序的指针和变量别名的支持，从而使得工具能够对实用的程序进行验证，论文各章的实验数据就是使用该工具产生的。

为了对切片执行方法和工具进行检验，我们将切片执行得出的实验数据与美国 UC Berkeley 的 BLAST[47]以及 Carnegie Mellon 大学的 MAGIC[48, 49]这两个工具进行了充分的对比。我们在实验时采用了与 BLAST 和 MAGIC 相同的硬件平台，并针对相同的验证用例（即 *openssl-0.9.6c*），验证了相同的性质集合，因此实验结果具有可比性。经过实验对比，切片执行在验证效率上显示了一定的优势。

1.5 论文结构

第一章是本论文的绪论部分，首先介绍了课题研究的背景和相关工作，再通过分析当前面向源代码验证理论的优缺点，介绍了本文研究的目标定位，引出了本文的研究内容，并概括了研究的技术特色，最后介绍了本文的创新点和论文的整体结构。

第二章介绍了切片执行的基本概念和理论，包括变量抽象、部分最强后置条件、切片执行图等概念，提出了基于 CEGAR 框架的切片执行过程，并给出了基于 *openssl-0.9.6c* 实用程序的实验结果。

第三章介绍了搜索复用框架，它是对 CEGAR 框架的改进，并介绍了其在切片执行过程中的应用，最后给出了应用两个不同框架的实验结果比较。

第四章介绍了部分最弱前置条件的概念，并给出集成了部分最弱前置条件的

新切片执行算法，最后比较了实验结果。

第五章介绍了面向有状态模型检验的动态偏序缩减方法，即有状态动态偏序缩减方法，它既保持了动态偏序缩减方法的精确性和状态空间缩减效率，又能够与有状态模型检验工具自然集成，从而能够更大幅度地缩减状态空间。

第六章提出了面向并发 C 程序验证的切片执行方法，并介绍了与动态偏序缩减方法的集成。为清晰起见，第六章提出了三种切片执行方法，分别是面向并发 C 程序的基本切片执行、集成无状态动态偏序缩减方法的切片执行和集成有状态动态偏序缩减方法的切片执行，最后通过实验对它们进行了比较。

第七章定义了一类包括整数除法、取余和位运算在内的整数线性一阶逻辑判定公式，并给出了轻量级的判定过程和实验结果。

第八章是结束语，分析了本文研究工作的特点和不足，并对未来工作进行了展望。

第二章　切片执行的基本概念和方法

软件源程序包含了软件的所有行为和动作，不同的软件行为和动作面向不同的功能需求，体现软件设计人员不同层次和角度的关注点。而待验证的性质往往站在某一特定的关注点上，描述某一特定的软件功能需求。因此，在实际的程序验证过程中，待验证的性质往往只涉及到一小部分程序变量。基于该事实的程序切片（Program Slicing）[62]技术在程序调试领域取得了很大的成功、得到了广泛的应用。同样，我们基于这个事实提出了一种简单的抽象方法，即变量抽象（Variable Abstraction），它与程序切片类似，根据程序语句中包含的变量判定该语句是否是与待验证性质相关的语句。但是，变量抽象不需要进行相关性分析（Dependence Analysis），从而其计算代价要大大小于程序切片。变量抽象的抽象准则（Abstraction Criterion）定义为一组程序变量的集合。由于抽象准则中变量的数量直接决定了切片执行的代价，因此我们使用反例制导的抽象精化（Counter-Example Guided Abstract Refinement，CEGAR）方法[56]，来自动地推断出包含变量数量尽可能少的抽象准则。

接下来，我们提出了部分最强后置条件（Partial Strongest Post-Condition）的概念，用于描述基于变量抽象的程序保守近似语义。切片执行被定义为在程序的保守近似语义下的一种轻量级的符号执行，它仅符号执行与待验证的性质相关的程序语句。切片执行遍历程序的所有执行路径，并计算程序每一点处的部分最强后置条件，然后基于部分最强后置条件保守地判定程序的可行分支。如果切片执行过程能够终止，它将产生一个抽象的符号执行图，我们称其为切片执行图（Slicing Execution Graph）。切片执行图就是我们所关心的程序的抽象模型，它是有穷的，并且从理论上可以保证包含程序的所有可能执行路径，因此如果切片执行图满足给定的时序安全性质，那么程序自身也一定满足。

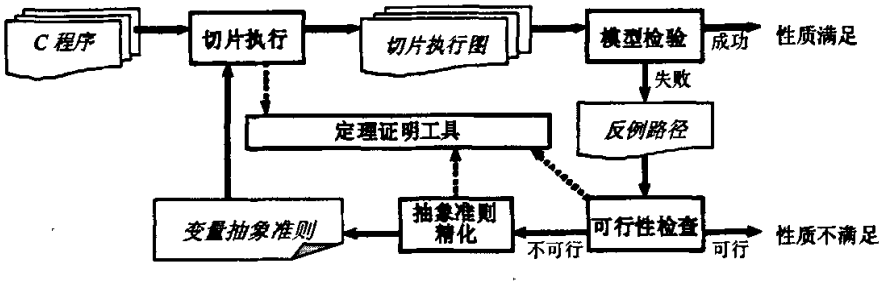

图 2.1　基于切片执行的 C 程序验证框架

图 2.1 描述了基于切片执行的 C 程序验证框架，它是一个迭代进行的过程，每

次迭代包括模型构建、模型检验、反例检查和模型精化四个主要步骤。初始时，变量抽象准则包含了时序安全性质中所涉及的程序变量，切片执行基于这个初始的抽象准则产生切片执行图，作为 C 程序的初始抽象模型。接下来切片执行图被送到模型检验工具进行检验，如果模型检验工具没有发现违背所给性质的反例路径，那么我们就证明了所给 C 程序满足给定的时序安全性质；否则，由于抽象出的模型中包含了大量在原来的程序中并不存在的路径，我们需要对模型检验工具找到的反例路径进行可行性检查，以确定它是否在原来的程序中真实存在，如果是，我们就找到了程序的一条不满足所给性质的反例路径，否则我们需要根据这条伪反例路径来精化变量抽象准则，从而在下一次迭代时生成精化的模型，而且我们可以保证精化的模型中不再包含这条伪反例路径。

我们知道，程序中的分支语句导致了程序的不同执行路径，而分支语句的串行组合则往往导致程序执行路径的数量呈指数增长。但是切片执行仅仅考虑少部分的程序变量，对于这些程序变量而言，往往大量的程序执行路径对它们进行了相同或相似的操作，从而这些执行路径将对应相同的部分最强后置条件而被切片执行合并到同一个状态，这将导致显著的状态空间缩减。为了证明这一点，基于 Linux 操作系统中 SSL 协议的实现程序 *openssl-0.9.6c*，我们验证了 SSL 协议的初始握手协议的若干个时序安全性质。实验结果表明，切片执行能够以接近标准流敏感数据流分析的代价，实现路径敏感的验证精度。

2.1 基本模型和概念

2.1.1 C 程序模型和时序安全性质

与 MAGIC[48]和 BLAST[47]一样，我们假设待验证的 C 程序不包含递归函数调用。通过将被调用函数的函数体嵌入到对该函数的调用点，我们能够得到一个只包含单一函数体的等价 C 程序。除此之外，我们将程序中的所有 *for* 和 *while* 循环改写为基于 *if* 和 *goto* 语句的等价形式。另外，我们还假设变换后的程序中的表达式没有副作用，对于有副作用的表达式，我们将之变换为多个等价的无副作用表达式。经过上述变换，我们可以假设程序中只包括单个函数，函数中只包括赋值语句（*assignment*）、*if-then-else* 分支语句、*goto* 语句和 *return* 语句（文献[48]有相同的假设），其中 *return* 语句表示程序控制流的结束。我们可以使用一些工具自动完成上述工作，例如 CIL[65]与 LLVM[66]等。

我们将 C 程序模型定义为标记迁移系统（Labeled Transition System，LTS）的形式。设 $CFG=\langle S,E\rangle$ 为一个 C 程序的控制流图(Control Flow Graph，CFG)，其中 S 是节点的集合，E 是边的集合。那么它对应的 LTS 定义为 $CP=\langle S,s_0,T,\Delta\rangle$，其

中 S 是程序位置（location）的集合，对应于 CFG 的节点集合，$s_0 \in S$ 是程序的初始位置或初始程序节点，T 是迁移的集合，$\Delta \subseteq S \times T \times S$ 是迁移关系。我们定义，对程序控制流图 $CFG = \langle S, E \rangle$ 中的每条边 $(s_1 \to s_2) \in E$（其中 $s_1, s_2 \in S$），都有一个对应的迁移关系 $\langle s_1, t, s_2 \rangle \in \Delta$，其中迁移 t 定义为：

- $t = s_1$；如果 s_1 是赋值语句（*assignment*），或者；
- 如果 s_1 是分支语句 "*if*(c) s_A; *else* s_B;"，那么 $t = assume(c)$ 如果 $s_2 = s_A$；或者 $t = assume(\neg c)$ 如果 $s_2 = s_B$。

对 C 程序所对应的标记迁移系统 $CP = \langle S, s_0, T, \Delta \rangle$ 来说，其迁移集合 T 中只包含两种语句，一种是赋值语句，另一种是 assume 语句。其中 assume 语句对应的迁移描述了系统迁移到下一个状态所必须满足的条件，如果当前状态的变量取值不能满足 assume 语句给出的条件，则该迁移不能发生。

如果 $t_i \in T$，并且存在 $s_1, s_2, \cdots, s_n \in S$ 使得对于所有满足 $1 \le i \le n$ 的 i 都有 $\langle s_{i-1}, t_i, s_i \rangle \in \Delta$（其中 s_0 是 $CP = \langle S, s_0, T, \Delta \rangle$ 的初始位置），我们就称 $p = t_1 t_2 \cdots t_n$ 是一条执行路径。

本文假设所有待验证的时序安全性质都描述为有限自动机（Finite State Automata，FSA）的形式，自动机的状态迁移事件应该能够直接映射到程序中，常用的状态迁移事件包括对某个函数的调用、对某个或某一类型变量的某种操作、执行到某个程序位置、满足某种约束条件等。此外，**我们定义性质自动机的接收状态表示性质的违背**，也就是说，如果程序的某条执行路径使得性质自动机迁移到了其接收状态，那么该执行路径就是程序违背给定性质的一条反例路径。我们规定，本论文中的所有"性质自动机"都是指这样的自动机。

2.1.2 最强后置条件

根据 Hoare 逻辑，程序语义可以基于最强后置条件[59]定义。程序语句 t 的最强后置条件记为 $SP(t)$，如果执行语句 t 之前程序变量的取值满足条件 c，那么 $SP(t)(c)$ 描述的是执行完语句 t 之后程序变量的取值所能满足的最强条件。例如，$SP(x := x+1)(x > 0) = (x > 1)$，表明如果在执行赋值语句 $x := x+1$ 之前变量 x 的取值满足条件 $x > 0$，那么执行完该赋值语句之后变量 x 所满足的最强条件是 $x > 1$。

赋值语句和 *assume* 语句的最强后置条件定义如下[42, 59]：

$$SP(x := e) = \lambda f. \exists x'. f[x'/x] \wedge (x = e[x'/x]) \tag{2.1}$$

$$SP(assume(c)) = \lambda f. f \wedge c \tag{2.2}$$

其中 $f[x'/x]$ 和 $e[x'/x]$ 分别定义为将公式 f 和表达式 e 中所有变量 x 的自由出现用一个新变量 x' 替换。

我们可以看出，每计算一次赋值语句的最强后置条件，都会引入一个存在量

词。根据文献[42]，我们通过引入 Skolem 常量的方法能够消除最强后置条件中的存在量词，从而既简化了最强后置条件自身，又简化了对其进行判定的复杂度。对一条执行路径 p 中的每个赋值语句 $x := e$ 和 assume 语句 $assume(c)$，如果表达式 e 或者 c 引用了某个之前没有定义的变量 y，那么我们就引入一个 Skolem 常量 θ_y 表示变量 y 的当前取值可以为任意值，并在上述赋值语句或 assume 语句之前插入一条新的赋值语句 $y := \theta_y$。设函数 $Vars(e)$返回表达式 e 中包含的所有变量的集合，例如 $Vars(x > y) = \{x, y\}$。我们将该函数延拓到执行路径上，即对某执行路径 $p = t_1 t_2 \cdots t_n$，我们定义 $Vars(p) = \bigcup_{1 \le i \le n} Vars(t_i)$。那么在引入合适的 Skolem 常量后，$Vars(p)$中的所有变量都拥有初值。

借用文献[42]的方法，我们使用二元偶 $\langle \Omega, \Phi \rangle$ 来描述和计算最强后置条件，其中 Ω 是一个部分函数 $\Omega : Vars(p) \mapsto [\![Exp]\!]$，这里 $[\![Exp]\!]$ 代表了表达式的全集，Φ 是一个集合，存储了由 assume 语句引入的所有布尔表达式。为了描述方便，我们将 Ω 按常规做法延拓到表达式域。基于 $\langle \Omega, \Phi \rangle$ 二元偶表示形式的最强后置条件定义如下[42]：

$$SP(x := e) = \lambda \langle \Omega, \Phi \rangle . \langle \Omega[x \to \Omega(e)], \Phi \rangle \tag{2.3}$$

$$SP(assume(c)) = \lambda \langle \Omega, \Phi \rangle . \langle \Omega, \Phi \cup \Omega(c) \rangle \tag{2.4}$$

其中函数 $\Omega[x \to e]$ 定义如下：

$$\Omega[x \to e](y) = \begin{cases} \Omega(y) & if\ y \neq x \\ e & if\ y = x \end{cases} \tag{2.5}$$

我们将最强后置条件延拓到执行路径上，执行路径 $p = t_1 t_2 \cdots t_n$ 的最强后置条件定义为 $SP(p) = SP(t_n) \circ SP(t_{n-1}) \circ \cdots \circ SP(t_1)$，其中函数组合符号“$\circ$”定义为从右向左组合，即 $g \circ h = \lambda x.g(h(x))$[42]。如果 $SP(p)(True) \neq False$，则执行路径 p 是可行的（Feasible），其中 $True$ 定义为 $\Omega = \Phi = \varnothing$。一条执行路径是否可行取决于路径中的所有 assume 语句是否有冲突，因此在实践中，我们常用与公式 $SP(p)(True) \neq False$ 等价的公式(2.6)来判定可行性：

$$\left(\bigwedge_{\phi \in \Phi} \phi \right) \neq False \tag{2.6}$$

2.2 基于变量抽象的程序保守近似语义

2.2.1 变量抽象

变量抽象的基本思想与静态程序切片[62]的思想类似，但是目标和计算复杂度都存在较大的差别。给定作为抽象准则的一组程序变量的集合 V，变量抽象的目标是判定一条迁移语句是否为相关语句，而不是像程序切片那样寻找程序中与某条

语句相关的所有语句。因此，变量抽象不需要进行代价高昂的依赖性分析，其计算代价相比程序切片小得多。

定义 2.1 在抽象准则 V 下，赋值语句 $x := e$ 的相关性定义如下，其中 x 是变量，e 是 C 语言表达式：

- 如果 $Vars(x) \subseteq V$ 且 $Vars(e) \subseteq V$，我们称 $x := e$ 为抽象准则 V 下的**完全相关**赋值语句；
- 如果 $Vars(x) \subseteq V$ 但 $Vars(e) \not\subseteq V$，我们称 $x := e$ 为抽象准则 V 下的**部分相关**赋值语句；
- 如果 $Vars(x) \not\subseteq V$，我们称 $x := e$ 为抽象准则 V 下的**无关**赋值语句。

定义 2.2 在抽象准则 V 下，assume 语句 *assume*(c)的相关性定义如下，其中 c 是 C 语言的布尔表达式：

- 如果 $Vars(c) \subseteq V$，我们称 *assume*(c)为抽象准则 V 下的**相关** assume 语句；
- 如果 $Vars(c) \not\subseteq V$，我们称 *assume*(c)为抽象准则 V 下的**无关** assume 语句。

例如，设抽象准则 $V = \{x, z\}$，那么 $x := z + 1$ 为完全相关的赋值语句，$x := z + y$ 为部分相关的赋值语句，而 $y := x$ 则为无关的赋值语句。同样的抽象准则 $V = \{x, z\}$ 下，*assume*$(x > z)$ 为相关的 assume 语句，而 *assume*$(x > y)$ 则为无关的赋值语句。

2.2.2 部分最强后置条件

传统的最强后置条件考查了所有的程序变量，从而能够精确地定义每条程序语句、每条执行路径和整个程序的行为。而在面向程序源代码进行验证时，为了缩减状态空间，往往只需要考查与待验证的性质相关的程序的抽象和近似行为。因此，我们在变量抽象的基础上提出了一种保守近似的最强后置条件，即抽象准则 V 下的部分最强后置条件，记为 $\widetilde{SP}_V$，用于描述程序中完全相关和部分相关语句的行为。同时，为了面向时序安全性质的验证，我们需要保证精确程序语义下的所有可行执行路径都是保守近似程序语义下的可行执行路径。下面我们将给出部分最强后置条件的定义和计算方法，并证明它满足这个约束条件。

定义 2.3 给定抽象准则 V，赋值语句 $x := e$ 的部分最强后置条件 $\widetilde{SP}_V(x := e)$ 定义为：

- 如果 $x := e$ 是抽象准则 V 下的完全相关赋值语句，那么
  $\widetilde{SP}_V(x := e) = \lambda f.\exists x'.f[x'/x] \wedge (x = e[x'/x])$
- 如果 $x := e$ 是抽象准则 V 下的部分相关赋值语句，那么
  $\widetilde{SP}_V(x := e) = \lambda f.\exists x'.f[x'/x]$

- 如果 $x:=e$ 是抽象准则 V 下的无关赋值语句，那么
  $\widetilde{SP}_V(x:=e)=\lambda f.f$

定义 2.4 给定抽象准则 V，assume 语句 $assume(c)$的部分最强后置条件 $\widetilde{SP}_V(assume(c))$定义为：

- 如果 $assume(c)$是抽象准则 V 下的相关 assume 语句，那么
  $\widetilde{SP}_V(assume(c))=\lambda f.f\wedge c$
- 如果 $assume(c)$是抽象准则 V 下的无关 assume 语句，那么
  $\widetilde{SP}_V(assume(c))=\lambda f.f$

引理 2.1 对任意抽象准则 V 和任意迁移语句 t，若 $f_1\Rightarrow f_2$，则 $\widetilde{SP}_V(t)(f_1)\Rightarrow\widetilde{SP}_V(t)(f_2)$。

证明：对迁移 t 分下列情况讨论：

- 若 t 为抽象准则 V 下的完全相关赋值语句 $x:=e$，则 $\widetilde{SP}_V(x:=e)(f_1)=\exists x'.f_1[x'/x]\wedge(x=e[x'/x])$ 且 $\widetilde{SP}_V(x:=e)(f_2)=\exists x'.f_2[x'/x]\wedge(x=e[x'/x])$。由于 $f_1\Rightarrow f_2$，故有 $f_1[x'/x]\Rightarrow f_2[x'/x]$，因此 $\widetilde{SP}_V(t)(f_1)\Rightarrow\widetilde{SP}_V(t)(f_2)$；
- 若 t 为抽象准则 V 下的部分相关赋值语句 $x:=e$，则 $\widetilde{SP}_V(x:=e)(f_1)=\exists x'.f_1[x'/x]$ 且 $\widetilde{SP}_V(x:=e)(f_2)=\exists x'.f_2[x'/x]$。由于 $f_1\Rightarrow f_2$，故有 $f_1[x'/x]\Rightarrow f_2[x'/x]$，从而 $\widetilde{SP}_V(t)(f_1)\Rightarrow\widetilde{SP}_V(t)(f_2)$；
- 若 t 为抽象准则 V 下的无关赋值语句 $x:=e$，则 $\widetilde{SP}_V(x:=e)(f_1)=f_1$、$\widetilde{SP}_V(x:=e)(f_2)=f_2$，从而 $\widetilde{SP}_V(t)(f_1)\Rightarrow\widetilde{SP}_V(t)(f_2)$；
- 若 t 为抽象准则 V 下的相关 assume 语句 $assume(c)$，则 $\widetilde{SP}_V(assume(c))(f_1)=f_1\wedge c$、$\widetilde{SP}_V(assume(c))(f_2)=f_2\wedge c$，根据 $f_1\Rightarrow f_2$ 知 $f_1\wedge c\Rightarrow f_2\wedge c$，从而 $\widetilde{SP}_V(t)(f_1)\Rightarrow\widetilde{SP}_V(t)(f_2)$；
- 若 t 为抽象准则 V 下的无关 assume 语句 $assume(c)$，则 $\widetilde{SP}_V(assume(c))(f_1)=f_1$、$\widetilde{SP}_V(assume(c))(f_2)=f_2$，根据 $f_1\Rightarrow f_2$ 知 $\widetilde{SP}_V(t)(f_1)\Rightarrow\widetilde{SP}_V(t)(f_2)$。

综上，引理证毕。 ■

直观地，引理 2.1 说明了部分最强后置条件是一个单调函数。为了消除部分最强后置条件中的存在量词，我们也将部分最强后置条件表示为二元偶 $\langle\Omega,\Phi\rangle$ 并基于其进行计算。此外，我们用 $[\![PSP]\!]$ 表示所有部分最强后置条件的全集。

定义 2.5 给定抽象准则 V，$\widetilde{SP}_V(x:=e)$ 和 $\widetilde{SP}_V(assume(c))$ 定义如下（其中函数 $\Omega[x\to\Omega(e)]$ 和 $\Omega[x\to\theta_e]$ 的定义见 2.1.2 节最强后置条件的公式(2.5)）：

$$\widetilde{SP}_V(x:=e)(\langle\Omega,\Phi\rangle) = \begin{cases}\langle\Omega[x\to\Omega(e)],\Phi\rangle & if\ x:=e\text{完全相关}\\ \langle\Omega[x\to\theta_e],\Phi\rangle & if\ x:=e\text{部分相关}\\ \langle\Omega,\Phi\rangle & if\ x:=e\text{无关}\end{cases} \tag{2.7}$$

$$\widetilde{SP}_V(assume(c))(\langle\Omega,\Phi\rangle)=\begin{cases}\langle\Omega,\Phi\cup\Omega(c)\rangle & if\ assume(c)\text{相关}\\ \langle\Omega,\Phi\rangle & if\ assume(c)\text{无关}\end{cases} \tag{2.8}$$

在公式(2.7)的第二种情况中，由于表达式 e 中的某些变量不属于抽象准则 V，因此我们无法计算得知表达式 e 的值，所以我们引入一个 Skolem 常量 θ_e 来表示表达式 e 的值。直观上来看，由于 Skolem 常量 θ_e 可以取值为任意值，这就相当于保守地假设部分相关赋值语句 $x:=e$ 执行后变量 x 可以取值为任意值，从而确保不会漏掉实际程序中的任何一种情况。

对于程序中的一条执行路径 $p=t_1t_2\cdots t_n$，其最强后置条件 $SP(p)(True)$ 定义为执行路径 p 后所有程序变量能够满足的最强条件。而 $\widetilde{SP}_V(p)(True)$ 则描述了执行完路径 p 后抽象准则 V 中的程序变量所满足的最强条件。我们同样定义 $\widetilde{SP}_V(p)=\widetilde{SP}_V(t_n)\circ\widetilde{SP}_V(t_{n-1})\circ\cdots\circ\widetilde{SP}_V(t_1)$，其中函数组合符号“$\circ$”也同样定义为从右向左组合，即 $g\circ h=\lambda x.g(h(x))$。

为了确知二元偶 $\langle\Omega,\Phi\rangle$ 所描述的约束条件，我们定义一个函数 h 将该二元偶转换到一个一阶逻辑公式，如下：

$$h(\langle\Omega,\Phi\rangle)\triangleq\exists\theta_1,\cdots,\theta_m.\left(\left(\bigwedge_{\omega\in\Omega}e2b(\omega)\right)\wedge\left(\bigwedge_{\phi\in\Phi}\phi\right)\right) \tag{2.9}$$

其中 $\theta_1,\cdots,\theta_m$ 为二元偶 $\langle\Omega,\Phi\rangle$ 中包含的所有 Skolem 常量，函数 $e2b(\omega)$ 将 Ω 中的每个变量映射 ω 转化为相应的布尔表达式，例如 $e2b(x\to 1)\triangleq(x=1)$。与传统的最强后置条件一样，我们有如下的等价公式：

$$\left(h(\langle\Omega,\Phi\rangle)=False\right)\Leftrightarrow\left(\bigwedge_{\phi\in\Phi}\phi=False\right) \tag{2.10}$$

定理 2.1 给定 C 程序模型 $CP=\langle S,s_0,T,\Delta\rangle$ 中的一条执行路径 $p=t_1t_2\cdots t_n$，对任意抽象准则 V，设 $\langle\Omega,\Phi\rangle=SP(p)(True)$ 以及 $\langle\widetilde{\Omega},\widetilde{\Phi}\rangle=\widetilde{SP}_V(p)(True)$，那么公式 $h(\langle\Omega,\Phi\rangle)\Rightarrow h(\langle\widetilde{\Omega},\widetilde{\Phi}\rangle)$ 成立。

证明：首先，如果执行路径 p 中不包含任何迁移，则 $\langle\Omega,\Phi\rangle=\langle\widetilde{\Omega},\widetilde{\Phi}\rangle=\langle\Omega_0,\Phi_0\rangle$，其中 $\Omega_0=\Phi_0=\varnothing$，因此定理成立。接下来根据归纳法，我们假设公式 $h(\langle\Omega_i,\Phi_i\rangle)\Rightarrow h(\langle\widetilde{\Omega}_i,\widetilde{\Phi}_i\rangle)$ 成立，其中 $\langle\Omega_i,\Phi_i\rangle=SP(t_1\cdots t_i)(True)$ 以及 $\langle\widetilde{\Omega}_i,\widetilde{\Phi}_i\rangle=\widetilde{SP}_V(t_1\cdots t_i)(True)$，下面我们证明公式 $h(\langle\Omega_{i+1},\Phi_{i+1}\rangle)\Rightarrow h(\langle\widetilde{\Omega}_{i+1},\widetilde{\Phi}_{i+1}\rangle)$ 成立，为此我们对迁移 t_{i+1} 的下列情况进行讨论：

- 如果 t_{i+1} 是无关的赋值语句或无关的 assume 语句，那么我们有

$\langle\Omega_{i+1},\Phi_{i+1}\rangle = SP(t_{i+1})(\langle\Omega_i,\Phi_i\rangle)$ 以及 $\langle\widetilde{\Omega}_{i+1},\widetilde{\Phi}_{i+1}\rangle = \langle\widetilde{\Omega}_i,\widetilde{\Phi}_i\rangle$ 成立。根据变量抽象的定义，无关迁移 t_{i+1} 包含的所有变量都不在抽象准则 V 中，因此对集合 V 中的变量而言，$\langle\Omega_{i+1},\Phi_{i+1}\rangle$ 与 $\langle\Omega_i,\Phi_i\rangle$ 描述了完全相同的条件，从而我们有 $h(\langle\Omega_{i+1},\Phi_{i+1}\rangle)\Rightarrow h(\langle\widetilde{\Omega}_{i+1},\widetilde{\Phi}_{i+1}\rangle)$ 成立；

- 如果 t_{i+1} 是完全相关赋值语句或相关 assume 语句，那么根据定义有 $\langle\Omega_{i+1},\Phi_{i+1}\rangle = SP(t_{i+1})(\langle\Omega_i,\Phi_i\rangle)$ 以及 $\langle\widetilde{\Omega}_{i+1},\widetilde{\Phi}_{i+1}\rangle = SP(t_{i+1})(\langle\widetilde{\Omega}_i,\widetilde{\Phi}_i\rangle)$ 成立。根据假设 $h(\langle\Omega_i,\Phi_i\rangle)\Rightarrow h(\langle\widetilde{\Omega}_i,\widetilde{\Phi}_i\rangle)$，再根据单调性引理 2.1 得到 $h(\langle\Omega_{i+1},\Phi_{i+1}\rangle)\Rightarrow h(\langle\widetilde{\Omega}_{i+1},\widetilde{\Phi}_{i+1}\rangle)$ 成立；
- 如果 t_{i+1} 是部分相关赋值语句 $x:=e$，那么根据定义有 $\Omega_{i+1}=\Omega_i[x\rightarrow\Omega_i(e)]$、$\Phi_{i+1}=\Phi_i$ 以及 $\widetilde{\Omega}_{i+1}=\widetilde{\Omega}_i[x\rightarrow\theta_e]$、$\widetilde{\Phi}_{i+1}=\widetilde{\Phi}_i$。对于任意一组对集合 V 中变量的赋值而言，如果该组赋值使得一阶逻辑公式 $h(\langle\Omega_{i+1},\Phi_{i+1}\rangle)$ 为真，那么通过取 Skolem 常量 θ_e 的值等于 $\Omega_i(e)$，我们能够保证该组赋值也使得一阶逻辑公式 $h(\langle\widetilde{\Omega}_{i+1},\widetilde{\Phi}_{i+1}\rangle)$ 为真。从而，公式 $h(\langle\Omega_{i+1},\Phi_{i+1}\rangle)\Rightarrow h(\langle\widetilde{\Omega}_{i+1},\widetilde{\Phi}_{i+1}\rangle)$ 成立。

综上，根据归纳法，定理得证。 ■

定理 2.1 告诉我们：如果 $\widetilde{SP}_V(p)(True)=False$，那么 $SP(p)(True)=False$。也就是说，如果某条执行路径 p 通过计算其部分最强后置条件被判定为不可行，那么它在程序实际执行过程中就一定是一条不可行路径。

2.3 切片执行

2.3.1 切片执行上下文

定义 2.6 对于 C 程序模型 $CP=\langle S,s_0,T,\Delta\rangle$，给定一个抽象准则 V 和一个程序位置 $s\in S$，设 P 是从初始程序位置 s_0 到程序位置 s 的一组执行路径的集合，则程序位置 s 对应于执行路径集合 P 的切片执行上下文，记作 $SEC_P(s)$，定义如下：

$$SEC_P(s)\triangleq\left\{\langle\Omega_p,\Phi_p\rangle=\widetilde{SP}_V(p)(True)\,\middle|\,p\in P\right\} \tag{2.11}$$

注意，切片执行上下文的定义并不排斥执行路径集合 P 中包含无穷多条路径的情况。另外，如果我们不关心执行路径集合 P，切片执行上下文 $SEC_P(s)$ 也可以简记为 $SEC(s)$。

定义 2.7 部分最强后置条件 $\langle\Omega,\Phi\rangle$ 蕴含（Imply）切片执行上下文 $SEC(s)$，记为 $\langle\Omega,\Phi\rangle\Rightarrow SEC(s)$，如果下列公式(2.12)成立：

$$h(\langle\Omega,\Phi\rangle)\Rightarrow\bigvee_{\langle\Omega',\Phi'\rangle\in SEC(s)}h(\langle\Omega',\Phi'\rangle) \tag{2.12}$$

$\langle\Omega,\Phi\rangle \Rightarrow SEC(s)$ 表明：$SEC(s)$描述了比部分最强后置条件 $\langle\Omega,\Phi\rangle$ 更为一般的变量取值约束条件。换言之，所有满足 $\langle\Omega,\Phi\rangle$ 的对抽象准则 V 中程序变量的赋值，都必满足 $SEC(s)$指定的变量取值约束条件。

如果 $\langle\Omega,\Phi\rangle \Rightarrow SEC(s)$，我们定义函数 $ImplySEC(\langle\Omega,\Phi\rangle)(SEC(s))$ 返回 $SEC(s)$ 中与 $\langle\Omega,\Phi\rangle$ “相交”的部分最强后置条件的集合。形式化地，我们定义：

$$ImplySEC(\langle\Omega,\Phi\rangle)(SEC(s)) \triangleq \{\langle\Omega',\Phi'\rangle \in SEC(s) \mid h(\langle\Omega,\Phi\rangle) \wedge h(\langle\Omega',\Phi'\rangle) \neq False\} \quad (2.13)$$

显然，我们有 $\langle\Omega,\Phi\rangle \Rightarrow ImplySEC(\langle\Omega,\Phi\rangle)(SEC(s))$ 成立，因此我们可以使用 $ImplySEC(\langle\Omega,\Phi\rangle)(SEC(s))$ 所描述的约束条件替代 $\langle\Omega,\Phi\rangle$ 所描述的约束条件。特殊地，如果 $h(\langle\Omega,\Phi\rangle) = False$，则我们有 $ImplySEC(\langle\Omega,\Phi\rangle)(SEC(s)) = \varnothing$。

2.3.2 切片执行图

众所周知，对程序进行符号执行能够得到一个所谓的符号执行图[53, 67]。切片执行是一种轻量级的符号执行，因此它也能产生一种抽象的符号执行图，我们称其为切片执行图。

```
SlicingExecution(CP = ⟨S, s0, T, Δ⟩)
{
1  WorkSet : WorkSet ⊆ S × [[PSP]];
2  SecMap : S ↦ 2^[PSP];
3  for each s ∈ S let SecMap(s) := ∅;
4  let WorkSet := {⟨s0, ⟨Ω0, Φ0⟩⟩} and SecMap(s0) := {⟨Ω0, Φ0⟩};
5  while WorkSet ≠ ∅ do {
6      remove an element ⟨s, ⟨Ω, Φ⟩⟩ from WorkSet;
7      add ⟨s, ⟨Ω, Φ⟩⟩ to Ψ;
8      for all t ∈ T such that ∃s'.⟨s, t, s'⟩ ∈ Δ do {
9          let ⟨Ω', Φ'⟩ := SP̃_V(t)(⟨Ω, Φ⟩);
10         if (⟨Ω', Φ'⟩ ⇒ SecMap(s')){
11             for all ⟨Ω'', Φ''⟩ ∈ ImplySEC(⟨Ω', Φ'⟩)(SecMap(s')) do
12                 set ⟨s, ⟨Ω, Φ⟩⟩ --t--> ⟨s', ⟨Ω'', Φ''⟩⟩;
13         }else{
14             set ⟨s, ⟨Ω, Φ⟩⟩ --t--> ⟨s', ⟨Ω', Φ'⟩⟩;
15             insert ⟨Ω', Φ'⟩ into SecMap(s');
16             insert ⟨s', ⟨Ω', Φ'⟩⟩ into WorkSet;
17         }
18     }
19 }
}
```

图 2.2 产生切片执行图的切片执行过程

给定 C 程序模型 $CP=\langle S,s_0,T,\Delta\rangle$，对其进行切片执行产生的切片执行图表示为 $SEG=\langle\Psi,\longrightarrow\rangle$，其中：$\Psi\subseteq S\times[\![PSP]\!]$ 是切片执行图的状态集合（$[\![PSP]\!]$ 是前面定义的部分最强后置条件的全集），也就是说切片执行图的状态是由程序位置和该位置的一个部分最强后置条件组合而成的；$\longrightarrow\subseteq\Psi\times T\times\Psi$ 是迁移集合，对于两个状态 $\psi,\psi'\in\Psi$，我们将 $\langle\psi,t,\psi'\rangle\in\longrightarrow$ 简记为 $\psi\xrightarrow{t}\psi'$。

C 程序模型 $CP=\langle S,s_0,T,\Delta\rangle$ 的切片执行图 $SEG=\langle\Psi,\longrightarrow\rangle$ 可以由图 2.2 所示的过程产生，该过程是一个不动点计算过程。过程的第 1、2 两行定义了两个局部变量：$WorkSet\subseteq\Psi$ 存储切片执行图 $SEG=\langle\Psi,\longrightarrow\rangle$ 中暂时未被处理的状态；$SecMap$ 则存储程序的每个位置 $s\in S$ 当前所对应的切片执行上下文。初始时，我们在过程第 4 行置 $SEC(s_0):=\{\langle\Omega_0,\Phi_0\rangle\}$，其中 $\Omega_0=\Phi_0=\varnothing$。换言之，初始程序位置 s_0 的部分最强后置条件被置为 $True$。而对其它程序位置，我们置 $SecMap(s):=\varnothing$，即置所有程序位置的切片执行上下文为 $False$。在过程第 7 行，我们将当前正在处理的程序位置及其部分最强后置条件的组合 $\langle s,\langle\Omega,\Phi\rangle\rangle$ 加入到切片执行图的状态集合 Ψ 中。过程第 8 到 16 行则产生状态 $\langle s,\langle\Omega,\Phi\rangle\rangle\in\Psi$ 的所有后继状态和相应的迁移关系。

在切片执行过程第 10 行，如果部分最强后置条件 $\langle\Omega',\Phi'\rangle$ 蕴含切片执行上下文 $SecMap(s')$，那么我们设置状态 $\langle s,\langle\Omega,\Phi\rangle\rangle$ 通过迁移 t 迁移到集合 $ImplySEC(\langle\Omega',\Phi'\rangle)(SecMap(s'))$ 中的每个状态 $\langle s',\langle\Omega'',\Phi''\rangle\rangle$，即设置迁移 $\langle s,\langle\Omega,\Phi\rangle\rangle\xrightarrow{t}\langle s',\langle\Omega'',\Phi''\rangle\rangle$。特殊情况下，如果 $h(\langle\Omega',\Phi'\rangle)=False$，那么蕴含关系 $\langle\Omega',\Phi'\rangle\Rightarrow SecMap(s')$ 仍然成立，但由于 $ImplySEC(\langle\Omega',\Phi'\rangle)(SecMap(s'))=\varnothing$，我们不会设置任何迁移关系。另一方面，如果 $\langle\Omega',\Phi'\rangle\not\Rightarrow SecMap(s')$，那么我们将迁移关系 $\langle s,\langle\Omega,\Phi\rangle\rangle\xrightarrow{t}\langle s',\langle\Omega',\Phi'\rangle\rangle$ 加入到切片执行图中。同时，新产生的状态 $\langle s',\langle\Omega',\Phi'\rangle\rangle$ 将在下次迭代的第 7 行被加入到切片执行图的状态集合 Ψ 中。

当 $WorkSet$ 为空时切片执行过程将终止，同时我们将得到完整的切片执行图。但是一般来说，由于程序终止性是不可判定的，故切片执行是否终止也是不可判定的。为了保证切片执行过程的终止，我们可以借鉴文献[10, 47, 48]的方法，引入对循环进行切片执行的上界次数来确保终止性。

定理 2.2 对于 C 程序模型 $CP=\langle S,s_0,T,\Delta\rangle$ 中的一条执行路径 $p=t_1t_2\cdots t_n$，如果 $\widetilde{SP}_V(p)(True)\neq False$，那么图 2.2 所示的切片执行过程所产生的切片执行图 $SEG=\langle\Psi,\longrightarrow\rangle$ 中存在 $n+1$ 个状态 $\psi_i=\langle s_i,\langle\Omega_i,\Phi_i\rangle\rangle\in\Psi$（其中 $0\le i\le n$），使得 $\langle s_{i-1},t_i,s_i\rangle\in\Delta$（其中 $0\le i\le n$）并且 $\psi_0\xrightarrow{t_1}\psi_1\xrightarrow{t_2}\cdots\xrightarrow{t_n}\psi_n$。

证明：基于反证法，我们假设所产生的切片执行图 $SEG=\langle\Psi,\longrightarrow\rangle$ 中并不包含所给路径 $\psi_0\xrightarrow{t_1}\psi_1\xrightarrow{t_2}\cdots\xrightarrow{t_n}\psi_n$，那么对于包含在 $SEG=\langle\Psi,\longrightarrow\rangle$ 中的该

路径的最长子路径$\psi_0 \xrightarrow{t_1} \cdots \xrightarrow{t_k} \psi_k$（其中$k<n$），我们设$\psi_k=\langle s_k,\langle \Omega_k,\Phi_k\rangle\rangle$。根据图 2.2 的切片执行过程，我们可以得出$\widetilde{SP}_V(t_{k+1})(\langle \Omega_k,\Phi_k\rangle)=False$，否则，必定存在某个状态$\psi'_{k+1}\in\Psi$，使得$\psi_k \xrightarrow{t_{k+1}} \psi'_{k+1}$。此外，由于不可行路径后面扩充任意多个迁移得到的仍是不可行路径，因此$\widetilde{SP}_V(t_{k+1}\cdots t_n)(\langle \Omega_k,\Phi_k\rangle)=False$。考虑切片执行图中存在的迁移$\psi_{k-1} \xrightarrow{t_k} \psi_k$，其中$\psi_{k-1}=\langle s_{k-1},\langle \Omega_{k-1},\Phi_{k-1}\rangle\rangle$，如果$\widetilde{SP}_V(t_k)(\langle \Omega_{k-1},\Phi_{k-1}\rangle)=\langle \Omega_k,\Phi_k\rangle$（对应于切片执行第 10 行蕴含条件不满足的情况，即$\langle \Omega_k,\Phi_k\rangle \not\Rightarrow SecMap(s_k)$），那么我们得到$\widetilde{SP}_V(t_kt_{k+1}\cdots t_n)(\langle \Omega_{k-1},\Phi_{k-1}\rangle)=False$。如果$\widetilde{SP}_V(t_k)(\langle \Omega_{k-1},\Phi_{k-1}\rangle)\neq\langle \Omega_k,\Phi_k\rangle$，设$\langle \Omega'_k,\Phi'_k\rangle=\widetilde{SP}_V(t_k)(\langle \Omega_{k-1},\Phi_{k-1}\rangle)$，那么我们有$\langle \Omega'_k,\Phi'_k\rangle\Rightarrow SecMap(s_k)$以及$\langle \Omega_k,\Phi_k\rangle\in ImplySEC(\langle \Omega'_k,\Phi'_k\rangle)(SecMap(s_k))$，则下面我们证明公式$\widetilde{SP}_V(t_kt_{k+1}\cdots t_n)(\langle \Omega_{k-1},\Phi_{k-1}\rangle)=False$仍然成立。

不失一般性，我们假设集合$ImplySEC(\langle \Omega'_k,\Phi'_k\rangle)(SecMap(s_k))$中除$\langle \Omega_k,\Phi_k\rangle$外的其它部分最强后置条件$\langle \Omega''_k,\Phi''_k\rangle$都满足$\widetilde{SP}_V(t_{k+1}\cdots t_n)(\langle \Omega''_k,\Phi''_k\rangle)=False$，否则切片执行图中就会存在路径$\psi''_k \xrightarrow{t_{k+1}} \cdots \xrightarrow{t_n} \psi''_n$（其中$\psi''_k=\langle s_k,\langle \Omega''_k,\Phi''_k\rangle\rangle$）。由于$\langle \Omega'_k,\Phi'_k\rangle\Rightarrow SecMap(s_k)$，因此公式$\widetilde{SP}_V(t_{k+1}\cdots t_n)(\langle \Omega'_k,\Phi'_k\rangle)=False$成立，故$\widetilde{SP}_V(t_kt_{k+1}\cdots t_n)(\langle \Omega_{k-1},\Phi_{k-1}\rangle)=False$。

重复进行上述过程，我们最终得到$\widetilde{SP}_V(t_1t_2\cdots t_n)(\langle \Omega_0,\Phi_0\rangle)=False$，这与定理的前提条件$\widetilde{SP}_V(p)(True)\neq False$矛盾，因此在切片执行图$SEG=\langle \Psi,\longrightarrow\rangle$中必定存在迁移路径$\psi_0 \xrightarrow{t_1} \psi_1 \xrightarrow{t_2} \cdots \xrightarrow{t_n} \psi_n$。 ■

对于 C 程序中的每条执行路径 p，如果$SP(p)(True)\neq False$，那么根据定理 2.1 知$\widetilde{SP}_V(p)(True)\neq False$，又根据定理 2.2 知所产生的切片执行图中一定包含一条对应于路径 p 的迁移序列。也就是说，切片执行图中一定包含对应程序中所有可行的执行路径。

2.3.3 讨论

一般来说，构建切片执行图的代价要远小于构建符号执行图的代价，这主要是因为现实程序中存在着由分支语句的组合所导致的大量（指数级、甚至无穷多）的执行路径，而在切片执行中，由于我们仅考虑了程序中相当少一部分程序变量，因此大量的分支路径都对这部分程序变量进行了相同或相似的操作，从而我们可以将这些路径看作一条等价路径而不用进行多次的切片执行，所以切片执行的代价要小于符号执行的代价。实际上，对某个程序位置而言，如果某条到达该位置的执行路径的部分最强后置条件蕴含该位置当前的切片执行上下文，我们就不需要继续往下切片执行该路径。特殊地，如果变量抽象准则的变量集合为空，则所有执行路径的部分最强后置条件都为 *True*，此时的切片执行实际上就是标准的流

敏感数据流分析。由于切片执行使用反例制导的抽象精化方法、仅考虑与待验证的性质相关的程序变量，而待验证的性质往往只涉及到少部分程序变量，因此切片执行的代价接近于流敏感的数据流分析的代价，并且抽象准则中的变量数量越少，其代价越接近于流敏感的数据流分析；另一方面，切片执行的精度却能够达到与路径敏感的符号执行完全相同，这是切片执行方法的最大特点。

此外，基于切片执行的模型抽象（即切片执行图的提取）适用于任意结构的程序，包括执行永不终止的程序，例如操作系统服务程序等。让我们来考查一般的 while 循环形式 "*while(c) B;*"，其中 c 为循环条件，B 为循环体，循环体中可能嵌套多个循环。设 $\langle s,\langle\Omega,\Phi\rangle\rangle$ 为切片执行刚好执行到 while 语句对应的程序点之前的状态，设 $\langle\Omega_i,\Phi_i\rangle$ 为切片执行循环体 i 次后 $\langle\Omega,\Phi\rangle$ 所对应的部分最强后置条件，即 $\langle\Omega_i,\Phi_i\rangle=\widetilde{SP}_V(B_1\cdots B_i)(\langle\Omega,\Phi\rangle)$，其中对任意 $j:1\le j\le i$ 都有 $B_j=B$。根据图 2.2 所示的切片执行过程，如果切片执行循环体 $i+1$ 次后的部分最强后置条件 $\langle\Omega_{i+1},\Phi_{i+1}\rangle$ 满足 $h(\langle\Omega_{i+1},\Phi_{i+1}\rangle)\Rightarrow h(\langle\Omega,\Phi\rangle)\vee h(\langle\Omega_1,\Phi_1\rangle)\vee\cdots\vee h(\langle\Omega_i,\Phi_i\rangle)$，则对循环体 B 的切片执行就会终止（因为再切片执行下去也不能得到更多可能的变量取值），注意这个条件是在图 2.2 第 10 行检查条件 $\langle\Omega_{i+1},\Phi_{i+1}\rangle\Rightarrow SEC(s)$ 是否满足时隐式检查的。如果循环体 B 中包含有嵌套的子循环，则每次切片执行循环体 B 时都要多次切片执行嵌套子循环，直到其满足上述终止条件。另外，如果上述 while 循环永远不终止（例如循环条件 c 为 *True*），切片执行也往往能够终止，因为上述条件的检查不依赖于循环是否终止。基于网络服务程序 *openssl-0.9.6c* 实用程序（它永不终止）的实验显示，切片执行往往不需要引入循环执行的上限执行次数也能保证终止性。

检查部分最强后置条件 $\langle\Omega,\Phi\rangle$ 是否蕴含切片执行上下文 $SEC(s)$（即公式(2.12)）是切片执行方法的难点。但是，我们注意到在实际的切片执行过程中，部分最强后置条件 $\langle\Omega,\Phi\rangle$ 以及切片执行上下文 $SEC(s)$ 中的大多数部分最强后置条件都是互不相交的，这是因为程序位置 s 处的每个部分最强后置条件都代表了一条到达 s 的执行路径，而不同的执行路径由于选择了某个分支语句的不同分支，使得它们对应的部分最强后置条件往往包含了互斥的分支条件。基于这个事实，对于切片执行上下文 $SEC(s)$ 中的每个部分最强后置条件 $\langle\Omega',\Phi'\rangle$，我们首先检查公式 $h(\langle\Omega,\Phi\rangle)\wedge h(\langle\Omega',\Phi'\rangle)\neq False$ 是否成立，即检查两个部分最强后置条件是否相交，该公式等价于如下的命题逻辑公式(2.14)：

$$\left(\bigwedge_{\omega\in\Omega}e2b(\omega)\right)\wedge\left(\bigwedge_{\phi\in\Phi}\phi\right)\wedge\left(\bigwedge_{\omega'\in\Omega'}e2b(\omega')\right)\wedge\left(\bigwedge_{\phi'\in\Phi'}\phi'\right)\neq False \tag{2.14}$$

如果公式(2.14)不满足，那么我们完全可以忽略 $\langle\Omega',\Phi'\rangle$。通常，切片执行上下文中与当前考虑的部分最强后置条件相交的部分最强后置条件的数量很少，因此，

最终需要判定的一阶逻辑公式(2.12)是比较简单的。

2.4 时序安全性质的验证

2.4.1 对切片执行图的模型检验

对程序进行切片执行所生成的切片执行图实际上就是程序的有限状态模型，它与待验证的时序安全性质一起，可以输入通用的模型检验工具进行验证，下面的定理 2.3 保证了验证的可行性。

定理 2.3 如果模型检验证明了某时序安全性质被某切片执行图满足，则该性质也一定被该切片执行图对应的程序所满足。

证明：基于反证法，假设程序中存在一条违背所给时序安全性质的执行路径 p，由于 p 是程序的可行路径，因此 $SP(p)(True) \neq False$，从而根据定理 2.1 和定理 2.2，该程序对应的切片执行图中必定包含一条对应于 p 的迁移序列，因此模型检验不可能证明该性质在切片执行图中被满足，从而导致了矛盾。 ■

另一方面，如果模型检验工具在切片执行图中找到了一条违反所给时序安全性质的反例路径 p，我们却不能断定切片执行图对应的程序一定违反所给性质。这是因为对执行路径 p，我们有公式 $\widetilde{SP}_V(p)(True) \neq False$ 成立，但根据定理 2.1，公式 $SP(p)(True) = False$ 仍然可能成立。也就是说，切片执行图中的可行执行路径在程序中并不一定可行。为了检查 p 是否可行，我们计算其最强后置条件 $\langle \Omega, \Phi \rangle = SP(p)(True)$，并根据公式(2.6)判定 $\langle \Omega, \Phi \rangle$ 是否为 $False$。除此之外，我们也可以基于文献[68]和文献[69]中提出的方法判定路径的可行性，并能够生成可行路径的一组输入数据。

如果对切片执行图进行模型检验找到的反例路径 p 是可行的，我们就报告性质被违背，并给出 p 作为一条违背性质的反例路径；否则，我们需要基于伪反例路径 p 对变量抽象的抽象准则进行精化，并保证基于精化的抽象准则生成的新切片执行图中不再包含路径 p，以保证整个验证过程的可终止性。基于伪反例路径的抽象准则精化在 5.3 小节介绍，其基本思想是推断出新的变量加入到抽象准则中。

2.4.2 切片执行与模型检验的集成

实际上，我们并不需要一个额外的通用模型检验工具对切片执行生成的切片执行图进行模型检验，取而代之的是，我们可以在切片执行过程中同时进行对时序安全性质的模型检验。集成切片执行与模型检验的好处包括：(1)对于验证过程的前几次迭代，由于与性质相关的程序变量还未被加入抽象准则，因此往往存在

伪反例路径，集成的模型检验发现一条伪反例路径后，就可以马上进行抽象精化，而无须生成整个程序的完整切片执行图，从而可以大大节约验证时间；(2)如果程序不满足给定的性质，那么集成的模型检验可以尽早地找到反例路径而终止切片执行，相反如果程序满足给定的性质，集成的模型检验也可以帮助切片执行丢弃切片执行图中不可能违背性质的大量状态，而只保存当前正在考查的极少数状态，从而大大节省存储切片执行图所需的存储空间，进而提高验证的可扩展性。

为了集成模型检验，我们将切片执行图 $SEG=\langle \Psi, \longrightarrow \rangle$ 的状态空间延拓为 $\Psi \subseteq S \times [\![PSP]\!] \times PS$，其中 PS 是给定性质自动机的状态空间。相应地，Ψ 的初始状态 ψ_0 也延拓为 $\langle s_0, \langle \Omega_0, \Phi_0 \rangle, \varepsilon_0 \rangle$，其中 ε_0 为性质自动机的初始状态。设状态 $\langle s_i, \langle \Omega_i, \Phi_i \rangle, \varepsilon_i \rangle$ 为状态 $\langle s_{i-1}, \langle \Omega_{i-1}, \Phi_{i-1} \rangle, \varepsilon_{i-1} \rangle$ 经过迁移 t_i 迁移到的后继状态，那么 ε_i 也应该是性质自动机中 ε_{i-1} 经过迁移 t_i 迁移到的后继状态，如果 t_i 也是性质自动机的状态迁移事件。注意，不论迁移 t_i 是相关还是无关语句，只要它是性质自动机的状态迁移事件，则当前状态中的性质自动机状态就要被更新。如果在切片执行过程中发现某状态中的性质自动机状态分量到达了性质自动机的接收态，则根据性质自动机的定义我们知道性质被违背；如果没有发现这样的状态，则性质被满足。

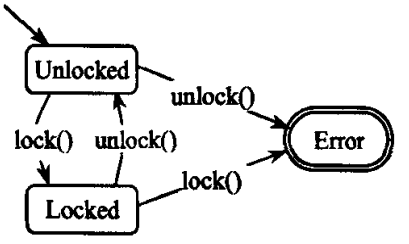

图 2.3 描述加/解锁用法的性质自动机示例

下面我们举一个验证的例子，首先我们给出一个描述加/解锁用法的性质自动机，如图 2.3 所示，它的初始状态是 *Unlocked*，接收状态是 *Error*，描述的时序安全性质是加锁和解锁必须交替进行。图 2.4 则给出了对一段代码基于切片准则 $V=\{old, new\}$ 的切片执行，用于对加/解锁性质进行验证。为简单起见，我们在每个程序位置列出了三个元素 $\langle \Omega, \Phi, \varepsilon \rangle$，其中 $\langle \Omega, \Phi \rangle$ 是当前执行路径所对应的部分最强后置条件，ε 是当前性质自动机的状态。初始状态为 $\langle \varnothing, \varnothing, \$U \rangle$，其中 $\$U$ 是性质自动机的初始状态 *Unlocked*。当切片执行语句“*old:=new*”时，我们为变量“*new*”引入一个 Skolem 常量，从而到达状态 $\langle \{old \to \theta, new \to \theta\}, \varnothing, \$U \rangle$。由于语句“*lock()*”是性质自动机的状态迁移事件，因此切片执行后的状态为 $\langle \{old \to \theta, new \to \theta\}, \varnothing, \$L \rangle$。对分支语句“*old==new*”之前的程序位置而言，有两条执行路径到达该程序位置，它们对应于切片执行图中的状态分别是

$\langle\{old\to\theta, new\to\theta\},\varnothing,\$L\rangle$ 和 $\langle\{old\to\theta, new\to\theta+1\},\varnothing,\$U\rangle$，基于后者切片执行完分支语句后的状态为 $\langle\{old\to\theta, new\to\theta+1\},\{\theta=\theta+1\},\$U\rangle$，$\theta=\theta+1$ 为 *False* 导致其部分最强后置条件也为 *False*，因此切片执行图中经该分支语句迁移到的后继状态只有一个，即 $\langle\{old\to\theta, new\to\theta\},\{\theta=\theta\},\$L\rangle$，它对应性质自动机的状态为 $\$L$，因此切片执行后继的 "*unlock*()" 语句时不会违背性质。

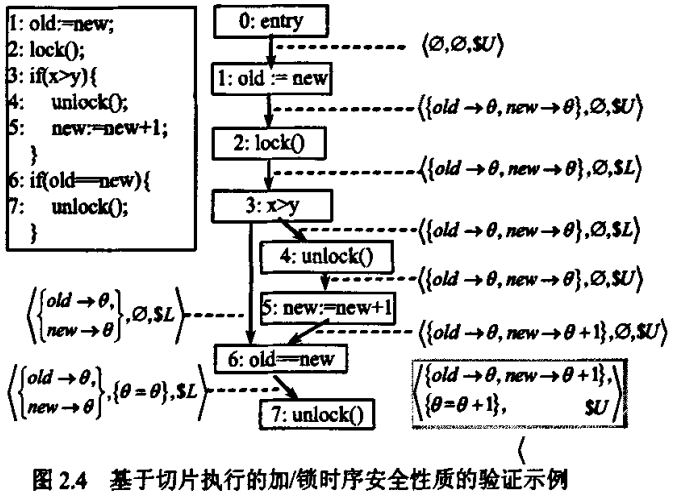

图 2.4 基于切片执行的加/锁时序安全性质的验证示例

对实际的程序验证来说，我们往往需要跟踪具有某种特征的多个程序变量，例如多个锁变量等。我们需要为每个程序变量都赋一个状态变量，用于记录其对应于性质自动机的状态。因此，我们将切片执行图 $SEG=\langle\Psi,\longrightarrow\rangle$ 的状态进一步延拓到 $\Psi=S\times[\![PSP]\!]\times[\![\Sigma]\!]$，其中 Σ 为一个部分函数，它将给定特征的程序变量映射到性质自动机的一个状态，此外我们用 $[\![\Sigma]\!]$ 表示这种部分函数的全集。例如，如果程序包含了两个锁变量 *lock*1 和 *lock*2，那么我们在 Σ 中引入两个映射 $lock1\to\$S_1$ 以及 $lock2\to\$S_2$，其中 $\$S_1$ 和 $\$S_2$ 分别记录了两个锁变量对应的性质自动机的状态。在切片执行过程中，如果我们发现 Σ 中的任何一个映射所对应的性质自动机的状态进入了接收状态集，那么我们就报告性质被违反。

2.4.3 抽象准则的精化

正如我们所讨论的，基于切片执行的时序安全性质验证是一个迭代的过程。第一次迭代时变量抽象的抽象准则仅包含了性质自动机中的所有程序变量，其它额外所需的变量将基于 CEGAR 框架[56]，在接下来的迭代中根据模型检验找到的伪反例路径被推断出来，并加入到抽象准则中以对其进行精化。对于一条伪反例路径而言，我们需要推断出一些必要的变量加入抽象准则，使得基于新的抽象准

则进行切片执行时该路径变为不可行。路径不可行表现在其包含的 assume 语句存在冲突，因此变量推断的基本思想是寻找冲突的 assume 语句，并将其作为切片请求进行一次动态切片[70]，以找到这些冲突的 assume 语句依赖的所有其它语句，最后再把所有这些依赖语句中所包含的变量加入到抽象准则即可，其实现在论文[71]中进行了详细的讨论。

除了使用动态切片方法之外，我们还可以在判定反例路径是否可行的同时进行变量的推断，其基本思想是在计算最强后置条件$\langle\Omega,\Phi\rangle$的同时记录下语句之间的依赖关系，如果$\langle\Omega,\Phi\rangle$变为*False*，则我们找出集合$\Phi$中的冲突条件所对应的 assume 语句，并根据记录的依赖关系找到这些冲突的 assume 语句依赖的所有语句即可。

定义 2.8 对一个迁移 t，我们定义两个函数 *DefVars*(t)和 *RefVars*(t)如下：

- 如果 t 是赋值语句 $x:=e$，则 $DefVars(t)\triangleq Vars(x)$、$RefVars(t)\triangleq Vars(e)$；
- 如果 t 是 assume 语句 *assume*(c)，则 $DefVars(t)\triangleq\varnothing$、$RefVars(t)\triangleq Vars(c)$。

直观地，迁移 t 执行时需要引用 *RefVars*(t)中的每个变量的值，同时为 *DefVars*(t)中的每个变量赋新值。如果 $x\in DefVars(t)$，则我们称 t **定义** x；如果 $x\in RefVars(t)$，则我们称 t **引用** x。

定义 2.9 执行路径 $p=t_1t_2\cdots t_n$ 上的依赖关系 $\triangleright_p$ 定义为集合 $\{t_1,t_2,\cdots,t_n\}$ 上满足下列两个条件的**最小**关系：

1. 若 $k<i$ 且 $\exists x\in RefVars(t_i):\left(x\in DefVars(t_k)\wedge\forall k<m<i:x\notin DefVars(t_m)\right)$，则 $t_i\triangleright_p t_k$；
2. $\triangleright_p$ 是满足上述条件 1 的传递闭包。

直观上，定义 2.9 的第一个条件说的是，t_i 引用了 t_k 所定义的某个变量 x，并且 t_k 与 t_i 之间的所有迁移都没有对变量 x 进行重新定义。换言之，t_i 引用的变量 x 的值是 t_k 定义的。

接下来，与基于动态切片的抽象精化方法类似，我们先找出导致冲突的 assume 语句，再找出依赖于这些语句的所有语句，最后将所有依赖语句中的变量加入抽象准则。

设$\langle\Omega,\Phi\rangle$是执行路径 $p=t_1t_2\cdots t_n$ 的最强后置条件 *SP*(p)(*True*)，从 2.1.2 节的公式(2.4)我们知道，路径 p 中的每个 assume 语句 *assume*(c)都在集合Φ中有一个对应条件$\Omega(c)$，而根据公式(2.6)，集合Φ中对应于各 assume 语句的条件的冲突导致了路径的不可行。为了最小化导致冲突的条件从而减少必需的变量数量，我们计算出集合Φ的一个最小的子集 C_{min}，使得 $\wedge_{\phi\in C_{min}}\phi=False$ 也同样成立。设 $\Phi^{-1}(C_{min})$ 为路径 p 中对应于集合 C_{min} 的迁移语句的集合（这些语句必定都是 assume 语句），那么路径 p 中 $\Phi^{-1}(C_{min})$ 依赖的所有迁移语句的集合为：

$$Dep\left(\Phi^{-1}\left(C_{min}\right)\right)\triangleq\left\{t\in p\middle|\exists t_k\in\Phi^{-1}\left(C_{min}\right):t_k\triangleright_p t\right\} \tag{2.15}$$

为了使得基于精化后的变量抽象准则V'切片执行时能够判定执行路径p为不可行路径，即使得$\widetilde{SP}_{V'}(p)(True)=False$，公式(2.15)计算出的语句集合必须被变量抽象判定为（完全）相关语句，因此，新的抽象准则应为公式(2.16)所示的形式，其中V为当前的抽象准则。

$$V'\triangleq V\cup\bigcup_{t\in Dep\left(\Phi^{-1}\left(C_{min}\right)\right)}Vars(t) \tag{2.16}$$

举一个例子，对于一条在抽象准则$V=\{x\}$下的伪反例路径"*assume*(x>0); y:=x; *assume*(z>0); *assume*(y<0);"，该路径的最强后置条件$\langle\Omega,\Phi\rangle=SP(p)(True)$为$\Omega=\{x\to\theta_x,\ y\to\theta_x,\ z\to\theta_z\}$、$\Phi=\{\theta_x>0,\ \theta_z>0,\ \theta_x<0\}$。我们很容易找到集合$\Phi$的最小不可行子集为$C_{min}=\{\theta_x>0,\ \theta_x<0\}$，该子集对应的 assume 语句集合为$\Phi^{-1}\left(C_{min}\right)=\left\{assume(x>0), assume(y<0)\right\}$。最后，我们得到的依赖语句的集合为$Dep\left(\Phi^{-1}\left(C_{min}\right)\right)=\left\{assume(x>0),\ y:=x,\ assume(y<0)\right\}$，因此变量集合$V'=V\cup\{x,y\}$为下次迭代的抽象准则，并且该路径在该抽象准则下的部分最强后置条件为*False*。

定理 2.4 设执行路径p为基于抽象准则V切片执行时的一条伪反例路径，设V'是公式(2.16)所定义的精化的抽象准则，那么路径p在抽象准则V'下的部分最强后置条件为*False*，即$\widetilde{SP}_{V'}(p)(True)=False$。

证明：设$\langle\Omega,\Phi\rangle=SP(p)(True)$、$\left\langle\widetilde{\Omega},\widetilde{\Phi}\right\rangle=\widetilde{SP}_{V'}(p)(True)$，我们来证明对每条 assume 语句$t\in\Phi^{-1}\left(C_{min}\right)$都有$\Phi(t)=\widetilde{\Phi}(t)$成立，其中$\Phi(t)$和$\widetilde{\Phi}(t)$分别代表 assume 语句$t$在集合$\Phi$与$\widetilde{\Phi}$中的对应条件。如果能够证明，则由于$\wedge_{\phi\in C_{min}}\phi=False$以及$C_{min}\subseteq\Phi$，我们得出$\wedge_{\phi\in\widetilde{\Phi}}\phi=False$成立，从而$\widetilde{SP}_{V'}(p)(True)=False$成立。

设$p=t_1t_2\cdots t_n$，则对每条 assume 语句$t_k\in\Phi^{-1}\left(C_{min}\right)$，设$t_k$为*assume*($c$)并设$\langle\Omega_{k-1},\Phi_{k-1}\rangle=SP\left(t_1\cdots t_{k-1}\right)(True)$、$\left\langle\widetilde{\Omega}_{k-1},\widetilde{\Phi}_{k-1}\right\rangle=\widetilde{SP}_{V'}\left(t_1\cdots t_{k-1}\right)(True)$。由于$t_k$依赖的所有迁移语句集合$\left\{t\middle|t_k\triangleright_p t\right\}$在新的抽象准则下都被判定为（完全）相关语句，因此对于每个变量$x\in Vars(c)$都有$\Omega_{k-1}(x)=\widetilde{\Omega}_{k-1}(x)$，所以有$\Omega_{k-1}(c)=\widetilde{\Omega}_{k-1}(c)$。由于$t_k$也是完全相关的，并且$\Phi(t_k)=\Omega_{k-1}(c)$、$\widetilde{\Phi}(t_k)=\widetilde{\Omega}_{k-1}(c)$（参见公式(2.4)），所以$\Phi(t)=\widetilde{\Phi}(t)$成立，故定理得证。 ■

定理 2.4 指出，对于一条伪反例路径p，当基于精化后的抽象准则V'对其进行切片执行时，我们将发现p不再是可行路径，因而也不会再被报告为反例路径。

由于所有可能的反例路径都进行了可行性检查，因此验证工具最终报告的反例路径一定是违反了给定的时序安全性质、并在程序实际执行时存在的真实反例

路径。也就是说，本文提出的基于切片执行的验证方法是可靠的。

2.5 验证工具与实验结果

2.5.1 验证工具

我们基于MAGIC[48]开源项目实现了切片执行工具，工具使用Simplify[72]作为定理证明工具以对切片执行产生的一阶逻辑公式进行判定，工具的结构与实现细节我们将在后续章节中讨论，这里我们重点讨论切片执行对指针和变量别名的处理。

为了支持指针和变量别名，我们首先基于Das的流不敏感指向分析算法[73]得到每个变量 x 的保守可能别名集合 $MayAlias(x)$，该集合确保包含了变量 x 的所有可能的别名。当变量抽象判定一条语句例如 $assume(c)$是否相关时，我们判定变量集合 $Vars(MayAlias(c))$是否为抽象准则 V 的子集，而不是基于变量集合 $Vars(c)$进行判定。例如，设抽象准则 $V=\{x\}$，设 $MayAlias(*p)=\{*p,x\}$，则 assume 语句 $assume(*p>0)$ 将被判定为相关 assume 语句（因为 $*p$ 的一个可能别名 x 属于 V）。由于Das的流不敏感指向分析算法是不精确的，因此保守可能别名集合 $MayAlias(x)$ 中可能包含了一些并非 x 别名的变量，从而可能造成某些本应无关的语句被判定为相关，这会增加切片执行的代价，但不会影响切片执行的可靠性和正确性。指针和变量别名的一种特殊情况是数组，由于 C 语言中数组名和指针是通用的，而静态分析往往无法精确地确定数组索引的取值，因此很难判定到底引用了哪个数组元素。我们采用了一种简单但保守的方法，即只看数组名而忽略索引，例如对 assume 语句 $assume(a[i]>0)$，只要其数组名 a 或其可能的别名、或者它的某个数组元素的别名属于抽象准则 V，我们就判定其为相关变量，而不管 i 取什么值。

当指针和变量别名存在时，部分最强后置条件的计算也需要进行修改。我们引用**变量位置**的概念，一个变量位置是指一个变量、一个结构的某个域或者某个变量位置所指向的变量位置。我们在计算部分最强后置条件时，为每个变量位置都引入一个 Skolem 常量。例如，变量 $*p$ 涉及了两个变量位置，一个是 p，另一个是 p 所指向的变量位置 $*p$。如果 $*p$ 及其某个可能的别名属于抽象准则 V，则我们有 $\widetilde{SP}_V\left(assume(*p>0)\right)\left(\langle\Omega,\Phi\rangle\right)=\left\langle\Omega[p\to\theta_p][*\theta_p\to\theta_{*_p}],\Phi\cup\{\theta_{*_p}>0\}\right\rangle$，其中 θ_p 是为变量位置 p 引入的 Skolem 常量，而 θ_{*_p} 则是为变量位置 $*\theta_p$ 引入的 Skolem 常量。

除此之外，当计算赋值语句 $*q:=e$ 的部分最强后置条件时，我们假设 $*q$ 的某个别名属于抽象准则，但变量 q 不在抽象准则中。此时如果 Ω 中的某个变量位置 $*\theta_p$ 可能是变量位置 $*q$ 的别名，但由于 $q\notin V$ 从而无法确定 θ_p 是否等于 q 的值，那么 Ω 将被更新为 $\Omega[*\theta_p\to\theta_{new}][q\to\theta_q][*\theta_q\to\Omega(e)]$。也就是说，我们首先为变量位置

$*\theta_p$ 引入一个新的 Skolem 常量 θ_{new}，并设定 $*\theta_p \rightarrow \theta_{new}$，再处理赋值语句 $*q := e$。这是因为，由于 $*\theta_p$ 可能为变量 $*q$ 的别名，因此执行赋值语句后变量 $*\theta_p$ 的值不再能够确定，所以我们引入 Skolem 常量 θ_{new} 表示 $*\theta_p$ 可以取值为任意值。这种做法保证了部分最强后置条件的一致性和正确性，但同时也放大了变量的取值范围，从而可能导致出现伪反例路径。所以，当从伪反例路径推断新的变量时，我们需要检查该伪反例路径是否因此而产生，如果是的话，我们需要将变量 $*p$ 和 $*q$ 包含的所有变量位置（即 p、q、$*p$ 和 $*q$）加入到抽象准则。由于部分最强后置条件是针对一条路径的，因此下次迭代就能够判定 $*p$ 和 $*q$ 是否为别名。

2.5.2 实验结果与分析

本节中，我们基于切片执行工具验证了与文献[74]完全相同的 C 程序和时序安全性质。所有的 C 程序都来自于 Linux 操作系统中 SSL 协议的实现程序 *openssl-0.9.6c*，它主要由服务器端程序和客户端程序两个模块组成，每个模块包含 2,000 多行 C 程序，它们实现了用于在 Internet 上安全传输数据的 SSL 协议。SSL 协议的一个关键的组成部分是通信初始阶段的握手协议，用于建立客户端和服务器端之间的安全连接。我们验证了握手协议的多个不同的时序安全性质，其中以 *"ssl-clnt"* 和 *"ssl-srvr"* 开头的性质名分别表示客户端和服务器端应满足的性质。与文献[74]的结果一样，切片执行验证了所有性质都被满足。

```
while(1){
    switch(s->state){
        case S1: do some actions; s->state = NEXT(S1); break;
        case S2: do some actions; s->state = NEXT(S2); break;
        ......
        case Sn: do some actions; s->state = NEXT(Sn); break;
    }
}
```

图 2.5 SSL 程序的代码结构

SSL 初始握手协议的客户端和服务器端都实现为状态机的形式，其代码的结构如图 2.5 所示。其顶层结构是一个包含大约 35 个 case 语句的 switch 语句，每个 case 语句即为协议状态机的一个状态，其中的代码分为两个部分，一部分完成协议在该状态的动作，另一部分决定当前状态的下一个状态。我们要验证的时序安全性质是，协议状态机的某些特定的状态迁移序列在握手过程中不允许存在。例如，名为 *"ssl-clnt-1"* 的性质规定，SSL 客户端程序中不允许出现如图 2.6 所示的状态迁移序列，其中第一个状态 SSL3_ST_CW_CLNT_HELLO 是性质状态机的初始状态，最后一个状态 SSL3_ST_CR_CERT_REQ 则是性质状态机的接收状态，该

性质状态机描述的性质的确切语义可参见文献[75]。

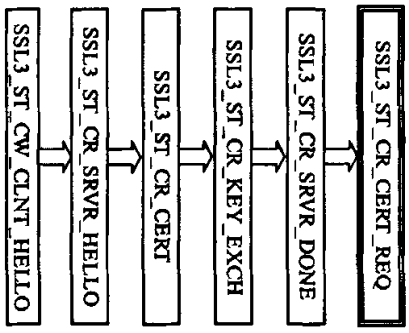

图 2.6 性质 ssl-clnt-1 描述的性质自动机

所有的实验都是在一台 CPU 为 1.6GHz AMD Athlon XP、物理内存总数为 224MB（256MB 内存减去共享给显示卡使用的 32MB 内存）的计算机上进行的，其软件环境是 Windows 2000 和 CygWin 2.427。

表 2.1 SSL 协议的初始握手过程验证的实验结果

Properties		Slicing Execution					BLAST	MAGIC
Name	Prop. States	Iters	Vars	SEC elements Max / Total	Theorem Calls	Time	Time	Time
ssl-clnt-1	6	6	8	1157 / 3729	32518	**28**	348	156
ssl-clnt-2	6	5	7	394 / 1543	8632	**10**	523	185
ssl-clnt-3	6	6	8	1533 / 4406	45327	**39**	469	195
ssl-clnt-4	6	6	8	1323 / 4027	37966	**34**	380	191
ssl-srvr-1	6	4	6	1703 / 5405	111495	**85**	2398	226
ssl-srvr-2	5	4	6	1342 / 4340	84791	**66**	691	216
ssl-srvr-3	5	4	6	1345 / 4412	86018	**67**	1162	200
ssl-srvr-4	5	4	6	1344 / 4398	87000	**68**	284	170
ssl-srvr-5	9	4	5	2609 / 7224	211516	**145**	1804	205
ssl-srvr-6	22	4	5	7836 / 24687	894003	698	*	**359**
ssl-srvr-7	9	4	5	2592 / 7218	221230	**150**	359	196
ssl-srvr-8	11	4	5	3483 / 10450	279124	**207**	*	211
ssl-srvr-9	9	4	5	2596 / 7180	213075	**145**	337	316
ssl-srvr-10	8	4	5	2388 / 7141	169836	**127**	8289	241
ssl-srvr-11	9	4	5	2608 / 7197	211821	**145**	547	356
ssl-srvr-12	13	4	5	4269 / 12845	385410	**280**	2434	301
ssl-srvr-13	9	4	5	2596 / 7169	210169	**144**	608	436
ssl-srvr-14	16	4	5	5428 / 16562	531295	**405**	10444	406
ssl-srvr-15	14	4	5	4608 / 13992	409549	305	*	**179**
ssl-srvr-16	19	4	5	6617 / 20507	700580	536	*	**356**

实验结果列于表 2.1 中，其中："Prop. states"是性质状态机的状态总数，例

如图 2.6 所示的性质状态机具有 6 个状态；“Iters”显示了验证过程中的总的迭代次数；“Vars”是最终抽象准则中变量的数量，对所有性质的验证都是从一个空的抽象准则开始的；“SEC elements”是所有程序位置的切片执行上下文中的部分最强后置条件的总数量，其中“Max”给出的数字是 SEC elements 数量最多的一次迭代的部分最强后置条件的总数量，而“Total”则是所有迭代的累加和；“Theorem Calls”是调用定理证明工具 Simplify 的次数；“Times”是以秒为单位的验证过程所需要的时间，我们同时也列出了来自文献[74]的 BLAST 和 MAGIC 所需的验证时间，其数据是在一台 CPU 为 1.6GHz Athlon XP 但物理内存总数为 900MB、操作系统为 Linux 的机器上得出来的，其中“*”表示验证时间超过 3 个小时，表中验证用时最少的结果用粗体标识了出来。从表 2.1 我们可以看出，如果性质自动机的状态数少于 16，则切片执行的验证时间少于 MAGIC，也大大少于 BLAST。对于性质自动机状态数为 16 的性质“ssl-srvr-14”，切片执行与 MAGIC 用时相当，而对于具有 19 个状态的性质“ssl-srvr-16”和具有 22 个状态的性质“ssl-srvr-6”，切片执行则使用了比 MAGIC 多的验证时间。这个结果是符合理论预期的，其原因在于切片执行对待验证的性质非常敏感，随着待验证性质的状态数增加，验证所需的时间和对定理证明工具的调用次数都随之增加。而 MAGIC 是基于谓词抽象的模型检验工具，它的验证复杂度是由谓词抽象所需的谓词数量决定的。实验中性质自动机的状态数对谓词抽象所需要的谓词数量影响不大，从而 MAGIC 对复杂的性质的验证所花费的时间并不会有太大的增加。因此，一般来说切片执行适用于验证较大规模的程序和比较简单的性质。

最后需要指出的是，切片执行工具大量地调用了定理证明工具，用于对切片执行过程中的一阶逻辑公式进行判定，由于我们采用了 2.3.3 节所讨论的优化方法，使得这些判定公式都相当简单，从而在实验中定理证明工具每秒钟能够判定 2000 多个简单的一阶逻辑公式。

2.6 相关工作

切片执行受到了程序切片、静态分析、符号执行以及基于谓词抽象的模型检验等相关工作的启发。

切片执行借鉴了静态程序切片[62]和动态程序切片[76]的思想和方法，分别应用于变量抽象和基于伪反例路径的抽象准则精化。不同的是，变量抽象的思想虽然与静态程序切片类似，但由于不需要计算依赖关系，其计算代价要大大小于静态程序切片。动态程序切片则可以直接用于精化抽象准则，当前的动态程序切片方法（如文献[70]所讨论的方法）的可扩展性很强，可以帮助提高切片执行的可扩展性。此外，我们在论文 2.4.3 节提出的抽象准则精化方法也需要用到动态切片技术

的指针和变量别名等处理方法。

局部化缩减（Localization Reduction）[77]的思想与变量抽象很相近，它的目标是根据变量之间的依赖关系去掉与给定性质不相关的程序变量，并且如果由于抽象过粗而导致性质不被满足，它也是根据反例路径推断出一些程序变量来精化模型。局部化缩减与变量抽象不同，它是基于变量依赖图进行抽象的，而我们的方法则只在模型检验工具给出的反例路径上考虑变量之间的依赖关系，从而节省了计算代价。此外，变量抽象的实现和应用也有别于局部化缩减，例如根据抽象准则判定程序语句的相关性、基于最强后置条件判定路径的可行性等等。

静态分析是一种直接分析程序代码，并检查路径错误的方法[4, 9, 78]，它并不像切片执行那样先从程序中抽象出一个有限状态模型，再基于模型检验进行验证。一般来说，静态分析不能保证验证的可靠性和完备性，这是由于静态分析通常只能检查程序的部分执行路径，并常常忽略复杂的变量别名，甚至忽略程序的数据流。为了保证找到程序中的所有错误，静态分析必须进行路径敏感的数据流分析，ESP 就是这一类的静态分析工作。ESP[10]的最大特点是，它能够对时序安全性质进行路径敏感的数据流分析，但其验证开销仅为多项式时间和空间。它基于一种名为“性质模拟”的思想，根据性质的状态合并等效的执行路径，从而有效地防止了路径组合爆炸。与 ESP 的方法类似，切片执行也是通过合并等效执行路径来缩减生成模型的状态空间，其最大的不同在于，切片执行仅考虑了程序中的少部分语句，因此能够合并更多的执行路径，从而有望进一步降低验证开销。

切片执行可被视为一种轻量级的符号执行，它与符号执行[67]一样，遍历程序的所有执行路径，并在遍历的同时生成和维护一个符号表达式作为路径条件（Path Condition）。切片执行的特点在于将抽象的思想引入到遍历的过程中，从而能够有效地缩减所需遍历的组合执行路径。美国宇航局（NASA）近年来将符号执行引入到 Java PathFinder 项目，他们设计了一种基于符号执行和不变式生成的新方法[53, 54]，该方法首先通过源对源转换完成对 Java 程序的标注，然后再符号执行标注后的 Java 程序从而对给定的性质进行验证。为了支持对循环的符号执行，研究者们提出了一种基于模型检验的循环不变式自动生成方法，基于自动生成的循环不变式将循环结构等价变换为非循环结构。与其相比，切片执行本身就支持循环，从而无需花费高昂的代价来生成循环不变式。

与切片执行的目标一样，谓词抽象[55]基于一组给定的谓词集合自动从程序中抽象出有限状态模型，模型的状态表示为对谓词的布尔赋值（详见第一章的介绍）。同样，谓词抽象也基于 CEGAR[56]方法自动地精化抽象准则。切片执行相对于谓词抽象的优点是，它能够直接对程序进行模型检查，无需生成完整的模型就能找到反例路径。另外，它也能有效克服谓词抽象的状态描述不精确以及单个赋值语句

可能的指数次的定理证明工具调用次数问题。

2.7 小结

本章提出了切片执行的基本概念体系，包含变量抽象、部分最强后置条件、切片执行上下文及切片执行图等，并给出了用于时序安全性质验证的切片执行过程。切片执行的显著特点是：在引入了多种思想，包括基于变量抽象的程序保守近似语义、抽象精化和符号执行等之后，切片执行能够以接近流敏感数据流分析的代价，对时序安全性质进行路径敏感的精确验证，保证了验证过程的可靠性。

基于 SSL 协议的实现程序 *openssl-0.9.6c*，我们基于切片执行工具验证了 SSL 协议的初始握手过程的大量时序安全性质，实验结果证明了切片执行方法对验证时序安全性质的高效性。

第三章　搜索复用框架及其在切片执行中的应用

在基于切片执行的验证过程中，我们基于反例指导的抽象精化（CEGAR）方法，根据每次迭代的伪反例路径自动对下一次迭代的抽象准则进行精化，从而将尽量少的变量加入到抽象准则中，以尽量减小验证代价。CEGAR 方法具有“懒惰”（只有出现了伪反例路径才对抽象模型进行精化）和“被动”（精化后的抽象模型只需要阻止伪反例路径的继续出现即可）的特点，从而能够在达到验证目标的前提下尽量减小验证代价。但同时，CEGAR 方法要求每次迭代都生成全新的抽象模型，而不支持不同精度抽象模型之间的信息重用。例如在切片执行过程中，每次对抽象准则进行精化后都需要重新生成切片执行图。而事实上，尽管上一次迭代出现了伪反例路径，但仍有大量的信息可以重用于下一次迭代过程中，从而能够进一步降低切片执行的代价。

直观地，在上一次切片执行生成的切片执行图 $SEG=\langle \Psi, \longrightarrow \rangle$ 中，设某状态 $\psi \in \Psi$ 对应的性质自动机的状态为 ε（不失一般性，我们假设只需要考虑一个性质自动机状态变量），如果切片执行图 SEG 中从状态 ψ 出发的所有迁移路径都不能使性质自动机从状态 ε 迁移到其任何一个接收态，则我们称 ψ 是切片执行图 SEG 的一个可靠状态。由于切片执行图中从可靠状态出发的所有迁移路径都不违背给定性质，因此可靠状态及其所有后续状态和相应的迁移都不需要在下一次迭代过程中重新构建，即它们就是可以在两次迭代之间可以重用的信息。

本章中，我们在 CEGAR 框架的基础上，提出了一种面向切片执行的搜索复用框架，用于代替 CEGAR 框架指导切片执行的模型精化过程。基于搜索复用框架的切片执行在每次迭代的切片执行过程中采用深度优先搜索策略，从而能够找出生成的切片执行图中哪些状态为可靠状态，并重用于下一次迭代过程。我们基于 *openssl-0.9.6c* 实用程序进行了实验，通过验证相同的性质，我们比较了本章的方法与第二章切片执行、以及 MAGIC 等验证方法和工具的效率。实验结果说明，对大部分待验证的性质，基于搜索复用框架的切片执行方法的验证效率得到了大幅度的提高。

3.1 面向切片执行的搜索复用框架

为了表述的简单和直观，同时也不失一般性，我们假设程序中只有一个变量对应于性质自动机的状态变量，并延续第二章的记法，用 ε 表示该程序变量对应的性质自动机的状态。一般地，如果我们需要考虑多个程序变量所对应的性质自动机的状态，则与第二章的方法相同，我们只需将 ε 扩展为程序变量到自动机状态变

量的映射Σ即可。

为了能够在不同精度的切片执行图之间复用信息，我们对切片执行上下文和切片执行图的定义进行延拓，使得其包含性质自动机的状态信息。

定义 3.1 对于 C 程序模型$CP=\langle S,s_0,T,\Delta\rangle$，给定一个抽象准则 V 和一个程序位置$s\in S$，设 P 是从初始程序位置s_0到程序位置 s 的一组执行路径的集合，则程序位置 s 对应于执行路径集合 P 的切片执行上下文 $SEC_P(s)$定义为公式(3.1)所示的形式，其中$\langle\Omega_p,\Phi_p\rangle=\widetilde{SP}_V(p)(True)$为执行路径 p 对应的部分最强后置条件，ε_p为从初始状态ε_0开始，经过执行路径 p 中的所有迁移后性质自动机到达的状态。

$$SEC_P(s)\triangleq\left\{\left\langle\langle\Omega_p,\Phi_p\rangle,\varepsilon_p\right\rangle\middle|\,p\in P\right\} \tag{3.1}$$

同样，如果我们不关心执行路径集合 P，则切片执行上下文 $SEC_P(s)$也可以简记为 $SEC(s)$。

定义 3.2 部分最强后置条件$\langle\Omega,\Phi\rangle$与性质自动机的状态$\varepsilon$的组合$\langle\langle\Omega,\Phi\rangle,\varepsilon\rangle$蕴含（Imply）切片执行上下文 $SEC(s)$，记为$\langle\langle\Omega,\Phi\rangle,\varepsilon\rangle\Rightarrow SEC(s)$，如果下列公式(3.2)成立：

$$h(\langle\Omega,\Phi\rangle)\Rightarrow\bigvee_{\langle\langle\Omega',\Phi'\rangle,\varepsilon'\rangle\in SEC(s)\wedge\varepsilon'=\varepsilon}h(\langle\Omega',\Phi'\rangle) \tag{3.2}$$

直观地讲，如果部分最强后置条件$\langle\Omega,\Phi\rangle$蕴含了集合 $SEC(s)$中所有性质自动机的状态为ε的元素（即公式(3.2)中描述的元素$\langle\langle\Omega',\Phi'\rangle,\varepsilon'\rangle\in SEC(s)\wedge\varepsilon'=\varepsilon$）的部分最强后置条件的析取，则$\langle\langle\Omega,\Phi\rangle,\varepsilon\rangle\Rightarrow SEC(s)$。

相应地，如果$\langle\langle\Omega,\Phi\rangle,\varepsilon\rangle\Rightarrow SEC(s)$，则函数$ImplySEC(\langle\langle\Omega,\Phi\rangle,\varepsilon\rangle)(SEC(s))$返回 $SEC(s)$中与$\langle\langle\Omega,\Phi\rangle,\varepsilon\rangle$有交集的元素的集合，即公式(3.3)所描述的集合：

$$\left\{\langle\langle\Omega',\Phi'\rangle,\varepsilon'\rangle\in SEC(s)\middle|\,\varepsilon'=\varepsilon\ \wedge\ \left(h(\langle\Omega',\Phi'\rangle)\wedge h(\langle\Omega,\Phi\rangle)\neq False\right)\right\} \tag{3.3}$$

C 程序模型$CP=\langle S,s_0,T,\Delta\rangle$的切片执行图仍然表示为$SEG=\langle\Psi,\longrightarrow\rangle$，但我们将其状态集合延拓为$\Psi\subseteq S\times[\![PSP]\!]\times[\![E]\!]$，其中E为性质自动机的状态集合，$[\![E]\!]$则表示性质自动机的状态全集。切片执行图中的一条边$\langle s,\langle\Omega,\Phi\rangle,\varepsilon\rangle\xrightarrow{t}\langle s',\langle\Omega',\Phi'\rangle,\varepsilon'\rangle$表示：$\langle s,t,s'\rangle\in\Delta$、$\langle\Omega',\Phi'\rangle=\widetilde{SP}_V(t)(\langle\Omega,\Phi\rangle)$以及迁移事件 t 让性质自动机从状态ε迁移到后继状态ε'，如果 t 不是性质自动机在状态ε的迁移事件，则$\varepsilon=\varepsilon'$。

定义 3.3 设$\psi=\langle s,\langle\Omega,\Phi\rangle,\varepsilon\rangle$为切片执行图$SEG=\langle\Psi,\longrightarrow\rangle$的一个状态，如果切片执行图中从状态$\psi$出发的所有迁移路径都不能使性质自动机从状态$\varepsilon$迁移到其接收状态，则称状态$\psi$是一个可靠状态。

对切片执行图进行模型检验时将遍历图中的所有迁移路径，从而判定每个状态是否可靠。显然，可靠状态的所有后继状态都是可靠状态。另外，如果切片执行图的初始状态$\langle s_0,\langle\Omega_0,\Phi_0\rangle,\varepsilon_0\rangle$为可靠状态，那么切片执行图的所有状态都是可靠状态，也即该切片执行图满足给定的时序安全性质。

3.2 基于搜索复用框架的切片执行

面向切片执行的搜索复用框架如图 3.1 所示，回忆第二章基于 CEGAR 的切片执行验证框架，如果通过模型检验找到了切片执行图中违背给定性质、但实际并不可行的一条伪反例路径，则需要基于该伪反例路径精化抽象准则，并重新生成新的切片执行图。而基于搜索复用的切片执行框架并不需要重新构建整个切片执行图，原切片执行图中的所有可靠状态及所有从可靠状态出发的迁移路径由于已经满足给定的性质，从而没有必要重新构建。

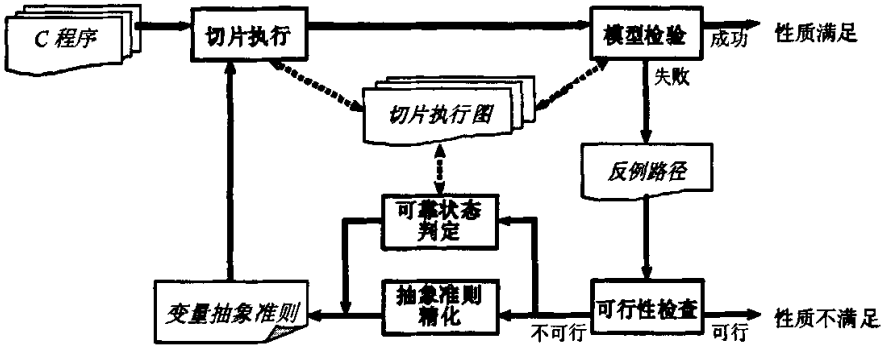

图 3.1 面向切片执行的搜索复用框架

在图 3.1 所示的搜索复用框架中，切片执行图不再由每次迭代过程重新生成，而是作为整个基于切片执行的验证过程的全局数据由各次迭代共享。同时，在反例路径检查后，多了一个“可靠状态判定”模块，用于去掉切片执行图中的不可靠状态。基于搜索复用框架的切片执行过程是这样的，每次迭代过程中，切片执行在已有的部分切片执行图的基础上生成切片执行图的其它部分。如果切片执行过程中发现了一条伪反例路径，则可靠状态判定模块将当前切片执行图中不可靠的状态及相应的迁移去掉，得到的切片执行图就是下一次迭代的基础，其中所有的可靠状态及相应的迁移可被下一次迭代复用。

按照上述框架，基于第二章图 2.2 所示的切片执行过程，集成搜索复用框架的切片执行过程的伪代码如图 3.2 所示。全局变量 *SecMap* 记录了 C 程序模型$CP=\langle S,s_0,T,\Delta\rangle$的每个程序位置$s\in S$对应的当前的切片执行上下文 *SecMap*(*s*)，

其中[E]表示性质自动的所有状态的全集。基于上一次迭代生成的部分切片执行图，*SecMap*(*s*)在过程第 1 行被置为对应于程序位置 *s* 的所有可靠状态的相应元素的并。切片执行过程经过第 2 行对初始程序位置 s_0 的切片执行上下文的初始化后，在第 3 行调用第二章图 2.2 所示切片执行过程构建切片执行图。如果在构建切片执行图的验证过程中性质未被违背，则验证结束（第 4 行）；否则，在第 5 行确定本次迭代本次迭代生成的所有可靠状态和相应可靠状态迁移，为下次迭代作准备。具体地，第 6 行将当前切片执行图中的不可靠状态去掉，第 7 行则只保留图中可靠状态之间的迁移。

$SecMap: S \mapsto 2^{[PSP]\rtimes[E]}$;
SlicingExecution($CP=\langle S,s_0,T,\Delta\rangle, SEG=\langle \Psi, \longrightarrow\rangle$)
{
1 for each $s\in S$ let $SecMap(s) := \{\langle\langle\Omega,\Phi\rangle,\varepsilon\rangle \mid \langle s,\langle\Omega,\Phi\rangle,\varepsilon\rangle\in\Psi\}$;
2 $SecMap(s_0) := \{\langle\langle\Omega_0,\Phi_0\rangle,\varepsilon_0\rangle\}$;
3 调用第二章图 2.2 所示切片执行过程生成切片执行图;
4 若性质满足，则退出，否则继续;
5 确定本次迭代生成的切片执行图中的可靠状态 $SoundStates^*\subseteq\Psi$;
6 令 $\Psi := SoundStates^*$;
7 对切片执行图中的每条边 $\psi_1\longrightarrow\psi_2$，若 $\psi_1,\psi_2\in SoundStates^*$，则保留此边，否则从图中移除;
}

图 3.2 基于搜索复用框架的切片执行过程伪代码

基于图 3.2 所示的集成搜索复用框架的切片执行过程，切片执行图的生成代价将会从直接和间接两个方面显著降低：(1)直接地，当前保留的部分切片执行图不需要重新构建，从而节约了其构建代价；(2)间接地，由于某些程序位置的切片执行上下文已经被初始化，因此在新迭代的切片执行过程中，到达这些位置的执行路径可能由于其部分最强后置条件蕴含该点的切片执行上下文而不需要继续执行，从而进一步降低了切片执行代价。

下面我们证明基于搜索复用框架的切片执行过程是正确的，即满足定理 3.1。为了对定理 3.1 进行证明，我们先证明引理 3.1。

引理 3.1 给定两个变量集合 V_1 和 V_2，其中 $V_1\subseteq V_2$，则对任意部分最强后置条件 $\langle\Omega,\Phi\rangle$ 和任意执行路径 $p=t_1t_2\cdots t_n$，令 $\langle\Omega_1,\Phi_1\rangle=\widetilde{SP}_{V_1}(p)(\langle\Omega,\Phi\rangle)$、$\langle\Omega_2,\Phi_2\rangle=\widetilde{SP}_{V_2}(p)(\langle\Omega,\Phi\rangle)$，如果 $\wedge_{\phi\in\Phi_1}\phi=False$，则 $\wedge_{\phi\in\Phi_2}\phi=False$。

证明：令 $\langle\Omega_1^i,\Phi_1^i\rangle=\widetilde{SP}_{V_1}(t_1\cdots t_i)(\langle\Omega,\Phi\rangle)$、$\langle\Omega_2^i,\Phi_2^i\rangle=\widetilde{SP}_{V_2}(t_1\cdots t_i)(\langle\Omega,\Phi\rangle)$，我们先证明下列公式(3.4)对所有 i:$1\le i\le n$ 都成立：

$$\left(\exists\theta_1,\cdots,\theta_{n2}:\bigwedge_{(x\to e)\in\Omega_2^i\wedge Vars(x)\subseteq V_1}x=e\right)\Rightarrow\left(\exists\theta_1,\cdots,\theta_{n1}:\bigwedge_{(x\to e)\in\Omega_1^i\wedge Vars(x)\subseteq V_1}x=e\right)\quad(3.4)$$

其中$\theta_1,\cdots,\theta_{n2}$为$\Omega_2^i$中包含的所有 Skolem 常量，而$\theta_1,\cdots,\theta_{n1}$则为$\Omega_1^i$中包含的所有 Skolem 常量。公式(3.4)的直观意思是，给定对集合V_1中所有变量的任意一组取值，如果该组取值对映射Ω_2^i而言不冲突，则它也必然满足映射Ω_1^i的约束。换言之，对集合V_1中的变量来说，Ω_1^i描述了比Ω_2^i更一般的条件。

基于归纳法，我们假设公式(3.4)对 i 成立，来证明其对 i+1 也成立，我们对迁移t_{i+1}分下列情况证明：

- 若t_{i+1}是 assume 语句，或者t_{i+1}是在抽象准则V_1和V_2下都是**无关**的赋值语句，则$\Omega_1^{i+1}=\Omega_1^i$、$\Omega_2^{i+1}=\Omega_2^i$，因此公式(3.4)对 i+1 也成立；
- 若t_{i+1}在抽象准则V_1和V_2下都是**完全相关**的赋值语句$x:=e$，则由于$Vars(e)\subseteq V_1$，且对集合V_1中的变量来说Ω_1^i描述了比Ω_2^i更一般的条件，故任何满足约束$x=\Omega_2^i(e)$的变量 x 的取值都能满足约束$x=\Omega_1^i(e)$，从而公式(3.4)对 i+1 也成立；
- 若t_{i+1}是抽象准则V_1下的**无关**赋值语句$x:=e$，则根据部分最强后置条件的定义，对$\left\langle\Omega_1^{i+1},\Phi_1^{i+1}\right\rangle$而言，$x$ 的任意取值都是合法的，而其它的变量在Ω_1^{i+1}中被映射的值则保持不变，从而公式(3.4)对 i+1 成立；
- 若t_{i+1}是抽象准则V_1下的**部分相关**赋值语句$x:=e$，则根据部分最强后置条件的定义，在映射Ω_1^{i+1}中 x 将被映射为一个新的 Skolem 常量θ_x，表明 x 可取任意值，从而公式(3.4)对 i+1 成立；

综上，我们可知公式(3.4)对所有$1\le i\le n$都成立。

接下来，我们考察路径 p 中抽象准则V_1下的每个相关 assume 语句t_i，令其为 $assume(c)$，从而有$Vars(c)\subseteq V_1$。根据部分最强后置条件的定义，$\Omega_1^i(c)$和$\Omega_2^i(c)$将分别被加入到集合Φ_1^{i+1}和Φ_2^{i+1}中。根据公式(3.4)知Ω_1^i描述了比Ω_2^i更一般的条件，故$\Omega_1^i(c)$描述了比$\Omega_2^i(c)$更一般的条件。因此，如果对集合V_1中变量的任意取值都有$\wedge_{\phi\in\Phi_1}\phi=False$，那么描述了更加严格条件的公式$\wedge_{\phi\in\Phi_2}\phi$也必然为 $False$。 ■

引理 3.1 说明，如果两个抽象准则V_1和V_2满足关系$V_1\subseteq V_2$，则对任意部分最强后置条件$\langle\Omega,\Phi\rangle$和任意路径 p，如果$\widetilde{SP}_{V_1}(p)(\langle\Omega,\Phi\rangle)=False$，则一定有$\widetilde{SP}_{V_2}(p)(\langle\Omega,\Phi\rangle)=False$。设基于抽象准则$V_1$和$V_2$生成的切片执行图分别为$SEG_1$和$SEG_2$，则对$SEG_1$和$SEG_2$的任意相同状态$\langle s,\langle\Omega,\Phi\rangle,\varepsilon\rangle$，从该状态出发的任意迁移路径 p 若在SEG_1中不可行，则一定在SEG_2中不可行。

定理 3.1 基于相同的精化后的抽象准则V'，设第二章图 2.2 所示的切片执行过程生成的切片执行图为$SEG=\langle\Psi,\longrightarrow\rangle$，而本章图 3.2 所示的基于搜索复用框架

的切片执行生成的切片执行图为 $SEG'=\left\langle \Psi', \xrightarrow{\cdot} \right\rangle$。对任意迁移序列 $t_1t_2\cdots t_n$，若存在 $n+1$ 个状态 $\psi_i \in \Psi$（其中 $0\le i\le n$），使得 $\psi_0 \xrightarrow{t_1} \psi_1 \xrightarrow{t_2} \cdots \xrightarrow{t_n} \psi_n$，则必定存在 $n+1$ 个状态 $\psi'_i \in \Psi'$（其中 $0\le i\le n$），使得 $\psi'_0 \xrightarrow{t_1} \psi'_1 \xrightarrow{t_2} \cdots \xrightarrow{t_n} \psi'_n$。

证明：基于反证法，我们假设所产生的切片执行图 SEG' 中并不包含所给路径 $\psi'_0 \xrightarrow{t_1} \psi'_1 \xrightarrow{t_2} \cdots \xrightarrow{t_n} \psi'_n$，那么对于包含在 SEG' 中的该路径的最长子路径 $\psi'_0 \xrightarrow{t_1} \cdots \xrightarrow{t_k} \psi'_k$（其中 $k<n$），我们设 $\psi'_k=\left\langle s'_k, \left\langle \Omega'_k, \Phi'_k \right\rangle, \varepsilon'_k \right\rangle$。根据第二章介绍的切片执行过程，我们可以得出 $\widetilde{SP}_{V'}\left(t_{k+1}\right)\left(\left\langle \Omega'_k, \Phi'_k \right\rangle\right)=False$，进而有 $\widetilde{SP}_{V'}\left(t_{k+1}\cdots t_n\right)\left(\left\langle \Omega'_k, \Phi'_k \right\rangle\right)=False$。考虑 SEG' 中存在的迁移 $\psi'_{k-1} \xrightarrow{t_k} \psi'_k$，其中 $\psi'_{k-1}=\left\langle s'_{k-1}, \left\langle \Omega'_{k-1}, \Phi'_{k-1} \right\rangle, \varepsilon'_{k-1} \right\rangle$，如果 $\widetilde{SP}_{V'}\left(t_k\right)\left(\left\langle \Omega'_{k-1}, \Phi'_{k-1} \right\rangle\right)=\left\langle \Omega'_k, \Phi'_k \right\rangle$（此时有 $\left\langle \left\langle \Omega'_k, \Phi'_k \right\rangle, \varepsilon'_k \right\rangle \Rightarrow SecMap\left(s'_k\right)$），则 $\widetilde{SP}_{V'}\left(t_kt_{k+1}\cdots t_n\right)\left(\left\langle \Omega'_{k-1}, \Phi'_{k-1} \right\rangle\right)=False$。另一方面，如果 $\widetilde{SP}_{V'}\left(t_k\right)\left(\left\langle \Omega'_{k-1}, \Phi'_{k-1} \right\rangle\right)\neq\left\langle \Omega'_k, \Phi'_k \right\rangle$，设 $\left\langle \Omega''_k, \Phi''_k \right\rangle=\widetilde{SP}_{V'}\left(t_k\right)\left(\left\langle \Omega'_{k-1}, \Phi'_{k-1} \right\rangle\right)$，则根据切片执行过程，必有 $\left\langle \left\langle \Omega''_k, \Phi''_k \right\rangle, \varepsilon'_k \right\rangle \Rightarrow SecMap\left(s'_k\right)$ 以及 $\left\langle \left\langle \Omega'_k, \Phi'_k \right\rangle, \varepsilon'_k \right\rangle \in ImplySEC\left(\left\langle \left\langle \Omega''_k, \Phi''_k \right\rangle, \varepsilon'_k \right\rangle\right)\left(SecMap\left(s'_k\right)\right)$。下面我们证明公式 $\widetilde{SP}_{V'}\left(t_kt_{k+1}\cdots t_n\right)\left(\left\langle \Omega'_{k-1}, \Phi'_{k-1} \right\rangle\right)=False$ 仍然成立。

我们考察集合 $ImplySEC\left(\left\langle \left\langle \Omega''_k, \Phi''_k \right\rangle, \varepsilon'_k \right\rangle\right)\left(SecMap\left(s'_k\right)\right)$ 中除 $\left\langle \left\langle \Omega'_k, \Phi'_k \right\rangle, \varepsilon'_k \right\rangle$ 外的其它元素 $\left\langle \left\langle \Omega'''_k, \Phi''_k \right\rangle, \varepsilon'_k \right\rangle$，如果它对应于本次切片执行生成的某个状态，则不失一般性，我们有 $\widetilde{SP}_{V'}\left(t_{k+1}\cdots t_n\right)\left(\left\langle \Omega'''_k, \Phi''_k \right\rangle\right)=False$，否则切片执行图中将会存在路径 $\psi'''_k \xrightarrow{t_{k+1}} \cdots \xrightarrow{t_n} \psi'''_n$（其中 $\psi'''_k=\left\langle s'_k, \left\langle \Omega'''_k, \Phi''_k \right\rangle \right\rangle$）。反之，如果 $\left\langle \left\langle \Omega'''_k, \Phi''_k \right\rangle, \varepsilon'_k \right\rangle$ 对应于上一次切片执行的可靠状态，则类似地，必有 $\widetilde{SP}_{V}\left(t_{k+1}\cdots t_n\right)\left(\left\langle \Omega'''_k, \Phi''_k \right\rangle\right)=False$，其中 V 为上一次切片执行的抽象准则且 $V\subseteq V'$。根据引理 3.1，我们知道 $\widetilde{SP}_{V'}\left(t_{k+1}\cdots t_n\right)\left(\left\langle \Omega'''_k, \Phi''_k \right\rangle\right)=False$。类似地，由于 $\left\langle \left\langle \Omega''_k, \Phi''_k \right\rangle, \varepsilon'_k \right\rangle \Rightarrow SecMap\left(s'_k\right)$，故 $\widetilde{SP}_{V'}\left(t_{k+1}\cdots t_n\right)\left(\left\langle \Omega''_k, \Phi''_k \right\rangle\right)=False$，从而 $\widetilde{SP}_{V'}\left(t_kt_{k+1}\cdots t_n\right)\left(\left\langle \Omega'_{k-1}, \Phi'_{k-1} \right\rangle\right)=False$。

重复进行上述过程，我们最终得到 $\widetilde{SP}_{V'}\left(t_1t_2\cdots t_n\right)\left(\left\langle \Omega_0, \Phi_0 \right\rangle\right)=False$。而定理前提条件则指出，切片执行图 SEG 中存在迁移路径 $\psi_0 \xrightarrow{t_1} \psi_1 \xrightarrow{t_2} \cdots \xrightarrow{t_n} \psi_n$，故 $\widetilde{SP}_{V'}\left(t_1t_2\cdots t_n\right)\left(\left\langle \Omega_0, \Phi_0 \right\rangle\right)\neq False$，从而得出矛盾，定理证毕。 ■

定理 3.2 基于相同的精化后的抽象准则 V'，设第二章图 2.2 所示的切片执行过程生成的切片执行图为 $SEG=\left\langle \Psi, \xrightarrow{\cdot} \right\rangle$，而本章图 3.2 所示的基于搜索复用框架的切片执行生成的切片执行图为 $SEG'=\left\langle \Psi', \xrightarrow{\cdot} \right\rangle$。如果 SEG' 中存在一条违背给定时序安全性质的反例迁移路径 $t_1\cdots t_n$，则 SEG 中也存在该路径。

证明：根据定理的条件，SEG' 中必存在 $n+1$ 个状态 $\psi'_i \in \Psi'$（其中 $0\le i\le n$），使得 $\psi'_0 \xrightarrow{t_1} \cdots \xrightarrow{t_n} \psi'_n$。根据可靠状态的定义，$\psi'_i \in \Psi'$（$0\le i\le n$）都不是上次切片执行的可靠状态（否则它们不会违背性质），因此所有状态 $\psi'_i \in \Psi'$（$0\le i\le n$）的生成方法与第二章介绍的切片执行是完全相同的，从而也存在于相应的切片执

行图 *SEG* 中。 ■

根据定理 3.1，*SEG* 中的每条路径都包含在了 *SEG'* 中，因此 *SEG* 中每条违背性质的反例迁移路径也必存在于 *SEG'* 中。所以，根据定理 3.1 和定理 3.2，对第二章图 2.2 所示的切片执行过程和本章图 3.2 所示的基于搜索复用框架的切片执行过程而言，当给定相同的抽象准则时，它们生成的切片执行图对时序安全性质的验证而言是等价的，就是说它们要么都满足给定的时序安全性质，要么都不满足且存在相同的（伪）反例路径。

3.3 可靠状态的确定

实现基于搜索复用框架的切片执行过程还剩下一个关键问题，即如何确定哪些状态是本次切片执行的可靠状态，从而能够复用于下一次切片执行过程中。回忆第二章的切片执行过程是一个不动点计算过程，每次处理的状态都是从一个待处理状态的集合中随机选取的。而在本章中，为了确定切片执行所生成的切片执行图中每个状态的可靠性，我们采用基于栈的深度优先搜索策略来选择当前需要处理的状态。每个状态都被保存在一个先进后出的栈中，直到其后继状态被遍历完成后再将其弹出。根据第二章给出的切片执行过程，一个状态被遍历完成是指其所有直接后继状态被遍历完成，或者该状态对应的部分最强后置条件蕴含相应程序位置的切片执行上下文。如果生成的切片执行图中没有圈，则当某状态处理完毕而从栈中弹出时，从该状态出发的所有执行路径都被切片执行完毕，并保证没有违背给定的时序安全性质，从而我们就确定了一个可靠状态。

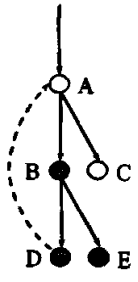

图 3.3 带圈的切片执行图示例

不幸的是，如果生成的切片执行图中存在圈，例如图 3.3 所示的切片执行图，则对切片执行图的某个圈上的状态来说，它被切片执行完毕并不意味着从其出发的所有执行路径都被切片执行完毕。例如在图 3.3 中，黑色圆圈表示已经切片执行完毕的状态，而白色圆圈则表示还未被切片执行完毕。假设状态 A 和状态 D 具有

相同的程序位置，即路径 A→B→D 构成一个圈。我们考虑状态 B，它的后继状态 D 由于其部分最强后置条件蕴含状态 A 当前的切片执行上下文（我们用虚线表示）而被切片执行完毕。当 B 的另一个后续 E 被切片执行完毕后，状态 B 就被切片执行完毕了，可是从 B 出发的一条路径 B→D→A→C 还没有被切片执行，因此不能保证状态 B 一定是可靠状态。

$SoundStates \subseteq \Psi$;
$SecMap : S \mapsto 2^{[PSP]\times[E]}$;
Explore($\langle s,\langle\Omega,\Phi\rangle,\varepsilon\rangle$)
{
1 add $\langle s,\langle\Omega,\Phi\rangle,\varepsilon\rangle$ to Ψ;
2 for all $t\in T, s'\in S$ such that $\langle s,t,s'\rangle\in\Delta$ do {
3 let $\langle\Omega',\Phi'\rangle := \widetilde{SP}_V(t)(\langle\Omega,\Phi\rangle)$;
4 let ε transits to ε' via the transition t in the property FSA;
5 if (ε' is an acceptance state of the property FSA) EXIT;
6 if ($\langle\langle\Omega',\Phi'\rangle,\varepsilon'\rangle \Rightarrow SecMap(s')$){
7 for all $\langle\langle\Omega'',\Phi''\rangle,\varepsilon''\rangle \in ImplySEC(\langle\langle\Omega',\Phi'\rangle,\varepsilon'\rangle)(SecMap(s'))$ do {
8 set $\langle s,\langle\Omega,\Phi\rangle,\varepsilon\rangle \xrightarrow{t} \langle s',\langle\Omega'',\Phi''\rangle,\varepsilon''\rangle$;
9 set $\langle s,\langle\Omega,\Phi\rangle,\varepsilon\rangle \rightarrow_{dep} \langle s',\langle\Omega'',\Phi''\rangle,\varepsilon''\rangle$;
10 }
11 }else{
12 set $\langle s,\langle\Omega,\Phi\rangle,\varepsilon\rangle \xrightarrow{t} \langle s',\langle\Omega',\Phi'\rangle,\varepsilon'\rangle$;
13 add $\langle\langle\Omega',\Phi'\rangle,\varepsilon'\rangle$ to $SecMap(s')$;
14 Explore($\langle s',\langle\Omega',\Phi'\rangle,\varepsilon'\rangle$);
15 for all $\langle s'',\langle\Omega'',\Phi''\rangle,\varepsilon''\rangle\in\Psi$ such that $\langle s'',\langle\Omega'',\Phi''\rangle,\varepsilon''\rangle \neq \langle s,\langle\Omega,\Phi\rangle,\varepsilon\rangle$ and $\langle s',\langle\Omega',\Phi'\rangle,\varepsilon'\rangle \rightarrow_{dep} \langle s'',\langle\Omega'',\Phi''\rangle,\varepsilon''\rangle$ do
16 set $\langle s,\langle\Omega,\Phi\rangle,\varepsilon\rangle \rightarrow_{dep} \langle s'',\langle\Omega'',\Phi''\rangle,\varepsilon''\rangle$
17 }
18 add $\langle s,\langle\Omega,\Phi\rangle,\varepsilon\rangle$ to *SoundStates*;
19 }
}
SlicingExecution($CP=\langle S,s_0,T,\Delta\rangle, SEG=\langle\Psi,\longrightarrow\rangle$)
{
20 $SoundStates := \Psi$;
21 for each $s\in S$ let $SecMap(s) := \{\langle\langle\Omega,\Phi\rangle,\varepsilon\rangle \mid \langle s,\langle\Omega,\Phi\rangle,\varepsilon\rangle\in\Psi\}$;
22 $SecMap(s_0) := \{\langle\langle\Omega_0,\Phi_0\rangle,\varepsilon_0\rangle\}$;
23 Explore($\langle s_0,\langle\Omega_0,\Phi_0\rangle,\varepsilon_0\rangle$);
}

图 3.4 面向搜索复用框架的深度优先切片执行过程

为了支持切片执行图中的圈，我们定义切片执行图的状态之间的依赖关系

$\rightarrow_{dep} \subseteq \Psi \times \Psi$。切片执行图的两个状态$\psi_1$和$\psi_2$之间存在依赖关系$\langle \psi_1, \psi_2 \rangle \in \rightarrow_{dep}$（简记为$\psi_1 \rightarrow_{dep} \psi_2$），是指状态$\psi_1$的可靠性依赖于状态$\psi_2$的可靠性。换言之，如果状态$\psi_2$不可靠，则状态$\psi_1$也不可靠。例如图 3.3 所示的切片执行图中，存在依赖关系$D \rightarrow_{dep} A$和$B \rightarrow_{dep} A$，表示状态 B 和状态 D 的可靠性依赖于状态 A。如果状态 A 被遍历完成且被证明为可靠状态，则状态 B、D 也都是可靠状态，这是由于路径 A→C 不违背给定性质，从而路径 B→D→A→C 也不违背给定性质。

图 3.4 给出了面向搜索复用框架的深度优先的切片执行过程，其输入参数除了给定的 C 程序模型$CP = \langle S, s_0, T, \Delta \rangle$外，还有上一次切片执行生成的切片执行图$SEG = \langle \Psi, \longrightarrow \rangle$的可靠状态及相应边（参见 3.2 节图 3.2 对上一次切片执行图的处理）。该过程不仅在给定切片执行图的基础上继续构建切片执行图$SEG = \langle \Psi, \longrightarrow \rangle$，而且还构建切片执行图的状态之间的依赖关系$\rightarrow_{dep} \subseteq \Psi \times \Psi$，并输出切片执行图中的可靠状态的备选集合 *SoundStates*。

初始时，全局变量 *SoundStates* 在过程第 20 行被置为输入切片执行图的所有可靠状态，另一个全局变量 *SecMap* 记录了 C 程序模型$CP = \langle S, s_0, T, \Delta \rangle$的每个程序位置$s \in S$对应的当前的切片执行上下文 *SecMap(s)*，*SecMap(s)*在过程第 21 行被置为对应于程序位置 s 的所有可靠状态的相应元素的并。切片执行过程经过第 22 行对初始程序位置s_0的切片执行上下文的初始化后，在第 23 行调用递归过程 Explore 构建切片执行图。

Explore 过程第 1 行将当前正在处理的状态加入切片执行图的状态集合Ψ，随后在第 2 行考虑程序位置 s 的所有后继状态s'及相应迁移 t，第 3—4 行计算出当前状态$\langle s, \langle \Omega, \Phi \rangle, \varepsilon \rangle$的可能后继状态$\langle s', \langle \Omega', \Phi' \rangle, \varepsilon' \rangle$。第 5 行中，如果性质自动机迁移到其接受态，则性质被违背，从而我们终止切片执行过程并报告反例路径。否则，如果第 6 行的蕴含关系成立，则除了在第 7—8 行设置相应迁移关系之外，还要在第 9 行设置状态之间的依赖关系，表示当前状态$\langle s, \langle \Omega, \Phi \rangle, \varepsilon \rangle$的可靠性依赖于所有状态$\langle s', \langle \Omega'', \Phi'' \rangle, \varepsilon'' \rangle$的可靠性。如果第 6 行蕴含关系不成立，则经过第 12—13 行的处理后，我们调用 Explore 递归地遍历当前状态的后继状态$\langle s', \langle \Omega', \Phi' \rangle, \varepsilon' \rangle$（第 14 行），当遍历完返回后，我们需要将所有后继状态的依赖关系传递到当前状态，即第 15—16 行，注意第 15 行的第一个条件将阻止一个状态依赖自身。如果当前状态蕴含相应程序位置的切片执行上下文，或者当前状态的所有后继状态被遍历完全且性质没有被违背，则第 18 行我们将当前状态加入到可靠状态集合中。

图 3.4 的切片执行过程终止时（不论是正常终止还是由于性质被违背而终止），我们定义如下集合 *SoundStates** 为当前切片执行图中的可靠状态集：

$$SoundStates^* \triangleq \left\{ \psi \in SoundStates \middle| \forall \psi \rightarrow_{dep} \psi' : \psi' \in SoundStates^* \right\} \tag{3.5}$$

直观地，*SoundStates** 中的所有状态ψ要么不依赖于其它状态，要么只依赖于

SoundStates*中的状态。从 *SoundStates* 构造 *SoundStates**时，我们可以逐个考察集合 *SoundStates* 中的每个状态，如果它依赖的某个状态不在 *SoundStates* 中，则将其去掉。重复执行这个过程，直到没有状态被去掉为止，最终得到的 *SoundStates* 集合就是 *SoundStates**。

定理 3.3 公式(3.5)定义的状态集合 *SoundStates**中的所有状态都是可靠状态。

证明：考察图 3.4 的切片执行过程第 18 行，只有当前状态ψ蕴含相应程序位置的切片执行上下文，或者ψ的所有后继状态被遍历完全且性质没有被违背的情况上，才会将ψ加入到 *SoundStates* 集合中。如果ψ不依赖于其它状态，则深度优先的切片执行过程能够保证从其出发的所有迁移路径都不违背给定性质，从而ψ是一个可靠状态。如果ψ依赖于某个其它的状态ψ'，则公式(3.5)中得到集合 *SoundStates**时，将保证ψ'的可靠性。综合两种情况，定理得证。 ■

3.4 实验结果

我们在上一章切片执行工具的基础上实现了基于搜索复用框架的切片执行工具，工具仍然使用 Simplify[72]作为定理证明工具，并基于 Das 的流不敏感的指向图算法[73]来处理指针和变量别名。该工具支持对函数指针、变量别名、结构和数组的处理，其局限是不支持递归函数调用和非直接跳转如 setjump/longjump 等，但这些限制在实验程序中都不存在。

我们仍然使用与 BLAST、MAGIC 和切片执行相同的测试用例来验证本文所提出的基于搜索复用的切片执行方法的有效性和实用性。所有的测试用例都取自于 openssl-0.9.6c 的程序源代码，我们的验证目标是 SSL 协议的初始握手协议所应满足的 20 个时序安全性质，与其它验证工具的结果一样，所有性质都被验证为被程序满足。

我们的实验平台是 1.6GHz AMD Athlon XP 的 CPU 和 1GB 内存，软件环境是 Windows 2000 和 CygWin 2.427。实验结果如表 3.1 所示，表中“Properties”为待验证的性质，“Name”为性质的名字，“States”为性质状态机的状态数。“Slicing Execution with Reuse of Searching”为本文给出的验证算法，“CE Paths”表示在验证过程中找到的反例路径（Counter-Example Paths）的数量，“Vars”为变量抽象的最终抽象准则中变量的数量，“Thm Calls”为验证过程中调用定理证明工具的次数，“Time”为验证所用的时间，单位是秒。表 3.1 中也给出了 BLAST、MAGIC 和切片执行(“Pure Slicing Execution”)的验证结果，其中 BLAST 和 MAGIC 的数据来自文献[74]，其实验平台是 1.6GHz AMD Athlon XP 的 CPU 和 900MB 内存，软件环境是 Linux。表中“*”表示验证时间超过 3 个小时，最好的验证结果在表

中表示为粗体。需要说明的是，虽然我们的实验平台比文献[74]的实验平台多了100MB的内存，但除了两个性质srvr-6和srvr-14外，其它性质的验证都只使用了不到200MB的内存。

表3.1 基于openssl程序的切片执行实验结果

Properties		Slicing Execution with Reuse of Searching				BLAST	MAGIC	Pure Slicing Execution	
Name	States	CE Paths	Vars	Thm Calls	Time	Time	Time	Thm Calls	Time
ssl-clnt-1	6	8	9	4362	**12**	348	156	32518	28
ssl-clnt-2	6	5	8	877	**8**	523	185	8632	10
ssl-clnt-3	6	6	8	1133	**10**	469	195	45327	39
ssl-clnt-4	6	8	9	4365	**12**	380	191	37966	34
ssl-srvr-1	6	5	8	5020	**24**	2398	226	111495	85
ssl-srvr-2	5	6	9	5821	**65**	691	216	84791	66
ssl-srvr-3	5	7	10	267314	439	1162	200	86018	**67**
ssl-srvr-4	5	6	9	9448	**35**	284	170	87000	68
ssl-srvr-5	9	14	17	111077	405	1804	205	211516	**145**
ssl-srvr-6	22	6	9	1424958	2552	*	**359**	894003	698
ssl-srvr-7	9	5	8	8868	**76**	359	196	221230	150
ssl-srvr-8	11	5	8	5584	**25**	*	211	279124	207
ssl-srvr-9	9	5	8	11164	**80**	337	316	213075	145
ssl-srvr-10	8	7	10	265409	444	8289	241	169836	**127**
ssl-srvr-11	9	5	8	8899	**76**	547	356	211821	145
ssl-srvr-12	13	13	17	99695	374	2434	301	385410	**280**
ssl-srvr-13	9	4	7	2209	**135**	608	436	210169	144
ssl-srvr-14	16	6	9	1422838	2529	10444	406	531295	**405**
ssl-srvr-15	14	7	10	273767	476	*	**179**	409549	305
ssl-srvr-16	19	13	17	102904	397	*	**356**	700580	536

从表3.1的实验结果中我们可以看到，在20个验证用例中有12个都是基于搜索复用的切片执行方法的验证时间最短，而且某些验证用例（如srvr-1、srvr-4、srvr-7、srvr-8、srvr-9及srvr-11等）的验证效果要比其它三种工具或方法优秀很多，这充分说明了本文所提方法的有效性和实用性。但是同时我们也注意到，某些验证用例，如srvr-6和srvr-14，的验证时间大大长于MAGIC和切片执行。经过仔细的分析我们发现，造成这种现象的原因有三个，如何解决这些问题也是我们后续的研究工作：一是由于基于搜索复用的切片执行是基于深度优先搜索实现的，如果算法一开始就选择了一个不会出现错误的分支，那么它就必须等该分支的所有后继被遍历完成后才会选择其它分支；二是造成一条伪反例路径不可行的变量可能有很多组，我们选择使用最少的一组来精化抽象准则未必是最佳选择；三是由于某些变量未被及时加入变量抽象的抽象准则而导致其取值范围被放大，从而增

加大量额外的程序执行路径。

```
while(1){
  switch(s->state){
    case Sm: ...; s->state = (s->s3)->tmp.next_state; break;
    case Sn: ...; (s->s3)->tmp.next_state = 8576; break;
    ......
  }
}
```

图 3.5 *ssl_srvr* 例程中的一段代码

例如 ssl_srvr 程序中有图 3.5 所示的代码，上述两个验证时间过长的性质都是由于没有及时将变量(s->s3)->tmp.next_state 加入抽象准则，而导致 s->state 的取值范围在执行 case S_m 后被放大到可以取任意值，从而在下一次循环中 s->state 被认为可能进入所有的 case。

3.5 相关工作

本章提出的搜索复用框架是对反例指导的抽象精化（CEGAR）框架[56]的改进，我们已经在本章中对两者进行了充分的比较，在此不再累述。CEGAR 框架几乎应用于当前所有基于保守近似抽象（Over-Approximated Abstraction）的模型检验工具中，如 SLAM[41-43]、Zing[44-46]、BLAST[47]、MAGIC[48, 49]及 ComFoRT[50, 52, 79]等，能够大大减小抽象模型的规模。同时，它也被成功地应用于对多级模型抽象[49]和混成系统模型抽象[80]等方面，具有较好的普适性。

实际上，搜索复用框架是先假设每个生成的切片执行图的状态都是可靠状态，再在后续切片执行过程中检验每个状态的可靠性。从这个意义来说，它与 Assume-Guarantee 推理（Assume-Guarantee Reasoning）方法具有相似的思想。Assume-Guarantee 推理广泛地应用于程序模块化组合验证，并取得了良好的效果，如[38, 50, 81]等。其基本思想是，如果需要验证多个程序模块或多个并发进程，我们可以对某几个模块或进程的上下文或输入输出参数进行假设，再基于这些假设条件分别对各个单独的模块或进程进行验证，最后验证这些假设条件得到满足即可。这种分而治之的方法往往能大量地降低验证开销，已经被 VeriSoft[38]、ComFoRT[50]等基于源代码的验证工具用来对构件化和并发程序的验证。文献[81]则论述了如何基于 Assume-Guarantee 推理实现对既包含串行模块、又包含并发进程的程序进行验证。Assume-Guarantee 推理除了用于验证的目的外，还可以用于程序测试，文献[82]给出了这方面的研究。本文提出的基于搜索复用框架的不同之处，在于本文的思路是针对顺序程序搜索中模块化复用，其假设的粒度更细，是在程序语句级。从另一方面来看，本文提出的方法与模块或进程级的 Assume-Guarantee

推理工作在程序的不同级别，可以综合运用。

上一章提出的切片执行方法的每一次迭代过程中的计算都是按不动点的方式进行的，即从待计算的程序位置集合中任取一个，计算其部分最强后置条件后将其后继程序位置加入待计算程序位置集合中，直到达到一个不动点为止。由于算法可以有选择性地遍历最有可能出错的语句分支，从而可以保证其验证时间不会出现类似本章提出的基于搜索复用框架的切片执行验证方法的大起大落（请参见表 3.1 所示的实验结果）。但是与此同时，由于一个新的迭代无法使用之前迭代的信息，一般来说其验证代价会比本章提出的切片执行方法大很多。

3.6 小结

本章提出了一种面向切片执行的搜索复用框架，并将其应用于切片执行中，使得切片执行能够在不同的迭代之间复用搜索信息，从而既能够降低切片执行的代价，又能缩减生成的切片执行图的状态空间。基于搜索复用框架的切片执行过程基于深度优先的状态空间搜索策略判定每个状态的可靠性，当每次迭代因找到伪反例路径而终止时，切片执行引入一个额外的过程，去除切片执行图中不可靠的状态和迁移，并将可靠的状态和迁移带入下一次迭代以进行复用。本文的方法被用于验证 SSL 协议在 Linux 下的实现程序 openssl-0.9.6c 满足 SSL 协议的初始握手规范，实验表明，该方法对大部分性质都较大程度地提高了切片执行的效率。

第四章 部分最弱前置条件及其在切片执行中的应用

在第三章中，基于搜索复用框架的切片执行以深度优先的方式遍历程序的每条执行路径，并同时计算搜索路径的部分最强后置条件。设切片执行到达某个程序位置 s 时的部分最强后置条件为 α，如果公式 $\alpha \Rightarrow \alpha_1 \vee \cdots \vee \alpha_n$ 成立（其中 $\alpha_1,\cdots,\alpha_n$ 分别为切片执行以前 n 次到达程序位置 s 时的部分最强后置条件），则此时切片执行无需搜索从 s 出发的任意执行路径，原因是从 s 出发且程序变量取值满足公式 α 的所有执行路径都在前 n 次到达 s 时检查过了。由于切片执行只考虑程序中与待验证的时序安全性质相关的小部分程序变量，因此多次到达某个程序位置时的部分最强后置条件往往能够满足公式 $\alpha \Rightarrow \alpha_1 \vee \cdots \vee \alpha_n$，从而既减小了生成切片执行图的代价，又大大缩减了切片执行图的状态空间。

令某次切片执行的抽象准则为 V，当切片执行到达分支语句 "*if(c) A else B*" 时，设其部分最强后置条件为 α。那么，如果条件 $\widetilde{SP}_V\big(assume(c)\big)(\alpha) \neq False$ 成立，则切片执行保守地假设分支 A 可行；如果条件 $\widetilde{SP}_V\big(assume(\neg c)\big)(\alpha) \neq False$ 成立，则切片执行保守地假设分支 B 可行。因此对每个程序位置 s 而言，它的部分最强后置条件 α 实际上决定了从该位置出发的所有执行路径的可行性。对从 s 出发的所有可行执行路径来说，我们能够求得它们在 s 处对应于抽象准则 V 中变量的最弱前置条件 pwp，使得只要 V 中的程序变量在 s 处满足条件 pwp，就能保证这些执行路径的可行性。显然，由于 pwp 是使得这些执行路径可行的 s 处的最弱条件，因此我们有公式 $\alpha \Rightarrow pwp$ 成立。考虑上文中切片执行对程序位置 s 的第 i 次访问，令其部分最强后置条件为 α_i。设从 s 出发且满足 α_i 的所有可行执行路径对应于 s 处的最弱前置条件为 pwp_i，那么在切片执行过程中我们可以用 pwp_i 代替 α_i 来描述在程序位置 s 处已经搜索过的所有可行执行路径。由于 $\alpha_i \Rightarrow pwp_i$，因此切片执行过程中可能存在 $\alpha \Rightarrow pwp_1 \vee \cdots \vee pwp_n$ 但 $\alpha \not\Rightarrow \alpha_1 \vee \cdots \vee \alpha_n$ 的情况。也就是说，引入了这样的最弱前置条件后，能够在保证切片执行正确性的前提下（本章会给出证明），进一步缩减切片执行代价和生成的切片执行图的状态空间。

```
1:  if (*)
2:      if (x <= 5) return;
3:  else if (x <= 2) return;
4:  if (x > 0) return;
5:  else ERROR:
```

图 4.1 部分最弱前置条件示例程序代码

例如，考察图 4.1 所示的程序代码（其中 "*" 表示无关条件，即它的两个分

支都可行），假设切片执行只考虑变量 x，那么当执行了标号为 1 和 2 的语句到达标号 4 所示的程序位置时，其部分最强后置条件为 $x>5$，在该条件下 *ERROR* 标号是不可到达的。而从标号 4 所示的程序位置出发的所有不到达 *ERROR* 标号的执行路径的最弱前置条件为 $x>0$，因此当切片执行执行了标号为 1 和 3 的语句再次到达标号 4 所示的程序位置时，其部分最强后置条件 $x>2$ 蕴含了最弱前置条件 $x>0$ 但不能蕴含之前的部分最强后置条件 $x>5$，因此如果用条件 $x>0$ 代替条件 $x>5$，我们就能避免再次搜索从标号 4 所示的程序位置出发的所有执行路径，同时也减小了生成模型的状态空间。

由于切片执行只考虑部分程序变量，因此本章提出了部分最弱前置条件的概念，它是对传统最弱前置条件的保守近似。将部分最弱前置条件应用于切片执行时，由于基于搜索复用的切片执行过程按深度优先的方式遍历从任意程序位置出发的所有执行路径，因此在搜索回溯时我们可以将当前程序位置的部分最弱前置条件“传递”到其前驱程序位置，所以部分最弱前置条件与切片执行过程的结合是很自然的。

与传统的最弱前置条件一样，部分最弱前置条件也可能面临公式规模的指数爆炸问题[83, 84]，即最弱前置条件的公式规模随着程序规模的增长成指数增长。当前的解决方法是通过变量重命名，预先将程序变换为被动形式[83, 84]。而本章提出的方法则能够在切片执行过程进行的同时，将程序变换为被动形式，并进行部分最弱前置条件的计算。另外，本章亦给出了部分最弱前置条件计算过程中对指针和别名的支持方法。

我们基于 *openssl-0.9.6c* 源程序对 SSL 协议的初始握手协议进行了验证，实验结果显示，通过将部分最弱前置条件引入切片执行过程，切片执行代价和抽象的程序模型的状态空间被缩减至大约 1/10。

4.1 部分最弱前置条件

根据传统最弱前置条件的定义，条件 Q 相对于迁移语句 t 的最弱前置条件记为 $WP(t)(Q)$，在 t 执行之前的程序位置处，定义了使得 t 执行终止后条件 Q 成立的最弱条件。也就是说，如果在执行 t 之前不满足条件 $WP(t)(Q)$，那么 t 执行终止后条件 Q 一定不满足。迁移语句 t 的两种情况，即赋值语句和 assume 语句，对应的最弱前置条件定义如下[59, 85]：

$$WP(x:=e)(Q)=Q[e/x] \tag{4.1}$$

$$WP(assume(c))(Q)=(c\Rightarrow Q) \tag{4.2}$$

其中，$Q[e/x]$ 表示将公式 Q 中的变量 x 的所有自由出现都用表达式 e 替换得到的新公式。

为了计算分支语句“*if*(*c*) {*P*} else {*Q*}”的最弱前置条件，其中 P 和 Q 分别为分支语句的两个分支所必须满足的后置条件，我们需要合并两个最弱前置条件 $WP(assume(c))(P)$ 和 $WP(assume(\neg c))(Q)$。我们称这种合并为“并行组合（Parallel Composition）”，记为 $P \| Q$，其定义如下：

$$P \| Q = P \wedge Q \tag{4.3}$$

因此，上述分支语句“*if*(*c*) {*P*} else {*Q*}”的最弱前置条件为 $WP(assume(c))(P) \wedge WP(assume(\neg c))(Q)$。

传统的最弱前置条件指定了对所有程序变量的所有可能的合法赋值组合，而在基于切片执行的模型抽象和性质验证时，只有与性质相关的少部分程序变量被关注。因此，在变量抽象的基础上，我们提出了部分最弱前置条件的概念，从另一个方面描述程序的保守近似语义。在抽象准则 V 下，部分最弱前置条件记为 $\widetilde{WP}_V$，下面我们给出其定义。

定义 4.1 给定抽象准则 V 和一阶逻辑公式 Q，其中 $Vars(Q) \subseteq V$，则 Q 相对于赋值语句 $x := e$ 的部分最弱前置条件 $\widetilde{WP}_V(x := e)(Q)$ 定义为：

- 如果 $x := e$ 是抽象准则 V 下的完全相关赋值语句，那么
  $\widetilde{WP}_V(x := e)(Q) = Q[e/x]$
- 如果 $x := e$ 是抽象准则 V 下的部分相关赋值语句，那么
  $\widetilde{WP}_V(x := e)(Q) = \exists\theta.Q[\theta/x]$
- 如果 $x := e$ 是抽象准则 V 下的无关赋值语句，那么
  $\widetilde{WP}_V(x := e)(Q) = Q$

我们定义的部分最弱前置条件是面向切片执行的，用于计算切片执行下的一组可行执行路径对应的最弱前置条件。对于定义的第二种情况，由于切片执行部分相关赋值语句 $x := e$ 时将引入一个 Skolem 常量 θ 来表示变量 x 可取值为任意值，因此从语句 $x := e$ 后的程序位置出发的所有执行路径的可行性与 x 无关。也就是说，在切片执行赋值语句 $x := e$ 之前，无论 x 取什么值，都不影响上述执行路径的可行性。所以，我们保守地定义公式 Q 相对于部分相关赋值语句 $x := e$ 的部分最弱前置条件为 $\exists\theta.Q[\theta/x]$。在下文中我们还会就该点进行讨论，并给出严格的证明。

例如，$\widetilde{WP}_{\{x\}}(x := y)(x > 0) = \exists\theta.(\theta > 0) = True$，即如果某条执行路径 p 的可行性取决于公式 $x > 0$ 是否成立，则由于基于抽象准则 $V = \{x\}$ 切片执行赋值语句 $x := y$ 时将引入一个 Skolem 常量表示变量 x 可取值为任意值，因此 p 一定可行（因为 x 的某些取值能够满足公式 $x > 0$）。也就是说，在切片执行 $x := y$ 之前无论 x 取什么值，在切片执行之后都能使路径 p 可行，从而切片执行之前最弱的条件为 *True*。

定义 4.2 给定抽象准则 V 和一阶逻辑公式 Q，其中 $Vars(Q) \subseteq V$，则 Q 相对于 assume 语句 *asstume*(*c*)的部分最弱前置条件 $\widetilde{WP}_V(assume(c))(Q)$ 定义为：

- 如果 *assume*(*c*)是抽象准则 V 下的相关 assume 语句，那么
  $\widetilde{WP}_V(assume(c))(Q) = (c \Rightarrow Q)$
- 如果 *assume*(*c*)是抽象准则 V 下的无关 assume 语句，那么
  $\widetilde{WP}_V(assume(c))(Q) = Q$

定义 4.3 两个部分最弱前置条件 P 与 Q 的并行组合 $P \| Q$ 定义如公式(4.4)所示。

$$P \| Q = P \wedge Q \tag{4.4}$$

与部分最强后置条件一样，部分最弱前置条件也是单调函数，如引理 4.1 所示。

引理 4.1 对任意抽象准则 V 和任意迁移语句 t，若 $P \Rightarrow Q$，则 $\widetilde{WP}_V(t)(P) \Rightarrow \widetilde{WP}_V(t)(Q)$。

证明：对迁移 t 分下列情况讨论：

- 若 t 为抽象准则 V 下的完全相关赋值语句 $x := e$，则 $\widetilde{WP}_V(t)(P) = P[e/x]$ 且 $\widetilde{WP}_V(t)(Q) = Q[e/x]$。由于 $P \Rightarrow Q$，故有 $P[e/x] \Rightarrow Q[e/x]$；
- 若 t 为抽象准则 V 下的部分相关赋值语句 $x := e$，则 $\widetilde{WP}_V(t)(P) = \exists\theta.P[\theta/x]$ 且 $\widetilde{WP}_V(t)(Q) = \exists\theta.Q[\theta/x]$。由于 $P \Rightarrow Q$，故有 $P[\theta/x] \Rightarrow Q[\theta/x]$，从而 $\exists\theta.P[\theta/x] \Rightarrow \exists\theta.Q[\theta/x]$；
- 若 t 为抽象准则 V 下的无关赋值语句 $x := e$，则 $\widetilde{WP}_V(t)(P) = P$、$\widetilde{WP}_V(t)(Q) = Q$，从而 $\widetilde{WP}_V(t)(P) \Rightarrow \widetilde{WP}_V(t)(Q)$；
- 若 t 为抽象准则 V 下的相关 assume 语句 $assume(c)$，则 $\widetilde{WP}_V(t)(P) = (c \Rightarrow P)$、$\widetilde{WP}_V(t)(Q) = (c \Rightarrow Q)$，根据 $P \Rightarrow Q$ 知 $(\neg c \vee P) \Rightarrow (\neg c \vee Q)$，从而 $\widetilde{WP}_V(t)(P) \Rightarrow \widetilde{WP}_V(t)(Q)$；
- 若 t 为抽象准则 V 下的无关 assume 语句 $assume(c)$，则 $\widetilde{WP}_V(t)(P) = P$、$\widetilde{WP}_V(t)(Q) = Q$，从而 $\widetilde{WP}_V(t)(P) \Rightarrow \widetilde{WP}_V(t)(Q)$。

综上，引理证毕。 ■

4.2 部分最弱前置条件在切片执行中的应用

采用了部分最弱前置条件的切片执行过程如图 4.2 所示，其中用方框标注出来的语句是与部分最弱前置条件相关的。为简洁清晰起见，该过程没有集成第三章介绍的搜索复用框架，但由于仍采用深度优先的搜索策略来遍历所有程序执行路径，因此搜索复用框架也能够自然地与之集成。

为了在切片执行框架中应用部分最弱前置条件，我们将其生成的切片执行图 $SEG=\langle \Psi, \longrightarrow \rangle$ 的状态集合 Ψ 的定义从第二章的 $\Psi \subseteq S \times [\![PSP]\!]$ 延拓到 $\Psi \subseteq S \times ([\![PSP]\!] \cup [\![PWP]\!])$，其中 $[\![PWP]\!]$ 代表所有部分最弱前置条件的全集，从而在延拓后的切片执行图中，某些状态可能基于部分最强后置条件描述，而某些状态可能基于部分最弱前置条件描述。

```
0  Initially: Explore(⟨s0, True⟩);
Explore(ψ)
{
1  let ψ = ⟨s, α⟩
2  pwp := True;
3  for all t and s' such that ⟨s, t, s'⟩ ∈ Δ do {
4      let ⟨s', α1⟩, ···, ⟨s', αn⟩ ∈ Ψ be all states till now in SEG
                corresponding to the location s';
5      let α' = SP̃V(t)(α);
6      if (α' ⇒ α1 ∨ ··· ∨ αn) {
7          let A be the minimal subset of {α1, ···, αn}
                                such that α' ⇒ ∨_{αi∈A} αi
8          for all αi ∈ A set ψ --t--> ⟨s', αi⟩;
9          let pwp' = ∨_{αi∈A} αi;
10     } else {
11         add state ψ' = ⟨s', α'⟩ to Ψ and set ψ --t--> ψ';
12         let pwp' = Explore(ψ');
13     }
14     pwp := pwp ‖ WP̃V(t)(pwp');
15 }
16 for state ψ = ⟨s, α⟩, set α := pwp;
17 return pwp;
}
```

图 4.2 集成了部分最弱前置条件的切片执行过程

图 4.2 所示的切片执行过程将递归地调用函数 Explore，以遍历切片执行图中某状态的后继状态。初始时，在切片执行过程第 0 行调用函数 Explore 遍历切片执行图的初始状态 $\langle s_0, True \rangle$，其中 $True$ 为部分最强后置条件。当调用函数 Explore 遍历某状态 ψ 时，令 $\psi=\langle s, \alpha \rangle$，其中 α 是该状态的部分最强后置条件，我们引入一个局部变量 pwp，用于记录对应于 α 的部分最弱前置条件，初始时我们令 $pwp=True$（第 2 行）。在过程第 3 行中，对给定的程序模型 $CP=\langle S, s_0, T, \Delta \rangle$ 中当前程序位置 s 的每个后继程序位置 s'，我们计算出对应的部分最强后置条件 α'

（第 5 行）。根据当前生成的切片执行图中对应于程序位置 s' 的所有状态 $\langle s',\alpha_1\rangle,\cdots,\langle s',\alpha_n\rangle\in\Psi$（第 4 行），我们可以得到程序位置 s' 处的切片执行上下文为 $SEC(s')=\{\alpha_1,\cdots,\alpha_n\}$。如果第 6 行的蕴含条件 $\alpha'\Rightarrow\alpha_1\vee\cdots\vee\alpha_n$ 成立，则我们保守地假设 $pwp'=\vee_{\alpha_i\in A}\alpha_i$ 为对应于状态 $\langle s',\alpha'\rangle$ 的部分最弱前置条件，其中集合 A 为使得蕴含式成立的 $SEC(s')$ 的极小子集（见过程第 7 行）。注意，如果 $\alpha'=False$，则集合 A 为空，从而不会有迁移被加入到切片执行图中（第 8 行），而且 $pwp'=False$（第 9 行）。另外，可能某个 $\alpha_i\in A$ 并不是程序位置 s' 处的部分最弱前置条件，而是该处的部分最强后置条件，这是因为只有当状态 $\langle s',\alpha_i\rangle$ 的所有后继状态被遍历完成后才将最强后置条件 α_i 替换为相应的部分最弱前置条件（参见第 16 行），而由于切片执行图中圈的存在导致状态 $\langle s',\alpha_i\rangle$ 的后继状态可能并没有被遍历完成。但是，稍后我们就会证明，这种处理方法是正确的。

另一方面，如果第 6 行的蕴含条件不满足，则过程 Explore 会被调用，以遍历 ψ 的后继状态 $\langle s',\alpha'\rangle$ 并生成相应的部分最弱前置条件 pwp'。通过第 14 行的代码，当前状态 $\langle s,\alpha\rangle$ 的所有后继状态的部分最弱前置条件都被并行组合起来，从而得到了 $\langle s,\alpha\rangle$ 对应的部分最弱前置条件 pwp。注意，如果当前程序位置 s 在程序模型 $CP=\langle S,s_0,T,\Delta\rangle$ 中没有后继状态，那么当前状态 $\langle s,\alpha\rangle$ 的部分最弱前置条件将为 $True$。最后在第 16 行，当前状态 $\langle s,\alpha\rangle$ 中的部分最强后置条件 α 将被相应的部分最弱前置条件 pwp 替换。最后，当前状态的部分最弱前置条件 pwp 被返回给其前驱状态，以便与前驱状态的其它后继状态的部分最弱前置条件进行并行组合。

下面我们通过几个定理证明上面的改进切片执行过程的正确性。

定理 4.1 图 4.2 所示切片执行过程的第 16 行中，对于状态 $\psi=\langle s,\alpha\rangle$ 的部分最强后置条件 α 和其相应的部分最弱前置条件 pwp 而言，皆有 $\alpha\Rightarrow pwp$ 成立。

证明：基于归纳法，如果程序位置 s 在程序模型 $CP=\langle S,s_0,T,\Delta\rangle$ 中没有后继状态，则 $pwp=True$，从而 $\alpha\Rightarrow pwp$ 成立。否则的话，我们先证明如下两个公式：

1) $\alpha\Rightarrow\widetilde{WP}_V(x:=e)\left(\widetilde{SP}_V(x:=e)(\alpha)\right)$，我们分如下三种情况证明：
   - 若 $x:=e$ 是完全相关的赋值语句，则根据定义知 $\widetilde{WP}_V(x:=e)\left(\widetilde{SP}_V(x:=e)(\alpha)\right)=\exists x'.\alpha[x'/x]\wedge(e=e[x'/x])$，若公式 α 为真，则 x' 取值为 x 的值可令公式 $\exists x'.\alpha[x'/x]\wedge(e=e[x'/x])$ 为真，故 $\alpha\Rightarrow\widetilde{WP}_V(x:=e)\left(\widetilde{SP}_V(x:=e)(\alpha)\right)$ 成立；
   - 若 $x:=e$ 是部分相关的赋值语句，则根据定义知 $\widetilde{WP}_V(x:=e)\left(\widetilde{SP}_V(x:=e)(\alpha)\right)=\exists x'.\alpha[x'/x]$，显然 $\alpha\Rightarrow\exists x'.\alpha[x'/x]$，故 $\alpha\Rightarrow\widetilde{WP}_V(x:=e)\left(\widetilde{SP}_V(x:=e)(\alpha)\right)$ 成立；
   - 若 $x:=e$ 是无关赋值语句，则 $\widetilde{WP}_V(x:=e)\left(\widetilde{SP}_V(x:=e)(\alpha)\right)=\alpha$，从而

$\alpha \Rightarrow \widetilde{WP}_V(x:=e)\left(\widetilde{SP}_V(x:=e)(\alpha)\right)$成立。

2) $\alpha \Rightarrow \alpha_1 \wedge \alpha_2$，其中 $\alpha_1 = \widetilde{WP}_V(assume(c))\left(\widetilde{SP}_V(assume(c))(\alpha)\right)$、$\alpha_2 = \widetilde{WP}_V(assume(\neg c))\left(\widetilde{SP}_V(assume(\neg c))(\alpha)\right)$。我们分两种情况证明：

- 若 *assume*(*c*) 与 *assume*($\neg c$) 为相关 assume 语句，则根据定义知 $\alpha_1 = (c \Rightarrow (\alpha \wedge c))$、$\alpha_2 = (\neg c \Rightarrow (\alpha \wedge \neg c))$，通过命题逻辑公式的等价变换易知$\alpha \Leftrightarrow \alpha_1 \wedge \alpha_2$；
- 若 *assume*(*c*) 与 *assume*($\neg c$) 为无关 assume 语句，则$\alpha_1 = \alpha$、$\alpha_2 = \alpha$，从而$\alpha \Rightarrow \alpha_1 \wedge \alpha_2$成立。

对程序位置 s 在程序模型 $CP = \langle S, s_0, T, \Delta \rangle$ 中的每个后继程序位置 s' 及其相应迁移 $\langle s,t,s' \rangle \in \Delta$，令 $\alpha' = \widetilde{SP}_V(t)(\alpha)$。如果图 4.2 中第 6 行的蕴含条件 $\alpha' \Rightarrow \alpha_1 \vee \cdots \vee \alpha_n$成立，则根据第 9 行定义的 *pwp'*，我们有$\alpha' \Rightarrow pwp'$成立；如果蕴含条件$\alpha' \Rightarrow \alpha_1 \vee \cdots \vee \alpha_n$不成立，则根据归纳法我们假设$\alpha' \Rightarrow pwp'$成立，其中 *pwp'* 是函数 Explore 返回的对应于α'的部分最弱前置条件。下面我们分如下两种情况证明$\alpha \Rightarrow pwp$成立。

第一种情况，如果 s 只有一条后继迁移$\langle s,t,s' \rangle \in \Delta$且 t 为赋值语句，则根据假设$\alpha' \Rightarrow pwp'$成立，其中$\alpha' = \widetilde{SP}_V(t)(\alpha)$，而 *pwp'* 则是对应于$\alpha'$的部分最弱前置条件。根据上述公式 1)，我们有 $\alpha \Rightarrow \widetilde{WP}_V(t)(\alpha')$，又由于 $\alpha' \Rightarrow pwp'$ 且 $pwp = \widetilde{WP}_V(t)(pwp')$，根据引理 4.1 知$\alpha \Rightarrow pwp$成立。

第二种情况，如果 s 有两条后继迁移$\langle s,t_1,s_1 \rangle \in \Delta$、$\langle s,t_2,s_2 \rangle \in \Delta$，且$t_1$为 assume 语句 *assume*($c$)、$t_2$ 为 assume 语句 *assume*($\neg c$)，则根据假设 $\alpha_1 \Rightarrow pwp_1$ 及 $\alpha_2 \Rightarrow pwp_2$成立，其中$\alpha_1 = \widetilde{SP}_V(t_1)(\alpha)$、$\alpha_2 = \widetilde{SP}_V(t_2)(\alpha)$，$pwp_1$与 pwp_2分别为α_1与 α_2 对应的部分最弱前置条件。根据上述公式 2)及引理 4.1，我们有 $\alpha \Rightarrow \widetilde{WP}_V(t_1)(\alpha_1) \wedge \widetilde{WP}_V(t_2)(\alpha_2) \Rightarrow \widetilde{WP}_V(t_1)(pwp_1) \wedge \widetilde{WP}_V(t_2)(pwp_2) = pwp$。

由于程序模型 $CP = \langle S, s_0, T, \Delta \rangle$ 中只存在两种类型的迁移语句，根据归纳法，我们就证明了对切片执行图中的所有状态都有$\alpha \Rightarrow pwp$。 ■

定理 4.2 在图 4.2 所示的切片执行过程所产生的切片执行图 $SEG = \langle \Psi, \longrightarrow \rangle$ 中，如果 C 程序模型 $CP = \langle S, s_0, T, \Delta \rangle$ 中的任意一条执行路径 $p = t_1 t_2 \cdots t_n$ 满足 $\widetilde{SP}_V(p)(True) \neq False$，则 SEG 中存在 $n+1$ 个状态 $\psi_i = \langle s_i, \alpha_i \rangle \in \Psi$（$i: 0 \le i \le n$），使得 $\langle s_{i-1}, t_i, s_i \rangle \in \Delta$（$i: 0 \le i \le n$）且 $\psi_0 \xrightarrow{t_1} \psi_1 \xrightarrow{t_2} \cdots \xrightarrow{t_n} \psi_n$。

证明：基于反证法，我们假设所产生的切片执行图 $SEG = \langle \Psi, \longrightarrow \rangle$ 中并不包含所给路径 $\psi_0 \xrightarrow{t_1} \psi_1 \xrightarrow{t_2} \cdots \xrightarrow{t_n} \psi_n$，那么对于包含在 $SEG = \langle \Psi, \longrightarrow \rangle$ 中的该路径的最长子路径 $\psi_0 \xrightarrow{t_1} \cdots \xrightarrow{t_k} \psi_k$（其中 $k < n$），我们设切片执行到状态 ψ_k 时的部分最强后置条件为 α_k。根据图 4.2 所示的切片执行过程，我们可以得出

$\widetilde{SP}_V(t_{k+1})(\alpha_k)=False$，否则，必定存在某个状态$\psi'_{k+1}\in\Psi$，使得$\psi_k\xrightarrow{t_{k+1}}\psi'_{k+1}$。进而我们有$\widetilde{SP}_V(t_{k+1}\cdots t_n)(\alpha_k)=False$。考虑切片执行图中存在的迁移$\psi_{k-1}\xrightarrow{t_k}\psi_k$，其中$\psi_{k-1}=\langle s_{k-1},\alpha_{k-1}\rangle$，如果$\widetilde{SP}_V(t_k)(\alpha_{k-1})=\alpha_k$（对应于切片执行过程第 6 行蕴含条件$\alpha_k\Rightarrow SEC(s_k)$不满足的情况），那么我们得到$\widetilde{SP}_V(t_kt_{k+1}\cdots t_n)(\alpha_{k-1})=False$。另一方面，如果$\widetilde{SP}_V(t_k)(\alpha_{k-1})\neq\alpha_k$，令$\alpha'_k=\widetilde{SP}_V(t_k)(\alpha_{k-1})$，则$\alpha'_k\Rightarrow SEC(s_k)$且$\alpha_k\in\mathrm{A}$，下面我们证明$\widetilde{SP}_V(t_kt_{k+1}\cdots t_n)(\alpha_{k-1})=False$仍然成立。

任取$\alpha''\in\mathrm{A}$，如果α''是程序位置s_k处的部分最强后置条件，则我们可以归纳地假设$\widetilde{SP}_V(t_{k+1}\cdots t_n)(\alpha'')=False$，否则必定存在从程序位置$s_k$出发的一条可行执行路径$t_{k+1}\cdots t_n$，从而与假设相矛盾。另一方面，如果$\alpha''$是程序位置$s_k$处的部分最弱前置条件，根据图 4.2 所示的切片执行过程我们知道，α''概括了使得从程序位置s_k出发的所有可行执行路径的最弱前置条件，也就是说，如果某执行路径p'可行，则必有$\widetilde{SP}_V(p')(\alpha'')\neq False$。反过来，由于执行路径$t_{k+1}\cdots t_n$不可行，因此必有$\widetilde{SP}_V(t_{k+1}\cdots t_n)(\alpha'')=False$。从而，不论集合A中的元素$\alpha''$是部分最强后置条件还是部分最弱前置条件，都有$\widetilde{SP}_V(t_{k+1}\cdots t_n)(\alpha'')=False$。由于$\alpha'_k\Rightarrow\bigvee_{\alpha_i\in\mathrm{A}}\alpha_i$，根据引理 4.1 知$\widetilde{SP}_V(t_kt_{k+1}\cdots t_n)(\alpha_{k-1})=False$。

归纳地，我们最终得到$\widetilde{SP}_V(t_1t_2\cdots t_n)(True)=False$，这与定理前提条件相矛盾，故证毕。■

推论 4.1 图 4.2 所示的集成了部分最弱前置条件的切片执行过程仍是可靠的，即如果对生成的切片执行图进行模型检验没有发现违背给定时序安全性质的反例路径，则在原程序中也不存在违背性质的反例路径。

证明：反证法，假设程序中存在一条违背性质的反例路径p，但p并不包含在生成的切片执行图中。根据最强后置条件的定义，有$SP(p)(True)\neq False$，再根据第二章定理 2.1 知$\widetilde{SP}_V(p)(True)\neq False$，而根据本章定理 4.2 知，生成的切片执行图中必存在一条对应于 p 的路径，因此在模型检验时将找到一条违背性质的反例路径，得出矛盾，故定理得证。■

此外，当蕴含条件$\alpha'\Rightarrow\alpha_1\vee\cdots\vee\alpha_n$成立时，切片执行就无需对状态$\langle s',\alpha'\rangle$的后继状态和迁移进行搜索，从而能够缩减状态空间。根据定理 4.1 知α_i所描述的条件比其相应的部分最弱前置条件pwp_i强，因此可能$\alpha'\Rightarrow\alpha_1\vee\cdots\vee pwp_i\vee\cdots\vee\alpha_n$但$\alpha'\not\Rightarrow\alpha_1\vee\cdots\vee\alpha_i\vee\cdots\vee\alpha_n$。也就是说，在保证可靠性的前提下（定理 4.2），集成部分最弱前置条件的切片执行的代价及所生成的切片执行图的状态空间将会减小，本章实验部分将证实该结论。

4.3 避免部分最弱前置条件公式规模的指数增长

4.3.1 最弱前置条件公式规模的指数增长问题与解决方法

按照最弱前置条件定义的计算方法，最弱前置条件的公式规模随着程序规模的增加可能呈指数量级的速度增长，程序中的赋值语句与分支语句都能够导致公式规模的指数增长[83, 84]。

对赋值语句 $x := e$，根据定义有 $WP(x := e)(Q) = Q[e/x]$，即将公式 Q 中所有 x 的自由出现都用表达式 e 替换，这样如果 Q 中包含 x 的 n 次自由出现，表达式 $Q[e/x]$ 中就会出现 n 个表达式 e，从而导致公式规模的迅速增长。更为严重的是，如果一条执行路径中的多个赋值语句都包含有依赖关系的程序变量，则公式规模将呈指数增长。例如对图 4.3 所示的由具有变量依赖关系的赋值语句组成的一条执行路径 p，其相对于 $x>0$ 的最弱前置条件为 $x+\cdots+x>0$，公式中一共包含 2^n 个 x。

l_1: $x := x + x$;
l_2: $x := x + x$;
.....;
l_n: $x := x + x$;

图 4.3 具有变量依赖关系的赋值语句组成的执行路径

对于分支语句 "*if* (*c*) *A else B*;"，若它的两个分支到达相同的程序位置，则根据定义有 $WP(if(c)\ A\ else\ B)(Q) = (c \Rightarrow WP(A)(Q)) \wedge (\neg c \Rightarrow WP(B)(Q))$，从而其最弱前置条件中出现了两次 Q（当然，Q 可能被 A 和 B 中包含的赋值语句修改）。更为严重的是，多个具有变量依赖关系的分支语句的顺序组合将使得最弱前置条件的公式规模呈指数增长。

l_1: $x_1 := x_0 + x_0$;
l_2: $x_2 := x_1 + x_1$;
.....;
l_n: $x_n := x_{n-1} + x_{n-1}$;

图 4.4 转换后被动化形式的执行路径

解决最弱前置条件的公式规模随程序规模的增长呈指数增长问题的一个方法，就是所谓的程序被动化[83, 84]，即通过对程序变量的重命名，将赋值语句转化为等价的 assume 语句，从而可以将相同的子公式合并。例如将图 4.3 中所示的程序可以转化为图 4.4 所示的程序，转化时先分析变量引用的依赖关系，再对变量 x 进行合适的重命名。对转化后的执行路径，其相对于 $x>0$ 的最弱前置条件为

$\forall x_0,\cdots,x_n : x_1 = x_0 + x_0 \Rightarrow \left(x_2 = x_1 + x_1 \Rightarrow \left(\cdots \Rightarrow \left(x_n = x_{n-1} + x_{n-1} \Rightarrow \left(x_n = x \Rightarrow x > 0\right)\right)\right)\right)$，其包含的变量数量降至$3n-2$个，其详细的过程及正确性证明请参见文献[84]。

4.3.2 紧凑的部分最弱前置条件表示和计算方法

部分最弱前置条件是传统最弱前置条件的变种，因此它也会面临公式规模的指数爆炸问题。为了解决该问题，我们将部分最弱前置条件表示为二元偶$\langle\sigma,\omega\rangle$的形式，其中：

- $\sigma : [\![Var]\!] \mapsto [\![Var]\!]$称为命名映射，它将程序变量映射到其当前的名字，这里$[\![Var]\!]$表示所有程序变量的全集，同时我们将σ按常规延拓到表达式和公式集合，初始时$\sigma = \bot$，其中$\bot \triangleq \lambda x.x$；
- ω为一个一阶逻辑公式，描述了当前的部分最弱前置条件，初始时$\omega = True$。

直观地，我们在计算一条语句的部分最弱前置条件时，先基于命名映射σ将该语句中的变量重命名为合适的名字，即将该语句被动化，再基于ω计算其部分最弱前置条件。

定义 4.4 赋值语句的部分最弱前置条件$\widetilde{WP}_V\left(x := e\right)\left(\langle\sigma,\omega\rangle\right)$定义为：

- $\left\langle\sigma[x \to x'], \left(\sigma(x) = \sigma[x \to x'](e)\right) \Rightarrow \omega\right\rangle$，当且仅当$x := e$为抽象准则$V$下的完全相关赋值语句；
- $\left\langle\sigma[x \to x'], \exists \sigma(x).\omega\right\rangle$，当且仅当$x := e$为抽象准则$V$下部分相关赋值语句；
- $\langle\sigma,\omega\rangle$，当且仅当$x := e$为抽象准则$V$下的无关赋值语句。

其中x'是一个新鲜变量，函数$\sigma[x \to x']$定义为公式(4.5)所示：

$$\sigma[x \to x'](y) = \begin{cases} \sigma(y) & if\ y \neq x \\ x' & if\ y = x \end{cases} \tag{4.5}$$

直观地来讲，如果$x := e$是抽象准则V下的完全相关赋值语句，则先基于命名映射σ将$x := e$重命名后得到赋值语句$\sigma(x) := \sigma[x \to x'](e)$，其中$\sigma[x \to x']$表示在$\sigma$中为变量$x$引入一个新的名字$x'$后得到的新命名映射，$x'$描述了$x$在赋值语句$x := e$之前的取值（相应地，我们用$x$当前的名字$\sigma(x)$描述该赋值语句执行后$x$的取值）。如果表达式$e$中包含变量$x$，则该变量$x$引用了赋值语句$x := e$之前的值，因此需要基于$\sigma[x \to x']$对表达式$e$进行重命名。由于每个重命名后的变量都不会被赋值两次，因此我们可将赋值语句$\sigma(x) := \sigma[x \to x'](e)$转换为等价的 assume 语句$assume\left(\sigma(x) = \sigma[x \to x'](e)\right)$，该 assume 语句相对于$\omega$的部分最弱前置条件为$\widetilde{WP}_V\left(assume\left(\sigma(x) = \sigma[x \to x'](e)\right)\right)(\omega) = \left(\sigma(x) = \sigma[x \to x'](e)\right) \Rightarrow \omega$。例如，$\widetilde{WP}_{\{x,y\}}\left(x := x + y\right)\left(\left\langle\{x \to x_1, y \to y_1\}, x_1 > 0\right\rangle\right) = \left\langle\{x \to x_2, y \to y_1\}, \left(x_1 = x_2 + y_1 \Rightarrow x_1 > 0\right)\right\rangle$，

赋值语句 $x:=x+y$ 左边的 x 需要使用当前的名字 x_1 替换，而右边的 x 则需要使用赋值语句执行前 x 的名字 x_2 替换。

如果 $x:=e$ 是抽象准则 V 下的部分相关赋值语句，由于 ω 中的变量 x 被命名为变量 $\sigma(x)$，且该命名在后续的计算中不再使用（因为 σ 被更新为 $\sigma[x\to x']$），因此没有必要引入 Skolem 常量，而是直接得到部分最弱前置条件 $\exists\sigma(x).\omega$。例如，$\widetilde{WP}_{\{x\}}(x:=x+y)(\langle\{x\to x_1\},x_1>0\rangle)=\langle\{x\to x_2\},\exists x_1.x_1>0\rangle$。

如果有多个赋值语句串行组合，根据等价公式 $q_1\Rightarrow(q_2\Rightarrow Q)\equiv(q_1\wedge q_2)\Rightarrow Q$，我们可以将蕴含连接词（$\Rightarrow$）的数量降至只有一个。

从定义 4.4 可以看到，当计算赋值语句的部分最弱前置条件时，不会进行变量的替换，因此赋值语句不会导致部分最弱前置条件公式规模的指数爆炸。

定义 4.5 assume 语句的部分最弱前置条件 $\widetilde{WP}_V(assume(c))(\langle\sigma,\omega\rangle)$ 定义为：

- $\langle\sigma,\sigma(c)\Rightarrow w\rangle$，当且仅当 $assume(c)$ 为抽象准则 V 下的相关 assume 语句；
- $\langle\sigma,\omega\rangle$，当且仅当 $assume(c)$ 为抽象准则 V 下的无关 assume 语句。

Assume 语句的部分最弱前置条件的计算比较直观，由于 assume 语句不改变任意变量的取值，因此命名映射 σ 保持不变。

我们定义 $dom(\sigma)$ 为变量命名映射 σ 的定义域，即所有被 σ 映射到新变量名的变量的集合。例如，若 $\sigma=\{x\to x_2,y\to y_1\}$，则 $dom(\sigma)=\{x,y\}$。根据定义，我们有 $dom(\bot)=\varnothing$。

定义 4.6 部分最弱前置条件的并行组合 $\langle\sigma_1,\omega_1\rangle\|\langle\sigma_2,\omega_2\rangle$ 定义为：

- $\langle\sigma_2,\sigma_2(\omega_1)\wedge\omega_2\rangle$ 如果 $\sigma_1=\bot$，或者 $\langle\sigma_1,\omega_1\wedge\sigma_1(\omega_2)\rangle$ 如果 $\sigma_2=\bot$；
- $\langle\sigma,\omega\rangle$，其中 $dom(\sigma)\triangleq dom(\sigma_1)\cup dom(\sigma_2)$ 并且

$$\sigma(x)=\begin{cases}\sigma_1(x) & if\ x\in dom(\sigma_1)\wedge x\notin dom(\sigma_2)\\ \sigma_1(x) & if\ x\in dom(\sigma_1)\cap dom(\sigma_2)\wedge\sigma_1(x)=\sigma_2(x)\\ x' & if\ x\in dom(\sigma_1)\cap dom(\sigma_2)\wedge\sigma_1(x)\neq\sigma_2(x)\\ \sigma_2(x) & if\ x\notin dom(\sigma_1)\wedge x\in dom(\sigma_2)\end{cases}$$

其中 x' 为新鲜变量，且 $\omega=(Eqs(\sigma,\sigma_1)\Rightarrow\omega_1)\wedge(Eqs(\sigma,\sigma_2)\Rightarrow\omega_2)$，这里 $Eqs(\sigma,\sigma_1)$ 与 $Eqs(\sigma,\sigma_2)$ 定义为：

$$Eqs(\sigma,\sigma_i)\triangleq\bigwedge_{x\in dom(\sigma_i)\wedge\sigma(x)\neq\sigma_i(x)}\sigma(x)=\sigma_i(x)\qquad(i=1,2)$$

定义 4.6 从形式上看起来比较复杂，实际上其原理很直观。根据定义，两个部分最弱前置条件的并行组合总的来说是求 ω_1 和 ω_2 的合取公式，但由于 ω_1 和 ω_2 中同一个变量可能被命名为不同的名字，因此我们需要先处理变量的命名冲突。如果 $\sigma_1\neq\bot$ 且 $\sigma_2\neq\bot$，则 $\sigma(x)$ 的定义中根据不同的情况为变量 x 引入不同的名字。最

复杂的情况是当$x\in dom(\sigma_1)$、$x\in dom(\sigma_2)$且$\sigma_1(x)\neq\sigma_2(x)$时，需要为变量$x$引入一个新的名字$x'$，并在命名映射$\sigma_1$和$\sigma_2$中将变量$x$重命名为$x'$。消除命名冲突后，需要在并行组合前为两个部分最弱前置条件分别引入一系列赋值语句，以将命名前的变量赋值为命名后的变量。例如，如果变量x的两个名字$\sigma_1(x)$和$\sigma_2(x)$在σ中被重命名为x'，则需要在部分最弱前置条件ω_1和ω_2前分别引入赋值语句$x':=\sigma_1(x)$（即$Eqs(\sigma,\sigma_1)$）和$x':=\sigma_2(x)$（即$Eqs(\sigma,\sigma_2)$）。将赋值语句转换为 assume 语句后再对ω_1和ω_2求部分最弱前置条件即得到并行组合后的部分最弱前置条件$\omega=(Eqs(\sigma,\sigma_1)\Rightarrow\omega_1)\wedge(Eqs(\sigma,\sigma_2)\Rightarrow\omega_2)$。例如，$\langle x\to x_2,x_1=x_2+1\Rightarrow x_1>0\rangle\|\langle x\to x_1,x_1>0\rangle=\langle x\to x',(x'=x_1\Rightarrow x_1>0)\wedge(x'=x_2\Rightarrow(x_1=x_2+1\Rightarrow x_1>0))\rangle$。

此外，当两个部分最弱前置条件并行组合时，即$\langle\sigma,\omega\rangle=\langle\sigma_1,\omega_1\rangle\|\langle\sigma_2,\omega_2\rangle$，如果$\omega$形如$(e_1\Rightarrow Q)\wedge(e_2\Rightarrow Q)$，即$\omega_1$和$\omega_2$具有共同的表达式$Q$，则根据等价变换公式$(e_1\Rightarrow Q)\wedge(e_2\Rightarrow Q)\equiv(e_1\vee e_2)\Rightarrow Q$，我们可以避免部分最弱前置条件$\omega$中表达式$Q$的多次重复。因此，我们也能够回避由分支语句导致的部分最弱前置条件公式规模爆炸问题。例如，$((x'=x_1\Rightarrow x_1>0)\wedge(x'=x_2\Rightarrow(x_1=x_2+1\Rightarrow x_1>0)))=((x'=x_1\vee(x'=x_2\wedge x_1=x_2+1))\Rightarrow x_1>0)$，从而避免了公式$x_1>0$的重复出现。

本节中，我们提出了另外一种表示和计算部分最弱前置条件的方法，定理 4.3 和定理 4.4 保证了该方法的正确性。

定理 4.3 令$\alpha_2=\widetilde{WP}_V(t)(\alpha_1)$，其中$\alpha_1$与$\alpha_2$都是一阶逻辑公式，令$\langle\sigma_2,\omega_2\rangle=\widetilde{WP}_V(t)(\langle\sigma_1,\omega_1\rangle)$。如果公式$\sigma_1(\alpha_1)\Leftrightarrow\forall x_1,\cdots,x_n:\omega_1$成立（其中$x_1,\cdots,x_n$是$\omega_1$中引入的所有新变量名），则公式$\sigma_2(\alpha_2)\Leftrightarrow\forall x_1,\cdots,x_n:\omega_2$必定成立（其中$x_1,\cdots,x_n$是$\omega_2$中引入的所有新变量名）。

证明：我们分情况证明，考虑迁移t的类型：

- 如果t是无关的赋值语句或无关的 assume 语句，则$\alpha_2=\alpha_1$、$\langle\sigma_2,\omega_2\rangle=\langle\sigma_1,\omega_1\rangle$，从而定理成立；
- 如果t是完全相关的赋值语句$x:=e$，则$\alpha_2=\alpha_1[e/x]$、$\sigma_2=\sigma_1[x\to x']$、$\omega_2=((\sigma_1(x)=\sigma_1[x\to x'](e))\Rightarrow\omega_1)$，从而有$\sigma_2(\alpha_2)=\sigma_1[x\to x'](\alpha_1[e/x])$。如果对所有程序变量的某组赋值$E$使得$\sigma_2(\alpha_2)$为*True*，则如果$x'$的取值使得$\sigma_1(x)=\sigma_1[x\to x'](e)$成立，就有$\sigma_1[x\to x'](\alpha_1[e/x])=\sigma_1(\alpha_1)$（因为$\alpha_1[e/x]$中只有表达式$e$包含变量$x$），从而$\omega_1$为*True*。也就是说，公式$\sigma_2(\alpha_2)\Rightarrow\forall x_1,\cdots,x_n:((\sigma_1(x)=\sigma_1[x\to x'](e))\Rightarrow\omega_1)$成立。另一方面，如果$E$使得$\forall x_1,\cdots,x_n:((\sigma_1(x)=\sigma_1[x\to x'](e))\Rightarrow\omega_1)$为*True*，则若公式$\sigma_1(x)=\sigma_1[x\to x'](e)$为*True*，就有$\sigma_1[x\to x'](\alpha_1[e/x])=\sigma_1(\alpha_1)$且$\omega_1$为*True*，从而$\sigma_2(\alpha_2)$为*True*。综上，有$\sigma_2(\alpha_2)\Leftrightarrow\forall x_1,\cdots,x_n:\omega_2$成立；

- 如果 t 是部分相关的赋值语句 $x:=e$，则 $\alpha_2=\exists\theta.\alpha_1[\theta/x]$、$\sigma_2=\sigma_1[x\to x']$、$\omega_2=\exists\sigma_1(x).\omega_1$，从而 $\sigma_2(\alpha_2)=\sigma_1[x\to x'](\exists\theta.\alpha_1[\theta/x])$。由于 $\exists\theta.\alpha_1[\theta/x]$ 中不包含 x，故 $\sigma_1[x\to x'](\exists\theta.\alpha_1[\theta/x])=\sigma_1(\exists\theta.\alpha_1[\theta/x])=\exists\theta.\sigma_1(\alpha_1[\theta/x])$，令 $\theta=\sigma_1(x)$ 得到 $\sigma_2(\alpha_2)=\exists\theta.\sigma_1(\alpha_1[\theta/x])=\exists\sigma_1(x).\sigma_1(\alpha_1)$，再根据定理条件 $\sigma_1(\alpha_1)\Leftrightarrow\forall x_1,\cdots,x_n:\omega_1$ 得到 $\sigma_2(\alpha_2)\Leftrightarrow\exists\sigma_1(x).\forall x_1,\cdots,x_n:\omega_1$，从而得到 $\sigma_2(\alpha_2)\Leftrightarrow\forall x_1,\cdots,x_n:\omega_2$；
- 如果 t 是相关的 assume 语句 $assume(c)$，则 $\alpha_2=(c\Rightarrow\alpha_1)$、$\sigma_2=\sigma_1$、$\omega_2=(\sigma_1(c)\Rightarrow\omega_1)$，从而 $\sigma_2(\alpha_2)=\sigma_1(c\Rightarrow\alpha_1)=(\sigma_1(c)\Rightarrow\sigma_1(\alpha_1))$。根据定理条件 $\sigma_1(\alpha_1)\Leftrightarrow\forall x_1,\cdots,x_n:\omega_1$ 得到 $\sigma_2(\alpha_2)\Leftrightarrow(\sigma_1(c)\Rightarrow\forall x_1,\cdots,x_n:\omega_1)$，而 $(\sigma_1(c)\Rightarrow\forall x_1,\cdots,x_n:\omega_1)=\forall x_1,\cdots,x_n:\omega_2$，故 $\sigma_2(\alpha_2)\Leftrightarrow\forall x_1,\cdots,x_n:\omega_2$ 成立。

综上，定理证毕。 ■

定理 4.4 令 $\alpha=\alpha_1\|\alpha_2$，其中 α、α_1 和 α_2 都是一阶逻辑公式，令 $\langle\sigma,\omega\rangle=\langle\sigma_1,\omega_1\rangle\|\langle\sigma_2,\omega_2\rangle$。如果对 $i=1,2$ 都有公式 $\sigma_i(\alpha_i)\Leftrightarrow\forall x_1,\cdots,x_n:\omega_i$ 成立（其中 $x_1,\cdots,x_n$ 是 ω_i 中引入的所有新变量名），则必有 $\sigma(\alpha)\Leftrightarrow\forall x_1,\cdots,x_n:\omega$ 成立。

证明：并行组合两个部分最弱前置条件 $\langle\sigma_1,\omega_1\rangle$ 和 $\langle\sigma_2,\omega_2\rangle$ 时，我们通过插入一系列赋值语句来消除两个变量命名映射 σ_1 和 σ_2 的冲突，从而根据定理 4.3 我们知道 $\sigma(\alpha_1)\Leftrightarrow\forall x_1,\cdots,x_n:(Eqs(\sigma,\sigma_1)\Rightarrow\omega_1)$、$\sigma(\alpha_2)\Leftrightarrow\forall x_1,\cdots,x_n:(Eqs(\sigma,\sigma_2)\Rightarrow\omega_2)$。又由于 $\sigma(\alpha)=\sigma(\alpha_1\wedge\alpha_2)=\sigma(\alpha_1)\wedge\sigma(\alpha_2)$，因此我们可以推导出 $\sigma(\alpha)\Leftrightarrow\forall x_1,\cdots,x_n:\big((Eqs(\sigma,\sigma_1)\Rightarrow\omega_1)\wedge(Eqs(\sigma,\sigma_2)\Rightarrow\omega_2)\big)$，即 $\sigma(\alpha)\Leftrightarrow\forall x_1,\cdots,x_n:\omega$。 ■

为了把基于二元偶 $\langle\sigma,\omega\rangle$ 的部分最弱前置条件表示与计算方法与切片执行过程结合，我们定义一个如公式(4.6)所示的辅助函数 h，以将 $\langle\sigma,\omega\rangle$ 转换为一个一阶逻辑公式，其中 $x_1,\cdots,x_n$ 为 $\langle\sigma,\omega\rangle$ 中包含的所有新变量名：

$$h(\langle\sigma,\omega\rangle)\triangleq\forall x_1,\cdots,x_n:\left(\left(\bigwedge_{x\in dom(\sigma)}x=\sigma(x)\right)\Rightarrow\omega\right)\tag{4.6}$$

对图 4.2 所示的切片执行过程的第 6 行，我们需要将蕴含式 $\alpha'\Rightarrow\alpha_1\vee\cdots\vee\alpha_n$ 对应地替换为 $h(\langle\sigma',\omega'\rangle)\Rightarrow h(\langle\sigma_1,\omega_1\rangle)\vee\cdots\vee h(\langle\sigma_n,\omega_n\rangle)$。另外，在图 4.2 所示的切片执行过程第 7、9、14 行也要进行相应的替换。

4.4 处理指针与变量别名

为了支持指针与变量别名，我们对“变量位置”进行重命名，而不是对程序变量进行重命名。如第二章所述，变量位置是对计算机存储器地址的抽象，一个变量位置对应于一个存储器地址所存储的内容，它可以是一个变量、一个结构的成员或者对一个变量位置的引用。例如，变量 $*p$ 涉及到两个变量位置，即 p 和 $*p_1$，

其中 p_1 是变量位置 p 的值。

如果两个变量共享一个变量位置，我们就称它们互为别名。例如，如果 $p=\&x$，那么变量 $*p$ 和变量 x 共享同一个变量位置，从而它们互为别名。当更新表示部分最弱前置条件的二元偶 $\langle\sigma,\omega\rangle$ 时，我们保证变量命名映射 σ 将所有互为别名的变量都映射到同一个名字。换言之，σ 将具有相同变量位置的所有变量都映射到相同的名字。

当我们在切片执行过程中更新赋值语句的部分最弱前置条件时，为了满足上述约束，我们需要知道抽象准则 V 中的任意两个变量是否具有相同的程序位置。幸运的是，由于图 4.2 所示的切片执行过程在搜索状态 $\psi=\langle s,\alpha\rangle$ 时，我们知道 α 是某一条特定执行路径的部分最强后置条件，因此该执行路径的所有别名能够通过 α 确定。

在上一节中，基于二元偶 $\langle\sigma,\omega\rangle$ 的部分最弱前置条件的定义没有考虑指针和变量别名问题，如果存在指针和变量别名，我们需要重新定义赋值语句和 assume 语句的部分最弱前置条件。首先，我们定义 $\sigma\langle e\rangle$ 为将表达式 e 分解为变量位置、并为每个变量位置都引入一个新名字后的得到的变量命名映射。在这个过程中，如果发现具有相同变量位置的其它变量已经存在于 σ 中（即遇到了别名），则我们对所有别名使用相同的名字重命名；否则，我们引入一个新名字对该变量位置进行重命名。例如，设 $\sigma=\{x\to x'\}$，如果变量 $*p$ 和变量 x 互为别名（即 $p=\&x$），则 $\sigma\langle *p\rangle=\{x\to x', p\to p', *p'\to x'\}$，其中 p' 是对变量位置 p 引入的新名字，由于 x 和 $*p'$ 共享同一个变量位置，因此我们将它们映射到同一个名字 x'。同样，我们定义 $\sigma(x)$ 为变量 x 的当前名字，例如在上面的例子中，$\sigma\langle *p\rangle(*p)=x'$。

定义 4.7 支持指针和变量别名的部分最弱前置条件 $\widetilde{WP}_V(x:=e)(\langle\sigma,\omega\rangle)$ 定义为：

- $\langle\sigma^*,(\sigma'(x)=\sigma'(e))\Rightarrow\omega\rangle$，当且仅当 $x:=e$ 为抽象准则 V 下的完全相关赋值语句；
- $\langle\sigma^*,\exists\sigma'(x).\omega\rangle$，当且仅当 $x:=e$ 为抽象准则 V 下的部分相关赋值语句；
- $\langle\sigma,\omega\rangle$，当且仅当 $x:=e$ 为抽象准则 V 下的无关赋值语句。

其中 $\sigma'=\sigma\langle x\rangle\langle e\rangle$，即将变量 x 和表达式 e 先后分解为变量位置、并为每个变量位置引入合适的名字后的映射；σ^* 则在 σ' 的基础上进行如下操作得到：将每个命名 $y\to y'\in\sigma'$ 更新为 $y[\sigma'(e)/\sigma'(x)]\to y^*$，其中 $y[\sigma'(e)/\sigma'(x)]$ 为将变量位置 y 中每个变量 $\sigma'(x)$ 的自由出现都用 $\sigma'(e)$ 替换后得到的新变量位置，而 y^* 则定义为：

$$y^*=\begin{cases}x' & \text{if } y'=\sigma'(x)\\ y' & \text{if } y'\neq\sigma'(x)\end{cases}$$

其中 x' 是变量 x 对应的新鲜或已存在的名字，这取决于在变量命名映射 σ' 中

是否存在变量 x 的别名。

直观地来说，就是先将赋值语句 $x:=e$ 对应的所有变量位置先分解、再引入合适的名字，从而得到 σ'。接下来，对每个映射 $(y\to y')\in\sigma'$，如果变量 y 中包含 $\sigma'(x)$，则需要将其更新为 $\sigma'(e)$。

例如，令变量命名映射 $\sigma=\{x\to x', p\to p', {}^*p'\to x'\}$，我们需要计算部分最弱前置条件 $\langle\sigma^*,\omega^*\rangle=\widetilde{WP}_V(p:=q)(\langle\sigma,\omega\rangle)$，其中 $p,q\in V$。第一步，我们需要基于 σ 计算 σ'，通过将变量 p 与 q 分解为变量位置并为它们各引入一个名字，我们得到 $\sigma'=\sigma\langle p\rangle\langle q\rangle=\{x\to x', p\to p', {}^*p'\to x', q\to q'\}$。第二步，是基于 ω 计算 ω^*，根据部分最弱前置条件的定义，$\omega^*=((\sigma'(p)=\sigma'(q))\Rightarrow\omega)=(p'=q'\Rightarrow\omega)$。第三步，是基于 σ' 计算 σ^*，我们先确定在 σ^* 中变量 p 的名字（对应于定义 4.7 的 x'），如果赋值语句 $x:=e$ 之前 p 在 σ' 中有别名，则使用该别名对应的名字；否则，需要引入一个新的名字。设 p'' 为变量 p 对应的新名字，那么我们考虑 σ' 中的每个映射：由于 $p'=\sigma'(p)$，因此 $p\to p'$ 被更新为 $p\to p''$；同时，映射 ${}^*p'\to x'$ 被更新为 ${}^*q'\to x'$（即将 p' 替换为 q'）；其它两个映射则保持不变。因此，我们得到最终的变量命名映射 $\sigma^*=\{x\to x', p\to p'', {}^*q'\to x', q\to q'\}$。

定义 4.8 支持指针和变量别名的部分最弱前置条件 $\widetilde{WP}_V(assume(c))(\langle\sigma,\omega\rangle)$ 定义为：

- $\langle\sigma\langle c\rangle,\sigma\langle c\rangle(c)\Rightarrow w\rangle$，当且仅当 $assume(c)$ 为相关 assume 语句；
- $\langle\sigma,\omega\rangle$，当且仅当 $assume(c)$ 为无关 assume 语句。

由于对 assume 语句而言所有变量的值都没有改变，因此计算支持指针和变量别名的 assume 语句的部分最弱前置条件是比较直接的。

另外，由于部分最弱前置条件的并行组合不受指针和变量别名的影响，因此支持指针和变量别名的部分最弱前置条件的组合如定义 4.6 所示。

4.5 实验结果与分析

在切片执行工具的基础上，我们实现了支持指针和变量别名的部分最弱前置条件，同时进行了一些实现上的优化。例如，实际程序中的一些赋值语句的部分最弱前置条件并不会引起公式规模的增长，如 $\widetilde{WP}_V(x:=y)(x>0)$，令 $x,y\in V$，由于赋值语句 $x:=y$ 不会引起公式的重复，同时 $x>0$ 中没有重复的变量 x，因此直接替换 x 比为 x 引入新的名字得到的公式规模小。因此，计算赋值语句的部分最弱前置条件时，我们比较进行变量替换和引入新变量名这两种计算方法的公式规模，选择公式规模小的一种计算方法。但是，如果上述赋值语句出现在分支语句的某个分支中，虽然替换代价较小，但可能导致分支语句无法合并相同的公式。例如，

$\widetilde{WP}_V\left(if(c)\,x:=y\ else\ x:=z\right)(x>0)$，令 $x,y,z,c\in V$，直接进行变量替换求得的部分最弱前置条件为 $(c\Rightarrow y>0)\wedge(\neg c\Rightarrow z>0)$，它包含了由 $x>0$ 得来的两个子公式 $y>0$ 和 $z>0$，而基于变量重命名方法得到的部分最弱前置条件为 $\left((c\wedge x=y)\vee(\neg c\wedge x=z)\right)\Rightarrow x>0$，如果 $x>0$ 是一个规模较大的公式，那么显然后者公式规模小。因此，在计算部分最弱前置条件时，对有两个以上前驱的程序位置，我们记录下其部分最弱前置条件，如果某个分支语句的两个分支在该程序位置汇合，则我们重新计算其部分最弱前置条件，以避免公式重复。

实验仍是基于实现 SSL 协议的 *openssl-0.9.6c* 程序源代码进行的，我们比较了集成部分最弱前置条件的切片执行与不集成部分最弱前置条件的切片执行方法生成切片执行图的代价，以及所生成的切片执行图的规模。我们考查了 20 个时序安全性质，这些性质描述了 SSL 协议的初始握手协议必须满足的约束，它们已经事先被验证为满足。为了比较的方便性和公平性，我们给出了性质验证所必须的程序变量，并事先将它们加入到抽象准则集合中，这样两种切片执行过程所生成的切片执行图经过模型检验都能够满足给定的性质，同时由于两种方法基于同样的抽象准则生成切片执行图，因此切片执行图的生成代价以及产生的切片执行图的规模具有可比性。

表 4.1 集成部分最弱前置条件的切片执行的实验结果比较

Property	SE				SE + PWP			
	States	Trans	TPC	Time	States	Trans	TPC	Time
ssl_clnt_1	12541	14451	8523	12.2	1528	1661	651	1.3
ssl_clnt_2	12517	14427	8482	12.3	1412	1535	602	1.1
ssl_clnt_3	12517	14427	8482	12.2	1412	1535	602	1.1
ssl_clnt_4	12517	14427	8482	12.4	1412	1535	602	1.1
ssl_srvr_1	12836	14675	8473	12.6	1442	1566	540	1.0
ssl_srvr_2	12765	14604	8227	12.4	1432	1556	532	1.0
ssl_srvr_3	12803	14642	8310	12.6	1438	1562	531	1.0
ssl_srvr_4	12765	14604	8227	12.5	1432	1556	532	1.0
ssl_srvr_5	29452	33724	17902	41.7	2793	3012	1071	3.2
ssl_srvr_6	39506	44980	23823	67.7	2258	2450	875	2.6
ssl_srvr_7	29556	33828	17902	41.8	2892	3136	1062	3.3
ssl_srvr_8	12884	14723	8473	12.7	1446	1570	540	1.0
ssl_srvr_9	29564	33836	17902	41.7	2986	3236	1088	3.4
ssl_srvr_10	12860	14699	8473	12.7	1444	1568	540	1.0
ssl_srvr_11	29603	33875	18084	42.0	2804	3042	1027	3.2
ssl_srvr_12	38873	44347	23823	64.9	2222	2414	875	2.2
ssl_srvr_13	29584	33856	18051	41.9	2720	2954	1031	3.0
ssl_srvr_14	39084	44558	23823	65.6	2234	2426	875	2.3
ssl_srvr_15	12908	14747	8473	12.7	1448	1572	540	1.0
ssl_srvr_16	39295	44769	23823	66.5	2246	2438	875	2.5

实验结果如表 4.1 所示，所有的实验数据都产生于一台 CPU 为 1.6GHz AMD Athlon XP、内存为 224MB 的台式机，其运行的操作系统为 Windows 2000 与 CygWin 2.427。表 4.1 中，“Property”是待验证的时序安全性质的名字；“SE”指不集成部分最弱前置条件的切片执行过程，而“SE+PWP”则表示集成了部分最弱前置条件的切片执行过程；“States”与“Trans”分别描述了生成的切片执行图中状态和迁移的数量，这两个数据描述了切片执行图的规模；“TPC”是 Theorem Prover Call 的缩写，描述了定理证明工具的调用次数，“Time”则是以秒为单位的切片执行过程的运行时间，这两个数据描述了生成切片执行图的代价。

从表 4.1 中我们可以看出，集成了部分最弱前置条件的切片执行过程（即 SE+PWP）所产生的切片执行图的状态数和迁移数大约都是不集成部分最弱前置条件的切片执行过程（即 SE）的 1/10。相应地，SE+PWP 生成切片执行图过程中的定理证明工具的调用次数（TPC）和生成时间都减至 SE 的大约 1/10。尽管通过理论分析我们知道集成了部分最弱前置条件后，切片执行产生的切片执行图的规模以及切片执行过程的代价会减小，但减小的幅度如此之大是超出预想的。在内存空间消耗方面，SE+PWP 以及 SE 这两种方法都是很小的，其峰值内存占用大约是 40MB，平均内存占用大约为 20MB。

通过实验，我们发现 SE+PWP 方法能够大幅减小切片执行图的状态空间和切片执行代价是基于程序的两个事实。首先，某个程序位置 s 处的部分最弱前置条件描述了使得一组从 s 出发的执行路径可行的最弱条件，实验中我们发现它的确比 s 处的部分最强后置条件弱很多，从而导致了 s 处状态的大量合并。其次，通过分析 *openssl* 程序，我们发现程序中的大部分相关变量都是局部使用的，也就是说只在程序的某一小段中使用。对于程序中的某个程序位置 s，如果从程序初始位置 s_0 到达 s 的某条执行路径 p 中使用了某局部使用的相关变量 x，那么 x 的信息将被保存在 s 处 p 的部分最强后置条件 α_p 中并被传递下去，尽管从 s 出发的所有执行路径中并不再包含对 x 的引用。此后，如果另一条到达 s 的执行路径 p' 的部分最强后置条件 $\alpha_{p'}$ 与 α_p 的区别仅在于变量 x 的约束不同，那么基于部分最强后置条件我们必须基于 $\alpha_{p'}$ 重新搜索 s 及其后继程序位置。实际上我们知道，这种搜索是完全没有必要的，因为 x 在从 s 出发的所有执行路径中不再出现。幸运的是，部分最弱前置条件由于不包含变量 x，从而能够帮助切片执行避免这种冗余搜索。此外，如果某变量 x 只在某程序位置 s 之后局部使用，则 s 处的部分最弱前置条件也不会包含 x，这是由于变量 x 要么在计算部分最弱前置条件的过程中被替换为别的变量，要么由于 x 的约束为 *True* 而被消除。事实上，包括 *openssl* 在内的几乎全部实用程序都包含有大量局部使用的变量，因此部分最弱前置条件的应用能够普遍提高切片执行的验证效率。

4.6 相关工作

最弱前置条件被广泛地应用于调试（例如[86]）和验证（例如[87, 88]）领域，其中 ESC/Java[87, 88]是应用最弱前置条件进行程序验证的成功范例。ESC/Java 将 Java 程序的每个方法及其应该满足的性质转换为一个最弱前置条件公式，即所谓的验证条件（Verification Condition），再基于自动判定过程如定理证明工具 Simplify[72]等判定验证条件是否满足以验证性质是否满足。本章提出的集成部分最弱前置条件的切片执行则采用了完全不同的方法，它利用一种称为部分最弱前置条件的保守最弱前置条件来符号化地描述程序的状态，以产生程序的有限状态模型，即切片执行图，再利用模型检验工具验证其是否满足时序安全性质。

更深入地，ESC/Java 主要面向结构化的程序、以及对程序中断言（Assertion）是否满足的验证，因此它使用的最弱前置条件中包括了对 assert 语句的最弱前置条件的定义。此外，由于 Java 语言支持异常，因此 ESC/Java 也扩充了对与异常语句相关的最弱前置条件定义。ESC/Java 通过将循环展开一定的次数，或者对循环进行分割并引入循环不变式的方式来计算循环的最弱前置条件，从而可能造成验证不可靠或高昂的验证代价。此外，对于非结构化的程序，对循环的识别也需要一定的代价。虽然文献[89]支持对非结构化的程序求最弱前置条件，但仍需要牺牲验证的可靠性或花费高昂的代价计算循环不变式。比较而言，切片执行过程中对部分最弱前置条件的计算是针对可行的执行路径的，从而适用于任意结构和包含任意循环的程序，尽管切片执行的终止性是不可判定的。

由于赋值语句和分支语句导致的公式重复，最弱前置条件公式可能随着程序规模的增长呈指数增长[83, 84]。当前的解决方法是在计算最弱前置条件之前将程序转换为被动的形式。本章提出的基于二元偶$\langle\sigma,\omega\rangle$的部分最弱前置条件表示和计算方法则不需要单独的程序被动化过程。事实上，我们可以认为该方法是在计算部分最弱前置条件的同时进行程序的被动化，即将赋值语句通过变量重命名转换为等价的 assume 语句。该方法的优点是支持任意结构的程序，并且按需被动化能够降低计算开销。

4.7 小结

本章中我们提出了一种保守的最弱前置条件，即部分最弱前置条件，用于描述变量抽象下程序的保守近似语义。我们定义了赋值语句、assume 语句以及并行组合的部分最弱前置条件的计算方法，还给出了集成了部分最弱前置条件的切片执行过程。为了避免部分最弱前置条件计算过程中出现的公式规模爆炸，我们提出了基于变量重命名的部分最弱前置条件表示和计算方法。在其基础上，我们还

提出了支持 C 程序中的指针和变量别名的部分最弱前置条件表示和计算方法。本章提出的基于部分最弱前置条件的切片执行过程被用于基于给定的变量集合生成 *openssl-0.9.6c* 程序的切片执行图，通过实验比较，我们发现集成部分最弱前置条件后的切片执行代价和生成的切片执行图都降为原来的大约 1/10。

第五章　有状态动态偏序缩减方法

状态空间爆炸问题一直是并发程序和并发模型验证面临的最大障碍，在对并发程序源代码的验证过程中，由于程序源代码包含了软件的所有细节，其状态空间爆炸问题尤为突出。对并发程序/模型而言，偏序缩减（Partial-Order Reduction）是一种缩减状态空间的简单但有效方法，偏序缩减的核心算法总的来说可以分为两种类型[90-93]：第一种是基于 Persistent/Stubborn 集合的方法，即根据“将来”的迁移计算出当前状态需要考虑的迁移集合；第二种是基于 Sleep 集合方法，即根据“过去”的迁移计算出当前状态不需要考虑的迁移的集合。本章提出的有状态动态偏序缩减方法属于第一类方法，但它能够与基于 Sleep 集合的方法很好地互补使用（参见实验一节）。

传统的基于 Persistent/Stubborn 集合的偏序缩减方法基于静态分析方法来得到用于计算 Persistent/Stubborn 集合的迁移之间的偏序信息（参见[91-95]），但正如文献[90]所指出的，由于静态分析方法的不精确性，使得我们不得不保守地考虑许多本不需考虑的迁移，从而大大降低了偏序缩减的效率。针对该问题，文献[90]提出了无状态动态偏序缩减方法[2]，其基本思想是在每个状态只考虑一个进程的迁移，接下来在搜索过程中针对当前迁移序列动态地收集各进程之间的通信和交互信息，这些信息包括每个共享变量以什么样的顺序被哪个进程访问等等，再根据对这些信息的分析在当前的迁移序列上增加必要的回溯点，每个回溯点记录了在该状态处还有哪些进程的迁移需要被考虑（我们将这些进程称为回溯进程），最后通过回溯遍历每个状态需要考虑的回溯进程，从而遍历所有具有不同偏序关系的迁移序列。由于动态偏序缩减方法收集的偏序信息是完全精确的，因此其状态空间缩减效果得到了大幅提高。

对面向时序安全性质的并发程序或模型的验证而言，有状态的状态空间搜索策略由于能够阻止对同一状态的多次重复搜索，从而能够大大降低搜索代价。相比而言，无状态的状态空间搜索策略虽然节省了状态存储的空间开销，但往往需要对同一个状态进行甚至是指数量级的重复搜索，这是因为软件程序往往具有多个串行组合的分支结构，从而导致指数量级的到达同一状态的执行路径数量。文献[90]中提出的无状态动态偏序缩减方法是面向无状态模型检验的，因此它适用于 Verisoft[38]以及 Java PathExplorer[39, 40]等模型检验工具。但是，由于无状态模型检

[2] “无状态动态偏序缩减方法”在文献[90]中的原名叫做“动态偏序缩减方法”，但它面向的是对状态空间的无状态搜索，即在搜索过程中不保存任何状态。为了区别于本章提出的有状态动态偏序缩减方法，我们将其进行了重命名。

验的效率较低，当前的大多数模型检验工具都是有状态的，如通用的模型检验工具 Spin[26, 32]、SMV[27]，用于 C 程序模型检验的 BLAST[47]、SLAM[41, 42]、MAGIC[48, 49]、ComFoRT[50, 52]及 Zing[45, 46]，以及用于 Java 程序模型检验的 Java PathFinder[53, 54]等。

无状态动态偏序缩减和有状态的状态空间搜索是两种互补的状态空间缩减策略，但正如文献[90]指出的，这两种方法不能直接结合。由于无状态动态偏序缩减方法在每个状态只考虑一个进程的迁移，该状态必须考虑的其它进程的迁移则是在对从该状态出发的迁移序列的搜索过程中确定的。因此，如果一条迁移序列 π 到达一个访问过的状态 s，我们不得不重新搜索 s，而不能像有状态搜索那样直接回溯，因为我们需要遍历从 s 出发的所有迁移序列，以得到迁移序列 π 所有的回溯点和每个回溯点对应的回溯进程。

本章中，我们提出了有状态动态偏序缩减方法，有效地将有状态搜索和动态偏序缩减方法结合在了一起。该方法的基本思想是为所有状态 s 生成一个总结（Summary），该总结描述了从 s 出发的所有迁移序列的迁移之间的交迭信息（Interleaving Information）。此后，如果状态 s 在状态搜索过程中重遇，则我们根据 s 的总结就能确定回溯点和相应的回溯进程，而不需要重新遍历从 s 出发的所有迁移序列。我们基于所谓的“发生前迁移映射（Happens-Before Transition Mapping）”来描述一条从 s 出发的迁移序列的交迭信息，进而状态 s 的总结就定义为一组发生前迁移映射的集合。在基于深度优先的切片执行过程中，要生成每个状态的总结只需要在切片执行回溯时进行一点额外的工作，以将当前状态的总结传递给其前趋。因此，动态偏序缩减能够与有状态模型检验如切片执行等自然地集成。

实现上述思想时，我们需要考虑到从每个状态 s 出发的迁移序列的数量可能很多、甚至是无限的，而所有迁移序列的交迭信息都必须被总结于状态 s 处。同时，对每个状态的总结进行操作的时间和空间代价都必须尽量小。此外，遍历带圈的状态空间时将到达一个状态 s，使得从 s 出发的某些迁移序列还没有被遍历，这样状态 s 的总结就是不完备的，进而基于 s 的总结推断出的回溯点和相应的回溯进程也是不完备的。为了解决这个问题，我们提出了一个修改的深度优先状态空间搜索策略，支持对任意结构的状态空间的搜索。

我们还基于两个来自文献[90]的实用并发程序片段进行了实验，实验结果证明有状态搜索和动态偏序缩减能够很好地互补，从而大大提高状态空间缩减的效果。例如，相比无状态动态编序缩减方法，有状态动态偏序缩减方法搜索的状态空间的大小、以及状态空间的搜索开销，都最多降低了 66 倍。此外，实验证明，用于存储每个状态的交迭信息总结所占用的空间相对于存储状态所需的空间而言是比较低的，平均只有后者的 1/10 左右。

5.1 并发 C 程序模型和无状态动态偏序缩减方法

5.1.1 并发 C 程序模型

我们考查的并发 C 程序模型由一个有限集合的进程组成，每个进程拥有自己的局部变量，并以确定的顺序执行一系列语句。进程之间的通信是通过对通信对象（Communication Object）的原子操作进行的，通信对象包括共享变量、信号量、锁等。本章中，我们约定进程之间的通信是通过对一些全局变量的操作进行的，为了进程间的同步，我们还引入如下两个同步原语 P 与 V：

$$P(x) :: block \text{ if } x \le 0 \mid x-- \text{ otherwise};$$

$$V(x) :: x++;$$

其中，*block*（阻塞）的意思是对原语 $P(x)$的执行当前不能完成。我们约定除了 $P(x)$之外的所有其它语句都不会阻塞。

我们将并发程序模型表示为一个标记迁移系统 LTS，令一个并发程序包含 m 个并发进程 $CP_i = \langle S_i, s_{0i}, T_i, \Delta_i \rangle$，其中整数 $i \in \{1,\cdots,m\}$ 表示第 i 个进程，则该并发程序模型表示为 $CCP = \langle S, s_0, T, \Delta \rangle$，其中 S 表示模型的状态空间，定义如下：

$$S \subseteq S_1 \times \cdots \times S_m \times LS_1 \times \cdots \times LS_m \times SS$$

上式中，S_i 为第 i 个进程 $CP_i = \langle S_i, s_{0i}, T_i, \Delta_i \rangle$ 的程序位置集合，LS_i 是第 i 个进程的所有局部变量的不同取值构成的局部状态空间，SS 则是所有共享变量的不同取值构成的全局状态空间。对每个状态 $s = \langle s_1,\cdots,s_m,ls_1,\cdots,ls_m,ss \rangle$，$ls_i$ 和 ss 都既可以显式地以各变量的取值来表示，又可以隐式地以各变量满足的一阶逻辑公式来符号化地表示。与文献[90]一样，为了简洁起见，本章采用显式状态表示法。

对于并发程序模型 $CCP = \langle S, s_0, T, \Delta \rangle$ 的其它三个元素，$s_0 \in S$ 为初始模型状态，$T = T_1 \cup \cdots \cup T_m$ 是模型的迁移语句的集合，$\Delta \subseteq S \times T \times S$ 则是模型的状态迁移关系集合。给定两个状态 $s_1 = \langle s_{11},\cdots,s_{1m},ls_{11},\cdots,ls_{1m},ss_1 \rangle$ 和 $s_2 = \langle s_{21},\cdots,s_{2m},ls_{21},\cdots,ls_{2m},ss_2 \rangle$ 和一个迁移语句 $t \in T$，我们定义 $\langle s_1, t, s_2 \rangle \in \Delta$ 当且仅当存在整数 $i \in \{1,\cdots,m\}$ 使得：

- $\langle s_{1i}, t, s_{2i} \rangle \in \Delta_i$ 且 $\forall 1 \le j \le m \wedge j \ne i : s_{1j} = s_{2j}$；
- 并发进程 i 的所有局部变量的取值 ls_{1i} 执行迁移语句后变为 ls_{2i}，且 $\forall 1 \le j \le m \wedge j \ne i : ls_{1j} = ls_{2j}$；
- 所有共享变量的取值 ss_1 执行迁移语句后变为 ss_2。

如果迁移语句 $t \in T$ 访问了一个或多个共享变量，我们就称 t 为可见（Visible）迁移；如果 t 不涉及任何共享变量，我们就称 t 为不可见（Invisible）迁移。

并发程序模型 $CCP = \langle S, s_0, T, \Delta \rangle$ 的一条迁移序列 π 是一系列迁移 $t_1 \cdots t_n$ 组成的序列，其中 $t_1,\cdots,t_n \in T$，并且存在状态 $s_1,\cdots,s_{n+1} \in S$ 使得 s_1 是初始状态 s_0，且对每个 $i : 1 \le i \le n$ 都有 $\langle s_i, t_i, s_{i+1} \rangle \in \Delta$。对一条迁移序列 $\pi = t_1 t_2 \cdots t_n$，其中 s_i 是由迁移序列

$t_1t_2\cdots t_{i-1}$到达的唯一状态，则我们给出下列定义[90]：

- π_i表示迁移t_i；
- $proc(t)$表示迁移t所属进程的整数标识；
- $\pi.t$、$t.\pi$和$\pi.\pi'$分别表示在迁移序列π之后扩展迁移t、将迁移t插入到π之前，以及将两条迁移序列π和π'连接成的一条迁移序列$\pi.\pi'$；
- $pre(\pi,i)$表示迁移t_i之前的状态s_i；
- $last(\pi)$表示π到达的最后一个状态s_{n+1}。如果$\pi=\varnothing$，则$last(\pi)=s_0$。

最后，如果涉及到多条迁移路径，我们就用πi（如$\pi 1$、$\pi 2$等）标识第i条。我们用Π表示一系列迁移路径的集合，用$[\![\Pi]\!]$表示所有迁移路径的全集。

令$enabled(s)$表示在状态$s=\langle s_1,\cdots,s_m,ls_1,\cdots,ls_m,ss\rangle$处非阻塞的进程标识的集合，对一个进程$i$，$i\in enabled(s)$（我们称进程$i$为非阻塞进程）当且仅当存在至少一个非阻塞的迁移$\langle s,t,s'\rangle\in\Delta$使得$proc(t)=i$。对并发程序模型的最后一个状态$s$，由于没有从$s$出发的迁移，因此$p\notin enabled(s)$。除该状态外，由于只有同步原语$P(x)$在$x\le 0$的情况下可能阻塞，因此只有当某进程的下一个迁移是原语P时，该进程才可能被阻塞。

5.1.2 无状态动态偏序缩减方法

本节中，我们简要介绍文献[90]提出的无状态动态偏序缩减方法，为下文介绍有状态动态偏序缩减方法作准备。

为了表述的简洁和方便起见，文献[90]仅考虑了时序安全性质中最简单的一种，即可达性（Reachability）安全性质，例如检测死锁及断言可满足性等。在此假设下，两条迁移路径的偏序关系与待验证的可达性安全性质无关。事实上，本节的无状态动态偏序缩减方法及下一章将要介绍的集成有状态动态偏序缩减的切片执行方法都能够方便地扩充到对一般时序安全性质的验证，只需要将程序中与性质自动机迁移相关的程序语句作为变量抽象下的全局相关语句即可。

同样，为了表述的简洁性，文献[90]还引入了一些假设：

- 假设在每个状态处每个并发进程都只有一条可行的迁移，在状态s处并发进程p的唯一迁移记为$next(s,p)$。我们知道每个并发进程最多只有两个可行迁移，对应于并发进程的分支语句，该假设的意思是分支语句只有一个分支是可行的。切片执行往往保守地假设分支语句的两个分支都可行，这种情况可以通过简单地扩充动态偏序缩减算法来支持；
- 假设并发程序模型$CCP=\langle S,s_0,T,\Delta\rangle$的每个迁移$t\in T$只访问最多一个共享变量。我们将可见迁移$t$访问的共享变量记为$obj(t)\in[\![Object]\!]$，其中$[\![Object]\!]$表示所有共享变量的全集。扩展对访问多个共享变量的迁移的支

持也是比较简单和直接的，下文中我们将涉及；

- 假设两个可见迁移t_1与t_2是相关的，当且仅当它们访问了同一个共享变量，即$obj(t_1)=obj(t_2)$。如果t_1与t_2访问了多个共享变量，则它们相关当且仅当它们访问了至少一个相同的共享变量。

对一条迁移序列π，文献[90]首先定义了“发生前（Happens-Before）”序关系$\rightarrow_\pi$。$\rightarrow_\pi$定义为满足如下两个条件的集合$\{t_1,\cdots,t_n\}$上的最小关系：

1. 如果$i\le j$且t_i与t_j相关，则$t_i\rightarrow_\pi t_j$；
2. $\rightarrow_\pi$是满足上述条件 1 的传递闭包。

注意，文献[90]将发生前关系定义在整数集合$\{1,\cdots,n\}$，而我们为了下文的描述方便，将其等价地定义在相应的迁移集合$\{t_1,\cdots,t_n\}$上。

从发生前序关系的构造过程可以看出它是一个偏序关系，而迁移序列π则是该偏序关系的一种线性化（Linearization）。为了鉴别迁移序列的回溯点，文献[90]又提出了发生前序关系的一个变种，即$\pi_i\rightarrow_\pi p$。对$i\in\{1,\cdots,n\}$及一个进程标识p，如果$proc(\pi_i)=p$或者存在$k\in\{i+1,\cdots,n\}$使得$\pi_i\rightarrow_\pi\pi_k$且$proc(\pi_k)=p$，则有关系$\pi_i\rightarrow_\pi p$成立。直观地，如果关系$\pi_i\rightarrow_\pi p$成立，则在迁移序列$\pi$以及所有通过互换$\pi$的无关相邻迁移得到的具有相同偏序关系的等价迁移序列中，进程 p 从状态$last(\pi)$出发的迁移$next(last(\pi),p)$都决不可能是进程 p 在迁移π_i之前的状态$pre(\pi,i)$的迁移$next(pre(\pi,i),p)$，即$next(last(\pi),p)\neq next(pre(\pi,i),p)$。

```
Stack: A list of transition sequence π;
backtrack, done: S ↦ N;
Explore( )
{
1  for all s∈S let backtrack(s)=done(s)=∅;
2  Stack.push(∅);
3  if (∃p∈enabled(s0)) backtrack(s0):={p};
4  while ( ! Stack.empty( ) ){
5      let π=Stack.top( ) and let s=last(π);
6      Stack.pop( );
7      if (∃p∈backtrack(s)\done(s)) {
8          done(s):=done(s)∪{p};
9          let π'=π.next(s,p) and s'=last(π');
10         RefineBackTrackDpor(π');
11         if(∃p∈Enabled(s')) backtrack(s'):={p};
12         Stack.push(π');
13     }else if (∃π',t: π't=π) Stack.push(π');
14 }
}
```

```
RefineBackTrackDpor(π)
{
15  let s = last(π);
16  for all processes p {
17      if ∃i = max({i∈dom(π) | π_i is dependent and may be co-enabled with
            next(s, p) and i ↛_π p}) {
18              let E = {q∈enabled(pre(π,i)) | q = p or ∃j ∈ dom(π): j > i and
                    q = proc(π_j) and j →_π p};
19              if (E ≠ ∅) then add any q∈E to backtrack(pre(π,i));
20              else add all q∈enabled(pre(π,i)) to backtrack(pre(π,i));
        }
    }
}
```

图 5.1 基于动态偏序缩减的无圈状态空间遍历过程

文献[90]提出的无状态动态偏序缩减方法是通过对函数 Explore 的递归调用实现的，在本节中，为了与下文的有状态动态偏序缩减方法保持一致，我们将其改写为基于栈的等价算法，如图 5.1 所示。在图 5.1 中，*Stack* 是一个标准的先进后出栈，它存储的每个元素都是一条迁移序列，同时它拥有三个标准的栈操作，即 *push*、*pop* 和 *top*。全局变量 *backtrack* 和 *done* 分别记录了每个状态的待搜索和已经搜索完毕的进程标识，初始时它们都为空集。图 5.1 所示的动态偏序缩减过程对无圈的状态空间进行标准的深度优先搜索，函数 RefineBackTrackDpor 用于鉴别迁移序列 π' 的回溯点和该点处需要搜索的进程标识，其实现细节请参见文献[90]。

在实现时，我们可以使用时钟向量（Clock Vector）来表示每条迁移序列的发生前关系，基于时钟向量能够很容易判定函数 RefineBackTrackDpor 第 17 行的关系 $i \not\rightarrow_\pi p$ 和第 18 行的关系 $j \rightarrow_\pi p$ 是否成立。但是，出于表述的简洁和清晰起见，本章中不会涉及到时钟向量等实现细节，尽管时钟向量能够与有状态偏序缩减方法自然地集成。在下一章中，我们将详细讨论时钟向量及其在集成有状态偏序缩减方法的切片执行过程中的应用。

5.2 有状态动态偏序缩减方法

5.2.1 交迭信息总结

我们引入发生前迁移映射（Happens-Before Transition Mapping）的概念，用来表示一条迁移序列的交迭信息总结（Summary of Interleaving Information）。首先，我们将函数 *obj* 的定义域从迁移集合延拓到迁移序列集合，即 $obj(\pi) = \{obj(\pi_i) \mid i \in dom(\pi)\}$ 定义为迁移序列 π 所涉及的所有共享变量的集合。对

一条迁移序列π，我们定义函数$MinIndex(\pi,o)$返回π中访问共享变量$o \in obj(\pi)$的第一个迁移，即$MinIndex(\pi,o)=\min\{i|obj(\pi_i)=o\}$。我们定义另外一个函数$MinObjTrans$返回$\pi$中访问共享变量集合$obj(\pi)$的第一个迁移的集合，如下所示：

$$MinObjTrans(\pi) \triangleq \{\pi_i | \exists o \in obj(\pi): i = MinIndex(\pi,o)\}$$

迁移序列π的发生前迁移映射记为Υ_π，其定义如下：

$$\Upsilon_\pi : MinObjTrans(\pi) \mapsto 2^T$$

我们定义$dom(\Upsilon_\pi)$为映射Υ_π的定义域，即$dom(\Upsilon_\pi)=MinObjTrans(\pi)$。基于发生前关系$\rightarrow_\pi$，对每个迁移$t \in dom(\Upsilon_\pi)$，$\Upsilon_\pi(t)$定义如下：

$$\Upsilon_\pi(t) \triangleq \{\pi_i | \pi_i \rightarrow_\pi t\}$$

换言之，对访问每个共享变量$o \in obj(\pi)$的第一个迁移π_i，根据Υ_π可以得到在π或与其具有相同偏序关系的等价迁移序列中必须发生在π_i之前的迁移集合。

例 5.1 考虑下列迁移序列：

$$\pi \quad = \quad p_1{:}x{+}{+};\ \ p_2{:}x{+}{+};\ \ p_1{:}y{+}{+};\ \ p_2{:}y{+}{+};$$

我们有$obj(\pi)=\{x,y\}$、$MinIndex(\pi,x)=1$以及$MinIndex(\pi,y)=3$，其中π_1和π_3分别代表迁移$p_1{:}x{+}{+}$和$p_1{:}y{+}{+}$。因此$dom(\Upsilon_\pi)=MinObjTrans(\pi)=\{p_1{:}x{+}{+}, p_1{:}y{+}{+}\}$并且$\Upsilon_\pi(p_1{:}x{+}{+})=\varnothing$、$\Upsilon_\pi(p_1{:}y{+}{+})=\{p_1{:}x{+}{+}\}$（因为$p_1{:}x{+}{+} \rightarrow_\pi p_1{:}y{+}{+}$）。

新迁移序列$t.\pi$（即在π之前插入迁移t得到的迁移序列）的发生前迁移映射$\Upsilon_{t.\pi}$可以从迁移序列π的发生前迁移映射Υ_π得到。首先，$\Upsilon_{t.\pi}$的定义域变为$dom(\Upsilon_{t.\pi})=dom(\Upsilon_\pi) \cup \{t\} \setminus \{t^*\}$，其中$t^* \in dom(\Upsilon_\pi)$为满足条件$obj(t)=obj(t^*)$的迁移（根据$\Upsilon_\pi$的定义，$t^*$是唯一的）。也就是说，我们将$t$加入$dom(\Upsilon_\pi)$并将$t^*$移出$dom(\Upsilon_\pi)$后得到$dom(\Upsilon_{t.\pi})$，其中$t^*$是$dom(\Upsilon_\pi)$中访问与迁移$t$相同的共享变量的迁移。进而，我们定义：

$$\Upsilon_{t.\pi}(t') \triangleq \begin{cases} \varnothing & if\ \ t=t' \\ \Upsilon_\pi(t') \cup \{t\} \setminus \{t^*\} & if\ \ t \rightarrow_{t.\pi} t' \vee \exists t'' \in \Upsilon_\pi(t').\ t \rightarrow_{t.\pi} t'' \\ \Upsilon_\pi(t') \setminus \{t^*\} & otherwise \end{cases}$$

一个状态s的交迭信息总结 SII（Summary of Interleaving Information）定义为一系列发生前迁移映射的集合，每个发生前迁移映射对应一条从s出发的迁移序列。令$\Pi=\{\pi 1,\cdots,\pi n\}$为从某个状态$s$出发的迁移序列的集合，则状态$s$相对于迁移序列集合$\Pi$的交迭信息总结记为$SII_\Pi(s)$，它定义如下：

$$SII_\Pi(s) \triangleq \{\Upsilon_{\pi 1},\cdots,\Upsilon_{\pi n}\}$$

同时，我们使用符号$[\![SII]\!]$表示所有交迭信息总结的全集。

如果给定并发程序模型的状态空间是有限和无圈的，那么从任意状态出发的迁移序列的数量都是有限的。对状态s，令Π为从s出发的所有迁移序列的有限集

合，令 SII_{Π} 为 s 相对于 Π 的交迭信息总结，如果沿着某条迁移序列 π 到达状态 s，则 π 的所有回溯点和相应的回溯进程能够根据 SII_{Π} 推断出来，图 5.2 给出了推断过程 RefineBackTrackSII。

```
RefineBackTrackSII(π, SII_Π)
{
1  for all Υ_πk ∈ SII_Π do
2      for all t ∈ dom(Υ_πk) do
3          if the following two conditions hold:
           - ∃i = max({i∈dom(π) | π_i is dependent and may be
                       co-enabled with t and π_i ↛_π,t t})
           - ∀t' ∈ Υ_πk(t): π_i ↛_π,t t'
           {
4              let E = {q∈enabled(pre(π, i)) | q = proc(t) or
                       ∃t' ∈ Υ_πk(t): q = proc(t') or ∃j ∈ dom(π): j > i
                       and q = proc(π_j) and j →_π proc(t)};
5              if (E ≠ ∅) then add any q ∈ E to backtrack(pre(π, i));
6              else add all q∈enabled(pre(π, i)) to backtrack(pre(π, i));
           }
}
```

图 5.2 面向无圈状态空间的回溯点和回溯进程鉴别过程

考察图 5.2 第 3 行给出的两个条件，直观地看，第一个条件的核心是 $\pi_i \not\rightarrow_{\pi,t} t$，第二个条件则规定对所有满足关系 $t' \rightarrow_{\pi k} t$ 的迁移 t' 都有 $\pi_i \not\rightarrow_{\pi,t} t'$，这两个条件结合起来可以推出 $\pi_i \not\rightarrow_{\pi,\pi k} t$（见引理 5.1）。由于迁移 t 是迁移序列 πk 的第一个访问共享变量 $obj(\pi_i)$ 的迁移，因此状态 $pre(\pi,i)$ 就是一个必要的回溯点，在该回溯点处需要搜索的进程的集合是由第 4 – 6 行确定的。定理 5.1 保证了该过程的正确性，为了证明定理 5.1，我们先证明引理 5.1。

引理 5.1 给定两个迁移序列 π 和 π'，对任意两个迁移 π_i 与 $t \in dom(\Upsilon_{\pi'})$，如果 $\pi_i \not\rightarrow_{\pi,t} t$ 以及对任意 $t' \in \Upsilon_{\pi'}(t)$ 都有 $\pi_i \not\rightarrow_{\pi,t} t'$，那么 $\pi_i \not\rightarrow_{\pi,\pi'} t$ 成立。

证明：令 $t = \pi'_k$，我们来考查迁移 π'_j，其中 $j = \max\left\{j \middle| j<k \wedge \pi'_j \not\rightarrow_{\pi'} \pi'_k\right\}$，即 π'_j 是迁移 t 之前但并不必须发生在 t 之前的最后一个迁移。我们可以推出，对所有 $l: j<l<k$ 都有 $\pi'_j \not\rightarrow_{\pi'} \pi'_l$ 成立，否则 $\pi'_j \rightarrow_{\pi'} t$ 就会成立。也就是说，迁移 π'_j 同样不必须发生于 π'_j 与 t 之间的任意迁移之前。因此，我们将 π'_j 向后移动到迁移 t 之后得到的迁移序列是与 π' 是具有相同偏序的等价迁移序列。重复进行上述移动，我们将得到一条与 π' 具有相同偏序的等价迁移序列 $\overline{\pi'}$，在该迁移序列上迁移 t 前的

所有迁移都属于集合$\Upsilon_{\pi\cdot}(t)$。根据引理条件我们知道，在组合迁移序列$\pi\pi'$中，迁移序列π'中迁移t之前的所有迁移都可以发生在π_i之前，因此$\pi_i \not\to_{\pi.\pi'} t$成立，进而$\pi_i \not\to_{\pi.\pi'} t$成立，引理证毕。 ■

定理 5.1 如果给定的并发程序模型的状态空间是有穷和无圈的，那么对任意迁移序列π的所有状态$pre(\pi,i)$，基于图 5.3 所示过程 RefineBackTrackSII 得到的回溯进程集合$backtrack(pre(\pi,i))$与基于图 5.2 所示动态偏序缩减方法对相应迁移序列集合Π进行遍历得到的回溯进程集合是完全相等的。

证明：我们只需要证明对每条迁移序列$\pi' \in \Pi$，由过程 RefineBackTrackSII 基于$\Upsilon_{\pi'}$所鉴别的每个回溯进程集合$backtrack(pre(\pi,i))$完全等于基于动态偏序缩减方法对迁移序列$\pi.\pi'$遍历时得到的回溯进程集合即可。

根据图 5.3，如果过程 RefineBackTrackSII 鉴别出状态$pre(\pi,i)$处的回溯进程集合$backtrack(pre(\pi,i))$需要被更新，则我们知道一定存在一个迁移$t \in dom(\Upsilon_{\pi'})$使得$\pi_i \not\to_{\pi.t} t$以及对所有$t' \in \Upsilon_{\pi'}(t)$都有$\pi_i \not\to_{\pi.t} t'$成立。根据引理 1，我们知道$\pi_i \not\to_{\pi.\pi'} t$成立。令$t = \pi'_l$，则我们知道$t = next(pre(\pi',l), proc(t))$，因此在状态$pre(\pi',l)$处有$\pi_i \not\to_{\pi.\pi'|l} proc(t)$成立，其中$\pi.\pi'|l \triangleq \pi.(\pi'_1\pi'_2\cdots\pi'_{l-1})$是迁移序列$\pi.\pi'$在迁移$\pi'_l$之前的部分迁移序列。当图 5.2 所示的动态偏序缩减搜索过程对迁移序列$\pi.\pi'$的遍历到达状态$pre(\pi',l)$时，它也会鉴别出状态$pre(\pi,i)$处的回溯进程集合$backtrack(pre(\pi,i))$需要被更新，这是因为$\pi_i \not\to_{\pi.\pi'|l} proc(t)$并且$t$是迁移序列$\pi'$中第一个访问共享变量$obj(\pi_i)$的迁移。

接下来我们证明，分别被过程 RefineBackTrackSII 和动态偏序缩减过程加入到$backtrack(pre(\pi,i))$中的进程集合是完全相等的，我们只需要证明过程 RefineBackTrackSII 和动态偏序缩减过程中的集合E是相等的即可。事实上，根据发生前迁移映射Υ的定义，条件"$\exists t' \in \Upsilon(t): q = proc(t')$ or $\exists j \in dom(\pi): j > i$ and $q = proc(\pi_j)$ and $j \to_{\pi} proc(t)$"完全等价于条件"$\exists j \in dom(\pi.\pi'|l): j > i$ and $q = proc((\pi.\pi'|l)_j)$ and $j \to_{\pi.\pi'|l} proc(t)$"，因此两个集合$E$相等。

另一方面，如果图 5.2 所示的动态偏序缩减方法鉴别出迁移序列$\pi.\pi'$的状态$pre(\pi,i)$处的回溯进程集合$backtrack(pre(\pi,i))$需要被更新，那么我们知道$\pi_i \not\to_{\pi.\pi'} \pi'_l$成立，同时$\pi_i$是迁移序列$\pi.\pi'|l$中最后一个访问共享变量$obj(\pi'_l)$的迁移，因此$\pi'_l$是迁移序列$\pi'$中第一个访问共享变量$obj(\pi'_l)$的迁移，故$\pi'_l \in dom(\Upsilon_{\pi'})$。此外，$\pi_i \not\to_{\pi.\pi'} \pi'_l$成立证明$\pi_i \not\to_{\pi.\pi'_l} \pi'_l$成立以及对所有$t' \in \Upsilon_{\pi'}(\pi'_l)$都有$\pi_i \not\to_{\pi.t'} t'$成立，否则由于$t' \to_{\pi'} \pi'_l$导致$\pi_i \to_{\pi.\pi'} \pi'_l$成立。上面已证过程 RefineBackTrackSII 与动态偏序缩减过程中的集合E是相等的，因此得到的回溯进程集合$backtrack(pre(\pi,i))$也相等，从而定理证毕。 ■

5.2.2 有状态动态偏序缩减方法

对可能含圈的并发程序模型状态空间的有状态动态偏序缩减遍历过程如图 5.3 所示。

```
backtrack, done: S ↦ 2^N;
Stack: A list of transition sequence π;
SII: S ↦ [SII];
⟶: ⟶ ⊆ [Π] × S;
Explore( )
{
1   for all s ∈ S let backtrack(s) = done(s) = SII(s) = ∅;
2   Stack.push(∅);
3   if (∃p ∈ enabled(s_0)) backtrack(s_0) := {p};
4   while ( ! Stack.empty( ) ){
5       let π = Stack.top( ) and let s = last(π);
6       Stack.pop( );
7       if (∃p ∈ backtrack(s) \ done(s)) {
8           done(s) := done(s) ∪ {p};
9           let π' = π.next(s, p) and s' = last(π');
10          if (s' has not been visited before) {
11              RefineBackTrackDpor(π');
12              if(∃p ∈ enabled(s')) backtrack(s') := {p};
13          }else {
14              RefineBackTrackSII(π', SII(s'));
15              set π' ⟶ s';
16          }
17          Stack.push(π');
18      }else if (∃π', t: π't = π) {
19          Stack.push(π');
20          let s' = last(π') and let oldSII = SII(s');
21          for all Υ_πi ∈ SII(s) add Υ_t.πi to SII(s');
22          if (oldSII ≠ SII(s'))
23              for all π'' such that π'' ⟶ s' do {
24                  RefineBackTrackSII(π'', SII(s'));
25                  Stack.push(π'');
26              }
27      }
28  }
}
```

图 5.3 基于有状态动态偏序缩减的含圈状态空间遍历过程

图 5.3 所示过程的全局变量 *backtracking*、*done* 和 *Stack* 已经在图 5.1 中定义过了，在此不再重复。全局映射 *SII* 记录了每个状态的交迭信息总结，以避免对同一状态的重复搜索。如果并发程序模型的状态空间含圈，则对某些状态 s 而言，由于某些从 s 出发的迁移序列未被遍历，从而导致 s 的交迭信息总结可能并不完整。因此，我们定义迁移序列和状态之间的依赖关系 $\longrightarrow \subseteq [\![\Pi]\!] \times S$。如果一条迁移序列 $\pi = t_1 \cdots t_n$ 到达了以前访问过的状态 s_{n+1}，则我们设置 $\pi \longrightarrow s_{n+1}$。如果状态 s_{n+1} 的交迭信息总结 $SII(s_{n+1})$ 不完全，则当 $SII(s_{n+1})$ 被更新时，所有依赖于 s_{n+1} 的迁移序列（包括 π）的回溯点和回溯进程都需要重新计算。细心的读者可能会注意到符号 $\longrightarrow$ 已经用于切片执行图 $SEG = (\Psi, \longrightarrow)$ 的定义中，但这并不要紧，因为下一章中我们将会看到该依赖关系可以直接从切片执行图中得出。

我们可以看出，图 5.3 所示的基于有状态动态偏序缩减的含圈状态空间遍历过程与图 5.1 所示的基于动态偏序缩减的无圈状态空间遍历过程的基本框架是相同的，都是基于栈的深度优先搜索过程。我们考查图 5.3 第 10 行的分支语句，如果跳转到第 13 行，则表示到达了一个已经被访问过的状态 s'。从 s' 出发的所有迁移序列不需要重复遍历，我们调用过程 RefineBackTrackSII 基于 s' 的交迭信息总结 $SII(s')$ 鉴别出迁移序列 π' 的所有回溯点和回溯进程（第 14 行）。同时，我们还需要设置依赖关系 $\pi' \longrightarrow s'$（第 15 行），以应付 $SII(s')$ 不完全以及此后可能出现的被更新的情况。当搜索从 s 回溯到 s' 时（第 18 行），每个发生前迁移映射 $\Upsilon_{\pi i} \in SII(s)$ 都被更新为 $\Upsilon_{t\pi i}$ 并加入到 $SII(s')$ 中（第 21 行）。在第 22 行中，如果合并后的 $SII(s')$ 不等于合并前的 $SII(s')$，那么依赖于 s' 的所有迁移序列 π'' 的回溯点和回溯进程都需要重新计算（第 24 行），因为 $SII(s')$ 被更新了，可能鉴别出新的回溯点和回溯进程。随后我们将迁移序列 π'' 加入到栈中以对其重新遍历（第 25 行），如果 π'' 的某个回溯点增加了一些新的回溯进程，则我们就对它们进行遍历。值得注意的是，如果基于图 5.3 所示的过程遍历不含圈的状态空间，则由于每个状态被遍历完成之前不会被重新访问（否则形成了圈），因此过程第 18 行从 s 回溯到 s' 时，不会有任何迁移序列依赖于 s'，也就是说，第 23 行中不会存在任何迁移序列 π'' 使得 $\pi'' \longrightarrow s'$ 成立，因此不会引入额外的遍历代价。

定理 5.2 当图 5.3 所示的基于有状态动态偏序缩减的含圈状态空间遍历过程终止时，每个状态被搜索的回溯进程对应的迁移集合是该状态的一个持久集合（Persistent Set）。

证明： 文献[90]中的定理 1 指出，对于无圈的状态空间，基于标准动态偏序缩减方法的状态空间遍历过程（如图 5.1 所示）在每个状态搜索的迁移集合是该状态的一个持久集合。同时，本章定理 5.1 指出，当遇到被访问过的状态 s 时，尽管我

们并不遍历从 s 出发的迁移序列，但过程 RefineBackTrackSII 能够鉴别出与标准动态偏序缩减方法完全相同的回溯点和回溯进程。因此，对于无圈的状态空间，定理 5.2 成立。

对于含圈的状态空间，如果我们能够证明每个状态 s 的交迭信息总结 $SII(s)$ 在遍历过程终止时都是完全的，即 $SII(s)$ 包含了所有从 s 出发的迁移序列的发生前迁移映射，那么过程 RefineBackTrackSII 必将鉴别出所有的回溯点和回溯进程，从而从每个状态搜索的迁移集合必为该位置的一个持久集合。事实上，对到达任意状态 s 的迁移序列集合 Π，只要 s 的交迭信息总结 $SII(s)$ 被更新，我们就更新 Π 中每条迁移序列的回溯点和回溯进程，同时将其加入栈中重新搜索。因此，我们能够保证每个状态的交迭信息总结的完全性，从而定理得证。 ■

最后需要指出的是图 5.3 所示的基于有状态动态偏序缩减的含圈状态空间遍历过程的可终止性问题。我们看到，该过程会不断地遍历状态空间中的圈，只到圈上的每个状态的交迭信息总结 SII 不再发生变化为止。在下一节中我们将指出，任意状态的交迭信息总结 SII 都会收敛，因此只要状态空间是有限的，图 5.3 所示的过程就一定终止。

5.3 有状态动态偏序缩减的实现

本节中我们讨论如何高效实现发生前迁移映射、交迭信息总结及图 5.3 中对二者的操作。为了讨论实现算法的复杂性，我们假设并发程序拥有 m 个进程和总共 n 个共享变量。

5.3.1 交迭信息总结的实现

交迭信息总结由发生前迁移映射的集合组成，因此我们先讨论发生前迁移映射的实现。发生前迁移映射 Υ_π 实际上只记录了迁移序列 π 的访问每个共享变量的第一个迁移 t 以及必须发生在 t 之前的迁移的集合。同时，对 Υ_π 中的每个迁移 t，我们只关心它的两个参数 $proc(t)$ 和 $obj(t)$。因此，我们可以使用二元偶 $\langle proc(t), obj(t)\rangle$ 表示迁移 t，并也称 $\langle proc(t), obj(t)\rangle$ 为迁移。基于该表示法，对一个发生前迁移映射 Υ_π，其简化的发生前迁移映射记为 $\hat{\Upsilon}_\pi$，定义如下：

$$\hat{\Upsilon}_\pi : \mathbb{N}\times[\![Object]\!] \mapsto 2^{\mathbb{N}\times[\![Object]\!]}$$

将发生前迁移映射 Υ_π 中的每个迁移 t 用二元偶 $\langle proc(t), obj(t)\rangle$ 替换即得到其简化的发生前迁移映射 $\hat{\Upsilon}_\pi$。如果两条迁移序列 $\pi 1$ 和 $\pi 2$ 对应的简化发生前迁移映射 $\hat{\Upsilon}_{\pi 1}$ 和 $\hat{\Upsilon}_{\pi 2}$ 完全相等，我们就称 $\pi 1$ 和 $\pi 2$ 为 $\hat{\Upsilon}$-等价。由于 $\hat{\Upsilon}_\pi$ 只涉及迁移序列 π 中访问每个共享变量的第一个迁移及其相关的发生前迁移，而且每个迁移只关心其

进程和访问的共享变量，因此导致不同 $\hat{\Upsilon}_\pi$ 的迁移序列最多只有 $(m\times n)!$ 条，即 $m\times n$ 个二元偶 $\langle proc(t), obj(t)\rangle$ 的全排列。换言之，总共只有 $(m\times n)!$ 个不同的发生前迁移映射 $\hat{\Upsilon}_\pi$。

令 $\Pi=\{\pi 1,\cdots,\pi n\}$ 为从同一个状态 s 出发的迁移序列的集合，则状态 s 相对迁移序列集合 Π 的交迭信息总结 $SII_\Pi(s)$ 可以被等价地定义为 $SII_\Pi(s)\triangleq\{\hat{\Upsilon}_{\pi 1},\cdots,\hat{\Upsilon}_{\pi n}\}$，并且由下列两个映射实现：

$$OI:\ \mathbb{N}\times[\![Object]\!]\mapsto 2^{\mathbb{N}}$$

$$DI:\ \mathbb{N}\times[\![Object]\!]\mapsto 2^{\mathbb{N}}$$

其中 OI 表示共享变量索引（Object Index）映射，DI 表示依赖迁移索引（Depend Index）映射。基于 $SII_\Pi(s)$ 构造 $\langle OI, DI\rangle$ 的方法是：对每个 $\hat{\Upsilon}_{\pi i}\in SII_\Pi(s)$，每个迁移 $\langle p,o\rangle\in dom(\hat{\Upsilon}_{\pi i})$ 都在映射 OI 中被赋予一个整数索引 k，并且对每个迁移 $\langle p',o'\rangle\in\hat{\Upsilon}_{\pi i}(\langle p,o\rangle)$ 都将整数索引 k 加入到集合 $DI(\langle p',o'\rangle)$ 中。直观上，对每条迁移序列 $\pi i\in\Pi$，OI 将 πi 中访问每个共享变量的第一个迁移 t 映射到一个整数索引 k，并且对所有满足 $t'\rightarrow_{\pi i} t$ 的迁移 t' 都有 $k\in DI(\langle proc(t'),obj(t')\rangle)$。换言之，如果存在一个整数 k 使得 $k\in OI(\langle proc(t),obj(t)\rangle)$ 以及 $k\in DI(\langle proc(t'),obj(t')\rangle)$ 成立，则一定存在一条迁移序列 πi 使得 $t'\rightarrow_{\pi i} t$。

由于上述整数 k 可以任意选取，两个形式上不同的交迭信息总结可能代表同样的意义，因此我们定义两个交迭信息总结 $SII_\Pi(s)=\langle OI, DI\rangle$ 与 $SII_{\Pi'}(s)=\langle OI', DI'\rangle$ 等价，记为 $SII_\Pi(s)\equiv SII_{\Pi'}(s)$，当且仅当存在一个一一映射 $KT:\mathbb{N}\mapsto\mathbb{N}$ 使得下列四个条件同时成立：

- $\forall\langle p,o\rangle\in dom(OI)$ 及 $\forall k\in OI(\langle p,o\rangle)$，都有 $\exists k'\in OI'(\langle p,o\rangle):k'=KT(k)$；
- $\forall\langle p,o\rangle\in dom(OI')$ 及 $\forall k'\in OI'(\langle p,o\rangle)$，都有 $\exists k\in OI(\langle p,o\rangle):k'=KT(k)$；
- $\forall\langle p,o\rangle\in dom(DI)$ 及 $\forall k\in DI(\langle p,o\rangle)$，都有 $\exists k'\in DI'(\langle p,o\rangle):k'=KT(k)$；
- $\forall\langle p,o\rangle\in dom(DI')$ 及 $\forall k'\in DI'(\langle p,o\rangle)$，都有 $\exists k\in DI(\langle p,o\rangle):k'=KT(k)$。

例 5.2 考察下列从同一状态 s 出发的两条迁移序列：

$$\pi 1=\langle 1,x\rangle\langle 2,x\rangle\langle 1,y\rangle\langle 2,y\rangle$$

$$\pi 2=\langle 1,y\rangle\langle 2,y\rangle\langle 2,x\rangle\langle 1,x\rangle$$

我们有 $dom(\hat{\Upsilon}_{\pi 1})=\{\langle 1,x\rangle,\langle 1,y\rangle\}$、$dom(\hat{\Upsilon}_{\pi 2})=\{\langle 1,y\rangle,\langle 2,x\rangle\}$，进而我们得到 $\hat{\Upsilon}_{\pi 1}(\langle 1,x\rangle)=\varnothing$、$\hat{\Upsilon}_{\pi 1}(\langle 1,y\rangle)=\{\langle 1,x\rangle\}$、$\hat{\Upsilon}_{\pi 2}(\langle 1,y\rangle)=\varnothing$ 及 $\hat{\Upsilon}_{\pi 2}(\langle 2,x\rangle)=\{\langle 1,y\rangle,\langle 2,y\rangle\}$。接下来我们构造映射 OI：对 $dom(\hat{\Upsilon}_{\pi 1})$，我们用整数 1 和 2 分别表示迁移 $\langle 1,x\rangle$ 和 $\langle 1,y\rangle$；对 $dom(\hat{\Upsilon}_{\pi 2})$，我们用整数 3 和 4 分别表示迁移 $\langle 1,y\rangle$ 和 $\langle 2,x\rangle$。构造出的映射 OI 如表 5.1 所示，例如 $OI(\langle 1,y\rangle)=\{2,3\}$。我们再构造映射 DI：由于 $\hat{\Upsilon}_{\pi 1}(\langle 1,y\rangle)=\{\langle 1,x\rangle\}$，因此代表迁移序列 $\pi 1$ 中的迁移 $\langle 1,y\rangle$ 的整数（即整数 2）应该

被加入到集合 $DI(\langle 1,x\rangle)$，注意我们并不再整数 3 加入到集合 $DI(\langle 1,x\rangle)$ 中，这是因为整数 3 代表的是迁移序列 $\pi 2$ 中的迁移 $\langle 1,y\rangle$。按此方法处理完其它三个发生前迁移映射后得到的映射 DI 也在表 5.1 中给出。

表 5.1 两条示例迁移序列的交迭信息总结

Transition t	$OI(t)$	Transition t	$DI(t)$
<1, x>	{1}	<1, x>	{2}
<1, y>	{2, 3}	<1, y>	{4}
<2, x>	{4}	<2, y>	{4}

需要指出的是，在表 5.1 中，如果给每个迁移（例如 $\langle 1,x\rangle$）赋不同的整数索引（例如 5），则我们将得到等价的交迭信息总结，因此表 5.1 不是唯一确定的。

除了假设并发程序模型拥有 m 个进程和总共 n 个共享变量外，我们假设每个状态最多有 l 条从该位置出发、且相互并不 $\hat{\Upsilon}$-等价的迁移序列（由于最多只有 $(m\times n)!$ 条互不 $\hat{\Upsilon}$-等价的迁移序列，因此 $l\le(m\times n)!$），则我们可以对映射 OI 和 DI 的空间代价进行一个很粗略、很保守的估计：映射 OI 的定义域中最多只有 $m\times n$ 个迁移，且每个迁移最多可拥有 l 个整数索引，因此 OI 最多包含 $m\times n\times l$ 个整数索引并最多占用 $O(m\times n\times l)$ 的空间；映射 DI 的定义域最多也只有 $m\times n$ 个迁移，且每个迁移最多拥有 $m\times n\times l$ 个整数索引（即映射 OI 的所有整数索引），因此映射 DI 最多占用 $O(m^2\times n^2\times l)$ 的空间，这也是 $SII_{\Pi}(s)$ 最高的空间代价。需要指出的是，由于上述估计非常粗略和保守，因此实际应用时的空间代价要远小于上述估计值，请参见实验一节。

由于映射 OI 和 DI 中不显含迁移序列，因此存储两个映射所需的空间代价得到大大的降低。而且，由于最多只可能有 $(m\times n)!$ 种不同的发生前迁移映射，因此不管迁移序列集合 Π 中有多少条（甚至无限条）迁移序列，状态 s 处不同的交迭信息总结 $SII_{\Pi}(s)\triangleq\langle OI, DI\rangle$ 的数量最多只可能有 $2^{(m\times n)!}$ 种，因此只要给定的状态空间是有限的，图 5.4 所示的基于有状态动态偏序缩减的含圈状态空间遍历过程就一定会终止。

另外，由于交迭信息总结 $SII_{\Pi}(s)\triangleq\langle OI, DI\rangle$ 中不显含迁移序列，因此下文中我们将其简记为 $SII(s)$。

5.3.2 基于有状态动态偏序缩减的含圈状态空间遍历过程的实现

为了实现图 5.3 所示的基于有状态动态偏序缩减的含圈状态空间遍历过程，我们用二元偶 $\langle OI, DI\rangle$ 表示每个交迭信息总结 $SII(s)$，并将其第 22 行判定两个交迭信息总结是否相等的条件 $oldSII\neq SII(s')$ 替换为 $oldSII\neq SII(s')$。除此之外，我们

还需要实现第 14 和 24 行用到的函数 RefineBackTrackSII，以及在第 21 行用到的另外两个函数：UpdateSII 和 MergeSII。函数 UpdateSII 用于计算在每条迁移序列头部插入一个迁移 t 后得到的新交迭信息总结 $SII(s)$，函数 MergeSII 用于将当前的交迭信息总结合并到原有的交迭信息结中。也就是说，图 5.3 所示的遍历过程的第 21 行应该由下列三个语句实现：

$SII' = SII(s)$;
UpdateSII(SII', t);
MergeSII($SII(s')$, SII');

基于新表示方法 $SII(s)=\langle OI,\ DI\rangle$ 鉴别一条迁移序列 π 的回溯点和回溯进程的过程 RefineBackTrackSII 定义如图 5.4 所示，其中 $s=last(\pi)$，即迁移序列 π 的最后一个状态为 s。

$BT:\mathbb{N}\times[\![Object]\!]\mapsto 2^{\mathbb{N}}$;
$BL:\mathbb{N}\times[\![Object]\!]\mapsto \mathbb{N}$;
RefineBackTrackSII(π, $SII=\langle OI,DI\rangle$)
{
1 for all $\langle p,o\rangle\in dom(OI)$ do
2 if $\exists i=max(\{i\in dom(\pi)\mid \pi_i$ is dependent and may be co-enabled with $\langle p,o\rangle$ and $\pi_i \not\rightarrow_\pi \langle p,o\rangle\})$ then set $BT(\langle p,o\rangle):=OI(\langle p,o\rangle)$ and $BL(\langle p,o\rangle):=i$;
3 for all $\langle p',o'\rangle\in dom(DI)$ do
4 for all k such that $k\in BT(\langle p,o\rangle)$ and $k\in DI(\langle p',o'\rangle)$ do
5 if $\pi_i\rightarrow_\pi\langle p',o'\rangle$ where $i=BL(\langle p,o\rangle)$ then remove k from $BT(\langle p,o\rangle)$;
6 for all $\langle p,o\rangle\in dom(BT)$ do
7 if $(BT(\langle p,o\rangle)\neq\varnothing)$ {
8 let $i=BL(\langle p,o\rangle)$;
9 if $p\in enabled(pre(\pi,i))$ then add p to $backtrack(pre(\pi,i))$;
10 else add $enabled(pre(\pi,i))$ to $backtrack(pre(\pi,i))$;
11 }
}

图 5.4 RefineBackTrackSII 过程的实现

当执行完图 5.4 的过程 RefineBackTrackSII 的第 1－2 行的 for 循环后，我们构建出了一个可能为回溯点的映射 BT，它的定义域的每个迁移 $\langle p,o\rangle\in dom(BT)$ 都能发生在迁移 π_i 之前，其中 π_i 是迁移序列 π 中与迁移 $\langle p,o\rangle$ 相关且能同时非阻塞的最后一个迁移。接下来在第 3 － 5 行中，我们检查是否存在一个整数 k 使得 $k\in BT(\langle p,o\rangle)$ 及 $k\in DI(\langle p',o'\rangle)$ 同时成立（参见第 4 行），如果它们同时成立，那

么我们知道必然存在一条迁移序列π'，使得对其发生前迁移映射$\hat{\Upsilon}_{\pi'}$来说，有$\langle p',o'\rangle \in \hat{\Upsilon}_{\pi'}(\langle p,o\rangle)$成立。如果条件$\pi_i \rightarrow_{\pi} \langle p',o'\rangle$满足（参见第 5 行），则由于$\langle p',o'\rangle \in \hat{\Upsilon}_{\pi'}(\langle p,o\rangle)$，因此$\pi_i \rightarrow_{\pi,\pi'} \langle p,o\rangle$成立，所以这种情况下应该将整数$k$从集合$BT(\langle p,o\rangle)$中移除，表示$\pi'$中的迁移$\langle p,o\rangle$不可能发生在迁移$\pi_i$之前。如果第 7 行$BT(\langle p,o\rangle) \neq \varnothing$成立，则令$k \in BT(\langle p,o\rangle)$，我们知道必定存在一条迁移序列，使得其迁移$\langle p,o\rangle$（整数$k$表示的迁移）能够发生在迁移$\pi_i$之前。因此，我们需要在第 9 – 10 行鉴别回溯点和回溯进程。理论上，过程 RefineBackTrackSII 的最坏时空代价为$O(m^2 \times n^2 \times l)$，但实际应用时的代价远远小于该值。另外，定理 5.3 说明了图 5.4 所示过程与图 5.2 所示过程的等价性。

定理 5.3 令二元偶$\langle OI, DI\rangle$表示某状态s相对于迁移序列集合$\Pi = \{\pi 1, \cdots, \pi n\}$的交迭信息总结$SII_{\Pi}(s) = \{\hat{\Upsilon}_{\pi 1}, \cdots, \hat{\Upsilon}_{\pi n}\}$，对任意到达状态$s$的迁移序列$\pi$的任意一个状态$pre(\pi,i)$，某进程$p$被图 5.4 所示的过程 RefineBackTrackSII 加入到集合$backtrack(pre(\pi,i))$中，当且仅当存在一个发生前迁移映射$\hat{\Upsilon}_{\pi i}$，使得图 5.2 所示的过程 RefineBackTrackSII 在考查$\hat{\Upsilon}_{\pi i}$时将p加入集合$backtrack(pre(\pi,i))$中。

证明：我们看到，图 5.4 所示过程的第 1 – 2 行的循环实际上找出了满足图 5.2 所示过程第 3 行第一个条件的所有发生前迁移映射$\hat{\Upsilon}_{\pi i}$及其满足条件$\pi_i \not\rightarrow_{\pi,t} t$的迁移$t \in dom(\hat{\Upsilon}_{\pi i})$，而图 5.4 所示过程的第 3 – 5 行则对图 5.2 所示过程的第 3 行的第二个条件$\forall t' \in \Upsilon_{\pi k}(t): \pi_i \not\rightarrow_{\pi,t'} t'$进行检查，只有满足这两个条件的发生前迁移映射$\hat{\Upsilon}_{\pi i}$及其相应的迁移$t \in dom(\hat{\Upsilon}_{\pi i})$才会被用来鉴别回溯点和回溯进程。另外，图 5.4 所示过程的第 9 – 10 行与图 5.2 的 4 – 6 行是等价的，因此能够鉴别出相同的回溯进程，从而定理证毕。 ■

例 5.3 我们基于表 5.1 所示的交迭信息总结$SII(s)$对下列迁移序列的回溯点和回溯进程进行鉴别：

$$\pi = p_1{:}x{+}{+};\ p_2{:}y{+}{+};\ p_1{:}y{+}{+};\ p_2{:}x{+}{+};$$

在例 5.2 中，我们有$dom(OI) = \{\langle 1,x\rangle, \langle 1,y\rangle, \langle 2,x\rangle\}$，对迁移$\langle 1,x\rangle$而言，迁移序列$\pi$中与其相关且可同时非阻塞的最后一个迁移为$\pi_4 = p_2{:}x{+}{+}$。由于$\pi_4 \rightarrow_{\pi} \langle 1,x\rangle$，则我们应该将迁移$\langle 1,x\rangle$加入到$dom(BT)$并设置$BT(\langle 1,x\rangle) = OI(\langle 1,x\rangle) = \{1\}$。由于整数 1 并没有出现在映射 DI 中，因此我们不会将整数 1 移出集合$BT(\langle 1,x\rangle)$。由于集合$BT(\langle 1,x\rangle)$非空，因此迁移$\langle 1,x\rangle$能够发生在π_4之前，从而我们设置相应的回溯点和回溯进程。

图 5.5 给出了更新交迭信息总结的过程 UpdateSII，用于将当前状态的交迭信息总结传递给其前趋状态。如果图 5.5 第 3 行的条件$o = o'$满足，则由于在任意迁

移序列的头部插入迁移$\langle p,o\rangle$后，迁移$\langle p',o'\rangle$不再是该更新的迁移序列中第一个访问共享变量o的迁移，因此迁移$\langle p',o'\rangle$应该从映射OI的定义域$dom(OI)$中去掉（第5行）。同时，迁移$\langle p',o'\rangle$在映射OI中对应的所有整数索引$OI(\langle p',o'\rangle)$也应该从映射DI中去掉（第12行）。第6行中，$p=p'$意味着迁移$\langle p,o\rangle$与迁移$\langle p',o'\rangle$相关，从而$\langle p,o\rangle$发生于$\langle p',o'\rangle$之前，因此我们在第7行进行相应的设置。接下来，我们为新增加的迁移$\langle p,o\rangle$赋一个整数索引（第10行）。在第11－15行中，我们去掉映射DI中无用的整数索引，同时构建必要的依赖关系。如果第13行的条件满足，我们知道迁移$\langle p,o\rangle$与迁移$\langle p',o'\rangle$相关，从而$\langle p,o\rangle$发生于$\langle p',o'\rangle$之前，因此所有发生于迁移$\langle p',o'\rangle$之前的迁移也必发生在迁移$\langle p,o\rangle$之前，对此我们在第14行进行了设置。同样，图5.5所示过程的最坏时空代价是$O(m^2\times n^2\times l)$。定理5.4说明了该过程的正确性。

```
RS : RS ⊂ ℕ;
UpdateSII( SII = ⟨OI, DI⟩ , t)
{
1  let  p = proc(t)  and  o = obj(t);
2  for all  ⟨p',o'⟩ ∈ dom(OI)  do {
3      if (o = o') {
4          add  OI(⟨p',o'⟩)  to  RS;
5          remove  ⟨p',o'⟩  from  dom(OI);
6      }else if (p = p') {
7          add  OI(⟨p',o'⟩)  to  DI(⟨p,o⟩);
8      }
9  }
10 add a new index  k ∈ ℕ  to  OI(⟨p,o⟩);
11 for all  ⟨p',o'⟩ ∈ dom(DI)  do {
12    DI(⟨p',o'⟩) = DI(⟨p',o'⟩) \ RS;
13     if (p = p'  or  o = o')
14         add  DI(⟨p',o'⟩)  to  DI(⟨p,o⟩);
15 }
}
```

图5.5　交迭信息总结的更新过程UpdateSII

定理5.4 令迁移序列πt到达状态s（即$s=last(\pi t)$），令SII'为将迁移t插入到交迭信息总结$SII(s)$对应的所有迁移序列头部后得到的更新的交迭信息总结，则对迁移序列π来说，过程RefineBackTrackSII(πt, $SII(s)$)与过程RefineBackTrackSII(π, SII')鉴别出的回溯点和回溯进程是完全相同的。

证明：从图5.5所示过程UpdateSII对更新交迭信息总结的构造过程可以看出，

SII' 的确表示状态 $last(\pi)$ 处的交迭信息总结，从而可以推断出正确的回溯点和回溯进程，故定理证毕。 ■

图 5.6 给出了合并两个交迭信息总结的过程 MergeSII，如果某个状态有两个或多个后继位置，就需要调用该过程合并后继状态的交迭信息总结。图 5.6 所示过程的第 1 – 8 行将映射 OI' 和 DI' 中的整数索引替换为新的整数索引，并将新的整数索引分别加入映射 OI 和 DI 中。第 9 – 14 行的代码则检查是否存在两个整数索引 $k,k' \in OI(\langle p,o \rangle)$ 使得发生在 k 表示的迁移之前的迁移集合（即 $IT(k)$）是发生在 k' 表示的迁移之前的迁移集合（即 $IT(k')$）的子集，如果 $IT(k) \subseteq IT(k')$ 成立，则整数索引 k' 是冗余的，从而将其从 OI 和 DI 中去掉。

```
CHG : N ↦ N;
IT : N ↦ 2^(N×[Obj]);
MergeSII(SII = ⟨OI, DI⟩, SII' = ⟨OI', DI'⟩)
{
1  for all ⟨p,o⟩ ∈ dom(OI') do
2      for all k ∈ OI'(⟨p,o⟩) do {
3          add a new index k' ∈ N to OI(⟨p,o⟩);
4          set CHG(k) = k';
5      }
6  for all ⟨p,o⟩ ∈ dom(DI') do
7      for all k ∈ DI'(⟨p,o⟩) do
8          add index k' = CHG(k) to DI(⟨p,o⟩);
9  for all ⟨p,o⟩ ∈ dom(DI) do
10     for all k ∈ DI(⟨p,o⟩) do
11         add ⟨p,o⟩ to IT(k);
12 for all ⟨p,o⟩ ∈ dom(OI) do
13     if there exist k,k' ∈ OI(⟨p,o⟩) such that IT(k) ⊆ IT(k')
14         remove k' from OI and DI;
}
```

图 5.6 交迭信息总结的合并过程 MergeSII

例 5.4 考察例 5.2 的表 5.1 对应的交迭信息总结 $\langle OI, DI \rangle$，迁移 $\langle 1, y \rangle$ 被映射到两个整数索引 2 和 3，发生在索引 2 和索引 3 表示的迁移之前的迁移集合分别为 $\{\langle 1, x \rangle\}$ 和 $\varnothing$。因此对某条迁移序列 π，要使迁移 $\langle 1, y \rangle$ 发生在某迁移 π_i 之前，则考虑索引 2 时，我们需要检查迁移 $\langle 1, x \rangle$ 是否能够发生在迁移 π_i 之前；但如果考虑索引 3，则我们不需要检查任何迁移是否能够发生在迁移 π_i 之前。也就是说，有没有索引 2 不影响对任意迁移序列的回溯点和回溯进程的鉴别，从而索引 2 是冗余的，它可以从两个映射中去掉。

图 5.6 所示过程的最坏时空代价同样是$O\left(m^2 \times n^2 \times l\right)$，其中$IT\left(k\right) \subseteq IT\left(k'\right)$被认为可以在构建映射 IT 时判断出。定理 5.5 说明了该方法的正确性，其证明是直接的，因此我们不累述。

定理 5.5 令 SII_1 和 SII_2 是同一个状态 s 的两个不同的交迭信息总结，对任意迁移序列 π，由先调用进程 MergeSII(SII_1, SII_2)再调用进程 RefineBackTrackSII(π, SII_1)得到的回溯点和回溯进程与先调用进程 RefineBackTrackSII(π, SII_1)再调用进程 RefineBackTrackSII(π, SII_2) 得到的回溯点和回溯进程是完全相同的。

5.4 实验结果

我们使用了与文献[90]完全相同的两个来自实用程序的例子，即 Indexer 和 File System，来例证有状态动态偏序缩减方法的有效性。

```
int table[128];
int mutex[128]; //For atomic change
void thread(int tid)
{
    int m = 0, w, h, cas;
    while(1){
        if(m < 4){
            m++;
            w = m*11+tid;
        }else return;
        h = (w * 7) % 128;
    AGAIN:
        P(mutex[h]);
        if(table[h] == 0){
            table[h] = w;
            cas = 1;
        }else cas = 0;
        V(mutex[h]);
        if(cas == 0){
            h = (h + 1) % 128;
            goto AGAIN;
        }
    }
}
```

图 5.7 Indexer 示例程序

Indexer 程序是由如图 5.7 所示多个并发进程 thread 组成的，用于操纵一个共享的哈希表。每当进程接收到消息 w，就将该消息插入到哈希表中索引为 $h = hash(w)$ 的表项中，如果该表项已被其它消息占用，就将其插入到下一个空闲表项中。另外，一个共享的互斥变量数组 *mutex* 被用于保护哈希表的每个表项，以防止其被多个进程同时读写。

```
int locki[32];
int inode[32];
int lockb[26];
int busy[26];
void thread(int tid)
{
    int i, b;
    i = tid;
    while(i >= 32) i = i - 32;
    P(locki[i]);
    if(inode[i] == 0){
        b = i + i;
        while(b >= 26) b = b - 26;
        while(1){
            P(lockb[b]);
            if(!busy[b]){
                busy[b] = 1;
                inode[i] = b + 1;
                V(lockb[b]);
                break;
            }
            V(lockb[b]);
            b = b + 1;
            while(b >= 26) b = b - 26;
        }   /* end while(1) */
    }   /* end if(inode[i] == 0) */
    V(locki[i]);
}   /* end of thread */
```

图 5.8 File System 示例程序

File System 的并发进程如图 5.8 所示，它拥有两个内核数据结构 *inode* 和 *busy*，以及两个分别保护这两个内核结构不被多个进程同时访问的共享互斥变量数组 *locki* 和 *lockb*。对每个进程每次选择的 inode 号 *i*，File System 程序判断是否已经分配了空闲块，如果没有则从 *busy* 数组中寻找一个空闲块分配给该 inode。

在本小节的实验中，我们比较了四种策略，即无状态动态偏序缩减、集成 Sleep 集合技术的无状态动态偏序缩减、有状态动态偏序缩减和集成 Sleep 集合的有状态动态偏序缩减。正如本章引言部分提到的，基于 Sleep 集合的偏序缩减方法收集已经遍历过的迁移序列中各迁移与当前状态非阻塞的迁移之间的依赖关系等信息，并根据这些信息计算出当前哪些非阻塞的迁移并不需要被考虑。正如文献[90]所指出的，Sleep 集合技术与动态偏序缩减方法能够很好地互补。

表 5.2 Indexer 程序的实验结果比较

Procs	DPOR		DPOR + sleep set		SDPOR			SDPOR + sleep set		
	All Trans.	Time (second)	All Trans.	Time (second)	All Trans.	Time (second)	Mem (KB)	All Trans.	Time (second)	Mem (KB)
11	604	0.2	604	0.2	604	0.4	4.3	604	0.4	4.3
12	14479	4.6	4546	1.4	2399	1.9	511	2355	1.7	499
13	169661	59.3	23529	7.7	5196	5.3	1403	4124	3.7	1064
14	3837429	1814.4	182841	70.2	20901	30.7	5436	14406	16.1	3659
15			1508101	695.9	94506	248.0	25573	47623	77.7	12995
16			12507473	7072.8	450340	1718.4	143208	188452	594.6	48084

表 5.3 File System 程序的实验结果比较

Procs	DPOR		DPOR + sleep set		SDPOR			SDPOR + sleep set		
	All Trans.	Time (second)	All Trans.	Time (second)	All Trans.	Time (second)	Mem (KB)	All Trans.	Time (second)	Mem (KB)
13	142	0.1	142	0.1	142	0.1	30	142	0.1	30
14	434	0.2	298	0.1	303	0.2	68	298	0.2	68
15	1102	0.4	505	0.2	691	0.4	162	505	0.3	120
16	3226	1.3	960	0.4	1776	1.2	432	960	0.6	228
17	9922	4.0	1943	0.8	4945	3.5	1276	1943	1.2	463
18	30946	13.2	4046	1.6	14173	10.6	3951	4046	2.5	981
19	96790	41.8	8517	3.5	40924	33.0	12407	8517	5.4	2134
20	302602	140.0	17980	7.7	118261	103.5	39014	17980	12.1	4701
21	944842	474.7	37939	16.7	341493	317.9	122682	37939	26.5	10403
22			79914	36.2				79914	59.0	23012
23			167969	79.4				167969	137.2	51086
24			352280	175.4				352280	321.4	113923
25			737295	394.8				737295	769.9	255187
26			1540102	895.3						

实验结果如表 5.2 和表 5.3 所示，所有实验数据都在一台 1.6GHz Athlon CPU 和 1GB 内存的机器上产生，其软件环境中 Windows 2000 + Cygwin 2.427。表中“DPOR”指无状态动态偏序缩减方法，“SDPOR”指有状态动态偏序缩减方法，“Procs”指并发程序中进程的数量，“All Trans”指遍历的所有迁移的总数，“Time”是以秒为单位的并发程序模型遍历时间，最后“Mem”是以 KB 为单位的用于存储每个状态的交迭信息总结所占用的内存空间，需要注意的是，该空间并没有将用于存在模型状态和时钟向量的空间计算在内。

在表 5.2 和表 5.3 中，如果并发进程的数量分别小于等于 11 和 13，则所有进程都不会访问相同的共享变量，因此两个并发程序的所有迁移序列都具有相同的偏序关系，从而可以被偏序缩减到只剩一条。当两个并发程序的并发进程数量分别超过 11 和 13 且持续增长时，我们看到，有状态动态偏序缩减能够进行更大程度上的状态空间缩减，集成 Sleep 集合的有状态动态偏序缩减也是如此。例如在表 5.2 中，当并发进程的数量为 16 时，基于集成 Sleep 集合的有状态动态偏序缩减比基于集成 Sleep 集合的无状态动态偏序缩减得到的状态空间小 66 倍。由于状态空间的大小直接决定了遍历时间，因此当 Indexer 程序的并发进程数量为 14 个时，无状态动态偏序缩减的状态空间遍历时间是 1814.4 秒，而有状态动态偏序所花费的时间则只有 30.7 秒。但是我们也注意到，在表 5.3 中，集成 Sleep 集合的有状态动态偏序缩减方法所花费的时间要大幅超过集成 Sleep 集合的无状态动态偏序缩减方法，尽管二者遍历的迁移数量是完全相同的。我们发现多出的时间是用来操纵状态的，比如状态存储、状态比较等，稍后我们还会对该现象进行讨论。

更为彻底地，我们从表 5.2 中看到，单纯的有状态动态偏序缩减方法的状态空间缩减效果甚至超过了集成 Sleep 集合的无状态动态偏序缩减方法，这是因为 Indexer 程序中存在大量到达同一状态时具有相同部分最强后置条件、但具有不同偏序的迁移序列，有状态动态偏序缩减方法能够避免对这些迁移序列的重复搜索。然而在表 5.3 中，集成 Sleep 集合的无状态动态偏序缩减方法的状态空间缩减效果比单纯的有状态动态偏序缩减方法要好。表 5.3 同时指出，应用了 Sleep 集合缩减技术后，基于有状态和无状态的动态偏序缩减方法得到的状态空间完全相同，这是由于 File System 程序的所有到达同一状态的迁移序列都是具有相同偏序的等价迁移序列，从而能够被集成了 Sleep 集合的无状态动态偏序缩减方法识别和消除。但是，正如大家所公认的，对实际程序进行有状态的模型检验的状态空间缩减效果要大大优于无状态的模型检验，因此有状态动态偏序缩减方法的状态空间缩减效果能够大大优于无状态动态偏序缩减方法。

有状态动态偏序缩减方法的缺点是需要额外的内存空间来存储每个状态的交迭信息总结，但是，正如表 5.2 和表 5.3 所示的，相对用于模型状态存储的内存空

间而言，用于存储交迭信息总结的额外空间是很小的。在实验中，Indexer 和 File System 两个并发程序分别有 256 和 116 个共享变量，同时它们都拥有数十个并发进程，但每个交迭信息总结所占用的内存空间大约只有 200 到 300 字节。对 File System 并发程序，当它包含 24 个并发进程时，其用于存储交迭信息总结的内存空间为 114MB，而用于存储状态和时钟向量等信息的空间则大约有 800MB。当前，我们在交迭信息总结中直接存储每个共享变量的名字，一个简单的空间优化方法是为每个共享变量指定一个索引，并用索引代替交迭信息总结中的变量名。另外，传统的状态压缩等方法也能够显著地降低内存消耗。

5.5 相关研究工作

与本章有状态动态偏序缩减方法最相关的工作是由 Flanagan 和 Godefroid 在文献[90]中提出的无状态动态偏序缩减方法，我们已经在文中对其进行了详细的介绍。相比该方法，在只引入少量时空开销的前提下，集成有状态动态偏序缩减的切片执行过程结合了两种互补的状态空间缩减方法，即有状态遍历和动态偏序缩减，从而能够显著改善状态空间缩减效果，尤其是对实用程序更是如此。信息总结的思想曾经被用于过程间分析，如文献[46, 96]，用于对过程进行总结。过程总结随后可以被该过程的每个调用点重用，例如验证工具 Zing[45, 46]等。基于同样的思想，本章提出的方法总结从某个状态出发的所有迁移序列的交迭信息，并将其用于动态偏序缩减。

传统的基于偏序缩减的状态空间缩减方法也与我们相关，它们主要基于两类技术实现，即 Persistent/Stubborn 集合和 Sleep 集合。Persistent/Stubborn 集合[91-93]（进一步有 Ample 集合[94, 95]）技术能够从当前的非阻塞迁移集合中选择一个子集，使得未被选中的其它迁移决不会导致不同偏序的迁移序列。相反，Sleep 集合技术（见文献[91]）则根据过去的搜索过程计算出当前非阻塞的迁移集合中无需考察的迁移。这两种技术在我们的方法中也具有很好的互补性，能够同时使用以进一步缩减状态空间（见实验）。偏序缩减方法的最新研究包括基于簇（Cluster）的偏序缩减方法[97, 98]等，它引入了簇层次的概念，用于将并发系统进行层次式分解，而在层次式结构的程序中能够更容易、更准确地获得进程之间的依赖关系。

5.6 小结

本章首先提出了基于 *P/V* 同步原语的共享内存并发 C 程序模型，并简要介绍了文献[90]提出的无状态动态偏序缩减。接下来我们在无状态动态偏序缩减的基础上提出了有状态动态偏序缩减，使得有状态模型检验和动态偏序缩减方法能够结

合，从而对状态空间进行更大程度的缩减。不仅如此，我们还考虑了对含圈状态空间的支持，提出的深度优先有状态模型检验方法能够支持对任意状态空间的搜索遍历。我们还提出了有状态动态偏序缩减的高效实现方法，能够保证对有穷状态空间搜索的可终止性。最后，我们针对两个来自文献[90]的程序进行了实验，实验结果显示了有状态动态偏序缩减方法及集成该方法的切片执行过程的高效性。

第六章　并发 C 程序的切片执行

软件验证一直被公认为软件科学和工程领域的一个难点问题[99]，对并发软件的验证更是如此。模型检验是一种对并发软件验证有效且实用的验证理论，但状态空间爆炸问题往往限制了模型检验能够验证的软件的规模。对于面向并发程序源代码的模型检验来说，面向待验证的性质抽象出状态空间尽可能小的并发程序模型成为了决定模型检验能否成功的关键因素。

本论文的第二、三、四章介绍了面向顺序 C 程序的切片执行理论，它能够基于待验证的时序安全性质从 C 程序中自动抽象出有限状态模型。本章中，我们将切片执行理论拓展到对并发 C 程序的验证。面向并发 C 程序的切片执行与面向顺序 C 程序的切片执行的总体框架是完全相同的，也是一个迭代过程，在每次迭代时先基于变量抽象生成切片执行图，再基于切片执行图进行模型检验，最后根据模型检验的伪反例路径推断出必要的程序变量以精化抽象准则。同时，我们将看到，面向顺序 C 程序的其它理论，包括搜索复用框架、部分最弱前置条件等，也完全适用于对并发 C 程序的切片执行，从而能够有效缩减从并发 C 程序中自动抽象出的有限状态模型的状态空间。

设并发 C 程序由 n 个并发进程组成，则每个“并发程序位置” s 实际上是所有并发进程的程序位置的组合，即 $s=\langle s_1,\cdots,s_n\rangle$，其中 s_i 为第 i 个进程的程序位置。也就是说，并发程序位置的集合是所有进程的程序位置集合的笛卡尔乘积。为了遍历并发程序模型的所有状态空间，切片执行基于深度优先搜索策略，在每个并发程序位置 $s=\langle s_1,\cdots,s_n\rangle$ 处搜索每个并发进程 i 在其程序位置 s_i 处的迁移，就如同对顺序 C 程序切片执行时搜索每个程序位置的所有迁移一样。当到达一个已经遍历完成的并发程序位置 s 时，与对顺序 C 程序的切片执行一样，切片执行无需考虑 s 而直接回溯到其前趋并发程序位置。我们称为种切片执行方法为“并发 C 程序的基本切片执行”，以区别于后续集成了动态偏序缩减的切片执行方法。

对并发程序/模型来说，偏序缩减、尤其是动态偏序缩减[90]是一种简单但有效的状态空间缩减方法。为了将动态偏序缩减方法应用于切片执行，我们先介绍时钟向量的概念，并基于时钟向量快速地鉴别第五章中介绍的“发生前序关系”，从而能够快速得到所谓的回溯点和回溯进程。将动态偏序缩减方法集成到切片执行过程中之后，切片执行在每个并发程序位置处不再同时考虑所有并发进程的迁移，而只考虑任意一个进程的迁移，该并发程序位置处其它必须考虑的进程的迁移是在切片执行后续迁移序列时由动态偏序方法鉴别出的。这样，切片执行的代价以及切片执行生成的切片执行图的状态空间都能够得到极大的缩减。但同时，

当到达一个已经遍历完成的并发程序位置 s 时，切片执行不能简单地回溯，而是必须重新遍历从 s 出发的所有迁移序列，因为只有这样动态偏序缩减方法才能鉴别出全部必要的回溯点和回溯进程。我们称这种切片执行为“集成无状态动态偏序缩减的切片执行”，因为这种切片执行方法不能避免对已经搜索过的并发程序位置和相应迁移序列的重复搜索，正如无状态模型检验一样。但是，该切片执行方法重复搜索已经遍历过的并发程序位置和相应迁移序列时，并不需要计算部分最强后置条件和部分最弱前置条件，也不需要调用定理证明工具进行一阶逻辑公式的判定。它只是简单地遍历确定的迁移序列的集合，以供动态偏序缩减方法鉴别回溯点和回溯进程。因此，其遍历代价很小，从而使得切片执行的代价和生成的切片执行图的状态空间都得到极大的缩减。

在第五章中，我们提出了有状态动态偏序缩减方法，有效地将有状态搜索和动态偏序缩减方法结合在了一起。将有状态动态偏序缩减方法应用到切片执行过程中之后，我们在遍历从某个并发程序位置 s 出发的所有迁移序列时，能够顺便生成 s 的交迭信息总结，以描述从 s 出发的所有迁移路径中迁移的交迭信息。此后，如果并发程序位置 s 在切片执行过程中重遇，则我们根据 s 的交迭信息总结就能确定回溯点和回溯进程，从而避免了对已经遍历完成的并发程序位置 s 和相应迁移序列的重复搜索。我们称这种切片执行为“集成有状态动态偏序缩减的切片执行”，由于它避免了上述重复搜索过程，因此其切片执行代价尤其是空间代价能够得到进一步的缩减。

我们基于上述三种切片执行方法，对由一个并发客户端进程和一个并发服务器端进程组成的并发 *openssl-0.9.6c* 程序进行了验证，实验结果说明了面向并发程序切片执行过程的实用性。

6.1 并发 C 程序模型及其保守近似语义

6.1.1 并发 C 程序模型

本章要考查的并发 C 程序模型同样由一个有限集合的并发进程组成，各并发进程在两个同步原语 *P/V* 的帮助下通过访问共享变量进行交互和通信。

与第五章一样，我们同样将并发 C 程序模型表示为一个标记迁移系统 LTS，但该 LTS 的定义发生了一些变化。令一个并发程序包含 m 个顺序进程 $CP_i=\langle S_i,s_{0i},T_i,\Delta_i\rangle$，其中整数 $i\in\{1,\cdots,m\}$ 表示第 i 个进程，则该并发程序表示为 $CCP=\langle S,s_0,T,\Delta\rangle$，其中：

- $S=S_1\times\cdots\times S_m$ 为各并发进程的程序位置集合的笛尔卡乘积，我们称 $s\in S$ 为模型的一个并发程序位置；

- $s_0=\langle s_{01},\cdots,s_{0m}\rangle$ 为初始并发程序位置；
- $T=T_1\cup\cdots\cup T_m$ 是并发程序模型的迁移语句集合；
- 给定两个并发程序位置 $s_1=\langle s_{11},\cdots,s_{1m}\rangle$、$s_2=\langle s_{21},\cdots,s_{2m}\rangle$ 和一个迁移语句 $t\in T$，我们定义 $\langle s_1,t,s_2\rangle\in\Delta$ 当且仅当存在一个整数 $i\in\{1,\cdots,m\}$ 使得 $\langle s_{1i},t,s_{2i}\rangle\in\Delta_i$ 并且 $\forall 1\le j\le m\wedge j\neq i: s_{1j}=s_{2j}$。

同样，如果迁移语句 $t\in T$ 访问了一个或多个共享变量，我们称 t 为可见（visible）迁移；如果 t 不涉及任何共享变量，我们就称 t 为不可见（invisible）迁移。

并发程序模型 $CCP=\langle S,s_0,T,\Delta\rangle$ 的切片执行图同样定义为 $SEG=\langle\Psi,\longrightarrow\rangle$，其中 $\Psi\subseteq S\times([\![PSP]\!]\cup[\![PWP]\!])$ 为状态集合，$\longrightarrow\subseteq\Psi\times T\times\Psi$ 为迁移集合。该切片执行图 SEG 中的一条迁移序列 π 是一系列迁移 $t_1\cdots t_n$ 组成的序列，其中 $t_1,\cdots,t_n\in T$，并且存在状态 $\psi_1,\cdots,\psi_{n+1}\in\Psi$ 使得 ψ_1 是切片执行图的初始状态 ψ_0，且对每个 $i:1\le i\le n$ 都有 $\psi_i\xrightarrow{t_i}\psi_{i+1}$。仿照第五章的相关定义，我们有：

- π_i 仍然表示迁移 t_i；
- $proc(t)$ 仍然表示迁移 t 所属进程的整数标识；
- $\pi.t$、$t.\pi$ 和 $\pi.\pi'$ 分别表示在迁移序列 π 之后扩展迁移 t、将迁移 t 插入到 π 之前，以及将两条迁移序列 π 和 π' 连接成的一条迁移序列 $\pi.\pi'$；
- $pre(\pi,i)$ 则表示迁移 t_i 之前的切片执行图的状态 ψ_i；
- $last(\pi)$ 则表示 π 到达的最后一个状态 ψ_{n+1}。如果 $\pi=\varnothing$，则 $last(\pi)=\psi_0$。

同样，如果涉及到多条迁移序列，我们就用 πi（如 $\pi 1$、$\pi 2$ 等）标识第 i 条。我们用 Π 表示一系列迁移路径的集合，用 $[\![\Pi]\!]$ 表示所有迁移路径的全集。

我们注意到，相比第五章中定义的并发 C 程序模型，本章中定义的并发程序模型 $CCP=\langle S,s_0,T,\Delta\rangle$ 的每个并发程序位置 $s\in S$ 不包含进程局部变量和全局共享变量取值的相关信息，因此集合 S 不能被称为状态集合。这样的定义是切片执行所需要的，事实上，切片执行的目标就是基于并发程序模型 $CCP=\langle S,s_0,T,\Delta\rangle$ 抽象出作为执行模型的切片执行图 $SEG=\langle\Psi,\longrightarrow\rangle$。相应地，我们称集合 Ψ 为状态空间，并在状态空间 Ψ 中定义迁移序列的概念。

6.1.2 变量抽象下的保守近似语义

首先，为了将变量抽象应用于并发 C 程序模型 $CCP=\langle S,s_0,T,\Delta\rangle$，我们需要对其包含的每个进程 $CP_i=\langle S_i,s_{0i},T_i,\Delta_i\rangle$ 中的所有局部变量进行重命名，例如将局部变量“x”重命名为“x_i”，以保证每个局部变量的全局唯一性，从而根据抽象准则 V 可以判定任意进程的赋值语句和 assume 语句的相关性。

我们可以看出，并发 C 程序模型 $CCP=\langle S,s_0,T,\Delta\rangle$ 的迁移集合 T 中除了包含有每个进程 T_i 的赋值语句和 assume 语句，还包含两个同步原语 P 与 V，下面我们

给出它们对应的变量抽象及部分最强后置条件与部分最弱前置条件的定义。

定义 6.1 在抽象准则 V 下，同步原语 $P(e)$和 $V(e)$的相关性定义如下，其中 e 为 C 语句表达式：

- 如果$Vars(e)\subseteq V$，我们称 $P(e)$和 $V(e)$为抽象准则 V 下的**相关**同步原语；
- 如果$Vars(e)\not\subseteq V$，我们称 $P(e)$和 $V(e)$为抽象准则 V 下的**无关**同步原语。

需要指出的是，如果 $P(e)$是无关的同步原语，则我们保守地认为 $P(e)$并不阻塞，这样在切片执行过程中可能会造成对本来不可行的并发程序执行路径的搜索，但基于伪反例路径的抽象精化会帮助切片执行消除这种不精确。

定义 6.2 给定抽象准则 V，同步原语 $V(e)$和 $P(e)$的部分最强后置条件定义如下：

- $\widetilde{SP}_V(V(e))=\widetilde{SP}_V(e:=e+1)$
- $\widetilde{SP}_V(P(e))=\widetilde{SP}_V(e:=e-1)\circ\widetilde{SP}_V(assume(e>0))$

其中函数组合符号“$\circ$”定义为从右向左组合，即$g\circ h=\lambda x.g(h(x))$。也就是说，同步原语“$P(e)$”被替换为两条语句“$assume(e>0);e:=e-1;$”。值得注意的是，在定义 6.2 中，如果相关的同步原语 $P(e)$被阻塞，即 $e\le 0$，则 $\widetilde{SP}_V(assume(e>0))=False$，从而 $\widetilde{SP}_V(P(e))=False$。也就是说，被阻塞的相关同步原语 P 的部分最强后置条件为 $False$，这属合同步原语 P 的语义，即该原语被阻塞将导致执行路径不可行。

定义 6.3 给定抽象准则 V，同步原语 $V(e)$和 $P(e)$的部分最弱前置条件定义如下：

- $\widetilde{WP}_V(V(e))=\widetilde{WP}_V(e:=e+1)$
- $\widetilde{WP}_V(P(e))=\widetilde{WP}_V(assume(e>0))\circ\widetilde{WP}_V(e:=e-1)$

令 $enabled(\psi)$表示在切片执行图 $SEG=\langle\Psi,\longrightarrow\rangle$的某个状态 $\psi=\langle s,\alpha\rangle$ 处非阻塞的进程标识的集合，其中 s 为状态 ψ 对应的并发程序位置，而 α 则为相应的部分最强后置条件。从第五章我们知道，对一个进程 i，$i\in enabled(\psi)$（我们称进程 i 为非阻塞进程）当且仅当进程 i 存在至少一个非阻塞的迁移 t。根据同步原语 P 的定义，迁移 t 非阻塞当且仅当 $\widetilde{SP}_V(t)(\langle\Omega,\Phi\rangle)\neq False$，从而我们能够求出切片执行图每个状态对应的非阻塞进程集合。但是，根据部分最强后置条件的语义，无关的同步原语 $P(e)$总被判定为非阻塞，这将导致切片执行多搜索一些交迭迁移序列，但时序安全性质验证的正确性仍然能够保证。如果发现一条违背时序安全性质的迁移序列是由于将阻塞的同步原语 $P(e)$判定为非阻塞造成的，则下文中我们将看到，基于精化后的抽象准则重新切片执行时将把同步原语 $P(e)$判定为阻塞。

6.2 并发 C 程序的基本切片执行

为并发 C 程序模型 $CCP=\langle S,s_0,T,\Delta\rangle$ 产生切片执行图 $SEG=\langle \Psi,\longrightarrow\rangle$ 的基本切片执行过程如图 6.1 所示，它与第四章图 4.2 所示的顺序 C 程序的切片执行过程的不同之处仅在于第 3 行。由于并发 C 程序引入了同步原语，因此对切片执行图的每个状态 ψ，都可能有一些进程被阻塞，根据并发程序的语义，阻塞进程不能被考虑，因此在过程第 3 行中我们只选择非阻塞进程的迁移。

```
0  Initially: Explore(⟨s0, True⟩);
Explore(ψ)
{
1  let ψ = ⟨s, α⟩
2  pwp := True;
3  for all ⟨s,t,s'⟩ ∈ Δ such that proc(t) ∈ enabled(ψ) do {
4      let ⟨s',α1⟩,···,⟨s',αn⟩ ∈ Ψ be all states till now in SEG
           corresponding to the concurrent program location s';
5      let α' = SP̃V(t)(α);
6      if (α' ⇒ α1 ∨ ··· ∨ αn) {
7          let A be the minimal subset of {α1,···,αn}
                                  such that α' ⇒ ⋁_{αi∈A} αi
8          for all αi ∈ A set ψ --t--> ⟨s',αi⟩;
9          let pwp' = ⋁_{αi∈A} αi;
10     } else {
11         add state ψ' = ⟨s',α'⟩ to Ψ and set ψ --t--> ψ';
12         let pwp' = Explore(ψ');
13     }
14     pwp := pwp || WP̃V(t)(pwp');
15 }
16 for state ψ = ⟨s,α⟩, set α := pwp;
17 return pwp;
}
```

图 6.1 并发 C 程序的基本切片执行过程

在图 6.1 所示的并发 C 程序切片执行过程的第 3 行中，由于并发 C 程序模型 $CCP=\langle S,s_0,T,\Delta\rangle$ 在并发程序位置 $s=\langle s_1,\cdots,s_m\rangle$ 的迁移包含了每个进程 i 在程序位置 s_i 的迁移，因此我们将遍历所有进程的迁移的交迭执行，并考虑所有交迭执行所到达的后继并发程序位置 $\langle s_1',\cdots,s_i,\cdots,s_m\rangle\cdots\langle s_1,\cdots s_i',\cdots,s_m\rangle\cdots\langle s_1,\cdots,s_i,\cdots,s_m'\rangle$（其中 s_i' 为进程 i 中程序位置 s_i 的后继程序位置），从而保证了对并发程序状态空间搜索

的完备性。另外，在第 14 行中，我们将当前并发程序位置 $s=\langle s_1,\cdots,s_m\rangle$ 的所有后继并发程序位置对应的部分最弱前置条件进行并行组合，因此得到的部分最弱前置条件描述了从 s 出发的所有可行迁移序列的最弱前置条件，从而保证了切片执行的正确性。

对图 6.1 所示的切片执行过程生成的切片执行图 $SEG=\langle \Psi, \longrightarrow \rangle$ 进行模型检验时，如果发现了一条反例迁移序列 π，则我们首先要检查迁移序列 π 的可行性。如果 π 中所有的同步原语 P 都不阻塞，且所有 assume 语句都不冲突，则 π 是可行的。如果我们将 π 中的同步原语 $P(e)$ 用两条等价语句 "$assume(e>0);\ e:=e-1;$" 替换、将 π 中的同步原语 $V(e)$ 用一条等价的语句 "$e:=e+1;$" 替换，则迁移序列 π 的可行性取决于其 assume 语句是否冲突。由于我们事先已经将每个进程的局部变量重命名为全局唯一的变量名，因此我们可以基于第二章介绍的基于最强后置条件的反例路径检查方法对该迁移序列 π 的可行性进行检查。如果 π 实际上在程序执行时是不可行的，即 $SP(\pi)(True)=False$，则我们同样可以基于第二章介绍的抽象准则精化方法推断出必要的程序变量，不管这些程序变量是全局共享变量还是某个进程的私有局部变量，由于它们的名字都是全局唯一的，因此能够保证基于精化后的抽象准则进行切片执行时 π 一定为不可行的迁移序列。

6.3 集成无状态动态偏序缩减的切片执行

本节中，我们介绍基于时钟向量的无状态动态偏序缩减方法在切片执行过程中的应用，我们假设并发 C 程序由 m 个并发进程 $p_1,\cdots,p_m\in P$ 组成，其中 P 为一个整数进程标识集合。

6.3.1 时钟向量

时钟向量（Clock Vector）[100]是从进程标识到当前迁移序列 π 上所有迁移的整数索引的一个映射，即 $CV=P\mapsto \mathbf{N}$，用于表示发生前序关系（Happens-Before Ordering Relation）[90]。根据文献[90]，对给定的迁移序列 π，我们维护其时钟向量 $CV(\pi)=\langle L,C\rangle$，其中 $L(o)$ 记录了迁移序列 π 中访问共享变量 o 的最后一个非同步原语 V 的迁移的索引，而 $C(p)$ 和 $C(o)$ 则用来表示发生前序关系。同样地，我们用符号 $[\![CV]\!]$ 表示所有时钟向量的全集。

我们为每个进程 p 维护一个时钟向量 $C(p)=\langle cp_1,\cdots,cp_m\rangle$，其中 cp_i 为迁移序列 π 中进程 p 的最后一个满足条件 $\pi_{cp_i}\rightarrow_\pi p$ 的迁移的索引。换言之，关系 $\pi_i\rightarrow_\pi p$ 成立当且仅当 $i\le C(p)(proc(\pi_i))$。同时，我们也为每个共享变量 o 维护一个时钟向量 $C(o)=\langle co_1,\cdots,co_m\rangle$，其中 co_i 为迁移序列 π 中进程 i 的最后一个访问共享变量 o

的迁移的索引。

例如，对下列迁移序列π：

π = p_1:x++; p_2:y++; p_1:y++; p_2:x++;

根据定义，它的四个迁移分别简记为π_1、π_2、π_3和π_4，则我们有$L(x)=4$、$L(y)=3$，表示π中最后一个访问共享变量x和y的迁移分别是π_4和π_3。另外，$C(p_1)=\langle 3,2\rangle$、$C(p_2)=\langle 1,4\rangle$，第一个时钟向量表示满足发生前序关系$\pi_i \rightarrow_\pi p_1$的进程$p_1$和进程$p_2$的最后一个迁移分别为$\pi_3$和$\pi_2$，而第二个时钟向量则表示满足发生前序关系$\pi_i \rightarrow_\pi p_2$的进程$p_1$和进程$p_2$的最后一个迁移分别为$\pi_1$和$\pi_4$。此外，我们还有$C(x)=\langle 1,4\rangle$、$C(y)=\langle 3,2\rangle$，表示最后一个访问共享变量$x$和$y$的进程$p_1$和进程$p_2$的迁移分别为$\pi_1$、$\pi_4$和$\pi_3$、$\pi_2$。

根据迁移序列π的时钟向量，我们可以很容易地判定条件“$\exists i = max(\{i \in dom(\pi) \mid \pi_i \text{ is dependent and may be co-enabled with } next(s,p) \text{ and } i \not\rightarrow_\pi p\})$”是否满足[90]，该条件等价于“$i \neq 0 \wedge i \not\leq C(p)(proc(\pi_i))$”，其中$i = L(obj(next(s,p)))$。

根据文献[90]，迁移序列πt（即在迁移序列π之后附加迁移t后得到的新迁移序列）对应的时钟向量$CV(\pi t)$可以从迁移序列π对应的时钟向量$CV(\pi)$计算得到。我们首先定义如下三个函数：

$$\max(\langle c_1,\cdots,c_m\rangle,\langle c'_1,\cdots,c'_m\rangle) \triangleq \langle \max(c_1,c'_1),\cdots,\max(c_m,c'_m)\rangle$$

$$\langle c_1,\cdots,c_m\rangle[p_i := c'_i] \triangleq \langle c_1,\cdots,c_{i-1},c'_i,c_{i+1},\cdots,c_m\rangle$$

$$\perp \triangleq \langle 0,\cdots,0\rangle$$

根据定义，从$CV(\pi)=\langle L,C\rangle$得到$CV(\pi t)=\langle L',C'\rangle$的算法如下：

let $o = obj(t)$ and let $p = proc(t)$;
let $cv = \max(C(p),C(o))[p := |\pi t|]$;
let $C' = C[p := cv, o := cv]$;
let $L' = (t \text{ is primitive } V)\ ?\ L :\ L[o := |\pi t|]$;

6.3.2 集成无状态动态偏序缩减的切片执行

基于并发C程序模型$CCP=\langle S,s_0,T,\Delta\rangle$产生切片执行图$SEG=\langle \Psi, \longrightarrow \rangle$的集成无状态动态偏序缩减的切片执行过程如图6.2所示。该过程仍然集成了第三章介绍的搜索复用框架和第四章介绍的部分最弱前置条件，从而尽可能地缩减了切片执行的代价和所生成切片执行图的状态空间。

在图6.2所示的集成无状态动态偏序缩减的切片执行过程中，其过程框架和很多代码都是我们在以前的章节中介绍过的，下面我们对几点内容进行必要的补充说明。首先，在调用过程Explore之前，我们需要设置切片执行图$SEG=\langle \Psi, \longrightarrow \rangle$的初始状态$\psi_0=\langle s_0,True\rangle$，其中$s_0$为并发程序模型$CCP=\langle S,s_0,T,\Delta\rangle$的初始并发程

序位置。另外，我们要设置 $backtrack(\psi_0) := \{p\}$ 以及 $done(\psi_0) := \varnothing$，其中进程 $p \in enabled(\psi_0)$ 为状态 ψ_0 处的任意一个非阻塞的进程。

```
0  Initially: Explore(∅, λx.0, λx.⊥);
backtrack, done: Ψ ↦ 2^N;
Explore(π, L, C)
{
1  let ψ = last(π);
2  pwp := True;
3  if (∃p ∈ backtrack(ψ) \ done(ψ)) {
4      add p to done(ψ) and let ψ = ⟨s,α⟩;
5      for all ⟨s,t,s'⟩ ∈ Δ and proc(t) = p do {
6          let o = obj(t) and let p = proc(t);
7          let cv = max(C(p), C(o))[p := |π.t|];
8          let C' = C[p := cv, o := cv];
9          let L' = (t is primitive V) ? L : L[o := |π.t|];
10         let ⟨s',α_1⟩,···,⟨s',α_n⟩ ∈ Ψ be all states till now in SEG
               corresponding to the concurrent program location s';
11         let α' = SP̃_V(t)(α);
12         if (α' ⇒ α_1 ∨ ··· ∨ α_n) {
13             let A be the minimal subset of {α_1,···,α_n}
                                    such that α' ⇒ ∨_{α_i∈A} α_i
14             for all α_i ∈ A set ψ --t--> ⟨s',α_i⟩;
15             RefineBacktrackStateless(π.t, L', C', ∅);
16             let pwp' = ∨_{α_i∈A} α_i;
17         } else {
18             add state ψ' = ⟨s',α'⟩ to Ψ and set ψ --t--> ψ';
19             RefineBacktrackStateless(π.t, L', C', ∅);
20             if (∃p' ∈ enabled(ψ'))
21                 set backtrack(ψ') := {p'} and done(ψ') := ∅;
22             let pwp' = Explore(π.t, L', C');
23         }
24         pwp := pwp || W̃P_V(t)(pwp');
25     }
26 }
27 for state ψ = ⟨s,α⟩, set α := pwp;
28 return pwp;
}
```

图 6.2 集成无状态动态偏序缩减的切片执行过程

接下来，我们通过 Explore(∅, λx.0, λx.⊥)过程调用从初始状态 ψ_0 开始构建

整个切片执行图，其中 $\lambda x.0$ 和 $\lambda x.\perp$ 分别表示空迁移序列对应的时钟向量 $CV(\varnothing)=\langle L,C\rangle$ 的两个元素 L 和 C。另外，在图 6.2 所示过程的第 6 – 9 行中，我们根据迁移序列 π 的时钟向量，来计算迁移序列 $\pi.t$ 对应的时钟向量，其计算方法在 6.3.1 小节中已经给出。

我们注意到，在每个并发程序位置 s 处，图 6.2 所示的切片执行都只任选一个非阻塞的进程执行其迁移（见第 3、5 和第 20 – 21 行），该并发程序位置 s 处的其它必需考虑的非阻塞进程是在对从 s 出发的所有迁移序列的遍历过程中由过程 RefineBacktrackStateless 鉴别出的（该过程在图 6.3 中定义）。另外，虽然从每个并发程序位置 s 出发的某些迁移序列并没有被考虑，但我们能够保证它的至少一条具有相同偏序的等价迁移序列被遍历了，因此基于从每个并发程序位置 s 出发被遍历的迁移序列的集合得到的部分最弱前置条件仍然是正确的。

根据文献[90]介绍的无状态动态偏序缩减方法，每当切片执行到达一个新的状态 ψ 时（令其相应的迁移序列为 π，即 $\psi=last(\pi)$），都要调用图 6.3 所示的过程 RefineBacktrackStateless（见图 6.2 所示切片执行过程的第 15 行和第 19 行），以检查状态 ψ 处每个进程的下一个迁移 $next(\psi,p)$ 是否能够发生在 π 的某个迁移 π_i 之前，如果可以的话，我们就需要将进程 p（可能还有其它进程）加入到回溯点 $pre(\pi,i)$ 的回溯进程集合 $backtrack(pre(\pi,i))$ 中。

```
RefineBacktrackStateless(π, L, C, D)
{
1  let ψ = last(π);
2  for all processes p {
3      let i = L(obj(next(ψ, p)));
4      if (i ≠ 0 ∧ i ⋪ C(p)(proc(π_i))) {
5          if (p ∈ enabled(pre(π,i))) add p to backtrack(pre(π,i));
6          else add enabled(pre(π,i)) to backtrack(pre(π,i));
7      }
8  }
9  let D' := D ∪ {ψ};
10 for all t and ψ' such that ψ --t--> ψ' do {
11     let o = obj(t) and let p = proc(t);
12     let cv = max(C(p), C(o))[p := |π.t|];
13     let C' = C[p := cv, o := cv];
14     let L' = (t is primitive V) ? L : L[o := |π.t|];
15     if (ψ' ∉ D') RefineBacktrackStateless(π.t, L', C', D');
16 }
}
```

图 6.3　无状态动态偏序缩减的回溯点和回溯进程鉴别过程

在图 6.3 所示过程的第 2 行，我们考查所有进程，而不论其是否阻塞。根据时钟向量的定义，$i \neq 0 \wedge i \not\leq C(p)(proc(\pi_i))$ 等价于条件“$\exists i = max(\{i \in dom(\pi) \mid \pi_i$ is dependent and may be co-enabled with $next(s,p)$ and $i \not\rightarrow_\pi p\})$”，因此如果该条件成立，则迁移 π_i 与迁移 $next(\psi,p)$ 可能是相关的，从而我们确定了回溯点 $pre(\pi,i)$，相应回溯进程的确定如图 5－6 行所示。

如果状态 ψ 是一个已经遍历完成的状态，则切片执行不会对其重新遍历（见图 6.2 所示切片执行过程第 12 行中条件 $\alpha' \Rightarrow \alpha_1 \vee \cdots \vee \alpha_n$ 满足的情况），但为了得到迁移序列 π 的所有回溯点和回溯进程，我们需要在过程 RefineBacktrackStateless 中遍历切片执行图中从状态 ψ 出发的所有迁移序列（见图 6.3 中第 10－16 行），并更新每遍历一个迁移 t 得到的新迁移序列 πt 对应的时钟向量 $CV(\pi t) = \langle L', C' \rangle$。

最后，如果切片执行图 SEG 中从状态 ψ 出发的某条迁移序列 π' 是一个圈，即 $\psi \xrightarrow{\pi'} \psi$，则令遍历完迁移序列 $\pi\pi'$ 后的时钟向量为 $CV(\pi\pi') = \langle L', C' \rangle$，当第二次遍历 π' 时，与 π' 上每个迁移相关的迁移序列 $\pi\pi'$ 的最后一个迁移必然仍是 π' 的迁移，因此对 π' 的二次遍历不会影响到对 π 的回溯点和回溯进程的鉴别。所以，我们引入一个集合 D，用于记录当前遍历的所有状态，从而避免对同一个状态的多次遍历（见第 15 行）

6.4 集成有状态动态偏序缩减的切片执行

6.4.1 基于时钟向量的有状态动态偏序缩减方法的实现

如果一条迁移序列 π 到达一个已经遍历完成的状态 ψ，则可以调用第五章中给出的 RefineBacktrackSII 过程，根据状态 ψ 的交迭信息总结 $SII(\psi)$ 鉴别迁移序列 π 的回溯点和回溯进程。RefineBacktrackSII 过程中用到了发生前序关系，例如判断 $\pi_i \not\rightarrow_\pi \langle p,o \rangle$ 是否成立等，我们可以基于时钟向量以线性复杂度实现这种判断[90]。

基于时钟向量的有状态动态偏序缩减方法的实现如图 6.4 所示，我们将其重命名为 RefineBacktrackStateful，与无状态动态偏序缩减方法 RefineBacktrackStateless 相对应。

图 6.4 所示过程的大部分代码都在第五章中详细介绍了，修改的地方主要体现在第 3 行和第 8 行，其中第 3 行对应条件 $\pi_i \not\rightarrow_\pi \langle p,o \rangle$，我们主要解释第 8 行。第 8 行中涉及到一个函数 $MaxDepIndex(C, \langle p',o' \rangle)$，令 $CV(\pi) = \langle L, C \rangle$，则该函数返回迁移序列 π 的最后一个必须发生在迁移 $\langle p',o' \rangle$ 之前的迁移 π_i 的索引 i。令 $C(p') = \langle cp_1, \cdots, cp_m \rangle$ 以及 $C(o') = \langle co_1, \cdots, co_m \rangle$，由于时钟向量 $C(p')$ 表示必须发生在进程 p' 之前 π 的每个进程的最后一个迁移的索引，而时钟向量 $C(o')$ 则表示 π 的每个进程的最后一个访问共享变量 o' 的迁移的索引，因此基于时钟向量 C，我们知

道 $MaxDepIndex(C,\langle p',o'\rangle)=\max\{cp_1,\cdots,cp_m,co_1,\cdots,co_m\}$。

```
BT : N×[[Object]] ↦ 2^N;
BL : N×[[Object]] ↦ N;
RefineBacktrackStateful(π, L, C, SII)
{
1   for all ⟨p,o⟩ ∈ dom(OI) do{
2       let i = L(o);
3       if (i ≠ 0 and i ⋠ C(p)(proc(π_i)))
4           BT(⟨p,o⟩) := OI(⟨p,o⟩), BL(⟨p,o⟩) := i;
5   }
6   for all ⟨p',o'⟩ ∈ dom(DI) do {
7       for all k such that k ∈ BT(⟨p',o'⟩) and k ∈ DI(⟨p',o'⟩) do
8           if (BL(⟨p',o'⟩) ≤ MaxDepIndex(C,⟨p',o'⟩))
9               remove k from BT(⟨p',o'⟩);
10  }
11  for all ⟨p,o⟩ ∈ dom(BT) do
12      if (BT(⟨p,o⟩) ≠ ∅) {
13          let i = BL(⟨p,o⟩);
14          if (p ∈ enabled(pre(π,i)))
15              add p to backtrack(pre(π,i));
16          else add enabled(pre(π,i)) to backtrack(pre(π,i));
17  }
}
```

图 6.4 基于时钟向量的有状态动态偏序缩减方法的实现

6.4.2 集成有状态动态偏序缩减方法的切片执行过程

集成有状态动态偏序缩减的切片执行过程如图 6.5 所示，其目标同样是基于并发 C 程序模型 $CCP=\langle S,s_0,T,\Delta\rangle$ 产生切片执行图 SEG。为了应用第五章介绍的有状态动态偏序缩减方法，图 6.5 所示的切片执行过程改为基于栈实现。

为了简化集成了部分最弱前置条件的切片执行过程的实现复杂度，我们扩展切片执行图 $SEG=\langle\Psi,\longrightarrow\rangle$，将其状态集合 Ψ 的定义从 $\Psi\subseteq S\times([\![PSP]\!]\cup[\![PWP]\!])$ 扩展到 $\Psi\subseteq S\times([\![PSP]\!]\cup[\![PWP]\!])\times[\![PWP]\!]$。

图 6.5 所示的切片执行过程增强了两个全局映射 SII 和 CV，SII 记录每个状态对应的交迭信息总结（详见第五章），CV 则记录了每条迁移序列对应的时钟向量。如果某迁移序列 π 到达一个已经遍历完成的状态 ψ，则调用 6.4.1 小节介绍的过程 RefineBacktrackStateful 可以根据 $CV(\pi)$ 和 $SII(\psi)$ 得到 π 的回溯点和回溯过程，从而能够避免对从状态 ψ 出发的所有迁移序列的重复遍历。

backtrack, *done*: $\Psi \mapsto 2^N$;
Stack: A list of transition sequences;
SII: $\Psi \mapsto [\![SII]\!]$;
CV: $[\![\Pi]\!] \mapsto [\![CV]\!]$;
SlicingExecution($CCP = \langle S, s_0, T, \Delta \rangle$)
{
1 set $\psi_0 = \langle s_0, True \rangle$, $CV(\varnothing) = \langle \lambda x.0, \lambda x.\perp \rangle$ and *Stack*.push($\varnothing$);
2 if ($\exists p \in enabled(\psi_0)$) $backtrack(\psi_0) := \{p\}$;
3 while (! *Stack*.empty( )){
4 let $\pi = Stack.\text{top}()$ and let $\psi = \langle s, psp, pwp \rangle = last(\pi)$;
5 *Stack*.pop( );
6 if ($\exists p \in backtrack(\psi) \setminus done(\psi)$) {
7 $done(\psi) := done(\psi) \cup \{p\}$;
8 let $t = next(\psi, p)$ and let $CV(\pi) = \langle L, C \rangle$;
9 let $o = obj(t)$ and let $p = proc(t)$;
10 let $cv = \max(C(p), C(o))[p := |\pi.t|]$;
11 let $C' = C[p := cv, o := cv]$;
12 let $L' = (t \text{ is primitive } V) \ ? \ L : \ L[o := |\pi.t|]$;
13 let $\pi' = \pi.t$ and set $CV(\pi') := \langle L', C' \rangle$;
14 let $psp' = \widetilde{SP}_V(t)(psp)$;
15 let $\langle s', psp_1, pwp_1 \rangle, \cdots, \langle s', psp_n, pwp_n \rangle \in \Psi$ be all states
till now in *SEG* corresponding to the location s';
16 if ($psp' \Rightarrow psp_1 \vee \cdots \vee psp_n$) {
17 let A be the minimal subset of $\{psp_1, \cdots, psp_n\}$
such that $psp' \Rightarrow \bigvee_{psp_i \in A} psp_i$;
18 let $pwp' = \bigvee_{psp_i \in A} psp_i$;
19 set $last(\pi') := \langle s', psp', pwp' \rangle$;
20 for all $psp_i \in A$ set $\langle s, psp, pwp \rangle \xrightarrow{t} \langle s', psp_i, pwp_i \rangle$;
21 RefineBacktrackStateful($\pi', L', C', SII(s')$);
22 }else {
23 add state $\langle s', psp', True \rangle$ to Ψ;
24 set $\langle s, psp, pwp \rangle \xrightarrow{t} \langle s', psp', True \rangle$;
25 RefineBacktrackStateless(π', L', C', $\varnothing$);
26 if($\exists p \in enabled(s')$) $backtrack(s') := \{p\}$;
27 }
28 *Stack*.push(π');
29 }else if ($\exists \pi', t: \ \pi'.t = \pi$) {
30 for current state $\psi = \langle s, psp, pwp \rangle$, set $psp := pwp$;
31 let $\psi' = \langle s', psp', pwp' \rangle = last(\pi')$;
32 set $pwp' := pwp' \| pwp$;
33 *Stack*.push(π');

```
34          let oldSII = SII(ψ');
35          set SII' := SII(ψ);
36          UpdateSII(SII', t);
37          MergeSII(SII(ψ'), SII');
38          if (oldSII ≠ SII(ψ'))
39              for all π'' such that ψ0 --π''--> ψ' in SEG do {
40                  RefineBacktrackStateful(π'', L', C', SII(s'));
41                  Stack.push(π'');
42              }
43      }
44  }
}
```

图 6.5 集成有状态动态偏序缩减的切片执行过程

初始时，我们在图 6.5 所示过程的第 1 – 2 行进行必要的初始化，同时，我们假设对于每个未初始化的切片执行图 $SEG=\langle \Psi, \longrightarrow \rangle$ 中的状态 $\psi \in \Psi$，都有 $backtrack(\psi)=done(\psi)=SII(\psi)=\varnothing$。

尽管图 6.5 所示的集成有状态动态偏序缩减的切片执行过程是基于栈实现的，但它与图 6.3 所示的集成无状态动态偏序缩减的切片执行过程的总体框架是基本相同的，其不同点在于到达一个已经遍历完成的切片执行图状态时采取的措施（即第 16 行的条件 $psp' \Rightarrow psp_1 \vee \cdots \vee psp_n$ 为真的情况）。图 6.5 所示的切片执行过程调用 RefineBacktrackStateful 来得到相应的回溯点和回溯进程，而无需像图 6.3 所示过程必须基于 RefineBacktrackStateless 对相应迁移序列进行重复的遍历。值得注意的是，在过程第 19 行我们设置 $last(\pi') := \langle s', psp', pwp' \rangle$，其中 pwp' 的定义在第 18 行。由于回溯进程集合 $backtrack(\langle s', psp', pwp' \rangle)$ 初始默认为空，因此接下来搜索该状态时就会回溯，同时将部分最弱前置条件 pwp' 传递给其前趋状态，从而满足我们的需要。

如果当前状态 ψ 对应的所有回溯进程都被遍历完毕（即第 6 行的条件 $\exists p \in backtrack(\psi) \setminus done(\psi)$ 为假），则过程跳转到第 29 行。如果当前状态 $\psi=\langle s, psp, pwp \rangle$ 有前趋状态（即 $\psi \neq \psi_0$），则由于对状态 ψ 的遍历已经完成，且切片执行图中从 ψ 出发的所有迁移序列都不违背给定的时序安全性质（否则切片执行过程会中止退出），我们就用其对应的部分最弱前置条件 pwp 代替其部分最强后置条件 psp（第 30 行），从而降低和减小切片执行的代价和生成切片执行图的状态空间，其正确性和相关讨论请见第 4 章。另外，在第 32 行我们还将状态 ψ 处的部分最弱前置条件传递到其前趋状态 ψ'，以保证当 ψ' 被遍历完成时，其部分最弱前置条件 pwp' 是正确的。值得注意的是，由于在第 23 – 24 行我们将新到达的状态 $last(\pi')=\langle s', psp', True \rangle$ 的部分最弱前置条件设为 $True$，这样如果 s' 是程序的

最后一个并发程序位置没有后继迁移，则其部分最弱前置条件保持为 *True*，从而满足我们的需要。

最后，图 6.5 所示过程第 34 – 37 行更新交迭信息总结 $SII(\psi)$，并将更新的交迭信息总结加入到 $SII(\psi')$，其实现在第五章 5.3.2 节中进行了详细讨论。最后要指出的是，依赖于某个状态 ψ' 的所有迁移序列都可由生成的切片执行图 *SEG* 得到，如图 6.5 第 39 行所示。

6.5 建模环境与实验结果

6.5.1 并发 C 程序建模环境

在对并发 C 程序进行验证之前，我们需要对其进行稍微的修改，以构建完整的并发程序模型。我们提供了如下预定义的函数作为建模环境，供用户进行建模：

- *init_proc*( )：该函数是验证工具提供给用户进行模型初始化用的，在该函数中可以进行进程创建，以及初始化共享和私有变量等工作；
- *pid* = *create_proc*(*procName*)：该函数用于创建一个并发进程，该进程执行函数名为 *procName* 的函数的代码，它返回被创建进程的整数 ID；
- *set_var*(*pid*, *var*, *value*)：该函数用于初始化每个进程的私有变量，即将进程 ID 为 *pid* 的进程中的名字为 *var* 的私有变量的值初始化为 *value*，全局变量则直接初始化；
- *P*(*var*)与 *V*(*var*)：这两个函数实现了同步原语 *P* 与 *V*。

```
#define PROC_NUM 13
int table[128];
int mutex[128]; //For atomic change
void thread(int tid)
{
    int m = 0, w, h, cas;
    while(1){
        if(m < 4){
            m++;
            w = m*11+tid;
        }else return;
        h = (w * 7) % 128;
AGAIN:
        P(mutex[h]);
        if(table[h] == 0){
            table[h] = w;
            cas = 1;
        }else cas = 0;
        V(mutex[h]);
        if(cas == 0){
            h = (h + 1) % 128;
            goto AGAIN;
        }
    }
}

void init_procs()
{
    int i, pid;
    for(i = 0; i < 128; i++) table[i] = 0;
    for(i = 0; i < 128; i++) mutex[i] = 1;
    for(i = 0; i < PROC_NUM; i++){
        pid = create_proc(thread);
        set_var(pid, tid, pid);
    }
}
```

图 6.6 基于并发 C 程序建模环境改写后的 Indexer 程序

图 6.6 给出了第五章介绍的 Indexer 程序基于上述建模环境改写后的并发 C 程序的例子，我们可以看出：函数 *P* 与 *V* 直接用于进程的代码中进行同步操作；函数 *create_proc* 用于初始化函数 *init_procs* 中，共创建了 PROC_NUM 个进程，每个

进程都执行相同的代码，即函数 *thread* 所示的代码；函数 *set_var* 将每个进程的 *tid* 变量（即函数 *thread* 的形参）初始化为分配给该进程的 ID。

6.5.2 实验结果

为了检验和比较本章介绍的三种切片执行过程，我们选择了两个例子，第一个例子是一个没有同步原语的并发 C 程序，第二个例子则是基于同步原语的并发 SSL 协议的初始握手过程。所有实验数据都在一台 1.6GHz Athlon CPU 和 1GB 内存的机器上产生，其软件环境是 Windows 2000 + Cygwin 2.427。

第一个例子是由若干相同的并发进程组成的玩具并发 C 程序，其每个并发进程都由下列四个赋值语句组成：

x++; *y*++; *x*--; *y*--;

其实现结果如表 6.1 所示，其中："Trans" 表示所有可见迁移数量，注意如果同一条语句被多次执行，则其也被多次统计；"Time" 指遍历并发程序的全部状态空间所花费的时间；"Procs" 是组成并发程序的并发进程的数量；"Stateless DPOR" 指应用了动态偏序缩减方法的无状态搜索，即无状态地遍历状态空间；"Base SE" 指基本的切片执行方法；"Stateless SE" 指集成了无状态动态偏序缩减的切片执行方法，其中 "Nml Trans" 是通常概念的迁移，包括符号化迁移条件计算和动态偏序信息计算，而 "Dmy Trans" 则只包含动态偏序信息计算（从而计算代价比 "Nml Trans" 低得多）；最后 "Stateful SE" 指集成了有状态动态偏序缩减的切片执行方法。

表 6.1 示例并发程序的实验结果

Procs	Stateless DPOR		Base SE		Stateless SE			Stateful SE	
	Trans	Time	Trans	Time	Nml Trans	Dmy Trans	Time	Trans	Time
4	3925944	164.5	623	0.08	623	988	0.17	623	0.16
5	-	-	3123	0.5	3123	7193	1.3	3123	1.2
6	-	-	15623	3.6	15623	47600	8.8	15623	8.1
7	-	-	78123	23.7	78123	297209	60.6	78123	54.0

我们可以看出，尽管该示例并发 C 程序的每个并发进程仅包含四条语句，但当并发进程数量增长时，其交迭执行路径的数量增长非常快，体现为 "Stateless DPOR" 无状态搜索方法的迁移数量非常巨大。对比后三种有状态搜索方法，我们发现在该例子中，基本切片执行过程的效率比其它两种策略强很多，这是由于不含同步原语的并发进程往往在每个状态都需要遍历所有进程的迁移，而这正是基本切片执行的状态空间遍历思想。对集成了无状态和有状态动态偏序缩减的切片执行来说，不仅需要大量的时间来鉴别回溯点和回溯进程，而且还要花费大量的

存储空间来存储时钟向量和交迭信息总结，从而导致效率的下降。最后要指出的是，三种有状态搜索方法的迁移数量是相等的，这是由于所有具有相同偏序的执行路径都具有相同的符号状态，从而能够被有状态搜索缩减掉。

上述示例程序仅代表了一种特殊情况，而实际上，几乎所有的实用程序都使用了同步原语进行同步操作，因此集成无状态和有状态动态偏序缩减方法的切片执行效率和生成的切片执行图的状态空间都要远远优于基本切片执行方法，例如在对并发 SSL 程序的验证实验中，基本切片执行方法无法在一个小时内完成验证。

我们验证的并发 SSL 程序由两个并发进程组成，即一个 SSL 客户端进程和一个 SSL 服务器端进程，它们的源代码都来自 SSL 协议在 Linux 操作系统中的实现程序 *openssl-0.9.6c*，每个进程都有大概 2000 行代码。为了模拟客户进程与服务器进程之间的通信网络，我们构建了两条消息队列，每条消息队列都可被用来在服务器端与客户端之间双向传递消息。每条消息队列能够存储一个消息，在 *P/V* 同步原语的保护下供客户端和服务器端的进程读写访问。验证时，我们只考虑消息的类型，因此我们重写了两个函数“*ssl3_do_write*” 和“*ssl3_get_message*”，这两个函数在 *openssl-0.9.6c* 程序中被用来发送和接收网络消息，重写后的相应函数则分别向两个消息队列发送和接收消息。

表 6.2 基于并发程序 openssl-0.9.6c 的 SSL 协议初始握手过程验证

Prop.	Stateless SE					Stateful SE			
	All Trans	Visible Trans	Dummy Trans	TPC	Time	All Trans	Visible Trans	TPC	Time
1	23188	1358	7888	12464	64.6	15300	1358	12464	51.7
2	8434	649	369	12077	60.7	8065	649	12077	57.9
3	5941	413	315	10033	27.2	5626	413	10033	27.3
4	107405	4315	55568	101737	715.9	51837	4315	101737	589.0
5	73263	2587	39254	296009	1028.3	34009	2587	296009	905.3
6	2719	130	1202	7308	28.1	1517	130	7308	27.6

根据 SSL 规范[101]，我们描述了一些必须被 SSL 协议的初始握手过程满足的时序安全性质，并使用集成了有状态动态偏序缩减方法的切片执行过程进行了验证。所有的性质都被验证为成立，其结果列于表 6.2 中。其中“Prop.”是待验证的时序安全性质：性质 1 规定握手过程必须由客户端发起；性质 2 规定服务器端状态机的两个状态 SR_KEY_EXCH 和 SR_CERT_VRFY 必须交迭到达，表示密钥的交换和验证必须交替进行；性质 3 说明另外两个服务器端的状态 SW_KEY_EXCH 和 SW_CERT 是有可能不交迭到达的；性质 4 规定客户端的 HELLO 和 FINISH 消息之间必须有一个服务器端的 HELLO 消息，同时客户端的 FINISH 消息之后一定不能有服务器端的 DONE 消息。由于 SSL 协议的初始握手过程既可以进行身份论证，

也可以不进行身份论证，性质 5 和性质 6 分别给出了这两个流程对应的状态机。

对表 6.2 标题的其它项，“All Trans”是切片执行的所有迁移的数量，“Visible Trans”是所有可见迁移的数量，“Dummy Trans”则是由 6.3.2 节中图 6.3 给出的过程 RefineBacktrackStateless 重新遍历切片执行图中从某个状态出发的所有迁移序列的迁移数量。对“Dummy Trans”的遍历不用计算最强后置条件和最弱前置条件，同时也不会调用定理证明工具，从而具有比正常迁移小得多的计算代价。最后，“TPC”是切片执行过程的定理调用次数，“Time”是切片执行过程花费的时间，以秒为单位。“Stateless SE”与“Stateful SE”两种策略的迁移数量和定理证明工具的调用次数都是相同的，这是由于二者的差别仅在于遇到相同状态时的处理方法不同。

根据表 6.2 的数据，“Stateless SE”与“Stateful SE”两种策略的时间开销相差不大，这也说明了切片执行的绝大部分时间开销被用在计算部分最强后置条件和部分最弱前置条件以及调用定理证明工具进行公式判定。尽管从时间花费上来看二者相差不大，但由于“Stateless SE”策略需要存储所有并发状态以供过程 RefineBacktrackStateless 重新遍历，而“Stateful SE”则能够在一个状态搜索完毕后立即将其存储空间释放，因此“Stateful SE”能够大大节省验证过程的空间开销，从而提高切片执行的可扩展性。

6.6 相关研究工作

偏序缩减方法最早在 Spin[26, 32]等通用模型检验工具中得到应用，正如文献[102]中指出的，偏序缩减方法是 Spin 用于对付状态空间爆炸问题的主要武器之一。早期的 Spin 是基于深度优先的方法搜索状态空间的，因此偏序缩减方法也是基于深度优先方式的[103, 104]，能够支持对所有 LTL 时序逻辑公式的模型检验。从版本 4.0.0 开始，Spin 提供了宽度优先搜索的选项，以供用户找到最短的反例路径。因此，文献[102]提出了能够与 Spin 的宽度优先搜索同时进行的新偏序缩减算法。

Java PathFinder（JPF）也采用了偏序缩减方法，以更好地支持对并发 Java 程序的验证[105]。JPF 实现了一个支持状态存储和恢复的虚拟机，并在其基础上遍历并发 Java 程序的状态空间。为了应用偏序缩减方法，JPF 将一直执行同一线程的代码，只到下一个迁移是与调试相关的指令，或者下一条指令将产生“非确定性”的结果为止。换言之，JPF 只有在当前线程的下一条指令可能与其它线程交互时，才进行线程的调度，从而避免了不必要的交迭执行。

Zing[45, 46]同样也采用了偏序缩减方法，能够在保证正确性的前提下将状态空间缩减指数倍。Zing 的偏序缩减方法采用了 Lipton 的缩减理论[106]，基于这样一个事实，即在同步良好的并发程序中，每个进程都可以看作是由一序列的原子事务

构成的，从而在遍历状态空间时，只要能够区分出每个进程的所有原子事务，再在事务的边界进行进程调度即可。例如，文献[107]给出了识别基于锁的共享变量方式的并发进程的所有原子事务的方法。

6.7 小结

本章首先提出了基于 *P/V* 同步原语的共享内存并发 C 程序模型，以及基于该并发 C 程序模型的保守近似语义，并将切片执行扩展到对该并发程序模型进行验证，我们还保证了前面章节提出的搜索复用框架以及部分最弱前置条件等技术可以无缝集成到面向并发程序模型的切片执行中，从而保证了验证效率。

接下来，我们提出了三种切片执行策略，分别是基本的切片执行、集成无状态动态偏序缩减的切片执行和集成有状态动态偏序缩减的切片执行，逐渐将第五章提出的有状态动态偏序缩减方法集成到切片执行过程中，从而使得并发程序模型的状态空间得到大幅度缩减。

最后，我们通过两个实验比较了三种切片执行方法，其中包括对来自 *openssl-0.9.6c* 的并发程序的实验，实验结果显示了切片执行在验证并发 C 程序过程中的高效和实用性。

第七章　面向切片执行的轻量级判定过程

从本论文前几章的实验数据中可以看到，切片执行需要大量地调用判定过程（Decision Procedure）对切片执行过程中产生的一阶逻辑公式进行判定。更一般地，除了切片执行工具外，其它基于谓词抽象[55]的面向程序源代码的验证工具，如SLAM[41-43]、Zing[44-46]、BLAST[47]、MAGIC[48, 49]及ComFoRT[50, 52, 108]等，也需要大量地调用判定过程对抽象和验证过程中产生的大量判定问题进行判定，高效、轻量级的判定过程成为提高这些验证工具效率的关键。当前面向程序源代码的验证工具都采用轻量级的定理证明工具作为其判定过程。例如，BLAST、MAGIC和ComFoRT使用了轻量级的全自动定理证明工具Simplify[72]，而SLAM则设计了专用的定理证明工具Zapato[109]以提高判定效率。

包括切片执行工具在内，上述面向程序源代码的验证工具所产生的绝大多数判定问题都可表示为一阶逻辑公式形式的整数线性判定问题，其中谓词抽象产生的判定问题是对所有变元隐式存在量化的，即等价于判定是否存在这些变元的一个取值，使得判定公式满足。因此提高整数线性判定工具的性能可以有效地提高面向源代码形式验证的效率。描述整数线性判定问题的一阶逻辑公式（又称为Presburger算术）形式化地定义为如图7.1所示的形式，其可满足性问题是可判定的（Presburger，1929）。注意，Presburger公式中的数字（*numeral*）只包含正整数，但我们可以使用表达式y-z代替变元x从而将公式中的数字扩展到整个整数域，因此在图7.1中我们直接将其定义到整数域。

$$
\begin{array}{lll}
\textit{formula} & ::= & \textit{formula} \wedge \textit{formula} \mid \textit{formula} \vee \textit{formula} \mid \neg \textit{formula} \mid \\
 & & \textit{term relop term} \mid \exists \textit{var}.\textit{formula} \mid \forall \textit{var}.\textit{formula} \\
\textit{term} & ::= & \textit{numeral} \mid \textit{term} + \textit{term} \mid -\textit{term} \mid \textit{numeral} * \textit{term} \mid \textit{var} \\
\textit{relop} & ::= & < \mid \leq \mid = \mid \neq \mid \geq \mid > \\
\textit{var} & ::= & x \mid y \mid z \ldots \\
\textit{numeral} & ::= & \cdots -2 \mid -1 \mid 0 \mid 1 \mid 2 \cdots
\end{array}
$$

图7.1　整数线性判定问题的一阶逻辑公式定义

当前常用的定理证明器主要有Simplify[72]、ICS[110]、CVC Lite[111]、VERIFUN[112]、Isabelle[113]、ACL2[114]、HOL[115]、Nuprl[116]、PVS[117]、Zapato[109]等，它们都能对上述某些或全部整数线性公式进行不同程度的判定，但在面向源代码的形式验证中将它们用作整数线性公式的判定过程存在诸多的不便之处。传统的定理证明工具如Isabelle、HOL、Nuprl、PVS等属高阶定理证明器，对一阶逻辑公式形式的整数线性判定问题而言，它们显得过于笨重，而且由于未对整数线性判定问题进行专门的考虑和优化，导致判定效率往往较低。另外，它们要求判

定问题描述为特定的高阶逻辑形式，并可能要求用户参与交互，这对轻量级、全自动的面向源代码的形式验证而言显然是不合适的。现代的轻量级、全自动的定理证明工具如 Simplify、ICS、CVC Lite、VERIFUN、Zapato、ACL2 等应用于判定整数线性判定问题时也存在各自的不足：有些定理证明工具如 ICS、CVC Lite、Zapato 等不支持显式含量词的一阶逻辑公式；有些定理证明工具如 Simplify、VERIFUN、ACL2 等（也包括 Isabelle、Nuprl、PVS 等）不能保证对整数线性判定问题判定的完备性。因此，有必要专门针对整数线性判定问题设计轻量级的、可靠的（Sound）和完备的（Complete）判定过程，以提高验证过程的效率和精度。

本章提出了一种优化的轻量级判定方法，该方法能够判定图 7.1 所示的任意整数线性一阶逻辑公式，其基本思想与 Omega 测试[118]类似，即逐步地消除公式中存在量化的变元，直至公式可判定为止。由于 Omega 测试只支持无量词的合取公式（事实上，Omega 测试假定合取公式中的所有变元都在公式的最外层存在量化，即认为 $P(x) \equiv \exists x.P(x)$），且不能消除公式中的指定变元，因此我们借鉴 Cooper 的方法[119]，通过引入整除关系来消除公式中的指定变元。文献[120]提出了一种与本章相似的完备的整数线性公式判定方法，并将其在定理证明工具 HOL 中进行了实现。与该文相比，本章提出的方法能够高效地判定包含形如 $x = y$ 的等式的公式，而该类型的判定公式在基于切片执行的 C 源代码验证工具中会大量出现。

为了更好地支持面向 C 源代码的验证，本章在整数线性一阶逻辑公式的基础上，增加了对 C 程序中常用的整数除法和取余数运算、以及位运算的支持，定义了 C 程序判定公式，并相应地扩充了判定方法。对整数除法和取余运算，其基本思想是将它们消除，并转换为等价的等式或不等式；而对位运算，我们则将其转换为比特变元之间的运算，并逐步消除比特变元，直至公式可判定为止。实验证明，扩充后的整数线性判定过程能够判定基于切片执行的 C 源代码验证工具产生的绝大部分判定问题。

本章提出的判定方法在切片执行工具中得到了实现，通过对 SSL 协议的验证实验，我们比较了两种策略，即使用定理证明工具 Simplify 作为判定过程以及使用本章提出的判定方法作为判定过程，的验证效率。实验所使用的源程序来自于 SSL 协议在 Linux 操作系统中的实现程序 *openssl-0.9.6c*，通过对比我们发现，使用本章提出的判定过程，其验证效率比定理证明工具 Simplify 平均提高了 10.5 倍。

7.1 完备的整数线性公式判定方法

对于形如图 7.1 所示的整数线性一阶逻辑公式，我们需要对其进行一系列的等价变换。首先，我们按如下方法将整数线性一阶逻辑公式中的全称量词（∀）用存在量词（∃）表示：

$$\forall x.P(x) \equiv \neg\exists x.\neg P(x) \tag{7.1}$$

接下来，我们按下列方法将存在量词∃尽可能地内移：

$$\exists x.(P(x) \vee Q(x)) \equiv \exists x.P(x) \vee \exists x.Q(x) \tag{7.2}$$

然后，我们将公式进行改写，使得公式中只包含 $x=y$ 形式的等式和 $x \le y$ 形式的不等式。其它四种关系运算可基于下列公式进行改写：

$$x<y \equiv x \le y-1,\ x \ge y \equiv y \le x,\ x>y \equiv y \le x-1,\ x \ne y \equiv x \le y-1 \vee y \le x-1 \tag{7.3}$$

最后，将公式变换为析取范式型（DNF）。经过这些等价变换，我们可以认为公式中的所有变元都是存在量化的（对于自由变元，如 $\exists x.x>y$ 中的变元 y，我们认为它是全称量化的，即 $\exists x.x>y \equiv \forall y \exists x.x>y$，再将其按公式(7.1)改写为存在量化的即可），且其最内层的存在量词所作用的公式必为如下形式：

$$\exists x.\left(\left(\bigwedge_i u_i \le a_i x\right) \wedge \left(\bigwedge_j b_j x \le v_j\right) \wedge \left(\bigwedge_k c_k x = w_k\right)\right) \tag{7.4}$$

其中 a_i、b_j、c_k 为变元 x 的系数，u_i、v_j、w_k 为不包含 x 的整数线性表达式。不失一般性，我们假设 $a_i>0$、$b_j>0$、$c_k>0$。我们的目标是将公式(7.4)变换为与之等价，但不包含变元 x 的整数线性公式，换言之，即消除存在量词∃x。

如果表达式中包含等式，我们就先消除这些等式。任选一个等式 $c_m x = w_m$，对任意不等式 $u_i \le a_i x$，将其系数乘以 c_m 后，再将等式的系数乘以 a_i 并代入到不等式中，得到 $c_m u_i \le a_i w_m$。$b_j x \le v_j$ 及 $c_k x = w_k$ 类型的公式采用相同的方法进行变换，最终得到公式(7.5)，其中整除关系 $c_m | w_m$ 表示 c_m 能够整除 w_m。

$$\left(\bigwedge_i c_m u_i \le a_i w_m\right) \wedge \left(\bigwedge_j b_j w_m \le c_m v_j\right) \wedge \left(\bigwedge_{k \ne m} c_k w_m = c_m w_k\right) \wedge \left(c_m | w_m\right) \tag{7.5}$$

定理 7.1 公式(7.4)与公式(7.5)等价。

证明：对于整数线性表达式 u_i、v_j、w_k 中包含的所有变元的任意取值，如果公式(7.4)满足，则显然公式(7.5)满足。如果公式(7.5)满足，由于 $c_m | w_m$，因此存在 x，使得 $c_m x = w_m$，将其代入公式(7.5)马上可以得到公式(7.4)。 ■

如果公式(7.5)中 $c_m=1$ 或者 w_m 中不包含变元，则我们可以很容易地计算出 $c_m | w_m$ 的真值，否则需要消除公式(7.5)中引入的整除关系表达式 $c_m | w_m$。为书写方便，我们将公式(7.4)表示简记为 P，其外层的存在量词公式经范式化为 DNF 后可能有两种形式，即 $\exists y.(Q \wedge P)$ 和 $\exists y.(Q \wedge \neg P)$，其中 Q 为包含 y 的整数线性公式。不失一般性，我们假设 w_m 中包含变元 y（否则可以将 $c_m | w_m$ 外提到量词∃y 之外），令 $w_m = dy+e$，其中 d 为常数，e 为可能含有其它变元的整型表达式。

对于形如 $\exists y.(Q \wedge P)$ 的公式，$c_m | w_m \equiv c_m | dy+e \equiv \exists z.c_m z = dy+e$，不失一般性，我们假设式中 $0<d<c_m$（否则取 $d = d \bmod c_m$）。由于 z 是新引入的变元，因此我们

可以将量词$\exists z$外提到量词$\exists y$之外（如前所述，我们假设所有变元都是存在量化的），并针对变元 y 消除等式$c_m z = dy + e$。消除该等式后，必然会出现另一个整除关系表达式$d \mid c_m z - e$，于是我们再针对变元 z 消除该整除关系。由于$d < c_m$，因此该过程不断进行下去后，必然出现$d = 1$的情况，从而最终可以消除该整除关系表达式。

对于形如$\exists y.(Q \wedge \neg P)$的公式，将公式 P 前的$\neg$分配到 P 的每个合取项后会出现关系式$\neg(c_m \mid w_m)$，我们根据下列等价关系式将其变换为多个整除关系式后利用上述方法消除。

$$\neg(c_m \mid w_m) \equiv \bigvee_{1 \le l \le c_m - 1} (c_m \mid w_m + l) \tag{7.6}$$

如果公式(7.4)中没有等式项，即形如公式(7.7)，定理 2[118, 120]给出其等价的表示形式。

$$\exists x.\left(\left(\bigwedge_i u_i \le a_i x\right) \wedge \left(\bigwedge_j b_j x \le v_j\right)\right) \tag{7.7}$$

定理 7.2 (Pugh, 1992) 设$L(x)$是 x 的下界约束$u_i \le a_i x$的合取，$U(x)$是 x 的上界约束$b_j x \le v_j$的合取，令 m 为b_j中的最大者，即$m = \max(b_1, \cdots, b_j, \cdots)$，那么：

$$\begin{aligned} \exists x.L(x) \wedge U(x) \equiv \quad & \left(\wedge_{i,j} (a_i - 1)(b_j - 1) \le a_i v_j - b_j u_i\right) \vee \\ & \vee_i \vee_{k=0}^{\left\lfloor \frac{m a_i - a_i - m}{m} \right\rfloor} \exists x.\left(a_i x = u_i + k \wedge L(x) \wedge U(x)\right) \end{aligned} \tag{7.8}$$

该定理的证明请参见文献[120]。需要说明的是，公式(7.8)中的等式$a_i x = u_i + k$可以使用上文介绍的等式消除方法消除。综合使用上述等式和不等式的处理方法，可以逐个消除整数线性公式中的变元，从而最终得到判定结果。

值得注意的是，如果公式(7.7)中的所有a_i或者所有b_j都等于 1，那么定理(7.8)可以简化为$\exists x.L(x) \wedge U(x) \equiv \wedge_{i,j} b_j u_i \le a_i v_j$，此时的计算复杂度将大幅下降。即使并非所有$a_i$或者所有$b_j$都等于 1，也可以应用公式(7.9)和公式(7.10)先对公式作初步判断，这两个公式可直接由公式(7.8)导出[118]。若公式(7.9)中$\wedge_{i,j} b_j u_i \le a_i v_j = False$，那么$\exists x.L(x) \wedge U(x) = False$；同理，若公式(7.10)满足$\wedge_{i,j} (a_i - 1)(b_j - 1) \le a_i v_j - b_j u_i = True$，那么$\exists x.L(x) \wedge U(x) = True$。Pugh[118]、Norrish[120]和本章的实验表明，实际应用中的绝大部分判定问题都可以由这两种简化的判定方法解决。

$$\exists x.L(x) \wedge U(x) \Rightarrow \bigwedge_{i,j} b_j u_i \le a_i v_j \tag{7.9}$$

$$\bigwedge_{i,j} (a_i - 1)(b_j - 1) \le a_i v_j - b_j u_i \Rightarrow \exists x.L(x) \wedge U(x) \tag{7.10}$$

与 Omega 测试一样，本章算法在最坏情况下的复杂度是超指数的，即 $O\left(m^{2^n}\right)$，其中 m 为等式和不等式的数量，n 为变元的数量，但是实际应用中的绝大部分判定问题的规模都很小，且可以应用上述两种简化的判定方法进行判断，因此该方法是实用的，正如 Omega 测试在对程序的相关性分析中得到了大量的应用一样，而本章的实验也说明了这一点。

7.2 面向 C 源代码验证的扩充判定方法

为了对 C 程序进行验证，我们需要支持 C 程序中的各种语法和运算符。对于 C 程序验证产生的判定公式中的指针、数组和结构形式的变元，我们通过指针处理和变元替换等方法可以将它们替换为满足图 7.1 所示变元定义的变元。为了支持复杂的 C 运算符，我们对图 7.1 所示的整数线性判定问题进行扩充，得到定义为图 7.2 所示的 C 程序判定公式。应该指出的是，图 7.2 定义的 C 程序判定公式并不能涵盖 C 程序验证过程中所能够产生的全部判定公式。

$$
\begin{array}{lll}
formula & ::= & term_W\ relop_W\ term_W \mid term_B\ relop_B\ term_B \mid \neg formula \mid \\
 & & formula \wedge formula \mid formula \vee formula \mid \exists var.formula \mid \forall var.formula \\
term_W & ::= & numeral \mid var \mid term_W + term_W \mid -term_W \mid numeral * term_W \mid \\
 & & term_W / numeral \mid term_W \% numeral \\
term_B & ::= & numeral \mid var \mid term_B >> numeral \mid term_B << numeral \mid \\
 & & term_B \,\&\, term_B \mid term_B \mid term_B \mid \sim term_B \\
relop_W & ::= & < \mid \leq \mid = \mid \neq \mid \geq \mid > \\
relop_B & ::= & \neq \mid = \\
var & ::= & x \mid y \mid z \cdots \\
numeral & ::= & \cdots -2 \mid -1 \mid 0 \mid 1 \mid 2 \cdots
\end{array}
$$

图 7.2 扩充整数除法、取余和位运算的 C 程序判定公式

我们称图 7.2 中的 $term_W$ 为字项(Word Term)，称 $term_B$ 为位项(Bit Term)。字项中的每个变元都被看作一个整数，而位项中的每个变元则被视为一个比特序列（从而位项之间只支持等于和不等于两种关系运算）。除了扩充了位运算符和位项之间的关系符之外，我们还扩充了图 7.1 所示的整数项（$term$），增加了整数除法和取余数两种 C 程序常用的运算。要说明的是，我们并不能对图 7.2 定义的所有公式进行判定。对图 7.2 定义的任意判定公式 f，我们定义函数 $Vars_W(f)$ 返回 f 的所有字项（$term_W$）中包含的变元集合，而函数 $Vars_B(f)$ 则返回 f 的所有位项（$term_B$）中包含的变元集合，那么满足定义 7.1 的公式都是可判定的。

定义 7.1 (字/位变量独立的判定公式). 如果一阶逻辑公式 f 满足图 7.2 所示的定义，且 $Vars_W(f) \cap Vars_B(f) = \varnothing$，则我们定义 f 为字/位变量独立的判定公式。换言之，字/位变量独立的判定公式中的任意变元不能既出现在字项中表示一个整

数，又出现在位项中表示一个比特序列。

应该指出的是，大多数 C 程序，乃至大部分操作系统内核源代码和大多数驱动程序等系统级的 C 程序，在验证过程中产生的判定公式都满足定义 7.1，即可被本章的方法判定。

对于 C 语言判定公式中的取余数运算（%），我们基于等价变换 $x\%c \equiv x-c*(x/c)$ 将其变换为整数除法运算。对于公式中的整数除法运算（/），我们则基于定理 7.3 将其消除。

定理 7.3 设满足定义 7.1 且不含量词的整数线性公式 f 中包含整数除法运算 u/c，其中 c 为整数且 $c>0$，u 为图 7.2 定义的整数字项（$term_w$），则有公式(7.11)成立，其中 $f[u/c\to z]$ 表示将公式 f 中的所有字项 u/c 用变元 z 替换。

$$f \equiv \exists z.\left(f[u/c\to z]\wedge\left((u\geq 0\wedge cz\leq u<cz+c)\vee(u<0\wedge cz-c<u\leq cz)\right)\right) \quad (7.11)$$

证明：首先，我们有 $(u/c=z) \equiv (u\geq 0\wedge cz\leq u<cz+c)\vee(u<0\wedge cz-c<u\leq cz)$，分别考查 $u\geq 0$ 和 $u<0$ 的情况可以证明该式，然后我们只要证明 $f \equiv \exists z.\left(f[u/c\to z]\wedge(u/c=z)\right)$ 即可。若该式左端为真，则对使 f 为真的每个 u，都存在唯一的 $z=u/c$，将其代入该式右端知右端也为真；若右端为真，则对使右端为真的任意 u，等式 $u/c=z$ 成立，则将 $f[u/c\to z]$ 中的 z 用 u/c 替换后，知 f 也为真，故定理证毕。 ■

定理 7.3 的应用可以和本章第 1 节给出的存在量词消除过程结合进行，消除公式 $\exists x.f(x)$ 中存在量化的变元 x 时，根据第 1 节的讨论知 $f(x)$ 中不包含量词，因此可以基于定理 7.3 消除 $f(x)$ 中的所有形如 u/c 的整数除法运算。

对于位项和位运算，根据定义 7.1，位项中的变元集合与字项中的变元集合不相交，从而消除位项中的变元不影响任何字项。因此若欲消除公式 $\exists x.f$ 中的变元 x（其中 f 为不含量词的析取范式型 DNF），不失一般性，我们可以假设公式 f 中只包含位项（否则可以将字项外移）。首先我们考查 f 中是否存在形如 $x=u$ 的等式，其中 u 为不含变元 x 的表达式，若存在，将其代入到其它包含变元 x 的位项中，即可消除变元 x。

若上述条件不满足，就将位项中的每个变元用 32 个比特变元来替换（假设机器的字长是 32 位）。例如：$z=x\&y \equiv \bigwedge_{0\leq i\leq 31}(z_i=x_i\&y_i)$；$z\neq x>>c \equiv \left(\bigvee_{c\leq i\leq 31} z_{i-c}\neq x_i\right)\vee\left(\bigvee_{0\leq i<c} z_{31-c}\neq 0\right)$。例中的移位操作假设用 0 来填充，当然也可以用其它方式。由于比特变元只能取值为 0 和 1，因此有 $x_i\neq y_i \equiv x_i=\sim y_i$ 成立。经过这些变换，移位运算符<<和>>以及不等号≠可以被消除，从而我们假设公式中只包含与、或、非三种位运算符，且位项之间以等号=连接。设经过上述变

换将公式 f 变换为 f_B，则 $\exists x.f \equiv \exists x_{31}\cdots x_0.f_B$，进一步地，按第 1 节给出的方法，将其变换为析取范式型 DNF（可能需要取非和/或将公式拆解为多个子公式的并）。

对变换后的每个 DNF 子公式 $\exists x_{31}\cdots x_0.f_B^k$，如果 f_B^k 中存在形如 $x_i = u_i$ 的等式，其中 u_i 为不含 x_i 的位项，则直接将其代入其它包含比特变元 x_i 的位项中即可消除 x_i。值得注意的是，如果与、或运算的两个操作数中只包含一个比特变元，即形如 $x_i \& c_i$、$x_i | c_i$，其中 c_i 为常数 0 或 1，则其值可以直接计算出，从而能够代入并消除 x_i。例如，$(x_i \& 1) | 0 = y_i \equiv x_i = y_i$。在很多 C 程序中，位运算都只涉及一个变元，例如 $(x \& 5) >> 3$ 或判定 $x \& 0x0010 = 0$ 等，这些情况下都可以通过直接代入而消除所有比特变元。

若不满足上述情况，则我们基于公式 $\exists x_i.f_B^k \equiv f_B^k[x_i \to 0] \vee f_B^k[x_i \to 1]$ 消除存在量化的比特变元 x_i，其中 $f_B^k[x_i \to c_i]$ 表示将公式 f_B^k 中的所有 x_i 用 c_i 替换。在这种最坏情况下，消除一个位项中的变元将可能产生多达 2^{32} 个析取项（假设机器字长 32 位）。因此，与第 1 节介绍的消除字项中的变元一样，上述消除过程的最坏复杂度是指数级的。但同样的，它是实用的，这是由于：①.大多数 C 程序判定公式位项中的变元通过前面介绍的变元代入方法都可以完全消除，而不需要最坏情况下的展开；②.在大多数 C 程序判定公式中，位运算表达式的数量和位项中的变元都很少，且相当简单，即使需要展开，其代价也很小；③.在展开过程中，会大量出现 $f_B^k[x_i \to 0]$ 或 $f_B^k[x_i \to 1]$ 等于 0 或 1，或等于某个其它的比特变元 x_j 的情况。从而在大多数情况下都只需要展开少量的几个比特变元，就可以通过代入法完全消除其它比特变元。例如消除公式 $\exists x_1 (x_1 \& x_3 \& x_4 = (\sim x_1 \& x_5) | x_2)$ 中的比特变元 x_1 时，令 $x_1 = 1$ 得到 $x_3 \& x_4 = x_2$；令 $x_1 = 0$ 得到 $0 = x_5 | x_2$，进而 $x_2 = 0 \wedge x_5 = 0$，从而无需对变元 x_2 和 x_5 进行展开。

7.3 对切片执行产生的一阶逻辑验证公式的判定

切片执行的部分最强后置条件表示为一个二元偶 $\langle \Omega, \Phi \rangle$，其中 Ω 是一个部分函数，将每个变量映射到一个由 Skolem 常量构成的表达式；Φ 是一个集合，存储了由 assume 语句引入的所有布尔表达式（同样由 Skolem 常量组成）。

在对 C 程序切片执行之前，我们先将 C 程序中包含多个复合条件的 *if* 语句进行如下变换：

$$if(a \,\&\&\, b)\ x\ else\ y; \quad \to \quad if(a)\ \{\ if(b)\ x\ else\ y\}\ else\ y;$$

$$if(a \,\|\, b)\ x\ else\ y; \quad \to \quad if(a)\ x\ else\ if(b)\ x\ else\ y;$$

从而所有的 assume 语句都只包含单个条件，进而可以保证集合 Φ 中的每个公式都是由 Skolem 常量为变元构成的等式和不等式的合取（$\wedge$）。

切片执行过程主要在两种情况下调用定理证明工具：第一种情况是在判定一

个 assume 语句是否可行（即判定分支语句的分支是否可行）时；第二种情况是在判定一个部分最强后置条件是否蕴含当前的切片执行上下文时。

设切片执行完 assume 语句 $assume(c)$ 后得到的部分最强后置条件为 $\langle\Omega,\Phi\rangle$，为了判定该 assume 语句是否可行，我们需要判定公式 $\left(\bigwedge_{\phi\in\Phi}\phi\right)\neq False$ 是否成立，该公式是一个命题逻辑公式，可直接判定。

为了判定一个部分最强后置条件是否蕴含当前的切片执行上下文，我们需要将部分最强后置条件 $\langle\Omega,\Phi\rangle$ 通过下列函数 h_1 转换为一阶逻辑公式（参见第二章）：

$$h_1\left(\langle\Omega,\Phi\rangle\right)\triangleq\exists\theta_1\cdots\theta_m:\left(\left(\bigwedge_{\omega\in\Omega}e2b(\omega)\right)\wedge\left(\bigwedge_{\phi\in\Phi}\phi\right)\right)\tag{7.12}$$

其中 $\theta_1,\cdots,\theta_m$ 为在计算过程中引入的 Skolem 常量，函数 $e2b(\omega)$ 将 Ω 中对每个变量的赋值转化为一个等式，例如 $e2b(x\to 1)\triangleq(x=1)$。

在集成了部分最弱前置条件的切片执行过程中，部分最弱前置条件用二元偶 $\langle\sigma,\omega\rangle$ 表示，其中 σ 为命名映射，它将程序变量映射到其当前的名字；ω 为一阶逻辑公式，描述了当前的部分最弱前置条件。我们同样需要将部分最弱前置条件 $\langle\sigma,\omega\rangle$ 通过下列函数 h_2 转换为一阶逻辑公式（参见第四章）：

$$h_2\left(\langle\sigma,\omega\rangle\right)\ \triangleq\ \forall\theta_1,\cdots,\theta_m:\left(\left(\bigwedge_{x\in dom(\sigma)}x=\sigma(x)\right)\Rightarrow\omega\right)\tag{7.13}$$

其中 $\theta_1,\cdots,\theta_m$ 为在计算过程中引入的所有变量名。

令某个（并发）程序位置 s 处的切片执行上下文为集合 $\{\alpha_1,\cdots,\alpha_n\}$，其中 α_i 或者为部分最强后置条件 $\langle\Omega_i,\Phi_i\rangle$，或者为部分最弱前置条件 $\langle\sigma_i,\omega_i\rangle$。因此，我们令 $\{\alpha_1,\cdots,\alpha_n\}=\{\langle\Omega_1,\Phi_1\rangle,\cdots,\langle\Omega_{n1},\Phi_{n1}\rangle,\langle\sigma_1,\omega_1\rangle,\cdots,\langle\sigma_{n2},\omega_{n2}\rangle\}$，其中 $n1+n2=n$。设某条执行路径或迁移序列到达 s 时的部分最强后置条件为 $\langle\Omega,\Phi\rangle$，则切片执行需要判定公式(7.14)是否成立，以判断部分最强后置条件 $\langle\Omega,\Phi\rangle$ 是否需要被考虑，其中函数 h_1 和 h_2 分别定义为公式(7.12)和公式(7.13)所示形式。

$$h_1\left(\langle\Omega,\Phi\rangle\right)\Rightarrow h_1\left(\langle\Omega_1,\Phi_1\rangle\right)\vee\cdots\vee h_1\left(\langle\Omega_{n1},\Phi_{n1}\rangle\right)\vee h_2\left(\langle\sigma_1,\omega_1\rangle\right)\vee\cdots\vee h_2\left(\langle\sigma_{n2},\omega_{n2}\rangle\right)\tag{7.14}$$

如果公式(7.14)中的所有等式和不等式都是整数线性的（这种情况在实用程序中大量存在），那么我们就可以使用本章讨论的判定方法进行判定。

部分最强后置条件 $\langle\Omega,\Phi\rangle$ 的部分函数 Ω 将每个程序变量映射到一个以 Skolem 常量为变元的表达式，令 Ω 中涉及的所有程序变量的集合为 $dom(\Omega)$（即函数 Ω 的定义域），首先我们通过引入必要的 Skolem 常量，使得对所有 $i:1\le i\le n1$ 和所有 $j:1\le j\le n2$，都有 $dom(\Omega)=dom(\Omega_i)=dom(\sigma_j)$。如果某个程序变量 $x\in dom(\Omega)$ 但 $x\notin dom(\Omega_i)$，我们就令 $\Omega_i(x)=\theta_x$，其中 θ_x 为新引入的 Skolem 常量，对 σ_i 的做法是类似的。这样，公式(7.14)可具体表示为公式(7.15)所示的形式，其中 $\theta_1\cdots\theta_m$ 与 $\theta_1^i\cdots\theta_{m_i}^i$ 分别为部分最强后置条件 $\langle\Omega,\Phi\rangle$ 与 $\langle\Omega_i,\Phi_i\rangle$ 中包含的所有 Skolem 常量，而

$\theta_1^j\cdots\theta_{m_j}^j$ 则为部分最弱前置条件 $\langle\sigma_j,\omega_j\rangle$ 中为变量重命名所引入的所有临时名字。

$$\left(\exists\theta_1\cdots\theta_m:\left(\left(\bigwedge_{x\in dom(\Omega)}x=\Omega(x)\right)\wedge\left(\bigwedge_{\phi\in\Phi}\phi\right)\right)\right)\Rightarrow \bigvee_{1\le i\le n1}\left(\exists\theta_1^i\cdots\theta_{m_i}^i:\left(\left(\bigwedge_{x\in dom(\Omega_i)}x=\Omega_i(x)\right)\wedge\left(\bigwedge_{\phi\in\Phi_i}\phi\right)\right)\right)\vee \bigvee_{1\le j\le n2}\left(\forall\theta_1^j\cdots\theta_{m_j}^j:\left(\left(\bigwedge_{x\in dom(\sigma_j)}x=\sigma_j(x)\right)\Rightarrow\omega_j\right)\right) \tag{7.15}$$

我们可将公式(7.15)简化为与之等价的公式(7.16)，其正确性由定理 7.3 保证。

$$\left(\bigwedge_{\phi\in\Phi}\phi\right)\Rightarrow\left(\begin{array}{l}\bigvee_{1\le i\le n1}\left(\exists\theta_1^i\cdots\theta_{m_i}^i:\left(\bigwedge_{x\in dom(\Omega)}\Omega(x)=\Omega_i(x)\right)\wedge\left(\bigwedge_{\phi\in\Phi_i}\phi\right)\right)\vee\\ \bigvee_{1\le j\le n2}\left(\forall\theta_1^j\cdots\theta_{m_j}^j:\left(\left(\bigwedge_{x\in dom(\Omega)}\Omega(x)=\sigma_j(x)\right)\Rightarrow\omega_j\right)\right)\end{array}\right) \tag{7.16}$$

定理 7.3 公式(7.15)与公式(7.16)等价。

证明：先假设公式(7.15)为真，来证明公式(7.16)为真。对变元 $\theta_1\cdots\theta_m$ 的任意一组取值，如果公式(7.16)左边表达式 $\bigwedge_{\phi\in\Phi}\phi$ 为假，则公式(7.16)为真；如果公式(7.16)左边表达式为真，则对每个程序变量 $x\in dom(\Omega)$ 的取值 $x=\Omega(x)$，公式(7.15)蕴含式左端 $\exists\theta_1\cdots\theta_m:\left(\left(\bigwedge_{x\in dom(\Omega)}x=\Omega(x)\right)\wedge\left(\bigwedge_{\phi\in\Phi}\phi\right)\right)$ 为真。根据假设，公式(7.15)对所有程序变量 $dom(\Omega)$ 的任意取值都为真，因此公式(7.15)蕴含式右端为真，将其中的 x 用 $\Omega(x)$ 替换后得出公式(7.16)中蕴含式的右端为真，从而公式(7.16)也为真，因此公式(7.15)⇒公式(7.16)。

再假设公式(7.16)为真，来证明公式(7.15)为真。对程序变量 $dom(\Omega)$ 的任意一组取值，我们同样只考虑公式(7.15)的前件 $\exists\theta_1\cdots\theta_m:\left(\left(\bigwedge_{x\in dom(\Omega)}x=\Omega(x)\right)\wedge\left(\bigwedge_{\phi\in\Phi}\phi\right)\right)$ 为真的情况，此时存在一组 $\theta_1\cdots\theta_m$，使得 $\bigwedge_{\phi\in\Phi}\phi$ 为真且每个 $x\in dom(\Omega)$ 的取值满足 $x=\Omega(x)$。根据假设，公式(7.16)为真和 $\bigwedge_{\phi\in\Phi}\phi$ 为真可以得出公式(7.16)的右端也为真，将其中的 $\Omega(x)$ 替换为 x 后得出公式(7.15)为真，因此公式(7.16)⇒公式(7.15)。故定理证毕。 ■

下面我们讨论如何消除公式(7.16)的存在量词和全称量词。对蕴含式(7.16)右端的第一个公式，我们消除公式 $\exists\theta_1^i\cdots\theta_{m_i}^i.\left(\wedge_{x\in dom(\Omega)}\Omega(x)=\Omega_i(x)\wedge\wedge_{\phi\in\Phi_i}\phi\right)$ 的存在量词 $\exists\theta_1^i\cdots\theta_{m_i}^i$，得到等价的公式 $\phi_1^i\wedge\cdots\wedge\phi_{l_i}^i\wedge c_1^i\left|w_1^i\wedge\cdots\wedge c_k^i\right|w_k^i$，其中 $\phi_1^i\wedge\cdots\wedge\phi_{l_i}^i$ 是消除了 Φ_i 中所有等式和不等式中的变元 $\theta_1^i\cdots\theta_{m_i}^i$ 后得到的新的等式和不等式的合取（∧），$c_1^i,\cdots,c_k^i$ 为正整数且不等于 1，$w_1^i,\cdots,w_k^i$ 为变元 $\theta_1,\cdots\theta_m$ 的表达式。实践中我们发现，在很多判定问题中，消除存在量词 $\exists\theta_1^i\cdots\theta_{m_i}^i$ 后得到的等价公式中根本不含整除关系表达式，这是由于等式 $\Omega(x)=\Omega_i(x)$ 中，$\theta_1^i\cdots\theta_{m_i}^i$ 的系数都是 1，或者产生的整除关系表达式在后续的等式消除过程中被消除。同时 Φ_i 中公式的数量

往往会减少，这是由于Φ_i中的公式只包括变元$\theta_1^i \cdots \theta_{m_i}^i$，而在消除这些变元时，将产生大量可合并的冗余公式。

为了消除蕴含式(7.16)右端的第二个公式中包含的全称量词$\forall \theta_1^j \cdots \theta_{m_j}^j$，我们先将公式$\forall \theta_1^j \cdots \theta_{m_j}^j : \left(\left(\wedge_{x \in dom(\Omega)} \Omega(x) = \sigma_j(x)\right) \Rightarrow \omega_j\right)$变换为只包含存在量词的等价公式$\neg \exists \theta_1^j \cdots \theta_{m_j}^j : \left(\left(\wedge_{x \in dom(\Omega)} \Omega(x) = \sigma_j(x)\right) \wedge \neg \omega_j\right)$，于是我们只需要考虑其子公式$\neg \omega_j$即可。根据第四章部分最弱前置条件的实现，$\omega_j$中包含两种逻辑运算，即$\omega_1 \wedge \omega_2$和$c \Rightarrow \omega_1$，对应的否定形式为$\neg \omega_1 \vee \neg \omega_2$和$c \wedge \neg \omega_1$。通过这种变换规则可以将逻辑非（$\neg$）逐层内置，从而最终得到只包含与、或、非逻辑连接词的公式，再将其转化为析取范式型（DNF）即可。

将公式(7.16)转换为析取标准型时，公式的数量将呈指数增加，因此其最坏情况下的复杂度很高（这也是 Omega 测试理论上复杂度很高的主要原因）。但实际应用中我们发现，公式(7.16)蕴含式右边的第一个公式中最内层析取公式的析取项的数量往往非常少，几乎所有整除关系都可以被消除，集合Φ的公式在消除变元$\theta_1^i \cdots \theta_{m_i}^i$后也会产生大量可被消除的冗余公式(即被其它公式所蕴含)；而公式(7.16)蕴含式右边的第二个公式则由于大量无关分支语句可以被合并，从而最终ω_j中的合取项并不多，因此实际应用的效率很高，下一节的实验结果也可以充分说明这一点。

另外，对得到的析取标准型公式，先消除等式后，可以得到形如公式(7.7)的只含$\leq$形式的不等式，然后我们充分利用简化公式(7.9)和(7.10)进行判定。

7.4 验证工具和实验结果

我们在 MAGIC[48]的基础上实现了基于切片执行的面向 C 源代码的验证工具，工具最初使用定理证明工具 Simplify[72]作为判定过程，基于 Das 的流不敏感的指向图算法[73]来处理指针和变量别名。验证工具支持对函数指针、变量别名、结构和数组的处理，暂不支持递归函数调用和非直接跳转如 setjump/longjump 等，但这些限制在实验程序中并不存在。

所有的测试用例都取自于 *openssl-0.9.6c* 程序源代码，它共有数千行 C 代码，实现了用于在 Internet 上进行安全信息传输的 SSL 协议。与 BLAST、MAGIC 一样，我们的验证目标是 SSL 协议的初始握手协议，所验证的性质是某些非法的状态转换序列不会在握手过程中出现。我们共验证了 4 个客户端性质和 16 个服务器端性质，分别使用 *ssl-clnt* 和 *ssl-srvr* 来标识它们，与其它验证工具的结果一样，所有性质都被验证为被程序满足。

通过实验我们发现，验证过程中提交给定理证明工具 Simplify 的所有判定问题都是我们可判定的 C 程序判定问题，由于判定问题的数量非常大，Simplify 成为

了验证工具的瓶颈。因此我们在切片执行工具中实现了本章提出的判定方法，对一个判定请求，先检查我们是否可判定，如果不是就提交给 Simplify 进行判定，否则就使用本章介绍的算法进行判定。我们对只使用 Simplify 和结合使用本章判定方法这两种策略做了对比实验，结果如表 7.1 所示。

我们的实验平台是 1.6GHz AMD Athlon XP CPU 和 224MB 内存，软件环境是 Windows 2000 和 CygWin 2.427。表 7.1 中“Properties”为待验证的性质；“Time”为验证所用的时间，单位是秒；“DP Calls”是判定过程（Decision Procedure）的调用次数；“Time_1”是使用 Simpilfy 作为判定过程的切片执行所用的验证时间，单位是秒；“Time_2”是使用本章方法作为判定过程的切片执行所用的验证时间，单位是秒；“Gain”是 Time_1/Time_2 的比值，表明效率提高的倍数。表 7.1 中也给出了 BLAST 和 MAGIC 的验证结果，其数据来自文献[74]，其实验平台是 1.6GHz AMD Athlon XP CPU 和 900MB 内存，软件环境是 Linux，表中“*”表示验证时间超过 3 小时。

表 7.1 轻量级判定过程与定理证明工具 Simplify 的实验结果对比

Properties	BLAST	MAGIC	Slicing Execution			
	Time	Time	DP Calls	Time_1	Time_2	Gain
ssl-clnt-1	348	156	32518	28	3.9	7.2
ssl-clnt-2	523	185	8632	10	1.2	8.3
ssl-clnt-3	469	195	45327	39	5.7	6.8
ssl-clnt-4	380	191	37966	34	4.7	7.2
ssl-srvr-1	2398	226	111495	85	6.3	13.5
ssl-srvr-2	691	216	84791	66	4.5	14.7
ssl-srvr-3	1162	200	86018	67	4.6	14.6
ssl-srvr-4	284	170	87000	68	4.6	14.8
ssl-srvr-5	1804	205	211516	145	20.4	7.1
ssl-srvr-6	*	359	894003	698	101.4	6.9
ssl-srvr-7	359	196	221230	150	11.6	12.9
ssl-srvr-8	*	211	279124	207	20.1	10.3
ssl-srvr-9	337	316	213075	145	11.2	12.9
ssl-srvr-10	8289	241	169836	127	10.5	12.1
ssl-srvr-11	547	356	211821	145	11.2	12.9
ssl-srvr-12	2434	301	385410	280	27.3	10.3
ssl-srvr-13	608	436	210169	144	11.2	12.9
ssl-srvr-14	10444	406	531295	405	52.6	7.7
ssl-srvr-15	*	179	409549	305	33.4	9.1
ssl-srvr-16	*	356	700580	536	75.7	7.1
Average	1942.3	255.1	246567	184.2	20.1	10.5

从表 7.1 中的数据可以看出，使用本章介绍的判定方法作为判定过程后，其验证效率比使用定理证明工具 Simplify 作为判定过程平均提高了 10.5 倍，这充分说

明了本章提出的判定方法的高效性和实用性。

7.5 相关工作

本章提出的整数线性公式的完备判定方法借鉴了 Omega 测试[118]和定理证明工具 HOL 的整数线性判定过程[120]。Omega 测试考虑的公式形如 $\exists x_1 \cdots x_n . P_1 \wedge \cdots \wedge P_m$，其中每个 P_i（$1 \le i \le m$）都是一个整数线性等式或不等式，$x_1, \cdots, x_n$ 是公式 $P_1, \cdots, P_m$ 中包含的所有变元。Omega 测试通过一种改进的 Generalized GCD 测试方法首先消除 $P_1, \cdots, P_m$ 中的所有等式，再使用与本章相同的方法来处理不等式。相比 Omega 测试的公式，本章所考虑的公式要复杂得多：一是公式中可以有任意的逻辑连接词，而不是只包含与（$\wedge$）；二是公式中可以有任意量词的交迭和嵌套，包括存在量词（$\exists$）和全称量词（$\forall$）。在判定方法上，二者最大的不同在于对等式的处理，由于本章的方法必须消除指定的变元，因此 Omega 测试中等式的消除方法不能适用。我们借鉴了 Cooper 的方法[119]，通过引入整除关系来有效地消除公式中的等式，实践证明，该方法简单实用、效率很高。

本章提出的方法借鉴了文献[120]中论述的定理证明工具 HOL 使用的整数线性判定方法，但该文献中的方法所考虑的公式中不包含等式，即它只考虑了形如公式(7.7)的整数线性判定问题。本章的方法考虑了更一般的整数线性判定问题，即形如公式(7.4)的带等式的判定问题，这是由于我们要将其应用于基于切片执行的面向源代码的验证工具中，而切片执行的判定问题中包含了大量的等式。如果将等式转换为等价的不等式，即 $x = y \equiv x \le y \wedge y \le x$，我们发现其判定效率将大大降低，原因是转化后的公式必须考虑所谓的“Splinter”[118, 120]，即必须考虑公式(7.8)右端的第二个析取项，该析取项包含了大量公式的析取（$\vee$），这意味着消除一个变元会引入大量的等式和不等式。

整数线性判定问题可以转化为 Presburger 算术并求解，如 Cooper 的方法[119]。但根据文献[120]描述的实验结果，我们发现基于 Omega 测试的方法要比基于 Cooper 的方法稍快一些，但也并不是每个判定问题前者都比后者快，因此 HOL 定理证明工具将两种方法都作了实现，实现和对比这两种方法是我们未来的研究工作之一。

其它的定理证明工具，包括 Simplify[72]、ICS[110]、CVC Lite[111]、VERIFUN[112]、Isabelle[113]、ACL2[114]、Nuprl[116]、PVS[117]、Zapato[109]等，也可以作为判定过程用于面向源代码的形式验证。Simplify 基于一个名为 Simplex[72]的算法对实数和整数线性判定问题进行判定，该算法对整数线性判定问题而言是不完备的。Isabelle、VERIFUN 和 ACL2 利用简化公式(7.9)和(7.10)对整数线性判定问题进行判定，因此也是不完备的。ICS 和 CVC Lite 面向不显含量词的整数线性判定公式，用 Omega

测试的方法进行判定，可以保证判定的完备性。为了缓解 Omega 测试过程中 DNF 标准化时的公式数量爆炸问题，它们将公式转化为布尔表达式，并引入 SAT 工具（如常用的 Chaff[121]、GRASP[122]等）辅助判定。Nuprl 采用了 Shostak 提出的名为 SUP-INF 的判定方法[123]，PVS 则采用了 Shostak 提出的 loop-residue 算法[124]，它们都是不完备的。Zapato 基于 Harvey 和 Stuckey 提出的对二变量问题（Unit Two-Variable Per Inequality，UTVPI）的完备判定算法[125]进行整数线性判定，这是因为 SLAM[41, 42]的大多数判定问题都可以转化为二变量问题。对于这些定理证明工具所能判定的整数线性判定问题，本章提出的判定方法都能高效地进行判定，而且还能够保证判定的完备性。借鉴 ICS 和 CVC Lite 的思想，引入 SAT 工具加速判定过程是我们未来的研究方向。

由于整数在计算机中存储为固定字长（如 32 位）的二进制 0-1 串，因此另一种判定思路是将整数公式转化为逻辑电路，再在比特级判定逻辑电路的可满足性，如最新的 CVC Lite 和 ICS 等。这种思路的最大优点是可以支持任意类型的整数公式，同时也支持包括位级运算在内的任意运算符，但其最大的缺点是转换后的电路非常复杂，不能判定较大规模的判定公式。最新的研究，如[126-128]等，都试图使用部分字级（Word-Level）判定来简化位级（Bit-Wise）判定的复杂度，取得了良好的效率提升。与这些工作的研究动机不同的是，本章仅针对判定问题的一个子集进行研究，但是提出的判定方法能够可靠、完备和非常高效地判定该子集中的所有判定问题。而另一方面，绝大多数实用程序验证过程中产生的判定问题都可用本章的方法判定，因此本章的研究具有良好的实用价值。

7.6 小结

本章提出了一种完备的轻量级判定方法，用于对描述整数线性判定问题的一阶逻辑公式进行判定。该方法支持任意交迭的存在量词和全称量词、以及任意的逻辑连接词。为了更好地支持面向 C 源代码的形式验证，本章在整数线性判定问题的基础上扩充了整数除法、取余数和位运算，提出的扩充判定方法能够应用于基于 C、Java 等程序源代码的验证工具中，用于替代定理证明工具对整数线性判定问题进行高效的判定。该判定方法在基于切片执行的 C 程序验证工具中得到了应用，并与定理证明工具 Simplify 进行对比实验，实验结果充分说明了本章提出的判定方法的高效和实用性。

第八章 结束语

8.1 工作总结

本论文面向串行 C 程序和并发 C 程序的时序安全性质的验证，提出了切片执行的概念，并围绕切片执行进行了比较系统和深入的研究，提出了比较完整的技术体系。主要工作和创新点体现在如下方面：

1. 我们首先分析了程序验证的基本规律，并提出了变量抽象方法，变量抽象只考虑程序源代码中与待验证的时序安全性质相关的程序变量和语句。基于变量抽象，我们定义了部分最强后置条件，它是对传统最强后置条件的保守近似。基于这两个概念，我们提出了切片执行的概念，它是一种轻量级的符号执行方法。基于 *openssl-0.9.6c* 实用程序的实验数据表明，切片执行总体上看比谓词抽象的验证效率高。
2. 切片执行基于 CEGAR 框架不断地根据伪反例路径对模型进行精化，为了在不同精度的模型之间进行充分的信息复用，我们提出了面向时序安全性质的搜索复用框架，并将其应用于切片执行。实验结果表明，搜索复用框架能够较大程度地提高大部分程序和性质的验证效率。
3. 同样基于变量抽象，我们提出了部分最弱前置条件的概念，并将其应用到切片执行中。部分最弱前置条件是对传统最弱前置条件的保守近似，在切片执行过程中可以部分地取代部分最强后置条件，生成更弱的一阶逻辑公式来描述生成模型的状态，进而在不影响模型精度的前提下大大缩减生成模型的状态空间，实验证明了上述结论。
4. 我们将切片执行方法扩展到了对并发 C 程序的验证，为了进一步提高状态空间缩减的效果，我们提出了有状态动态偏序缩减方法，并将其自然地集成到切片执行框架中，用于指导切片执行，使其避免搜索多条具有相同偏序关系的并发进/线程交迭执行路径。实验结果证明，切片执行和有状态动态偏序缩减这两种正交的状态空间缩减方法的集成，大大缩减了所生成的并发程序模型的状态空间。
5. 我们基于开放源代码的 MAGIC 实现了切片执行工具，我们还在切片执行工具中实现了对 C 程序的指针和变量别名的支持，从而使得工具能够对实用的程序进行验证。此外，我们还定义了一类整数线性一阶逻辑判定公式，此类判定公式支持 C 程序中常用的整数线性运算，并扩充了对整数除法、取余和位运算的支持。实验表明，扩充的判定公式能够覆盖面向 C 程序的切片执行过程中所产生的绝大多数验证公式。在基于 openssl 实用程序的

切片执行过程中我们发现，采用该判定方法后，验证效率比使用定理证明工具 Simplify 提高了 10.5 倍。

8.2 研究展望

本文围绕提出的切片执行的概念，对 C 程序的验证进行较为系统的研究，取得了一定的成果。但是切片执行方法和工具也存在一些不足，且其所涉及的研究工作还有很多，以下我们列出一些我们正在进行或准备进行的研究工作：

- 对于 C 程序中的函数调用，切片执行工具目前采用的方法是与 BLAST、MAGIC 等工具一样，将被调用函数代码内嵌到函数调用点，从而最终只考虑单个函数的代码。这样做的缺点是程序的代码量将大量增长，而且不能处理递归函数调用。其解决方法是扩充切片执行，使其支持过程间代码分析和模型抽象，从而无需进行调用函数的代码内嵌；
- 切片执行工具当前基于 MAGIC 项目实现，只支持标准 C 语言程序，只能处理单个 C 程序文件，而且对指针和别名的处理还不完善。其解决方法是实现基于 GCC 等实用开源编译器项目的切片执行工具，并修改连接器，以及实现更为完善的指针和别名支持，使得切片执行工具能够验证更大规模的实用程序；
- 目前切片执行只支持共享内存方式的并发程序建模与验证，而很多并发程序采用消息传递的方式建模更方便、更有利于验证，如并发 SSL 程序在 MAGIC 项目中就是采用消息传递方式建模的。扩展对消息传递方式的并发程序的支持是切片执行在并发 C 程序验证方向的重要研究内容；
- 切片执行的另一个重要的工作就是将其应用到实际程序开发过程中，包括基于一个大型实用程序的案例分析（Case Study），这样一方面检验切片执行的效率，另一方面从实际应用中发现问题，以做进一步的研究和改进；
- 变量抽象准则中的变量数量极大地影响了切片执行的效率，因此在保证性质验证的前提下尽量减小抽象准则中的变量数量对提高切片执行的可扩展性是至关重要的。当前，切片执行在每次迭代之后都向抽象准则中增加必要的变量以得到精化的抽象准则，即抽象准则的变量集合是单调递增的。借鉴文献[129]和文献[74]的思想，通过考查多条伪反例路径，我们能够得到排除这些路径的最小的变量集合，从而尽量提高切片执行的效率；
- 此外，切片执行与谓词抽象的结合、基于 SAT 的高效一阶逻辑公式判定方法以及扩展切片执行使之支持时序安全性质以外的性质的验证也是未来的研究内容。

致 谢

首先，我要真诚地感谢我的导师杨学军教授。自从2000年本科毕业以来，我就一直拜在杨老师门下。杨老师是学院对学生最有责任感和指导次数最多、内容最细致的导师之一，能成为杨老师的学生是我莫大的荣幸。从硕士到现在六年来，杨老师言传身教，您那敏锐的洞察力、执着的科学探索精神和严谨的治学思想都令我深深地折服。杨老师总能高屋建瓴，站在全局和战略的位置上对我们课题研究进行把握和规划，使我们的课题能够始终紧扣国际研究前沿。您还教导我们从哲学高度对课题进行分析和把握，帮助我们理清每个研究方向的发展趋势，保证了我们研究始终沿着正确的道路推进。杨老师始终坚持利用休息时间进行每周一次的学术例会，与我们进行深入的学术讨论，并以此为乐，令我们很受感动。杨老师胸怀坦荡、为人正直、大公无私，事事从学院大局出发，展现了高尚的人格和魄力，是我们永远的学习榜样。师母唐玉华教授和杨老师一样，不论是在学习还是在生活上，都时刻关心着我们，不断对我们进行鞭策和鼓励，不辞辛劳地为我们解决这样那样的问题，鼓励我们克服学习和生活上的不如意。谢谢您，唐老师！

我要衷心感谢我的协助指导老师王戟教授，两年多来，您为我和我的博士课题研究倾注了数不清的心血和时间，我的博士课题的每一点进展都与您的指导是分不开的。您深厚的学术功底、广阔的知识面、清晰敏捷的逻辑思维能力和对问题的准确把握和分析能力都是我一直在努力学习的。王老师极富责任感、对待科研非常严谨，这种品质深深地影响着我，促使我形成严谨求实的学术作风，这将成为我人生的一大笔财富。王老师还对我的论文写作方面进行了细心的指导，无私地传授给我论文写作的大量规律和经验。这两年多来，您对我们的每一篇文章，都从框架结构、语言文字等方方面面进行了详细的指导，并亲力亲为，对论文进行多遍逐字逐句的修改，才使得论文能够发表。当我的研究陷入低潮时，您的鼓励让我能够沿着这条应该来说并不太好走的研究道路继续走下去。

感谢何连跃、丁澔、周良源、陈松政、唐晓东、孔金珠等领导和老师，感谢各位领导和老师在我在 605 教研室做工程时对我的关心和帮助，使我提高了工程能力，与您们一起工作我觉得很开心。感谢廖湘科老师、罗军老师、吴庆波老师、卢凯老师和我的师兄戴华东老师，感谢您们对我的信任和培养以及无私的帮助，令我受益良多。感谢 602 教研室的毛晓光、董威等老师，您们对我的指导和关心我铭记于心。

感谢师兄师姐戴华东、周海芳、刘军、王磊、蒋艳凰、夏军、李春江、杨沙

洲、温璞，谢谢你们对我的关心和帮助。感谢易会战、高珑、胡湘华、邓宇、富弘毅、曲向丽、晏小波、陈娟、杜静、张英、杜云飞、王攀峰、汪黎、所光、刘光辉、王之元、吴俊杰等师弟师妹，感谢杨灿群、朱晓谦等在职老师的帮助。

感谢 2000 级硕士同学和 2002 级博士同学，与你们一起学习生活是我终生难忘的回忆，特别感谢同过寝室的室友王学斌、陈俊峰、李磊、赵学秘、张承义、陈永然、王圣、王勐等。感谢班长刘祥远每次不辞辛劳地通知我各种事情。感谢同一机房学习和工作的李姗姗、所光等。

感谢硕博期间各位学员队干部胡慎信、汪长江、吉炳安、郑光辉、蔡素实、田保华等各位领导，感谢您们对我的关心和教导。感谢教务办汪审权、潘晓辉等领导和老师，感谢您们对学习期间特别是本论文付出的劳动。

衷心感谢父母的养育之恩，二十多载的寒窗苦读过程中，您们无怨无悔的鼓励和支持一直常伴我左右，令我奋发图强、砺志做人。衷心感谢岳父岳母这么多年来对我学习工作上的大力支持，和对我生活上的细致照顾，让我可以专注于我的课题研究。最后，感谢我的爱人姚姚，她给予了我深深的理解和支持，在生活上给予了我细致的照顾，在精神上是我坚定的后盾。

最后，向所有关心我、支持我的老师、同学、朋友和亲人们表示最真诚的谢意，谢谢！

参考文献

[1] J.Foster, M.Fahndrich and A.Aiken, A theory of type qualifiers. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '99) 192-203 (1999).

[2] Xiaolan Zhang, Antony Edwards and Trent Jaeger, Using CQUAL for Static Analysis of Authorization Hook Placement. Proceedings of the 11th USENIX Security Symposium (2002).

[3] Antony Edwards, Trent Jaeger and Xiaolan Zhang, Runtime Verification of Authorization Hook Placement for the Linux Security Modules Framework. ACM CCS'02 (2002).

[4] Dawson R.Engler, Benjamin Chelf, Andy Chou and Seth Hallem, Checking System Rules Using System-Specific, Programmer-Written Compiler Extensions. OSDI2000 (2000).

[5] Ken Ashcraft and Dawson R.Engler, Using Programmer-Written Compiler Extensions to Catch Security Holes. IEEE Security and Privacy 2002 (2002).

[6] M.F.Kaashoek, Dawson R.Engler, G.R.Ganger, H.M.Briceno, R.Hunt, D.Mazieres, T.Pinckney, R.Grimm, J.Jannotti and K.Mackenzie, Application Performance and flexibility on exokernel systmes. 16th ACM Symposium on Operating Systems Principles (1997).

[7] J.Kuskin, D.Ofelt, M.Heinrich, J.Heinlein, R.Simoni, K.Gharachorloo, J.Chapin, D.Nakahira, J.Baxter, M.Horowitz, A.Gupta, M.Rosenblum and J.Hennessy, The Stanford FLASH multiprocessor. 21st International Symposium on Computer Architecture (1994).

[8] W.Bush, J.Pincus and D.Sielaff, A static analyzer for finding dynamic programming errors. Software - Practice and Experience, 30(7):775-802 (2000).

[9] D.Evans, Static detection of dynamic memory errors. In Proceedings of the ACM SIGPLAN 96 Conference on Programming Language Design and Implementation (1996).

[10] Manuvir Das, Sorin Lerner and Mark Seigle, ESP: Path-Sensitive Program Verification in Polynomial Time. ACM PLDI (2002).

[11] Patrick Tullmann, Jeff Turner, John McCorquodale, Jay Lepreau, Ajay Chitturi and Godmar Back. Formal Methods: A Practical Tool for OS Implementors. IEEE. 1997.

[12] N.Shankar, S.Owre and J.M.Rushby, A Tutorial on Specification and Verification using PVS. SRI International (1993).

[13] L.C.Paulson, Isabelle: A Generic Theorem Prover. LNCS 828, (1994).

[14] R.L.Constable, S.F.Allen, H.M.Bromley, W.R.Cleaveland, J.F.Cremer, R.W.Harper, D.J.Howe, T.B.Knoblock, N.P.Mendler, P.Panangaden, J.T.Saski and S.F.Smith, Implementing mathematics with the Nuprl proof development system. Prentice Hall (1986).

[15] G.Nelson. Techniques for program verification. Stanford . 1980.

[16] Michael Hohmuth, Shane G.Stephens and Gendrik Tews, Applying source-code verification to a microkernel - The VFiasco project. Technical Report (2002).

[17] Tho Ne Win, Michael D.Ernst, Stephen J.Garland, Dilsun Kirh and Nancy A.Lynch, Using Simulated Execution in Verifying Distributed Algorithms. Fourth International Conference on Verification, Model Checking and Abstract Interpretation 283-297 (2004).

[18] P.Cousot and R.Cousot, Abstract Interpretation: a unified lattice mode for static analysis of programs by construction or approximation of fixpoints. In Proceedings of the 4th POPL, pp 238-252, Los Angeles, CA (1977).

[19] P.Cousot, R.Cousot, J.Feret, L.Mauborgne, A.Mine, D.Monniaux and X.Rival, The ASTREE analyser. In ESOP 2005 - The European Symposium on Programming LNCS 3444, Springer (2005).

[20] B.Blanchet, P.Cousot, R.Cousot, J.Feret, L.Mauborgne, A.Mine, D.Monniaux and X.Rival, Design and implementation of a special-purpose static program analyzer for safety-critical real-time embedded software. The Essence of Computation: Complexity, Analysis, Transformation LNCS 2566, Springer (2002).

[21] B.Blanchet, P.Cousot, R.Cousot, J.Feret, L.Mauborgne, A.Mine, D.Monniaux and X.Rival, A static analyzer for large safety-critical software. In Proc ACMSIGPLAN'03 Conf PLDI, pp 196-207, ACM Press (2003).

[22] P.Cousot, Verification by abstract interpretation. In Proc Int Symp On Verification - Theory & Practice - Honoring Zohar Manna's 64th Birthday, volume 2772, pp 243-268, Springer (2003).

[23] O.Grumberg, F.Lerda, O.Strichman and M.Theobald, Proof-guided underapproximation-widening for multi-process systems. In Proceedings of POPL 2005, pp 122-131 (2005).

[24] Arnaud Venet and Guillaume Brat, C Global Surveyor: Designing a Static Analyzer for NASA Flight Software. In Proceedings of VMCAI'05 (2005).

[25] Joost-Pieter Katoen. Concepts, Algorithms, and Tools for Model Checking. Lecture Notes of the Course "Mechanised Validation of Parallel Systems" . 1999.

[26] G.J.Holzmann, The Spin Model Checker. IEEE Transactions on Software Engineering 23(5), 279-295 (1997).

[27] K.L.McMillan, The SMV language. Cadence Berkeley Labs (1999).

[28] Klaus Havelund, Mike Lowry and John Penix, Formal Analysis of a Space Craft Controller using SPIN. 4th International SPIN Workshop (1998).

[29] Patrick Tullmann, Godmar Back, Ajay Chitturi, John McCorquodale and Jeff Turner, Fluke IPC Verification Project Report. Utah (1997).

[30] James C.Corbett, Matthew B.Dwyer, John Hatcliff, Shawn Laubach, Corina S.Pasareanu, Robby and Hongjun Zheng, Bandera: Extracting Finite-state Models from Java Source Code. ICSE2000 (2000).

[31] John Hatcliff, Software Model Checking using Bogor - a Modular and Extensible Model Checking Framework. ESSCaSS '04 Pedase, Estonia (2004).

[32] G.J.Holzmann, Logic Verification of ANSI-C code with SPIN. SPIN2000 Springer Verlag, LNCS 1885, 131-147 (2000).

[33] Javier Esparza, David Hansel, Peter Rossmanith and Stefan Schwoon. Efficient Algorithms for Model Checking Pushdown Systems. CAV 2000 LNCS 1855, 232-247. 2000. Springer-Verlag.

[34] Hao Chen, Drew Dean and David Wagner. Model Checking One Million Lines of C Code. Proceedings of the 11th Annual Network and Distributed System Security Symposium (NDSS) . 2004.

[35] Hao Chen and David Wagner, MOPS: an Infrastructure for Examining Security Properties of Software. ACM CCS'02 (2002).

[36] Madanlal Musuvathi, David Y.W.Park, Andy Chou, Dawson R.Engler and David L.Dill, CMC: A Pragmatic Approach to Model Checking Real Code. OSDI (2002).

[37] Madanlal Musuvathi and Dawson R.Engler, Model Checking Large Network Protocol Implementations. NSDI 2004 (2004).

[38] Juergen Dingel, Computer-Assisted Assume/Guarantee Reasoning with VeriSoft. In proceedings of the 25th International Conference on Software Engineering (ICSE'03) pp. 138-148 (2003).

[39] Klaus Havelund and Grigore Rosu. Java PathExplorer - A Runtime Verification Tool. Proc.ISAIRAS '01: 6th Int'l Symp.on AI, Robotics and Automation in Space . 2001. Nordwijk, The Netherlands.

[40] Klaus Havelund and Grigore Rosu. Monitoring Java Programs with Java PathExplorer. In proceedings of RV '01: 1st Workshop on Runtime Verification, Springer LNCS, vol.55, issue 2 . 2001. Paris, France.

[41] T.Ball, R.Majumdar, T.Millstein and S.K.Rajamani, Automatic predicate abstraction of C programs. PLDI2001: Programming Language Design and Implementation (2001).

[42] T.Ball and S.K.Rajamani, Generating abstract explanations of spurious counterexamples in C programs. Technical Report MSR-TR-2002-09, Microsoft Research, Microsoft Corporation (2002).

[43] Thomas Ball and Sriram K.Rajamani, The SLAM Project: Debugging System Software via Static Analysis. POPL 2002 (2002).

[44] S.Qadeer and S.K.Rajamani, Deciding Assertions in Programs with References. MSR Technical Report: MSR-TR-2005-08 (2006).

[45] T.Andrews, S.Qadeer, S.K.Rajamani, J.Rehof and Y.Xie, Zing: A Model Checker for Concurrent Software. MSR Technical Report: MSR-TR-2004-10 (2004).

[46] T.Andrews, S.Qadeer, S.K.Rajamani, J.Rehof and Y.Xie, Zing: Exploiting Program Structure for Model Checking Concurrent Software. In Proceedings of CONCUR 2004 (2004).

[47] Thomas A.Henzinger, Ranjit Jhala, Rupak Majumdar and Gregoire Sutre, Software Verification with BLAST. 10th Int SPIN Workshop (SPIN'2003) 2648 of Lecture Notes in Computer Science, 235-239 (2003).

[48] Sagar Chaki, Edmund Clarke and Alex Groce, Modular Verification of Software Components in C. ACM-SIGSOFT Distinguished Paper in the 25th International Conference on Software Engineering (ICSE) 2003 385-395 (2003).

[49] Sagar Chaki, Joel Ouaknine, Karen Yorav and Edmund Clarke. Automated Compositional Abstraction Refinement for Concurrent C Programs: A Two-Level Approach. 2nd Workshop on Software Model Checking (SoftMC) . 2003.

[50] Sagar Chaki, Edmund Clarke, Nishant Sinha and Prasanna Thati. Automated Assume-Guarantee Reasoning for Simulation Conformance. LNCS 3576, pp. 534-547. 2005. Proceedings of Computer Aided Verification (CAV), 2005.

[51] Sagar Chaki, Edmund Clarke, Somesh Jha and Helmut Veith, An Iterative Framework for Simulation Conformance. Journal of Logic and Computation (JLC) Vol. 15, No. 4, pp 465-488 (2005).

[52] Sagar Chaki, James Ivers, Natasha Sharygina and Kurt Wallnau. The ComFoRT Reasoning Framework. pp. 164-169. 2005. Proceedings of Computer Aided Verification (CAV), 2005, LNCS 3576.

[53] S.Khurshid, C.S.Pasareanu and W.Visser, Generalized symbolic execution for model checking and testing. TACAS, 2003 (2003).

[54] Corina S.Pasareanu and Willem Visser, Verification of Java Programs using Symbolic Execution and Invariant Generation. SPIN 2004 (2004).

[55] S.Graf and H.Saidi. Construction of abstract state graphs with PVS. CAV 97: Computer-aided Verification, LNCS 1254 , 72-83. 1997. Springer-Verlag.

[56] E.M.Clarke, O.Grumberg, S.Jha, Y.Lu and H.Veith. Counterexample-guided abstraction refinement. In Preceedings of CAV 1855, 154-169. 2000. Springer LNCS.

[57] Sagar Chaki, Edmund Clarke, Somesh Jha and Helmut Veith, An Iterative Framework for Simulation Conformance. Journal of Logic and Computation (JLC) Vol. 15, No. 4, pp 465-488 (2005).

[58] Zohar Manna and Amir Pnueli, Temporal Verification of Reactive Systems - Safety, Springer-Verlag, New York Berlin Heidelberg 1995.

[59] D.Gries, The Science of Programming. Springer-Verlag (1981).

[60] Dawson R.Engler and Madanlal Musuvathi, Static analysis versus software model checking for bug finding. ACM SoftMC 2003 (2003).

[61] Dawson R.Engler and Madanlal Musuvathi, Static analysis versus software model checking for bug finding. VMCAI04 (2004).

[62] M.Weiser. Program slicing. IEEE Transactions on Software Engineering (TSE) SE-10[4], 352-357. 1982.

[63] J.Ferrante, K.J.Ottenstein and J.D.Warren, The Program Dependence Graph and

Its Use in Optimization. ACM Transaction on Programming Language and System, Vol 9, No 3, pp 319-349, 1987 (2006).

[64] F.Tip, A survey of program slicing techniques. Journal of Programming Languages (JPL) 3, 121-189 (1995).

[65] George C.Necula, Scott McPeak and Wes Weimer. CIL: Infrastructure for C Program Analysis and Transformation. 2004.

[66] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04) . 2004.

[67] J.C.King. Symbolic execution and program testing. Communications of the ACM 19(7), 385-394. 1976.

[68] Jinhui Shan, Ji Wang and Zhichang Qi, Improved Method to Generate Path-Wise Test Data. Journal of Computer Science and Technology Vol.18(2), pp.235-240 (2003).

[69] Jian Zhang and Xiaoxu Wang, A Constraint Solver and Its Application to Path Feasibility Analysis. International Journal of Software Engineering & Knowledge Engineering Vol. 11 No. 2, pp.139-156 (2001).

[70] Xiangyu Zhang, Rajiv Gupta and Youtao Zhang, Precise Dynamic Slicing Algorithms. IEEE/ACM International Conference on Software Engineering (ICSE) 319-329 (2003).

[71] Xiaodong Yi, Ji Wang and Xuejun Yang. Verification of C Programs using Slicing Execution. In proceeding of Fifth International Conference on Quality Software (QSIC'05), Melbourne, Australia . 2005. IEEE Computer Society press.

[72] D.Detlefs, G.Nelson and J.Saxe, Simplify: A theorem prover for program checking. http://research.compaq.com/src/esc/simplify.html (2003).

[73] M.Das, Unification-based pointer analysis with directional assignments. In PLDI 00: Programming Language Design and Implementation, ACM 35-46 (2000).

[74] Sagar Chaki, Edmund Clarke, Alex Groce and Ofer Strichman. Predicate Abstraction with Minimum Predicates. 12th Advanced Research Working Conference on Correct Hardware Design and Verification Methods (CHARME) . 2003.

[75] SSL 3.0 Specification. http://wp.netscape.com/eng/ssl3 . 2005.

[76] B.Korel and J.Laski, Dynamic Program Slicing. Information Processing Letters (IPL) 29, 155-163 (1988).

[77] R.P.Kurshan, Computer-Aided Verification of Coordinating Processes - The Automata-Theoretic Approach. Princeton University Press, Princeton, NJ (1994).

[78] W.R.Bush, J.D.Pincus and D.J.Sielaff, A static analyzer for finding dynamic programming errors. Software: Practice and Experience 30(7), 775-802 (2000).

[79] Sagar Chaki, Edmund Clarke, Somesh Jha and Helmut Veith, An Iterative Framework for Simulation Conformance. Journal of Logic and Computation

(JLC) Vol. 15, No. 4, pp 465-488 (2005).

[80] Edmund Clarke, Ansgar Fehnker, Zhi Han, Bruce Krogh, Joel Ouaknine, Olaf Stursberg and Michael Theobald, Abstraction and Counterexample-Guided Refinement in Model Checking of Hybrid Systems. International Journal of Foundations of Computer Science Vol. 14(4), (2003).

[81] Thomas A.Henzinger, Marius Minea and Vinayak Prabhu. Assume-Guarantee Reasoning for Hierarchical Hybrid Systems. [Hybrid Systems: Computation and Control. 4th International Workshop. HSCC 2001], Springer Verlag, pp. 275-290. 2001.

[82] Colin Blundell, Dimitra Giannakopoulou and Corina S.Pasareanu. Assume-Guarantee Testing. 2005. In proceedings of SAVCBS'05.

[83] K.Rustan M.Leino. Efficient Weakest Preconditions. Information Processing Letters archive Volume 93, Issue 6, pp. 281-288. 2005.

[84] Cormac Flanagan and James B.Saxe. Avoiding Exponential Explosion: Generating Compact Verification Conditions. In proceedings of 28th ACM Symposium on Principles of Programming Languages (POPL'01) . 2001.

[85] Edsger W.Dijkstra. A Discipline of Programming. 1976. Prentice Hall, Englewood Cliffs, NJ.

[86] Haifeng he and Neelam Gupta. Automated Debugging using Path-Based Weakest Preconditions. In proceedins of Fundamental Approaches to Software Engineering (FASE 2004), Barcelona, Spain . 2004.

[87] Cormac Flanagan, K.Rustan, M.Leino, Mark Lillibridge, Greg Nelson, James B.Saxe and Raymie Stata. Extended Static Checking for Java. In proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI) Volume 37, number 5, pp.234-245. 2002.

[88] David L.Detlefs, K.Rustan, M.Leino, Greg Nelson and James B.Saxe. Extended Static Checking. Research Report 159, Compaq Systems Research Center . 1998.

[89] Mike Barnett and K.Rustan M.Leino. Weakest-Precondition of Unstructured Programs. In Proceedings of the 6th Workshop on Program Analysis for Software Tools and Engineering , pp82-87. 2005.

[90] Cormac Flanagan and Patrice Godefroid. Dynamic partial-order reduction for model checking software. [In proceedings of POPL 2005]. 2005. Long Beach, California, USA.

[91] P.Godefroid. Partial-Order Methods for the Verification of Concurrent Systems - An Approach to the State-Explosion Problem. 1996. Vol. 1032 of Lecture Notes in Computer Science.

[92] A.Valmari. Stubborn sets for reduced state space generation. pp. 491-515. 1991. In Advances in Petri Nets 1990.

[93] Kimmo Varpaaniemi, Minimizing the Number of Successor States in the Stubborn Set Method. Journal of Fundamental Informatics 51, 215-234 (2001).

[94] D.Peled, Combining partial order reductions with on-the-fly model checking. Computer Aided Verification, CAV '94, LNCS 818, Springer (1994).

[95] G.J.Holzmann and D.Peled, An improvement in formal verification. In Formal Descriptions Techniques VII, FORTE'94, Chapman & Hall (1995).

[96] Shaz Qadeer, Sriram K.Rajamani and Jakob Rehof, Summarizing Procedures in Concurrent Programs. In proceedings of POPL '04 (2004).

[97] Twan Basten, Dragan Bosnacki and Marc Geilen, Cluster-based Partial-Order Reduction. Autom Softw Eng 11(4) 365-402 (2004).

[98] T.Basten and D.Bosnacki, Enhancing Partial-Order Reduction via Process Clustering. In proceedings of Automated Software Engineering, ASE '01, IEEE Computer Society Press (2001).

[99] Tony Hoare and Jay Misra, Verified software: theories, tools, experiments. Microsoft Research Ltd and the University of Texas at Austin (2005).

[100] F.Mattern, Virtual Time and Global States of Distributed Systems. In Proceedings of Workshop on Parallel and Distributed Algorithms 215-226 (1989).

[101] SSL 3.0 Specification. http://wp.netscape.com/eng/ssl3 . 2005.

[102] Dragan Bosnacki and Gerard J.Holzmann, Improving Spin's Partial Order Reduction for Breadth First Search. In Proceedings of the 12th SPIN Workshop on Model Checking Software San Francisco, USA (2005).

[103] D.A.Peled, Combining Partial Order Reductions with On-the-Fly Model Checking. Formal Methods on Systems Design, 8: 39-64, 1996 (1996).

[104] P.Godefroid, Partial Order Methods for the Verification of Concurrent Systems: An Approach to the State Space Explosion. LNCS 1032, Springer (1996).

[105] JPF Documentation: On-the-fly Partial Order Reduction. http://javapathfinder.sourceforge.net/On-the-fly_Partial_Order_Reduction.html (2006).

[106] R.J.Lipton, Reduction: A method of proving properties of parallel programs. In Communications of the ACM, volume 18:12, pages 717–721 (1975).

[107] C.Flanagan and S.Qadeer, Transactions for software model checking. In SoftMC 03: Software Model Checking Workshop (2003).

[108] Sagar Chaki, Edmund Clarke, Somesh Jha and Helmut Veith, An Iterative Framework for Simulation Conformance. Journal of Logic and Computation (JLC) Vol. 15, No. 4, pp 465-488 (2005).

[109] Thomas Ball, Byron Cook, Shuvendu K.Lahiri and Lintao Zhang, Zapato: Automatic Theorem Proving for Predicate Abstraction Refinement. In proceedings of CAV'03: International Conference on Computer-Aided Verification (2004).

[110] J.C.Filliatre, S.Owre, H.Rueb and N.Shankar, ICS: Integerated Canonizer and Solver. In CAV'01: International Conference on Computer-Aided Verification, pp 246-249 (2001).

[111] Clark Barrett and Sergey Berezin, CVC Lite: A New Implementation of the Cooperating Validity Checker. In Proceedings of CAV'04: International

Conference on Computer-Aided Verification (2004).

[112] Cormac Flanagan, Rajeev Joshi, Xinming Ou and James B.Saxe, Theorem Proving using Lazy Proof Explicatioin. In CAV'03: 15th International Conference on Computer-Aided Verification (2003).

[113] Lawrence C.Paulson, Isabelle: A Generic Theorem Prover. Lecture Notes in Computer Science vol 828, Springer-Verlag, Berlin (1994).

[114] Matt Kaufmann and J Moore, An industrial strength theorem prover for a logic based on Common Lisp. IEEE Transactions on Software Engineering, 23(4):203-213 (1997).

[115] Michael Norrish and Konrad Slind, A thread of HOL development. Computer Journal, 45(1):37-45 (2002).

[116] Paul B.Jackson, The Nuprl Proof Development System. Version 4 1 Reference Manual and User's Guide, Cornell University, Ithaca (1994).

[117] J. R. S.Owre and N.Shankar, PVS: A Prototype Verification System. 11th International Conference on Automated Deduction, volume 607 of Lecture Notes in Artificial Intelligence, 748-752, Springer-Verlag (1992).

[118] William Pugh, The Omega Test: a fast and practical integer programming algorithm for dependence analysis. Communications of the ACM, 35(8):102-114 (1992).

[119] D.C.Cooper, Theorem proving in arighmetic without multiplication. In Machine Intelligence, volume 7, pages 91-99, New York, American Elsevier (1972).

[120] Michael Norrish, Complete Integer Decision Procedures as Derived Rules in HOL. In proceedings of 16th International Conference on Theorem Proving in Higher Order Logics, LNCS 2758, pp 71-86, Springer-Verlag (2003).

[121] Matthew W.Moskewicz, Conor F.Madigan, Ying Zhao, Lintao Zhang and Sharad Malik, Chaff: Engineering an Efficient SAT Solver. In Design Automation Conference, pp 530-535 (2001).

[122] J.P.M.Silva and K.A.Sakallah, GRASP - A New Search Algorithm for Satisfiability. In Proceedings of the International Conference on Computer-Aided Design (1996).

[123] R.E.Shostak, On the SUP-INF method for proving Presburger formulas. Journal of the ACM 24, 529-543 (1977).

[124] R.E.Shostak, Deciding linear inequalities by computing loop residues. Journal of the ACM 28, 769-779 (1981).

[125] W.Harvey and P.Stuckey, A unit two variable per inequality integer constraint solver for constraint logic programming. In Australian Computer Science Conference (Australian Computer Science Communications) (1997).

[126] Himanshu Jain, Daniel Kroening, Natasha Sharygina and Edmund Clarke, Word Level Predicate Abstraction and Refinement for Verifying RTL Verilog. In proceedings of Design Automation Conference (DAC'05) (2005).

[127] Byron Cook, Daniel Kroening and Natasha Sharygina, COGENT: Accurate

Theorem Proving for Program Verification. In proceedings of the Computer-Aided Verification (CAV) 2005 (2005).

[128] Daniel Kroening, Linear Arithmetic with Bit-Vectors using Omega and SAT. Technical Reports 483, ETH Zurich, Computer Systems Institute, April 2005 (2005).

[129] K.L.McMillan and Nina Amla, Automatic Abstraction without Counterexamples. Ninth International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2003), Warsaw, Poland 2-17 (2003).

作者在学期间取得的学术成果

参加的项目：

1. 实时系统软件可靠性测试与验证，国家自然科学基金重点项目，No. 60233020，2003.1 – 2006.12
2. 自治式软件开发关键技术研究，国家 863 计划项目，No. 2005AA113130，2005.7 – 2006.6
3. 银河麒麟服务器操作系统内核，国家 863 重大软件专项，No. 2002AA1Z2101，2002.1 – 2005.12

获得的专利：

1. 易晓东，何连跃，罗军. 安全操作系统角色定权框架，已公示，申请号 200410046919，公示号 1773413

发表的文章：

1. Xiaodong Yi, Ji Wang and Xuejun Yang. Slicing Execution for Model Checking C Programs. International Journal of Software Engineering and Knowledge Engineering (IJSEKE), Vol. 16, No. 5, World Scientific 2006 (国外期刊，SCI 检索)
2. Xiaodong Yi, Ji Wang and Xuejun Yang. Stateful Dynamic Partial-Order Reduction. In Proceedings of the Eighth International Conference on Formal Engineering Methods (ICFEM 2006), Macau, China, Lecture Notes in Computer Science 4260, Springer 2006 (SCI 检索)
3. Xiaodong Yi, Xuejun Yang. A Security Verification Method for Information Flow Security Policies Implemented in Operating Systems. In proceedings of ICICS 2003, Lecture Notes in Computer Science 2836, pp.280-291, Springer 2003 (SCI 检索)
4. Xiaodong Yi, Ji Wang, Xuejun Yang. Verification of C Programs using Slicing Execution. In proceedings of Fifth International Conference on Quality Software (QSIC 2005), pp.109-116, Melbourne, Australia, IEEE Computer Society 2005.
5. Ji Wang, Xiaodong Yi and Xuejun Yang. Towards a Framework for Scalable Model Checking of Concurrent C Programs. In proceedings of ISoLA 2006,

IEEE press, Cyprus 2006

6. Xiaodong Yi, Xuejun Yang. An Information Flow Security Policy Verification Methodology and Its Application in Operating Systems, In Proceedings of the 11th Joint International Computer Conference (JICC 2005), pp.700-703, World Scientific 2005
7. 易晓东，王戟，杨学军. 基于 Assume-Guarantee 搜索复用的 C 程序验证方法. 软件学报，已录用，2006.10
8. 易晓东，何连跃，杨学军. 一个提高安全操作系统易用性的修改 BLP 模型及其实现. 第三届研究生学术活动周论文集优秀论文，2003
9. 易晓东，何连跃，杨学军. 安全操作系统基于角色的授权机制. 计算机工程与科学，第 2004A1 期，2004
10. 易晓东，杨学军. 一个灵活的操作系统安全框架 FMAC. 计算机科学，Vol. 33, No. 1, 2006
11. 易晓东，杨学军. 一种 C 程序断言的全自动静态验证方法. 计算机科学，Vol. 33, No. 9, 2006
12. 易晓东，杨学军. 基于谓词抽象的软件安全性验证. 2005 中国计算机大会论文集，清华大学出版社, 2005
13. Xuejun Yang, Xiaodong Yi and Ji Wang. Slicing Execution with Partial Weakest Precondition for Model Abstraction of C Programs. 已投稿
14. 易晓东，王戟，杨学军. 一种面向源代码形式验证的完备整数线性判定方法. 计算机学报，已投稿