学校代码：10385　　分类号：_____

研究生学号：1100214010　　密　级：_____

华侨大学

硕 士 学 位 论 文

基于概率模型检测的分布式算法验证和分析

Verification and Analysis of Distributed Algorithms based on Probabilistic Model Checking

作者姓名：刘来

指导教师：骆翔宇　副教授

学　　科：计算机应用技术

研究方向：模型检测

所在学院：计算机科学与技术学院

论文提交日期：二零一四年三月三十一日

学位论文独创性声明

本人声明兹呈交的学位论文是本人在导师指导下完成的研究成果。论文写作中不包含其他人已经发表或撰写过的研究内容，如参考他人或集体的科研成果，均在论文中以明确的方式说明。本人依法享有和承担由此论文所产生的权利和责任。

论文作者签名：_____ 签名日期：_____

学位论文版权使用授权声明

本人同意授权华侨大学有权保留并向国家机关或机构送交学位论文的复印件和电子版，允许学位论文被查阅和借阅。本人授权华侨大学可以将本学位论文的全部内容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和汇编本学位论文。

论文作者签名：_____ 指导教师签名：_____

签 名 日 期：_____ 签 名 日 期：_____

摘　要

分布式算法在实际应用中具有重要的价值意义，本文采用一种基于概率模型检测技术验证和分析了两类分布式算法的性质，并这些性质给出了相应的证明。由于概率模型检测可以穷举状态空间，相比于传统方法，使得它更接近真实应用环境,更利于我们研究算法的安全性和有效性；同时概率模型检测器中的马尔科夫技术可以分析出模型中那些不确定因素。

随机分布式算法中有一种自动恢复状态的算法—自稳定算法，Mitchell Flatebo 提出的自稳定算法是通过一个简单随机的解决方案解决 n 个进程令牌环中的故障，使故障自动恢复正常。他证明出最坏的情况有 n-1 个令牌，所需要的时间的上界是 $0.5N^2$-0.5N-2。从实验中，发现这个上限出现的可能很小。而在实际应用中，我们更注重系统预期恢复稳定状态的时间—预期最差时间。实验中，主要采用 PRISM 工具验证了 Flatebo 算法的性质，并且把它与同类型 Herman 自稳定算法做了比较，当系统中的结点数目增加时，它的性能会逐渐提高，优于 Herman 算法；另外，还用线性回归从实验结果中拟合出预期最差时间在 $O(N^{1.45})$ 和 $O(N^{1.47})$之间；最后还给出一个验证和分析自稳定算法的简单方法。

本文研究的另一个分布式算法是随机互斥算法，提出了一个基于概率模型检测工具 PRISM 方法,用于验证和分析 Kerry Raymond 提出的分布式 K 互斥算法，首先验证了互斥算法无死锁和无饥饿两种基本性质；然后通过 PRISM 验证某一进程进入临界区平均及时时间；接着分析我们的实验结果，当各个进程的相对访问临界区资源的时间相差不大时，增加临界资源 k 的数量，不能很明显提高进程的平均及时时间，但如果其中某一进程访问临界资源的时间较长，那么增加 k 的值，就会大大提高运行效率。随后我们通过添加额外的存储结构，获得了新的改进 k 互斥算法，可以大大降低进程之间的发送和接受延时。最后给出改进的 k 分布式算法。

关键词：自稳定算法　Flatebo 算法　预期最差时间　Kerry 算法　概率模型检测

Abstract

The value of distributed algorithms has important significance in practical application. We propose a formal verification and analysis method for two classes of distributed algorithms, and prove the experiment of algorithm nature, based on probabilistic model checking. Compared to the traditional methods, the technology could be exhaustive state space, it makes the model more close to the real environment, then more conductive to validity and safety of researching algorithm by us; in addition, Markov method can analysis and verify uncertain factors in the probabilistic model checker.

The classes of algorithms for automatic recovery state in randomized distributed algorithm, self-stabling algorithm. An algorithm resolves from faults in an N-process token ring by putting for a simple randomize solution by Mitchell Flatebo. He gives a conclusion for maximum stable time, $0.5N^2$-0.5N-2, in the worst condition. But the probability is very small. The expected worst time is important to our system in practical application. We verify and analysis the algorithm by Flatebo, based on PRISM, and compared to another self-stabling algorithm by Ted Herman. With the increase of number of nodes, we find that its performance is gradually improver than Herman's algorithm. Finally, we use linear regression to matching approximate value of the expected worst time, and give a simple validation and analysis method of self-stabling algorithms.

The paper researches another distributed algorithm, randomized mutual exclusion algorithm. It proposes a verification and analysis method for K-mutual exclusion algorithm by Kerry Raymond, based on probabilistic model checking. We first verify two basic properties of mutual exclusion, no deadlock and no starvation. Then we verify the expected worst time for a process enter a critical region by PRISM; in the experiment, we find that the change of the number K of critical regions has slight effect on the average timely time of one or any process to enter critical region. We further find that when the running time of one process is much longer than others,

running efficiency of the algorithm can be improved by increasing the value of K. And we put forward a new k mutual exclusion by appending addition storage structure. It cloud reduce number of sending and receiving messages for processes. Finally, it gives improved K distributed algorithm

Keywords： self-stabling Flatebo's algorithm the expected worst time Kerry's algorithm probabilistic model checking

目　录

第 1 章　引言

1.1 研究背景和意义

1.1.1 概率模型检测的背景

如今，硬件和软件系统广泛地应用于各个领域，如：电子商务、电话交换网络、航空领域控制系统和医疗设备等等。同时也伴随着一些不可忽略的故障和危险，这些问题可能出现在各类软硬件系统中。例如，1996 年 6 月 4 日，Ariane 5 号火箭发射 40 秒后爆炸，后来发现软件系统在传输火箭运行数据时，错误把一个浮点型的数据转换成整型，由于没有对于这种数据进行异常处理而导致了整个计算失败。显然，在一些先进技术的领域，可靠的硬件系统和软件系统将变得至关重要。

数理逻辑能够准确地描述程序的各种不同的属性，可以对系统中的有穷状态数建立模型，通过穷举整个模型来验证这些属性。早在 1980 年，Clarke 和 Emerson 提出了模型检测，一种用于自动验证有限状态并行系统的方法；另外，Quielle 和 Sifakis 也提出了相同的方法。这种方法成功地应用于验证各类复杂的时序电路设计和网络协议等。首先，建立的模型需满足安全性、灵活性和公平性等约束；利用模型描述高层系统的方式，例如：进程代数，Petri 网络，这些模型将会生成非确定的有限状态的自动机，这个自动机包含了系统产生的所有可能的状态和属性，可以被看做成一个有向状态传递图；原子命题表示在每一个状态成立的基本属性；我们用特殊的规格方式来表示这种形式化的属性：LTL(线性时态逻辑)，CTL(计算树逻辑)。此时，我们就可以通过建立的模型来验证一个形式化的属性。

1.1.2 概率模型检测的分布式应用意义

模型检测对于软硬件中的固有属性，相关性质等的验证具有非常大的意义；但对于实际生活中，可能会涉及到一些不确定因素，例如：系统一直稳定的概率是多少？概率模型检测的提出，很好地解决这一类问题。把马尔科夫链的转换技术用于系统中各个状态的迁移关系。同时，除了能解决传统模型检查解决的问题，还能用于分析具有概率性的逻辑公式；因此，它能分析各类具有概率

性的领域，如：安全协议、分布式算法验证、DNA 遗传分析、心理咨询等等。

在随机分布式算法的验证，概率模型检测还是较为广泛的，早期的有对 Byzantine 协议，哲学家进餐，同步和异步领导选举协议，随机互斥算法等相关的验证，但没有深刻研究这些算法的性质和意义，后期在 2012 年，有人对 Herman 自稳定算法做出了详细的验证和分析，得出了比较显著的结果，从对各类自稳定算法的分析中得出 Herman 算法的效率相对其它算法最高效，同时利用概率模型检测工具 PRISM 让 Herman 算法的预期稳定时间精确地定位在 $0.64N^2$。而在国内，用概率模型检测技术来验证和分析分布式算法还较少。如果我们能对一些常用的分布式算法做出更深的验证和分析，这将非常有利于我们在实际生产中对分布式算法的使用。

1.2 概率模型检测的研究现状

在 20 年代，随着概率模型检测的相继提出，特别是在 1980~1990 年之间，从符号化模型检测到概率模型检测的扩展越来越多，但由于实际应用不大，直到后 10 年之间才出现一些机构研发出各种各样的工具，主要有英国 Birmingham 大学的 PRISM[17,18]，德国 Erlangen-Nürnberg 大学和荷兰 Twente 大学的 ETMCC 和德国 RWTH Aachen 大学和荷兰 Twente 大学的 MRMC， 这三种工具主要用于数字分析技术；美国匹兹堡 Carnegie Mellon 大学的 YMER 和伊利诺斯州 Urbana Champaign 大学的 VESTA，这两种工具主要用于统计分析。其中 PRISM[17, 18]工具总体优于其它，随后在 20 年代后，出现了大量基于 PRISM[17, 18]工具在分布式算法上的验证的文章，还有通讯，网络，多媒体协议的验证，安全协议、生物学、规划及合成、博弈论和能源管理等。近几年，在国内也有不少相关的工作，对各类算法的验证[55,57,58]，其中，包括还包括用概率模型检测技术对其他领域的应用[56]，等。

概率模型检测是一种自动分析系统中随机属性的形式化验证技术[43]；它类似于传统的模型检测，主要区别在于它添加了状态之间迁移的可能性和迁移的时间等，使得它能模拟概率随机属性，并且验证它们。概率模型检测引用一系列的技术计算事件在系统运行中发生的概率，我们可以对此建立各种不同的属性，例如：“系统自动关闭的概率是多少”，“系统终止前发生了多少次故障”，“消息发送 5 秒钟后被接收的概率”[48]等等。如下图 1.1，它表示一个概率模型检测

工具获得一个概率模型和一个属性验证公式，然后建立模型，计算出结果，最后返回 Yes（当属性满足条件，否则为 No）或者返回一个概率值（时间值）。

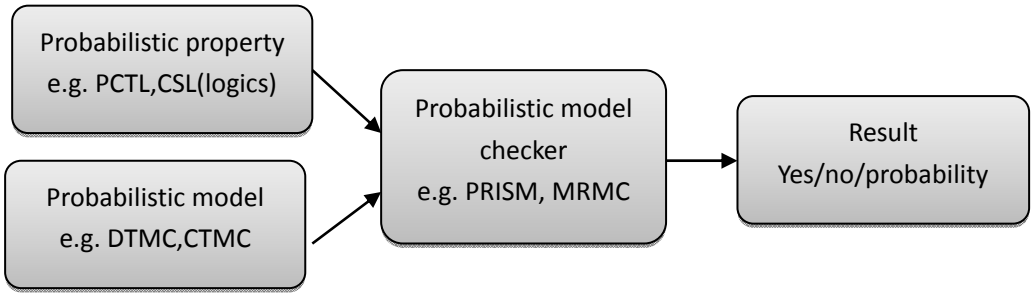


图 1.1　概率模型检测工具概述

1.3 分布式算法

分布式系统是由一组独立的实体构成的集合，这些实体相互协作就能够解决单个实体不能解决的问题。分布式系统很早就存在，从狼群到人类，再到整个生态系统中，都可以发现自然界中的各类生物之间的信息交互。随着 Internet 的迅速发展，分布式系统将越来越被人们广泛使用，并把它们部署都人们所需要的地方。如：电信、分布式处理系统，科学计算，实时控制系统等。分布式算法甚至还能有助于生物学和心理学上的研究，从计算机系统的角度来看，一个分布式系统共有 4 种特征：

（1）当你正在使用计算机时，突然操作系统出故障，但却不会影响你的正常工作；

（2）在一组计算机中，它们不共享内存，通过相互传递消息来通信，这样的计算机组一般来讲是半自治的，它们需要通过松散耦合来解决一个问题；

（3）虽然这是一组相互独立的计算机系统，但是从外在看来，它们具有一致性功能；

（4）可以用弱耦合来描述一系列计算机，例如广域网；也可以是强偶然，如局域网；更强耦合，如多处理系统。

分布式算法的覆盖的范围很广，主要是被广泛使用的各类并发算法，如上面所说的，分布式算法不仅应用于局域网中，在多处理器中共享存储器也被大量使用。虽然是不同的环境，但这些环境都有相似之处，使得这些分布式算法

能够应用于各种各样的环境中。分布式算法有很多种类型，进程之间通讯，时序模型，故障模型和需解决的问题。

上面我们已经讲到分布式算法的重要性，那么在分布式系统中，进程是否能够正常工作，或者分布式算法是否能够满足我们系统的要求，例如：当某个系统中进程在获得资源时出现故障，此时我们通过自稳定算法来使得系统中的进程恢复到正常工作中去，并且必须在规定的时间内完成这个过程。如果自稳定算法的效率较低，那就不符合我们的需求，因此我们的目的就是要找出高效的自稳定算法；从另一个角度讲，如果我们知道某个自稳定算法的性能，那么我们就可以设置系统出现故障到恢复正常的时间。本文主要讲的是对两类分布式算法的验证，分析和证明：自稳定算法和互斥算法。这两种算法中，我们都需要区分它们在分布式系统中进程的同步性或异步性，以此来适宜描述不同的系统进程行为。

对于分布式算法而言，国外有不少学者用概率模型检测 PRISM 已经对其作出了研究，包括有随机自稳定算法、同步领导选举协议、异步领导选举协议、哲学家进餐、随机互斥算法、拜占庭协议等。采用概率模型检测的穷举状态空间的思想来验证分布式算法，这样得出来的结果更加具有实际意义。对于这些算法的研究，主要对它们做了简单的验证，进一步研究较少，近年有对 Herman 自稳定算法做了相关的分析和验证，取得了较为客观的实验价值。本文中对自稳定算法做了进一步分析、验证和比较。另外还研究过一些比较典型的互斥算法，主要有 Kerry 互斥算法等。

1.4 研究的内容和贡献

1.4.1 论文的主要研究内容和工作

本文主要通过对分布式算法进行验证和分析，得出一些符合实际应用中的结果；同时，对于这些结果，还采用一些不同的方法来分析它们的正确性和可行性。主要研究的算法：Flatebo 异步自稳定算法和 Kerry 多进程互斥算法。对于这两种类型的算法，还提出了一套简单的验证和分析方法。

（1）自稳定算法，是采用一个简单的机制，使得系统中的所有进程按照这个机制来执行，不管系统中的进程当前处于什么状态，经过有限步后，系统将会自动达到一个合法状态（稳定状态），并且会一直保持这个合法状态一直运行

下去。对于本文研究的 Flatebo 算法，我们在这里用 PRISM[17, 18]工具对它进行简单的建模，然后使用概率模型检测中的概率时序逻辑公式（如 PCTL，CSL 等）验证算法的属性：无饥饿、无死锁、闭合性和可达性。紧接着验证其他算法的性质，例如在实际应用中，系统可能会出现某个故障（系统中进程出现死锁），然后需要多久才能再次达到合法状态。对于这一性质的验证，Herman[1,7]同步算法已经给出了一个高效的机制，但对于异步系统，这种机制就不适用了，我们通过实验证明 Flatebo 算法在这方面的效果会更合适。

（2）多进程互斥算法，是指多个进程互斥访问临界资源。最开始在 1981 年，多进程互斥算法被提出，当时算法中的临界区资源只有一个，在后来的研究中扩展了这一临界资源的数量，理论上使得进程在访问临界资源时的时间减少。Kerry 算法[9]就是这样一种算法，它表示 n 个进程访问 k（k<n）个临界区资源。可是临界资源的增加多少才能使得进程能够快速进入临界区并不明确，这里就可以通过 PRISM[17, 18]工具对其算法建立模型，然后通过概率时序逻辑公式获得出这一类结果。另外，我们进一步得出当有进程在临界区执行时间相对于其他进程很长时，才会需要增加临界资源来提高系统运行效率。

1.4.2 论文研究的特色及创新点

上节给出了两类算法简单的综述，本文的特别和创新：

（1）对于自稳定算法，本文除了验证它本身的属性外，另一重要的特点是，分析它在实际中应用的合理性，而且还采用线性回归对 Flatebo 算法这一性质进行分析；从得出的结果中，我们发现它的性能要优于其它同类算法。本文还把它与高效的 Herman 同步算法做了比较。

（2）在 Kerry 互斥算法[9]中，我们通过实验建立模型，同样除了验证它的本身性质外，还分析它在实际中的意义，得出了不同临界资源的平均及时时间，并且对其结果进行了证明，证明结果与实验结果完全相同，这一结论适合任意进程数的互斥算法。并且给出了进一步假设条件，得到更好的实验结果和证明分析。在 k 互斥算法中，我们还提出新的 k 互斥算法，大大降低了进程之间的相互发送消息时间。

从上述，对于其他自稳定算法和互斥算法，我们还可以用同样的方法来进行的验证和分析，并且对一些较为的特出的算法给出一些合理的证明过程。通过后面的改进算法，我们也可以从中看出算法验证也有利于新算法的提出。

1.5 论文结构

全文共有 5 章，各章的内容如下：

第 1 章是引言，主要讲解了从模型检测到概率模型检测的背景，概况了它们在实际领域中的应用，同时还介绍了分布式算法的应用意义，以及国内外在概率模型检测上验证分布式算法的现状；接着讲解本文的研究内容和工作，并且在以往的基础上提出了新的问题和证明结果，并给出建模，验证，分析和证明等方法。

第 2 章主要介绍概率模型检测工具 PRISM[17, 18]，首先是它的简单概要，然后详细讲解它的结构和它的应用领域，其中包括马尔科夫链的介绍，以及各类概率模型检测工具的相关比较。

第 3 章讲解 Flatebo 算法及其算法性质的简单介绍，然后对其算法建立模型，接着就是对其性质进行验证分析；扩展它在实际应用中的属性和利用线性回归分析我们的实验结果，比较它和 Herman 算法的差异性，以及它们之间的优缺点，最后在总结中给出分析同类自稳定算法的方法。

第 4 章介绍 Kerry 互斥算法[9]和它的性质，然后是对算法建模和验证它的性质，最后如 Flatebo 算法一样，提出它在实际中可能出现的性质，然后对其验证，并证明我们的结果，然后从此基础上扩展到同类的互斥算法的分析方法，给出总结。并提出新的算法和算法结论。

第 5 章简要地介绍当前研究工作总结和未来研究展望。

第 2 章　概率模型检测

2.1 引言

与一般的模型检测一样，概率模型检测也是验证系统中的随机行为；但不同的是，它能检测达到某一个状态的概率，同时也能给出到达此状态的预期运行时间在前一章中已经简单的提到了概率模型检测，接下来主要介绍各类模型检测工具，这些工具主要用来分析数值型和统计型的概率模型。在所有的概率模型检测工具中都通过把马尔科夫链技术应用到模型检测中的各个状态迁移的关系中，概率模型检测工具就可以把概率时序逻辑属性公式融入这些状态迁移中去，这样我们就可以获得由概率时序逻辑属性公式的实验验证结果。

为了验证模型中的一些随机行为，首先，我们需要给系统建立一个形式化的概率模型；一般我们会用的概率模型有 5 种：离散时间马尔科夫链（DTMC）、连续时间马尔科夫链（CTMC）、马尔科夫决策过程（MDP）、随机 Petri 网络和 Bayesian 网络。这些模型是通过概率理论和图论来实现的。它的构造包括状态、迁移和连接它们的弧。目前为止我们主要是通过马尔科夫进程[53]来完成的，这些模型将会用到本文后面将要提到的所有模型检测工具中。在模型中，马尔科夫链被看出一个迁移系统，状态将通过一些概率分配值被分配到指定的状态中去。在马尔科夫链中，某一状态只受前一个状态的影响，而与前面其他所有状态无关，这就能满足我们模型中产生的状态图的迁移关系了。

当模型已经建立完成后，接下来我们需要知道它是否满足一个特定的形式化规格（或者属性），这些属性是用时态逻辑来描述的，本文中主要用到的时态逻辑有 2 种：概率计算数逻辑（PCTL）和连续随机逻辑（CSL），它们分别用于 DTMC 模型和 CTMC 模型中。

最后当模型和属性都完成后，现在我们需要来某种方法来验证属性，例如：$Sat(\Phi)=\{s \in S \mid s \vDash \Phi\}$，表示如果状态 s 属于集合 S，则它是否满足 Φ。本章解决这个问题的方法有 2 种：数值化方法和统计方法。

本章结构如下：2.2 节介绍 DTMC 模型和 CTMC 模型；2.3 节介绍概率模型时序逻辑：PCTL 和 CSL；2.4 节介绍模型检测马尔科夫链；2.5 节介绍给类模型检测工具；2.6 节是本章小结。

2.2 概率模型

本节主要讲解前面所提到的概率模型中的两种模型：DTMC 模型和 CTMC 模型。更多详细请看[41]。

2.2.1 DTMC 模型

DTMC 是指从一个状态转移到另一个状态的概率迁移系统模型。

定义 2.1：DTMC（离散时间马尔可夫链）DTMC 模型 $M = (S, \bar{s}, P, L)$：

S：有穷状态集合；

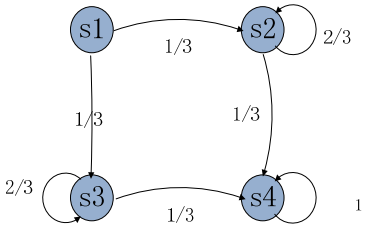
$\bar{s}$：是模型的初始状态集合，$\forall s \in \bar{s} \rightarrow s \in S$。

P：$S \times S \rightarrow [0,1]$，一个转移概率矩阵，对于所有的 s，满足 $\sum_{s' \in S} P(s, s') = 1$;

L：$S \rightarrow 2^{AP}$，每个状态映射到原子命题集合的标记函数，这个集合属于 AP。

DTMC[47]是一个可变状态（任意时刻 n=0,1,2,…）的有限状态空间的随机模型系统，它的性质：假如系统在时间 n 时进入状态 s，在一个时间单元之后，它将在时间 n+1 进入 s' 的概率为 $P(s, s')$；把系统在时间 n 进入状态 s 换成时间 n-1，后边的不变，其概率值也不会变化。定义中给出了标记原子命题的状态，也就是上面系统中的状态。系统可以根据给定的迁移概率矩阵来改变状态的迁移变化，例如：当 $P(s, s') > 0$ 时，状态 s 才能迁移到状态 s'，否则不能发生迁移。从图 2.1，我们可以看出系统也能在某一迁移之前（或之后）重复地占用某一个状态，例如系统可以包含一个自环图。从定义中我们也能得出一条有状态和迁移组成的有限（无限）序列 $\omega = s_0 \xrightarrow{p_0} s_1 \xrightarrow{p_1} s_2 \xrightarrow{p_2} \cdots \xrightarrow{p_{i-1}} s_i \xrightarrow{p_i} \cdots$，$i \in N, s_i \in S, p_i \in (0,1]$，在这里对于所有的 $i \geq 0$，都有 $p_i > 0$ 和 $p_i = P(s_i, s_{i+1})$。

现在我们把 DTMC 描述成一个状态迁移图，图中包含状态和迁移（有概率值）。如图 2.2 所示，就是一个状态迁移图，以及它的状态转换矩阵，图 2.2(a) 表示一个概率模型检测生成的概率迁移图，从图中我们可以看出，状态 s1 到状态 s3 的概率为 1/3，状态 s3 到状态 s3 的概率为 2/3，每个状态的出度之和为 1，图中的迁移关系可以无穷尽地迁移下去。



$$P=\begin{matrix}0 & 1/3 & 1/3 & 1/3\\ 0 & 2/3 & 0 & 1/3\\ 0 & 0 & 2/3 & 1/3\\ 0 & 0 & 0 & 1\end{matrix}$$

图 2.1（a） 状态迁移图　　图 2.1（b） 状态迁移矩阵

2.2.2 CTMC 模型

CTMC 模型[24]扩展了 DTMC 模型，它们之间的不同在于 DTMC 模型描述的是分步离散时间，而 CTMC 模型可以允许连续时间。这表示 CTMC 模型中的状态可以发生在任意的时间里。所以当前状态对于前面状态没有任何联系，下一次迁移只能依靠当前状态。CTMC 模型一般用于分析系统的性能和可靠性。例如在实时应用，它能判别在一个安全临界系统中失败的平均时间，以及鉴定在高速通信网络中的瓶颈。

定义 2.2：CTMC（连续时间马尔可夫链）CTMC 模型 $M=(S,\bar{s},\mathrm{R},L)$：

S：有穷状态集合；

$\bar{s}$：是模型的初始状态集合，$\forall s\in\bar{s}\rightarrow s\in S$；

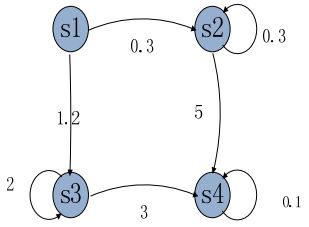
R：$S\times S\rightarrow\Omega_{\geq 0}$ 一个比率转移矩阵，对于所有的 s，满足 $\sum_{s'\in S}R(s,s')=1$；

L：$S\rightarrow 2^{\mathrm{AP}}$，每个状态映射到原子命题集合的标记函数，这个集合属于 AP。

除了概率矩阵，在 CTMC 模型集合中，初始状态与 DTMC 相同；在 CTMC 中，概率矩阵 P 被换成了比率矩阵 R，如果 $R(s,s')=0$ 表示状态 s 不能到状态 s'，因为它们之间的概率值为 0；假如 $R(s,s')>0$，那么状态 s 就有一个后继状态 s'。在单位时间 t 内，从状态 s 到它的后继状态 s' 的概率为 $1-e^{R(s,s')\cdot t}$。此时，状态 s 可能不止一个后继状态 s'，那么我们需要对状态 s 的后继做一些处理；假设状态 s 有多个后继状态，当系统进入状态 s 时，我们开始一个倒计时钟给状态 s 的所有出度迁移，最早计数到 0 的就是状态 s 的迁移了。当然这里我们需要所有的出度比率了，$E(s)=\sum_{s'\in S}R(s,s')$。在时间 t 内，状态 s 移动到状态 s' 的概率值为：$P(s,s',t)=(1-e^{R(s,s')\cdot t})\cdot R(s,s')/E(s)$，其中 $(E(s)\neq 0)$。另外，

CTMC 模型还可以才用归一化的方法转换成 DTMC 模型[41,45,39]。

在图 2.3（a）中显示了一个 CTMC 模型的状态迁移图，右边是与它的比率矩阵图。



$$P = \begin{matrix} 0 & 0.3 & 1.2 & 0 \\ 0 & 0.3 & 0 & 5 \\ 0 & 0 & 2 & 3 \\ 0 & 0 & 0 & 0.1 \end{matrix}$$

图 2.2（a） 状态迁移图　　图 2.2（b） 状态迁移图

2.2.3 MDP 模型

定义 2.1：MDP（马尔可夫决策过程）模型 $M = (S, \bar{s}, Q, L)$：

S：有穷状态集合；

$\bar{s}$：是模型的初始状态集合，$\forall s \in \bar{s} \rightarrow s \in S$。

Q：$S \times S \rightarrow [0,1]$，一个转移概率矩阵，对于所有的 s；

L：$S \rightarrow 2^{AP}$，每个状态映射到原子命题集合的标记函数，这个集合属于 AP。

从上面我们可以看出 MDP 与 DTMC 区别不大，这说明在概率模型检测工具中，这些模型都可以相互转换，在第 4 章的实验验证中，我们验证了这一特点。MDP 模型的状态迁移图和状态矩阵也类似于 DTMC。试验中我们发现它们建立的模型完全相同，所不同的是它们表达的时序逻辑属性公式的概率不一样，后面我们将会详细讲解。

2.3 概率模型时序逻辑

前面提到当系统建立成一个模型后，接着我们需要检验这个模型是否满足形式化规格（属性），这些属性用形式化的时序逻辑来表示，这种逻辑方式能够使我们定性化或者定量化出概率系统中的属性。本节主要介绍 PCTL[37]和 CSL[24]。

2.3.1 概率计算树逻辑（PCTL）

概率计算树逻辑由 Hansson 和 Jonsson 最先提出，它是一个对计算树逻辑（CTL）中添加离散时间和概率性问题的扩展。例如：PCTL 表达式可以描述“系统通过状态集合到达一个指定的步数的目标状态的最大（最小）概率是多少”。PCTL 能够在 DTMC 中描述状态公式属性和路径公式属性。

定义 2.3（PCTL 语法）$p \in [0,1], k_i \in N, \theta \in \{>, <, \geq, \leq\}$。在原子命题集合 AP 中，PCTL 公式的语法有如下几种：

（1） 在状态公式时，为真；

（2） 每个原子命题（$a \in AP$）都是一个状态公式；

（3） 如果 ψ 和 φ 是状态公式，那么¬ψ 和 ψ∧φ 也是状态公式；

（4） 如果 ψ 和 φ 是状态公式，那么 Xψ，ψ∪φ 和 $\psi \cup^{[k_1,k_2]} \varphi$ 是路径公式；

（5） 如果 ω 是路径公式，那么 $P_{\theta p}(\omega)$是状态公式。

布尔操作符¬和∩还有其他的意义，它们能被用来延伸操作符⇒和∪。路径公式中包含操作符 X（下一个），操作符∪（无界）和$\cup^{[k_1,k_2]}$（有界）等。前两个与 CTL 中对应的操作符是意义的，公式 $\psi \cup^{[k_1,k_2]} \phi$ 表示：在 $k' \in [k_1,k_2]$时，φ 为真，当然在所有 k’之前的状态也成立，在 k’之后的状态中，ψ 为真。

我们用⊨M 来表示 PCTL 公式成立，对于 DTMC 模型 M 中的状态 s 和路径 ω，则有 s ⊨Mφ，表示公式 φ 在 DTMC 模型中的状态 s 下为真；路径公式也如此。

定义 2.4（PCTL 语义）：$p \in [0,1]$, $k_i \in N, \theta \in \{>,<,\geq,\leq\}$，$s_i$ 表示路径 ω 的第 i 个状态，关系⊨M，状态 s 和路径 ω，以及 DTMC 模型 $M = (S, \bar{s}, P, L)$ 定义如下：

（1） s ⊨M true　所有状态；

（2） s ⊨M a　当且仅当 a 是在状态 s 下满足的原子命题，a∈Lable(s)；

（3） s ⊨M ¬φ 当且仅当 s ⊭M φ；

（4） s ⊨M ψ∧φ 当且仅当 s ⊨M ψ∧s ⊨Mφ；

（5） s ⊨M $P_{\theta p}$ (ψ) 当且仅当 $Pr\{\omega \in Path^M(s) | \omega \vDash_M \psi\} \theta p$；

（6） s ⊨M Xψ 当且仅当 ω(1)⊨M ψ；

（7） $\omega \vDash_M \psi \cup^{[k_1,k_2]} \varphi$　当且仅当 $k' \in [k_1,k_2]$ $(\omega[k'] \vDash_M \varphi) \wedge (\forall k'') \in [0,k')$ $\omega[k'] \vDash_M \psi$；

（8） s ⊨M $L_{\theta p}$ [ψ] 当且仅当 $\lim_{k\to\infty} Pr\{\omega \in Path^M(s) | \omega \vDash_M \psi\}$。

公式 $Pr\{\omega \in Path^M(s) | \omega \vDash_M \psi\} \theta p$ 表示从开始状态，经过所有 ω∈Path 的路径，

公式 ψ 都会在条件 θp 下成立。

现在 PCTL 语法和语义都定义完成后，现在我们就可以定义一个实例了，例如：系统在 100 步内，达到稳定的概率是否小于 0.2；对应的 PCTL 是：P<0.2[F<100 “stable”]。

2.3.2 连续随机逻辑（CSL）

连续随机逻辑是由 Aziz[21]提出的，后来 Baier[26]在此基础上进行了扩展。它是基于计算树时序逻辑（CTL[32]）和 PCTL。它能表示 CTMC 模型中的 steady-state 和 transient 的行为，它类似于 PCTL，是在 PCTL 上的延伸，但它们用于不同的模型中。PCTL 不能用于表达某个自然数区间的行为，而 CSL 可以描述。

定义 2.5 （CSL 语法[24]）：$p\in[0,1], I\subseteq\mathbb{R}_{\geq 0}, \theta\in\{>,<,\geq,\leq\}$，I 表示一个非空区间。在原子命题集合 AP 中，CSL 公式的语法有如下几种：

（1） 在状态公式时，为真；

（2） 每个原子命题（a∈AP）都是一个状态公式；

（3） 如果 ψ 和 φ 是状态公式，那么¬ψ 和 ψ∧φ 也是状态公式；

（4） 如果 ψ 是状态公式，那么 $S_{\theta p}[\psi]$也是状态公式；

（5） 如果 ω 是路径公式，那么 $P_{\theta p}(\omega)$是状态公式；

（6） 如果 ψ 和 φ 是状态公式，那么 $X^I\psi$，$\psi \cup \varphi$ 和 $\psi \cup^I \varphi$ 是路径公式。

除了 steady-state $S_{\theta p}[\psi]$以外，在 CSL 中的状态公式与 PCTL 中的没有区别，它类似于 PCTL 中的 $L_{\theta p}[\psi]$，表示在满足 θp 的情况下，将会保持在状态 φ 很长时间。路径公式 $X^I\psi$ 表示状态 ψ 在时间 I 内的某个点能生成一个迁移。公式 $\psi \cup^I \varphi$ 表示 φ 在时间 I 内的某些时间点为真，而在这些时间点的前面 ψ 为真。我们采用同样的满足关系⊨M 来定义 CSL 公式为真。对于状态 s 和路径 ω，我们有 CTMC 模型 $M = (S, \bar{s}, R, L)$

定义 2.6（CSL 语义）：$p\in[0,1], I\subseteq\mathbb{R}_{\geq 0}, \theta\in\{>,<,\geq,\leq\}$。$\omega[i]=s_i$ 表示路径 ω 中的第 i 个状态。$\delta(\omega,i)=t_i$ 表示在状态 s_i 中花费的时间，ωΘt 表示在路径 ω 中时刻 t 的状态。对于状态 s 和路径 ω，以及与 DTMC 模型类似的 CTMC 模型 M，定义如下：s ⊨M true 所有状态；

（1） s ⊨M a 当且仅当 a 是在状态 s 下满足的原子命题，a∈Label(s)；

（2） s ⊨M ¬φ 当且仅当 s ⊭M φ；

（3） s ⊨M ψ∧φ 当且仅当 s ⊨M ψ∧s ⊨Mφ；

（4） $s \vDash M\ S_{\theta p}(\psi)$ 当且仅当 $\lim_{t\to\infty} Pr_s\{\omega \in Path^M(s) | \omega \Theta t \vDash M\psi\}$；

（5） $s \vDash M\ P_{\theta p}(\psi)$ 当且仅当 $Pr\{\omega \in Path^M(s) | \omega \vDash M\psi\} \theta p$；

（6） $\omega \vDash M\ X^I\psi$ 当且仅当 $\omega[1] \vDash M\psi \wedge \delta(\omega,i) \in I$；

（7） $\omega \vDash M\ \psi \cup^I \varphi$ 当且仅当$\exists t \in I(\omega \Theta t \vDash M\varphi \wedge (\forall t' \in [0,t).\omega \Theta t' \vDash M\psi))$。

现在 CSL 语法和语义都完成了，我们就可以用它们来验证系统中模型的属性。例如：这个系统在下一个 10 年被毁掉的概率是否在 0.03 以内，对应的 CSL 属性公式是：$P \leq 0.03[true\ \cup^{\leq 10}$ “crash”]。

2.4 模型检测马尔科夫链解决方法

上一节已经讲到，当系统模型建立和系统属性公式化后，模型检测工具就可以验证属性公式是否能满足这个模型。这个验证通过逻辑表达式是否为真来表示。为了验证一个状态 s 是否满足公式 ψ，我们可以计算出一个满足公式ψ的集合 $Sat(\Phi)=\{s \in S \mid s \vDash \Phi\}$，现在我们只需要验证状态 s 是否在此集合中就可以了。这里就需要一些方法和算法了，本节主要简单地讲解在模型检测工具中用马尔科夫链的技术方法来解决可满足公式的问题。

2.4.1 数值方法和统计方法

系统中的随机属性主要有两种分析方法：数值法和统计法；其中数值法还可以继续分为数值化方法[41]和符号化方法[51]。当并行系统中生成一个状态空间，模型检测工具实际上要处理的是快速增长的状态数和迁移数。

符号化算法：采用 BDD[45,34]（二元决策图）和 MTBDD（多终端二元决策图）来实现的，它会产生状态爆炸问题。此算法可以避免重复建立状态空间中的节点和迁移。

数值化算法：采用一些技术来解决系统中包含有操作符的线性方程组；它的优点在于精确性很高，然而与符号化算法一样，它同样会产生大量状态空间，所以它需要大量的内存空间。

统计算法：它的思想是模拟和样本。它不需要像数值方法那样建立完整的状态空间模型来验证属性公式是否在每个状态中成立；它可以获得一些具有代表性样本路径作为属性公式的验证途径；如果属性公式成立在生成的样本中，那么它就能通过这些值估计出最终结果。很显然这种方法不能像数值方法那么精

确，但是它的优点在于不需要大量的内存空间，因此不会产生状态空间爆炸问题。在统计工具里面，常常需要我们选择为样本设置一个固定的数量值或者采用顺序验收抽样方法[38,35]；顺序验收抽样方法中，工具中的程序需要考虑它所产生的观测值。另外，如果我们设置一个固定的样本数量，那么当产生的样本数到达我们设置好的预期最大数目并且样本集合能够给出结果时，样本集合就会停止。

2.4.2 DTMC 模型中的 PCTL

本节主要介绍 PCTL[37]模型检测。在马尔科夫链中的 PCTL 模型可以被处理为系统中的多项式时间和公式中的限行时间。假如我们有一个 PCTL 公式 ψ，模型检测工具就能给公式 ψ 建立一个语法树，公式ψ的节点表示它的子公式，就像前面提到的一样，这将能计算出满足 Sat(ψ)的集合。这一过程是从语法树的叶子节点中计算子公式倒推得来的；例如：Sat(ψ∧φ)=Sat(ψ)∩Sat(φ)。这样就可以从叶子节点一步步往上判断真假了。下面我们简单地介绍模型检测中 PCTL 操作符，这里我们用 S 表示状态的数目，E 表示迁移的数目。

X 操作符：表示属性公式在下一个迁移中是否成立；它只需要一个矩阵向量乘积很简单；

Bounded U 操作符：通过$\cup^{\leq k}$来获得一些图解分析和 k 个矩阵向量乘积。它的最大时间复杂度为 $O(k_{max}\times(S\times E)\times|\psi|)$或者 $O(S^3\times|\psi|)$，其中 k_{max} 是最大时间参数，|ψ|表示公式的大小；

Unbounded U：此公式与上面不同的是它需要无穷多个矩阵向量乘积，我们可以用$\cup^{\leq\infty}$来表示它。不过它们的方法是相似的；首先我们需要通过 BDD 定点[34]预先计算 O(S+E)[31]的时间，假如在属性公式 $P_{\theta p}$ 中的 p 的值为 0 或者 1，就不用就再进一步计算；如果 0<p<1，这时需要用数值计算方法来解决系统中的线性方程组，线性方程组可以直接法（如：Gaussian 消除法或者 LU 分解法），还可以用迭代法（如：Jacobi 或者 Gauss-Seidel[41]）；如果系统中有大量的概率矩阵，那么用迭代法会更好。

Long Run 操作符：L 操作符表示在系统中会运行很长时间。首先，我们使用图像分析来找出所有的底部强连通组件（BSCC[40]），这要花费 O(S+E)的时间，接着每个底部强连通组件系统线性方程组被计算后，最后就可以计算每个可达 BCSS 的概率了。它的最差时间为 $O(S^3)$。

2.4.3 CTMC 模型中的 CSL

模型检测 CSL[24,21,45]和 PCTL 类似，它将递归地计算出可满足集合。对于无界时间 CSL 公式，可以采用 DTMC 的方法来解决；Baier 等人[25]证明 CSL 的有界时间公式可简化为 CTMC 中的瞬时分析（尤其是均一化）。

X 操作符：首先，我们计算集合$Sat(\psi)$和一个矩阵向量 b 乘积，这个向量可表示为：

$$b(s) = \begin{cases} e^{-E(s)\cdot t_1} - e^{-E(s)\cdot t_2}, & \text{如果} s \in Sat(\varphi) \\ 0, & \text{其它} \end{cases}$$

Bounded U 操作符：模型检测U^I需要多个矩阵向量乘积和瞬时分析。这需要 CTMC 均一化方法来计算它的概率矩阵。最差时间为 $O(E\cdot q\cdot t_2)$[24]，q 表示均一化的比率，t_2 表示区间 I 的上界。实际上，这里可能会提高一定的效率。例如，较大的 t_2 可能会被降低[46,52]。

Steady-state 操作符：可以通过用图形分析方法解决的系统线性方程组，来计算公式 $s \vDash_M S_{\theta p}[\psi]$是否满足；也就是搜索所有的底部强连通组件（BSCC），所需时间为 O(S+E)[40]。Steady-state 分析会在每个底部强连通组件中执行，然后那些就能获得可到达的 BSCC 的概率。Steady-state 最差的时间复杂度为 O(S+E)。

2.4.4 求解系统中线性方程组

上一节已经介绍了涉及到系统线性方程组的模型检测公式 $S_{\theta p}[\psi]$，这些等式可以用 Ax=b 的形式来表示。一般我们有两种方法来解决线性方程组：直接法和迭代法。

直接法：例如 Gaussian 消去法[41]，采用固定的操作次数来计算系统线性方程组。而只有当系统中的模型较小时，使用直接法会方便些（不要超过数千状态），计算工作量为 S^3，S 表示状态数目。另外，虽然这种方法具有高可信赖性，但是它必须在算法的每一步都必须修改系数矩阵，增加了系统开销。由于在矩阵中的元素被不断地修改，以至于它很难识别紧凑的储存方案，从而导致大量的内存消耗。所以我们在模型检测工具中一般采用迭代法来计算。

迭代法：此方法不需要改变矩阵的形式，因此能采用紧凑的存储方案。在迭代法中，它不需要预先知道在实现一个精确的答案中有多少计算。假如我们有一个离散时间马尔科夫链的概率矩阵：

$$P = \begin{pmatrix} 0 & 0.7 & 0.3 \\ 0.2 & 0.1 & 0.7 \\ 0 & 0.5 & 0.5 \end{pmatrix}$$

系统从状态 1 开始，初始概率向量为 $\alpha^0=(1,0,0)$。经过一次迁移后，系统到状态 2 的概率为 0.7，到状态 3 的概率为 0.3；此时系统的概率分布 $\alpha^1=(0,0.7,0.3)$, ($\alpha^1=\alpha^0\cdot P$)，系统经过两步的概率分布 $\alpha^2=\alpha^1\cdot P=(0.14,0.22,0.64)$,($\alpha^2=\alpha^1\cdot P$)，由此可得 $\alpha^k=\alpha^0\cdot P^k$。当马尔科夫链是有限的、非周期的、不可约的和遍历的。这时我们可以忽略初始向量，让 α^k 聚集到一个固定的概率向量 $\alpha=\lim_{k\to\infty}\alpha^k$。由此我们可以看出经过一定数量的迭代后，就能达到稳定状态了。而且我们不需要再用知道向量乘积里面的值。当达到平衡时，那么对于 k 次和 k-1 次就没有区别了我们可以用特征等式来表示 $\alpha\cdot P=\alpha$。这种方法可以用理论中的收敛来完成，但非常慢。一般我们采用其他的迭代方法，例如：Jacobi，Gauss-Seidel，JOR 和 SOR，它们可以用于获得解决系统线性方程组的方案，Jacobi 和 Gauss-Seidel 都不能保证收敛性，Gauss-Seidel 消耗的内存比 Jacobi 少。可以用一个松弛的方法来提高 Jacobi 和 Gauss-Seidel 的收敛比率[22,29]，它们对应的方法是 JOR（Jacobi Over-Relaxation）和 SOR（Successive Over-Relaxation）。

2.5 模型检测工具 PRISM

本节主要介绍概率模型检测工具 PRISM，其他的还有： ETMCC，MRMC，YMER，VESTA，以及它们的简要背景和模型、规格语言、实现算法和数据结构。本文主要是采用验证性能较好的 PRISM 工具来完成我们的实验结论。

PRISM 是一种概率性符号化模型检测工具，它是由英国伯明翰大学开发用于分析概率性的系统。这个工具的开发语言是 Java 和 C++，它的用户界面和语法是用 Java 写的，内核算法主要由 C++实现。PRISM[17, 18]里面采用了由美国科罗拉多大学的 Fabio Somenzi 用 C 开发的一个改进的 CUDD（CU 决策图[44]）软件包和一个 MTBDD（多终端决策图）类库。这个工具能够在 Linux、Windows、Solaris 和 Mac OS X 等操作系统上运行，PRISM 工具提供了两种用户界面类型：命令行和图形化用户界面（GUI）；在 GUI 里面提供了文本编辑器，属性编辑框架和实验性能快等。

在 PRISM 中，系统模型被描述成特定的、高层的基于状态的描述语言。在

这种语言的描述下，系统模型被表达成一个平行的模块集合。每个模块由一个有限变量集合和由 guard 命令定义的状态迁移行为。模块之间通过全局变量或者共同标签来交换信息[49]。PRISM 除了支持自己的描述语言外，还能支持 PEPA[42]（随机进程代数）集合，以及基于模型检测时间自动机的非确定性，实时时钟（Digital Clocks[50]和 KRONOS[33]）。PRISM 能够输出被 ETMCC 和 MRMC 支持的形式语言。它主要支持的概率模型有 3 种：DTMC（离散时间马尔科夫链），CTMC（连续时间马尔科夫链），MDP（马尔科夫决策过程）。

PRISM 属性规约：在 DMTC 和 MDP 中，属性规约用 PCTL 表示；在 CMTC 中，属性规约用 CSL 表示。这些规约可表示成满足一个给定边界的概率值是否正确或者输出一个真实的概率值；同时它还支持基于 costs 和 rewards 模块提供规约属性分析。表 2.1 表示 PRISM 支持的 PCTL 和 CSL 的集合。

时序逻辑	PCTL	CSL
Basic	√	√
$L_{\theta p}[\psi]$		
$S_{\theta p}[\psi]$		√
$P_{\theta p}[\psi \cup \varphi]$	√	√
$P_{\theta p}[\psi \cup^{\leq k} \varphi]$		√
$P_{\theta p}[\psi \cup^{\geq k} \varphi]$		√
$P_{\theta p}[\psi \cup^{[k1,k2]} \varphi]$	√	√
$P_{\theta p}[X^{\leq k} \varphi]$		
$P_{\theta p}[X^{\leq [k1,k2]} \varphi]$		

表 2.1　PRISM 支持的 PCTL 和 CSL（$p \in [0,1], \theta \in \{>,<,\geq,\leq\}$）

PRISM 算法和数据结构：在 PRISM 中，可以选择不同的模型对属性规约 PCTL[37,23,27,28]和 CSL[45,25]进行验证；它们中所采用的迭代法有：Gauss-Seidel，Jacobi，JOR，Power，SOR。它们用于解决系统中的线性方程组，操作符 S 和U 稳定时所需要的时间。PRISM 还提供了如下三种模型检测数据结构：

（1）MTBDD(Mulit-Terminal Binary Decision Dagram)：一个扩展 BDD 的多终端二分决策图[36,34]；

（2）Sparse matrix：稀疏矩阵[54,30]；

（3）Hybrid：结合 MTBDD 和 Sparse matrix 的结合体。

系统引擎都执行相同的计算，所以选择以上三种都没有任何区别；但是它们

产生的时间性能和空间性能不同，Sparse 引擎要快于 MTBDD，但是需要更多的内存空间；而 Hybrid 正好需要少于 Sparse 的内存，同时要快于 MTBDD；所以一般情况下，我们都采用 Hybrid 引擎工作机制。

2.6 本章小结

本章主要介绍了概率模型检测的相关知识，首先从概率模型检测中的主要模型 DTMC 和 CTMC 展开，然后介绍模型中的规格化属性公式，用于表达系统中某些性质的行为，然后通过建立的模型来验证它们；接着讲解模型检测马尔科夫链，其中包括 PCTL 在 DTMC 中的应用和 CSL 在 CTMC 中的应用，以及解决模型中的线性方程组问题，采用的方法有直接法和迭代法，迭代法包括 Jacobi 和 Gauss-Seidel，以及 JOR 和 SOR 等。最后我们介绍了主要的模型检测工具，PRISM，其他的工具还有 ETMCC，MRMC，YMER 和 VESTA。前三种主要用于数字化分析，后两种用于统计分析。在 PRISM 模型中支持的模型有 DTMC，CTMC 和 MDP。PRISM 中的数据结构有 Sparse，MTBDD 和混合类型；其他四种均只有一种结构。在解决方法中，PRISM 使用的最多，其它相对较少。在上述各个工具中，PRISM 工具总体上要优于其它四种工具，关于后面实验验证，我们主要采用 PRISM 工具来验证本文中要验证的各类分布式算法。

第 3 章　Flatebo 算法建模验证和分析

3.1 引言

分布式中的自稳定算法最早是由 Dijkstra 提出的，它的思想是系统中的进程运行到某个时刻出现故障，这些进程能够在不需要外接的协助下，重新恢复正常。例如在一个班上课的学生，下课后他们之间可能有学生换位置了，但是他们都能正常听课（无空位，无逃课）。这个系统中，下课后，系统就出现故障了，当上课铃响了，系统又恢复正常，只是学生的位置可以发生了变化，但不影响他们的学习。

1974 年，Dijkstra 提出自稳定算法 [6]，用于解决分布式环境下的错容问题。自稳定算法是指从一个任意的初始状态开始，都能通过有限步而达到一个稳定的状态。Dijkstra 给出了一个自稳定系统的例子，该系统中共有 N 个进程，当某个进程拥有特权时，便可以改变自己的状态；随后，人们尝试着采用形式化的方法来验证自稳定算法的性能，Arora 和 Gouda[8]做了最早实验，但没有完全形式化地分析出来。在这一时期里，Herman 和 Flatebo 采用概率型模型实现了 2 状态的自稳定算法。本章节主要是研究 Flatebo 算法，接着用线性回归拟合分析它的近似复杂度，并把它与其它同类算法比较。最后还给出一个研究自稳定算法的方法论。

在 Flatebo 算法[2]中，第 0 个进程执行一个特定的规则，当所有的进程的局部状态相同时（即 00…0 或 11…1），进程 0 获得特权；其它的 N-1 个进程执行另一类规则，当前进程的局部状态与它的前一进程的局部状态不同时（即…01…或…10…），当前进程获得特权；因此，整个系统中，最多只有 N-1 个特权。Flatebo 算法与 Dijkstra 算法不同的是，每个进程只有 2 个状态，同时修改第 0 个进程的规则，而提出自己的解决方法。当然自稳定算法必须满足以下的四个条件：无死锁，无饥饿，闭合性和可达性。

Flatebo 已经从理论上证明了上述 4 个条件，这里我们将使用概率模型检测工具 PRISM[17, 18]来分析 Flatebo 算法[2]，并且给出了一种形式化的描述来构建和分析由此建立的有限状态概率模型。我们通过设置进程的个数来验证了 Flatebo 算法最大稳定时间，但发现出现最大稳定时间的可能性基本可以忽略，意义不

大；后面我们将从理论和实验方面验证这个结论。

而实际中我们一般考虑的是预期中的最差时间。与 Herman 算法（最大上限 $0.64N^{2}$[1, 7]，N 为奇数）不同的是，Flatebo 采用的是异步移动，同时进程可以为奇数或偶数；不过，它们建立的模型都是单向移动环。接着，我们用线性回归对预期最差时间进行了分析，并得出了近似的预期最差时间。

本章结构如下：第 3.2 主要讲解自稳定算法定义，特点和性质，第 3.3 节主要讲 Flatebo 算法[2]，第 3.4 节给出了实验建模和结果，以及采用线性回归分析过程，第 3.5 节是本章小结。

3.2 自稳定算法

3.2.1 自稳定性定义

我们用谓词 P 表示系统 S 稳定性的集合，可以 P 来识别系统是否正常工作。如果系统满足 P，则系统处于合法状态，如果不满足，则处于非法状态。系统 S 关于谓词 P 和 R 是稳定的，它必须满足两个性质：

（1）闭包：如果在系统 S 上建立谓词 P，那么 P 就不能被修改；

（2）收敛：从任何一个满足 R 的状态开始，经过有限次状态迁移，系统 S 能够保证达到满足 P 的全局状态。

3.2.2 自稳定算法的特点和性质

分布式自稳定系统通常由很多独立的单元组成，如果要很好地描述这些单元，我们需要解决一些问题：（1）、系统中的状态数目，（2）、系统中的一致性和非一致性，（3）、中央进程和它的从属进程，（4）、令牌环中的状态数目，（5）、共享内存模型，（6）、自稳定性的成本。

首先，在分布式自稳定系统中，它的合法性必须包括如下：

（1）无死锁：系统中至少有一个特权进程；

（2）无饥饿：在无限多次运行中，系统中的每个进程都能无限多次的获得特权；

（3）闭合性：系统到达一个稳定状态后，下一步的任何一次移动都将达到另一个稳定状态；

（4）可达性：对于任意的两个合法状态，系统总可以从其中某一状态经过

有限次移动而进入另一合法状态。

（5）而在 Dijkstra 提到的令牌环中，当一个进程获得一个特权时，表示此进程获得一个临界资源，当进程放弃特权时，表示它离开临界资源，这样我们就可以很方便地描述它的行为。

上面讲到了自稳定的特点和性质，现在我们来简单地描述一个自稳定算法，例如系统中病毒后，可能会有一些软件被毁坏，导致系统被破坏，自稳定算法能够使系统恢复。而此处，就是我们使用那种自稳定算法更合适我们的系统所出现的故障。因此，我们就需要知道各个自稳定算法的性质和性能。这样会有利于我们对自稳定算法的选择。

3.3 Flatebo 算法简介

假设在一个网络拓扑中，有 N 个不同的进程组成的环，从 0 到 N-1，协议以异步方式运行，各个进程可以拥有特权，并且特权绕环单向移动或消失。在算法中，每次都会随机地选择其中一个特权进程执行，随着特权数逐渐减少到为 1 后，将保持不变，随后每个进程逆时针依次获得特权，达到 N-1 时，再次从头循环。

在算法中，我们用 s_i 表示第 i 个进程的状态，用 s_{i-1} 表示进程 i 的左邻居的状态，算法每次通过检查进程 i 的状态是否不等于左邻居（$s_i!=s_{i-1}$），来判断进程 i 是否获得特权，如果进程 i 的状态与左进程状态不同，则需修改自己的状态（即获得特权）。 而进程 0 跟其他的 N-1 个进程获得特权的规则不同，当 $\bigcap_1^{N-1} S_i = S_0$，第 0 个进程获得特权。

如图 3.1 所示，一共展示了 5 个进程的五次移动直到稳定状态，我们用 s_i 和 s_i`分别表示 1 和 0。拥有特权的进程用灰色表示。最开始，进程 0 没有特权，进程 1，2，3，4 有特权，此时我们可以定义进程 0 的状态为 1，则进程 1，2，3，4 的状态分别为 0，1，0，1；当算法执行下一步时，我们随机地选择特权进程中的进程 3，此时进程 3 换位为 1，那么当前的特权进程就只有进程 1，2；接着算法再从进程 1、2 中随机选择进程 2，则此时的进程 2 的特权移动到进程 3；然后，算法再从进程 1 和进程 3 中随机选择进程 3，此时特权从进程 3 移动到进程 4。算法由此执行下去，最后只剩下一个特权进程（稳定状态）；当然图中所展示的只是其中一条迁移路径（10101->10111->10011->10001->11001->11000）。

如果当系统选择了其它含有特权的结点，那么将还会出现更多的迁移路径，这里只做一个路径来描述算法运行的机制。另外，从图中我们也可以看出特权都是逆时针移动的（或者消失）。

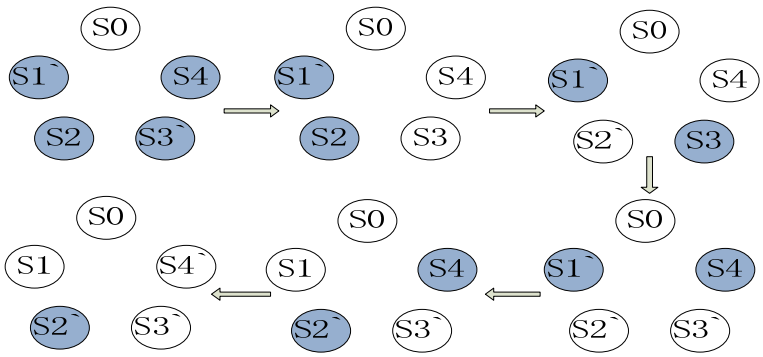


图 3.1　上图展示了 5 个进程的 Flatebo 环，s_i 和 s_i`分别表示 1，0。拥有特权的进程用灰色表示

3．4 Flatebo 算法的概率验证

现在，我们采用 PRISM[17, 18]来描述 Flatebo 算法的建模，然后，我们通过验证各种各样的属性来分析算法的正确性（闭合性、可达性、无死锁和无饥饿）和它的其它性能，其中包括 Flatebo 提出的最坏情况下，所需 $0.5N^2-0.5N-2$ 的验证和预期最差时间。

第 2 章已经简单地介绍了 PRISM 工具和概率模型，及其概率模型时序逻辑的相关知识，上面的介绍中我们可以知道 PRISM 工具可以验证系统中的一些属性，例如：“系统在 4 个小时内失败的概率是多少”、“半个小时后，消息的发送数量是多少”、“算法终止的最大预期概率是多少”……。

我们可以采用前面给出的时序逻辑公式完成我们的工作，如： $P_{<0.2}[F_{<100}$ “stable”]表示在 100 步内，系统稳定的概率是否在 0.2 以内；然而对于 MDP 模型，我们需要对它们稍作修改，例如： $Rmax_{=?}$ [F “terminate”]表示算法终止的最大时间是多少。以上这些概率逻辑表达式就可以很好的描述我们算法中的属性行为了。

3.4.1 Flatebo 算法建模

现在我们把 Flatebo 算法建立成一个 DTMC 模型。这里我使用 $M = (S, \bar{s}, P, L)$：

S：$\{0,1\}^N$ 每个状态都是一个矢量$\vec{s}^k = (s_0, s_1, s_2, \cdots, s_{N-1})$ $s^k=(s_0,s_1,s_2,\ldots,s_{N-1})$，第 i-1 位元素表示第 i 个进程，k 表示此状态下特权进程数;

$\bar{s}$：系统的初始状态集合;

P：$S\times S\rightarrow[0,1]$，如对于任意的两个可达状态 $s_1^{k1}=(s_0,s_1,s_2,\ldots,s_{N-1})$ 和 $s_2^{k2}=(s`_0,s`_1,s`_2,\ldots,s`_{N-1})$（k1,k2 分别表示状态 s_1,s_2 的特权数），我们有 $P(s_1^{k1},s_2^{k2})=1/k1$;

L：原子命题集合 AP={ $token_0,token_1,token_2,\ldots,token_{N-1}$, stable}（$token_i$ 表示进程 i 获得特权,并且最多只有 N-1 个特权），如果 $s_1^{k1}(i-1)\neq s_1^{k1}(i)$ ($s_1^{k1}(i)$表示 s_1^{k1} 的第 i 个元素)， $token_i \in L(s_i^k)$；另外，如果 $token_0,token_1,token_2,\ldots,token_{N-1}$ 中有且只有一个在 $L(s_i^k)$中，那么 stable$\in L(s_i^k)$。

对于我们之前提到的图 3.1 中的 DTMC 迁移状态其中的一部分，如：(10101)→(10111)和(11001)→(11000)：

$P_1((11001),(11000))=1/4$

$P_2((11001),(11000))=1/2$

上面 P_1 后面第二项中的 1/4，表示在 10101 状态下，有 4 个特权进程，所以第 3 个进程发生的概率为 1/4。接下来，我们采用模型检测。PRISM[17, 18]的基本组成元素是有一些模块和变量组成的。整个模型由模块之间的相互作用而组成；一个模块中又包含多个局部变量，整个模型的全局变量是由各个模块中的局部变量组成的。每个模块的行为都有一个命令集合来描述。如下：

[] guard -> prob_1 : update_1 + ... + prob_n : update_n;

guard 基于整个模型上所有变量的谓语（包括其它模块的变量）。当 guard 为真时，update 表示此模块能发生的迁移。一个迁移表示给次模块中的某些变量赋于新的值，也可能是作为其他变量的函数；prob 表示各个 update 发生的概率值（或者在某种情况下的速度）。

Flatebo 算法的 PRISM[17, 18]模型（共有 7 个进程如图 3.2）。每个进程由一个模块表示，每个模块都包含一个变量表示模块的当前状态。前面讲到，进程 0 获得特权的条件是所有进程的状态都相同（即 $s_0=s_1\&\cdots\&s_0=s_6$）；进程 1 获得特权的条件是与前者不同（即 $s_0\neq s_1$），随后的进程都与进程 1 的结构相同。另外，因为算法是异步的，所以特权进程发生迁移的概率都是均等的。这就是前面讲到的每个进程的迁移概率矩阵 P。为了计算满足属性的预期时间，我们使用了一个 reward 结构，并且需要设置 reward 的值为 1。同时我们也可以用 init……endinit

来配置系统初始状态。在我们模型中还将添加计算在某一状态下特权数目的公式和标签。以下就是我们设计的模型（共有 7 个进程）。

```
1   dtmc
2
3   module p0
4       s0:[0..1];
5       [step] (s0=s1)&(s0=s2)&(s0=s3)&(s0=s4)-> 1:(s0'=1-s0);
6   endmodule
7
8   module p1
9       s1:[0..1];
10      [step] (!s1=s0) -> 1:(s1'=s0);
11  endmodule
12
13  module p2=p1[s1=s2,s0=s1]endmodule
14  ……
15  module p6=p1[s1=s6,s0=s5]endmodule
16
17  rewards "steps"
18      true :1;
19  endrewards
20
21  formula token_stable = (s0<=s1&s1<=s2&s2<=s3&s3<=s4&s4<=s5&s5<=s6)
22                          |(s0>=s1&s1>=s2&s2>=s3&s3>=s4&s4>=s5&s5>=s6);
23
24  label "stable" = token_stable;
25
26  init   true endinit
27
28  formula tokens =(s0=s1?0:1)+(s1=s2?0:1)+(s2=s3?0:1)
29                      +(s4=s3?0:1)+(s4=s5?0:1)+(s5=s6?0:1);
```

图 3.2　PRISM 代码（共有 7 个进程）

进程数（N）	状态数（states）	节点数（nodes）	迁移数（transtions）
4	16	57	26
6	64	145	162
8	256	273	898
10	1024	441	4610
12	4096	649	22530

14	16384	897	106498
16	65536	1185	491522
18	262144	1513	2228226
19	524288	1692	4718594
20	1048576	1881	9961474

表3.1　进程数不同下，分别生成的状态数、节点数和迁移数

图3.2中，第3行到第6行表示进程0的迁移条件行为，第8行到第11行表示进程1的迁移条件行为，第13行到第15行表示进程2到进程6的迁移条件行为；第17行到第19行表示模型每次执行一步的时间花费（reward指的是所花费的时间，1表示一步）；第21行到第22行表示系统稳定的条件（即00…01…11或11…10…00）；最后两行表示系统中的特权数（两相邻进程状态不同时，后者获得特权）。表3.1采用PRISM工具计算出各个进程数下的状态数、节点数和迁移数；并且能快速得出此结果。

3.4.2　Flatebo 算法正确性验证

前面我们提到了自稳定算法应该满足的4个属性：（1）无死锁；（2）无饥饿；（3）闭合性；（4）可达性。

无死锁：图2中，无论系统处于什么状态，特权进程数至少大于或等于1，即tokens>=1。对应的概率性属性表示：

P>=1 [F tokens>=1]

无饥饿：经过无限多次运行，所有进程都将会无限多次地享有进程。针对本文中给出的算法，进程1、2……6可以互换，所以只要验证其中一个即可；另外，进程0比较特殊，需另外验证。对应的属性表示如下：

P>=1 [G F (s0=0&s1=1&s2=1&s3=1&s4=1&s5=1&s6=1)|
(s0=1&s1=0&s2=0&s3=0&s4=0&s5=0&s6=0)]

(进程1获得特权过程)

P>=1 [G F (s0=0&s1=0&s2=0&s3=0&s4=0&s5=0&s6=0)|
(s0=1&s1=1&s2=1&s3=1&s4=1&s5=1&s6=1)]

(进程0获得特权过程)

闭合性：从某个合法状态开始的每次迁移，都将进入另一个合法状态。由于我们的算法是单向移动的（即特权移动方向s0->s1->…->s6->s0），所以，进程1

到进程6的移动是相同的，而进程6到进程0或者进程0到进程1就要另作验证。相应的属性表示如下：

P>=1 [(s0=0&s1=1&s2=1&s3=1&s4=1&s5=1&s6=1) =>
(s0=0&s1=0&s2=1&s3=1&
s4=1&s5=1&s6=1)|(s0=1&s1=0&s2=0&s3=0&s4=0&s5=0&s6=0)
=>(s0=1&s1=1&s2=0 &s3=0&s4=0&s5=0&s6=0)]

(进程1到进程2特权迁移过程)

P>=1 [(s0=0&s1=0&s2=0&s3=0&s4=0&s5=0&s6=1) =>
(s0=0&s1=0&s2=0&s3=0&
s4=0&s5=0&s6=0)|(s0=1&s1=1&s2=1&s3=1&s4=1&s5=1&s6=0) =>
(s0=1&s1=1&s2=1 &s3=1&s4=1&s5=1&s6=1)]

(进程6到进程0特权迁移过程)

可达性：从某一合法状态，经过有限多次将进入另一合法状态。在这里我们假设当前合法状态是进程2获得特权，另一合法状态是进程4获得特权。属性表示如下：

P>=1[(s0=1&s1=1&s2=0&s3=0&s4=0&s5=0&s6=0)
=>(s0=1&s1=1&s2=1&s3=1&
s4=0&s5=0&s6=0)|(s0=0&s1=0&s2=1&s3=1&s4=1&s5=1&s6=1) =>
0=0&s1=0&s2=0& s3=0&s4=1&s5=1&s6=1)]

另外，我们还需要验证算法是否能从初始状态达到合法状态，所以在算法中，我们添加了函数token_stable标签stable来表示系统已经达到合法状态。属性表示如下：

P>=1 [F stable]

由此，我们就证明了算法的正确性。

3.4.3 预期最差时间和最大稳定时间

最大稳定时间[3]：Flatebo 已经证明最大稳定时间从最坏情况（即系统中有 N-1 个特权进程）到达稳定状态所需要的最多时间。但由于这种情况发生的几率非常小；因此意义不大。例如当 N=7 时，最大稳定时间为 19，我们用 filter(print, P=? [F<=18 "stable"])计算出最坏情况下，前 18 单位时间内到达的概率为 0.9999922573748967（状态为 0101010 或 1010101），所以最大稳定时间出现的

概率大约为 1-0.9999922573748967 ≈ 0.000008。通过实验我们得出当进程数目增加时，此概率会变得更小。由此我们可以看出最大稳定时间意义不大。因此，我们引入了预期最差时间。

预期最差时间：假设当前特权数为 k，当系统达到稳定状态时，所需要的最大平均期望时间；由于在 N 个进程中，特权数 k 可为 1 到 N-1，所以在此，我们取这些特权数达到稳定时间的最大平均期望时间为预期最差时间。

现在，我们主要对算法的预期最差时间和最大稳定时间进行实验和分析。简单来讲，就是从初始状态到达稳定状态需要多少次迁移。例如：从初始状态到达稳定状态所需时间的属性验证：

R=? [F “stable”]

当然，如果我需要得出预期最差时间，就得添加一个过滤器 fitler；过滤器的属性验证形式为 filter(op,prop，states)，op 表示操作符，prop 是一个 PRISM[17, 18] 属性（例如 R=? [F “stable”]），states 是一个 PRISM 属性布尔表达式（例如 init…endinit）不过虽然添加了这个过滤器，但实际上当前的初始状态还是在前者中，所以这个验证所需要的时间也不会高于前者，此时的验证属性为：

fitler（max,R=? [F “stable”]， “init”）

上述属性表示，假设“init”集合中有 m 个不同状态，那么从这 m 个不同状态开始到“stable”状态将有 m 个平均预期时间，而我们要找的就是这 m 个预期时间中最大的那个。

不过对于本文提到的算法，主要考虑的是当进程数不变时，从某一最坏的非法状态到达合法状态。由于非法状态数目不好确定，同时初始状态 init…endinit 里也不方便设置，所以我们添加了公式 tokens（图 3.2）和 pctl 代码中不变量 k 来确定最坏情况下的实验结果。以下是对应的属性：

fitler（max,R=? [F “stable”]， “tokens”）

跟前者相同，只是“init”变成“tokens”；tokens 表示当前的特权数目，我们可以通过每种情况下的不变量 k 来改变它的值。

以下通过上面的规格来验证在最大特权数相同的情况下，Flatebo 算法和 Herman 算法的区别：

Flatebo algorithm（异步、进程个数>=3（整数））				Herman algorithm（同步、进程个数>=3（奇数））			
最大特权数	进程数	建立混合	预期最差时间	最大特权数	进程数	建立混合	预期最差时间

		MTBDD 所需内存（KB）				MTBDD 所需内存（KB）	
9	10	9.4	10.292	9	9	9.3	8.921
11	12	14.1	14.122	11	11	14.5	13.206
13	14	19.7	18.246	13	13	20.9	18.346
15	16	26.3	22.702	15	15	28.4	24.342
17	18	33.9	27.562	17	17	37.0	31.195

表 3.2　当 k=9 11 13 15 17 时，Flatebo 算法和 Herman 算法的区别

从表 3.2 中，我们可以看出当特权数 k 增大时，Flatebo 算法建立混合 MTBDD 所需的空间要小于 Herman 算法，特别是预期最差时间，在特权数为 13 时，Flatebo 算法需要的时间还较长，但随着特权数目的提高时，Flatebo 算法的预期最差时间就变得更小了，这样 Flatebo 算法比 Herman 算法高效。它们的不同之处是，Flatebo 必须是运行在异步系统中，而 Herman 算法运行于同步系统中。

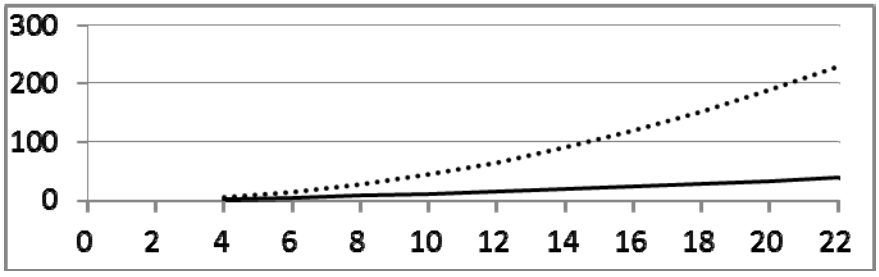


图 3.3　虚线表示达到最大稳定时间（即 $0.5N^2$-0.5N-2），实线表示预期最差时间

图 3.3 中的实线曲线就是我们通过给出的实验结果描述的预期最差时间。另外，通过简单地修改 DTMC 模型，而建立相应的 MDP 模型，采用相应的属性验证公式，例如：*Rmax=? [F "stable" {" tokens "}{max}]*(当特权数为 tokens 时，系统达到稳定所需要的最大时间)，这里的实验结果也验证了 Flatebo[3]的结论 $0.5N^2$-0.5N-2。通过实验我们还可以得出，当进程数为 4，6，8，10，12，14，16，18，20 时，预期最差时间下的 tokens 为 2，3，5，5，7，9，10，12，14。同时我们得出在各个进程数时的不同最大稳定时间，也就是图中描绘的实线。

3.4.4 算法结论分析

接下来，我们主要从理论和实验结果两方面对预期最差时间进行分析。首先，我们需要了解算法的执行过程；然后根据 Flatebo 给出的 $0.5N^2$-0.5N-2 和实验结

果来证明预期最差时间的时间复杂度。

这里我们分析共有 5 个进程的预期最差时间。上节已经给出最大预期时间必须是 2 个 tokens，而在实验中，我们又发现当序列为 10011（或 01100）时，预期时间最大。如图 3.4 中所示，首先我们可以从状态 10011 开始，它可以分别产生两个状态 11011 和 10001（由于异步迁移，两个状态的概率为 1/2），而 11011 又可以迁移到另外两个状态 11111（稳定状态）和 11001，就这样一直迁移下去，直到最后所有的可能都到达稳定状态；现在我们从树状的终端结点开始，反向推导，最后的 11101 到 11111 和 11100 的概率分别为 1/2，那么 11101 的预期时间为 1/2+1/2=1，依次到达树的初始状态，便可得出 10011 到达稳定的期望时间为 2.75。这个状态迁移图中，展现了可能出现的所有路径和状态。

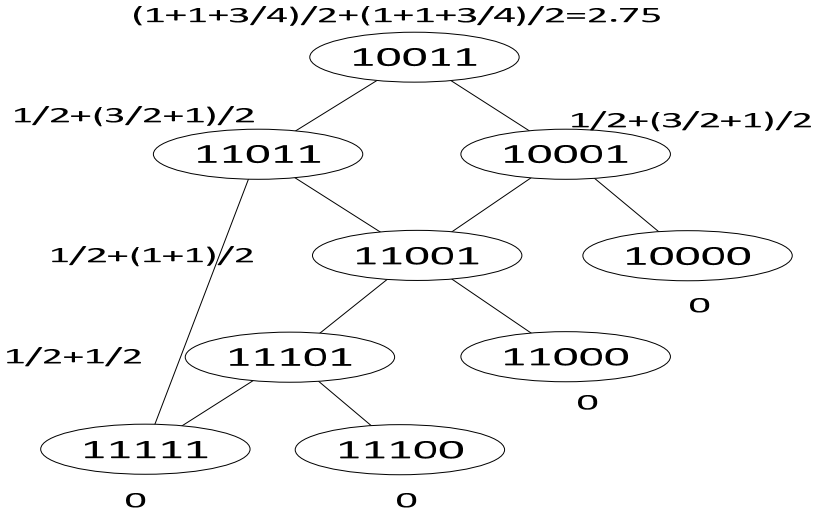


图 3.4　状态 10011 达到稳定的分析图

另外，我们在试验中，我们得出了各个进程数下的预期最差时间。前面我们已经提到最差时间为 $0.5N^2-0.5N-2$。又因为，预期最差时间不可能超过最大稳定期时间，所以预期最差时间不可能超过 $O(N^2)$。这里我们采用 excel 对实验结果进行回归分析，分别设置 N 的最高项为 N^2 、$N^{1.55}$ 、$N^{1.5}$ 、$N^{1.47}$ 、$N^{1.45}$ 、$N^{1.43}$ 得出相应的残差(估计值与实际值的差)图像。

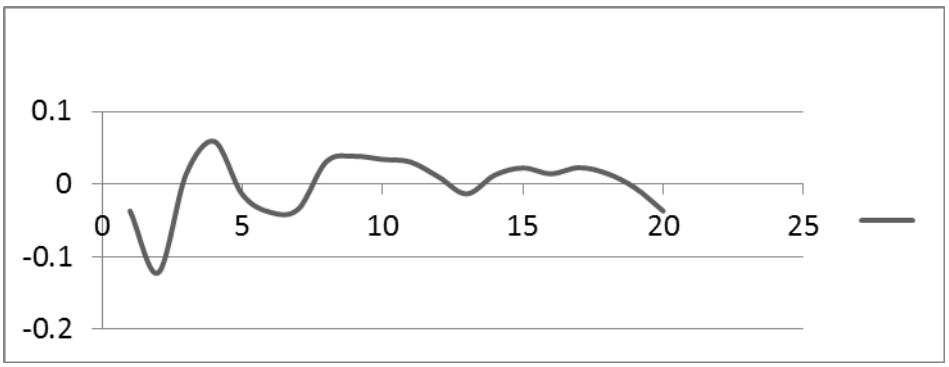


图 3.5　残差分析图

进程数	预期差时间	残差 N^2	残差 $N^{1.55}$	残差 $N^{1.5}$	残差 $N^{1.47}$	残差 $N^{1.45}$	残差 $N^{1.43}$
4	1.75	-0.19713	-0.04421	-0.0394	-0.03773	-0.03696	-0.03672
5	2.75	-0.36872	-0.15431	-0.13764	-0.12832	-0.12224	-0.1165
6	4.123	-0.26497	-0.03067	-0.00817	0.005012	0.013808	0.022412
7	5.542	-0.21288	0.010755	0.034766	0.049121	0.058789	0.068366
8	6.969	-0.25047	-0.05939	-0.03693	-0.02332	-0.01409	-0.00487
9	8.561	-0.22071	-0.07709	-0.05832	-0.04678	-0.03888	-0.03094
10	10.292	-0.14963	-0.06253	-0.04881	-0.04019	-0.03422	-0.02816
11	12.187	-0.01221	0.014251	0.02216	0.027365	0.031072	0.034902
12	14.122	0.067542	0.033535	0.035372	0.03698	0.038291	0.039752
13	16.139	0.131629	0.041073	0.036994	0.035077	0.034028	0.033148
14	18.246	0.18805	0.048186	0.038705	0.033546	0.030316	0.027264
15	20.422	0.215805	0.036843	0.02278	0.014843	0.009731	0.004798
16	22.678	0.225894	0.020722	0.00316	-0.00693	-0.01353	-0.01994
17	25.063	0.267317	0.051251	0.031505	0.020013	0.012434	0.00501
18	27.509	0.272074	0.062649	0.042237	0.030223	0.022227	0.014359
19	30.012	0.236165	0.052952	0.033572	0.022018	0.014245	0.006559
20	32.604	0.19159	0.056039	0.03955	0.02953	0.022679	0.015863
21	35.25	0.103349	0.038649	0.027055	0.019726	0.014551	0.009343
22	37.953	-0.02556	0.005402	0.000833	-0.00257	-0.00527	-0.00808
23	40.711	-0.19713	-0.04419	-0.03949	-0.03767	-0.03705	-0.03664

表 3.3　不同进程数下，不同首项的残差数据

从表 3.3 中，我们可以看出中 $O(N^{1.45})$和 $O(N^{1.47})$的残差最小，由此，我们可以近似得出预期最差时间的时间复杂度在 $O(N^{1.45})$~$O(N^{1.47})$之间。从图 3.5 我们

可以得出当进程数增加时，我们通过函数拟合的残差成衰减状态变小。另外，我们可以从表 3.3 中的值大致看出残差相对较小的范围在表 3.3 的倒数第二列和倒数第三列，即预期最差时间的时间复杂度在$O(N^{1.45})\sim O(N^{1.47})$之间。

3.5 本章小结

这里我们用概率模型检测验证了一个简单的、单向的、异步执行的随机自稳定算法（Flatebo[2]），并且对 Flatebo 提出的基本属性和 Flatebo[3]提出的最大稳定时间进行了验证和分析；另外，更重要的是我们用实验计算出了预期最差时间；同时采用线性回归近似得出预期最差时间在（$O(N^{1.43})\sim O(N^{1.47})$）之间，比 Herman 算法要高效，但它们的运行方式不同。当然，实验中的结果也遇到了状态空间爆炸问题。接下来的进一步工作，我们将改进算法，降低算法所产生的状态数和迁移数；另外，我们所得出的预期最差时间也只是从实验而来的，所以，我们还得从理论方面证明预期最差时间的时间复杂度。

自稳定算法研究方法：在自稳定算法中，我们首先从实际生产的角度给出算法相关属性，然后建立模型并验证，接着根据实验得出的结果，用合适的方法（线性回归）来证明实验结果的效率。

如果在实际应用中，故障状态达到稳定状态的时间较长，我们还可以通过 PRISM 工具对当前状态建立模型，可以很方便地获取最近路径，在这种情况下，如果时间能分配得当，就可以优化自稳定算法的恢复速度了。对此我们发现，借助 PRISM 概率模型检测工具和一些额外的资源开销，就可以很好提高系统运行效率；这也验证了一个非常高效的自稳定算法。当它并不一定绝对满足我们的实际应用时，我们可以通过额外的系统开销来满足我们系统中的需求。这里还可以看出，在引入概率模型检测技术的情况下，比较各类自稳定算法的依据不一定是预期最差时间了，可能会变成预期最快时间，只有 PRISM 能够找到最短路径，让系统执行这条路径；不过这样产生的总开销是否合理的，还需要我们的进一步验证和分析。

第 4 章　Kerry 算法建模验证和分析及其改进算法

4.1 引言

分布式算法中的互斥问题不仅非常重要，而且也很难设计。共享资源的互斥访问时分布式并发计算的一个本质问题，因此操作系统等安全攸关的分布式并发系统的一个核心问题时如何设计实现用于仲裁和控制共享资源并发访问的分布式互斥算法。在互斥问题中，多个进程访问的工具资源叫做临界资源。它可以定义为：在一组并发的进程中，同时竞争某个临界资源，一次只能有一个进程访问临界资源。

1981 年，Ricart 和 Agrawala 提出了分布式互斥算法，但从实际的角度上看，它很可能满足不了我们的实际需求，假如系统需要进程能够快速地访问临界资源，这时我们可以通过添加临界资源的数量来满足我们的需求。所以后面的 Kerry 提出了分布式 k 互斥算法，随后几年，这种算法被不断地改进，直到 1995 年，Shailaja 提出了一个全新的算法，它的性能要优于以往的 k 互斥算法。在这个算法中，它可以保证发送消息时间和访问临界区资源的时间最小。但它需要添加额外的数据结构来完成这样的任务；但此算法中，当某个进程退出临界区时，必须保证有另外一个进程进入临界区；也就是临界资源区必须被进程全部占有，而不能在某段时间内有空余的临界资源；另外，这个算法还需要额外的数据结构来存储各个进程申请访问临界资源的相关信息，这样就增加了系统中更多的系统开销了，所以这个方法可能在在实际应用存在较为明显的弊端。

此前，Ricart 和 Agrawala 提出了一种分布式互斥算法[15]（它最多允许一个进程进入临界区），每个进程都要给其他进程发送消息来确定它们是否获得临界资源。在后几年里，Maekawa[14]和 Suzuki[16]等人提出了互斥算法的各种新的性质和观点，其中有 Raynal[20]出版专门与互斥算法有关的书籍；随后，Kerry 提出一个新的算法[9]，扩展了 Ricart 互斥算法中临界区的数量，与此算法类似的还有其他相关算法[10,11,12]。在 Kerry 算法中，有 n 个进程共同访问 k 个临界资源，当有 m（m<k）个进程在临界区时，那么在临界区外有 n-m 个进程，它们只需要发送 n-m-1 个消息，最好的情况下，只需发送 n-(k-1)-1 个。

我们将采用概率模型检测工具对上面的算法建模，由于上述算法的结构比较

复杂，我们会对模型进行简化，减少模型的状态空间。在模型中，我们首先需要验证的是互斥算法的基本性质：(1)无死锁，(2)无饥饿。

上面提出的 Kerry 算法[9]已经从理论上验证了这两种结果，并且后来 Shailaja[13]通过数据结构和错容问题[19]对它做出了实验分析，而我们的概率模型检测工具是通过穷举迁移状态来建立模型的，它能够全面模拟给出的算法，从而得出系统结构，并且还能获得更多的相关实验结论的验证和分析。

实验中，我们除了分析出它的基本属性，我们还根据 PRISM[17, 18]工具提供的规约属性验证方法，单独分析发送延时和接收延时所占比重对进程进入临界区的影响，这种方法抛开了其它因素对延时的影响；同时我们通过实验得出，当各个进程占用临界资源的时间相同时，k 的大小对进程进入临界区的及时时间影响不大，但如果各个进程占用资源的比例较大时，会出现不同的情况；如果在现实中，各个进程占用临界资源相对及时，那么我们可以增加少量的额外临界资源提供给那些需求资源的进程。

分布式 K 互斥算法的定义：在一个分布式互斥系统中，有 n 个进程同时访问 k（k≤n）个临界区资源；每次只能有一个进程进入临界区，最多有 k 个进程在临界区；当进程退出临界区时，空闲的临界区资源可以被新申请访问的进程访问。

Kerry 互斥算法方法：这里我们首先也是提出算法在实际中的属性，然后设计出合理的算法模型，得到验证结果，最后从理论上证明我们的实验结果。最后，我们还提出新的改进算法。

本文结构如下：第 4.2 简单描述 MDP 模型，另外主要讲解 Kerry 算法[9]建模和实验结果，第 4.3 节讲解 Kerry 算法的实验验证结果，第 4.4 节给出分析和证明，4.5 节是改进 k 互斥算法，4.6 节是本章小结。

4.2 Kerry 算法建模

马尔科夫决策过程（MDP）是一个 4 元组 $M = (S, \bar{s}, Q, L)$：

S：有穷状态集合；

$\bar{s}$： $\bar{s}$ 是模型的初始状态集合， $\forall s \in S' \rightarrow s \in S$。

P：$S \times S \rightarrow [0,1]$，一个转移概率矩阵，对于所有的 s，满足 $\sum_{s' \in S} P(s, s') = 1$；

L：$S \to 2^{AP}$，每个状态映射到原子命题集合的标记函数，这个集合属于 L。

在 PRISM 工具中，我们用其形式语言描述 MDP 模型，这样就可以用 PRISM 来解决含有马尔科夫决策过程的问题。

MDP 模型与 DTMC 有一些区别，它们的逻辑属性表达方式不一样，DTMC 模型中，直接取平均值，MDP 模型中主要取最大或者最小值，但可以改变逻辑属性公式来表达我们所需要的描述的行为。在 MDP 中，我们使用 Rmin=?[F p1=3] 表示进程 1 的状态最终为 3 的最少时间是多少；在 DTMC 中，就变成 R=?[F p1=3] 表示进程 1 的状态最终为 3 的预期平均时间是多少。

我们将把 Kerry 算法[9]建立成一个 MDP 模型，在一个 MDP 模型中有多个模块，每个模块表示一个进程，一个模块里面含有多行命令，用来描述每个模块中状态的迁移过程，与上一章节一样，guard 表示迁移的条件，prob_1,…prob_n 表示状态迁移后的概率值，它们的和为 1，update_1,…update_n 表示发生迁移后的状态有 n 种可能。

```
1   mdp
2   const double delay;const k_vary;const max_seq=6;
3   formula
4   k=(p1=3?1:0)+(p2=3?1:0)+(p3=3?1:0)+(p4=3?1:0)+(p5=3?1:0)+(p6=3?1:0);
5   formula cs=(cs1=true?1:0) + (cs2=true?1:0) + (cs3=true?1:0) + (cs4=true?1:0) +
6   (cs5=true?1:0) + (cs6=true?1:0);
7   module pro1
8   p1:[0..3] ;seq1:[0..max_seq+1];   cs1:bool;
9   [step1] p1=0 -> (p1'=1)&(seq1'=max_seq);
10  [request1] p1=1 -> (p1'=2);
11  [step2] (p2=0) & (seq1>1) -> (seq1'=seq1-1);
12  ……
13  [response1] (p1=2) & k<=k_vary &
14  ((k=0&seq1<seq2&seq1<seq2&seq1<seq3&seq1<seq4&seq1<seq5&seq1<seq6
15  )|(k=1&(seq1<min(seq2,seq3,seq4,seq5)|……))|(k=2&(seq1<min(seq2,seq3,seq
16  4)|……))|(k=3&(seq1<min(seq2,seq3)|……))|(k=4&(seq1<seq2|……)) ->
17  (p1'=3)&(seq1'=0);
18  [execute1] p1=3 -> (p1'=0)&(seq1'=max_seq+1);
19  endmodule
20  module
21  pro2=pro1[cs1=cs2,p1=p2,seq1=seq2,p2=p1,seq2=seq1,request1=request2,
22  response1=response2,step1=step2,step2=step1,execute1= execute2] endmodule
```

```
23    ……
24    init （p1=0&……&p6=0)&(seq1=max_seq+1&……&seq6= max_seq+1)&
25    (cs1=false &……&cs6=false) endinit
26    rewards [] true :1; [step1] true :1;……;[step6] true :1; [execute1]
27    true :1;……;[ execute6] true :1;
28    [request1] true :（max_seq-1）*delay;……; [request6] true : （max_seq-1）
29    *delay;
30    [response1] true : (max_seq-1-k)*(1-delay);……;[response6] true :
31    (max_seq-1-k)*(1- delay);endrewards
```

图 4.1　6 进程的 Kerry 算法模型

上图是用 PRISM[17, 18]工具建立 Kerry 算法[9]的 MDP 模型，每个进程之间相互发送的信息的情况；而这一过程，将隐含在状态迁移的过程中，后面我们可以通过属性验证规约来验证。

图 4.1 中，我们只保留了进程的状态和它的序列编号，delay（延时），max_seq（最大编号）和新的变量 k_vary(k 的值)，formula k 用于确认临界区的进程个数，formula cs 用于后面平均及时时间的计算；

由于建模需要，它简化了 Kerry 算法，另外，也增加了一些我们所需要的参数，因为在 Kerry 算法的伪代码中，有些参数可能会溢出建模的上限，例如上图中的最大编号（max_seq），在原来的算法中，这个值会不停的递增，然后再分配给各个进程，但是如果我们的概率模型检测是用穷举状态空间的方法来抉择的，这样就会使得最大编号无限增长，所以在这里我们需要通过其他方法来改变这个值，使其不会在模型中溢出，在模块进程 1 中的行 11 的 step2 就表示除了进程 1 以外的其他进程中，如果有些进程正要申请临界资源（0->1），那么这个进程的编号为最大编号，而进程 1 的编号要减小一个时间单位；同理，其他的所有进程都必须减少一个时间单位。上面的 cs 就是用于表达在临界区有哪些进程，这样有助于后面的平均及时时间的验证。还有行 26-31 中，[]，[step1]等，用于设置每个模块中发生不同迁移的运行时间，而所有的[step1]表示此步是同步的（同一运行机制），如果有那个模块中的这个不满足迁移条件，那么就含有[step1]的模块的此步就不会发生。在设计算法建模时，就特别需要注意了。

行 7-19 描述模块 1（进程 1）的状态和序列号的迁移规则，进程状态 p1（0 空闲、1 申请访问、2 发送消息、3 进入临界区），行 9 表示进程 1 尝试申请访问时，它的状态变为 1，并且序列号变为最大，这表示当前它是最后一个申请访问

的进程；行 10 表示发送消息；行 11 表示当进程 2 发生了和进程 1 类似的迁移，那么进程 1 的序列号要减少，在 Kerry 算法[9]中是通过不断增加 max_seq 的值来解决序列号的大小问题的，但在 PRISM[17, 18]工具中，它需要穷举所有迁移状态，这样 max_seq 的值会变得无穷大，所以我们需要通过这种方法来解决 max_seq 变大的问题；行 12 与 11 类似，表示进程 3，进程 4，进程 5，进程 6；行 11-13 表示进程 1 获得进入临界区的条件，例如：当进程 1 的状态 p1 为 2，在临界区中的进程个数 ＜k_vary 时，并且进程 1 的序列号 ＜ 在外面进程的序列号，那么进程 1 就可以进入临界区了;其中行 14 中的 k=1 表示当临界区只有 1 个进程时，进程 1 的序列号只需要小于临界区外面的任意 4 个进程的序列号就可以进入临界区；行 18 表示进程 1 退出临界区；行 21-23 表示进程 2、进程 3…进程 6。行 24 行 25 表示初始状态。行 26-31 用于描述进程之间进入临界区前，相互发送的延时，同时需要通过后面的属性规约来结合使用；在行 26 和行 27 中，可以把除了发生延时外的其他时间设置为 0，这样就可以只考虑延时的影响了；行 28 和行 29 表示每个进程给其他进程发送的消息数量，最后两行表示某进程收到其他在临界区外的进程发送给它的消息个数。

4.3 Kerry 算法验证

4.3.1 Kerry 算法的无死锁和无饥饿

K 互斥算法首先要保证各个进程之间无死锁和无饥饿；（1）无死锁：从初始状态开始，系统中的各个进程将能连续不断的进入和退出临界区；（2）无饥饿：某个进程申请进入临界区，最终一定能得到临界区资源。

无死锁：Kerry 算法[9]在建模过程中，PRISM[17，18]工具的 Build model 就能检验出模型是否存在死锁；

无饥饿：设置 seq1=1、p2=3、delay=0.5、k_vary=1, 表示进程 1 当前正准备发送申请，进程 2 已在临界区，那么在临界区外有 5 个进程，用 Rmin=? [F p1=3] 和 Rmax=? [F p1=3]验证进程 1 进入临界区的最小时间和最大时间，结果分别是 4.5 和 15（从初始状态开始）。

最开始进程 1 需要给在其他的 5 个进程发送消息，但只需要获得在临界区外面的 4 个进程的回复即可，所以它的延时为（5+4）×delay=4.5，我们的验证结果满足条件（N-1）+（N-K-1）。

4.3.2 Kerry 算法的延时比例的影响

我们的算法中，主要涉及到延时有发送消息和接收消息，分别用 delay 和 1-delay 表示。K_vary 设置为 1，这里我采用 filter 在不同 delay 中，提取某一进程进入临界区最小值的平均值，时序逻辑属性验证公式为 $R_{max=?}$[F p1=3]和 $R_{min=?}$[F p1=3]。从图 4.2，可以看出当发送消息的比重越大时，进程进入临界区的最小延时的平均时间没有变化；而进程进入临界区的最大延时的平均时间会变得越大；接着我们通过修改模型为 DTMC 后，用规约属性公式 $R_{=?}$[F p1=3]得出它们的共同的平均延时会变大。

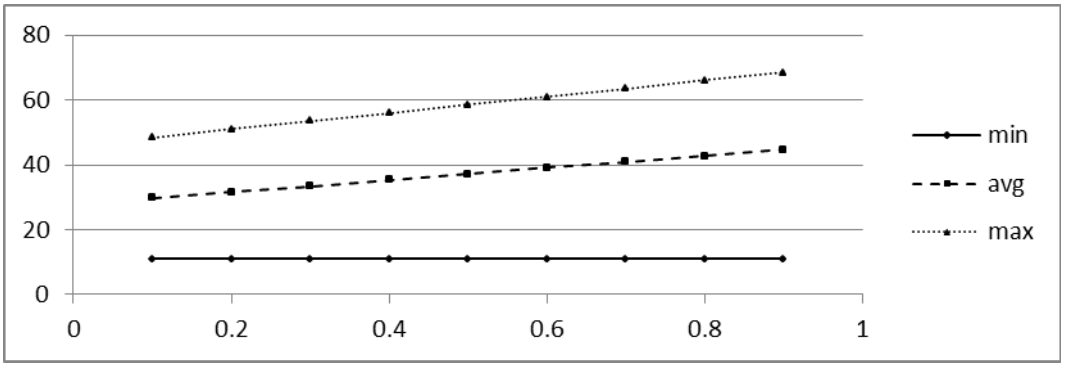


图 4.2　不同比例延时的影响

4.3.3 临界资源数量 K 的变化对算法的影响

我们通过改变 k_vary=4，则此时能够进入临界区的最大进程数目为 5，设置 delay=0.5 属性验证公式为 $R_{min=?}$ [F p1=3 & p2=3 & p3=3 & p4=3 & p5=3]，所需要的时间为 26。另外，这里我们和上面一样通过改变 k_vary 的值来观察，它对某一进程进入临界区的影响，这里我们用的规约属性验证公式为($R_{min=?}$ [F p1=3&cs=1]+ $R_{min=?}$ [F p1=3&cs=2]+$R_{min=?}$ [F p1=3&cs=3]+ $R_{min=?}$ [F p1=3&cs=4]+$R_{min=?}$ [F p1=3&cs=5]+ $R_{min=?}$ [F p1=3&cs=6]) / 6；图 4.3（a）是实验结果，从结果中，我们可以看出当所有进程占用临界资源的时间相同时，k_vary 的变化对于某个进程进入临界区的平均及时时间影响不大；但如果某些在临界区的进程占用临界资源的时间相对较长（execute6=100 表示进程 6 将长时间占用临界资源），那么对于某一进程进入临界区的平均及时时间会增加，则需增加少量临界资源来降低平均及时时间；如图中所示，我们发现当 k_vary=2 基本上就能满足以上条件了。

如果把某一进程改为任意一进程，如图 4.3（b），规约属性验证公式为 $R_{=?}$ [F

k=a])（a=1,…，5）（DTMC），那么 k_vary 的大小几乎对于任意一进程进入临界区的平均时间没有任何影响。这也从侧面证明了我们的模拟过程对所有的进程都是公平的，当然如果增加或减少临界区资源数量（k），对任意一进程进入临界区进入临界区的时间还是有显著的区别。

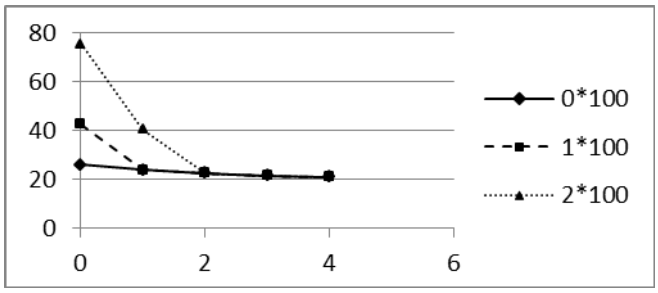


图 4.3（a） k_vary 对某进程进入临界区的影响

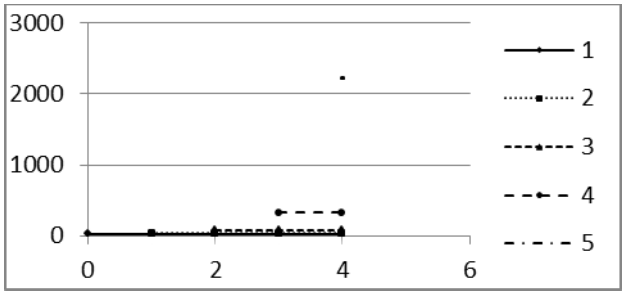


图 4.3（b） k_vary 对任意一进程进入临界区的影响

4.4 Kerry 算法分析和证明

4.4.1 Kerry 算法实验分析

在图 4.3（a）中的 3 条线段分别表示 0 个，1 个和 2 个进程在临界区的执行时间为 1。则其它进程中的某一个进程及时进入临界区的最短时间。例如进程 1，从初始状态开始，它可以为第 1 个进入临界区的进程，可以是第 2 个，第 3 个……，第 6 个；这 6 种情况下，此进程进入临界区的时间都最及时（最快）。当所有进程的执行时间为 1 时，并且只有一个临界区（这里 k_vary=0）进程 1 是第 1 个、第 2 个……第 6 个，进入临界区的最短时间为 11，17，23，29，35，41。所以进程 1 及时进入临界区的最短时间为(11 + 17 + 23 + 29 + 35 + 41) / 6 = 26;依次可得，2 个临界区（k_vary=1），进程 1 进程的最短时间为 23.917，3 个临界区最短平均时间为 22.417，4 个临界区最短平均时间为 21.417，5 个临界区最短平均

时间为 20.833；也就是图 4.3（a）中那条实线，从图中可以看出临界区数量增加了对于进程 1 进入临界区的时间变并不大。但是当系统中存在其他进程中的某一个的执行时间相对于其他进程的执行时间要高出不少时，例如这个进程的执行时间为 100，进程 1 的平均及时时间大大增加；从图中，我们可以发现当增加 1 个临界区时，进程 1 进程及时进入临界区的时间和以前一样；当有 2 个进程在临界区的执行时间为 100 时，则需增加 2 个临界区资源同样可以让进程 1 进入临界区的平均及时时间和原来保持一致。现在我们就可以简单地总结出，在各个进程访问临界资源的时间相差不大下，增加临界资源的数目对于进程访问临界资源没有显著的提高；如果访问临界资源的时间相差较大，那么增加临界资源可以显著提高它的效率。

4.4.2 Kerry 算法实验证明

从以上的实验分析，我们可以得出两个结论；接下来我们将从理论上证明我们的结论对于整个 Kerry 算法都是有效的。

结论 1：在 N 个进程中，如果 N 个进程在临界区的执行时间相差不大，那么增加临界区的数量（k）对某一进程及时进入临界区的时间影响不大。

证明：

预设条件：当 k=0 时(只有一个临界区)

假设共有 N 个进程，进入 k 个临界区中，任意一进程申请临界区的时间为 1，发送给其他任意一进程消息时间为 0.5，接收某一进程的消息时间为 0.5，临界区执行时间为 1，初始状态下，所有进程的序列号为最大。

（1）当 k=1 时，进程 1 第一个进入临界区时，需要给其他 N-1 个进程发送消息，并且接收消息，每个进程都需要申请临界区获得序列号，则进程 1 第一个进入临界区的最短时间为 $2 \times (N-1) \times 0.5 + N = 2 \times N - 1$；

（2）当下一进程进入临界区时，进程 1 需退出临界区，所需时间为 1，同时此进程需给其他 N-1 个进程发送和接收消息，则此时，对于第 2 个进入临界区的进程来说，进程的最短时间为 $2 \times (N-1) \times 0.5 + N + 1 + 2 \times (N-1) \times 0.5 = 3 \times N - 1$；

（3）同理可得，后面的依次为 $4 \times N - 1$，$5 \times N - 1$, $6 \times N - 1$, $7 \times N - 1$，……，$(N+1) \times N - 1$（共有 N 项）；

因此进程 1 及时进入临界区的平均时间为$(2 \times N - 1 + 3 \times N - 1 + 4 \times N - 1 +$

5 × N – 1 + ⋯ + (N+1) × N – 1) / N =(N+1) × (N+2) / 2 - 2;

当 k=1 时(只有两个临界区)

（1）进程 1 是第 1 个进入临界区的最短时间为 2 × N - 1（同上）；

（2）第 2 个进程进入临界区时，进程 1 不需要退出，那么，第 2 个进程的最短时间就会减少，则最短时间为 2 × N – 1 + ((N-1) + (N-2)) × 0.5 = 3 × N - 2.5；

（3）当第 3 个进程进入时，临界区里面的两个进程需要退出 1 个，则最短时间为 3 × N - 2.5 + 1 + ((N-1) + (N-2)) × 0.5 = 4 × N – 3；

（4）同理，后面的依次为 4 × N – 3 + 1 + ((N-1) + (N-2)) × 0.5 = 5 × N - 3.5，6 × N - 4，……，(N+1) × N - (0.5 × N + 1.5)；

（5）因此进程 1 及时进入临界区的平均时间为(2 × N – 1 + 3 × N - 2.5 + 4 × N – 3 + 5 × N - 3.5 + ⋯ + (N+1) × N - (0.5 × N + 1.5)) / N = (N+1) × (N+2) / 2 - (N-1) / 4 – 3 + 1/N；

同理，当 k=2 时，进程 1 及时进入临界区的平均时间为(N+1) × (N+2) / 2 - (N+1) / 2 + 2.5 / N - 2.5 等等。现在我们就可以推理出在添加临界资源数目的情况下，进程访问所需时间是否有较大差距，k=0 与 k=1 之间的最短时间差为((N+4) × (N-1)) / 4N ≈ (N+4) / 4，k=1 与 k=2 之间的最短时间差为((N+3) × (N-2)) / 4N ≈ (N+3) / 4, 时间差依次递减，而且它们远小于 k=0（或者 k=1，2）的平均及时时间。以上就说明了增加临界资源对于各个进程在临界区执行时间不大的情况下，没有太大的提高。

结论 2：在结论 1 中，如果某一进程的执行时间相对较大，则可以适当添加 1 个临界区数量来降低其他进程进入临界区的时间；此时整个系统的运行效果与减少 1 个单位临界区和降低那个进程的执行时间到结论 1 中所描述的情况；同时，临界资源数目设置为执行时间相对较高的进程的数目+1 最合理。

证明：

假设：在结论 1 的假设中，当除进程 1 以外的其他某一进程的执行时间增加 M 个时间单位，在 k=0 时；

（1）当进程 1 第 1 个进程临界区时，那个增量为 M 的进程对进程 1 没有任何影响，进程 1 进入临界区的最短时间为 2 × N - 1 不变；

（2）进程 1 第 2 个进入临界区时，其他 N-1 个进程中，任意一个进程第 1 个进入临界区的概率为 1 / (N-1)，但进程 1 需要在最快进入临界区，所以增量为 M 的进程不能第 1 个进入临界区，那么进程 1 进入临界区的最短时间为 3 × N - 1

不变；

（3） 同理可得，进程 1 第 3 个进入临界区的最短时间和结论 1 中相同，将保持不变；

（4）但进程 1 最后一个进入临界区时，增量为 M 的进程则需在进程 1 之前进入临界区，在这种情况下，与结论 1 中，不同的地方在于，增量为 M 的进程的执行时间占用了不少时间，进程 1 的最后一个访问的时间，需要添加 M 个单位，那么最短时间为(N+1) × N – 1 + M，所以平均及时时间为(N+1) × (N+2) / 2 – 2 + M / N，图 4.3（a）中给出了实验结果。

当 k=1 时，与结论 1 中进程 1 相比较，需要注意的地方在于最后进入临界区的时间，由于添加了临界资源的数目，这时增量为 M 的进程可以进入临界区不出来，这时的效果与结论 1 基本一致，那么对进程 1 的平均及时时间毫无影响。从我们的实验结果中也可看出。

同理可得，当有 2 个增量为 M 的进程，需要额外添加 2 个临界资源，最关键的地方在于，某一进程在最后进入临界区时，它是否可以在增量为 M 的进程不出来的情况下，就能进入临界区。这样就可以保证进程的及时访问临界资源的时间了。

从结论 2 中，我们可以看出临界资源的增加主要对那些需要快速访问临界资源的进程有较大提高，而对于那些在临界区执行时间较长的进程来说，它访问临界资源的速度在于其他进程在临界区的执行时间，所以不会有明显的提高；但如果有执行时间较长的进程长时间在临界区，那么它对其他执行时间较长的进程还是有影响的。这些结论告诉我们，在多进程访问临界区的系统中，增加临界资源所带来的高效性，主要在于当临界区中存在大量的长时间进程时，它是否能够使得在临界区外的进程快速进入临界区。

4.5 改进 k 互斥算法

4.5.1 改进 K 互斥算法简介

从 Kerry 算法，可以看出进程在访问临界区时，需要相互发送信息，来确认是否有权利进入临界区，以及后面的 Shailaja 引入数据结构减少了接受消息的时间，从而提高系统效率；这里我们通过上述两种算法得出了一个新的算法。在系统中，各进程不需要相互发送消息，而是发送给特定的分析器，通过对构建

临界区结构的当前相关信息，而使得各进程直接通过此结构，就能确认是否可以进入临界区。

系统中共有 k 个临界区资源，此时有 N 个进程需要临界资源，N > k。临界区信息结构器 CS_array，在 CS_array 中共有 N 个存储单元，用于存储进程的编号，编号在前面 k 个单元中的进程表示能够进入临界区，编号在后面剩余 N-k 个表示等候进入临界区。初始状态全部为空。

某一进程在访问临界区时，首先向临界区信息结构器发送消息，如果前 k 个单位为空，临界区信息数组把此进程的编号放入前 k 个单位中的其中一个；如果前 k 个单元不为空，那么临界区信息结构器把它放入后面的单元中，不给此进程发送消息，当有一个进程从临界区退出时，这个进程给临界区信息结构器发送退出消息；此时，临界区信息结构器把含有此进程编号的单元设置为空，这时后面的剩余单元组中最前面的单位中的编号进入前面的空闲单元，然后临界区信息结构器给此编号的进程发送进入临界区信息，此时当前进程就可以访问临界区。

从以上可以看出，某一进程访问临界区时，就不在需要给其他进程发送消息了。而只要给临界区信息结构器发送，同时临界区结构器给此进程发送确认消息，这个进程就可以访问临界区了。一个进程进入临界区的时间为：发送的时间+接受时间。这里假设发送时间和接受时间都为 1，那么最快访问临界区时间为 2，明显比其它 k 互斥算法要高效；一个进程最后访问临界区时，那么前面的 N-1 个进程中必须有 N - k 个进程退出临界区，这 N-k 个进程都需要给临界区信息结构器发送退出消息，共需发送 N-k 次消息。那么此进程最后访问临界区，共需发送和接受的消息的时间为：1 + 1 + N – k = N – k + 2；也要高于一般的 k 互斥算法。

4.5.2 改进 K 互斥算法实现

进程的数据结构：访问临界区 Req_CS：boolean， 执行临界区 Exe_CS：boolean，进程编号：Own_Seq：> 0；

临界区信息结构器 CS_Rem：

1、存储进程区进程编号：CS_array[1..k]，初始值均为 0，CS_cur_null_flag 标记存储进程区当前为空的单元下标，初始值为 1，CS_all_flag：boolean，标示存储进程区是否全部占满，初始值为 false；

2、存储剩余进程编号：Rem_array[1..N-k]，初始值为均 0；Rem_cur_null_flag 标记当前剩余进程区为空的单元下标，初始值为 1，Rem_null_flag：boolean，标示剩余进程区是否全部为空，初始值为 true，Rem_cur_frist_Waiting 标记当前剩余进程区最前端等待进入进程临界区的下标，初始值为 1。

3、系统编号：S_seq；

改进 k 互斥算法：

1、进程 X 申请访问临界区资源

```
Req_CS : = true ;
Exe_CS : = false ;
Own_Seq : = S_seq + 1 ;
S_seq : = S_seq + 1 ;
    begin
        send request(Own_Seq) to CS_Rem；
    end
```

2、 存储进程区接受请求

```
receive request(Own_Seq) from proc X
if CS_all_flag = false and Rem_null_flag = true then
    CS_array[CS_cur_null_flag] : = Own_Seq ;
    send inform(true) to CS_array[CS_cur_null_flag] ;
end if ;
i : = CS_cur_null_flag ;
for i < = k
    if CS_array[ i + 1 ] = 0 then
        CS_cur_null_flag : = i ;
    else
        i : = i + 1 ;
    end if ;
if   i = k
    CS_all_flag := true ;
else
   CS_all_flag = false ;
```

```
end if ;
```

3、剩余进程区 Rem_array

```
if CS_all_flag = true
    Rem_array[Rem_cur_null_flag] : = Own_Seq ;
end if ;
   for i <= N - k
    i : = Rem_cur_null_flag ;
    if CS_array[i] = 0 then
        Rem_cur_null_flag : = i ;
    else
        i : = i + 1 ;
```

4、进程 X 获得临界区资源

```
if inform is true from CS_Rem
    begin
        Req_CS : = false ;
        Exe_CS : = true ;
        enter critical section ;
    end
end if ;
```

5、进程 X 退出临界区

```
proc X exit Critical Section
    begin
        Exe_CS : = false ;
        send exit(Own_Seq) to CS_Rem;
    end
```

6、CS_Rem 区接受进程 X 的退出消息

```
receive exit(Own_Seq) from proc X
i : = 1 ;
for i < = k
    if CS_array[i] = Own_Seq then
    CS_array[i] : = 0 ;
```

```
    exit ;
    end if ;
if CS_all_flag   =   false then
    for   Rem_cur_first_Waiting ! = Rem_null_flag and i < = k
        CS_array[CS_cur_null_flag] : = Rem_array[Rem_cur_first_Waiting] ;
        send inform(true) to CS_array[CS_cur_null_flag] ;
        Rem_cur_first_Waiting : = mod( Rem_cur_first_Waiting + 1 , N – k )
        num = CS_cur_null_flag ;
        j = CS_cur_null_flag ;
        for j < = k and mod(j + 1 , num) ! = 0
                if CS_array[ mod(j + 1, k)] : = 0 then
                    CS_cur_null_flag : = j;
                else
                    j : = j + 1;
                end if;
        i : = i + 1 ;
    if   Rem_cur_first_Waiting = Rem_null_flag
        Rem_null_flag := true ;
     Else
        Rem_null_flag := false ;
    end if ;
end if ;
```

算法 4.5.1 表示进程 X 进入临界区时，Req_CS 为真，Exe_CS 为假，进程 X 的编号设置为系统编号，然后系统编号需递增，因为它需要保证每个进程的编号不相等，然后把消息发送给临界区结构器；

算法 4.5.2 表示存储进程区获取来自进程 X 的请求临界资源的消息，当前状态下，存储进程区没有被占满时，那么就可以直接接受进程 X 的请求，从而给进程 X 返回确认信息，后面的 for 循环来查找新的空闲存储单元，为 CS_cur_null_flag 设置新值；

算法 4.5.3 表示当在 中的存储进程区被占满时，那么进程 X 的请求消息就会被保留在此区域，同时也需要查找新的存储空闲空间，把此处的下标赋值给

Rem_cur_null_flag；

算法 4.5.4 表示进程 X 获得来自临界区结构器的确认消息，然后就可以访问临界区资源；

算法 4.5.5 表示进程 X 执行完临界资源后，就需要从此区域退出，然后需要把自己的退出信息发送给临界区结构器。

算法 4.5.6 表示临界区结构器接受来自进程 X 的退出信息后，需要做内部调整，把剩余进程区中的进程编号调整到存储进程区，这里的剩余进程区使用 FIFO 机制进入存储进程区，然后再给在此区域的进程发送进入临界资源区的确认消息。

从算法中，我们可以看出进程发送和接受的信息的次数大量的减少了，它们只需要与临界区结构器发送和接受消息。另外，从系统中，我们可以发现存储进程区的进程在退出临界区的时间各不相等，以至于在此区域的进程空闲出来的空间可以不连续，而在剩余进程区的进程是排队进入存储进程区；如图 4.4 所示，我们在调整剩余进程区中的编号到存储进程区时，前者是采用环状循环提供进程编号的方式，后者则是采用循环搜索的方式接受来自剩余进程区的进程编号。

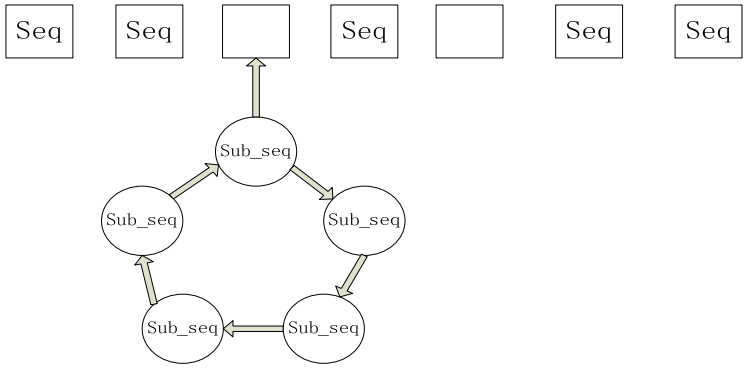


图 4.4 N=12，k=5 临界区信息结构器 其中上面表示存储进程区，下面的环表示剩余进程区

上述分析，我们可以看出系统的主要开销在临界区结构器中，需要不断获取进程的请求消息和退出消息，并且返回确认消息，还需要做内部结构调整，在这一过程中，需要做循环判断，例如在算法 4.5.6 中，嵌套了 2 个 for 循环，此处最复杂情况是，存储进程区全部为空，剩余进程区的所有进程都需调整到存储进程区中，那么 $k \geq N - k$，那么最大时间为 $k \times N - k \leq (k + (N - k))^2 / 4 = N^2 / 4$，由于进程编号从剩余进程区被调整到存储进程区时，就会立刻发送消息给进程，那么上述时间对某一进程就不会有太大的影响；另外，如果有一进程退出

临界区就会给临界区结构器发送消息，存储进程区和剩余进程区就会做调整，此时，也不会出现上述情况。

另外在 PRISM 中，我们也实现了这个算法，但由于临界区结构器的复杂性，我们只能通过在 PRISM 模块中设置相关参数来简化此结构器的结构。在算法中，与 Kerry 算法一样，验证了某一进程进入临界区的发送和接受消息的次数，实验结果与我们结论相同。

4.6 本章小结

4.6.1 Kerry 算法小结

从前面的建模中，我们可以发现在概率模型检测中构建 Kerry 算法，不仅需要理解 Kerry 算法，同时在设计模型时，还需要注意算法在概率模型检测中可能出现的弊端，例如那个最大编号问题，这时就需要通过为所有进程重新设定它们的编号变量来满足概率模型检测器的建模规则；同时，在我们设定的适合实际生产中的那些逻辑属性，还需要通过借助额外的公式和变量用于这些逻辑属性的验证。在后面的分析和证明中，基本上完全给出了在实现算法的系统中，对临界资源的需求问题，什么情况下需要添加临界资源，什么情况不需要添加，这样有利于系统合理的分配宝贵的资源。

4.6.2 改进 K 互斥算法小结

在改进的算法中，我们添加了额外的存储空间，使得各个进程间不需要再相互发送消息，这样就减少了发送和接受消息的延时；而此处的添加的存储空间，主要是调整队列中的进程编号，此处的运行都是临界区结构器内部的运行过程，并不需要发送和接受延时，总体而言，系统开销也不太大。

4.6.3 k 互斥算法小结

k 互斥算法验证研究方法：

（1）首先选择合适的算法，并分析算法的互斥思想和它的运行机制；

（2）给出简洁的算法概率构造模型，使其模型能够满足算法的运行机制，否则需要通过增加或修改一些变量使得算法满足模型的规则，但需保证算法的运行机制不变；

（3）给出在实际应用中合理的逻辑属性，然后验证它，如果有必要，还需添加额外的变量和公式，以此来满足验证需求；

（4）接着需要对实验结果做出分析，重点在于算法在实际中的资源需求，相关效率，以及它的应用场景；

（5）对于我们的分析结果，还需要给出理论的证明过程，主要还是证明与实际相关的结论。

在上述方法的最后第 5 条，可能由于算法模拟出的系统比较复杂，不便于理论上的证明，那么可以采用其他的方式，比如在上一章，我们用了线性回归的方式拟合出我们的实验结果。不过这个方法不能保证我们的结论绝对严谨。

我们实验中的 k 互斥算法，在系统中执行时，各个进程请求和访问临界资源的方式相对简单，但它的一般性却较为实用。前面讲到了其他的高效算法，但它需要添加额外的数据结构，而且必须保证临界区时满负荷状态，不利于实际需求，所以没有做出解释，而我们研究的这种情况将普遍发生在实际应用中，整个系统对于临界资源的需求到底是什么程度。都给出了详细的分析和证明。在我们的实验，主要是通过简化地模拟整个系统的运行过程，然后通过不同的规约属性验证公式得出实际中可能需要的结果；从实验中，可以很容易地看出临界资源的多少对于进程进入临界区的影响，需要考虑其他进程占用资源的时间大小。所以在选取算法时，尽量保证它的合理性。

在后面，我们也给出了自己的算法，虽然也添加了额外的数据空间，但效率有了明显的提高，对于繁忙的信道消息传递系统来讲，此算法还是有实用价值的。

第 5 章 结论

5.1 研究总结

分布式的各类算法已经广泛应用于不同的领域，特别是在近几年中，随着互联网中云计算、大数据等的提出，使得各种分布式技术也得到了发展。如果我们能知道相关分布式算法的各种性能，那么对于网络中的不同需求，我们就可以采用合适的分布式算法。本文主要采用模型检测工具研究了 Flatebo 自稳定算法和 Kerry 互斥算法，并给出了它们的相关性能分析和证明结果，还有给出了两类不同算法的研究方法，以及它们在实际应用中可能出现的情况和我们对此的需求。

总体来讲，本文主要的工作有：

（1）验证 Flatebo 自稳定算法的各种属性，从中我得到此算法要比同类的异步自稳定算法高效；并且比较它与 Herman 同步自稳定做了相关比较；随后我们采用线性回归证明此算法的时间复杂度在 $O(N^{1.45})$和 $O(N^{1.47})$之间。这将更有利于我们在实际中使用。简单介绍对于各类自稳定算法的分析思路和过程；

（2）对于互斥算法，我们选择具有代表性的 Kerry 算法，它简单地描述了 n 个进程访问 k 个临界区的思路；试验中，我们简化了实验模型，但也添加了更多利于实验分析的参数；除了验证它的各类性质外，我们还增加了 k 的变化会带来不同实验结果的验证，从结果中我们可以看出，如果各个进程享用临界区的时间比例相近，那么我们不需要添加更多的临界资源；但如果这个比例较大，则添加更多的资源将会有利于其它进程对临界区的访问；后面还给出了添加临界资源的具体是在情况下。对于互斥算法，我们也给出了一套研究方法，它可以使得我们在互斥算法领域对更多的有利于实际应用中的合理的算法做出分析和验证；

（3）文中我们还通过 kerry 互斥算法和其他相关的 k 互斥算法提出了新的 k 互斥算法，它通过增加临界区结构器替换了进程间相互发送消息的情况，减轻了信道的传输负荷。

5.2 进一步开展的工作

上述给出了通过 PRISM[17, 18]工具，我们获得两类分布式算法的属性和相关性质，并且我们还证明了其中一部分的性质，从中我们知道在什么情况适合使用它们；而对于这两类算法，甚至整个分布式算法领域，我们还需要更多地进行扩展和分析，进一步工作：

（1）实验完成了 Flatebo 算法的验证，并一定程度上证明了其性质，而从理论上证明它的精确复杂度时，我们发现它与马尔科夫链技术在概率模型检测上的应用有着很深的联系，如果我们能够对此作出更好的分析，那么它将能更好地解释 PRISM[17, 18]工具的运行机制，也会有利于我们对它的使用。

（2）这里只给出了一个典型的互斥算法，它能描述一般情况下的互斥性质，实验和证明已经告诉我们，我们可以在什么情况使用它，但是对于要求更高的应用环境，我们还得采用更高效的算法，其中本文中有提到 Shailaja 提出新的采用数据结构的方式的高效互斥算法；对于互斥算法，我们进一步的工作对此算法做出类似的验证和分析。另外，更主要的是要选取适当的互斥算法，它对于我们的模型构造，分析和证明，以及包含它在实际中具有重大意义；

（3）从对算法的验证研究中，我们更改参数，引入新的思想来降低算法带来的系统开销，从而也进一步的对算法进行了改进。

以上给出了我们接下来的工作，从中我们可以看到，主要是对分布式算法的自稳定算法和互斥算法的研究，在我们的工作之前，已经出现了概率模型检测对各类分布式算法的验证和分析，我们的进一步工作将更加有利于完善分布式算法的各种性质，以此将它们应用到实际生活中。并且对于合适的算法，我们会根据实际需要，还可以提出进一步完善和提高的算法。

参考文献

[1] Marta Kwiatkowska, Gethin Norman, David Parker.et al. Probabilistic Verification of Herman's Self-Stabilisation Algorithm [J]. Formal Aspects of Computing, page(s):661-670 Volume: 24 Issue: 6, 2012,

[2] Mitchell, Flatebo, Datta.et al. Two-State Self-Stabilizing Algorithms for Token Rings [J]. IEEE Transaction On Software Engineering, page(s):500-504 Volume: 20 Issue: 6, 1994

[3] Mitchell, Flatebo, Datta,et al. Two- State Self-Stabilizing Algorithms[A]. In IEEE ed. Proceedings of the 6th International Parallel Processing Symposium[C], United States 1992 page(s):198-203 ISSN: 01903918

[4] Annabelle McIver and Carroll Morgan. An elementary proof that Herman's Ring is $\Theta(N^2)$ [J]. Infromation Processing Letters, page(s):79-84 Volume: 94 Issue:2, 2005

[5] Herman, Ted. Probabilistic Self-Stabilization [J]. Information Processing Letters, page(s):63-67 Volume:35 Issue:2, 1990

[6] E.Dijkstra, Self-stabilizing systems in spite of distributed control [J], Communications of the ACM. Page (s): 643-644 Volume:17, 1974.

[7] Kiefer Stefan, James, Zhang Lijun et al. On Stabilization in Herman's Algorithm [A]. In Springer Verlag ed. 38th International Colloquium on Automata, Languages and Programming, ICALP 2011[C] Switzerland 2011 page(s):466-477 ISSN:03029743

[8] P.J.A.Lentfert, S.D.Swierstra. Towards the formal design of self-stabilizing distributed algorithms [A]. In Springer Verlag ed, Proceedings of the 10th Annual Symposium on Theoretical Aspects of Computer Science[C]. 1993 pages 440-451.

[9] K Raydond. A distributed algorithm for multiple entries to a critical section [J]. Information Processing Letters 1989: 30(1989) 189-193

[10] P K. Srimani and R L Reddy. Another distributed algorithm for multiple entries to a critical section[J]. Information Processing Letters.1992: 41 51-57

[11] K Makki, P Banta, K Been et al. A token based distributed k mutual exclusion algorithm[C]. In IEEE Proceedings of the Symposium on Parallel and Distributed Processing. Arlington TX, 1992

[12] S T Huang, J R Jiang, and Y C Kuo. k coteries for fault-tolerant k entries to a critical section[C]. In International Conf. Distributed Computing Systems. Pittsburgh PA. 1993

[13] S Bulgannawar and N H Vaidya. A Distributed K-Mutual Exclusion Algorithm[C]. Department of Computer Science, Proceedings of the 15th International Conference on, Vancouver BC, 1995

[14] M Maekawa. A $\sqrt{N}$ algorithm for mutual exclusion in decentralized systems[J]. ACM Transactions on Computer Systems. 1985: 3(2), 145 - 159

[15] G Ricart and A K Agrawala. An optimal algorithm for mutual exclusion in computer networks[J]. Communications of the ACM. 1981: 24(1) 9-17

[16] I Suzuki and T Kasami. A Distributed Mutual Exclusion Algorithm[J]. ACM Transactions on Computer Systems. 1985: 3(4) 344-349

[17] A Hinton, M Kwiatkowska, G Norman. PRISM: A Tool for Automatic Verification of Probabilistic Systems[M]. Tools and Algorithms for the Construction and Analysis of Systems Lecture Notes in Computer Science. 2006: 3920, 441-444

[18] M Kwiatkowska, G Norman, D Parker. PRISM: Probabilistic symbolic model checker[J]. Computer Performance Evaluation: Modelling Techniques and Tools Lecture Notes in Computer Science. 2002: 2324, 200-204

[19] M Trehel and M Naimi. A distributed algorithm for mutual exclusion based on data structures and fault tolerance. In 6th Annual International Phoenix Conference on Computers and Communications. 1987.

[20] M Raynal. Algorithms for Mutual Exclusion[M]. United States. MIT Press. 1986

[21] Adnan Aziz, Kumud Sanwal, Vigyan Singhal, and Robert K. Brayton. Verifying continuous time Markov chains. In Rajeev Alur and Thomas A. Henzinger, editors, Proc. 8th International Conference on Computer Aided Verification (CAV'96), volume 1102 of LNCS, pages 269–276, Berlin, 1996. Springer.

[22] David M. Young. Iterative Solution of Large Linear Systems. Acedemic Press, NY, USA, 1971. ISBN 0-12-773050-8.

[23] C. Baier. On algorithmic verification methods for probabilistic systems. habilitation thesis, University of Mannheim, 1998.

[24] Christel Baier, Boudewijn Haverkort, Holger Hermanns, and Joost-Pieter Katoen. Modelchecking algorithms for continuous-time Markov chains. IEEE Transactions on Software Engineering, 29(6):524–541, 2003.

[25] Christel Baier, Boudewijn R. Haverkort, Holger Hermanns, and Joost-Pieter Katoen. Model checking continuous-time Markov chains by transient analysis. In E. Allen Emerson and A. Prasad Sistla, editors, Proc. 12th International Conference on Computer Aided Verification (CAV'00), volume 1855 of LNCS, pages 358–372, Berlin, 2000. Springer.

[26] Christel Baier, Joost-Pieter Katoen, and Holger Hermanns. Approximate symbolic model checking of continuous-time Markov chains. In Jos C. M. Baeten and Sjouke Mauw, editors, Proc. 10th International Conference on Concurrency Theory (CONCUR'99), volume 1664 of LNCS, pages 146–161, Berlin, 1999. Springer.

[27] Christel Baier and Marta Z. Kwiatkowska. Model checking for a probabilistic branching time

logic with fairness. Distributed Computing, 11(3):125–155, 1998.

[28] Andrea Bianco and Luca de Alfaro. Model checking of probabilistic and nondeterministic systems. In P. S. Thiagarajan, editor, Foundations of Software Technology and Theoretical Computer Science, volume 1026 of LNCS, pages 499–513, Berlin, 1995. Springer.

[29] Peter Buchholz, Gianfranco Ciardo, Susanna Donatelli, and Peter Kemper. Complexity of memory-efficient Kronecker operations with applications to the solution of Markov models. INFORMS J. on Computing, 12(3):203–222, 2000.

[30] Sergio Pissanetzky. Sparse Matrix Technology. Academic Press, London, UK, 1984. ISBN 0-12-557580-7.

[31] Frank Ciesinski and Marcus Gr¨oßer. On probabilistic computation tree logic. In Christel Baier, Boudewijn R. Haverkort, Holger Hermanns, Joost-Pieter Katoen, and Markus Siegle, editors, Validation of Stochastic Systems, volume 2925 of LNCS, pages 147–188, Berlin, 2004. Springer.

[32] Edmund M. Clarke, E. Allen Emerson, and A. Prasad Sistla. Automatic verification of finitestate concurrent systems using temporal logic specifications. ACM Trans. Program. Lang. Syst., 8(2):244–263, 1986.

[33] Conrado Daws, Marta Z. Kwiatkowska, and Gethin Norman. Automatic verification of the IEEE 1394 root contention protocol with KRONOS and PRISM. International Journal on Software Tools for Technology Transfer (STTT'04), 5(2-3):221–236, 2004.

[34] Luca de Alfaro, Marta Z. Kwiatkowska, Gethin Norman, David Parker, and Roberto Segala. Symbolic model checking of probabilistic processes using MTBDDs and the Kronecker representation. In Susanne Graf and Michael I. Schwartzbach, editors, Proc. 6th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'00), volume 1785 of LNCS, pages 395–410, Berlin, 2000. Springer.

[35] H åkan L. S. Younes and Reid G. Simmons. Probabilistic verification of discrete event systems using acceptance sampling. In Ed Brinksma and Kim Guldstrand Larsen, editors, Proc. 14th International Conference on Computer Aided Verification (CAV'02), volume 2404 of LNCS, pages 223–235, Berlin, 2002. Springer.

[36] M. Fujita, P.C. McGeer, and J. C.-Y. Yang. Multi-terminal binary decision diagrams: An efficient datastructure for matrix representation. Form. Methods Syst. Des., 10(2-3):149–169, 1997.

[37] Hans Hansson and Bengt Jonsson. A logic for reasoning about time and reliability. Formal Aspects of Computing, 6(5):512–535, 1994.

[38] Abraham Wald. Sequential tests of statistical hypotheses. Annals of Mathematical Statistics, 16(2):117–186, 2007.

[39] Henk C. Tijms. A First Course in Stochastic Models. John Wiley & Sons Ltd, West Sussex, UK, 2003. ISBN 0-471-49881-5.

[40] Robert E. Tarjan. Depth-first search and linear graph algorithms. SIAM J. Compting, 1(2):146–160, 1972.

[41] William J. Stewart. Introduction to the Numerical Solution of Markov Chains. Princeton University Press, New Jersey, UK, 1994. ISBN 0-691-03699-3.

[42] Jane Hillston. Compositional Approach to Performance Modelling. Cambridge University Press, Cambridge, UK, 1996. ISBN 0-521-57189-8.

[43] A. Hinton, M. Kwiatkowska, G. Norman, and D. Parker. PRISM: A tool for automatic verification of probabilistic systems. In Holger Hermanns and Jens Palsberg, editors, Proc. 12th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'06), volume 3920 of LNCS, pages 441–444, Berlin, 2006. Springer.

[44] Fabio Somenzi. CUDD: CU Decision Diagram package. Public software, Colorado University, Boulder, 1997.

[45] Joost-Pieter Katoen, Marta Z. Kwiatkowska, Gethin Norman, and David Parker. Faster and symbolic CTMC model checking. In Luca de Alfaro and Stephen Gilmore, editors, Proc. 1st Joint International Workshop on Process Algebra and Probabilistic Methods, Performance Modeling and Verification (PAPM/PROBMIV'01), volume 2165 of LNCS, pages 23–38, Berlin, 2001. Springer.

[46] Joost-Pieter Katoen and Ivan S. Zapreev. Safe on-the-fly steady-state detection for timebounded reachability. In Third International Conference on the Quantitative Evaluation of Systems (QEST'06), pages 301–310, Washington D. C., 2006. IEEE Computer Society.

[47] Vidyadhar G. Kulkarni. Modeling and analysis of stochastic systems. Chapman & Hall, London, UK, 1995. ISBN 0-412-04991-0.

[48] Marta Z. Kwiatkowska, Gethin Norman, and David Parker. PRISM: Probabilistic symbolic model checker. In Tony Field, Peter G. Harrison, Jeremy T. Bradley, and Uli Harder, editors, Proc. 12th International Conference on Modelling Techniques and Tools for Computer Performance Evaluation (TOOLS'02), volume 2324 of LNCS, pages 200–204, Berlin, 2002. Springer.

[49] Marta Z. Kwiatkowska, Gethin Norman, and David Parker. PRISM 2.0: A tool for probabilistic model checking. In Proc. 1st International Conference on Quantitative Evaluation of Systems (QEST'04), pages 322–323, Washington D. C., 2004. IEEE Computer Society.

[50] Marta Z. Kwiatkowska, Gethin Norman, David Parker, and Jeremy Sproston. Performance analysis of probabilistic timed automata using Digital Clocks. In Kim Guldstrand Larsen and Peter Niebert, editors, 1st International Workshop on Formal Modeling and Analysis of Timed Systems (FORMATS'03), volume 2791 of LNCS, pages 105–120, Berlin, 2003. Springer.

[51] Kenneth L. McMillan. Symbolic Model Checking. Kluwer Academic, Dordrecht, NL, 1993.

ISBN 0-7923-9380-5.

[52] Jogesh K. Muppala and Kishor S. Trivedi. Numerical transient solution of finite markovian queueing systems. In U. Narayan Bhat and Ishwar V. Basawa, editors, Queueing and Related Models, pages 262–284, USA, 1992. Oxford University Press.

[53] James Norris. Markov chains. Cambridge series on statistical and probabilistic mathematics no. 2. Cambridge University Press, Cambridge, UK, 1997. ISBN 0-521-48181-3.

[54] Bernard Philippe, Youcef Saad, and William Stewart. Numerical methods in Markov chain modelling. Operations Research, 40(6):1156–1179, 1992.

[55] 李倩. 采用概率模型检测技术对无线传感网络聚簇协议的分析：[硕士学位论文]. 山东大学：计算机科学与技术学院，2012

[56] 韩东涛. 基于概率模型的基因组从头测序算法研究：[硕士学位论文]. 哈尔滨工业大学：计算机科学与技术学院，2012

[57] 张兵. 基于概率模型检测的生存性分析与验证方法研究：[硕士学位论文]. 郑州大学：信息工程学院，2012

[58] 文英，董荣胜. 一种高效的动态概率广播算法及其概率模型检测分析. 小型微型计算机系统. Vol 29 No 3 2008

致谢

感谢我的导师骆翔宇教授，骆老师是一名优秀的、经验丰富的教师，在他的指导下，我完成了我的学位论文。在此期间，他给我指定了好的方向，在我遇到问题时，及时给与帮助，同时对于在我完成工作后，又进一步对我的问题提出更深的疑问。这使得我不仅增长了知识，开阔了视野，并且也培养出一个良好的心态和科研精神。

还有要感谢我的同学许兴旺，感谢你在学习生活中给与我帮助，感谢你一直以来对我的支持、关心。在这里还要感谢华侨大学计算机学院的各位老师和我们全班同学们，与你们在一起生活的日子，让我受益良多。

最后，再次感谢骆翔宇老师和所有的老师和同学们，谢谢你们，祝福你们！

个人简历、在校期间发表的学术论文与研究结果

个人简介

刘来，男，1989 年 3 月 22 出生。2007.9-2011.6 就读于乐山师范学院计算机科学学院，并于 2011.6 获得工学学士学位。2011.9-2014.6 就读于华侨大学计算机科学与技术学院计算机应用技术专业，研究方向为模型检测。

攻读学位期间发表的学术论文

刘来, 骆翔宇. 一个分布式 k 互斥算法的概率模型检测[J]. 计算机应用研究. 2014. (录用)