中国科学院大学

University of Chinese Academy of Sciences

博士学位论文

基于模型检测和定理证明的形式化验证：基础理论与相关工具

作者姓名：刘坚

指导教师：蒋颖 研究员

中国科学院软件研究所

学位类别：工学博士

学科专业：计算机软件与理论

培养单位：中国科学院软件研究所

2018 年 03 月

Towards Combing Model Checking and Theorem Proving:

Theory and Related Tools

A dissertation submitted to the
University of Chinese Academy of Sciences
in partial fulfillment of the requirement
for the degree of
Doctor of Philosophy
in Computer Software and Theory

By

Jian Liu

Supervisor: Professor Ying Jiang

Institute of Software, Chinese Academy of Sciences

March, 2018

中国科学院大学
研究生学位论文原创性声明

本人郑重声明：所呈交的学位论文是本人在导师的指导下独立进行研究工作所取得的成果。尽我所知，除文中已经注明引用的内容外，本论文不包含任何其他个人或集体已经发表或撰写过的研究成果。对论文所涉及的研究工作做出贡献的其他个人和集体，均已在文中以明确方式标明或致谢。

作者签名：

日　　期：

中国科学院大学
学位论文授权使用声明

本人完全了解并同意遵守中国科学院有关保存和使用学位论文的规定，即中国科学院有权保留送交学位论文的副本，允许该论文被查阅，可以按照学术研究公开原则和保护知识产权的原则公布该论文的全部或部分内容，可以采用影印、缩印或其他复制手段保存、汇编本学位论文。

涉密及延迟公开的学位论文在解密或延迟期后适用本声明。

作者签名：　　导师签名：

日　　期：　　日　　期：

摘　要

模型检测和逻辑推演是目前用于形式化验证系统正确性的主要的两种方法。模型检测的优点是可以实现完全自动化，验证过程不需要人工干预，缺点是在处理大型系统或者无穷状态系统的过程中通常会面临状态爆炸问题；逻辑推演的优点是通常不关心系统的状态，从而避免了状态爆炸问题的产生，而缺点则是通常不能实现完全自动化，在验证的过程中通常需要人工干预。因此，如何结合模型检测和逻辑推演两种方法的优点来建立新的形式化验证方法是本文的主要研究内容。

本文的第一个工作是，建立一个逻辑系统 CTL_P，CTL_P 以 Kripke 模型为参数，对于一个给定的 Kripke 模型 $\mathcal{M}$，一个 $\mathsf{CTL}_P(\mathcal{M})$ 公式是有效的当且仅当这个公式在 M 中是满足的。相比于在 CTL 逻辑中只能讨论模型中当前状态的性质，在 CTL_P 中我们可以定义模型中的状态之间的关系，从而丰富了 CTL 逻辑的表达性；然后，我们根据逻辑 CTL_P 建立了一个证明系统 SCTL（Sequent-calculus-like proof system for CTL_P），并证明了 SCTL 系统的可靠性和完备性，使得一个公式在 SCTL 中是可证的当且仅当它在给定的模型中是满足的。

本文的第二个工作是，提出了一个 SCTL 证明系统的工具实现 SCTLProV，SCTLProV 既可以看作为定理证明器也可看作为模型检测工具：相比于定理证明器，SCTLProV 可以应用更多的优化策略，比如利用 BDD（Binary Decision Diagram）来存储状态集合从而减小内存占用；相比于模型检测工具，SCTLProV 的输出是完整的证明树，比模型检测工具的输出更丰富。

本文的第三个工作是，实现了一个 3D 证明可视化工具 VMDV（Visualization for Modeling, Demonstration, and Verification）。VMDV 目前可以完整的显示 SCTLProV 的证明树以及高亮显示 SCTLProV 的证明过程。同时 VMDV 是一个一般化的证明可视化工具，并提供了不同的接口用来与不同的定理证明器（比如 coq）协同工作。

关键词：模型检测，定理证明，工具实现

Abstract

This paper is a help documentation for the LaTeX class ucasthesis, which is a thesis template for the University of Chinese Academy of Sciences. The main content is about how to use the ucasthesis, as well as how to write thesis efficiently by using LaTeX.

Keywords: University of Chinese Academy of Sciences (UCAS), Thesis, LaTeX Template

目 录

图形列表

表格列表

第 1 章　定理证明工具 SCTLProV

在本章中，我们介绍对 SCTL 的证明搜索算法的一个编程实现—定理证明工具 SCTLProV（图1.1）以及该工具与其他模型检测工具的对比。SCTLProV 的工作方式如下：首先，SCTLProV 读入一个输入文件，并将该输入文件解析到一个 Kripke 模型以及若干个 SCTL 公式；然后，对于每个公式，SCTLProV 搜索该公式的证明，如果该公式可证，则并输出该证明（或者只输出 True），如果该公式不可证，则输出该公式的非的证明（或者只输出 False）。

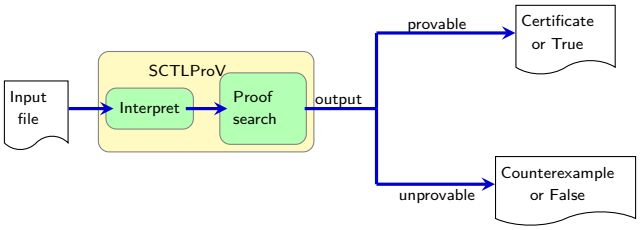

图 1.1 SCTLProV.

1.1　SCTLProV 的输入语言

本节讨论的是 SCTLProV 的输入语言。在 SCTLProV 中，输入语言的作用是定义 SCTLProV 的输入文件内容，即对计算机系统进行形式化建模，并定义要验证的性质，然后，SCTLProV 可在该模型上验证该性质。

在绝大多数传统的 CTL 模型检测工具的输入语言中，Kripke 模型中的每个状态通常用固定个数的值来表示，其中值的类型通常相对较简单，比如布尔或枚举类型等。这类输入语言通常无法将状态表示为复杂的数据结构，比如任意有限长度的链表。在 SCTLProV 的输入语言中，我们可以利用多种类型的表达式来定义 Kripke 模型的状态，比如元组（tuple）、记录（record）、变体（variant）或者任意有限长度的链表等。当用这种表达式来表示状态时，状态之间的迁移则可用一个从状态映射到它的所有后继的函数来表示，而且原子公式（即状态的性质或者状态之间的关系）则可用一个从一个或者多个状态映射到一个布尔值的函数来表示。此类输入语言通常可以用来对复杂的系统进行建模，比如第1.3.2节中的小型飞机场运输系统。下面我们详细介绍 SCTLProV 的输入语言。

1.1.1 词法标记与关键字

输入文件中通常包含的是有限个依次排列的字符，这些字符通过词法解析器的解析生成有限个词法标记。在词法标记中，一个整数用有限个连续的数字来表示，而一个标识符以字母开头，紧跟着的是有限个连续的字母、数字或者下划线字符。所有的词法标记用空格、制表符或者换行符隔开。关键字是一种特殊的词法标记，也叫做保留字，用来在输入文件中表示特殊的含义。SCTLProV 的输入语言中的关键字如下：

```
Model     Var     Define    Init   Transition  Atomic  Fairness   Spec
 int      bool     list     array     true      false     TRUE     FALSE
 not       AX       EX       AF        AF        EG       AR       EU
datatype  value  function    let      match     with      if       then
 else
```

在输入文件中关键字不能用来定义值或者函数的名字。其他的词法标记及用法在接下来的语法描述中具体描述出来。

1.1.2 类型

SCTLProV 的输入语言中有三种类型：基本类型、复合类型以及自定义类型。

基本类型 基本类型包含以下几种：

- `unit`: `unit` 类型的值只有一个，即 `()`。
- `int`: `int` 代表整数类型，一个值

1.2 其他 CTL 模型检测方法的对比

在这里，我们讨论 SCTLProV 的证明搜索算法与其他 CTL 模型检测方法的对比。

基于 BDD 的符号模型检测 当 Kripke 模型中绝大多数状态变量是布尔类型（比如在硬件模型检测问题中）的时候，BDD 的应用可以用来减少模型检测算法的空间占用。迄今为止，最好的基于 BDD 的符号模型检测工具是 NuSMV[1,2] 以及 NuSMV 的扩展 NuXMV[3]。下面我们举例说明基于 BDD 的符号模型检测方法的原理：假设存在一个以 s_0 为初始状态，T 为迁移规则的 Kripke 模型 $\mathcal{M}$。若要验证 $\mathcal{M}, s_0 \models EF\phi$，基于 BDD 的符号模型检测工具（例如 NuSMV）会首先

计算一个最小不动点 $\mathrm{lfp} = \mu Y.(\phi \vee EXY)$，然后，若 $s_0 \in \mathrm{lfp}$，则 $\mathcal{M}, s_0 \models EF\phi$ 成立，反之则不成立。计算不动点的过程中会不断地对迁移规则 T 进行展开直到得到不动点，其中 s_0 不可达的状态也可能被计算在不动点之内。

与基于 BDD 的符号模型检测工具不同，SCTLProV 在验证过程中没有必要计算不动点：迁移规则 T 是动态展开，即展开 T 直到可以判定公式是否可证为止。SCTLProV 的验证过程只访问初始状态 s_0 可达的状态，因此验证过程会节省空间的占用。不止如此，当 Kripke 模型的状态变量绝大多数为布尔类型的时候，SCTLProV 可以用 BDD 来记录访问过的状态，并以此来进一步节省空间占用；反之，当 Kripke 模型中包含多个非布尔类型的状态变量的时候，SCTLProV 可以选择直接记录（通常用哈希表）访问过的状态。不同于符号模型检测中将 Kripke 结构和要证明的性质都编码到 BDD 的做法，SCTLProV 在 Kripke 结构上直接做状态搜索，BDD 只被用来记录搜索过的状态。

即时（On-the-fly）模型检测 在验证时序逻辑公式的正确性时，利用即时模型检测的方法可以避免访问 Kripke 模型的整个状态空间，而只是访问由初始状态可达的状态集合。传统的 CTL 即时模型检测方法[4,5] 通常是基于递归的：即子公式的验证以及迁移规则的展开都是递归进行的。基于递归的 CTL 即时模型检测通常会涉及到大量的栈操作，尤其是在验证大型系统的过程中。验证算法通常会在栈操作上浪费大量的时间。

与传统的 CTL 即时模型检测方法不同，在 SCTLProV 中，公式和迁移规则均按需展开，而且验证算法基于连续（连续传递风格）而不是递归。基于连续的算法只需占用常数大小的栈空间[6–8]，而在递归算法中，栈空间的占用大小与递归深度成正比。

由于目前没有完整的基于传统的即时模型检测算法的工具存在，因此，为了将其与 SCTLProV 的证明搜索算法进行对比，我们开发了一个递归版本的 SCTLProV，即 SCTLProV$_R$[1]。不同于 SCTLProV，SCTLProV$_R$ 利用基于递归的算法来证明子公式并搜索状态空间。SCTLProV 与 SCTLProV$_R$ 的实验结果对比见第1.3节。

限界模型检测 在传统的限界模型检测工具中，若要证明一个时序逻辑公式，首先需要人为规定（或程序给定）一个限界，并将 Kripke 模型的迁移规则在限界

[1] https://github.com/terminatorlxj/SCTLProV_R

之内展开，然后判断该公式在迁移规则的有限步展开之内满足与否。若在当前的限界之内可以判断公式满足与否，则算法终止；否则，继续扩大限界。举例来说，若要判断 $\mathcal{M}, s_0 \models_{k+1} EF\phi$ 满足与否，则要先在限界 $k+1$ 之内展开公式与迁移规则[9]：

$$[\mathcal{M}, EF\phi]_{k+1} := \bigwedge_{i=0}^{k-1} T(s_i, s_{i+1}) \wedge \bigvee_{j=0}^{k} \phi(s_j)$$

与限界模型检测工具不同，SCTLProV 不需要限界展开迁移规则，而是在展开证明公式的同时按需展开迁移规则。例如在证明 $\vdash EF_x(\phi)(s_0)$ 的时候，公式和迁移规则的展开方式如下：

$$\mathsf{unfold}(S, \vdash EF_x(\phi)(s_i)) := \phi(s_i) \vee ((s_i \notin S) \wedge T(s_i, s_{i+1}) \wedge \mathsf{unfold}(S \cup \{s_i\}, \vdash EF_x(\phi)(s_{i+1})))$$

其中，S 是在证明过程中已经访问过的状态集合。

1.3 案例分析与实验结果

在本节中，我们首先讨论 SCTLProV 的两个应用案例：一个进程互斥问题和一个针对 NASA 提出的小型飞机场控制系统的形式化验证问题；然后，分别对比 SCTLProV 和其他 5 个工具在 5 个测试集上的实验结果。被用于与 SCTLProV 对比的 5 个工具分别为：基于 BDD 的符号模型检测工具 NuSMV 及 NuXMV，基于 QBF 的限界模型检测工具 Verds，基于消解（Resolution）的定理证明工具 iProver Modulo，以及形式验证工具包 CADP。本节所有工具的运行环境均为：Linux 操作系统，内存 3.0GB，2.93GHz×4 CPU；每个测试用例的最大运行时间限制为 20 分钟。本节所有的测试用例均可在因特网[2]下载。

1.3.1 案例一：进程互斥问题

例子 1.1 (进程互斥问题[10])**.** 本例讲的是关于两个进程（进程 A 和进程 B）的互斥问题。进程的互斥指的是在程序执行的任意时刻，最多只有一个进程在临界区内。一个关于此类问题的算法描述如图**??**所示，其中 $flag$ 是一个布尔变量，指的是当前是否有进程在运行，而 $mutex$ 是一个整型变量，指的是当前进入临界区的进程个数（初始值为 0）。如果在程序执行的某个时刻有多于 1 个进程进入临界区（即 $mutex = 2$），那么则称该程序违反了进程互斥性质。

[2] https://github.com/terminatorlxj/ctl_benchmarks

```
/* Process A */                 /* Process B */
1: while(flag);/*wait*/         1: while(flag);/*wait*/
2: flag = true;                 2: flag = true;
3: mutex ++;                    3: mutex ++;
/*critical section*/            /*critical section*/
4: mutex --;                    4: mutex --;
5: flag = false;                5: flag = false;
```

图 1.2 进程 A 和进程 B 的一个简单描述。

```
Model mutual()
{
Var {
flag : Bool; mutex : (0 .. 2); a : (1 .. 5); b : (1 .. 5);
}
Init {
flag := false; mutex := 0; a := 1; b := 1;
}
Transition {
a = 1 && flag = false : {a := 2;};
a = 2 : {a := 3; flag := true;};
a = 3 : {a := 4; mutex := mutex + 1;}; /*A has entered the critical section*/
a = 4 : {a := 5; mutex := mutex - 1;}; /*A has left the critical section*/
a = 5 : {flag := 0;};
b = 1 && flag = false : {b := 2;};
b = 2 : {b := 3; flag := true;};
b = 3 : {b := 4; mutex := mutex + 1;}; /*B has entered the critical section*/
b = 4 : {b := 5; mutex := mutex - 1;}; /*B has left the critical section*/
b = 5 : {flag := 0;};
/*If none of the conditions above are satisfied, then the current state goes to itself.*/
(a = 1 || b = 1) && flag = true: {}
}
Atomic {
bug(s) := s(mutex = 2);
}
Spec {
find_bug := EU(x, y, TRUE, bug(y), ini);
}
}
```

图 1.3 输入文件 “mutual.model”。

如图*1.3*所示，输入文件*"mutual.model"* 中变量 a 和变量 b 分别指代进程 A 和进程 B 的程序计数器 (*Program Counter*)，变量 ini 指代初始状态。本例中要验证的性质为：是否存在程序执行的某个时刻使得两个进程同时进入临界区。利用 *SCTLProV*，我们可以在计算机的命令行中输入以下命令来验证该性质。

```
sctl -output output.out mutual.model
```

SCTLProV 的运行结果如下所示：该程序存在漏洞，即该程序违反了进程互斥性质。

```
verifying on the model mutual...
find_bug: EU(x,y, TRUE, bug(y), ini)
find_bug is true.
```

该性质的证明树输出在文件*"output.out"*中，如下图所示。证明树的每个节点被输出为 $id : seqt\ [id_1, ..., id_n]$ 形式，其中 id 是该节点的 ID，$seqt$ 是当前的相继式，而 $id_1, ..., id_n$ 则是当前的相继式的所有的前提的 ID。

```
0: |- EU(x,y,TRUE,bug(y),{flag:=false;mutex:=0;a:=1;b:=1}) [4, 1]
4: {flag:=false;mutex:=0;a:=1;b:=1}
|- EU(x,y,TRUE,bug(y),{flag:=false;mutex:=0;a:=2;b:=1}) [7, 5]
1: |- TRUE []
7: {flag:=false;mutex:=0;a:=1;b:=1}
{flag:=false;mutex:=0;a:=2;b:=1}
|- EU(x,y,TRUE,bug(y),{flag:=false;mutex:=0;a:=2;b:=2}) [23, 20]
5: |- TRUE []
23:{flag:=false;mutex:=0;a:=1;b:=1}
{flag:=false;mutex:=0;a:=2;b:=1}
{flag:=false;mutex:=0;a:=2;b:=2}
|- EU(x,y,TRUE,bug(y),{flag:=true;mutex:=0;a:=3;b:=2}) [27, 24]
20: |- TRUE []
27:{flag:=false;mutex:=0;a:=1;b:=1}
{flag:=false;mutex:=0;a:=2;b:=1}
{flag:=false;mutex:=0;a:=2;b:=2}
{flag:=true;mutex:=0;a:=3;b:=2}
|- EU(x,y,TRUE,bug(y),{flag:=true;mutex:=1;a:=4;b:=2}) [31, 28]
24: |- TRUE []
31:{flag:=false;mutex:=0;a:=1;b:=1}
{flag:=false;mutex:=0;a:=2;b:=1}
{flag:=false;mutex:=0;a:=2;b:=2}
{flag:=true;mutex:=0;a:=3;b:=2}
{flag:=true;mutex:=1;a:=4;b:=2}
|- EU(x,y,TRUE,bug(y),{flag:=true;mutex:=1;a:=4;b:=3}) [35, 32]
28: |- TRUE []
35:{flag:=false;mutex:=0;a:=1;b:=1}
{flag:=false;mutex:=0;a:=2;b:=1}
{flag:=false;mutex:=0;a:=2;b:=2}
{flag:=true;mutex:=0;a:=3;b:=2}
{flag:=true;mutex:=1;a:=4;b:=2}
{flag:=true;mutex:=1;a:=4;b:=3}
|- EU(x,y,TRUE,bug(y),{flag:=true;mutex:=2;a:=4;b:=4}) [37]
32: |- TRUE []
37: |- bug({flag:=true;mutex:=2;a:=4;b:=4}) []
```

由以上的证明树输出可知，当进程 A 进入临界区之后，进程 B 同样进入临界区。

如图*1.4*所示，*SCTLProV* 验证该例子时的输出（证明树和 *Kripke* 模型）可由可视化工具 *VMDV*（*Visualization for Modeling, Demonstration, and Verification*）实现三维可视化显示。

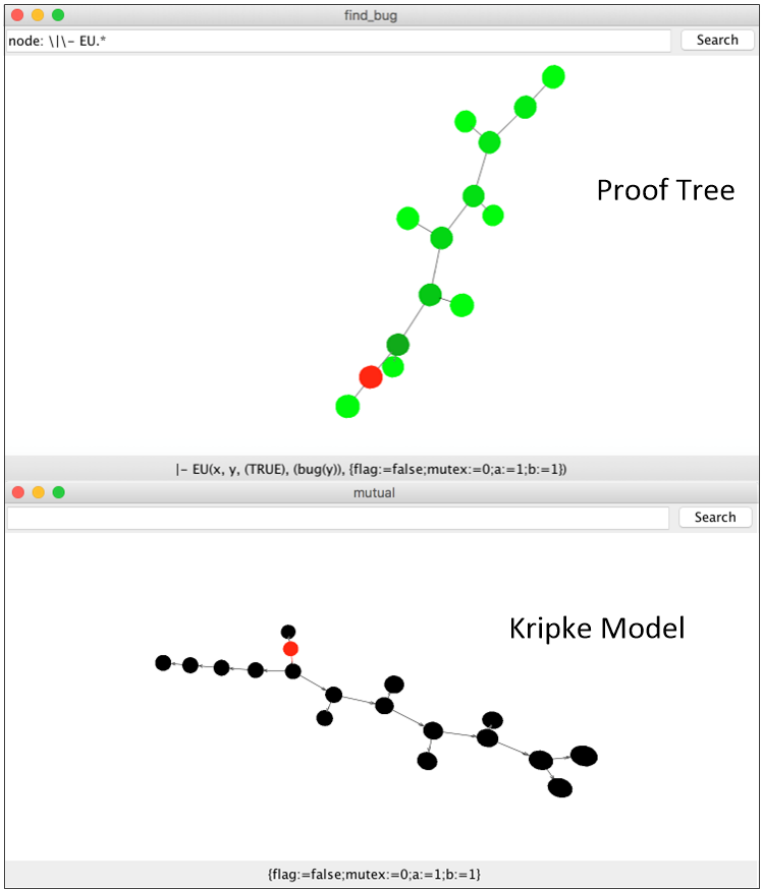

图 1.4 进程互斥问题的验证中证明树和模型的可视化

通过对原程序进行修改[10]，可以使程序满足进程互斥性质，修改后的程序如图*1.5*所示。修改后的程序可形式化描述为图*1.6*所示的输入文件。在此输入文件中，变量 x 和变量 y 均为布尔变量，分别指代当前状态下进程 A 和进程 B 是否在运行，而 $turn$ 则表示进程 A 和进程 B 轮流处在临界区内。

如下所示，修改后的程序满足进程互斥性质。

```
verifying on the model mutual...
find_bug: EU(x, y, TRUE, bug(y), ini)
find_bug is false.
```

1.3.2 案例二：小型飞机场运输系统

在本小节中，我们介绍对于一个工程问题的形式化验证：由美国国家航空与航天局（National Aeronautics and Space Administration，简称 NASA）为主导提出的小型飞机场运输系统（Small Aircraft Transportation System，简称

```
/* Process A */                  /* Process B */
1: x = true;                     1: y = true;
2: turn = 1;                     2: turn = 2;
3: while(y&&turn!=2); /*wait*/   3: while(x&&turn!=1); /*wait*/
4: mutex ++;                     4: mutex ++;
/*critical section*/             /*critical section*/
5: mutex --;                     5: mutex --;
6: x = false;                    6: y = false;
```

图 1.5 修改后的进程互斥程序。

```
Model mutual()
{
Var {
x:Bool; y:Bool; mutex:(0 .. 2); turn:(1 .. 2); a:(1 .. 6); b:(1 .. 6);
}
Init {
x := false; y := false; mutex := 0; turn := 1; a := 1; b := 1;
}
Transition {
a = 1 : {a := 2; x := true;};
a = 2 : {a := 3; turn := 1;};
a = 3 && (y = false || turn = 2): {a := 4;};
/*A has entered the critical section*/
a = 4 : {a := 5; mutex := mutex + 1;};
/*A has left the critical section*/
a = 5 : {a := 6; mutex := mutex - 1;};
a = 6 : {x := false;};
b = 1 : {b := 2; y := true;};
b = 2 : {b := 3; turn := 2;};
b = 3 && (x = false || turn = 1): {b := 4;};
/*B has entered the critical section*/
b = 4 : {b := 5; mutex := mutex + 1;};
/*B has left the critical section*/
b = 5 : {b := 6; mutex := mutex - 1;};
b = 6 : {y := false;};
/*If none of the conditions above are satisfied,
then the current state goes to itself.*/
(a != 3 && (y = true && turn = 1)) || (b != 3 && (x = true && turn = 2)) : {};
}
Atomic {
bug(s) := s(mutex = 2);
}
Spec {
find_bug := EU(x, y, TRUE, bug(y), ini);
}
}
```

图 1.6 输入文件 “mutual_solution.model”

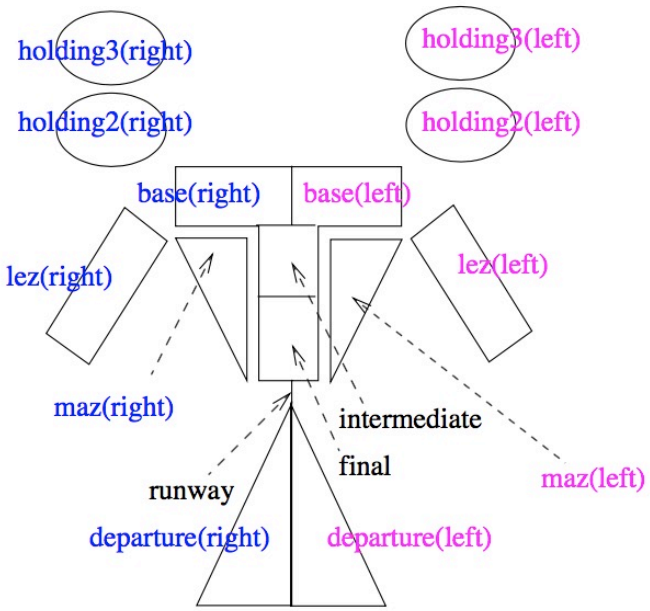

图 1.7 飞机场自我控制区域的划分（以飞行员视角区分左右）

SATS)[11,12]。在 SCTLProV 中，我们对 SATS 系统进行形式化描述，并验证该系统的安全性。

在 SATS 系统中，整个飞机场区域被称为自我控制区（Self Control Area，简称 SCA）。如图1.7所示，该模型将 SCA 被分为 15 个子区域：

- **holding3(right/left)**：等待航线，高度 3000 英尺（右/左）；
- **holding2(right/left)**：等待航线，高度 2000 英尺（右/左）；
- **lez(right/left)**：水平降落航线（右/左）；
- **base(right/left)**：基地航线（右/左）；
- **intermediate**：中间航线；
- **final**：最终航线；
- **runway**：机场跑道；
- **maz(right/left)**：重新降落航线（右/左）；
- **departure(right/left)**：起飞航线（右/左）。

在任意时刻，SCA 的每个子区域内都有若干架飞机，每个子区域内的飞机遵循先进先出的顺序依次进出。在整个 SCA 中，飞机的进出子区域的方式有 24 种，分别如下：

- 进入 3000 英尺高度等待航线（右/左）；
- 进入水平降落航线（右/左）；
- 从 3000 英尺等待航线进入 2000 英尺等待航线（右/左）；
- 从 2000 英尺等待航线进入基地航线（右/左）；
- 从水平降落航线进入基地航线（右/左）；
- 从基地航线（右/左）进入中间航线；
- 从中间航线进入重新降落航线（右/左）；
- 从中间航线进入最终航线；
- 准备降落，从最终航线进入机场跑道；
- 降落成功，飞机驶离机场跑道；
- 降落失败，从最终航线进入重新降落航线（右/左）；
- 从重新降落航线进入高度最低，而且可以进入（当前等待航线中没有飞机）的等待航线；
- 进入机场跑道并准备起飞；
- 从机场跑道进入起飞航线；
- 从起飞航线（右/左）离开，最终离开 SCA。

在任意时刻，飞机进出 SCA 的子区域的方式必须是安全的，即 SATS 系统必须满足 8 个性质：

- SCA 中不超过 4 架飞机同时准备降落：即等待航线，水平降落航线，重新降落航线，基地航线，中间航线，以及最终航线上飞机的总数不超过 4;

- SCA 中左右两侧的航线（等待航线，水平降落航线，以及重新降落航线）中，同侧的飞机数总和不超过 2，同时 SCA 中不超过 2 架飞机准备从同侧重新降落航线重新降落；
- 进入 SCA 的航线（等待航线和水平进入航线）中，每个航线每侧的飞机数不超过 2，同时基地航线的飞机数总和不超过 3；
- 在每侧的水平进入航线中最多只有一架飞机，同时如果某侧水平进入航线中有飞机，则 SCA 的同侧其他航线（水平降落航线以及重新降落航线）中没有飞机；
- SCA 中的飞机按照事先给定的顺序依次进入 SCA；
- 最先进入 SCA 中飞机一定最先降落；
- 机场跑道上最多只有一架飞机；
- 起飞航线上的飞机彼此必须相隔足够远的距离。

在 SCTLProV 中，我们针对 SCA 建立一个 Kripke 模型：模型中的状态用 15 个状态变量来表示，分别代表 SCA 中 15 个子区域，每个状态变量都是列表类型，代表该子区域内的若干架飞机；模型中包含 24 条迁移规则，分别指代 SCA 中的所有飞机在每个子区域间的 24 种进出方式；要验证的性质是一个 $\mathsf{CTL}_P(\mathcal{M})$ 公式，即在每个状态上，SCA 都满足安全性质。该模型的输入文件可在因特网[3]上下载。

SCTLProV 验证该模型用时 26 秒左右，运行环境为：Linux 操作系统，内存 3.0GB，2.93GHz×4 CPU。验证过程中共访问 54221 个状态（Dowek，Muñoz 和 Carreño 提出了对 SATS 系统的一个简化版的 PVS 建模[11]，在该模型中可访问的状态数为 2811）。经 SCTLProV 验证，该模型满足安全性性质。

值得注意的是，虽然这是一个典型的模型检测问题，但是传统的模型检测工具均无法验证该模型[11]，理由如下：

1. 模型的状态由复杂的数据结构所表示：每个状态变量的值均为列表类型，而且列表的长度可能为无穷。

[3] https://github.com/terminatorlxj/SATS-model

2. 状态的迁移规则必须由复杂的算法所描述：在某些状态迁移的过程中需要对飞机列表进行递归操作。

3. 模型的性质必须由复杂的算法所描述：某些原子命题的定义需要对飞机列表进行递归操作。

SCTLProV 的输入语言表达能力强于绝大多数模型检测工具，并能完整的表示该模型，同时成功进行验证。

1.3.3 随机生成的布尔程序的验证

本小节包含三个测试集：测试用例集一在首次提出[13] 时被用作对比限界模型检测工具 Verds 和符号模型检测工具 NuSMV 的性能；并紧接着被用作对比定理证明器 iProver Modulo 与 Verds 的性能；在测试集一的基础上，我们通过增大模型中的状态变量的个数而得到测试集二与测试集三。每个测试集中均包含 2880 个测试用例，每个测试用例的 Kripke 模型都是随机生成的，每个模型中的状态变量绝大多数为布尔类型。大量的随机的测试用例对于 SCTLProV 与不同的工具来说都是相对公平的，而且通过对比不同工具的实验结果数据，我们可以清晰的得出有关各个工具在验证不同的模型以及不同的性质时的优势与劣势的结论。

以下分别介绍这三个测试集。

1.3.3.1 测试集一

测试集一中包含两类测试用例：并发进程（Concurrent Processes，简称 CP）和并发顺序进程（Concurrent Sequential Processes，简称 CSP）。

并发进程 在描述并发进程需要用到以下 4 个变量：

a: 进程个数
b: 所有进程的共享变量和局部变量的个数
c: 进程间共享变量的个数
d: 每个进程的局部变量的个数

进程间的共享变量的初始值均为 $\{0,1\}$ 中的随机值，而每个进程的局部变量的初始值均为 0。每个进程的共享变量和局部变量的每次赋值均为随机选择的某个变量的值的逻辑非。我们令每个测试用例中进程个数为 3，即 $a = 3$；令 b 在

$\{12, 24, 36\}$ 中取值；同时令 $c = b/2$，以及 $d = c/a$。对于每个 b 的取值有 20 个 Kripke 模型，然后在每个 Kripke 模型分别验证 24 个 CTL 性质。因此，此测试集中共有 $3 \times 20 \times 24 = 1440$ 个并发进程测试用例。

并发顺序进程 在并发顺序进程测试用例中，除了以上定义的 a, b, c, d 变量之外，描述该类型测试用例还需用到以 2 个变量：

t: 每个进程的迁移的个数
p: 在每个迁移过程中同时进行的赋值的个数

除了在并发进程中介绍的 b 个布尔变量之外，在每个并发顺序进程中还用到一个局部变量来表示进程当前执行的位置，共有 c 个取值。进程间的共享变量的初始值均为 $\{0, 1\}$ 中的随机值，而每个进程的局部变量的初始值均为 0。每个进程共有 t 种迁移（状态变换，即对变量的赋值操作），在每个迁移种对随机选择的 p 个共享变量和局部变量进行赋值操作。随着进程的运行，所有的迁移依次周期性地进行。我们令每个测试用例包含 2 个进程，即 $a = 2$；令 b 在 $\{12, 16, 20\}$ 中取值；同时令 $c = b/2, d = c/a, t = c, p = 4$。对于每个 b 的取值有 20 个 Kripke 模型，然后在每个 Kripke 模型分别验证 24 个 CTL 性质。因此，此测试集中共有 $3 \times 20 \times 24 = 1440$ 个并发顺序进程测试用例。

在本测试集中，我们验证 24 个 CTL 性质，其中性质 P_{01} 至 P_{12} 如图1.8所示，而性质 P_{13} 至 P_{24} 为依次将 P_{01} 至 P_{12} 中的 $\wedge$ 替换成 $\vee$，以及将 $\vee$ 替换成 $\wedge$。

P_{01}	$AG(\bigvee_{i=1}^{c} v_i)$	P_{07}	$AU(v_1, AU(v_2, \bigvee_{i=3}^{c} v_i))$
P_{02}	$AF(\bigvee_{i=1}^{c} v_i)$	P_{08}	$AU(v_1, EU(v_2, \bigvee_{i=3}^{c} v_i))$
P_{03}	$AG(v_1 \Rightarrow AF(v_2 \wedge \bigvee_{i=3}^{c} v_i))$	P_{09}	$AU(v_1, AR(v_2, \bigvee_{i=3}^{c} v_i))$
P_{04}	$AG(v_1 \Rightarrow EF(v_2 \wedge \bigvee_{i=3}^{c} v_i))$	P_{10}	$AU(v_1, ER(v_2, \bigvee_{i=3}^{c} v_i))$
P_{05}	$EG(v_1 \Rightarrow AF(v_2 \wedge \bigvee_{i=3}^{c} v_i))$	P_{11}	$AR(AXv_1, AXAU(v_2, \bigvee_{i=3}^{c} v_i))$
P_{06}	$EG(v_1 \Rightarrow EF(v_2 \wedge \bigvee_{i=3}^{c} v_i))$	P_{12}	$AR(EXv_1, EXEU(v_2, \bigvee_{i=3}^{c} v_i))$

图 1.8 测试集一中需要验证的性质 P_{01} 至 P_{12}

1.3.3.2 测试集二

在测试集一的基础上，我们分别将并发进程测试用例中 b 的值分别扩大为 48、60、72、252、504、1008，将并发顺序进程测试用例中 b 的值分别扩大为 24、28、32、252、504、1008。由此，我们得到包含 5760 个新的测试用例的测试集

二。与测试集一一样，测试集二中的测试用例的模型的初始状态和迁移规则也是随机生成的。测试集二中要验证的性质与测试集一一致。

1.3.3.3 实验数据

在测试集一、二上，我们分别对比了 SCTLProV 与 iProver Modulo、Verds、NuSMV，以及 NuXMV 的实验结果。

测试集一的实验结果 由表1.1与表1.2可知：在测试集一的 2880 个测试用例中，iProver Modulo、Verds、NuSMV、NuXMV、SCTLProV 分别能验证 1816（63.1%）、2230（77.4%）、2880（100%）、2880（100%）、2862（99.4%）个测试用例；同时 SCTLProV 分别在 2823（98.2%）、2858（99.2%）、2741（95.2%）、2763（95.9%）个测试用例上占用时间和空间少于 iProver Modulo、Verds、NuSMV、NuXMV。各个工具的时间占用随着状态变量的个数的变化趋势如图1.9所示；各个工具占用空间随着状态变量的个数的变化趋势如图1.10所示。

程序类型	iProver Modulo	Verds	NuSMV	NuXMV	SCTLProV
CP ($b = 12$)	467(97.3%)	433(90.2%)	480(100%)	480(100%)	480(100%)
CP ($b = 24$)	372(77.5%)	428(89.2%)	480(100%)	480(100%)	480(100%)
CP ($b = 36$)	383(79.8%)	416(86.7%)	480(100%)	480(100%)	470(97.9%)
CSP ($b = 12$)	177(36.9%)	370(77.1%)	480(100%)	480(100%)	480(100%)
CSP ($b = 16$)	164(34.2%)	315(65.6%)	480(100%)	480(100%)	474(98.8%)
CSP ($b = 20$)	253(52.7%)	268(55.8%)	480(100%)	480(100%)	478(99.6%)
Sum	1816(63.1%)	2230(77.4%)	2880(100%)	2880(100%)	2862(99.4%)

表 1.1 测试集一中 5 个工具能成功验证的测试用例个数

程序类型	iProver Modulo	Verds	NuSMV	NuXMV
CP ($b = 12$)	480(100%)	480(100%)	430(89.6%)	431(89.8%)
CP ($b = 24$)	480(100%)	480(100%)	456(95.0%)	458(95.4%)
CP ($b = 36$)	454(94.6%)	467(97.3%)	441(91.9%)	446(92.9%)
CSP ($b = 12$)	480(100%)	480(100%)	464(96.7%)	465(96.9%)
CSP ($b = 16$)	474(98.6%)	473(98.5%)	472(98.3%)	474(98.6%)
CSP ($b = 20$)	455(94.8%)	478(99.6%)	478(99.6%)	479(99.8%)
Sum	2823(98.2%)	2858(99.2%)	2741(95.2%)	2763(95.9%)

表 1.2 测试集一中 SCTLProV 相比其他工具占用资源（时间和空间）少的测试用例个数

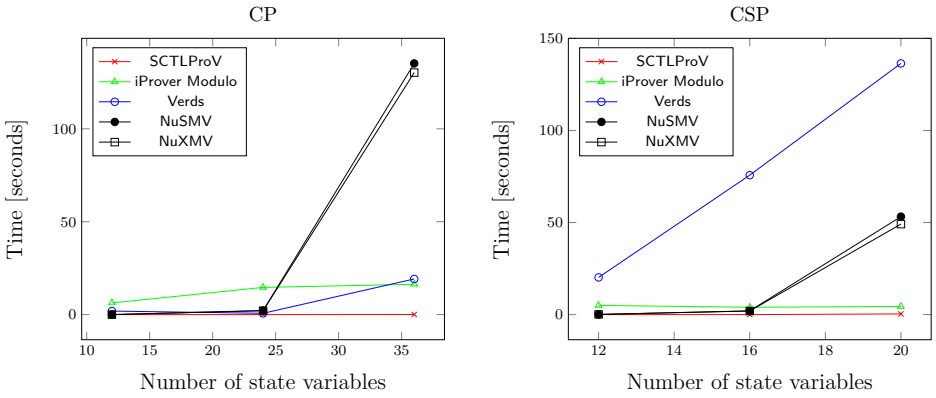

图 1.9 在测试集一上各个工具的平均占用时间

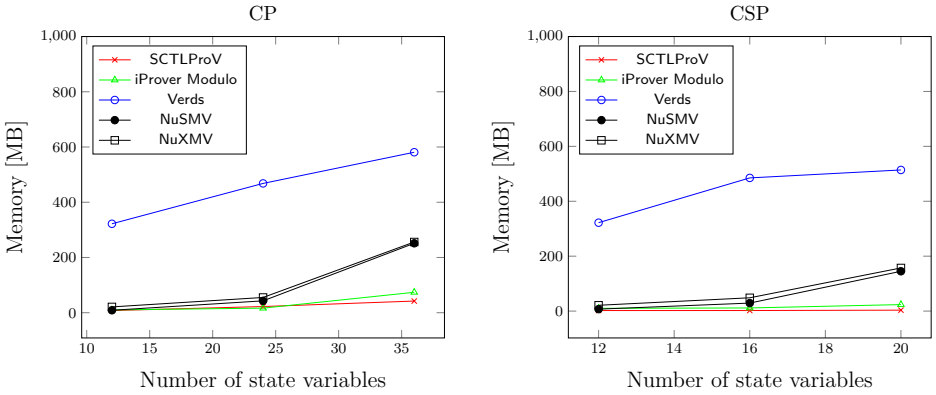

图 1.10 在测试集一上各个工具的平均占用内存

测试集二的实验结果 由表1.3与表1.4可知：在测试集二的 5760 个测试用例中，iProver Modulo、Verds、NuSMV、NuXMV、SCTLProV 分别能验证 2748（44.7%）、2226（38.6%）、728（12.6%）、736（12.8%）、4441（77.1%）个测试用例；同时 SCTLProV 分别在 4441（77.1%）、4438（77.0%）、4432（76.9%）、4432（76.9%）个测试用例上占用时间和空间少于 iProver Modulo、Verds、NuSMV、NuXMV。各个工具的时间占用随着状态变量的个数的变化趋势如图1.11所示；各个工具占用空间随着状态变量的个数的变化趋势如图1.12所示。

1.3.3.4 连续 vs. 递归

传统的即时模型检测工具通常使用递归算法来进行公式的证明和状态空间的搜索。不同于递归算法，SCTLProV 的验证算法是连续传递风格（Continuation-Passing Style，简称 CPS），CPS 的应用可以大大减少栈的操作，从而节省验证所需的时间。为了对比使用连续传递风格的算法和递归算法的效率，我们对比了 SCTLProV 和 SCTLProV$_R$ 分别在测试集一、二上的实验数据。其中 SCTLProV$_R$ 与 SCTLProV 的唯一不同是使用递归算法来证明公式和搜索状态空间。如表1.5所示，

程序类型	iProver Modulo	Verds	NuXMV	NuXMV	SCTLProV
CP ($b = 48$)	375(78.1%)	400(83.3%)	171(35.6%)	176(36.7%)	446(92.9%)
CP ($b = 60$)	360(75.0%)	403(84.0%)	22(4.6%)	23(4.8%)	440(91.7%)
CP ($b = 72$)	347(72.3%)	383(79.8%)	0	0	437(91.0%)
CP ($b = 252$)	299(62.3%)	216(45.0%)	0	0	371(77.3%)
CP ($b = 504$)	292(60.8%)	0	0	0	335(69.8%)
CP ($b = 1008$)	271(56.5%)	0	0	0	278(57.9%)
CSP ($b = 24$)	190(39.6%)	235(49.0%)	421(87.7%)	423(88.1%)	430(89.6%)
CSP ($b = 28$)	172(35.8%)	229(47.7%)	106(22.1%)	108(22.5%)	426(88.8%)
CSP ($b = 32$)	158(32.9%)	224(46.7%)	8(1.7%)	6(1.3%)	418(87.1%)
CSP ($b = 252$)	114(23.6%)	136(28.3%)	0	0	312(65.0%)
CSP ($b = 504$)	108(22.5%)	0	0	0	295(61.5%)
CSP ($b = 1008$)	62(12.9%)	0	0	0	253(52.7%)
Sum	2748(47.7%)	2226(38.6%)	728(12.6%)	736(12.8%)	4441(77.1%)

表 1.3 测试集二中 5 个工具能成功验证的测试用例个数

程序类型	iProver Modulo	Verds	NuSMV	NuXMV
CP ($b = 48$)	446(92.9%)	444(92.5%)	442(92.1%)	442(92.1%)
CP ($b = 60$)	440(91.7%)	440(91.7%)	440(91.7%)	440(91.7%)
CP ($b = 72$)	437(91.0%)	437(91.0%)	437(91.0%)	437(91.0%)
CP ($b = 252$)	371(77.3%)	371(77.3%)	371(77.3%)	371(77.3%)
CP ($b = 504$)	335(69.8%)	335(69.8%)	335(69.8%)	335(69.8%)
CP ($b = 1008$)	278(57.9%)	278(57.9%)	278(57.9%)	278(57.9%)
CSP ($b = 24$)	430(89.6%)	429(89.4%)	426(88.8%)	426(88.8%)
CSP ($b = 28$)	426(88.8%)	426(88.8%)	425(88.5%)	425(88.5%)
CSP ($b = 32$)	418(87.1%)	418(87.1%)	418(87.1%)	418(87.1%)
CSP ($b = 252$)	312(65.0%)	312(65.0%)	312(65.0%)	312(65.0%)
CSP ($b = 504$)	295(61.5%)	295(61.5%)	295(61.5%)	295(61.5%)
CSP ($b = 1008$)	253(52.7%)	253(52.7%)	253(52.7%)	253(52.7%)
Sum	4441(77.1%)	4438(77.0%)	4432(76.9%)	4432(76.9%)

表 1.4 测试集二中 SCTLProV 相比其他工具占用资源（时间和空间）少的测试用例个数

SCTLProV 能成功验证的测试用例个数比 SCTLProV$_R$ 多 10%，而且 SCTLProV 在绝大多数能成功运行的测试用例中比 SCTLProV$_R$ 所用时间短。如图1.13所示，随着状态变量数的增加，SCTLProV$_R$ 的平均运行时间多于 SCTLProV，而且时间的变化幅度更大。

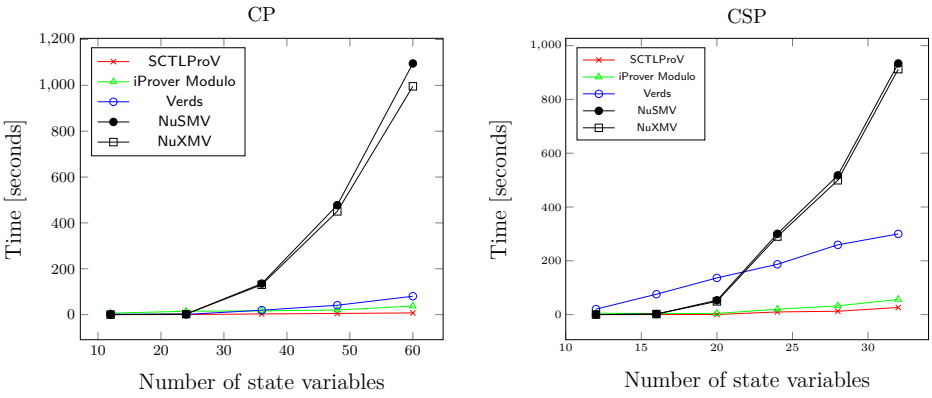

图 1.11 在测试集一、二上各个工具的平均占用时间

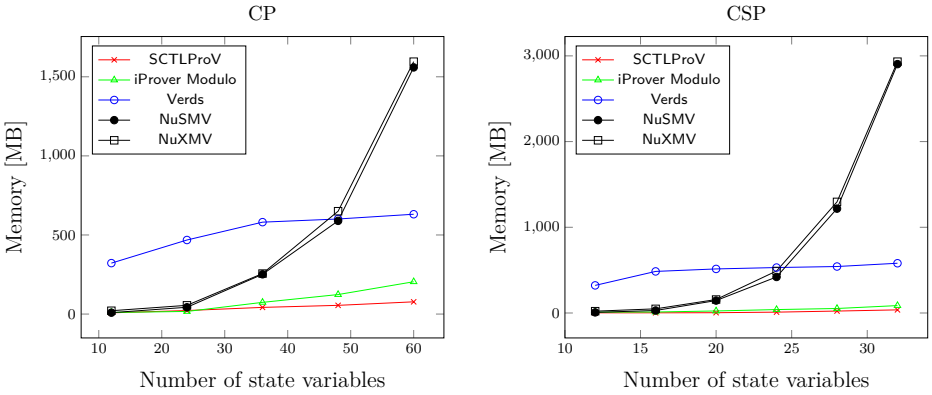

图 1.12 在测试集一、二上各个工具的平均占用内存

测试集	Solvable		t(SCTLProV) < t(SCTLProV$_R$)
	SCTLProV	SCTLProV$_R$	
一	2862(99.4%)	2682(93.1%)	2598(90.2%)
二	4446(77.2%)	3826(66.4%)	3841(71.9%)

表 1.5 在测试集一、二上 SCTLProV 和 SCTLProV$_R$ 的实验数据对比

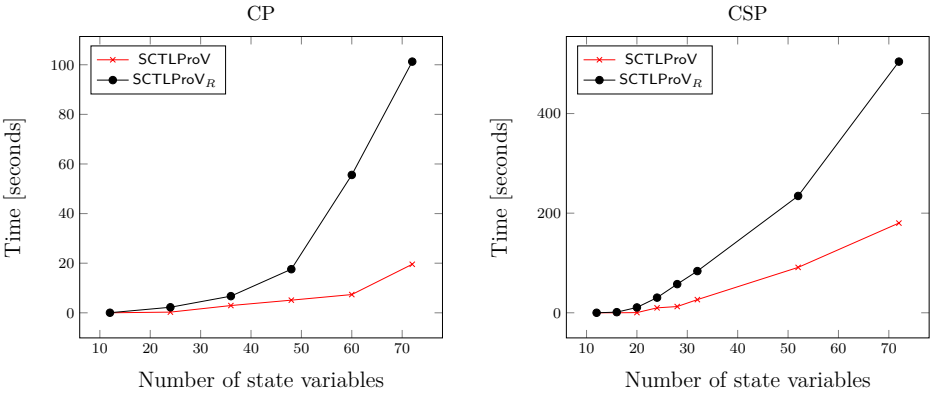

图 1.13 SCTLProV 和 SCTLProV$_R$ 平均运行时间

1.3.4 公平性性质的验证

在本小节，我们来对比 SCTLProV 和 Verds、NuSMV、NuXMV 在验证公平性性质时的效率。此次对比没有考虑 iProver Modulo，这是因为 iProver Modulo 无法验证公平性性质。此次对比所用的所有测试用例（测试集四）同样分为两种：互斥算法和环算法[4]。下面我们介绍这两种测试用例。

1.3.4.1 测试集三

测试集三种包含两类测试用例：互斥算法和环算法。

每个互斥算法包含 n 个进程，n 个进程的调度方式如下：对于 $0 \leq i \leq n-2$，进程 $i+1$ 的迁移在进程 i 之后，进程 0 的迁移在进程 $n-1$ 之后。互斥算法中 n 的取值范围为 $\{6, ..., 51\}$。要验证的性质如表1.14所示，其中 non_i，try_i 以及 cri_i 分别表示进程 i 的内部状态为 noncritical，trying 以及 critical。所有的性质必须在公平性的前提下进行验证，即在互斥算法的执行过程中没有进程饿死（永远处在等待状态）。

性质	互斥算法公式
P_1	$EF(cri_0 \wedge cri_1)$
P_2	$AG(try_0 \Rightarrow AF(cri_0))$
P_3	$AG(try_1 \Rightarrow AF(cri_1))$
P_4	$AG(cri_0 \Rightarrow Acri_0U(\neg cri_0 \wedge A\neg cri_0Ucri_1))$
P_5	$AG(cri_1 \Rightarrow Acri_1U(\neg cri_1 \wedge A\neg cri_1Ucri_0))$

图 1.14 互斥算法的性质

每个环算法包含 n 个进程，n 个进程的调度方式如下：对于 $1 \leq i \leq n-1$，进程 i 的内部状态由进程 $i-1$ 的输出决定，进程 0 的内部状态由进程 $n-1$ 的输出决定；每个进程的输出取决于它的内部状态；每个进程的内部状态由 5 个布尔变量的值表示；每个进程的输出由 1 个布尔变量表示。环算法中 n 的取值范围为 $\{3, ..., 10\}$。要验证的性质如表1.15所示，其中 out_i 表示进程 i 的输出为布尔值 true。所有的性质必须在公平性的前提下进行验证，即在环算法的执行过程中没有进程饿死（永远处在等待状态）。

测试集三的实验结果 如表1.6所示，在本测试集的 262 个测试用例中，Verds、NuSMV、NuXMV、SCTLProV 分别能解决 152 (58.0%)、71 (27.1%)、71 (27.1%)、211

[4] http://lcs.ios.ac.cn/~zwh/verds/verds_code/bp12.rar

性质	环算法公式
P_1	$AGAFout_0 \wedge AGAF\neg out_0$
P_2	$AGEFout_0 \wedge AGEF\neg out_0$
P_3	$EGAFout_0 \wedge EGAF\neg out_0$
P_4	$EGEFout_0 \wedge EGEF\neg out_0$

图 1.15 环算法的性质

(80.2%) 个测试用例。如表1.7所示，SCTLProV 分别在 200 (76.3%)、211 (80.2%)、211 (80.2%) 个测试用例上占用的时间和空间少于 Verds、NuSMV、NuXMV。测试集三上各个工具的详细实验数据如表1.8和表1.9所示。

Programs	Verds	NuSMV	NuXMV	SCTLProV
`mutual exclusion`	136 (59.1%)	50 (21.7%)	50 (21.7%)	191 (83.0%)
`ring`	16 (50.0%)	21 (65.6%)	21 (65.6%)	20 (62.5%)
Sum	152(58.0%)	71(27.1%)	71(27.1%)	211 (80.5%)

表 1.6 测试集三中各个工具能成功验证的测试用例的个数

Programs	Verds	NuSMV	NuXMV
`mutual exclusion`	187 (81.3%)	191 (83.0%)	191 (83.0%)
`ring`	13 (40.6%)	20 (62.5%)	20 (62.5%)
Sum	200(76.3%)	211(80.5%)	211(80.5%)

表 1.7 测试集三中 SCTLProV 占用资源更少的的测试用例的个数

1.3.5 工业级测试用例的验证

在本小节中，我们在工业级测试用例上（测试集四）对比 SCTLProV 与其他工具的效率。不同于测试集一、二、三，本次对比所用的测试用例均为符号迁移系统（Labeled Transition System，简称 LTS），而且所有的 LTS 均以 BCG（Binary-Coded Graph）格式表示，其中 BCG 格式可以用来表示较大的状态空间。测试集四是形式化验证工具包 CADP[14] 的一部分，也被称作 VLTS（Very Large Transition Systems）测试集。不同于本文中其他的形式化验证工具，CADP 是专门验证基于动作的形式化系统的工具，比如符号迁移系统、马尔可夫链等。测试集四中的例子均由对不同的传输协议以及并发系统的建模而得到的，其中许多例子是对工业级系统的建模[5]。

[5] http://cadp.inria.fr/resources/vlts/

Prop	**NoP**	Mutual Exclusion Algorithms							
		Verds		NuSMV		NuXMV		SCTLProV	
		sec	MB	sec	MB	sec	MB	sec	MB
P_1	6	0.286	321.99	0.153	9.07	0.270	21.18	0.005	2.25
	12	1.278	322.08	19.506	76.98	21.848	89.25	0.016	3.70
	18	4.719	426.45	-	-	-	-	0.037	5.44
	24	11.989	601.55	-	-	-	-	0.091	9.36
	30	26.511	926.25	-	-	-	-	0.200	16.49
	36	52.473	1287.57	-	-	-	-	0.418	27.46
	42	100.071	1944.95	-	-	-	-	0.682	48.28
	48	-	-	-	-	-	-	1.119	66.63
	51	-	-	-	-	-	-	1.392	82.32
P_2	6	0.375	322.07	0.054	9.07	0.048	21.31	0.012	3.07
	12	2.011	322.02	22.774	76.96	21.733	89.24	0.035	4.44
	18	7.958	446.71	-	-	-	-	0.101	8.09
	24	23.448	692.30	-	-	-	-	0.252	14.57
	30	48.800	1026.48	-	-	-	-	0.509	23.61
	36	105.183	1619.01	-	-	-	-	1.005	50.49
	42	-	-	-	-	-	-	1.791	57.93
	48	-	-	-	-	-	-	2.679	86.67
	51	-	-	-	-	-	-	3.453	129.83
P_3	6	0.331	322.02	0.089	9.04	0.033	21.27	0.012	3.03
	12	2.059	322.07	22.749	76.91	21.897	89.22	0.035	4.93
	18	7.995	449.13	-	-	-	-	0.110	9.59
	24	23.578	696.74	-	-	-	-	0.286	21.04
	30	51.774	1138.27	-	-	-	-	0.643	30.09
	36	106.027	1628.84	-	-	-	-	1.287	66.14
	42	-	-	-	-	-	-	2.138	86.29
	48	-	-	-	-	-	-	3.369	170.94
	51	-	-	-	-	-	-	4.333	149.03
P_4	6	0.446	321.97	0.089	9.04	0.033	21.27	0.039	3.38
	12	8.289	552.62	22.749	76.91	21.897	89.22	150.115	986.64
	18	-	-	-	-	-	-	-	-
	24	-	-	-	-	-	-	-	-
	30	-	-	-	-	-	-	-	-
	36	-	-	-	-	-	-	-	-
	42	-	-	-	-	-	-	-	-
	48	-	-	-	-	-	-	-	-
	51	-	-	-	-	-	-	-	-
P_5	6	0.430	322.03	0.031	9.09	0.047	21.19	0.011	3.10
	12	3.398	363.78	22.747	77.01	22.029	89.17	0.040	4.81
	18	18.176	783.24	-	-	-	-	0.115	10.99
	24	87.432	2382.82	-	-	-	-	0.322	18.68
	30	-	-	-	-	-	-	1.414	47.68
	36	-	-	-	-	-	-	1.287	66.35
	42	-	-	-	-	-	-	2.405	142.86
	48	-	-	-	-	-	-	4.848	225.55
	51	-	-	-	-	-	-	5.177	225.66

表 1.8 测试集三中互斥算法测试用例的实验数据

测试集四中共有 40 个测试用例，针对每个测试用例我们分别验证有无死锁与活锁。由于 SCTLProV 是基于 Kripke 模型的验证工具，因此在验证之前，我们需将每个 LTS 转换到相应的 Kripke 模型，然后在转换后的 Kripke 模型中进行验证。

给定一个 LTS $\mathcal{L} = \langle s_0, S, Act, \rightarrow \rangle$，其中 s_0 是初始状态；s 是一个有穷的状

Prop	**NoP**	Ring Algorithms							
		Verds		NuSMV		NuXMV		SCTLProV	
		sec	MB	sec	MB	sec	MB	sec	MB
P_1	3	0.168	322.09	0.040	10.02	0.045	22.08	4.622	62.22
	4	0.216	322.12	0.299	22.46	0.255	34.96	-	-
	5	0.301	322.07	2.421	59.31	1.195	71.53	-	-
	6	0.449	322.13	22.127	80.49	17.967	92.82	-	-
	7	0.740	322.19	147.895	224.17	131.735	236.50	-	-
	8	1.115	322.09	1135.882	865.04	1083.48	877.36	-	-
	9	1.646	322.07	-	-	-	-	-	-
	10	2.232	321.96	-	-	-	-	-	-
P_2	3	-	-	0.058	10.74	0.068	22.73	0.031	3.22
	4	-	-	0.583	40.29	0.562	52.61	0.125	3.73
	5	-	-	5.164	62.29	5.295	74.62	0.444	4.05
	6	-	-	39.085	81.85	37.969	93.96	1.373	4.71
	7	-	-	246.123	229.07	241.375	241.15	3.745	6.03
	8	-	-	-	-	-	-	9.154	7.61
	9	-	-	-	-	-	-	19.997	10.07
	10	-	-	-	-	-	-	40.331	13.05
P_3	3	-	-	0.045	10.03	0.071	22.32	0.022	3.20
	4	-	-	0.296	22.46	0.299	34.96	0.820	13.11
	5	-	-	2.357	59.31	2.526	71.63	111.96	676.29
	6	-	-	22.147	80.49	21.304	92.93	-	-
	7	-	-	147.567	224.17	141.134	236.74	-	-
	8	-	-	-	-	-	-	-	-
	9	-	-	-	-	-	-	-	-
	10	-	-	-	-	-	-	-	-
P_4	3	0.158	322.09	0.066	10.00	0.171	22.32	0.024	3.24
	4	0.190	322.05	0.356	22.46	0.367	34.95	0.104	3.82
	5	0.263	322.04	2.726	59.31	2.781	71.63	0.385	3.99
	6	0.385	322.07	27.013	80.48	24.794	94.95	1.289	4.57
	7	0.528	322.07	181.007	224.16	166.725	236.61	3.727	5.29
	8	0.815	322.14	-	-	-	-	9.525	7.14
	9	1.138	322.19	-	-	-	-	21.568	9.31
	10	1.574	321.98	-	-	-	-	45.097	12.95

表 1.9 测试集三中环算法测试用例的实验数据

态集合；Act 是一个有穷的动作集合；$\rightarrow \subseteq S \times Act \times S$ 是迁移规则。那么 $\mathcal{L}$ 到相应的 Kripke 模型 $\mathcal{M} = \langle s_0', S', \longrightarrow, \mathcal{P} \rangle$ 的转换过程如下：

- 令 s_0' 为 $(s_0, \cdot)$，其中 $\cdot \notin Act$ 是一个特殊的动作符号。
- 分别将 $(s_d, \cdot)$ 与 S' 与 $(s_d, \cdot) \longrightarrow (s_d, \cdot)$ 添加到 S' 与 $\longrightarrow$ 中，其中 $(s_d, \cdot)$ 区分于 S' 中的所有其他状态。
- 重复以下步骤直到没有更多的状态和迁移被添加到 $\mathcal{M}$ 中：
  - 如果 $\mathcal{L}$ 中存在一个迁移 $s_1 \xrightarrow{a} s_2$，那么对于所有的 $(s_1, b) \in S'$，将 $(s_1, b) \longrightarrow (s_2, a)$ 添加到 $\mathcal{M}$ 的迁移关系 $\longrightarrow$ 中，其中 $a, b \in Act \cup \{\cdot\}$；同时将 (s_2, a) 添加到 $\mathcal{M}$ 的状态集合 S' 中；

  - 对于 S' 中的状态 (s, a)，如果 s 在 $\mathcal{L}$ 中没有后继，那么将 $(s, a) \longrightarrow (s_d, \cdot)$ 添加到 $\mathcal{M}$ 的迁移关系 $\longrightarrow$。

- 最后，令 $P = \{(s_d, \cdot)\}$ 而且 $Q = \{(s, a) \mid s \in S \wedge a = \tau\}$。

经过从 LTS 到 Kripke 模型的转换之后，我们就可以在 SCTLProV 中验证死锁与活锁的存在了。

死锁 在 LTS 中，死锁状态指的是没有后继的状态。当验证一个 LTS $\mathcal{L}$ 中是否存在可达的死锁状态时，我们首先将 $\mathcal{L}$ 经过以上的转换方法转换到一个 Kripke 模型 $\mathcal{M}$。经过观察得知，$\mathcal{L}$ 中存在一个可达的死锁状态当且仅当 $\mathcal{M}$ 中状态 $(s_d, \cdot)$ 时可达的。

然后，我们可以通过证明如下的公式来验证 $\mathcal{M}$ 中 $(s_d, \cdot)$ 是否可达：

$$EF_x(P(x))((s_0, \cdot))$$

这个公式是可证的当且仅当 $\mathcal{M}$ 中存在一条形式为 $(s_0, \cdot) \longrightarrow^* (s, a) \longrightarrow (s_d, \cdot)$ 的路径，其中 a 是一个动作符号，而且 s 在 $\mathcal{L}$ 中没有后继。因此，这个公式可以被用来验证 $\mathcal{L}$ 中有没有可达的死锁状态。

活锁 在 LTS 中，活锁指的是所有动作均为 τ 的无穷的环状路径。在 LTS 中检测活锁相比检测死锁更复杂，原因是当观察一个迁移的时候，状态和动作都要考察到。与在检测死锁的方法一样，当验证一个 LTS $\mathcal{L}$ 中是否存在可达的活锁时，我们首先将 $\mathcal{L}$ 经过以上的转换方法转换到一个 Kripke 模型 $\mathcal{M}$。通过以下分析可知，$\mathcal{L}$ 中存在一个可达的活锁当且仅当 $\mathcal{M}$ 中存在一个环状路径，而且该路径上的所有状态均满足 Q。

- ($\Rightarrow$) 如果 $\mathcal{L}$ 中存在一个可达的环状路径，那么这个环状路径具有 $s_p \xrightarrow{\tau} s_{p+1} \xrightarrow{\tau} \cdots \xrightarrow{\tau} s_n \xrightarrow{\tau} s_p$ 形式，其中 $s_p = s_0$，或者存在一个从 s_0 到 s_p 的路径 $s_0 \xrightarrow{a_0} \cdots \xrightarrow{a_{p-1}} s_p$。那么根据由 $\mathcal{L}$ 到 $\mathcal{M}$ 的转换方法可知，$\mathcal{M}$ 中一定存在一个环状路径 $(s_p, a) \longrightarrow (s_{p+1}, \tau) \longrightarrow \cdots \longrightarrow (s_n, \tau) \longrightarrow (s_p, \tau) \longrightarrow (s_{p+1}, \tau)$，其中 $a = a_{p-1}$，而且 $(s_0, \cdot) \longrightarrow^* (s_p, a)$。

- ($\Leftarrow$) 如果 $\mathcal{M}$ 中存在一个具有 $(s_p, \tau) \longrightarrow (s_{p+1}, \tau) \longrightarrow \cdots \longrightarrow (s_n, \tau) \longrightarrow (s_p, \tau)$ 形式的环状路径，而且其中对于此路径中的某个状态 (s_m, τ)，有

Name	**Deadlocks**	SCTLProV		CADP	
		sec	MB	sec	MB
vasy_0_1	No	0.13	27.16	0.40	10.95
cwi_1_2	No	0.13	27.67	0.39	10.80
vasy_1_4	No	0.14	27.75	0.39	10.71
cwi_3_14	Yes	0.14	25.70	0.40	10.82
vasy_5_9	Yes	0.14	25.70	0.40	10.82
vasy_8_24	No	0.17	28.90	0.43	10.79
vasy_8_38	Yes	0.14	25.76	0.39	10.74
vasy_10_56	No	0.20	29.90	0.43	10.86
vasy_18_73	No	0.24	31.68	0.47	11.81
vasy_25_25	Yes	0.97	33.52	2.18	23.26
vasy_40_60	No	0.21	29.42	0.46	15.08
vasy_52_318	No	0.59	41.09	0.65	16.69
vasy_65_2621	No	1.41	77.02	2.09	109.03
vasy_66_1302	No	0.89	34.92	1.25	14.13
vasy_69_520	Yes	0.23	27.47	0.51	11.84
vasy_83_325	Yes	0.21	27.96	0.48	11.32
vasy_116_368	No	0.67	35.27	0.77	14.40
cwi_142_925	Yes	0.28	28.33	0.57	12.72
vasy_157_297	Yes	0.18	27.14	0.45	11.48
vasy_164_1619	No	2.53	48.39	1.53	22.90
vasy_166_651	Yes	0.29	31.19	0.55	13.30
cwi_214_684	Yes	0.39	34.39	0.63	22.94
cwi_371_641	No	1.36	40.41	1.24	42.92
vasy_386_1171	No	2.14	74.11	1.66	45.12
cwi_566_3984	Yes	0.78	38.53	1.11	21.92
vasy_574_13561	No	18.23	246.97	9.72	188.21
vasy_720_390	Yes	0.23	28.49	0.48	12.89
vasy_1112_5290	No	10.2	89.81	6.54	97.47
cwi_2165_8723	No	16.51	166.74	14.55	185.58
cwi_2416_17605	Yes	3.19	87.61	3.38	71.80
vasy_2581_11442	Yes	2.40	74.11	2.68	58.43
vasy_4220_13944	Yes	2.85	89.50	3.20	73.82
vasy_4338_15666	Yes	3.41	96.21	3.83	80.59
vasy_6020_19353	No	37.19	456.34	74.24	649.41
vasy_6120_11031	Yes	2.35	82.57	2.60	67.01
cwi_7838_59101	No	72.76	1013.67	140.21	1019.55
vasy_8082_42933	No	7.85	309.74	7.82	240.69
vasy_11026_24660	Yes	4.82	149.80	5.15	134.17
vasy_12323_27667	Yes	5.40	164.73	5.67	149.09
cwi_33949_165318	No	366.51	2368.22	636.39	2972.61

表 1.10 SCTLProV 与 CADP 分别验证测试集四中测试用例的死锁性质的实验数据

$(s_0, \cdot) \longrightarrow^* (s_m, \tau)$，那么在 $\mathcal{L}$ 中存在一个环状路径 $s_p \xrightarrow{\tau} \cdots \xrightarrow{\tau} s_m \xrightarrow{\tau} \cdots \xrightarrow{\tau} s_p$，其中此路径是从 s_0 状态可达的。

然后，我们可以通过在 $\mathcal{M}$ 中证明如下的公式来验证 $\mathcal{L}$ 中是否有可达的活锁：

$$EF_x(EG_y(Q(y))(x))((s_0, \cdot))$$

这个公式是可证的当且仅当 $\mathcal{L}$ 中存在一个可达的活锁。

Name	Livelocks	SCTLProV		CADP	
		sec	MB	sec	MB
vasy_0_1	No	0.13	27.45	0.48	14.57
cwi_1_2	No	0.14	27.74	0.50	14.61
vasy_1_4	No	0.14	27.70	0.48	14.66
cwi_3_14	No	0.16	28.18	0.50	14.57
vasy_5_9	No	0.16	28.08	0.51	14.68
vasy_8_24	No	0.18	29.09	0.53	14.75
vasy_8_38	No	0.20	28.86	0.52	14.73
vasy_10_56	No	0.23	30.33	0.55	15.34
vasy_18_73	No	0.28	33.07	0.56	16.25
vasy_25_25	No	0.96	33.54	2.25	27.84
vasy_40_60	No	0.26	30.23	0.57	19.65
vasy_52_318	Yes	0.21	27.66	0.55	15.45
vasy_65_2621	No	5.31	280.57	1.98	113.40
vasy_66_1302	No	2.40	38.33	1.30	18.50
vasy_69_520	No	1.04	33.41	0.85	16.39
vasy_83_325	No	0.83	39.27	0.76	19.40
vasy_116_368	No	1.19	42.14	0.86	15.74
cwi_142_925	No	2.67	46.30	1.22	17.89
vasy_157_297	No	0.80	33.99	0.78	17.91
vasy_164_1619	No	3.69	53.30	1.51	23.44
vasy_166_651	No	1.60	49.98	1.02	26.02
cwi_214_684	Yes	0.26	29.93	0.63	16.81
cwi_371_641	Yes	0.26	30.63	0.62	17.41
vasy_386_1171	No	2.91	80.16	1.55	41.75
cwi_566_3984	No	13.32	106.25	3.95	54.08
vasy_574_13561	No	27.02	272.11	8.17	188.69
vasy_720_390	No	0.86	31.45	0.76	17.45
vasy_1112_5290	No	10.49	89.86	5.24	97.93
cwi_2165_8723	Yes	1.87	61.94	2.15	48.80
cwi_2416_17605	Yes	3.10	87.61	3.44	76.30
vasy_2581_11442	No	32.70	326.38	14.78	214.93
vasy_4220_13944	No	43.93	423.03	24.71	330.85
vasy_4338_15666	No	47.55	479.15	28.06	344.64
vasy_6020_19353	Yes	3.23	100.24	3.57	106.43
vasy_6120_11031	No	30.59	425.64	38.37	437.71
cwi_7838_59101	Yes	11.34	250.68	11.58	236.09
vasy_8082_42933	No	119.77	1123.85	106.49	908.29
vasy_11026_24660	No	60.86	698.85	108.97	804.34
vasy_12323_27667	No	68.50	793.83	134.44	898.61
cwi_33949_165318	Yes	33.89	732.05	34.60	738.78

表 1.11 SCTLProV 与 CADP 分别验证测试集四中测试用例的活锁性质的实验数据

测试集四的实验结果 我们用 SCTLProV 与 CADP 分别验证了测试集四中所有测试用例。如表1.12所示，在 40 个测试用例中，SCTLProV 和 CADP 均能成功验证所有的例子。在验证死锁性质时，SCTLProV 在 33 (82.5%) 个测试用例中用时比 CADP 短；在 7 (17.5%) 个测试用例中占用内存比 CADP 少。在验证活锁性质时，SCTLProV 在 22 (55.0%) 个测试用例中用时比 CADP 短；在 6 (15.0%) 个测试用例中占用内存比 CADP 少。测试集四的详细实验数据见表1.10和表1.11。

性质	t(SCTLProV) < t(CADP)	m(SCTLProV) < m(CADP)
死锁	33 (82.5%)	7 (17.5%)
活锁	22 (55.0%)	6 (15.0%)

表 1.12 测试集四中 SCTLProV 相比 CADP 用时短以及占用内存少的测试用例的个数

1.3.6 关于实验结果的讨论

由测试集一、二、三的实验结果可知，NuSMV、NuXMV、Verds、iProver Modulo、SCTLProV 的实验数据主要受两方面因素影响：状态变量的个数和要验证的公式的类型。其中，NuSMV、NuXMV 的实验数据主要受状态变量个数的影响，而 Verds、iProver Modulo、SCTLProV 的实验数据主要受公式类型的影响。当状态变量的个数较小的时候（比如测试集一），NuSMV 和 NuXMV 的表现往往优于 Verds，iProver Modulo 以及 SCTLProV；然而，在状态变量数较大的测试集中（比如测试集二、三），Verds、iProver Modulo、SCTLProV 的表现更好。如果在验证公式的过程中需要访问到几乎整个状态空间（比如一些 AG 性质），那么 NuSMV 和 NuXMV 的表现优于 Verds，iProver Modulo 以及 SCTLProV；然而，在验证其他公式的时候，Verds，iProver Modulo、SCTLProV 的表现更好，这是因为在验证此类性质的时候 Verds，iProver Modulo 以及 SCTLProV 不必访问整个状态空间。因此，Verds，iProver Modulo、SCTLProV 相比 NuSMV、NuXMV 在状态变量个数上扩展性更好，其中 SCTLProV 在状态变量个数上比 Verds 和 iProver Modulo 扩展性更好，而且在几乎所有能验证的例子上比 Verds 和 iProver Modulo 占用资源更少。

由测试集四的实验结果可知，SCTLProV 和 CADP 的实验数据也受两方面因素影响：状态空间的大小以及是否满足要验证的性质。当状态空间较小时，SCTLProV 和 CADP 均占用资源较少；而且当要验证的性质（死锁和活锁）满足的时候，SCTLProV 和 CADP 的占用资源也较少。同时，在超过一半的例子中，SCTLProV 用时相比 CADP 更少。另外，值得注意的是，在验证测试集四种的测试用例时，SCTLProV 需要将 LTS 转换成 Kripke 模型，而在转换的过程中往往状态空间也会增大，因此在某些测试用例中 SCTLProV 的空间占用比 CADP 大。如果 SCTLProV 能直接在 LTS 上进行验证，那么则可避免增大状态空间，这是本文的下一步工作。

参考文献

[1] MCMILLAN K L. Symbolic model checking[M]. USA: Springer, 1993.

[2] CIMATTI A, CLARKE E M, GIUNCHIGLIA F, et al. Nusmv: A new symbolic model verifier[C]//Proceedings of CAV'99. Trento, Italy: Springer-Verlag, Berlin, 1999: 495–499.

[3] CAVADA R, CIMATTI A, DORIGATTI M, et al. The nuxmv symbolic model checker[C]//Proceedings of Computer Aided Verification - 26th International Conference, CAV 2014. Vienna, Austria: Springer International Publishing, Switzerland, 2014: 334–342.

[4] VERGAUWEN B, LEWI J. A linear local model checking algorithm for CTL [C]//Proceedings of CONCUR '93, 4th International Conference on Concurrency Theory. Hildesheim, Germany: Springer-Verlag, Berlin, 1993: 447–461.

[5] BHAT G, CLEAVELAND R, GRUMBERG O. Efficient on-the-fly model checking for ctl^*[C]//Proceedings of LICS'95. San Diego, California, USA: IEEE Computer Society, USA, 1995: 388–397.

[6] REYNOLDS J C. The discoveries of continuations[J]. Lisp and Symbolic Computation, 1993, 6(3-4): 233–248.

[7] APPEL A W. Compiling with continuations (corr. version)[M]. UK: Cambridge University Press, 2006.

[8] SESTOFT P. Undergraduate topics in computer science: volume 50 programming language concepts[M]. Switzerland: Springer International Publishing, 2012.

[9] BIERE A, CIMATTI A, CLARKE E, et al. Symbolic model checking without BDDs[C]//CLEAVELAND W R. LNCS: volume 1579 Proceedings of TACAS'99. Amsterdam, the Netherlands: Springer, USA, 1999: 193–207.

[10] PETERSON G L. Myths about the mutual exclusion problem[J]. Inf. Process. Lett., 1981, 12(3): 115–116.

[11] MUÑOZ C A, DOWEK G, CARREÑO V. Modeling and verification of an air traffic concept of operations[C]//Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2004. Boston, Massachusetts, USA: ACM, USA, 2004: 175–182.

[12] NASA/TM-2004-213006. Abstract model of sats concept of operations: Initial results and recommendations[M]. USA: NASA, 2004.

[13] ZHANG W. QBF Encoding of Temporal Properties and QBF-based Verification [C]//Proceedings of IJCAR 2014. Vienna: Springer-Verlag, Berlin, 2014: 224–239.

[14] GARAVEL H, LANG F, MATEESCU R, et al. CADP 2011: a toolbox for the construction and analysis of distributed processes[J]. STTT, 2013, 15(2): 89–107.

发表学术论文

学术论文

[1] Jian Liu, Ying Jiang, Yanyun Chen. VMDV: A 3D Visualization Tool for Modeling, Demonstration, and Verification. TASE 2017. accepted.

[2] Ying Jiang, Jian Liu, Gilles Dowek, Kailiang Ji. SCTL: Towards Combining Model Checking and Proof Checking. The Computer Journal. submitted.

项目资助情况

1. 中法合作项目 VIP（项目编号 GJHZ1844）

2. 中法合作项目 LOCALI (项目编号 NSFC 61161130530 和 ANR 11 IS02 002 01)

简　历

基本情况

刘坚，男，山东省茌平县人，1989 年出生，中国科学院软件研究所博士研究生。

教育背景

- 2011 年 9 月至今：中国科学院软件研究所，计算机软件与理论，硕博连读
- 2007 年 9 月至 2011 年 7 月：山东农业大学，信息与计算科学，本科

联系方式

通讯地址：北京市海淀区中关村南四街 4 号，中国科学院软件研究所，5 号楼 3 层计算机科学国家重点实验室

邮编：100090

E-mail: dreammaker2010@yeah.net

致　谢

值此论文完成之际，谨在此向多年来给予我关心和帮助的老师、学长、同学、朋友和家人表示衷心的感谢！