第35卷 第2期 计 算 机 学 报 Vol. 35 No. 2
2012年2月 CHINESE JOURNAL OF COMPUTERS Feb. 2012

一种提高时序安全属性静态检测实用性的方法

霍 玮[1] 李 丰[1),2)] 丁兆伟[1] 桑春雷[1),2)] 张兆庆[1] 冯晓兵[1]
[1]（中国科学院计算技术研究所计算机体系结构国家重点实验室 北京 100190）
[2]（中国科学院研究生院 北京 100049）

摘 要 程序时序安全属性可以用有限状态自动机（FSM）来描述，对该属性的静态检测是当前研究的热点之一.该文提出了FSM切片技术，以需求驱动的模式抽取出关于时序安全属性等价的程序切片.该切片使检测规模减小、程序结构简化，因而减小了检测中组合爆炸情形出现的机会，最终使时序安全属性的静态检测在准确性和可伸缩性上都得到了提高.实验表明，FSM切片可以使Saturn的可伸缩性平均提高到原来的6.34倍，使Fastcheck的准确性平均提高到原来的1.20倍.

关键词 有限状态自动机；时序安全属性；切片技术；程序静态检测；F-衡量
中图法分类号 TP311 **DOI号**：10.3724/SP.J.1016.2012.00244

A Precise and Scalable Static Checking Approach for Temporal Safety Property

HUO Wei[1] LI Feng[1),2)] DING Zhao-Wei[1] SANG Chun-Lei[1),2)] ZHANG Zhao-Qing[1] FENG Xiao-Bing[1]
[1]（*State Key Laboratory of Computer Architecture*, *Institute of Computing Technology*, *Chinese Academy of Sciences*, *Beijing* 100190）
[2]（*Graduate University*, *Chinese Academy of Sciences*, *Beijing* 100049）

Abstract Static program checking on temporal safety property that can be described by finite state machine (FSM) has been one of the hot research topics recently. In this paper, we propose a new approach to improve both of the precision and scalability of static program checking. We used FSM slicing to reduce the size of the programs being checked in a demand-driven manner without checking precision loss. Such reduction can simplify the structure of the programs thus reduce the complexity of the program analysis used by the program checking. The experiment results show that the FSM slices can improve the scalability of the Saturn to 6.34 times on average, and can improve the precision of the Fastcheck to 1.20 times on average.

Keywords finite state machine; temporal safety property; slicing; static program checking; F-measure

1 引 言

当今软件系统已经逐渐深入到社会的各个领域，软件的正确性问题得到人们的广泛关注，成为一个研究的热点.使用自动化的程序静态检测工具辅助程序开发是提高软件正确性的有效方法之一.当前，对于程序静态检测工具的研究取得了很大的进

收稿日期：2009-07-15；最终修改稿收到日期：2012-01-04.本课题得到国家“八六三”高技术研究发展计划项目基金（2008AA01Z115）、国家自然科学基金青年科学基金项目（61100011）、国家自然科学基金创新研究群体科学基金项目（60921002）、国家“核高基”重大专项基金项目（2009ZX01036-001-002）、国家“九七三”重点基础研究发展规划项目基金（2011CB302504）资助.**霍 玮**，男，1981年生，博士，主要研究方向为静态程序分析与检测、编译技术.E-mail：huowei@ict.ac.cn.**李 丰**，女，1985年生，博士研究生，主要研究方向为静态程序分析与检测、编译技术.**丁兆伟**，男，1982年生，硕士，主要研究方向为静态程序分析与检测.**桑春雷**，男，1979年生，博士研究生，主要研究方向为切片技术与并行程序分析.**张兆庆**，女，1938年生，研究员，博士生导师，主要研究领域为编译技术及相关工具环境.**冯晓兵**，男，1969年生，研究员，博士生导师，主要研究领域为编译技术及相关工具环境.

展，产生了很多优秀的成果．根据其使用的典型技术不同，可以将这些工具粗略的分为三大类：扩展静态检测器（extended static checker）[1-2]、软件模型检测器（software model checker）[3-4]和程序静态分析器（program static analyzer）[5-6]．

程序的时序安全属性具有非常广泛的描述能力，可以描述计算机软硬件资源的安全使用规则（比如内存、文件或锁等），也可以描述软件系统中的编程规则[3,7]等．所以，对于程序时序安全属性的检测是程序静态检测领域的重要研究课题．通常，可以使用有限状态自动机（FSM）来描述程序的时序安全属性．

定义1（有限状态自动机）．　M 是一个五元组：$M=(Q,\Sigma,\delta,q_0,F)$．其中，$Q$ 为状态的非空有限集合；Σ 是动作集合；δ 是状态转移函数，$\delta:Q\times\Sigma\rightarrow Q$；$q_0$ 是 M 的初始状态；F 是 M 的终止状态集合．

图1和图2分别给出了使用有限状态自动机描述的关于文件操作规则和内存操作规则的时序安全属性（双线框状态表示终止状态）．

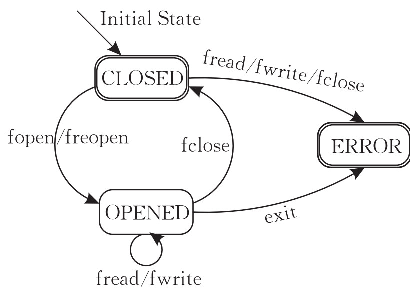

图1　文件操作时序安全属性

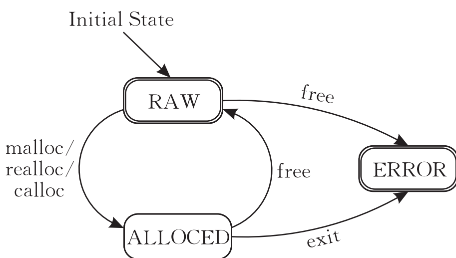

图2　内存操作时序安全属性

目前已有很多静态检测工具可以检测程序的时序安全属性，其中具有代表性的有软件模型检测器SLAM[3]、BLAST[4]；程序静态分析器Saturn[5]、Fastcheck[6]、ESP[8]、MYGCC[9]等等．虽然这些工具使用的技术不同，但一般从检测的准确性（precision）和可伸缩性（scalability）两方面来衡量它们的水平．实际上，检测的准确性和可伸缩性是一对矛盾．为了获得更高的检测准确性，就需要使用更高精度的程序分析，这就会增加分析的时空开销，在最坏情况下可能导致指数级的增加，使得检测的可伸缩性受到很大限制．所以上述工具在准确性和可伸缩性上进行了不同的权衡．然而，由于工具在检测的过程中没有充分利用程序信息，它们在准确性和可伸缩性上都存在提高的空间．对于BLAST、Saturn等高准确性的检测工具，它们在使用高精度的程序分析技术（比如路径敏感分析）时，由于没有利用检测的相关性而导致分析的时空开销代价过高，使得它们的可伸缩性受到很大影响；对于Fastcheck、MYGCC等高可伸缩性的检测工具，它们在追求高效、高可伸缩时，由于不“安全”地使用保守的数据流信息，甚至没有使用数据流信息，从而降低了它们的准确性．

针对以上问题，本文提出了FSM切片技术，使得程序时序安全属性的静态检测在准确性和可伸缩性两方面都得到了显著提高，并获得了更好的平衡．不同于传统的切片标准，该技术使用有限状态自动机作为切片标准，以需求驱动的模式利用切片技术提取出部分原程序，使得切片后程序与原程序关于时序安全属性具有语义等价性．由于程序切片减小了待检测程序的规模，从而提高了检测的可伸缩性；更重要的是，程序切片简化了待检测程序的结构，特别是降低了复杂程序结构对程序分析复杂度的影响，减少了检测中组合爆炸情形出现的机会，因而提高了分析的精度，最终提高了检测的准确性．实验表明，使用FSM切片技术可以使Saturn的可伸缩性在最好情况下提高到原来的1314倍，平均提高到原来的6.34倍；使Fastcheck的准确性在最好情况下提高到原来的2.19倍，平均提高到原来的1.20倍．

本文第2节给出研究的动机和示例；第3节给出FSM切片的定义和计算FSM切片的算法；第4节讨论FSM切片的实现要点；第5节讨论静态检测中准确性和可伸缩性的衡量标准；第6节对FSM切片的作用进行实验；最后是结论以及下一步的工作计划．

2　研究动机及示例

下面通过分析典型程序静态检测工具检测的工作过程，讨论这些工具存在的问题，并展示FSM切片技术在提高检测准确性和可伸缩性上的作用．

考虑使用Saturn对图3(a)中代码进行内存时序安全属性检测．该段代码来源于SPEC CPU2000中的ammp并稍作简化．Saturn的工作流程是：首先对源程序进行建模；然后对编码后的程序进行路

径敏感的指针分析，并将结果记录在带路径信息的位置集合[①]中；最后根据检测目标文件指定的步骤完成检测. 其中后两个步骤占检测总时间的 90％以上. 这段大约 1067 行的程序(包含空行)，含有大量的循环和分支语句. 由于 Saturn 在检测含有循环的代码时，会将循环展开有限次，然后再简化成分支语句进行检测. 所以根据我们的统计，这段代码大约含有 4 亿多条程序路径. 由于路径数量太过巨大，使得 Saturn 中路径敏感的指针分析和对错误路径的符号执行这两步都无法在设定的时间和空间内(100 s 处理器时间和 512 MB 内存)完成，最终使得 Saturn 无法完成对该段代码的检测. 实际上程序中大部分的循环和分支语句并不影响对该段代码进行时序安全属性静态检测的结果，如图中循环和分支语句所示，所以可以使用 FSM 切片技术简化该段代码，只保留与检测相关的语句. 这样经过切片预处理后，该段代码只含有约 26 万条程序路径，路径减少了 99.9％以上. 使用 Saturn 检测切片后的程序，不仅可以完成检测，而且能够准确地报告出这段代码关于内存的时序安全属性是正确的. 可见，FSM 切片技术可以提高 Saturn 检测时序安全属性的准确性和可伸缩性.

```
1.  int mm_fv_update_nonbon(float lambda){
2.     nodelist=malloc(nx*ny*nz*sizeof(MMNODE));
3.      if(nodelist==NULL)
4.         return 0;
5.      for(ix=0; ix<nx; ix++)
6.         for(iy=0; iy<ny; iy++)
7.            for(iz=0; iz<nz; iz++) {
8.               //Some operations on nodelist
9.            }
10.     …
11.     for(inode=0; inode <nx*ny*nz; inode++)
12.        for(ii=0; ii<imax; ii++)
13.           if((*atomlist)[ii].which==inode) {
14.               //Some operations on nodelist
15.               break;
16.           }
17.     …
18.     free(nodelist);
19.     return 1;
20. }
```

(a)

```
1.   FILE *f;
2.   void myclose() {
3.      fclose(f);
4.   }
5.   void main() {
6.      FILE *g;
7.      f=fopen();
8.      myclose();
9.      g=fopen();
10.     f=g;
11. }
```

(b)

```
1.   void main( ) {
2.      if(a>0)
3.         f=fopen( );
4.      i=0;
5.      while(i<1000){
6.         c=c+i;
7.         i=i+1;
8.      }
9.      if(a>0)
10.        fclose(f);
11. }
```

(c)

图 3 代码示例

考虑使用 Fastcheck 检测图 3(b)中代码. 该工具使用基于值流分析的检测技术. 它首先利用定值引用分析给全程序建立值流图，然后使用上下文敏感的切片技术计算出值流切片并对该切片进行(部分)路径可行性分析来完成检测. 但是为了提高可伸缩性，它的定值引用分析是基于流不敏感、上下文不敏感指针分析(Steensgaard 指针分析)的结果，在这样建立的值流图中可能会存在伪值流关系(即定值引用关系与控制流关系不符)，将导致检测的准确性降低. 对于该段代码，语句 10 使得指针分析分析出指针 f 和 g 别名，于是在值流图((b)图下半部，节点中标出其对应的语句号)中使用一个节点表示(阴影节点). 由于语句 9 对该节点代表的变量进行定值而语句 3 对该节点代表的变量进行引用，这样在值流图上会存在由语句 9 到语句 3 的值流关系，这个值流关系是违反了控制流顺序的伪值流关系. 但它的存在使得 Fastcheck 认为语句 9 打开的文件在程序结束前由语句 3 关闭，从而产生漏报. 使用切片预处理后，语句 10 由于与检测语句 9 打开的文件是否关闭无关而被切除，这样消除了由于 f 和 g 别名带来的伪值流影响，使得 Fastcheck 可以正确地检测出语句 9 打开的文件指针在程序结束前泄漏，从而在提高 Fastcheck 的可伸缩性的同时提高了 Fastcheck 的准确性.

考虑使用 BLAST 检测图 3(c)中的代码. 该代码是 BLAST 相关文章中常用示例. BLAST 是典型的软件模型检测器，它使用的是反例制导的抽象精化(counter-example guided abstraction refinement)检测机制. 该机制的主要流程是：首先根据初始谓词对程序进行抽象(即谓词抽象)，该步骤需要定理证

① Guarded location sets

明器的支持，是检测过程中主要的时间开销之一；然后在抽象的程序上(一般为布尔程序)检测程序的时序安全属性. 如果没有发现错误则证明原程序正确；如果发现错误，则根据抽象程序中的错误路径在原程序上找到对应的程序路径，并且判断该路径的可行性，这个步骤需要符号执行，也是检测过程中主要的时间开销之一. 如果该路径可行，则报告该错误；如果路径不可行，则根据不可行的原因修改谓词，然后再进入谓词抽象阶段，重复上述过程. 基于这个机制，BLAST 在检测图 3(c)中代码时，为了发现在语句 2 中条件为假的情况下语句 10 是否可达，需要依次地发现 1000 个谓词{i==0}，{i==1}，{i==2}，…，{i==999}(即进行谓词抽象 1000 次)，直到当{i≥1000}时才能在抽象程序中发现一条可以到达语句 10 的可能路径. 找到该路径在原程序中对应的路径〈2，4，$\underbrace{5，6，7，\cdots 5，6，7}_{1000次}$，5，9，10〉，然后在这条长度为 3005 的路径上进行符号执行，计算得到路径条件为{a≥0∧a<0}，所以这条路径不可行，语句 10 在语句 2 中条件为假时不可达. 然而从需求驱动的角度考虑，语句 10 是否可达直接取决于语句 9 中条件是否为真. 而语句 9 对语句 4、5、6、7 既没有控制依赖，也没有数据依赖，所以语句 9 中条件的值与语句 4、5、6、7 无关. 这正是切片技术可以给出的结论. 所以经切片预处理后，BLAST 对语句 1、2、3、9、10、11 组成的程序切片进行验证可以得到相同的结论，但是可以避免上述步骤，从而在保持 BLAST 检测准确性的同时，提高它的可伸缩性.

综上可见，使用切片技术对程序进行“预处理”后，不仅仅使得程序的规模减少，而且使得检测工具可以仅对与检测目标相关的程序部分进行重量级的程序分析，最大程度地避免无关程序部分特别是可能导致组合爆炸的程序成分(如分支、上下文等)对程序分析时空开销的影响，从而有效地缓和了分析精度和效率的矛盾，使得检测工具可以达到更高的准确性和可伸缩性.

3 FSM 切片

FSM 切片技术是提高程序静态检测准确性和可伸缩性的一个关键技术. 相对于传统切片技术中使用给定语句的变量作为切片标准，FSM 切片技术使用描述程序时序安全属性的有限状态自动机作为切片标准. 同时，与传统程序切片应用于程序调试、测试或理解等不同，FSM 切片是服务于程序时序安全属性的静态检测. 所以，FSM 切片利用了传统切片并与领域知识相结合，是切片技术的拓广和新应用. 这种新应用也缓解了切片技术的主要问题：切片程序往往过大而无法应用于人工分析. 因为 FSM 切片是服务于自动的程序静态检测，所以 FSM 切片对于原程序任何的减少都直接导致检测时间的减少和可伸缩性的提高. 并且由于切掉了与 FSM 无关的程序部分，往往还简化了程序结构和复杂的变量关系，从而有利于提高检测的准确性. 本文首先给出 FSM 切片的定义，然后讨论计算 FSM 切片的算法.

3.1 FSM 切片的定义

定义 2(FSM 切片). 程序 P 的一个 FSM 切片 $S(fsm)$是一个程序 P'，其中 fsm 是给定的时序安全属性描述，并且：

(1) P'是从 P 中删除 0 条或多条语句获得；

(2) 对于 fsm 描述的时序安全属性，P'是 P 的安全抽象.

定义 3(安全抽象). 给定程序的一个时序安全属性 p，用有限状态自动机 $M=(Q,\Sigma,\delta,q_0,F)$描述，称程序 P'是 P 的安全抽象，

(1) 如果 P 中存在路径 $path=s_1,s_2,\cdots,s_n$，使得 M 终止于错误状态 $f\in F$，则在 P'中存在路径 $path'=s'_1,s'_2,\cdots,s'_m$使得 M 的同一实例终止于相同的错误状态；并且设 $path$ 在 P 中的路径条件为 pc，$path'$在 P'中的路径条件为 pc'，则 $pc\mapsto pc'$.

(2) 如果 P 中存在路径 $path=s_1,s_2,\cdots,s_n$，使得 M 终止于正常状态 $f\in F$，则在 P'中存在路径 $path'=s'_1,s'_2,\cdots,s'_m$使得 M 的同一实例终止于相同的正常状态；并且设 $path$ 在 P 中的路径条件为 pc，$path'$在 P'中的路径条件为 pc'，则 $pc\mapsto pc'$.

这里的路径条件是指：对于给定的程序路径 p，通过符号执行得到它的一组关于变量初始值的约束，并且 p 是可行的，当且仅当路径条件可被满足. 这里使用有限状态自动机 M 的实例这个概念，是因为对于给定的有限状态自动机，一个程序中通常存在多组该自动机的动作语句，为保证检测算法的正确和叙述方便，我们称每组动作语句构成该自动机的一个实例. 比如给定的时序安全属性为文件操作，则我们称程序中所有与该自动机相关的文件操作函数为动作语句. 因为程序中每个被操作的文件都应该遵循该安全属性，所以我们首先需要将这些动作语句按照其操作的文件对象进行分组(每一组动作

语句是关于同一个文件的操作),然后对每组动作语句所代表的不同文件操作自动机实例分别进行检测.安全抽象指出了如果原程序中存在违反时序安全属性的错误,则该错误也存在于抽象后的程序中;如果原程序中的操作遵守时序安全属性,则抽象后的程序也遵守该属性.

3.2 FSM 切片的算法

FSM 切片的算法是以文献[11]中上下文敏感的过程间切片算法为基础设计的.因为相比于传统的切片技术,FSM 切片的切片标准不是确定程序位置上的程序变量而是描述时序安全属性的有限状态自动机,所以在进行 FSM 切片的过程中首先需要进行有限状态自动机的实例化.然后对于每一个自动机实例,以其中的动作语句作为切片标准,使用上下文敏感的过程间切片算法,从而得到这个自动机实例的 FSM 切片.

3.2.1 实例化有限状态自动机

实例化有限状态自动机,即根据给定有限状态自动机的状态迁移函数找到程序中的所有动作语句并进行分组.算法 1 给出了实例化的算法.

算法 1. 实例化有限状态自动机.

```
procedure InstFsm(P,M)
                              //P is program M=(Q,Σ,δ,q0,F)
  n←1
  E[1..m]←∅   //clear array E
  for all e: q0 --action--> q∈δ do
    for all st∈P do
      //if st is an instance of action
      if Match(st,action) then
        E[n]←E[n]∪{st}
        n←n+1
      end if
    end for
  end for
  for i←1,n do
    for all st∈P do
      //if st and I[i] belong to the same instantiation
      if Belong(st,E[i]) then
        E[i]←E[i]∪{st}
      end if
    end for
  end for
  return E
end procedure
```

该算法分为两步,第 1 步确定实例化标准.对于描述计算机系统软硬件资源操作的有限状态自动机,它的状态迁移函数在程序中对应的动作语句一般是函数调用,并且该函数操作的资源一般使用指针变量来代表,所以可以根据指针变量对象作为实例化的标准.为此我们首先确定程序中所有被操作的资源,即在程序中找到与导致初始状态转移的状态迁移函数所匹配的所有动作语句,这些语句中所操作的指针变量就代表所有被操作的资源.我们将这些语句记录并把每一条语句(准确地说是语句所操作的指针变量对象)作为一个实例化的标准.算法 1 中第 1 个最外层 For 循环即完成上述功能.其中 Match(*st*,*action*)用来判断函数调用 *st* 的函数名是否与状态迁移函数 *action* 的函数名相同.

第 2 步根据实例化标准完成实例化.我们对于每一个实例化标准,从这个标准开始遍历程序所有语句,如果动作语句所操作的指针变量与该标准所操作的指针变量是别名关系,则它们操作的是同一资源,该语句与实例化标准属于同一个有限状态自动机实例.这样便可完成有限状态自动机的实例化.算法 1 中第 2 个最外层 For 循环即完成上述功能.其中 Belong(*st*,*st_set*)用来判断函数调用 *st* 与集合 *st_set* 中的任意函数调用是否操作的是同一资源,即属于同一个有限状态自动机实例;一个数组 E 中的元素用来记录一个有限状态自动机的实例.

3.2.2 FSM 切片

算法 2 给出了完整的 FSM 切片过程.算法 2 主要包含 3 个步骤.首先对有限状态自动机进行实例化;然后,使用文献[12]中算法建立系统依赖图,再对每一个有限状态自动机实例,依次以每一个动作语句作为切片标准,使用文献[11]中算法(以 InterSlicing 函数代表)进行上下文敏感的过程间切片,并将切片结果合并,计算得到一个有限状态自动机实例的 FSM 切片.通常,切片的结果只是语句的集合,所以最后还需要使用 FSM 切片的结果,在原程序中提取出切片结果指示的语句从而得到最终的自动机实例的 FSM 切片.该算法将原程序的 FSM 切片结果记录在数组 P' 中,数组中的每一个元素记录的是每一个实例的 FSM 切片结果.可以证明算法 2 得到的结果就是 FSM 切片(见附录).

算法 2. FSM 切片算法.

```
precedure FsmSlicing(P,M)
                              //P is program M=(Q,Σ,δ,q0,F)
  //1. Instantiate the FSM
  E←InstFsm(P,M)
  //2. FSM Slicing
```

```
    Building System Dependence Graph G for P
    S[1..|I|]←∅
    for i←1,|E| do
      for all st∈E[i] do
        //Perform context-sensitive inter-procedural
          slicing for G, given the slicing criterion st
        S_tmp←InterSlicing(G,st)
        S[i]←S[i]∪S_tmp
      end for
    end for
    //3. Construct Slice
    for i←1,|S| do
      //Extract the statements that is in S[i] from P
      P'[i]←Construct(P,S[i])
    end for
    return P'
  end procedure
```

以图 3(b)作为示例说明.在实例化阶段,找到文件操作有限状态自动机的实例化标准分别是语句 7 和语句 9.对于语句 7,在程序的遍历过程中,找到语句 3 与其处于同一个实例化标准下(操作同一个文件指针对象),所以语句 7 和语句 3 属于有限状态自动机的同一个实例.对于语句 9,由于从语句 9 开始遍历无法到达语句 3,所以语句 3 与语句 9 不属于同一个有限状态自动机实例.这样,程序中有两个自动机实例{$E[1]=\{3,7\}$,$E[2]=\{9\}$}.以 $E[1]$ 作为切片标准进行切片,得到的结果如图 4(d1)所示;以 $E[2]$作为切片标准进行切片,得到的结果如图 4(d2)所示.

```
1. FILE *f;                     1. void main() {
2. void myclose() {             2.     FILE *g;
3.     fclose(f);               3.     g=fopen();
4. }                            4. }
5. void main() {
6.     f=fopen();
7.     myclose();
8. }
        (d1)                            (d2)
```

图 4　FSM 切片示例

4　FSM 切片实现要点

以 FSM 切片技术为核心,我们在 Open64 开放源码高级编译系统的基础上实现了一个 FSM 模型驱动的检测编译框架,称为 Fedora.该检测编译框架以编译技术为基础支撑,以 FSM 切片技术为核心,对输入的 C 程序检测给定的程序时序安全属性.图 5 给出了 Fedora 的总体结构(阴影部分表示对 Open64 的扩充与改造).Open64[①] 是一个开发源码的编译研究平台,具有良好的程序分析框架,非常适合作为实现检测工具的基础架构.

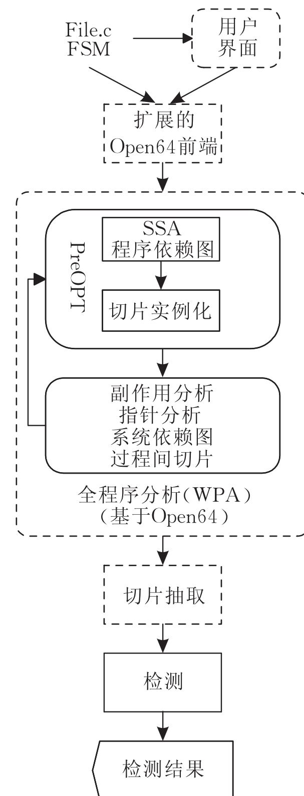

图 5　Fedora 总体结构

Fedora 提供了图形用户接口,使用户可以直观、方便、快速地描述有限状态自动机,提供待检测的时序安全属性.在该框架下,我们与 MYGCC 进行了集成实验.通过利用图形用户接口描述有限状态自动机,并将该自动机自动地转换成 MYGCC 能够接受的 condate(MYGCC 中描述待测性质的格式),可以极大地减少用户负担,这说明 Fedora 具有很好的易用性.

同时,我们改造 Open64 中的过程间分析框架为全程序分析框架,支持实现流敏感、上下文敏感的过程间分析,能够提高切片的精度,满足 FSM 切片的需求.在当前的框架上,使用的是 Steensgaard 的流不敏感、上下文不敏感指针分析,并在构建程序的静态单赋值形式之前进行全程序的副作用分析以精化定值引用信息,建立更精确的静态单赋值形式.在这个以静态单赋值形式为中间表示的全程序分析框架上,我们实现了程序依赖图以及系统依赖图的建

① http://www.open64.net/

立,并按照算法2实现了FSM切片算法.然后,我们根据FSM切片的结果提取出FSM切片,再使用第三方的检测工具,完成时序安全属性的检测.

5 可伸缩性和准确性的评价标准

本文主要是衡量FSM切片技术对于时序安全属性静态检测可伸缩性和准确性的提高,所以需要确定合理的可伸缩性和准确性的评价标准.

5.1 可伸缩性评价标准

对于可伸缩性的评价,当前主要有两个标准:每秒钟能够检测程序的行数(检测速率,单位千行/s);在合理的时间内能够检测的最大程序规模.对于第2个标准,由于“合理的时间”这种说法很难量化,并且很多工具在检测中可以设定最大的时空开销来强制完成检测(如Saturn),使得其在理论上可以检测任意规模的程序,所以第2个标准并不合理,故本文使用第一个标准来量化评价检测的可伸缩性.

5.2 准确性评价标准

对于准确性的评价,当前主要使用误报率作为衡量标准.然而,当前的程序静态检测工具由于技术上的原因或实现上的原因,一般既不是可靠的(即存在漏报)也不是完备的(即存在误报).所以仅用误报率不能全面客观地评价准确性.本文提出了借鉴信息检索领域中的F-衡量来作为静态检测工具准确性的评价标准.

定义4(F-衡量). TP、FP、TN、FN的含义见6.1节.

$$精确率(precision)=\frac{|TP|}{|TP|+|FP|},$$

$$召回率(recall)=\frac{|TP|}{|TP|+|FN|},$$

$$F_{\beta}\text{-衡量}(F_{\beta}\text{-measure})=\frac{(\beta^2+1)\times precision\times recall}{\beta^2\times precision+recall}.$$

精确率的含义是对于产生的关于程序错误的检测报告中,所包含的程序中真实存在的错误的比例.所以精确率是衡量误报情况的一个指标.召回率的含义是对于程序中真实存在的错误,能够包含在生成的程序错误报告中的比例.可见召回率是衡量漏报情况的一个指标.

一个好的检测工具应该具有高精确率和高召回率.但是,通常情况下,精确率和召回率很难同时做到最好.所以评价一个检测工具的准确性,应该能够综合检测的精确率和召回率,使这两个指标达到一个好的平衡.这时,可以使用精确率和召回率的加权的调和平均来作为指标.这就是F-衡量.其中$\beta\in[0,\infty)$,表明的是召回率比精确率的重要性高的比例.我们认为,检测工具应该尽可能多地发现程序中的真正错误,而用户对误报具有一定的容忍性,也就是召回率要比精确率重要,故基本的要求是$\beta>1$.在本文实验中,选取$\beta=4$①.

6 实 验

6.1 实验方法

对于时序安全属性,本文选取文件时序安全属性(文件FSM)和内存时序安全属性(内存FSM),如图1和图2所示,检测程序中是否存在文件指针泄露和内存泄露的错误.

对于第三方的检测工具,本文选取Fastcheck和Saturn.Fastcheck和Saturn都是代表性的程序静态分析工具,但是两者各具不同的特点,如第2部分所讨论.

Fastcheck由康奈尔大学开发,具有高可伸缩性,但由于不安全地使用别名信息、只进行部分的路径敏感分析(不计算路径条件)等而影响了准确性.本文实验中选取的是Fastcheck的最新版本,Fastcheck 1.0.Fastcheck内置了检测文件时序安全属性和内存时序安全属性的功能,可以直接进行本实验.Saturn由斯坦福大学开发,由于支持路径敏感的过程间分析而具有很好的检测准确性,但可伸缩性受到较大限制.本文实验中选取的是Saturn的最新版本,Saturn 1.1.在使用Saturn检测文件时序安全属性和内存时序安全属性时,需要提供属性描述文件.为此我们在能力范围内反复地手工调整该属性描述文件以尽可能地发挥Saturn的检测能力.

本实验的过程如下:对于每一个待测用例,首先分别针对每一个时序安全属性,计算FSM切片(将每个实例的FSM切片结果合并求得原程序的FSM切片),并根据切片结果利用分析工具exceptor(基于Crystal实现②)在原程序中取出FSM切片.在给定时序安全属性和待测用例的情况下,使用Fast-

① 根据经验,$\beta=4$.因为我们分别实验了$\beta=2$、$\beta=4$、$\beta=6$、$\beta=8$、$\beta=10$,发现当$\beta>4$以后,F-衡量的值随β增大变化很小.

② http://www.cs.cornell.edu/projects/crystal/

check 和 Saturn 分别对原程序和 FSM 切片进行检测，共得到 4 个测试报告. 这 4 个报告其实是关于同一个程序检测同一种错误. 所以，对每个报告的每一个条目逐一进行人工确认，并将每一个条目划定到以下 4 类之一：报告是错误且是实际存在的违反时序安全属性的错误（*TP*），报告是错误但实际是遵守时序安全属性的情况（*FP*），报告是正确且实际上也是遵守时序安全属性的情况（*TN*），报告是正确但是实际存在违反时序安全属性的错误（*FN*）.

由于 Fastcheck 和 Saturn 只能检测 C 程序，所以我们根据程序规模，选取了 SPEC CPU2000、SPEC CPU2006[①] 中的部分 C 程序（包括浮点程序和定点程序）和 OpenSSH[②] 中部分程序. 这些测试用例都是源自现实中广泛使用的应用程序，覆盖了相当广泛的应用领域，能够体现不同的应用程序特征，代码规模从几千行到几十万行，能够全面客观地衡量 FSM 切片和 Fedora 的有效性.

6.2　实验环境

本实验的所有实验数据均是在同一台服务器上测得. 该服务器是一台双处理器的机器，每个处理器都是 Intel Xeon(R) E5430，含有 4 个处理器核，主频为 2.66 GHz，每个核私有 64 KB 的一级 Cache，每两个核共享 6 MB 的二级 Cache，并支持 64 位指令集. 该服务器具有 16 GB 的内存和 1T 的硬盘，预装有 64 位的 Red Hat 企业版 Linux5.2，内核版本 2.6.18.

Fastcheck 的运行选项是其默认的选项；Saturn 的运行选项也是其默认选项，对于每个函数的处理，限制使用 512 MB 的内存和 100 s 的处理器时间.

测试用例的简单描述如表 1 所述.

表 1　测试用例描述

用例名称	描述	用例名称	描述
OpenSSH 系列	远程访问服务器和客户端	300.twolf	微芯片布局和互联程序
175.vpr	FPGA 线路布局	403.gcc	C 程序编译器
177.mesa	3 维图形库	433.milc	量子色动力学
188.ammp	计算化学	445.gobmk	人工智能程序GO
253.perlbmk	Perl 程序解释器	458.sjeng	国际象棋计算
254.gap	聚合理论解释器	482.sphinx3	语音识别

6.3　实验结果

6.3.1　可伸缩性

表 2 和表 3 分别给出了使用 FSM 切片对于 Fastcheck 和 Saturn 进行可伸缩性实验的结果. 对于每个工具检测的每个时序安全属性，分别给出了检测原程序的时间（第 3 列和第 5 列）和检测 FSM 切片的时间（第 4 列和第 6 列，该时间已包含计算 FSM 切片的时间，并在括号内给出）. 表中黑体表示最好情况. 这样我们就可以根据程序的规模来计算每次检测的检测速率. FSM 切片对于 Fastcheck 和 Saturn 的可伸缩性提高如图 6 和图 7 所示. 两图中 *Y* 轴表示的是 FSM 切片使得可伸缩性提高的比例（将原程序的检测速率进行归一化处理）.

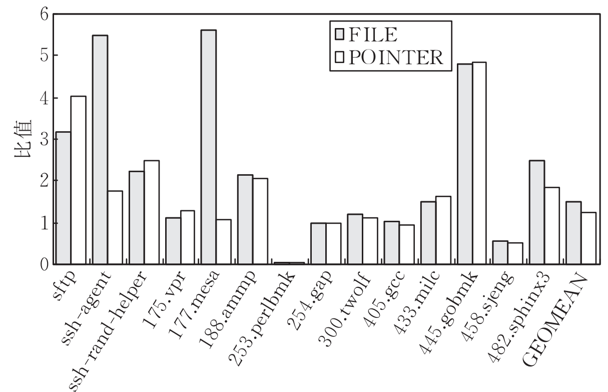

图 6　Fastcheck 可伸缩性提高比例

表 2　Fastcheck 可伸缩性实验结果

用例名称	规模/千行	文件 FSM 的检测时间/s		内存 FSM 的检测时间/s	
		原程序	FSM 切片（切片时间）	原程序	FSM 切片（切片时间）
sftp	5.8	2.3	0.7(0.3)	3.0	0.7(0.3)
ssh-agent	2.5	0.7	0.1(0.1)	0.8	0.4(0.1)
ssh-rand-helper	1.5	0.6	0.3(0.1)	0.6	0.2(0.1)
175.vpr	17.7	3.5	3.1(1.2)	3.7	2.9(1.2)
177.mesa	61.3	**12.9**	**2.3(2.0)**	14.9	13.7(2.0)
188.ammp	13.5	3.5	1.7(0.6)	3.8	1.9(0.6)
253.perlbmk	85.5	11.1	286.1(278.5)	13.2	285.4(277.3)
254.gap	71.4	7.9	8.1(0.9)	7.9	7.9(0.9)
300.twolf	20.5	5.9	4.8(2.5)	5.7	5.2(2.5)
403.gcc	521.1	119.7	117.4(59.6)	118.3	124.9(60.8)
433.milc	15.0	4.7	3.1(0.7)	4.9	3.0(0.7)
445.gobmk	197.2	40.3	8.4(4.0)	**43.5**	**9.0(4.1)**
458.sjeng	13.8	3.3	6.2(4.3)	3.3	6.3(4.3)
482.sphinx3	25.1	8.9	3.6(1.1)	17.7	9.7(1.1)

① http://www.spec.org
② http://www.openssh.com

表3 Saturn可伸缩性实验结果

用例名称	规模/千行	文件FSM的检测时间/s		内存FSM的检测时间/s	
		原程序	FSM切片(切片时间)	原程序	FSM切片(切片时间)
sftp	5.8	181.2	66.4(0.3)	197.1	151.4(0.3)
ssh-agent	2.5	46.09	0.1(0.1)	53.05	40.18(0.1)
ssh-rand-helper	1.5	29.10	4.2(0.1)	33.08	10.28(0.1)
175.vpr	17.7	810.1	394.3(1.2)	1672	1456(1.2)
177.mesa	61.3	**6726**	**5.1(2.0)**	7301	2072(2.0)
188.ammp	13.5	3267	697.9(0.6)	**3439**	**841.8(0.6)**
253.perlbmk	85.5	78694	55495(281.5)	79026	59881(277.3)
254.gap	71.4	105738	87912(0.9)	105662	87878(0.9)
300.twolf	20.5	5591	2392(2.5)	6115	4003(2.5)
433.milc	15.0	659.2	190.8(0.7)	819.2	418.9(0.7)
445.gobmk	197.2	9141	1807(4.0)	9050	2329(4.1)
458.sjeng	13.8	2212	1318(4.3)	2293	1699(4.3)
482.sphinx3	25.1	2080	834.2(1.1)	2484	1594(1.1)

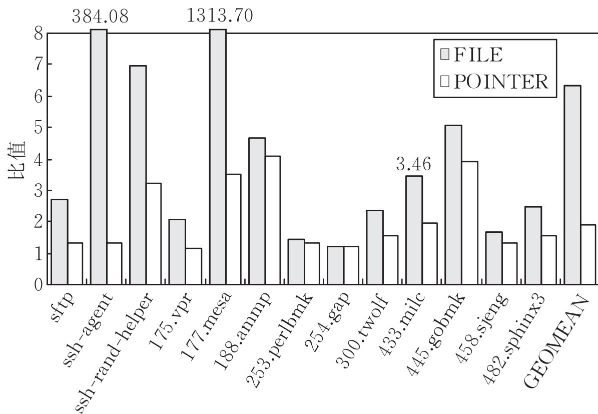

图7 Saturn可伸缩性提高比例

对于177.mesa,文件FSM切片可以使Fastcheck的可伸缩性提升高达原来的5.62倍,尤其使Saturn的可伸缩性提升高达原来的1314倍.这主要因为mesa的文件FSM切片仅为原程序的0.56%.平均而言,文件FSM切片使得Fastcheck可伸缩性提高到原来的1.48倍,使Saturn的可伸缩性提高到原来的6.34倍.

在内存时序安全属性的实验中,内存FSM切片可以使Fastcheck的可伸缩性提升高达原来的4.82倍,使Saturn的可伸缩性提升高达原来的4.09倍.平均而言,内存FSM切片使得Fastcheck的可伸缩性提高到原来的1.23倍,使得Saturn的可伸缩性提高到原来的1.88倍.

6.3.2 准确性

对于文件时序安全属性和内存时序安全属性,我们使用Fastcheck和Saturn分别检测原程序和FSM切片,并对所得到的报告逐一进行确认和分类.我们对我们能够确定的检测报告(排除复杂测试用例和FSM切片算法的实现对报告确认的干扰)分别列入表4和表5中,同时给出了根据5.3节的公式计算出的F-衡量的值.其中黑体部分表示FSM切片可以提高检测准确性的测试用例.FSM切片对于Fastcheck和Saturn检测文件时序安全属性和内存时序安全属性准确性的提高分别如图8和图9所示.两图中Y轴表示的是FSM切片对于准确性提高的比例(将原程序的F-衡量值进行归一化处理).

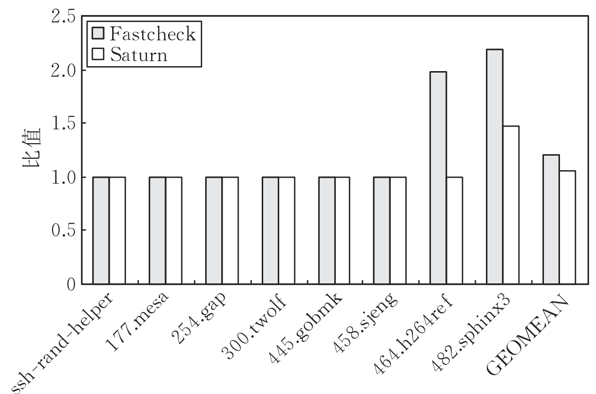

图8 FSM切片对检测文件时序安全属性准确性的提高

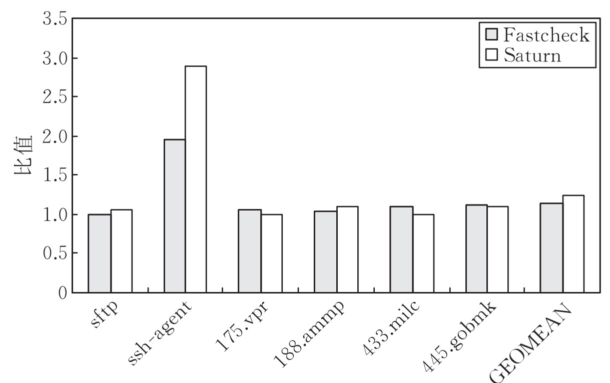

图9 FSM切片对检测内存时序安全属性准确性的提高

表 4　文件时序安全属性检测结果

例子名称	FastCheck 检测结果										Saturn 检测结果									
	原程序检测结果					FSM 切片检测结果					原程序检测结果					FSM 切片检测结果				
	TP	*FP*	*TN*	*FN*	F_4	*TP*	*FP*	*TN*	*FN*	F_4	*TP*	*FP*	*TN*	*FN*	F_4	*TP*	*FP*	*TN*	*FM*	F_4
ssh-rand-helper	1	0	0	0	1.00	1	0	0	0	1.00	1	0	0	0	1.00	1	0	0	0	1.00
177.mesa	2	0	2	0	1.00	2	0	2	0	1.00	2	0	2	0	1.00	2	0	2	0	1.00
254.gap	1	0	0	2	0.35	1	0	0	2	0.35	3	0	0	0	1.00	3	0	0	0	1.00
300.twolf	3	0	29	0	1.00	3	0	29	0	1.00	3	0	29	0	1.00	3	0	29	0	1.00
445.gobmk	2	0	4	0	1.00	2	0	4	0	1.00	2	0	4	0	1.00	2	0	4	0	1.00
458.sjeng	2	0	1	0	1.00	2	0	1	0	1.00	2	0	1	0	1.00	2	0	1	0	1.00
464.h264ref	**1**	**0**	**6**	**4**	**0.21**	**2**	**0**	**6**	**3**	**0.41**	4	0	6	1	0.81	4	0	6	1	0.81
482.sphinx3	**4**	**0**	**20**	**7**	**0.38**	**9**	**0**	**20**	**2**	**0.83**	**6**	**0**	**20**	**5**	**0.56**	**9**	**0**	**20**	**2**	**0.83**

表 5　内存时序安全属性检测结果

例子名称	FastCheck 检测结果										Saturn 检测结果									
	原程序检测结果					FSM 切片检测结果					原程序检测结果					FSM 切片检测结果				
	TP	*FP*	*TN*	*FN*	F_4	*TP*	*FP*	*TN*	*FN*	F_4	*TP*	*FP*	*TN*	*FN*	F_4	*TP*	*FP*	*TN*	*FM*	F_4
sftp	6	0	31	14	0.31	6	0	31	14	0.31	**16**	**0**	**31**	**4**	**0.81**	**17**	**0**	**31**	**3**	**0.86**
ssh-agent	**1**	**0**	**2**	**2**	**0.35**	**2**	**0**	**2**	**1**	**0.68**	**1**	**0**	**2**	**2**	**0.35**	**3**	**0**	**2**	**0**	**1.00**
175.vpr	**32**	**0**	**53**	**72**	**0.32**	**34**	**0**	**53**	**70**	**0.34**	103	7	46	1	0.99	103	7	46	1	0.99
188.ammp	**30**	**0**	**4**	**3**	**0.91**	**31**	**0**	**4**	**2**	**0.94**	**29**	**0**	**4**	**4**	**0.89**	**32**	**0**	**4**	**1**	**0.97**
433.milc	**20**	**0**	**27**	**35**	**0.38**	**22**	**2**	**25**	**33**	**0.41**	51	2	25	4	0.93	51	2	25	4	0.93
445.gobmk	**8**	**0**	**9**	**14**	**0.38**	**9**	**0**	**9**	**13**	**0.42**	**20**	**0**	**9**	**2**	**0.91**	**22**	**0**	**9**	**0**	**1.00**

从图 8 中可以看出，文件 FSM 切片没有使任何一个工具的检测准确性降低．对于 Fastcheck，文件 FSM 切片可以使其准确性提升高达原来的 2.19 倍，平均提高到原来的 1.20 倍；对于 Saturn，文件 FSM 切片可以使其准确性提升高达原来的 1.48 倍，平均提高到原来的 1.05 倍．

从图 9 中可以看出，内存 FSM 切片没有使任何一个工具的检测准确性降低．对于 Fastcheck，内存 FSM 切片可以使其准确性提升高达原来的 1.96 倍，平均提高到原来的 1.15 倍；对于 Saturn，内存 FSM 切片可以使其准确性提升高达原来的 2.88 倍，平均提高到原来的 1.24 倍．

6.4　讨　论

6.4.1　可伸缩性

从横向上看，文件 FSM 切片对于可伸缩性的提高要明显好于内存 FSM 切片，这两个工具都体现出了这一特点．这是因为，通常情况下，程序中与操作文件相关的语句要明显少于与操作内存相关的语句，所以文件 FSM 切片要明显小于内存 FSM 切片，相应的，文件 FSM 切片对于检测时间的减小也要大于内存 FSM 切片对于检测时间的减小，这样文件 FSM 切片对于可伸缩性的改善要好于内存 FSM 切片．

从纵向上看，FSM 切片对于 Saturn 的可伸缩性改善要明显优于对 Fastcheck 的改善，实验的两个时序安全属性都体现了这一点．这与工具本身的特点相关．Fastcheck 的设计理念就是尽可能高效地检测程序，所以它的优势就在于可伸缩性比较好；相反，Saturn 由于使用路径敏感的分析，可伸缩性则受到了限制．FSM 切片不仅减少了程序规模，而且简化了程序结构和复杂的变量关系，特别是减少了可能导致程序分析指数爆炸的程序成分比如分支语句，这样就提高了 Saturn 中的路径敏感的程序分析效率，这加速了 FSM 切片对于 Saturn 可伸缩性的改善．

6.4.2　准确性

无论是文件 FSM 切片，还是内存 FSM 切片，都可以使 Fastcheck 检测时序安全属性准确性提高．这主要是因为召回率的提高，而召回率提高的原因，主要是 FSM 切片克服了 Fastcheck 中不“安全”技巧的影响．FSM 切片去掉了程序中与检测 FSM 无关的部分，降低了程序的复杂性，可以使指针分析得到更精确的结果以间接地消除部分不“安全”的使用，如图 3(b)代码所示，从而使漏报减少，相对召回率提高．

无论是文件 FSM 切片，还是内存 FSM 切片，都可以使 Saturn 检测时序安全属性的准确性提高．准确性的提高同样也主要来源于召回率的提高，而召回率提高的原因，主要是 FSM 切片可以提高 Saturn 中指针分析的效率．值得指出的是，因为 Saturn 使用的路径敏感的指针分析可能导致组合爆炸问题，所以 Saturn 中设定了分析每个函数所可以使用的最大内存量和最长处理器用时，使得发生组合爆炸的情况下，检测仍得以进行．对于测试用例中

的大函数，Saturn 很难在这些规定值内完成预定分析，从而导致漏报.而 FSM 切片有效地减小了函数规模，简化了程序结构，降低了程序复杂性，使得 Saturn 可以在规定值内完成预定分析，如图 3(a)中代码所示，使得召回率提高.

7 结论及下一步工作计划

软件正确性问题，随着信息技术的发展，越来越多地得到了人们的广泛重视.在软件开发中使用程序静态检测工具是一种有效的提高软件正确性的方法.当前有代表性的静态检测工具由于使用的技术不同而在检测的准确性和可伸缩性上进行不同的权衡.然而，这些工具在检测过程中，没有以检测需求为驱动，没能充分利用已有程序信息，使得它们的准确性和可伸缩性都受到了不同程度的损失.

本文在分析和总结这些典型检测工具工作特点的基础上，针对时序安全属性的检测，提出了 FSM 切片技术.不同于传统的切片标准，该技术使用有限状态自动机作为切片标准，以需求驱动的模式利用切片技术提取出部分原程序，在减少程序规模的基础上简化了程序成分和复杂变量关系，是使检测的准确性和可伸缩性获得更好平衡的关键技术.本文基于传统上下文敏感的过程间切片算法，给出了 FSM 切片算法，并以该算法为核心实现了 Fedora 检测编译框架原型系统.

实验表明，FSM 切片可以使 Saturn 和 Fastcheck 检测的可伸缩性和准确性都取得明显提高.对于文件时序安全属性的 FSM 切片，可以使 Saturn 的可伸缩性平均提高到原来的 6.34 倍，准确性平均提高到原来的 1.05 倍；可以使 Fastcheck 的可伸缩性平均提高到原来的 1.48 倍，准确性平均提高到原来的 1.20 倍.对于内存时序安全属性的 FSM 切片，可以使 Saturn 的可伸缩性平均提高到原来的 1.88 倍，准确性平均提高到原来的 1.24 倍；可以使 Fastcheck 的可伸缩性平均提高到原来的 1.23 倍，准确性平均提高到原来的 1.15 倍.

对于 FSM 切片技术，我们的进一步工作包括：一方面是改进现有切片算法，进一步提高切片的效率和准确性；另一方面是在切片的过程中，进一步使用有限状态自动机所描述的语义，构造不确定状态切片.不确定状态切片是在现有 FSM 切片的基础上，将确定能够导致有限状态自动机转移到正确终止状态或错误终止状态的语句进一步切除，只保留导致有限状态自动机进行不确定转移的程序成分，这样可以进一步减少 FSM 切片规模，使得静态检测在可伸缩性和准确性上得到更好的改善.

参 考 文 献

[1] Flanagan C, Leino K R M, Lillibridge M et al. Extended static checking for Java//Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation. Berlin , Germany, 2002: 234-245

[2] Babic D, Hu A J. Calysto: Scalable and precise extended static checking//Proceedings of the 30th International Conference on Software Engineering. Leipzig, Germany, 2008: 211-220

[3] Ball T, Rajamani S K. The SLAM project: Debugging system software via static analysis//Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. Portland, OR, USA, 2002: 1-3

[4] Henzinger T A, Jhala R, Majumdar R. Lazy abstraction//Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. Portland, OR, USA, 2002: 58-70

[5] Aiken A, Bugrara S, Dilling I et al. An overview of the saturn project//Proceedings of the 7th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering. San Diego, California, USA, 2007: 43-48

[6] Cherem S, Princehouse L, Rugina R. Practical memory leak detection using guarded value-flow analysis//Proceedings of ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation. San Diego, California, USA, 2007: 480-491

[7] Engler D, Chelf B, Chou A, Hallem S. Checking system rules using system-specific, programmer-written compiler extensions//Proceedings of the 4th Symposium on Operating Systems Design and Implementation. San Diego, California, USA, 2000: 1-16

[8] Das M, Lerner S, Seigle M. ESP: Path-sensitive program verification in polynomial time//Proceedings of ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation. Berlin , Germany, 2002: 57-68

[9] Volanschi N. A portable compiler-integrated approach to permanent checking//Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering. Tokyo, Japan, 2006: 103-112

[10] Steensgaard B. Points-to analysis in almost linear time//Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. St. Petersburg Beach, Florida, USA, 1996: 32-41

[11] Krinke J. Evaluating context-sensitive slicing and chopping//Proceedings of the International Conference on Software Maintenance. Atlanta, Georgia, USA: IEEE Computer Society, 2002: 22-31

[12] Horwitz S, Reps T, Binkley D. Interprocedural slicing using dependence graphs//Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation. Montréal, Canada, 1988: 35-46

[13] Manning C D, Raghavan P, Schutze H. Introduction to information retrieval. Cambridge: Cambridge University Press, 2008

附　录.

下面的内容将证明由算法 2 计算得到的切片符合 FSM 切片的定义. 为了使证明简化但不失一般性，假设程序中任意一条可执行路径上仅含有 4 类语句：*assign*、*assume*、*goto* 和 *action*，分别代表赋值语句，分支条件语句，跳转语句和有限状态自动机动作语句（即对 guarded command① 扩充有限状态自动机动作语句）.

程序状态是一个 4 元组，$X=(\varphi,s,v,st)$，其中 φ 表示程序变量状态，s 表示有限状态自动机状态，v 表示当前关心的变量集合，st 表示最新导致 v 更新的语句.

程序状态由程序变量状态和有限状态自动机的状态组成. 为了记录程序如何影响有限状态自动机的状态，使用变量集合 v 记录与有限状态自动机状态迁移相关的变量，即那些影响有限状态自动机中动作语句中变量的值或该动作语句执行的变量. 由于 v 将随着程序的执行序列更新，故使用 st 记录最近使得 v 更新的语句. 如果 v 为空，表示不记录程序如何影响有限状态自动机的状态转移.

给定程序的执行序列 $s_1,s_2,\cdots,s_n$，对该执行序列的程序执行规则如附表 1 所示②.

附表 1　程序执行规则

语句	规则	序号
$s_i\equiv x:=e$	$\dfrac{X(\varphi,s,\Phi,null)}{X(\varphi[e/x],s,\Phi,null)}$	1
$s_i\equiv assume(e)$	$\dfrac{X(\varphi,s,\Phi,null)}{X(\varphi\vee\neg e,s,\Phi,null)}$	2
$s_i\equiv goto$	$\dfrac{X(\varphi,s,\Phi,null)}{X(\varphi,s,\Phi,null)}$	3
$s_i\equiv action_i$	$\dfrac{X(\varphi,s,\Phi,null),s'\xrightarrow{action_i}s}{X(\varphi,s',\Phi,null)}$	4

该规则指出如果语句执行后程序状态为分子所示内容的话，那么语句执行前的程序状态就变成分母所示，所以程序执行规则给出了给定语句对于程序状态的最弱前件推导规则.

给定程序的执行序列 $s_1,s_2,\cdots,s_n$，对该执行序列的切片执行规则如附表 2 所示.

附表 2　切片执行规则

语句	规则	条件	序号
$s_i\equiv x:=e$	$\dfrac{X(\varphi_s\wedge\varphi_r,s,v,st)}{X((\varphi_s[e/x])\wedge\varphi_r,s,v/\{x\}\cup Vars(e),s_i)}$	$x\in v$	1
	$\dfrac{X(\varphi_s\wedge\varphi_r,s,v,st)}{X(\varphi_s\wedge(\varphi_r[e/x]),s,v,st)}$	$x\notin v$	2
$s_i\equiv assume(e)$	$\dfrac{X(\varphi_s\wedge\varphi_r,s,v,st)}{X((\varphi_s\vee\neg e)\wedge\varphi_r,s,v\cup Vars(e),s_i)}$	$s_i\in CD(st)$	3
	$\dfrac{X(\varphi_s\wedge\varphi_r,s,v,st)}{X(\varphi_s\wedge(\varphi_r\vee\neg e),s,v,st)}$	$s_i\notin CD(st)$	4
$s_i\equiv goto$	$\dfrac{X(\varphi_s\wedge\varphi_r,s,v,st)}{X(\varphi_s\wedge\varphi_r,s,v,st)}$	NONE	5
$s_i\equiv action_i$	$\dfrac{X(\varphi_s\wedge\varphi_r,s,v,st),s'\xrightarrow{action_i}s}{X(\varphi_s\wedge\varphi_r,s',v\cup Vars(s_i),s_i)}$	NONE	6

在该规则中，函数 $Vars(e)$ 表示表达式 e 中引用的变量集合，$CD(st)$ 表示语句 st 的控制依赖语句. 该规则将程序变量状态分成两个部分分别计算：φ_s 表示的切片相关变量状态和 φ_r 表示的切片无关变量状态. 切片相关变量状态记录的是改变 v 中变量的值的语句或决定语句 st 的执行（用控制依赖来判断）的语句对程序状态的更新. 而切片无关变量状态正好相反，记录其余语句对程序状态的更新. 同样，切片执行规则给出了给定语句对于程序状态的最弱前件推导规则，但与程序执行规则所不同的是对程序变量状态的计算分为两部分.

记 $PE(s_i)(Q)$ 为当语句 s_i 执行后的程序状态为 Q 时，使用程序执行规则计算得到的语句 s_i 执行前的程序状态；记 $SE(s_i)(Q)$ 为当语句 s_i 执行后的程序状态为 Q 时，使用切片执行规则计算得到的语句 s_i 执行前的程序状态. 有如下引理.

引理.　对于给定程序的执行序列 $seq\equiv s_1,s_2,\cdots,s_n$，令 $Q\equiv\{\varphi_q,s_q,\Phi,null\}$ 为该序列执行完毕时的程序状态，并记

$$PE(seq)(Q)\equiv\{\varphi_{pe},s_{pe},v_{pe},st_{pe}\},$$

$$SE(seq)(Q)\equiv\{\varphi_{se},s_{se},v_{se},st_{se}\},$$

则

$$\varphi_{pe}=\varphi_{se}\text{ 并且 }s_{pe}=s_{se}.$$

证明.　根据程序执行规则以及切片执行规则可以直接得到.　　证毕.

在引理 1 的基础上，很容易证明如下定理.

定理.　FSM 切片算法得到的结果是 FSM 切片，FSM 切片与原程序关于时序安全属性具有语义等价性.

证明.　对程序 P 和描述时序安全属性的有限状态机 $M=(Q,\Sigma,\delta,q_0,F)$ 使用 FSM 切片算法，记结果为 S.

(1) 显然，S 符合 FSM 切片定义的第 1 条.

(2) 下面证明 S 是 P 的安全抽象. 设 M_i 是 M 在 P 中的任意一个实例.

对于安全抽象的第 1 点，设 $path=s_1,s_2,\cdots,s_n$ 是 P 中导致 M_i 到达错误终止状态的路径，记该状态为 $e\in F$. 记其中的有限状态自动机动作语句分别为 $s_{i_1},s_{i_2},\cdots,s_{i_k}$ $(1\le i_1<i_2<\cdots<i_k\le n)$.

设沿 $path$ 执行终止时程序状态 $Q\equiv\{\varphi_q,e,\Phi,null\}$. 使用程序执行规则计算 $path$ 的最弱前件，记作 $\{\varphi_{pe},s_{pe},\Phi,null\}$. 这个最弱前件中的程序变量状态是根据传统的程序最弱前件计算规则计算得到的③，所以就是路径 $path$ 的路径条件. 使用切片执行规则计算 $path$ 的最弱前件，记作 $\{\varphi_s\wedge\varphi_r,s_{se},v_{se},st_{se}\}$，并将应用规则第 1、3、5、6 条的语句按照访问的反序记入 $path''=s''_1,s''_2,\cdots,s''_m$ 中.

因为只有程序执行规则的第 4 条与切片执行规则的第 6 条将更新程序状态中的有限自动状态机的状态，并且这两

① 参见 Dijkstra E W. Guarded commands, non-determinacy and formal derivation of programs. Communications of the ACM, 1975, 18: 453-457

② $\varphi[e/x]$ 表示用 e 代替 φ 中所有自由出现的 x.

③ 参见 Dijkstra E W. A Discipline of Programming. Prentice Hall, 1976

条规则在分别计算最弱前件的过程中有相同的应用方式和应用次数，所以执行 $path''$ 终止时也将使有限状态自动机到达错误终止状态 e.

又因为使用切片执行规则的第 1 和 3 条，对 φ_s 的计算同样等价于按照传统的程序最弱前件计算规则计算，所以 φ_s 是 $path''$ 的路径条件. 又根据引理有，

$$\varphi=\varphi_s\wedge\varphi_r,$$

$$s_{pe}=s_{se},$$

所以

$$\varphi\mapsto\varphi_s.$$

下面如果证明 S 中存在与 $path''$ 相同的路径，即可证明 S 是 P 的安全抽象. 因为算法 2 中，

① 将有限状态自动机的动作语句实例作为切片标准，等价于使用切片执行规则的第 6 条，将动作语句实例用 st 记录，并将其中涉及的变量记入 v 中；

② 切片时沿数据依赖边的访问，等价于使用切片执行规则的第 1 条，将与 v 有数据依赖的语句用 st 记录，并将该语句使用的变量更新到 v 中，同时计算程序变量状态的变化；

③ 切片时沿控制依赖边的访问，等价于使用切片执行规则的第 3 条，对与 st 有控制依赖的语句(根据控制依赖的定义，只可能是分支条件语句)，用该语句更新 st，并用其中使用的变量更新 v，同时计算程序变量状态的变化.

可见，$path''$ 是可以由算法 2 计算得到的，也就是 S 中存在路径 $path'=s_1',s_2',\cdots,s_m'$ 使得 $s_1'=s_1'',s_2'=s_2'',\cdots,s_m'=s_m''$，所以安全抽象第 1 条得证.

对于安全抽象第 2 条可以类似证明.

所以，FSM 切片算法得到的结果是 FSM 切片.

(3) 因为算法 2 根据 P 中的有限状态机实例逐一进行切片，所以 FSM 切片具有与原程序完全一样数量的有限状态自动机实例. 这样安全抽象的概念就能够保证了 FSM 切片与原程序关于时序安全属性具有语义等价性. 这是因为：设原程序与 FSM 切片中都共有 n 个实例，并且在原程序中 $n\equiv n_r+n_w$(其中 n_r 与 n_w 分别代表到达正确终止状态的实例数和到达错误终止状态的实例数)以及在 FSM 切片中 $n\equiv n_{sr}+n_{sw}$(其中 n_{sr} 与 n_{sw} 有类似定义)，根据安全抽象的定义，有如下方程组：$\begin{cases}n_r+n_w=n_{sr}+n_{sw}\\0\le n_r\le n_{sr}\\0\le n_w\le n_{sw}\end{cases}$，所以 $\begin{cases}n_r=n_{sr}\\n_w=n_{sw}\end{cases}$，即 FSM 切片与原程序关于时序安全属性具有语义等价性. 证毕.

Huo Wei, born in 1981, Ph. D.. His main research interests include static analysis and detection techniques, compiler techniques.

LI Feng, born in 1985, Ph. D. candidate. Her main research interests include program analysis and debugging techniques.

DING Zhao-Wei, born in 1982, M. S.. His main research interests include program analysis and testing techniques.

SANG Chun-Lei, born in 1979, Ph. D.. His main research interests include static analysis and detection techniques, compiler techniques.

ZHANG Zhao-Qing, born in 1938, professor, Ph. D. supervisor. Her main research interests include compiler techniques and programming environment.

FENG Xiao-Bing, born in 1969, professor, Ph. D. supervisor. His main research interests include compiler techniques and programming environment.

Background

The work attributes to the project "The Compiler-Based Framework for the Development of High-Reliable Software", which is supported by the National High Technology Research and Development Program (863 Program) of China under Grant No. 2008AA01Z115.

Software plays a pivotal role in our society. But the poor quality of software is one of the big problems. Nowadays, the increasing importance of software correctness has led to a large body of research aimed at identifying violations of temporal safety properties. However, the current static program checking tools are not widely used due to the limits of either their precision or scalability.

In this paper, we propose a new approach to improve both of the precision and scalability of static program checking by adopting FSM slicing. FSM slicing integrates the traditional slicing techniques with temporal safety property checking, therefore not only reduces the size of programs being checked without checking precision loss but also simplifies the structure of the programs. Such reduction and simplification can reduce the complexity of the program analysis used by the program checking further. So the more precise program information could be obtained. With the help of program reduction and precise information, both of the precision and scalability of static program checking on temporal safety properties could be improved.