Software Engineering Group
Department of Computer Science
Nanjing University
http://seg.nju.edu.cn

Technical Report No. NJU-SEG-201T-CJ-001

201TπCJ-001

遍历常用数据结构的循环摘要的自动生成方法及其应用

翟娟, 汤震浩, 李彬, 赵建华, 李宣东

软件学报

遍历常用数据结构的循环摘要的自动生成方法及其应用

翟 娟[1,2]，汤震浩[1,3]，李 彬[1,3]，赵建华[1,3]，李宣东[1,3]

[1](计算机软件新技术国家重点实验室(南京大学)，江苏 南京 210023)

[2](南京大学 软件学院，江苏 南京 210023)

[3](南京大学 计算机科学与技术系，江苏 南京 210023)

通讯作者：赵建华，E-mail: zhaojh@nju.edu.cn

摘 要：采用形式化方法证明软件的正确性是保障软件可靠性的有效方法，而对循环语句的分析与验证是形式化证明中的关键，对循环语句的处理一直是程序分析与验证中的一个难点问题。本文提出使用循环语句修改的内存地址和这些内存地址中存放的新值来描述循环语句的执行效果，并将该执行效果定义为循环摘要。同时，本文提出了一种自动生成循环摘要的方法，可以为操作常用数据结构的循环自动生成循环摘要，包含嵌套循环。此外，基于循环摘要，我们可以自动生成循环语句的规约，包括循环不变式、循环的前置条件以及循环的后置条件。我们已经实现了自动生成循环摘要以及循环规约的方法，并将它们集成到验证工具 Accumulator 中，实验表明，我们的方法可以有效地生成循环摘要，并生成多种类型的规约，从而辅助软件程序的形式化证明，提高验证的自动化程度和效率，减轻验证人员的负担。

关键词：循环摘要；循环不变式；前置条件；后置条件；程序验证

中图法分类号：TP311

Automatic Loop Summarization and Application

ZHAI Juan[1,2], TANG Zhen-hao[1,3], Li Bin[1,3], ZHAO Jian-hua[1,3], Li Xuan-dong[1,3]

[1](State Key Laboratory for Novel Software Technology (Nanjing University), Nanjing 210023, China)

[2](Software Institute, Nanjing University, Nanjing 210023, China)

[3](Department of Computer Science and Technology, Nanjing University, Nanjing 210023, China)

Abstract: Formal verification is an effective method to guarantee software reliability by proving the correctness of a program. Analyzing and verifying loops which are important and frequently-used statements is not only vital for formal verification, but also a hot topic in the research area of software development. This paper proposes using memories modified by a loop and new values stored in these memories after executing the loop to describe the execution effect of the loop. Such execution effect is defined as loop summary. Also this paper proposes an approach to automatically synthesize loop summaries for loops manipulating commonly-used data structures, including nested loops. Based on loop summaries, specifications can be generated automatically, including loop invariants, preconditions and post-conditions of loops. We have implemented the proposed approach and integrated it into the code-verification tool Accumulator. We have evaluated the approach with a variety of programs, and the results show that our approach is able to generate loop summaries and different kinds of specifications, which eases the verification task by reducing the burden for programmers and improves the automatic level and efficiency.

Key words: loop summary; loop invariant; precondition; post-condition; program verification

随着计算机科学技术的快速发展，软件已经深入到人类社会生产和生活的各个方面，涉及到学习、生活、生产、医疗、军事和环保等，为我们的生活带来了极大的便利，但同时软件的任何错误也都可能导致非常严重的后果，因此软件的正确性和可靠性尤为重要。如何保证软件的正确性、提高软件的可靠性是计算机科学中一个非常重要的研究课题，也是推动计算机科学发展的重要动力之一。

目前保证软件可靠性的方法主要分为两种，分别是软件测试(Software Testing)和形式化方法(Formal Method)。软件测试通过运行程序来检验程序是否满足规定的需求或者预期结果与实际结果之间的差别，在一定程度上可以保证软件的可靠性，但是软件测试只能发现程序中的错误，而无法保证程序没有错误。形式化方法通过形式规约(Formal Specification)来描述程序的行为或者程序应该满足的性质，采用形式化验证(Formal Verification)来验证已有的程序是否满足规约。形式化方法是通过数学方法严格证明一个程序的正确性，确保程序没有错误。自从 1967 年，Floyd 提出采用形式化方法证明程序的正确性[1]之后，许多著名的

学者都致力于形式化分析与验证的研究工作中，并取得了重大发展。1969 年，Hoare 在 Floyd 的基础上提出了著名的 Hoare 逻辑[2]，提出了一套公理系统来验证程序的正确性。Hoare 逻辑的核心是 Hoare 三元组，表示为{P} S {Q}，P、Q 均为逻辑公式，分别是程序 S 的前置条件和后置条件，即规约。Hoare 三元组表示当 S 执行前 P 成立，那么 S 执行并终止后有 Q 成立。1976 年，Dijkstra 提出了最弱前置谓词(Weakest Precondition)和谓词转换器(Predicate Transformer)的概念[3]，通过一个映射，将一个程序执行后的状态集合转换成同一程序执行前所有可能的状态集合，这些状态集合即规约的集合。最弱前置条件是指保证一条语句执行正常结束并满足语句的后置条件的最弱的前提条件。最弱前置条件是一个谓词公式，通常用wp(S, Q)表示。我们可以通过证明P ⇒ wp(S, Q)来证明{P} S {Q}。对于基本的程序结构，如赋值语句、分支语句和顺序语句，Hoare 逻辑给出了生成最弱前置谓词的方法，但对于循环语句，并没有通用的规则可以为其生成最弱前置谓词，Hoare 提出使用一个断言 *I* 构造规则来验证循环程序，将 *I* 描述为“在任意一次执行循环体前后均为真的断言”。Gries[4]将该类断言定义为循环不变式，即在循环体的每次执行前后均为真的谓词。循环不变式是理解、证明和推导循环程序的基础。由此可见，规约是理解程序和验证程序的基础，在形式化证明中具有举足轻重的作用，因此，规约的生成是形式化方法的关键，而生成循环语句的规约更是重中之重。循环语句的形式多种多样，循环中可能还嵌套循环，而且循环处理的数据类型不仅有简单的类型，还包括数组或者自定义的复杂数据类型，因此无论是分析循环语句还是生成循环语句的规约都是非常困难的。众多的研究者们致力于对循环语句进行分析，分析循环语句一直是一个非常具有挑战性以及创造性的难题，是形式化证明领域中的核心问题之一。

对循环语句的分析验证工作主要集中在自动生成循环不变式，然后基于循环不变式来证明循环的正确性、安全性等。Hoare 逻辑指出要得到合适的循环不变式通常需要人工参与，现在的工作只能对特定的情形提出一些启发式的方法，没有通用的方法对所有的情况都能得到正确的结果[5]。目前生成循环不变式的方法主要分为基于迭代不动点(Fixed Point)的方法、基于参数化模板(Template)的方法以及基于动态执行(Dynamic Execution)的方法。基于迭代不动点的方法是进行连续计算直到发现程序状态不再变化时，计算结束。这种方法通常需要使用加宽算子（Widening）、提高抽象粒度等方法来保证和加速收敛的过程，会带来精度损失问题。基于参数化模板的方法主要针对线性数值计算程序，计算时首先指定目标循环不变式的模板形式，其中变量的系数未知，然后求解模板中的系数从而得到循环不变式。这种方法需要用户指定模板形式，不仅给用户带来负担，而且用户可以提供的模板有限，因此，这种方法的适应性不强。Daikon[6]通过动态执行程序的方法来生成循环不变式。这种方法依赖于测试用例，无法保证生成的循环不变式一定是正确的。

关于循环语句的另一个研究热点是自动生成循环语句的摘要，然后将循环语句的摘要应用于停机检验、软件的动态分析、缺陷检测等领域中。通常情况下，不同的研究领域对循环摘要的定义也有所不同：[7]将循环摘要定义为转换不变式，即循环语句的前置条件和后置条件之间的关系，并将转换不变式应用于程序的终止性分析中；[8]将循环摘要定义为一组符号化的执行路径，用于辅助动态测试的生成；[9]将循环摘要定义为符号化的谓词转换，用于发现程序中的缓冲区溢出问题。根据循环摘要定义的不同，生成循环摘要的方法也各不相同，主要分为三种方法，分别是动态执行循环生成循环摘要、基于抽象解释[10]生成循环摘要以及计算符号化的抽象转换生成循环摘要。采用动态执行的方法依赖于测试用例，生成的循环摘要可能有偏差。基于抽象解释的方法需要逼近不动点，这会引起精度问题。

通过对*memcached[11]*、*Apache httpd[12]* 和*nginx[13]*等开源软件中操作常用数据结构的循环程序进行统计分析，我们发现这类循环程序中约百分之八十的程序是对被操作的数据结构的元素进行依次遍历来完成预期的功能的，因此，我们认为为这类循环自动生成循环摘要以及规约可以辅助很多实际的应用程序的验证过程，从而提高程序验证的自动化程度以及验证的效率，给验证人员减轻负担并且降低因为人工提供循环规约带来的出错概率。

本文将语句的摘要定义为语句修改的内存地址以及语句执行结束以后内存地址中存放的新值。基于该定义，本文提出了自动生成程序语句的摘要的方法，可以高效地为赋值语句、顺序语句、条件语句以及操作常用数据结构的循环语句自动生成摘要。此外，根据摘要，我们还可以生成语句的规约，包括后置条件、

前置条件以及循环不变式，从而辅助程序的形式化验证过程。我们的分析方法所需要的时间和循环的个数是线性关系，相比于以往借助于抽象解释的方法，效率更高。本文所处理的循环是对数据结构中的元素进行依次遍历进行操作的循环，可以嵌套循环结构和分支结构。本文处理的数据结构包括封闭整数区间、一维数组、二维数组和非循环单链表。

本文第 1 节简单介绍了理论背景，主要是本文工作中使用的 Scope 逻辑的相关内容。第 2 节给出了摘要的定义以及自动生成摘要的方法。第 3 节介绍了摘要的实际应用，即生成程序语句的规约。第 4 节是实现的简单介绍，并给出了一个实例，使用本文的方法为该实例生成循环摘要以及程序规约。第 5 节是相关工作的比较。最后是对本文工作的总结。

1 理论背景

本节将简单地介绍 Scope 逻辑[14]以及 Scope 逻辑中的一些重要概念。

1.1 Scope 逻辑

本节将对 Scope 逻辑进行简单的介绍，Scope 逻辑是本文工作的基础。Scope 逻辑是对 Hoare 逻辑的扩展，可以处理指针程序和自定义的递归数据结构。

Scope 逻辑的基本思想在于一个表达式 e 的值决定于一个有限的内存集合所存储的值。如果该内存集合中的内存没有被语句修改，那么 e 的值将保持不变，这个内存集合用M(e)表示。基于 Scope 逻辑，我们开发了一个交互性验证工具 Accumulator[15]，采用携带证明的方式对程序进行证明。图 1 给出了一个示例程序，该程序有三个子目标，分别是：将数组 b 中所有元素的值相加得到的和赋值给变量 sum，将数组 a 的第 k 个元素来存放数组 b 中前 k 个元素的和，以及将数组 b 中的元素全部赋值为 0。位于语句前后的是程序点以及描述该程序点上的程序状态的公式。

```
{1: a ≠ b}
s = 100;
{2: s == 100, a ≠ b}
i = 0;
{3: i == 0, s == 100, a ≠ b}
sum = 0;
{4: sum == 0, i == 0, s == 100, a ≠ b}
while(i < s)
{
    {5: i < s, s == 100, sum == (∑_{x=0}^{i-1} b[x] @4)}
    sum = sum + b[i];
    {6: sum == sum@5 + b[i], i < s, s == 100}
    a[i] = sum;
    {7: a[i] == sum@6, sum == sum@5 + b[i], i < s, s == 100}
    b[i] = 0;
    {8: b[i] == 0, a[i] == sum@6, sum == sum@5 + b[i], i < s, s == 100}
    i = i + 1;
    {9: i == i@8 + 1, b[i − 1] == 0, a[i − 1] == sum@6, sum == sum@5 + b[i − 1], s == 100}
}
{10: sum == (∑_{x=0}^{s-1} b[x])@4, ∀x ∈ [0, s − 1]. a[x] == (∑_{y=0}^{x} b[y])}
```

Fig.1 A program operating arrays

图 1 数组操作程序

1.2 自定义函数及相关性质

Scope 逻辑不仅可以处理诸如int、bool等基本数据类型，还可以处理数组以及用户自定义的递归结构类型，例如，用户可以定义结构类型“struct Node{∗ Node next; int data};”用于操作链表结构的程序。此外，Scope 逻辑允许用户自己定义函数描述自定义数据类型的性质。例如，图 2 中给出了四个用户自定义的

递归函数，描述了非循环单链表的性质，其中setof(type t)表示类型为 type 的一系列元素的集合。函数isList描述了如果一个节点 x 是空指针或者 x 的 next 字段指向一个非循环单链表，那么 x 是一个非循环单链表。函数nodes描述了一个节点集合，囊括了非循环单链表 x 的所有节点，如果 x 是空指针，那么该节点集合为空集。函数isListSeg描述了如果从节点 x 出发，沿着 next 字段可以到达节点y，那么从节点 x 到节点 y 是一个非循环单链表的片段。函数nodesSeg囊括了从节点 x 到节点 y 的所有节点，不包含节点 y。

$$\text{bool } \textbf{isList}(*\,\text{Node x}) \coloneqq (\text{x} == \text{null})?\,\text{true}:\text{isList}(\text{x} \to \text{next})$$
$$\text{setof}(*\,\text{Node})\ \textbf{nodes}(*\,\text{Node x}) := (\text{x} == \text{null})?\,\emptyset:(\{\text{x}\} \cup \text{nodes}(\text{x} \to \text{next}))$$
$$\text{bool } \textbf{isListSeg}(*\,\text{Node x},*\,\text{Node y}) := (\text{x} == \text{null})?\,\text{false}:((\text{x} == \text{y})?\,\text{true}:\text{isListSeg}(\text{x} \to \text{next},\text{y}))$$
$$\text{setof}(*\,\text{Node})\ \textbf{nodesSeg}(*\,\text{Node x},*\,\text{Node y}) := (\text{x} == \text{null})?\,\emptyset:((\text{x} == \text{y})?\,\emptyset:(\{\text{x}\} \cup \text{nodesSeg}(\text{x} \to \text{next},\text{y})))$$

Fig.2 User-defined recursive functions

图 2 自定义递归函数

一阶逻辑无法直接处理用户自定义的递归函数，因此，我们需要提供一些性质来支持逻辑推导。例如，表 1 给出的性质用来支持图 2 中自定义函数的局部推导。以第一个性质为例，它描述了如果一个表达式 x 是 null，那么 x 是一个非循环单链表，而且 x 的节点集合为空。

Table 1 Properties of acyclic singly-linked lists

表 1 非循环单链表的相关性质

编号	性质
1	$\forall x(x == null) => (isList(x) \wedge nodes(x) == \emptyset)$
2	$\forall x(x \neq null \wedge isList(x)) => (nodes(x) == \{x\} \cup nodes(x \to next) \wedge (x \notin nodes(x \to next)))$
3	$\forall x \forall y(x \neq null \wedge y == null) => (nodesSeg(x,y) == nodes(x))$
4	$\forall x \forall y(x == y) => (nodesSeg(x,y) == \emptyset)$
5	$\forall x \forall y(isList(x) \wedge isListSeg(x,y)) => (isList(x) \wedge nodes(x) == nodesSeg(x,y) \cup nodes(y))$

1.3 特殊类型表达式

本节主要介绍 Scope 逻辑中定义的特殊类型的表达式，包括程序点相关表达式、投影表达式以及量化表达式。

1.3.1 程序点相关表达式

在 Scope 逻辑中，我们采用携带证明的方式对程序进行验证，我们为每一条语句的前后指定一个独一无二的编号，用于识别不同的程序点，描述当前程序状态的公式写在这些程序点上。对于任意的顺序语句 s1; s2，语句 s1 之后的程序点正是语句 s2 之前的程序点。程序在执行的过程中会经过这些程序点。位于某个程序点上的一个公式意味着程序每次经过该程序点上时，这个公式成立，也就意味着该程序在当前这个程序点上满足该公式描述的性质。

当我们只考虑待证明程序中的某条语句 s 时，程序规约可以描述为形如{i: P} s {j: Q}的 Hoare 三元组。这里，i 和 j 分别是位于语句 s 之前的程序点和位于语句 s 之后的程序点，P 和 Q 分别表示位于程序点 i 和程序点 j 上的公式集合。

Scope 逻辑中定义了程序点相关表达式，用于描述不同程序点上程序状态之间的关系。如果程序执行到程序点 i 之前必须先经过程序点 j，那么我们可以在程序点 i 上用表达式e@j表示程序最后一次在程序点 j 上时表达式 e 的值，当位于程序点 j 上时，e@j等于 e，当位于非 j 的程序点上时，e@j被视为一个常数。在图 1 中，程序执行到程序点 9 时必然先经过程序点 1，因此我们可以在程序点 9 上用形如 exp@1 的表达式来描述 exp 在程序点 1 上时的值。例如，在图 1 中，位于程序点 7 上的公式a[i] == sum@6描述了元素 a[i]的值等于变量 sum 在程序点 6 上的值。

对于任意一条语句 s，在 s 之前的程序点上的公式是该语句的前置条件，在语句 s 之后的程序点上的公式是该语句的后置条件。例如，在图 1 中，循环语句 while 之前和之后的程序点的编号分别为 4 和 10，

公式sum == 0位于程序点 4 上，是 while 语句的前置条件，公式sum == $(\sum_{x=0}^{s-1} b[x])$@4位于程序点 10 上，是 while 语句的后置条件。

1.3.2 投影表达式

形如$\lambda x.\exp[set]$的表达式被称为投影表达式，$\lambda x.\exp$相当于一个匿名函数，set 是变量 x 的值域，$\lambda x.\exp[set]$表示将 set 中的每个元素 x 调用函数$\lambda x.\exp$得到的结果。例如，表达式$\lambda x.x \to data[nodes(sl)]$表示单链表 sl 中的每个节点的 data 字段的集合，即单链表 sl 的所有节点存储的数据的集合。

1.3.3 量化表达式

形如$\forall x \in set.\exp$的表达式被称为量化表达式，它表示对于集合 set 中的任何一个元素 x 都有表达式 exp 成立。例如，表达式$\forall x \in nodes(sl).x \to data > 0$表示单链表 sl 中的每个节点的 data 字段的值都大于 0。

2 摘要的自动生成方法

本节将全面阐述为语句自动生成摘要的方法。首先，给出摘要以及等价表达式的定义。接着，介绍生成等价表达式的方法。最后，详细介绍摘要的自动生成方法。

2.1 摘要的定义

执行一条程序语句实际上就是操作和该语句相关的有限的内存集合。在本文中，内存是使用内存表达式来表示的，我们将内存表达式归纳为两种类型，分别定义如下：

定义 1. 单一内存表达式（Single Memory Expression）是类型为 *P(t)*的内存表达式，即指向类型 *t* 的指针类型。其中，*t* 可以是整型、布尔型或者其它指针类型。

定义 2. 集合内存表达式（Set Memory Expression）是形如$\lambda x.e[set]$的内存表达式。其中 *e* 是一个内存表达式，*set* 是一个集合表达式，*x* 的值域是 *set*。

例如，在图 1 中，&sum是单一内存表达式，它指向的内存地址中存放的内容类型为int；$\lambda x.$&a[x][0,99]是集合内存表达式，它表示了一系列的内存地址，其中每一个内存地址存放数组 *a* 中的一个元素。

对于任意一个内存表达式 *m*，我们为其定义了**核心（Kernel）**和**界定变量（Range Variables）**，分别表示为 *κ(m)*和 *γ(m)*，定义分别如图 3 和图 4 所示。

$$\kappa(m) = \begin{cases} \{m\}，\text{当 m 是单一内存表达式时} \\ \kappa(m')，\text{当 m 是集合内存表达式且形如}\lambda x.m'[set]\text{时} \end{cases}$$

Fig.3 Definition of kernel κ(m)

图 3 核心κ(m)的定义

$$\gamma(m) = \begin{cases} \phi，\text{当 m 是单一内存表达式时} \\ \{m\} \cup \gamma(m')，\text{当 m 是集合内存表达式且形如}\lambda x.m'[set]\text{时} \end{cases}$$

Fig.4 Definition of kernel γ(m)

图 4 界定变量γ(m)的定义

从直观上看，执行一条语句所产生的效果是更新部分有限的内存，在这些内存中存放新的内容，而保持其它内存中的内容不变。我们将执行一条语句对内存所产生的影响定义为该语句的摘要。该摘要反映了执行该语句所修改的内存地址和执行该语句之后存储在这些内存中的新值。具体定义如下：

定义 3. 语句 *s* 的**摘要(Summary)**定义为有限的一组元组，表示为 *τ(s)*。其中每个元组表示为$<\mathrm{m},\mathrm{v}>$，该元组表示内存表达式 *m* 和值表达式 *v* 之间的映射关系。内存表达式 *m* 表示执行语句 *s* 所修改的内存，相应的值表达式 *v* 是执行语句 *s* 之后存储在内存 *m* 中的新值。这里需要强调的是表达式 *m* 和 *v* 均是在执行语句 *s* 之前的程序点上进行求值的。

当 *m* 属于单一内存表达式时，那么存储在 *m* 中的新值即为 *v*。当 *m* 属于集合内存表达式时，假设 *m* 形如$\lambda x_1.(\ldots(\lambda x_k.e[set_k])\ldots)[set_1]$，那么对于集合内存中的任意一个简单内存$m[v_1/x_1][v_2/x_2]\ldots[v_k/x_k]$而言，相应的新值为$v[v_1/x_1][v_2/x_2]\ldots[v_k/x_k]$，其中对于任意的$\mathrm{i}=1,2,3,\ldots,\mathrm{k}$，$v_i$属于集合$set_i$。在本文中，当位于内存地址 *m* 中的新值不能确定时，我们使用特殊的占位符 ϱ 来表示值 *v*。

我们以图 1 为例，该程序中循环语句的摘要如下所示：

$$\left\{\begin{array}{c} <\&\mathrm{i},\mathrm{s}>, \\ <\&\mathrm{sum},\sum_{x=0}^{s-1}\mathrm{b}[x]>, \\ <\lambda x.(\&\mathrm{b}[x])[0,\mathrm{s}-1],0>, \\ <\lambda x.(\&a[x])[0,\mathrm{s}-1],\sum_{y=0}^{x}\mathrm{b}[y]> \end{array}\right\}$$

前两个元组说明循环语句的执行效果是将 $\sum_x^{s-1}b[x]$和 *s* 分别赋值给变量 *sum* 和 *i*。其中，$\sum_x^{s-1}b[x]$和 *s* 是在程序点 4（循环入口点）上进行求值的。第三个元组说明循环语句执行结束以后，数组 *b* 下标从 0 到 *s-1* 的所有元素被赋值为 0。最后一个元组说明循环语句执行结束以后，数组 *a* 下标从 0 到 *s-1* 的所有元素被赋值为$\sum_{y=0}^{x}\mathrm{b}[y]$，其中表达式$\sum_{y=0}^{x}\mathrm{b}[y]$是在程序点 4 上进行求值的。

在本文中，我们将执行语句 s 所修改的内存地址的集合表示为χ(s)。集合χ(s)中的内存表达式可能相互重叠，当两个内存表达式所表示的内存有重叠时，那么重叠的内存表达式所对应的值表达式必须是一致的。例如顺序语句“*p=1; *q=2;”的摘要表示为$\{<\mathrm{p},(\mathrm{p}==\mathrm{q})?2:1>,<\mathrm{q},2>\}$。表达式 p 和 q 可能描述的是同一个内存，当$\mathrm{p}==\mathrm{q}$成立时，它们所指向的内存中存放的新值为 2。

综上所述，我们可以看出，对于包含条件语句和循环语句在内的任何程序语句而言，一条语句的摘要实际上相当于将该语句转换为一系列抽象的赋值语句。众所周知，计算赋值语句的最弱前置条件和最强后置条件是有规则可循的，因此，通过将条件语句和循环语句进行抽象得到它们的摘要，我们就可以使用赋值语句相关的规则来处理条件语句和循环语句了。

2.2 等价表达式的定义及计算

本节将首先介绍等价表达式的定义，然后分别介绍等价表达式的计算方法和简化方法。

2.2.1 等价表达式的定义

定义 4. 如果表达式 *e′*是 *e* 的**等价表达式（Equivalent Expression）**，那么对于语句 *s* 的每次执行，表达式 *e′*在执行语句 *s* 之前的程序状态上的取值等于表达式 *e* 在执行语句 *s* 之后的程序状态上的取值。

例如，表达式 *a[j]*和 *a[i]*分别位于语句“*i=j;*”之前和之后的程序点上，显然，*a[j]*在语句“*i=j;*”执行之前的取值等于 *a[i]*在语句“*i=j;*”执行之后的取值，因此，表达式 *a[j]*是位于语句“*i=j;*”之前的程序点上的 *a[i]*的等价表达式。

实际上，等价表达式的概念是最弱前置条件的一个泛化。最弱前置条件是针对谓词而言的，而等价表达式是针对诸如一个变量的表达式而言的，谓词是表达式的一个子集。假设 *a[i]==0* 是语句“*i=j;*”的后置条件，那么 *a[j]==0* 即为 *a[i]==0* 关于语句“*i=j;*”的最弱前置条件。

2.2.2 等价表达式的计算

假设我们有赋值语句$\mathrm{v1}=\mathrm{v2}$以及位于该赋值语句之前的程序点 i 和该赋值语句之后的程序点 j，程序点 i 和程序点 j 这两个程序状态之间的差别在于内存地址(&v1)@i中存储的值。因此，对于任意的一个内存单元 x，在程序点 i 上，表达式$(\mathrm{x}\neq(\&\mathrm{v1})@\mathrm{i})?*\mathrm{x}:\mathrm{v2}@\mathrm{i}$的值和程序点 j 上*x 的值相同。鉴于此，我们提出了根据语句的摘要计算关于语句的等价表达式的方法。给定程序语句 *s*，以及位于语句 *s* 之后程序点上的表

达式 e，我们可以递归地计算表达式 e 位于语句 s 之前的程序点上的等价表达式 e'，用 $\epsilon(e, s)$表示。表 2 给出了 $\epsilon(e, s)$的计算规则。列“e”给出了表达式的形式，列“ϵ(e, s)”展示了如何计算表达式 e 关于语句 s 的等价表达式，j 是位于语句 s 之后的程序点。

Table 2 Calculation rules for equivalent expression

表 2 等价表达式的计算规则

e	**ϵ(e, s)**	**e**	**ϵ(e, s)**
常数/量化的变量	e	$e_0?\,e_1:e_2$	$\epsilon(e_0,s)?\,\epsilon(e_1,s):\epsilon(e_2,s)$
e'@k (k ≠ j)	e'@k	op e' (op ≠∗)	op ϵ(e', s)
&v	&v	$e_1\ op\ e_2$	$\epsilon(e_1,s)\ op\ \epsilon(e_2,s)$
∗ e'	ϵ(∗ (ϵ(e', s)), s)	&(e'. n)	&(ϵ(&e', s) → n)
v	ϵ(∗ (&v), s)	&(e' → n)	&(ϵ(e', s) → n)
e'. n	ϵ(∗ (&e'. n), s)	$\&(e_1[e_2])$	$\&(*\,\epsilon(\&e_1,s)[\epsilon(e_2,s)])$
e' → n	ϵ(∗ (&e' → n), s)	$\lambda x.\,e_1[e_2]$	$\lambda x.\,\epsilon(e_1,s)[\epsilon(e_2,s)]$
$\boldsymbol{e_1[e_2]}$	ϵ(∗ (&e_1[e_2]), s)	$\forall x \in e_1.\,e_2$	$\forall x \in \epsilon(e_1,s).\,\epsilon(e_2,s).$

从表 2 中，我们可以看出计算等价表达式 $\epsilon(e, s)$是一个递归过程，表达式 e 关于语句 s 的等价表达式是通过递归地计算 e 的子表达式关于语句 s 的等价表达式来求解的。因此，只要我们可以计算 e'关于语句 s 的等价表达式，其它表达式关于语句 s 的等价表达式都可以迎刃而解。

表达式 $\epsilon(*e', s)$的值取决于 e'所指向的内存地址是否被语句 s 所修改。如果没有修改，那么等价表达式 $\epsilon(*e', s)$的值就是 e'在语句 s 之前的程序点上的值，否则，$\epsilon(*e', s)$的值与 e'所指向的内存地址中存放的新值密切相关。在第 2.1 节中已经指出，τ(s)表示被语句 s 所修改的内存地址以及这些内存中存放的新值的元组集合，因此我们可以通过将 e'所指向的内存地址依次与τ(s)中的每个元组进行比较分析从而计算 $\epsilon(*e', s)$的值。假设 p 表示位于语句 s 之前的程序点，对于τ(s)$中的任意一个元组< m, v >，∗ (e'@p)的值表示为 c? e'':∗ (e'@p)，其中 c 是一个布尔表达式，用于判断内存地址e'@p是否和 m 所指定的内存地址有交集，表达式 e''的值根据表达式 v 的值而定。

- 当 m 属于单一内存表达式时，相应的等价表达式为 (m == e')@p? v@p : ∗ (e'@p)；
- 当 m 属于集合内存表达式，$\gamma(m) = \{x_1, x_2, \ldots, x_n\}$，且$x_1$、$x_2$、…、$x_n$相应的值域依次为$set_1$、$set_2$、…、$set_n$时，
  - 如果存在 n 个表达式e_1、e_2、…、e_n，使得表达式 e'和将核心 $\kappa(m)$中的界定变量x_1、x_2、…、x_n替换为e_1、e_2、…、e_n得到的表达式相同，那么公式e ∈ m成立当且仅当对于任意的i = 1,2, … , n有$e_i \in set_i$成立。在这种情况下，相应的等价表达式为
    $\big((e_1 \in set_1) \wedge (e_2 \in set_2) \wedge \ldots \wedge (e_n \in set_n)\big)$@p? $(v[e_1/x_1][e_2/x_2] \ldots [e_n/x_n])$@p:∗ (e'@p)
  - 否则，相应的等价表达式为$(e' \in m)$@p? ϱ:∗ (e')@p

2.2.3 等价表达式的简化

使用前面一节的方法所生成的等价表达式可能包含程序点相关表达式、条件表达式以及特殊表达式ϱ。这些表达式的存在使得等价表达式相对复杂，不便于处理，因此，本小节将对等价表达式进行简化。

等价表达式是在语句之前的程序点上进行求值的，假设该程序点用 p 表示，由于在任意一个程序点 i 上有e == e@i成立，因此我们可以删除等价表达式中的所有子表达式@p来简化等价表达式。

对于任意的条件表达式c? e_1:e_2而言，当条件 c 成立时，该条件表达式可以被简化为e_1，相反地，当条件¬c成立时，该条件表达式可以被简化为e_2。下面给出一些用于判断 c 或者¬c是否成立的依据。

- 语句 s 的已知的前置条件；
- 程序语言的内存分布的性质，如表 3 所示；
- 待简化的子表达式中包含的隐藏条件。

- 对于任意的条件表达式c? e_1: e_2，当简化e_1时，意味着 c 是成立的，当简化e_2时，意味着¬c是成立的；
- 对于形如$\lambda x.e[set]$和∀x ∈ set. e的表达式，当简化 e 时，x ∈ set是成立的，可以用作简化的依据。

Table 3 Axioms for memory layout and memory access

表 3 内存布局以及内存访问的公理

名称	公理	名称	公理
DEREF-REF	$*\,\&e == e$	REF-DEREF	$e \neq null \Rightarrow *e == e$
PVAR1	$\&v \neq null$	PVAR2	$\&v_1 \neq \&v_2$
PVAR3	$\&v \neq \&r \rightarrow n$	PVAR4	$\&v \neq \&a[i]$
REC1	$r \neq null \Rightarrow \&r \rightarrow n \neq null$	REC2	$r_1 \rightarrow n == r_2 \rightarrow n \Rightarrow r_1 == r_2$
REC3	$r_1 \rightarrow n_1 \neq r_2 \rightarrow n_2$	ARR1	$a \neq null \wedge (0 \leq i < c) \Rightarrow \&(*a)[i] \neq null$
ARR2	$(\&((*a_1)[i_1]) == \&((*a_2)[i_2])) \Leftrightarrow (a_1 == a_2 \wedge i_1 == i_2 \wedge 0 \leq i_1, i_2 \leq c)$	ARR-REC	$\&a[i] \neq \&r \rightarrow n$

在对等价表达式进行简化时，我们使用 SMT 求解器 Z3[16]进行公式的推导证明。经过上述的简化过程，如果等价表达式中仍然包含表达式ϱ，那么我们就将该表达式简化为ϱ。

2.2.4 实例研究

假设图 1 中位于循环语句之后的程序点 10 上有一个后置条件$\forall x \in [0, s-1].a[x] > 0$，我们以该公式为例展示等价表达式的计算方法，在后面的介绍中，我们使用 L 表示该循环语句。

在表达式$\forall x \in [0, s-1].a[x] > 0$中，涉及到内存访问的子表达式分别是 s 和$a[x]$。首先，我们将 s 和$a[x]$分别转换为$*(\&s)$和$*(\&a[x])$，然后分别计算它们关于摘要 $\tau(L)$中每一个元组的等价表达式。

针对 $\tau(L)$中的第一个元组$< \&sum, \sum_{x=0}^{s-1} b[x] >$而言，$\forall x \in [0, s-1].a[x] > 0$对应的等价表达式为

$$\forall x \in \left[0, \left((\&sum == \&s)@4?\left(\sum_{x=0}^{s-1} b[x]\right)@4{:}\, s\right) - 1\right].((\&sum == \&a[x])@4?(\left(\sum_{x=0}^{s-1} b[x]\right)@4{:}\, a[x])) > 0$$

根据表 3 中的公理 PVAR2 和公理 PVAR4，我们可以推断出条件¬(&sum == &s)@4和¬(&sum == &$a[x]$)@4成立，因此该等价表达式被化简为$\forall x \in [0, s-1].a[x] > 0$，即原表达式。同样地，原表达式关于元组$< \&i, s >$和$< \lambda x.(\&b[x])[0, s-1], 0 >$的等价表达式仍然是自身。针对最后一个元组$< \lambda x.(\&a[x])[0, s-1], \sum_{y=0}^{x} b[y] >$，我们生成等价表达式

$$\forall x \in \left[0, \left((\&s \in \lambda x.(\&a[x])[0, s-1])@4?\, \varrho{:}\, s\right) - 1\right].\left((x \in [0, s-1])@4?\left(\sum_{y=0}^{x} b[y]\right)@4{:}\, a[x]\right) > 0$$

因为等价表达式是在语句之前的程序点上计算的，在本例中，即在程序点 4 上进行计算的，所以我们可以删除等价表达式中的子表达式@4，此外，根据表 3 中的公理 PVAR4，我们可以进一步将该表达式进行简化，从而得到最终的等价表达式$\forall x \in [0, s-1].\sum_{y=0}^{x} b[y] > 0$。

2.3 非循环语句的摘要的自动生成

本节将详细介绍如何生成赋值语句、顺序语句以及条件语句的摘要。

2.3.1 赋值语句的摘要生成

赋值语句$e_1 = e_2$的摘要是只包含一个元组的集合，即$\{< \&e_1, e_2 >\}$，这里的$\&e_1$和e_2均是在该赋值语句之前的程序点进行求值的。例如，赋值语句*"sum=sum+b[i];"*的摘要为$\{< \&sum, sum + b[i] >\}$。

2.3.2 顺序语句的摘要生成

假设 s 表示顺序语句“s_1；s_2;”，那么被语句 s 修改的内存地址$\chi(s)$是被语句 s_1 所修改的内存地址$\chi(s_1)$和语句 s_2 所修改的内存地址$\chi(s_2)$的并集。因为$\chi(s_2)$是在语句 s_2 之前的程序点也就是语句 s_1 之后的程序点上进行求值的，所以我们首先要计算$\chi(s_2)$在语句 s_1 之前的程序点上的等价表达式。因此$\chi(s)$的值即为表达式$\chi(s_1) \cup \{\epsilon(e, s_1) | e \in \chi(s_2)\}$的值。对于集合$\chi(s)$中的任意一个内存 m，在摘要 $\tau(s)$中它对应的新值可以通过$\epsilon(\epsilon(*(\kappa(m)@p), s_2), s_1)$来计算，其中 p 表示位于语句 s 之前的程序点。例如，摘要 τ(“t=t+2;a[t]=0”)为集合$\{< \&t, t+2 >, < \&a[t+2], 0 >\}$。

2.3.3 条件语句的摘要生成

条件语句的摘要可以通过该语句的分支语句的摘要进行合成。假设 s 表示条件语句“if c then s_1 else s_2;”。那么语句 s 所修改的内存地址$\chi(s)$是它的两个分支语句所修改的内存地址的并集，即$\chi(s_1) \cup \chi(s_2)$。对于$\chi(s)$中的任何一个内存单元 m，它所对应的新值可以根据表 4 进行计算，其中，v_1和v_2分别是内存 m 在摘要$\chi(s_1)$和$\chi(s_2)$中的新值，p_1 和 p_2 分别表示位于 else 分支和 then 分支之前的程序点。

Table 4 Calculation rules for values of conditional statements

表 4 条件语句中内存的新值的计算规则

条件	新值
$\boldsymbol{m \in \chi(s_1) \wedge m \in \chi(s_2)}$	$c?\, v_1 : v_2$
$\boldsymbol{m \in \chi(s_1) \wedge m \notin \chi(s_2)}$	$c?\, v_1 : \epsilon(*(\kappa(m)@p_1), s_2)$
$\boldsymbol{m \notin \chi(s_1) \wedge m \in \chi(s_2)}$	$c?\, \epsilon(*(\kappa(m)@p_2), s_1) : v_2$

```
if(cur→data > 0){
        absSum = absSum + cur→data;
        posSum = posSum + cur→data;
}else{
        absSum = absSum - cur→data;
        negSum = negSum + cur→data;
}
```

Fig.5 Program *summation* of an acyclic singly-linked list

图 3 非循环单链表的求和程序

以图 5 中的条件语句为例，then 分支和 else 分支都是顺序语句，可以通过 2.3.2 节中的方法计算它们的摘要，结果分别为

$$\left\{ \begin{array}{l} < \&absSum, absSum + cur \rightarrow data >, \\ < \&posSum, posSum + cur \rightarrow data > \end{array} \right\}$$

和

$$\left\{ \begin{array}{l} < \&absSum, absSum - cur \rightarrow data >, \\ < \&negSum, negSum + cur \rightarrow data > \end{array} \right\}$$

根据表 3 中的计算规则，我们计算出该条件语句的摘要为

$$\left\{\begin{array}{l} < \&absSum, cur \to data > 0 ? absSum + cur \to data : absSum - cur \to data >, \\ < \&posSum, cur \to data > 0 ? posSum + cur \to data : posSum >, \\ < \&negSum, cur \to data > 0 ? negSum : negSum + cur \to data > \end{array}\right\}$$

2.4 循环语句的摘要生成

本节将首先介绍我们的方法所处理的循环语句类型，然后详细介绍如何生成该类循环语句的摘要。

2.4.1 循环语句的类型

我们的方法处理对某一类常用数据结构的元素进行依次遍历并进行操作的 while 循环，包括嵌套 while 循环。目前我们所处理的数据结构包括整数集合、一维数组、二维数组和非循环单链表。这类循环通常由一个循环控制变量控制着每一次迭代的执行，循环控制变量的取值范围是一个整数区间或者是某个链表结点的集合。通过对一些开源软件的统计分析，我们发现这类循环的迭代过程对于内存的更新操作遵循一定的模式。具体而言，给定一个循环语句L: while(c) s，假设该循环入口处的程序点为 q，循环体前面的程序点为 p，我们所处理的循环属于以下两种类型中的一种：

- 循环控制条件 c 形如$w \sim e$或者$e \sim w$，其中，w 是一个类型为 int 的变量，e 是一个类型为 int 的表达式，$\sim$表示一个操作符，$\sim$可以是集合$\{<, \leq, >, \geq, \neq\}$中的任意一个。此外，本文还要求循环满足下面的条件：
  1) 谓词$\epsilon(e, s) == e$在程序点 p 上成立，这意味着循环的终止条件是固定的；
  2) 如果 c 形如$w < e$，$w \leq e$，$e > w$或$e \geq w$，那么$\epsilon(w, s) == w + 1$在程序点 p 上需要成立；
  3) 如果 c 形如$e < w$，$e \leq w$，$w > e$或$w \geq e$，那么$\epsilon(w, s) == w - 1$在程序点 p 上需要成立；
  4) 如果 c 形如$w \neq e$或$e \neq w$，那么$\epsilon(w, s) == w + 1 \ \vee \ \epsilon(w, s) == w - 1$在程序点 p 上需要成立；

  在这类循环中，w 是循环控制变量，循环通过 w 控制着迭代的依次执行，显而易见，符合上述条件的循环一定会终止。例如，图 1 所示的循环满足上述条件，i 是循环控制变量，循环通过 i 依次访问数组 a 和数组 b 中的每个元素并进行相应的操作。循环控制变量 i 的初值为 0，循环在每次迭代中递增 i 的值，当 i 的值等于 100 时，循环执行结束。
- 循环控制条件 c 形如$null \neq w$或$w \neq null$，其中 w 是一个指针，指向非循环单链表的一个节点。此外，本文还要求循环满足下面的条件：
  1) 在程序点 q 上有谓词isList(w)成立；
  2) 谓词$\epsilon(w, s) == w \to next$程序点 p 上成立；
  3) 对于循环体内任意一个赋值语句$e1 = e2$，在该赋值语句之前的程序点上有$\neg(\&e1 \in M(isList(list)))$成立，其中，list 是 w 在循环入口点处的初值，M(isList(list))包含链表 list 所有节点的 next 字段的地址。该条件可以保证循环的执行过程中没有改变这些节点的 next 字段，也就意味着没有改变非循环单链表的结构。

  在这类循环中，w 是循环控制变量，循环通过 w 控制着迭代的依次执行，显而易见，符合上述条件的循环一定会终止。

循环的执行过程是通过循环控制变量对数据结构的元素进行访问的，因此循环控制变量的取值范围是至关重要的，本文为循环控制变量的取值范围定义了有效值域以及已遍历值域两个概念。

定义 5. 循环控制变量 v 的**有效值域(Valid Values)**定义为一系列值的集合，表示为ξ(v)。当 v 等于有效值域中的任意一个值时，通过该值可以访问到数据结构中的某个元素。

循环控制变量的有效值域可以通过对程序的静态分析得到。例如，在图 1 中，循环执行结束时，循环控制变量 i 的值为 100，当 i 取值100时去访问数组a会出现数组下标越界，因此 i 的有效值域是[0,99]。

定义 6. 在循环的执行过程中，我们将第一次迭代开始前到当前迭代结束后循环控制变量 v 的取值范围定义为循环控制变量的**已遍历值域(Iterated Values)**，用ς(v)表示。

1) 假设 p 指向一个非循环单链表，v 是用于访问 p 的所有节点的循环控制变量，那么 v 的已遍历值域$\varepsilon(v)$为$\{\text{nodesSeg}(\text{p}, v)\}$

2) 假设 v 是用于从左向右访问数组元素的循环控制变量，且 v 的下界为 *low*

   a) 当循环先使用 v 的值然后递增 v，那么 v 的已遍历值域$\varsigma(v)$是$[\text{low}, v-1]$。例如，在图 1 中，i 是循环控制变量，i 的已遍历值域$\varsigma(\text{i}) = [0, \text{i}-1]$

   b) 当循环先递增 v 然后使用 v 的值，那么 v 的已遍历值域$\varsigma(v)$是$[\text{low}, v]$

3) 假设 v 是用于从右向左访问数组元素的循环控制变量，且 v 的上界为 *up*

   a) 当循环先使用 v 的值然后递减 v，那么 v 的已遍历值域$\varsigma(v)[v+1, \text{up}]$

   b) 当循环先递减 v 然后使用 v 的值，那么 v 的已遍历值域$\varsigma(v)$是$[v, \text{up}]$

通过对这类循环进行分析，我们发现被循环体改变的内存地址可以被分为两类，一类地址是固定的，循环的每次迭代都会修改该内存；一类地址是非固定的，是随着循环控制变量的值的改变而改变的。我们将它们分别定义如下：

定义 7. 对于任意的内存表达式 m 而言，当$\epsilon(\text{m}, \text{s}) == \text{m}$成立时，$m$ 被称为**固定地址（Fixed Address）**，其中 s 表示循环体。

定义 8. 对于任意的内存表达式 m 而言，当$\epsilon(\text{m}, \text{s}) == \text{m}[\epsilon(\text{w}, \text{s})/\text{w}]$成立时，$m$ 被称为**移位地址（Shifting Address）**，其中，s 表示循环体，w 表示循环控制变量。

在图 1 中，内存表达式*&sum* 表示的是固定地址，而内存表达式*&a[i]*表示的是移位地址。

2.4.2 元组合成方法

接下来，我们介绍生成 2.4.1 节中描述的循环语句的摘要的具体方法。首先我们引入相关定义，然后通过对循环体的摘要中的每个元组进行一定的转换生成循环语句的摘要中的一个元组。在本节中，我们使用 L 表示一个循环语句，s 表示 L 的循环体。

$$\mu(\text{w}, \text{w}_0) = \begin{cases} \{x | x \in \xi(w) \wedge x < w_0\}, & \text{当 } w \text{ 是 int 型变量，且在每次迭代中 } w \text{ 值依次加 1 时，} \\ \{x | x \in \xi(w) \wedge x > w_0\}, & \text{当 } w \text{ 是 int 型变量，且在每次迭代中 } w \text{ 值依次减 1 时，} \\ \text{nodesSeg}(\text{w@p}, w_0), & \text{当 } w \text{ 指向一个非循环单链表的节点时} \end{cases}$$

Fig.6 Definition of $\mu(\text{w}, \text{w0})$

图 4 $\mu(\text{w}, \text{w0})$的定义

假设 w 表示循环控制变量，w_0 是 w 的有效值域$\xi(\text{w})$中的某一个值，我们将在循环的执行过程中 w 的取值为 w_0 之前的所有取值用$\mu(\text{w}, w_0)$表示，定义如图 6 所示，将在循环的执行过程中 w 取值为 w_0 之后的所有取值用$\nu(\text{w}, w_0)$表示，定义如图 7 所示

$$\nu(\text{w}, \text{w}_0) = \begin{cases} \{x | x \in \xi(w) \wedge x > w_0\}, & \text{当 } w \text{ 是 int 型变量，且在每次迭代中 } w \text{ 值依次加 1 时，} \\ \{x | x \in \xi(w) \wedge x < w_0\}, & \text{当 } w \text{ 是 int 型变量，且在每次迭代中 } w \text{ 值依次减 1 时，} \\ \text{nodesSeg}(w_0, \text{w@p}), & \text{当 } w \text{ 指向一个非循环单链表的节点时} \end{cases}$$

Fig.7 Definition of $\nu(\text{w}, \text{w0})$

图 5 $\nu(\text{w}, w_0)$的定义

基于$\mu(\text{w}, w_0)$，我们定义谓词$\mathcal{NB}(\text{e}, w_0)$来描述在循环控制变量 w 取值为 w_0 之前的每一次迭代中，表达式 e 没有被修改过，即

$$\mathcal{NB}(\text{e}, w_0) \stackrel{\text{def}}{=} (w \in \mu(\text{w}, w_0)) \Rightarrow (\epsilon(\text{e}, \text{s}) == \text{e})$$

同样地，基于$\nu(\text{w}, w_0)$，我们定义谓词$\mathcal{NA}(\text{e}, w_0)$来描述在循环控制变量 w 取值为 w_0 之后的每一次迭代中，表达式 e 没有被修改过，即

$$\mathcal{NA}(\text{e}, w_0) \stackrel{\text{def}}{=} (w \in \nu(\text{w}, w_0)) \Rightarrow (\epsilon(\text{e}, \text{s}) == \text{e})$$

假设$< \text{m}, \text{v} >$是循环体 s 的摘要 $\tau(s)$中的一个元组，我们通过对该元组进行分析操作可以生成循环语

句 L 的摘要 $\tau(L)$中的一个元组，具体做法如下：

1) 内存表达式 m 表示固定地址

当 m 表示固定地址时，这意味着循环的每一次迭代都会修改 m 所表示的内存地址，那么这个内存地址也是被循环语句 L 所修改的，因此 m 属于$\chi(\mathrm{L})$，经过循环语句的执行，m 中所存放的内容分为下面几种情况：

- 如果条件$\epsilon(\mathrm{v},\mathrm{s}) == \mathrm{v}$成立，那么 v 没有被循环体 s 所修改，也就意味着每一次迭代都将 v 所表示的内容存放到内存地址 m 中。在这种情况下，基于 $\tau(s)$中的元组$<\mathrm{m},\mathrm{v}>$，我们可以生成 $\tau(L)$中的元组$<\mathrm{m},\mathrm{c?\,v}:*\mathrm{m}>$。
- 如果 m 是单一内存表达式，v 形如$*\,\mathrm{m}\ \ \mathrm{op}\ \ \mathrm{e}$或者$\mathrm{e}\ \ \mathrm{op}*\mathrm{m}$，其中 op 属于操作符集合{+，−，×，÷，∧，∨}，而且对于$\xi(\mathrm{w})$中的任何一个值 w_0 而言，都有$\mathcal{NB}(\mathrm{e}[w_0/\mathrm{w}],w_0)$成立，那么在 $\tau(L)$中，m 所对应的新值可以根据表 5 生成。在表 5 中，第一列给出了操作符，第二列给出了根据该操作符生成的在摘要 $\tau(L)$中内存表达式 m 的值。

  例如，在图 1 中，$<\&\mathrm{sum},\mathrm{sum}+\mathrm{b[i]}>$是循环体摘要中的一个元组，&sum是固定地址，值sum + b[i]形如$*\,\mathrm{m}\ \ \mathrm{op}\ \ \mathrm{e}$，而且对于$\xi(\mathrm{i})$中的任何一个值$i_0$而言有$\mathcal{NB}(\mathrm{b}[i_0],i_0)$成立，因此根据表 5 我们可以得到在 $\tau(L)$中，&sum中存放的新值为$\mathrm{sum}+\sum_{x=0}^{s-1}\mathrm{b}[x]$。
- 在其它情况下，我们为 $\tau(L)$生成的关于内存地址 m 的元组为$<\mathrm{m},\varrho>$。

Table 5 Calculation rules for value expressions

表 5 值表达式的计算规则

操作符 *op*	$\tau(L)$中 m 的新值
+，−	$*\,\mathrm{m}\ \ \mathrm{op}\ \sum_{x\in\xi(\mathrm{w})}\mathrm{e}[x/\mathrm{w}]$
×，÷	$*\,\mathrm{m}\ \ \mathrm{op}\ \prod_{x\in\xi(\mathrm{w})}\mathrm{e}[x/\mathrm{w}]$
∧	$*\,\mathrm{m}\ \ \mathrm{op}\ \bigvee_{x\in\xi(\mathrm{w})}\mathrm{e}[x/\mathrm{w}]$
∨	$*\,\mathrm{m}\ \ \mathrm{op}\ \bigwedge_{x\in\xi(\mathrm{w})}\mathrm{e}[x/\mathrm{w}]$

2) 内存表达式 m 表示移位地址

当 m 表示移位地址时，这意味着循环的每一次迭代都会根据循环控制变量的不同取值而修改不同的内存地址。给定 $\tau(s)$中的元组$<\mathrm{m},\mathrm{v}>$，当循环控制变量 w 的取值为 w_0时，当前迭代所修改的内存地址 m 以及当前迭代结束后内存地址中存放的新值 v 分别是$\mathrm{m}[w_0/\mathrm{w}]$和$\mathrm{v}[w_0/\mathrm{w}]$，$\mathrm{m}[w_0/\mathrm{w}]$和$\mathrm{v}[w_0/\mathrm{w}]$均在当前迭代开始执行之前的程序点上进行求值的。

如果条件$\forall[w_0\in\xi(w).\mathcal{NB}(\kappa(\mathrm{m})[w_0/\mathrm{w}],w_0)$满足，那么表达式$\kappa(\mathrm{m})[w_0/\mathrm{w}]$在循环入口点的取值和循环控制变量 w 取值为 w_0的迭代开始执行之前的程序点上的取值是相同的，在这种情况下，我们通过对内存地址 m 中的循环控制变量进行全局量化来得到循环语句修改的内存地址，即$\lambda\mathrm{x}.(\mathrm{m}[\mathrm{x}/\mathrm{w}][\xi(w)])$。

此外，对于元组中的值表达式 v 而言，如果条件$\forall[w_0\in\xi(w).\mathcal{NB}(\mathrm{v}[w_0/\mathrm{w}],w_0)$和条件$\forall[w_0\in\xi(w).\mathcal{NA}(*\,\kappa(\mathrm{m})[w_0/\mathrm{w}],w_0)$都满足，那么$\mathrm{v}[w_0/\mathrm{w}]$在循环入口点的取值和$*\,\kappa(\mathrm{m})[w_0/\mathrm{w}]$在循环出口点的取值是相同的。在这种情况下，表达式$\mathrm{v}[w_0/\mathrm{w}]$就是在循环执行结束之后内存表达式$\mathrm{m}[w_0/\mathrm{w}]$中存放的新值。因此，元组$<\lambda\mathrm{x}.(\mathrm{m}[\mathrm{x}/\mathrm{w}][\xi(w)]),\mathrm{v}[\mathrm{x}/\mathrm{w}]>$是摘要 $\tau(L)$中的一个元组。否则，根据$<\mathrm{m},\mathrm{v}>$我们生成的元组为$<\lambda\mathrm{x}.(\mathrm{m}[\mathrm{x}/\mathrm{w}][\xi(w)]),\varrho>$。

以图 1 为例，i 是循环控制变量，i 的有效值域$\xi(i)$为[0,99]。$<\&\mathrm{b[i]},0>$是循环体摘要的一个元组，对该元组而言，上述描述的三个条件$\forall[i_0\in[0,99].\mathcal{NB}(\&\mathrm{b}[i_0],i_0)$、$\forall[i_0\in[0,99].\mathcal{NB}(0,i_0)$和$\forall[i_0\in[0,99].\mathcal{NA}(*\,(\&\mathrm{b}[i_0]),i_0)$均满足，因此根据该元组，我们可以生成循环语句的一个元组，即$<\lambda\mathrm{x}.(\&\mathrm{b}[\mathrm{x}])[0,99],0>$。

2.4.3 元组转换方法

在一些情况下，由于一些条件没有满足，我们无法根据循环体的某个元组直接生成循环语句的一个元组，但是通过将该元组中包含的部分子表达式替换为其对应的等价表达式往往可以使得本没有满足的条件变成满足的条件。因此，我们提出了一种元组替换方法，通过将元组的值表达式中所包含的除循环控制变量以外的固定地址的内存访问表达式替换为当前迭代执行结束后该内存地址中存放的内容来对元组进行转换，经过转换得到的元组往往可以满足 2.4.2 中所描述的条件，从而可以生成循环语句摘要中的元组。

Table 6 Calculation rules for new value

表 6 新值的计算规则

操作符 *op*	m_0存储的值
+，−	$*\mathrm{m}\ \ \mathrm{op}\ \sum_{x\in\varsigma(\mathrm{w})}\mathrm{e}[x/\mathrm{w}]$
×，÷	$*\mathrm{m}\ \ \mathrm{op}\ \prod_{x\in\varsigma(\mathrm{w})}\mathrm{e}[x/\mathrm{w}]$
∧	$*\mathrm{m}\ \ \mathrm{op}\ \bigvee_{x\in\varsigma(\mathrm{w})}\mathrm{e}[x/\mathrm{w}]$
∨	$*\mathrm{m}\ \ \mathrm{op}\ \bigwedge_{x\in\varsigma(\mathrm{w})}\mathrm{e}[x/\mathrm{w}]$

假设$< m_0, v_0 >$和$< m_1, v_1 >$是循环体的摘要中的两个元组，其中m_0表示固定地址，v_1包含子表达式$*m_0$，那么我们将表达式v_1中的$*m_0$替换为当前迭代执行之后内存m_0中存放的内容。具体替换如下：

- 如果条件$\epsilon(v_0, \mathrm{s}) == v_0$成立，那么$v_0$没有被循环体 *s* 所修改，也就意味着每一次迭代执行结束之后m_0中存放的内容均为v_0，因此直接将$*m_0$替换为v_0即可；
- 如果v_0形如$*m_0\ \ \mathrm{op}\ \ \mathrm{e}$或者$\mathrm{e}\ \ \mathrm{op} *m_0$，其中 op 属于操作符集合{+，−，×，÷，∧，∨}，而且对于循环控制变量的已遍历值域$\varsigma(\mathrm{w})$中的任何一个值 w_0 而言，都有$\mathcal{NB}(\mathrm{e}[w_0/\mathrm{w}], w_0)$成立，那么我们将根据表 6 生成每一次迭代执行结束之后m_0中存放的内容。在表 6 中，第一列给出了操作符，第二列给出了根据该操作符识别递推关系生成的迭代执行结束后内存表达式m_0中存放的值。

  例如，在图 1 中，$<$ &sum, sum + b[i] $>$和$<$ &a[i], sum + b[i] $>$是循环体摘要中的两个元组，其中&sum是固定地址，值sum + b[i]形如$*\ \mathrm{m}\ \ \mathrm{op}\ \ \mathrm{e}$，而且对于循环控制变量的已遍历值域$\varsigma(\mathrm{i})$中的任何一个值$i_0$而言有$\mathcal{NB}(\mathrm{b}[i_0], i_0)$成立，因此根据表 6 我们可以得到在每一次迭代执行结束之后，&sum中存放的新值为$\mathrm{sum} + \sum_{x=0}^{i-1}\mathrm{b}[x]$，从而我们可以将元组$<$ &a[i], sum + b[i] $>$转换为$<$ &a[i], $\sum_{x=0}^{i}\mathrm{b}[x]$ $>$。根据元组$<$ &a[i], sum + b[i] $>$，我们无法生成循环语句的摘要中的元组，而根据转换后的元组$<$ &a[i], $\sum_{x=0}^{i}\mathrm{b}[x]$ $>$，我们可以合成元组$<\lambda x.(\&a[x])[0, \mathrm{s}-1], \sum_{y=0}^{x}\mathrm{b}[y]>$。

2.4.4 嵌套循环

我们的方法可以很好地处理嵌套循环，我们生成摘要的方法是一个递归过程，首先，我们先计算最内层循环的摘要，然后处理外层循环时，内层循环已经被抽象成摘要，那么外层循环实际上已经转换成一个单层循环，对外层循环的分析就转换为对单层循环的分析。

图 8 给出了一个矩阵乘法，包含三层循环，从内到外依次标记为 L_1，L_2 和 L_3，程序点 p_1、p_2 分别表示循环 L_1 和循环 L_2 的入口点。使用我们的方法，我们首先生成最内层循环的摘要，即 $\tau(L_1)$：

$$\left\{\begin{array}{c} < \&\mathrm{k}, \mathrm{p} >, \\ < \&\mathrm{c[i][j]}, \left(\mathrm{c[i][j]} + \prod_{x=0}^{p-1} a[i][x] * b[x][j]\right) @p_1 > \end{array}\right\}$$

有了摘要 $\tau(L_1)$，循环 L_1 就相当于被转换为一系列的赋值语句了。

接着在生成循环 L_2 的摘要时，对循环 L_1 的分析就转换为对摘要 $\tau(L_1)$的分析，L_2 的摘要 $\tau(L_2)$即为：

$$\left\{\begin{array}{c} <\&k, p>, \\ <\&j, m>, \\ <\lambda x.(\&c[i][x])[0, m-1], \left(\prod_{x=0}^{m-1}\prod_{y=0}^{p-1} a[i][x] * b[x][y]\right)@p_2> \end{array}\right\}$$

同样地，我们可以生成最外层循环 L_3 的循环摘要。

```
    i = 0;
L3: while(i<n) {
        j = 0;
      {p2: j==0}
    L2: while(j<m) {
            c[i][j] = 0;
            k = 0;
          {p1: k==0}
        L1: while(k<p) {
                c[i][j] = c[i][j] + a[i][k] * b[k][j];
                k = k + 1;
            }
            j = j + 1;
        }
        i = i + 1;
    }
```

Fig. 8 The program matrix_multiplication

图 6 矩阵乘法

3 摘要的应用

本节将详细介绍摘要的应用，包括生成语句的后置条件、前置条件以及生成循环不变式。

3.1 后置条件的生成

假设 p 表示程序语句 s 前面的程序点，那么对于语句 s 的摘要中的任何一个元组$<m, v>$，我们可以根据下面的规则生成语句 s 的后置条件。

- 如果 m 表示单一内存地址，那么后置条件为$*m == v@p$
- 如果 m 表示集合内存地址，且形如$\lambda x.m'[set]$，那么后置条件为$\forall x \in [set].*m' == v@p$

在图 1 中，循环语句前面的程序点为 4，$<\&sum, \sum_{x=0}^{s-1} b[x]>$是该循环语句的一个元组，&sum 表示单一内存地址，根据第一个规则，我们可以生成该循环的一个后置条件$sum == (\sum_{y=0}^{s-1} b[y])@4$。对于该循环语句的另外一个元组$<\lambda x.(\&a[x])[0, s-1], \sum_{y=0}^{x} b[y]>$而言，内存表达式$\lambda x.(\&a[x])[0, s-1]$表示集合地址，根据第二条规则，我们可以生成后置条件$\forall x \in [0, s-1].a[x] == (\sum_{y=0}^{x} b[y])@4$。

3.2 前置条件的生成

在 2.2.1 节中等价表达式的定义中，我们已经提到，当位于语句 s 之后的程序点上的表达式e_1是一个谓词时，该谓词实际上就是 s 的一个后置条件，经过 2.2.2 节等价表达式的计算得到位于 s 之前的程序点上的等价表达式e_2实际上就是该后置条件e_1关于 s 的前置条件，当前置条件e_2成立时，后置条件e_1肯定成立。

3.3 循环不变式的生成

假设$<m, v>$是循环体 s 的摘要中的一个元组，我们通过对该元组进行分析生成循环不变式：

1) 内存表达式 m 表示固定地址
   - 如果条件$\epsilon(v, s) == v$成立，那么可以生成循环不变式$*m == v$；

- 如果 m 是单一内存表达式，v 形如$*\mathrm{m}\ \ \mathrm{op}\ \ \mathrm{e}$或者$\mathrm{e}\ \ \mathrm{op}*\mathrm{m}$，其中 op 属于操作符集合{+，−，×，÷，∧，∨}，那么可以通过寻找递推关系生成循环不变式，如表 7 所示。在表 7 中，第一列给出了操作符，第二列给出了根据该操作符识别递推关系所能生成的循环不变式的候选，其中 p 表示循环语句的入口点，w 表示循环控制变量。

  例如，在图 1 中，$<\&\mathrm{sum}, \mathrm{sum}+\mathrm{b[i]}>$是循环体摘要中的一个元组，&sum是固定地址，值$\mathrm{sum}+\mathrm{b[i]}$形如$*\mathrm{m}\ \ \mathrm{op}\ \ \mathrm{e}$，循环控制变量 i 的已遍历值域$\varsigma(\mathrm{i})$为$[0,\mathrm{i}-1]$，因此，根据表 7，我们可以得到循环不变式$\mathrm{sum}==\mathrm{sum@4}+\left(\sum_{x=0}^{i-1}\mathrm{b}[x]\right)@4$。经过数据流分析我们知道在程序点 4 上有$\mathrm{sum}==0$成立，因此该循环不变式可以被简化为$\mathrm{sum}==\left(\sum_{x=0}^{i-1}\mathrm{b}[x]\right)@4$。

Table 7 Generation rules for loop invariants

表 7 循环不变式的生成规则

操作符 *op*	循环不变式
+，−	$*\mathrm{m}==*\mathrm{m@p}\ \ \mathrm{op}\ \ \sum_{x\in\varsigma(\mathrm{w})}\mathrm{e}[x/\mathrm{w}]$
×，÷	$*\mathrm{m}==*\mathrm{m@p}\ \ \mathrm{op}\ \ \prod_{x\in\varsigma(\mathrm{w})}\mathrm{e}[x/w]$
∧	$*\mathrm{m}==*\mathrm{m@p}\ \ \mathrm{op}\ \ \bigvee_{x\in\varsigma(\mathrm{w})}\mathrm{e}[x/\mathrm{w}]$
∨	$*\mathrm{m}==*\mathrm{m@p}\ \ \mathrm{op}\ \ \bigwedge_{x\in\varsigma(\mathrm{w})}\mathrm{e}[x/\mathrm{w}]$

2) 内存表达式 m 表示移位地址

当 m 表示移位地址时，这意味着循环的每一次迭代都会根据循环控制变量的不同取值而修改不同的内存地址，我们可以通过对内存地址 m 中的循环控制变量进行局部量化得到循环不变式，即$\forall \mathrm{x}\in\varsigma(w).(*\mathrm{m}[x/\mathrm{w}]==\mathrm{v}[x/\mathrm{w}])$。

例如，在图 1 中，$<\&\mathrm{b[i]},0>$是循环体摘要中的一个元组，&b[i]是移位地址，循环控制变量 i 的已遍历值域$\varsigma(i)$为$[0,\mathrm{i}-1]$，因此我们可以生成循环不变式$\forall \mathrm{x}\in[0,\mathrm{i}-1].(\mathrm{b[x]}==0)$。再如，根据经过 2.4.3 节的元组转换后得到的元组$<\&\mathrm{a[i]},\sum_{x=0}^{i}\mathrm{b}[x]>$，我们可以生成循环不变式$\forall \mathrm{x}\in[0,\mathrm{i}-1].(\mathrm{a[x]}==\sum_{y=0}^{x}\mathrm{b}[y])$。

4 实现与案例研究

本文提出的自动生成包括循环语句在内的程序语句的摘要的方法使用 Java 语言实现，已经集成到程序验证工具 Accumulator 中，能够在多个平台上直接运行。Accumulator 中集成了一些自动化和半自动化的技术，比如别名分析、数据流分析、计算前置条件等。

Table 8 Experimental programs

表 8 实验程序

数据结构	程序
封闭整数区间	count, sum, multiply
一维数组	copy, assign, count, sum, multiply, search, maximum, minimum
二维数组	copy, assign, count, sum, multiply, search, maximum, minimum, matrix_multiplication
非循环单链表	assign, count, sum, multiply, search, maximum, minimum

我们应用本文提出的方法为一系列循环程序自动生成了摘要，同时根据生成的摘要自动生成了循环语句的规约，包括前置条件、后置条件以及循环不变式，提高程序验证的自动化程度和效率，为验证人员减轻负担。表 8 给出了一部分实验程序，第一列“数据结构”给出了循环遍历的数据结构的类型，第二列“程序”给出了在数据结构上进行的操作，第二列中的一个程序名称指的是一系列相似操作的程序，而不仅仅是一个程序，比如，“count”程序既可以是找出满足某个给定条件的元素的个数，条件可以是大于某个数值或者非空等，也可以是简单地找出被遍历到的所有元素的个数。另外，操作二维数组的程序都包含嵌套循环，我们的方法可以为其生成摘要以及规约。为这些程序生成自动生成摘要和规约的时间都不超过 5 秒，在效率上我们认为是可以接受的。

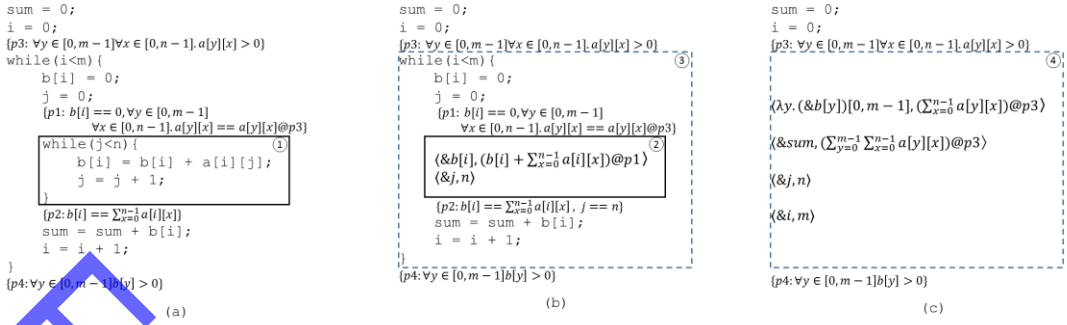

Fig.9 Experimental programs

图 7 实验程序

图 9(a)所示的程序操作一维数组 b 和二维数组 a，它将数组 a 的第 k 行所有元素的和赋值给数组 b 的第 k 个元素，并将数组 a 所有元素的和赋值给变量 sum。程序点 p_1 和 p_2 分别是内层循环的入口点和出口点，程序点 p_3 和 p_4 分别是外层循环的入口点和出口点。

首先，我们分析内层循环①的循环体，该循环体是由两个赋值语句组成的顺序语句构成，其摘要如下：

$$\left\{\begin{array}{c} < \&b[i], b[i] + a[i][j] >, \\ < \&j, j + 1 > \end{array}\right\}$$

对于内层循环①而言，变量j是循环控制变量，因此内存地址&b[i]是固定地址。相应的值表达式b[i] + a[i][j]形如$* m + e$，而且对于循环控制变量的已遍历集合 ξ(j)中的任意一个值 j_0 而言，有$\mathcal{NB}(a[i][j_0], j_0)$成立。因此，根据表 5，我们可以得到内层循环①执行结束之后，&b[i]中存放的值为$b[i] + \sum_{x=0}^{n-1} a[i][x]$。内存地址&j也是固定地址，我们以同样的方式进行处理。最后，我们可以得到内层循环①的摘要，即

$$\left\{\begin{array}{c} < \&b[i], b[i] + \sum_{x=0}^{n-1} a[i][x] >, \\ < \&j, n > \end{array}\right\}$$

基于该摘要，我们可以生成内层循环①的后置条件。&b[i]是固定地址，因此，我们可以生成程序点 p_2 上的后置条件$b[i] == (b[i] + \sum_{x=0}^{n-1} a[i][x])@p1$。经过数据流分析，我们知道$b[i] == 0$和$\forall y \in [0, m-1] \forall x \in [0, n-1]. a[y][x] == a[y][x]@p3$在程序点 p_1 上成立，且内层循环①没有修改数组 a，因此，后置条件简化为$b[i] == (\sum_{x=0}^{n-1} a[i][x])@p3$。同样地，我们可以生成后置条件$j == n$。其中，被简化后的后置条件$b[i] == (\sum_{x=0}^{n-1} a[i][x])@p3$是我们验证该程序的正确性必须要证明的性质之一。

根据内层循环①的摘要，我们可以将图 9(a)中内层循环①替换为其摘要②，从而将外层循环转换为单层循环③，如图 9(b)所示。通过分析循环③的循环体，我们可以生成摘要：

$$\left\{\begin{array}{c} < \&b[i], \sum_{x=0}^{n-1} a[i][x] >, \\ < \&sum, sum + \sum_{x=0}^{n-1} a[i][x] >, \\ < \&j, n >, \\ < \&i, i + 1 > \end{array}\right\}$$

对于外层循环③而言，i是循环控制变量，因此，第一个元组中的&b[i]是移位地址，而且条件$\forall i_0 \in [0, m-1]. \mathcal{NB}(b[i_0], i_0)$成立。因此，对于任意的值$i_0$而言，&b[$i_0$]在 p_3 上的值和&b[i_0]在循环控制变量 i 的值等于i_0之前的迭代中的取值是相同的。根据该元组，我们知道外层循环③修改了表达式$\lambda y. (\&b[y])[0, m-1]$所表示的地址。对于任意的值$i_0$，条件$\forall i_0 \in [0, m-1]. \mathcal{NB}(\sum_{x=0}^{n-1} a[i_0][x], i_0)$和$\forall i_0 \in [0, m-1]. \mathcal{NA}(* (\&b[i_0]), i_0)$均成立，因此，$\sum_{x=0}^{n-1} a[i_0][x]$在外层循环的入口点上的值和$* (\&b[i_0])$在外层循环的出口点上的值相同。根据该元组，我们可以为外层循环③的摘要生成元组$< \lambda y. (\&b[y])[0, m-1], \sum_{x=0}^{n-1} a[y][x] >$。

在第三个元组中，内存地址&j是一个固定地址，而且$\epsilon(n, body) == n$成立，其中 *body* 表示外层循环的循环体。因此，我们知道外层循环修改了内存地址&j，而且循环执行结束之后，内存&j中存放的内容为 n，即$< \&j, n >$是外层循环摘要中的一个元组。

我们可以用同样的方式来处理第二个元组和第四个元组。最后，我们得到外层循环的摘要：

$$\left\{\begin{array}{c} <\lambda y.(\&b[y])[0,m-1],\ (\sum_{x=0}^{n-1} a[y][x])@p3>, \\ <\&sum,(\sum_{y=0}^{m-1}\sum_{x=0}^{n-1} a[y][x])@p3>, \\ <\&j,n>, \\ <\&i,m> \end{array}\right\}$$

基于该摘要，我们将外层循环③转换为图 9(c)中顺序语句④。通过这个过程，我们可以看出我们可以将循环语句替换为其摘要，将一个有循环的程序转换为没有循环的顺序语句。

基于外层循环的摘要，我们可以生成后置条件$\forall y \in [0,m-1].b[y] == \sum_{x=0}^{n-1} a[y][x]\ @p3$、sum == $(\sum_{y=0}^{m-1}\sum_{x=0}^{n-1} a[y][x])@p3$、j == n以及i == m。前两个后置条件是我们在证明图 9(a)中的程序时必须要证明的两个性质。

对于外层循环的后置条件$\forall y \in [0,m-1].b[y] > 0$，根据其摘要，我们生成它在程序点 p_3 上的前置条件$\forall y \in [0,m-1].\sum_{x=0}^{n-1} a[y][x] > 0$。在程序点 p_3 上，有断言$\forall y \in [0,m-1]\forall x \in [0,n-1].a[y][x] > 0$成立，通过定理证明器 Z3，我们可以证明该前置条件的正确性，从而证明了后置条件的正确性。同样地，我们可以证明后置条件sum == $(\sum_{y=0}^{m-1}\sum_{x=0}^{n-1} a[y][x])@p3$。

从上述过程，我们可以看出本文提出的自动生成循环摘要以及生成规约的方法可以有效地辅助程序的证明，提高程序验证的自动化程度和效率，减轻验证人员的负担。

5 相关工作

我们的工作主要和自动摘要技术以及自动生成规约技术密切相关。

5.1 自动摘要技术

近年来，很多学者致力于自动生成循环语句的摘要。[7]将循环语句抽象为循环语句的前置条件和后置条件之间的关系，称为转换不变式。该方法依赖于一组抽象域的模板，利用这些模板生成转换不变式的候选集，然后利用 SMT 求解器(SMT Solver)证明转换不变式的正确性，最后借助于转换不变式分析程序的终止性。该方法的操作对象是程序的控制流图。[8]借助于动态符号执行技术将依赖输入循环(Input-dependent Loop)抽象为一组符号化的执行路径，目标在于解决动态测试用例生成中循环语句引起的路径爆炸问题。该方法首先利用一组模式匹配的规则猜测循环迭代的次数以及自动生成与归纳变量(Induction Variable)相关的循环不变式，然后将循环抽象为前置条件和后置条件。前置条件约束了循环中变量的输入值，后置条件反映了循环的执行产生的影响，后置条件中出现的程序变量是和输入值有关的符号化的值。[17]将循环摘要抽象为线性或者近似线性的函数关系，这种函数关系描述了循环次数与输入之间的依赖关系，该方法要求循环摘要以约束求解器所支持的变量类型和表达式来表示。该工作生成循环摘要的目标在于解决动态测试数据生成中循环语句引起的路径爆炸问题以及路径约束求解困难的问题。[9,18]将循环和函数表示为符号化的抽象转换(Symbolic Abstract Transformer)[19]，即两个抽象状态之间的关系，生成的符号化的抽象转换符合相应的程序片段的语义。该工作可以生成诊断信息，可以帮助发现 UNIX 程序中的缓冲区溢出错误。不同于上述方法，本文将语句抽象为语句修改的内存地址以及语句执行结束以后内存中存放的新值，内存地址和新值都用符号化的。不同于传统的基于抽象解释的方法，本文在生成语句的摘要过程中所需要的时间是线性的，不用经过多次迭代计算不动点，从而也没有精度损失问题。此外，不同于以往的工作，本文所生成的摘要可以用于生成语句的规约，包括循环不变式等，规约对于程序的分析验证具有举足轻重的作用。

5.2 自动生成规约

5.2.1 前置条件

不同的工作生成前置条件的目标有所不同。[20,21]生成前置条件来验证程序的中执行。[22,23]旨在为循环语句中安全性相关的断言生成前置条件。[24]为了编译器的优化而计算循环语句的最弱前置条件。[25-

29]为了验证程序的正确性而生成前置条件，这和本文的目标是一致的。[25,26]借助于循环不变式来简化循环。[28]通过识别 while 循环中的不变式关系(Invariant Relation)[30]来计算后置条件关于循环语句的前置条件。[29]通过删除控制流图中的所有回边来消除循环并添加循环不变式来控制循环体。尽管很多工作致力于研究循环不变式，但是自动生成循环不变式本身依旧是一个非常具有挑战性的工作。不同于所有这些方法，本文不需要借助于循环不变式就可以生成循环语句的前置条件。[27]首先通过对循环迭代的次数进行限制来将循环转换为非循环程序，这种方式会带来精度损失，而本文的方法是精确的。[31]根据循环语句的后置条件和循环体中的中间断言来生成循环语句的前置条件，生成的依据是给定的一些规则，而本文依据的是自动生成的循环摘要，减少了人工参与的成本。

5.2.2 循环不变式

常用的生成循环不变式的方法包括基于抽象解释的方法、基于谓词抽象的方法、基于约束求解的方法以及基于断言的方法。基于抽象解释技术，[32-36]可以有效地生成线性不变式和多项式不变式。不同于基于抽象解释的方法，本文不仅可以生成线性不变式和多项式不变式，还可以为操作非循环单链表等自定义的数据结构生成量化的循环不变式。[37]根据给定的谓词构造布尔组合形式的循环不变式，该方法已经实现并作为[38]的一部分。基于约束求解，[39-43]首先生成指定目标不变式的模板形式，其中变量的系数未知，然后使用不同的技术求解模板中的参数从而得到不变式。[44]依赖于循环语句的中间断言来生成循环不变式。[45-46]根据规则将后置条件进行弱化生成循环不变式。不同于基于谓词抽象、约束求解和断言的方法，本文的方法不需要用户提供任何信息，不会给验证人员任何负担，同时降低了出错概率。[47]通过记录每次循环条件变化时程序变量的值获取程序变量之间的函数依赖关系生成形如$y = f(x1, x2, \ldots, xn)$的循环不变式，本文生成的循环不变式种类更多。[48-49]介绍了形状图和形状图逻辑的概念，并提出了基于形状图逻辑来自动生成循环不变形状图的推断方法。该方法要求源程序增加形状声明来处理指针程序。[50]提出了一种基于分离逻辑推断链表程序的循环不变式的方法，它利用符号执行机制和一套重写规则，迭代计算不动点。不同于[48-50]，本文提出的方法基于 Scope 逻辑，Scope 逻辑是对 Hoare 逻辑的扩展，可以有效地处理指针程序，但不要求在源程序中增加任何内容。

6 结论

自动分析并证明循环语句是程序验证领域中的一个非常重要而且具有挑战性的课题。本文提出将语句的摘要用该语句修改的内存地址以及该语句执行结束以后存储在该内存中的新值来定义，同时提出了一种自动生成循环语句的摘要并基于摘要自动生成循环语句的规约的方法，可以有效地将包含循环语句的程序转换为没有循环语句的程序，并自动生成并证明循环语句的相关性质。

我们实现了本文提出的方法，并集成到验证工具 Accumulator 中，可以在多个平台上使用。我们同时使用了很多实际软件中常用的循环验证了本文的方法，实验数据表明，本文提出的方法可以有效地为操作常用数据结构的循环自动生成摘要以及包括前置条件、后置条件以及循环不变式在内的规约，提高程序验证的自动化程度和效率，减轻验证人员的负担。

本文提出的方法可适用性较广，我们的方法还可以为操作双链表结构以及多维数组的循环程序生成摘要。此外，本文生成循环不变式的思想具备较强的可扩展性，首先，该思想可以用于包含 break、continue 等语句的循环中，只需要将不变式中遍历的集合进行修改即可；其次，该思想可以用于 foreach 形式的循环，比如 Java 语言中的形如“for (循环变量类型 循环变量名称：要被遍历的对象) ”形式的循环，虽然循环的类型不同，但都是对某个对象的依次遍历；再次，该思想可以用于步长非 1 的数组程序中，只需要将和循环控制变量相关的有效值域以及已遍历值域进行重新计算并将访问数组元素的下标根据具体的步长进行修改即可，而这些都是通过数据流分析自动获取的。

References:

[1] Floyd R W. Assigning Meaning to Programs. 1967(19)

[2] Hoare C.A.R. An Axiomatic Basis for Computer Programming. 1969(10)

[3] Dijkstra E W. A Discipline of Programming. 1976

[4] Gries D. Loop Invariant[M] //Ralston A, Reilly E D, eds. Encyclopedia of Computer Science. Third Edition. New York: Van Nostrand Reinhold, 1993:796-797.

[5] Carlo Alberto Furia and Bertrand Meyer. Inferring loop invariants using postconditions. In Fields of logic and computation, pages 277–300. Springer, 2010.

[6] Ernst M D, Perkins J H, Guo P J, et al. The Daikon System for Dynamic Detection of Likely Invariants[J]. Science of Computer Programming, 2007, 69(1-3): 35-45.

[7] Tsitovich, Aliaksei, et al. "Loop Summarization and Termination Analysis."TOOLS and Algorithms for the Construction and Analysis of Systems, International Conference, Tacas 2011, 81-95.

[8] Papaioannou G, Keller J B. Automatic partial loop summarization in dynamic test generation[C]// International Symposium on Software Testing and Analysis, ISSTA 2011, Toronto, On, Canada, July. 2011:23-33.

[9] Kroening, Daniel, et al. Loop Summarization Using Abstract Transformers. Automated Technology for Verification and Analysis, International Symposium, ATVA 2008, Seoul, Korea, October 20-23, 2008. Proceedings 2008:111-125.

[10] Chawdhary, A., Cook, B., Gulwani, S., Sagiv, M., Yang, H.: Ranking abstractions. In: Programming Languages and Systems. Volume 4960 of LNCS. Springer (2008), 148-162

[11] https://memcached.org/

[12] https://httpd.apache.org/

[13] https://nginx.org/en/

[14] Jianhua, Z., Xuandong, L.: Scope Logic: An extension to Hoare Logic for pointers and recursive data structures. In: Theoretical Aspects of Computing–ICTAC 2013. Springer (2013) 409–426

[15] http://seg.nju.edu.cn/toolweb/index.html

[16] De Moura, L., Bjorner, N.: Z3: An efficient SMT solver. In: Tools and Algorithms for the Construction and Analysis of Systems. Springer (2008)

[17] 聂楚江, 刘海峰, 苏璞睿,等. 一种面向程序动态分析的循环摘要生成方法[J]. 电子学报, 2014, 42(6):1110-1117.

[18] Kroening D, Sharygina N, Tonetta S, et al. Loop summarization using state and transition invariants[J]. Formal Methods in System Design, 2013, 42(3):221-261.

[19] Reps, T.W., Sagiv, S., Yorsh, G.: Symbolic Implementation of the Best Transformer. In: Steffen, B., Levi, G. (eds.) VMCAI 2004. LNCS, vol. 2937, pp. 252–266. Springer, Heidelberg (2004).

[20] M. Bozga, R. Iosif, and F. Kone˘cn`y. Deciding conditional termination. In Tools and Algorithms for the Construction and Analysis of Systems, pages 252–266. Springer, 2012.

[21] B. Cook, S. Gulwani, T. Lev-Ami, A. Rybalchenko, and M. Sagiv. Proving conditional termination. In Computer Aided Verification, pages 328–340. Springer, 2008.

[22] Y. Moy. Sufficient preconditions for modular assertion checking. In Verification, Model Checking, and Abstract Interpretation, pages 188–202. Springer, 2008.

[23] M. N. Seghir and D. Kroening. Counterexample-guided precondition inference. In Programming Languages and Systems, pages 451–471. Springer, 2013.

[24] N. P. Lopes and J. Monteiro. Weakest precondition synthesis for compiler optimizations. In Verification, Model Checking, and Abstract Interpretation, pages 203–221. Springer, 2014.

[25] C. Flanagan and J. B. Saxe. Avoiding exponential explosion: Generating compact verification conditions. In SYMPOSIUM ON PRINCIPLES OF PROGRAMMING LANGUAGES, pages 193–205. ACM, 2001.

[26] K. R. M. Leino. Efficient weakest preconditions. Information Processing Letters, 93(6):281–288, 2005.

[27] I. Jager and D. Brumley. Efficient directionless weakest preconditions. Technical report, Technical Report CMU-CyLab-10-002, Carnegie Mellon University, CyLab, 2010.

[28] O. Mraihi, W. Ghardallou, A. Louhichi, L. L. Jilani, K. Bsaies, and A. Mili. Computing preconditions and postconditions of while loops. In Theoretical Aspects of Computing–ICTAC 2011, pages 173–193. Springer, 2011.

[29] M. Barnett and K. R. M. Leino. Weakest-precondition of unstructured programs. In ACM SIGSOFT Software Engineering Notes, volume 31, pages 82–87. ACM, 2005.

[30] A. Louhichi, O. Mraihi, L. L. Jilani, and A. Mili. Invariant assertions, invariant relations, and invariant functions. In WING 2009 WORKSHOP ON INVARIANT GENERATION, page 60. Citeseer, 2009.

[31] J. Zhai, H. Wang, and J. Zhao. Assertion-directed precondition synthesis for loops over data structures. In Dependable Software Engineering: Theories, Tools, and Applications - First International Symposium, SETTA 2015, Proceedings, pages 258–274, 2015.

[32] Denis Gopan, Thomas Reps, and Mooly Sagiv. A framework for numeric analysis of array operations. ACM SIGPLAN Notices, 40(1):338–350, 2005.

[33] Antoine Mine. The octagon abstract domain. Higher-Order and Symbolic Computation, 19(1):31–100, 2006.

[34] Patrick Cousot and Nicolas Halbwachs. Automatic discovery of linear restraints among variables of a program. In Proceedings of the 5th ACM SIGACTSIGPLAN symposium on Principles of programming languages, pages 84–96. ACM, 1978.

[35] Enric Rodrıguez-Carbonell and Deepak Kapur. An abstract interpretation approach for automatic generation of polynomial invariants. In Static Analysis, pages 280–295. Springer, 2004.

[36] 邢建英, 李梦君, 李舟军. CILinear:一个线性不变式自动构造工具[J]. 计算机科学, 2010, 37(12):91-95.

[37] Cormac Flanagan and Shaz Qadeer. Predicate abstraction for software verification. ACM SIGPLAN Notices, 37(1):191–202, 2002.
[38] Cormac Flanagan, K Rustan M Leino, Mark Lillibridge, Greg Nelson, James B Saxe, and Raymie Stata. Extended static checking for java. In ACM Sigplan Notices, volume 37, pages 234–245. ACM, 2002.
[39] Michael A Colon, Sriram Sankaranarayanan, and Henny B Sipma. Linear invariant generation using non-linear constraint solving. In Computer Aided Verification, pages 420–432. Springer, 2003.
[40] Dirk Beyer, Thomas A Henzinger, Rupak Majumdar, and Andrey Rybalchenko. Invariant synthesis for combined theories. In Verification, Model Checking, and Abstract Interpretation, pages 378–394. Springer, 2007.
[41] 毕思勤, 曾振柄, 郭远华. 非线性循环不变式的自动生成[J]. 计算机应用, 2008, 28(7):1854-1857.
[42] 余伟, 冯勇. 用Dixon结式产生非线性循环不变式[J]. 四川大学学报:工程科学版, 2012, 44(4):115-121.
[43] 刘自恒, 曾庆凯. 一种自适应的循环不变式生成方法[J]. 计算机工程, 2013(6):76-81.
[44] Mikolas Janota. Assertion-based loop invariant generation. RISC-Linz, page 15, 2007.
[45] Carlo Alberto Furia and Bertrand Meyer. Inferring loop invariants using postconditions. In Fields of logic and computation, pages 277–300. Springer, 2010.
[46] Zhai, J., Wang, H., Zhao, J.: Post-condition-directed invariant inference for loops over data structures. In: Software Security and Reliability-Companion (SERE-C), 2014 IEEE Eighth International Conference on, IEEE (2014)
[47] 马竹根, 王灿明. 利用基因表达式编程自动生成循环不变式[J]. 计算机与数字工程, 2009, 37(7):7-10.
[48] 刘刚, 陈意云, 张志天. 循环不变形状图的自动推断[J]. 电子技术, 2011, 38(8):4-6.
[49] Liang, H., Zhang, Y., Chen, Y., & Li, Z. A Shape System and Loop Invariant Inference.
[50] Magill, Stephen, et al. "Inferring invariants in separation logic for imperative list-processing programs." SPACE 1.1 (2006): 5-7.