清 华 大 学

综 合 论 文 训 练

题目：基于数据流分析的字符串缺陷检测算法设计与实现

系　　别：软件学院

专　　业：计算机软件

姓　　名：王程鹏

指导教师：孙家广　教 授

2016 年 6 月 8 日

关于学位论文使用授权的说明

本人完全了解清华大学有关保留、使用学位论文的规定，即：学校有权保留学位论文的复印件，允许该论文被查阅和借阅；学校可以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该论文。

（涉密的学位论文在解密后应遵守此规定）

签　名：_____导师签名：_____日　期：____

中文摘要

C 语言中，字符串缺陷是指字符串本身因具备某种不当的特性导致程序运行错误或者意外终止，它会给含有字符串操作的大型项目造成一定隐患。在 C 语言中，用于表示字符串的 char 型数组若缺少字符'\0'，在调用字符串库函数时将会引起程序意外终止或者得到一个错误的结果。由于这一错误在字符串有关的项目中极为常见，且使用人工方法不易检测，需要设计一个自动检测的方法。

本文针对这一常见的字符串缺陷的检测进行研究，在传统的数据流分析的方法基础上，基于 CPAchecker 框架实现了缺陷检测算法。本文具有以下几个方面的贡献：

一、提出了一个分析域，针对这一特定的字符串缺陷进行分析，能够完整地分析常用的程序结构。

二、实现了跨函数的分析算法，设计了较为精确的过程间分析机制。

三、在多种测试用例和实际项目中应用实践，检测出项目中存在的字符串缺陷。

本文提出的基于数据流分析的字符串缺陷检测算法能够满足实用项目中的字符串缺陷的检测需要。在评估部分中，设计了充分的测试样例。结果显示算法具有很好的检测效果。

关键词：数据流分析；程序分析；字符串缺陷检测

ABSTRACT

In C program, string fault is the program errors and terminations caused by string which has a wrong characteristics. An array of char representing a string without the typical char '\0' may cause the program termination or a wrong error when passed into a string lib function. This fault is common in the projects using string lib functions, and manual work usually cannot detect it. Hence this fault needs an automatic detecting method.

This paper proposes a detecting algorithm for the fault, which is based on the traditional framework of data flow analysis, and implemented on the platform CPAchecker. The main contributions of this paper are following:

I. Propose a field of analysis, analyze aiming at this string fault, and can handle common program structures.

II. Implement an interprocedural analysis algorithm, and design a meticulous mechanism of interprocedural analysis.

III. Practice on testing cases and the practical project, and detect the string fault in the projects.

The string fault detecting algorithm proposed in this paper can meet the need to detect the string fault in practical projects. In the evaluation part, sufficient testing cases are designed, and the results show that the algorithm has a satisfactory performance.

Key words: Data flow analysis; program analysis; string fault detection

目 录

主要符号表

符号	符号含义
D	抽象域
ε	变迁关系
sf(x)	节点名称 x 在 StringState 的 StructField 中的 structFields 映射下的像
ca(x)	节点名称 x 在 StringState 的 StructField 中的 isCharArray 映射下的像
ti(x)	节点名称 x 在 StringState 的 StructField 中的 terminateIndex 映射下的像
l(x)	节点名称 x 在 StringState 的 StructField 中的 length 映射下的像
type(x)	路径名称为 x 的节点对应的类型
gc(x)	判断路径为 x 的节点对应的类型是否为 char 型数组或变量
gtiv(x)	路径名称为 x 的节点对应的’\0’位置下标组成的向量
gl(x)	路径名称为 x 的节点对应的数组长度
pm(x)	名称为 x 的指针在 PointerMap 下指向内存变量的名称
sp(x,k)	对表达式 x 中的第 k 个元素反复运用 pm 运算，得到的新的表达式
s(x)	对表达式 x 反复运用 sp 运算，得到的新的表达式
E	CPA 中抽象状态的集合

[prop]	boolean 类型的表达式，当命题 prop 成立时取值为 true，否则取值为 false
fix(f, x)	单变量函数 f 在变量取值为 x 时反复调用 f 得到的不动点的取值
length(a)	序列 a 的长度
Op	CPAchecker 中程序操作的等价类集合
D	声明操作集合（Declaration）
S	语句操作集合（Statement）
RS	返回语句操作集合（Return Statement）
FR	函数返回操作集合（Function Return）
A	假设操作集合（Assume）
FC	函数摘要操作集合（Function Call）
B	空白语句边
CTR	调用返回操作集合（Call To Return）
ite(P, a, b)	若 P 为真，取值为 a，否则取值为 b
gsf(a1. … .an)	根据 a1. … .an 构造 StructField 中的四个映射

第 1 章　引言

1.1　研究背景

程序分析技术在 20 世纪 70 年代起步，发展至今已有近半个世纪的历程。其中数据流分析是一种重要的程序分析方式。与此同时，模型检测也成为检测程序正确性的一种重要手段。两种程序缺陷分析方法都存在彼此的优缺点。其中传统的数据流分析方式效率较高，但是存在较为严重的误报的情况。模型检测方法关注的是如何减少结果中的误报，但是检测的效率较低。一直以来，程序缺陷分析中的正确性与效率如何做出权衡是程序分析领域关注的问题。如何在保证一定效率的前提下提高结果的正确性是众多研究者研究的重要课题之一。

1.2　研究现状

2007 年，加拿大计算机科学家 Dirk Beyer 提出了一个处理程序缺陷检测的新方法[2]：通过可配置的程序分析器（Configurable Program Analysis）将传统的数据流分析与模型检测结合起来，从而在缺陷检测的正确性和效率上做出权衡。2008 年，Dirk Beyer 在 2008 年的论文的基础上提出了 precision adjustment 的概念[5]，使得程序分析在有限的时间开销内结果更精细。Dirk Beyer 在论文的基础上实现了 CPAchecker 开源框架，为程序分析与模型检测提供了一个融合的平台。

下面将从传统的数据流分析的框架与 CPAchecker 开源框架两个方面做出简要介绍。

1.2.1　基本数据流分析的基本框架

数据流分析的组成部分如下所示：

（1）控制流图（V, E）

（2）有限高度的半格 $(S,\acute{o})$

（3）对入口 entry 的赋值 I

（4）一个变迁函数的集合，$\forall v \in V - entry$，存在一个节点变迁函数 f_v

算法一　数据流分析[1]

$DATA_{entry} = I$

$\forall v \in (V - entry) : DATA_v \leftarrow \bullet_V$

$ToVisit \leftarrow V - entry$

$While(ToVisit.size > 0)\{$

$v \leftarrow any\ node\ in\ ToVisit$

$ToVisit-- = v$

$MEET_v \leftarrow \acute{o}_{w \in pred(v)}\ DATA_w$

If ($DATA_v \neq f_v(MEET_v)$) $ToVisit \cup = succ(v)$

$DATA_v \leftarrow f_v(MEET_v)$.

}

1.2.2 可配置的数据流分析框架

可配置的程序分析器（Configurable Program Analysis，CPA）$\mathrm{D} = (D, \hat{I}, \mathbf{merge}, \mathbf{stop})$，其中各个组件的含义如下所示：

抽象域$D = (C, \varepsilon, ⊪)$由程序的具体状态集合 C、半格ε、具体化函数⊪组成，它表示程序分析中抽象状态的取值空间。

变迁关系$\hat{I} \subset E \times G \times E$将每个抽象状态 e 映射到一个可能的后继抽象状态e′。

merge 操作符：$E \times E \to E$。两个抽象状态的信息将由 merge 操作符进行组合得到一个合并状态。merge 操作符有两个类型：sep 类型和 join 类型，定义如下：

$$\mathbf{merge}^{sep}(e, e') = e' \quad \text{and} \quad \mathbf{merge}^{join}(e, e') = e \grave{o}\ e'$$

其中⊔为抽象域中半格ε中的交汇运算。

stop 操作符：$E \times 2^E \to \mathrm{B}$。stop 操作符用来检查一个抽象状态是否被一个抽象状态集合覆盖。stop 操作符满足条件：

$$\mathrm{stop}(\mathrm{e}, \mathrm{R}) = true => \left|\left|e\right|\right| \subseteq \cup_{e' \in R}\ ||e'||$$

与 merge 操作符相同，stop 操作符也有两种类型：sep 类型和 join 类型，定义如下：

$$\mathbf{stop}^{sep} = (\exists e' \in R : e \text{趁} e')\ \ and\ \ \mathbf{stop}^{join}(e, R) = (e\ \ ?_{e' \in R}\ e').$$

其中⊑为抽象域中半格ε中的偏序关系。

下面介绍 CPAchecker 框架的核心算法：CPA 可达性算法。

算法二 CPA(D, e_0**)[2]**

输入： $\mathrm{D} = (D, \hat{I}, \mathbf{merge}, \mathbf{stop})$，初始抽象状态 $e_0 \in E$, E 为抽象状态集合

输出: 可达的程序位置

变量: 可达的程序状态集合 **reached**, 等待处理的程序状态集合 **waitlist**

$\textbf{waitlist} = \{e_0\}$

$\textbf{reached} = \{e_0\}$

while$\{\textbf{waitlist} \neq \varnothing\}$**do**

pop e from **waitlist**
for each e' with $e\hat{I}\ e'$ **do**
for each $e'' \in \textbf{reached}$ **do**
//Combine with exisiting abstract state.
$e_{new} = \textbf{merge}(e', e'')$
If $e_{new} \neq e''$ **then**
$\textbf{waitlist} = (\textbf{waitlist} \cup \{e_{new}\})\ ,\ \{e''\}$
$\textbf{reached} = (\textbf{reached} \cup \{e_{new}\})\ ,\ \{e''\}$
if $\textbf{stop}(e', \textbf{reached}) == \textbf{false}$ **then**
$\textbf{waitlist} = \textbf{waitlist} \cup \{e'\}$
$\textbf{reached} = \textbf{reached} \cup \{e'\}$

return reached

通常情况下，对于一个具体的程序分析问题，一个可配置的程序分析器（CPA）是远远不够的，需要将多个程序分析器（CPA）组合起来，构造出一个组合程序分析器来进行分析。组合程序分析器中的状态为组合状态，变迁方式为各个组件状态在各自的 CPA 中做变迁然后进行组合。需要说明的是在组合程序分析器中还有两个重要的操作符↓ 和 $\prec$。[2]

strengthening 操作符$\downarrow: E_1 \times E_2 \to E_1$根据第二个状态计算出一个比第一个状态更加精细的状态，即：

$$\downarrow (e, e') \sqsubseteq e$$

compare 操作符$\prec \subseteq E_1 \times E_2$用于比较两个 CPA 中状态的大小。

1.3 本文主要工作

本文致力于检测 C 语言的一个常见的字符串缺陷：在 C 语言中字符串是用 char 型数组表示的，合法的字符串中应该包含字符'\0'，不含'\0'的非法的字符串在调用 C 语言字符串库函数时会导致程序崩溃或者得到一个错误的结果。这是 C 语言中关于字符串操作的一个常见的错误，出现的情形十分普遍，且目前的编译器在编译阶段不会检测到此类错误。因此对这个问题展开研究具有重要的意义。

本文的研究是基于 CPAchecker 框架，自定义一个可配置的程序分析器（CPA），与开源框架中已有的基础的 CPA 进行组合，达到分析这个字符串缺陷的目的。在算法设计与实现中，需要处理结构体域的嵌套分析、指针分析、过程间分析以及字符串库函数的语义建模等等。基于设计的算法，可以在调用 C 语言字符串库函数时分析字符串参数，若存在缺陷，则予以报错处理。

1.4 本文组织结构

第一章中介绍本文的研究背景以及研究现状，介绍近几年流行的程序分析框架——CPAchecker。接着阐明了本文的主要工作及贡献。

第二章中阐述了本文的研究思路，讲述了本文中算法设计的理念。

第三章为本文的核心部分，从 CPA 的四个组件以及组合 CPA 中的操作符出发，用形式化的语言介绍了各个组件的定义方式。

第四章展示了典型测试样例下的测试结果，验证系统设计的正确性。

第五章是对系统的综合评价。

第 2 章　研究思路

2.1　本章引言

C 语言中的字符串缺陷指的是在用 char 型数组表示字符串时，因字符串本身具备的不当特征导致在调用字符串库函数时遇到错误。

下面介绍本文关注的 C 语言中的一个具体的字符串缺陷。合法的字符串中应该包含字符'\0'，不含'\0'的非法的字符串在调用 C 语言字符串库函数时会导致程序崩溃或者得到一个错误的结果。

本文从字符串缺陷分析的难点出发，介绍了针对四个难点的算法设计方法：结构体域的跟踪、指针分析、过程间分析、库函数的语义建模。

2.2　结构体域的跟踪

在C语言中，结构体是较难分析的结构，因其具有的嵌套的特点而难以分析。在字符串缺陷检测中，我们根据实际需要，只需关注结构体中的 char 型变量、char 型数组、结构体变量、指针等信息。因此我们将名为 a 的结构体映射到一个集合 A，a 中的每个元素又是形如 b->B 的映射，其中 b 为结构体域的名称，B 为域中所包含的子域的信息，B 具有和 A 相同的形式。通过这样的递归定义，可以完整地分析结构体中的成员域。

2.3　指针分析

通过查阅资料，发现指针与它指向的变量存在一一映射的关系，因此可以用一个集合存储所有的指针的指向关系。这种基于指向的分析方法为指针的指向图分析。

2.4　过程间分析

过程间分析分为两个部分：传参分析和返回值分析。在传参分析中，需要将形参用实参进行赋值，并和局部变量一起加入状态中，模拟函数调用中栈空间的

申请；当函数返回时，将形参和局部变量从状态中移除，模拟函数返回时栈空间的释放。在返回值分析中，需要跟踪 return 语句后的变量的结构进行储存，再回到主函数的相应位置进行处理。

2.5 库函数的语义建模

由于分析中字符串在调用字符串库函数的过程中会改变取值，因此需要对常见的字符串库函数进行语义建模。根据字符串库函数对参数以及返回值的作用将库函数分为多个等价类，每个等价类采用同样的模型进行建模分析。

第 3 章　基于数据流分析的缺陷检测算法

3.1　本章引言

在本章中，我们将深入讨论基于数据流分析的缺陷检测算法。我们采用的开源框架是 CPAchecker，这个框架的两篇理论论文分别于 2007 和 2008 年由 Dirk Beyer 提出。2008 年，开源的 CPAchecker 框架正式公布，它为程序分析和模型检测提供了一个融合的平台，可以通过自定义 Configurable Program Analysis（CPA），即可配置程序分析器，利用数据流分析的框架，即第一部分中介绍的 CPA 可达性算法，得到一系列的程序分析状态，从而对程序的性质有了更具体的了解。下面我们介绍这章中需要用到的符号和一些形式化的操作。

type(x): 路径名称为 x 的节点对应的类型。

[prop]: boolean 类型的表达式，当命题 prop 成立时取值为 true，否则取值为 false。

fix(f, x): 单变量函数 f 在变量取值为 x 时反复调用 f 得到的不动点的取值。

filter(s, c): 序列 s 中符合特征字符（串）c 的所有下标组成的序列。

length(a): 序列 a 的长度。

ite(P, a, b): 若 P 为真，取值为 a，否则取值为 b。

3.2　CPAchecker 设计框架

基于 CPA 的定义以及 CPA 可达性算法，Dirk Beyer 设计了一个完成的程序分析框架。下图为框架的架构图。[4]

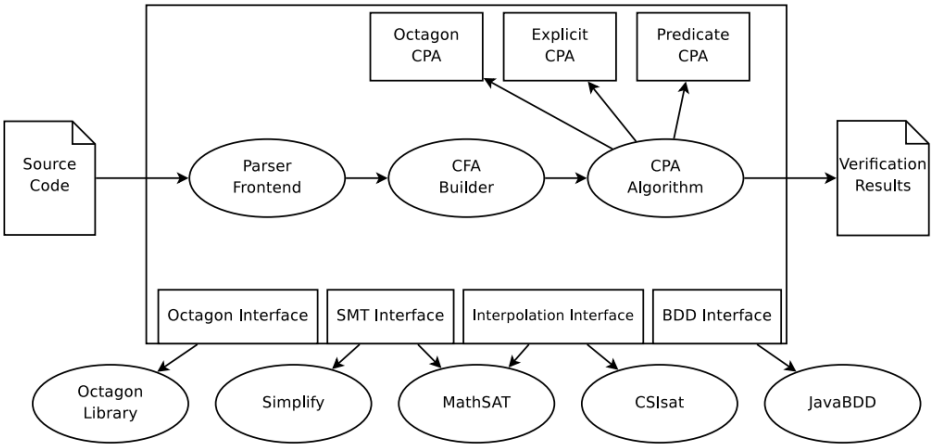

图2.1　CPAchecker架构图

利用 CPAchecker 框架进行程序分析的主要步骤为：

（1）在进行分析之前，将测试代码转化成语法树，再转化成控制流自动机（CFA）。

（2）框架提供了 SMT 求解器，从而使用者可以更准确方便地描述 CPA 中的操作符。

（3）运行 CPAchecker 核心算法——CPA 可达性算法，即算法 2。

（4）过滤可达状态集合，提取出目标状态。

（5）返回结果，输出目标状态及报错信息。

3.3　典型的 CPA 分析器

Dirk Beyer 在提出可配置程序分析的理论基础之后，开发了与之对应的 CPAchecker 开源框架，框架为可配置程序分析提供了平台：通过组合可配置程序分析器（CPA），进行程序分析。他在实现框架的基础了率先实现了几个基础且常用的 CPA，比如进行符号分析的 SignCPA、进行位置分析的 LocationCPA、进行区间分析的 IntervalAnalysisCPA。

在字符串缺陷检测中，位置分析和区间分析是两个重要的辅助 CPA，它们与自定义的字符串缺陷分析器 StringCPA 一起，共同完成字符串缺陷检测的任务。

3.3.1　位置分析

位置分析用于分析程序中的各个程序点的可达性。可以表示为如下形式：

$$\mathrm{L} = (\mathrm{D_L}, \rightarrow_{\mathrm{L}}, \mathrm{merge_L}, \mathrm{stop_L})$$

它跟踪了控制流自动机（CFA）中各个位置的可达性。各个组件的含义如下：

抽象域$\mathrm{D_L}$是建立在 CFA 位置集合 L 上的一个半格。

变迁关系$\rightarrow_{\mathrm{L}}$定义为：$\mathrm{l} \rightarrow_{\mathrm{L}} \mathrm{l}'$当且仅当存在边 g，使得$\mathrm{g} = (\mathrm{l}, \mathrm{op}, \mathrm{l}') \in G$。若不存在这样的 g，则$\mathrm{l} \rightarrow_{\mathrm{L}} \perp$。

merge 操作符定义为：$\mathrm{merge_L} = \mathrm{merge}^{\mathrm{sep}}$。

stop 操作符定义为：$\mathrm{stop_L} = \mathrm{stop}^{\mathrm{sep}}$。

在实际的程序分析中每个组合程序分析器都要加上位置分析，以区分不同的程序状态，其表现出的效果是对测试程序对应的控制流自动机（CFA）中的每个节点进行标号。

3.3.2 区间分析

区间分析用于分析程序中的整型变量的取值范围，每个整型变量的取值范围用一个区间[a,b]表示。当 a 等于 b 时，可以确定整型变量的值。区间分析可以表示为如下形式：

$$\mathrm{I} = (\mathrm{D_I}, \rightarrow_{\mathrm{I}}, \mathrm{merge_I}, \mathrm{stop_I})$$

抽象域$\mathrm{D_I}$定义为一些绑定变量名的区间集合的超集，即$\mathrm{D_I} = \{\mathrm{A_i}\}\ \mathrm{i} \in \mathrm{J}$，其中$\mathrm{A_i} = \{[name, [a, b]] \mid name\ is\ string, a, b \in Z\}$。

变迁关系$\rightarrow_{\mathrm{I}}$当且仅当对整型变量的值进行修改才进行状态转移，否则变迁后的状态与变迁前状态相同。

merge 操作符定义为：$\mathrm{merge_I} = \mathrm{merge}^{\mathrm{sep}}$。

stop 操作符定义为：$\mathrm{stop_I} = \mathrm{stop}^{\mathrm{sep}}$。

下面举一个简单的例子说明区间分析的作用。下表的左侧为代码，右侧为对应的 IntervalAnalysisState 的取值。

表3.1　区间分析举例

测试代码（部分）	**IntervalAnalysisState**
int i;	{}
i = 1;	{[i, [1,1]]}
int j = i + 1;	{[i, [1,1]], [j, [2,2]]}
int k = j + i;	{[i, [1,1]], [j, [2,2]], [k, [3,3]]}

3.4 字符串缺陷分析

字符串缺陷分析分析器 S，与上述位置分析 L、区间分析 I 一样为一种执行特性分析功能的可配置程序分析器。不同的是 S 为专用的程序分析器，只能用来检测 C 语言字符串的'\0'缺失的缺陷。而 L 和 I 使用返回就更加广泛，它们可以作为辅助程序分析器参与其他各种缺陷的检测。在本次缺陷检测的过程中，L 和 I 就承担了重要的角色。一般而言，所有的组合程序分析都需要位置分析 L。对于需要对整型变量做求值分析的组合程序分析也都需要区间分析 I。

如第一部分所述，一个完整的程序分析器包含四个部分：abstract domain、transfer relation、merge operator、stop operator。下面将从这四个方面介绍 S 的四个组件的取值设定。

3.4.1 抽象域（Abstract domain）

S 的 abstract domain 为$(C, \epsilon, ||\cdot||)$。其中 C 为程序的具体状态，是对程序中所有变量的一次赋值。ϵ 为偏序集，$\varepsilon = (E, \bullet, \perp, \text{超})$，E 为程序的抽象状态集合。$||\cdot||$为状态具体化投射函数，将一个抽象状态映射为一个由具体状态组成的集合。

为了完整定义 S 的 abstract domain，只需要完整地定义抽象状态。由于在 S 中的 merge 和 stop 采用的都是 sep 模式，而偏序关系和交汇运算只在 join 模式下才会用到，因此偏序关系和交汇运算也不需要定理。采用 sep 模式而不是 join 模式的原因将在 3.3.3 节进行详细阐述。

E 中的单个元素 StringState 为一个集合，包含以下几个元素。

第一个元素为 StringStructField：类型为 StructField，包含四个记录结构体中成员域的性质的算符，包括 sf、ca、ti、l。

sf：String->StructField。

$$sf(x) = \{sf', ca', tI', l'\}$$

其中sf', ca', tI', l'为定义在名称为 x 的结构体的成员域名称上的算符。这里的结构体是一般情况下的结构体，非结构体的类型 A 可视为只包含一个类型为 A 的单成员结构体。

ca：String->Boolean。

$$ca(x) = [\text{名称为 x 的节点为字符或字符数组}]$$

ca 算符是名称为 x 的节点的特征函数，刻画了该节点的类型属性，将所有的

类型化成了两个等价类：字符或字符数组、其他。

ti: String->Vector

$$\mathrm{ti(x)} = \mathrm{filter(a,'\backslash 0'),a}\ \text{为名称为 x 的节点对应的数组结构}$$

名称为 x 的节点对应的数组结构指的是：若 x 为数组，则 a 为 x 对应的变量，否则 a 为空数组。ti 算符刻画了名称为 x 的节点中'\0'字符的位置信息。需要注意的是，当 x 为字符变量时，若取值为'\0'，$\mathrm{ti(x)} = [-1]$，否则为[]。

l：String->Integer

$$\mathrm{l(x)} = \mathrm{length(a)},\mathrm{a}\ \text{为名称为 x 的节点对应的数组结构}$$

为了方便实际操作，在定义了以上四个算符的基础上，还定义了下列三种算符以保证操作的简洁性。

gc: String->Boolean。设原像$\mathrm{x} \in \mathrm{String}$，且具有形式

$$\mathrm{x = a1.a2.\ldots.an}$$

$$\mathrm{gc(x) = [type\big(sf(a1).sf(a2).\ldots.ca(an)\big) == char\ or\ string]}$$

其中$\mathrm{sf(a1).sf(a2)}$为选出$\mathrm{sf(a1)}$中的 sf 算符，传入参数 a2 得到的运算结果。以下同理。

gtiv: String->Vector。设原像$\mathrm{x} \in \mathrm{String}$，且具有形式

$$\mathrm{x = a1.a2.\ldots.an}$$
$$\mathrm{gtiv(x) = sf(a1).sf(a2).\ldots.ti(an)}$$

gl: String->Integer。设原像$\mathrm{x} \in \mathrm{String}$，且具有形式

$$\mathrm{x = a1.a2.\ldots.an}$$
$$\mathrm{gl(x) = sf(a1).sf(a2).\ldots.l(an)}$$

根据上述定义，可以看出 gc、gtiv、gl 分别得到的是结构体域访问路径名称 x 下的 ca、ti、l 算符的取值。实现的主要方法是递归操作。

StringState 中包含的第二个元素为 pm：String->String。

$$\mathrm{pm(x)} = \text{名称为 x 的指针指向的变量名称}$$

需要说明的是，若 x 为一个多级指针的名称，则 pm(x)的取值为 x 指向的一级指针指向的变量名称，即最终的变量名称。这里的指针分析的原理为基于指针指向图的分析方式。为了方便指针分析，定义了以下操作符。

sp：(String, Integer)->String。设原像$(x, k) \in String \times Integer$，且具有形式：

$$x = a1\ X\ a2\ X \ldots X\ an, \text{其中 } X \text{ 为字符}'.' \text{或字符串} \rightarrow$$
$$m = length(filter(fix(pm, a1Xa2X \ldots Xak), X))$$
$$sp(x, k) = g(fix(pm, a1Xa2X \ldots Xak)Xa_{k+1}X \ldots Xan, m + 1)$$

根据定义，sp 算符将指针表达式根据'.'和"->"分成 n 个小段，sp(x,k)的运算结果将 x 的前 k 个小段映射到最终指向的非指针变量的名称。

s：String->String。设原像$x \in String$，且具有形式：

$$x = a1\ X\ a2\ X \ldots X\ an, \text{其中 } X \text{ 为字符}'.' \text{或字符串} \rightarrow$$
$$s(x) = sp(x, 1)$$

根据定义，s 算符的运算结果为名称为 x 的指针表达式指向的非指针对象的名称。

StringState 中包含的第三个元素为在自然数集 N 上取值的元素 callNumber，即：

$$callNumber \in N$$

callNumber 的语义为：记录当前函数调用的层数。在模拟栈空间扩展和释放的过程中，需要在局部变量的首部加上 callNumber 前缀，以标记各个局部变量的位置信息。

StringState 中包含的第四个元素也是最后一个元素为 returnValue：类型为 StructField，同样包含四个记录结构体中成员域的性质的算符，包括：

$$sf'', ca'', ti'', l''$$

这四个算符是定义在当前函数返回值名称 x（类型为 String）上的单点映射，在$String \setminus \{x\}$上无定义。sf'', ca'', ti'', l''的映射类型与 StringState 的第一个元素 StringStructField 中的四个算符对应相同。

StringStructField 中的算符 sf 的实现代码由软件学院 2014 级硕士研究生方镇澎学长编写，其他部分均由本人独立编写完成，特此说明。

3.4.2 变迁关系（Transfer relation）

在 CPAchecker 的框架下，C 语言的程序操作被划分成以下八个等价类。

$$Op = D \sqcup S \sqcup RS \sqcup FR \sqcup A \sqcup FC \sqcup B \sqcup CTR$$

其中⊔表示集合之间的不交并，应当与前面的偏序集合上的交汇运算予以区分。下面详细介绍这八个等价类的名称和具体含义。

D 为声明操作集合（Declaration），包括函数的声明，类型的声明，以及变量的声明。

S 为语句操作集合（Statement），包括赋值语句，函数调用语句，以及表达式语句。

RS 为返回语句操作集合（Return Statement），即函数中 return 语句所代表的操作。

FR 为函数返回操作集合（Function Return），即函数摘要中类似于 a=func()这样的操作。

A 为假设操作集合（Assume），即 if、for、while 等语句中逻辑判断部分的操作。

FC 为函数调用操作集合（Function Call），即函数调用过程中的传参操作。

B 为空白语句边，即不做任何操作，对程序本身不产生任何影响的操作。

CTR 为调用返回操作集合（Call To Return），即函数摘要的调用和返回操作。

在第一部分中我们介绍了 CPAchecker 在进行状态收集过程的几个阶段，其中最重要的阶段就是 transfer 阶段。Transfer 阶段中系统的主要行为可以用下面的形式化语进行描述：

在字符串缺陷分析器 S 中，

$$S = (D, \rightarrow, \mathbf{merge}, \mathbf{stop})$$

其中 D 中包含一个由抽象状态组成的偏序集 E。

S 中的 transfer relation：→的系统行为可以这样表示：

对于$\forall s \in E, \forall g \in Op$, 根据 g 属于八个不同的等价类的情况求出对应的 s 的后继s'。得到 S 下的 s 的后继s'后，再与其他的 CPA，如区间分析 I，进行通信，得到一个更精确的状态s''。其中s'和s''满足以下关系：

$$s'' \sqsubseteq s'$$

其中⊑为偏序集 E 上的偏序关系。这一节将系统地描述如何根据 g 所处的等价类的不同求出s'。s''的计算方法将在 3.4 节对组合 CPA 的 strengthen 操作进行介绍。

根据 g 所处的等价类的不同，根据 s 计算s'的方法也有所不同。由于当$g \in B \sqcup CTR$时系统状态不改变或者不会出现这样的情况，因此下面只针对前六种等价类

下s'的计算方法进行介绍。

3.4.2.1 处理声明边（Declaration Edge）

当$g \in D$时，我们将 g 所具有的形式进行分类，主要分成以下五个大类：

（a）char x[n] =y。y 可为空，其中 y 具有两种形式 type-y1 和 type-y2。type-y1 为双引号构建的字符串数组初始化子。type-y2 为花括号构建的字符串数组初始化子。举例如下：

type-y0: char x[3]

type-y1: char x[3] = “12”

type-y2: char x[3] = {‘1’, ‘2’, ‘\0’}

在情况（a）中，根据初始化子的不同得到s'也有所不同。

设变迁前状态 s 的 StringStructField 中的四个映射为：

$$sf、ca、ti、l$$

则经过 tranfer 阶段后得到的变迁后状态s'中除去 StringStructField 以外的所有元素取值不变，StringStructField 中的四个映射$\{sf', iCA', tI', l'\}$对应为之前四个映射 {sf、ca、ti、l}在声明节点名称 x 上的延拓。设之前四个映射的定义域为 A，则变迁后的四个映射的定义域为：

$$A' = A \cup \{x\}$$

更具体的刻画如下所示：

$$sf'|_A = sf$$
$$ca'|_A = ca$$
$$ti'|_A = ti$$
$$l'|_A = l$$
$$sf'(x) = \text{null}$$
$$ca'(x) = \text{true}$$
$$ti'(x) = \text{filter}\left(\text{contentY}, '\backslash 0'\right)$$
$$l'(x) = n$$

其中 contentY 为 y 中的字符从左到右构成的字符序列，若 y 为空，则 contentY 为空序列。

（b）char x = y，其中 y 可为空。经过 tranfer 阶段后得到的变迁后状态s'中除去 StringStructField 以外的所有元素取值不变，StringStructField 中的四个映射$\{sf', iCA', tI', l'\}$对应为之前四个映射 {sf、ca、ti、l}在声明节点名称 x 上的延拓。

设之前四个映射的定义域为 A，则变迁后的四个映射的定义域为：

$$A' = A \cup \{x\}$$

更具体的刻画如下所示：

$$sf'|_A = sf$$
$$ca'|_A = ca$$
$$ti'|_A = ti$$
$$l'|_A = l$$
$$sf'(x) = null$$
$$ca'(x) = true$$
$$ti'(x) = ite(y ==' \backslash 0', [-1], [\,])$$
$$l'(x) = 0$$

（c）type-A* x = y，其中 type-A 为指针类型时 x 为多级指针，y 可为空。根据 y 的不同求出 contentY 如下，设初始状态中包含的指针分析的映射为 pm。

当 y 为空时，$contentY = ""$。

当 y 具有形式&z 或者 z 时，contentY=s(z)，即为 z 经过多级指针运算指向的非指针变量。

计算出 contentY 后可以很容易地对s'进行求值。s'中的指针分析的映射pm'为初始状态中映射 pm 在 x 上的延拓，状态中的其他元素的值都不变。设 pm 的定义域为 B，则pm'的定义域为：

$$A' = A \cup \{x\}$$

其中pm'满足如下性质：

$$pm'|_A = pm$$
$$pm'(x) = contentY$$

（d）struct A x，其中结构体类型 A 中或者其嵌套成员中含有以上三种类型：即 char 型变量、char 型数组、指针。

对于所有从 x 开始存在路径 x-a1-a2-…-an，使得 x.a1.a2. … .an 为以上三种类型之一，执行以下操作：运用递归的思路构造出s'的值。

（d.1）若 x.a1.a2. … .an 为 char 型数组或者变量，则先定义 gsf 算符：

$$gsf(a1.a2 \ldots.an) = \{sf'', iCA'', tI'', l''\}$$

其中sf'', iCA'', tI'', l''为单点函数，只在 a1 上定义，定义如下：

$$sf''(a1) = gsf(a2. \ldots.an)$$

$$ca''(a1) = ite(\text{名称为 a1 的节点为 char 型变量或数组，true，false})$$

$$ti''(a1) = [\,]$$

$$l''(a1) = ite(\text{名称为 a1 的节点为 char 型数组 c，} length(c), 0)$$

下面再开始定义s'中 StringStructField 中的四个算符，这四个算符在除去 x 的定义域上取值与原来相同，即为原来的算符在 x 点处的延拓。在 x 处的定义如下：

$$sf'(x) = gsf(a1.a2.\ldots.an)$$
$$ca'(x) = false$$
$$ti'(x) = [\,]$$
$$l'(x) = 0$$

（d.2）若 x.a1.a2. … .an 为指针，则s'中除去映射pm'与 s 中的 pm 不同，其他元素的取值均相同。具体地说，pm'为 pm 在 x.a1.a2. … .an 处的延拓。设 pm 的定义域为 A，则pm'的定义域为：

$$A' = A \cup \{x.a1.a2.\ldots.an\}$$

pm' 在A'上的定义如下：

$$pm'|_A = pm$$
$$pm'(x.a1.a2.\ldots.an) = ""$$

（e）其他情况。如果声明操作不为以上四种情况中的任意一种，则：

$$s' = s$$

此时，声明操作对分析状态无影响，此时 g 的效果等同于等价类 B 中的元素，不需要做状态迁移。

3.4.2.2 处理语句边（Statement Edge）

当$g \in S$时，CPAchecker 将语句操作分成三类：函数调用、表达式语句、赋值语句。根据实际分析需要，只对函数调用和赋值语句做处理。其他情况下，$s' = s$。

（a）函数调用。本次毕业设计的定位为针对常用的字符串函数进行分析，因此选取了常用 18 个字符串库函数，根据参数类型以及操作语义分厂以下几类。

第一大类是对参数列表中第一个参数的取值有修改：strcpy 含有两个参数，会将第二个参数的取值拷贝到第一个参数的前一部分；strcat 含有两个参数，会将第二个参数连接到第一个参数的末尾；strrev 含有一个参数，语义为将字符串做翻转操作。

第二大类是对参数列表中的所有参数没有修改。只有一个参数且为字符串数组的函数有：strdup、strlen。有两个参数，且第一个参数为字符串数组的函数有：strchr、strrchr。有两个参数，且都为字符串数组的函数有：strcmp、strcspn、stricmp、strcmpi、strpkrk、strspn、strstr、strtok。有三个参数且前两个参数为字符串的函数为：strncpi、strncmp、strnicmp。

根据上述分类，我们可以发现在常用的字符串库函数中，对参数中的字符串有修改的函数只有 strrev、strcpy 和 strcat。它们的固有语义会对字符串变量的值进行修改，因此需要针对这三个个函数进行语义建模分析。另一方面，字符串缺陷检测的根本任务在于，当不含有'\0'的字符数组调用字符串库函数的时候，应当予以报错处理，因此在$g \in S$且为字符串库函数的调用时应当检查相关算符在特定值下的取值。由于一共分为 7 个类型进行处理，具体分析如下。

（a.1）g 调用了 strlen 或者 strdup 函数，调用形式为 strlen(a)或者 strdup（a），其中 a 可能为指针。在这种情况下，令

$$s' = s$$

针对如下分析适时给s'加入报错属性，从而报错机制会收集状态s'，最终在输出端显示错误信息。

设初始状态 s 中含有指针指向算符 pm 以及辅助算符 sp 和 s。首先计算 contentA 如下：

$$\mathrm{contentA} = \mathrm{ite}(s(a) == \mathrm{null}, a, s(a))$$

根据定义，若 a 为包含指针，则 contentA 为 a 等价的不含指针的变量；若 a 不包含指针，则 a 即为所求。

取出状态 s 中的辅助算符 gc、gtiv、gl。计算如下逻辑表达式：

$$gc(\mathrm{contentA}) \wedge \mathrm{length}\big(gtiv(\mathrm{contentA})\big) == 0 \wedge gl(\mathrm{contentA}) > 0$$

若上述表达式取值为真，则在s'中加入报错属性，表明 a 中不含'\0'，为非法操作。

由于 strlen 本身的语义不会对参数做修改，因此不需要对 strlen 进行语义建模。

（a.2）g 调用了 strcpy 函数或者 strcat 函数。设第一个参数为 a，第二个参数为 b。针对这两个函数，由于函数本身对第一个参数有修改，因此需要做两步处理。

首先是和（a.1）中的 strlen 做相同的处理，处理过程如下：设初始状态 s 中含有指针指向算符 pm 以及辅助算符 sp 和 s。

$$\text{contentA} = \text{ite}(\text{s(a)} == \text{null}, \text{a}, \text{s(a)})$$

$$\text{contentB} = \text{ite}\big(\text{s(b)} == \text{null}, \text{b}, \text{s(b)}\big)$$

取出状态 s 中的辅助算符 gc、gtiv、gl。计算如下逻辑表达式：

$$\big(\text{gc(contentA)} \wedge \text{length}\big(\text{gtiv(contentA)}\big) == 0 \wedge \text{gl(contentA)} > 0\big)$$

$$\vee\ (\text{gc(contentB)} \wedge \text{length}\big(\text{gtiv(contentB)}\big) == 0 \wedge \text{gl(contentB)} > 0)$$

若表达式取值为 true，则在s′中加入报错属性，表明 a 或者 b 中不含’\0’，为非法操作。

其次需要对 strcpy 和 strcat 进行语义建模，根据 gtiv(contentA) 和 gtiv(contentB)以及gl(contentA)和gl(contentB)确定 a 和 b 的长度以及’\0’的下标，计算出进行 strcpy 或者 strcat 操作后的’\0’位置组成的向量 v，令：

$$\text{gtiv(contentA)} = \text{v}$$

即可。

（a.3）g 调用了 strrev 函数。设函数参数为 a。针对这个函数，由于对参数 a 的取值有修改，因此需要与（a.2）相同，需要做两步处理。

设初始状态 s 中含有指针指向算符 pm 以及辅助算符 sp 和 s。

$$\text{contentA} = \text{ite}(\text{s(a)} == \text{null}, \text{a}, \text{s(a)})$$

取出状态 s 中的辅助算符 gc、gtiv、gl。计算如下逻辑表达式：

$$\text{gc(contentA)} \wedge \text{length}\big(\text{gtiv(contentA)}\big) == 0 \wedge \text{gl(contentA)} > 0$$

若表达式取值为 true，则s′中加入报错属性，表明 a 中不含’\0’，为非法操作。

其次需要对 strrev 进行语义建模，取出状态 s 中的辅助算符 gtiv、gl。

$$\text{len} = \text{gl(contentA)}$$
$$\text{v1} = \text{gtiv(contengA)}$$
$$\text{v2} = [\text{len} - 1 - \text{i} \,|\text{i} \in \text{v1}]$$
$$\text{gtiv(contentA)} = \text{v2}$$

根据上述操作定义，可以将 strrev 的语义抽象成向量的运算，将字符串'\0'的下标以长度值的一半为中心做反射变换。

（a.4-a.7）剩下的四种类型由于对参数没有做出修改，因此不需要对它们进行语义建模，只需要进行报错检查处理即可。根据调用参数个数的不同以及字符串参数的位置分别进行处理，处理的思路与 strlen、strdup 函数的方式相同。关键步骤如下：提取出参数列表中的字符串类型的参数；进行指针分析，找到指向的变量；检查是否为字符串、是否含有'\0'，如果为不含'\0'的字符串，在s'中加入报错信息。

对$g \in S$中函数调用的处理过程对于字符串缺陷分析十分重要，其中不仅仅需要报错处理，还需要对特定字符串库函数进行语义建模。

（b）赋值语句。若$g \in S$且 g 为赋值操作时，设赋值语句具有形式：

$$a = b$$

则根据 a 的类型的不同采用不同的方式计算当前状态 s 的后继状态s'。

（b.1）type(a)为指针，则状态s'除了指针指向映射pm'与初始状态 s 中的 pm 不同以外，其他元素都保持相同。设 A 为 pm 的定义域，令：

$$A_1 = A \setminus \{a\}$$
$$A_2 = A_1 \cup \{a\}$$

pm' 满足以下性质：

$$pm'|_{A_1} = pm|_{A_1}$$
$$pm'(a) = s(b)$$

这样就计算得到了变迁后的状态s'。

（b.2）type(a)为 char，则状态s'除了 StringStructField 中的算符ti'可能与 s 中的 ti 有所不同以外，其他的算符或元素成员均与状态 s 中对应的相同。定义ti'如下，设 A 为 ti 的定义域，令：

$$A_1 = A \setminus \{a\}$$
$$A_2 = A_1 \cup \{a\}$$

ti' 满足以下性质：

$$ti'|_{A_1} = ti|_{A_1}$$

$$ti'(a) = ite\big(length\big(gtiv(b)\big) == 0, [\,], [-1]\big)$$

（b.3）type(a)为 string 下标访问，假设 a 具有形式：c[i],其中 i 为常数下标，对于表达式下标将在 strengthen 操作中进行讨论。

判断 b 是否为'\0'，这里由于篇幅所限，不做详细介绍。判断 b 指向的变量是否为'\0'是一个复杂琐碎的过程，需要分指针和非指针、数组元素与 char 型变量等多种情况进行分析。假定我们能正确获取 b 指向的变量是否为'\0'这一重要信息。

状态s′除了 StringStructField 中的算符ti′可能与 s 中的 ti 有所不同以外，其他的算符或元素成员均与状态 s 中对应的相同。定义ti′如下，设 A 为 ti 的定义域，令：

$$A_1 = A \setminus \{a\}$$
$$A_2 = A_1 \cup \{a\}$$

ti′ 满足以下性质：

$$contentA = ite\big(s(a) == null, a, s(a)\big)$$

$$v = gtiv(contentA)$$

$$ti'(contentA) = ite(b\ 为 \backslash 0, v + \{i\}, v - \{i\})$$

$$ti'|_{A_1} = ti|_{A_1}$$

3.4.2.3 处理函数返回边（Return Statement Edge）

在函数的返回值分析中，我们将指针返回值与非指针返回值同意处理，做法是将指针返回值指向的变量作为返回值返回，而不是返回指针本身。当$g \in RS$，即 g 为 returnStatement 时，根据 return 语句后的返回值，将指针指向的变量或者非指针变量的结构保存在 returnValue 中，其中 returnValue 包括四个算符：

$$returnValue = \{sf', ca', ti', l'\}$$

四个算符根据返回值的不同构造方式也不同。主要分成以下四种情况进行构造。

（a）返回值为字符常量，则：

$$sf'("") = null$$
$$ca'("") = true$$
$$l'("") = 0$$

若字符常量为'\0'，则：

$$ti'("") = [-1]$$

否则：

$$ti'("") = []$$

其中""为返回值的默认 id。

（b）返回值为字符变量或者字符指针，设返回语句具有形式：return rVarExp。设初始状态 s 中指针分析辅助算符 s。

$$pointContent = ite(s(rVarExp) == null, rVarExp, s(rVarExp))$$
$$sf'("") = null$$
$$ca'("")=true$$
$$l'("") = 0$$

若 pointContent 为'\0'，则：

$$ti'("")=[-1]$$

否则：

$$ti'("")=[]$$

（c）返回值为 char 型的数组元素，设返回语句具有形式：return rVarExp[i]。设初始状态 s 中 StringStructField 中有辅助算符 gtiv。

$$v = gtiv(rVarExp)$$
$$ti'("")=ite(i \in v, [-1], [\,])$$
$$sf'("") = null$$
$$ca'("")=true$$
$$l'("") = 0$$

（d）返回值为结构体或者结构体指针，设返回语句具有形式：return rVarExp。设初始状态 s 中含有指针分析辅助算符 s，StringStructField 中含有四个算符：sf、ca、ti、l。

$$pointContent = ite(s(rVarExp) == null, rVarExp, s(rVarExp))$$
$$ti'("")=ti(pointContent)$$
$$sf'("") = sf(pointContent)$$
$$ca'("")=ca(pointContent)$$
$$l'("") = l(pointContent)$$

3.4.2.4 处理函数调用边（Function Call Edge）

由于本次程序分析中只关心 char 型变量及数组的取值空间，因此在函数调用处的传参分析时只考虑 char 型变量及数组以及包含 char 型变量及数组的结构体及其指针的传参。在进行传参分析前，为了防止调用函数中的变量与被调用函数的形参及函数体内的变量重名，将原函数状态中的 StringStructField 所有映射的原像开头加上“callNumber::”，表示为调用第几层函数中的成员，最内层函数没有

这样的前缀。

设参数的形参为 pa，实参为 a。设初始状态 s 中的 StringStructField 含有算符 $\{\mathrm{sf}, \mathrm{ca}, \mathrm{ti}, \mathrm{l}\}$，定义域为 A；含有指针分析算符 pm 和辅助算符 s，其中 pm 的定义域为 B。

（a）若为 char 型变量或者结构体变量或者 char 型数组

$$\mathrm{contentA} = \mathrm{ite}(\mathrm{s}(\mathrm{a}) == \mathrm{null}, \mathrm{a}, \mathrm{s}(\mathrm{a}))$$

则后继状态 s' 中 StringStructField 含有的算符 $\{\mathrm{sf}', \mathrm{ca}', \mathrm{ti}', \mathrm{l}'\}$ 分别对应为 $\{\mathrm{sf}, \mathrm{ca}, \mathrm{ti}, \mathrm{l}\}$ 中的算符在点 a 处的延拓，定义域为：

$$\mathrm{A}' = \mathrm{A} \cup \{\mathrm{pa}\}$$

其中 $\{\mathrm{sf}', \mathrm{ca}', \mathrm{ti}', \mathrm{l}'\}$ 满足如下限制条件：

$$\begin{gathered}
\mathrm{sf}'(\mathrm{pa}) = \mathrm{sf}(\mathrm{contentA}) \\
\mathrm{sf}'|_{\mathrm{A}} = \mathrm{sf}|_{\mathrm{A}} \\
\mathrm{ca}'(\mathrm{pa}) = \mathrm{ca}(\mathrm{contentA}) \\
\mathrm{ca}'|_{\mathrm{A}} = \mathrm{ca}|_{\mathrm{A}} \\
\mathrm{ti}'(\mathrm{pa}) = \mathrm{ti}(\mathrm{contentA}) \\
\mathrm{ti}'|_{\mathrm{A}} = \mathrm{ti}|_{\mathrm{A}} \\
\mathrm{l}'(\mathrm{pa}) = \mathrm{l}(\mathrm{contentA}) \\
\mathrm{l}'|_{\mathrm{A}} = \mathrm{l}|_{\mathrm{A}}
\end{gathered}$$

后继状态 s' 中的其他元素的取值均与初始状态 s 相同。

（b）若为 char 型指针（非 char 型数组）或者结构体指针，则后继状态 s' 中指针分析算符 pm' 为初始状态中的 pm 在 pa 处的延拓，设定义域为：

$$\mathrm{B}' = \mathrm{B} \cup \{\mathrm{pa}\}$$

若实参 a 具有形式&q，则：

$$\mathrm{contentA} = \mathrm{ite}\big(\mathrm{s}(\mathrm{q}) == \mathrm{null}, \mathrm{q}, \mathrm{s}(\mathrm{q})\big)$$

否则：

$$\mathrm{contentA} = \mathrm{ite}\big(\mathrm{s}(\mathrm{a}) == \mathrm{null}, \mathrm{a}, \mathrm{s}(\mathrm{a})\big)$$

pm' 满足以下限制条件：

$$\begin{gathered}
\mathrm{pm}'(\mathrm{pa}) = \mathrm{contentA} \\
\mathrm{pm}'|_{\mathrm{B}} = \mathrm{pm}|_{\mathrm{B}}
\end{gathered}$$

后继状态 s' 中的其他元素的取值均与初始状态 s 相同。

3.4.2.5 处理函数摘要返回边（Function Return Edge）

函数返回边的分析较为简单，由于在函数返回语句处已经得到返回值的结构，因此在函数返回边处只需要对形如 x=func()的赋值语句进行操作即可，s′中元素值的求解方法与g ∈ S中赋值语句中的求解方式相同，在此不再赘述。

需要说明的是，在函数返回处需要对栈空间做模拟，操作步骤与函数调用处相反，即需要将最内层即不含 callNumber 前缀的变量及指针从状态中移除，这个过程模拟的是栈空间释放；同时将上一层函数中的变量及指针的前缀消除，表明回到上一层函数进行执行。

3.4.2.6 处理假设边（Assume Edge）

由于在字符串缺陷检测这个特定问题的分析过程中，并不需要在假设判断边上做操作，只是需要在条件转移时有后继状态与之对应，因此g ∈ A处理函数设计极为简单，初始状态为 s，后继状态为s′，其中：

$$s' = s$$

3.4.3 程序控制操作符（merge 和 stop）

在第一部分介绍的 CPA 可达性算法中，merge 和 stop 操作符具有两种可配置的类型，分别为 sep 模式和 join 模式。在数据流分析中，由于条件执行，数据流的路劲可能会出现分叉和汇合。在汇合时，需要将两个状态合并成一个状态。在 sep 模式下，后到达的状态会覆盖先到达的状态，因此合并的状态即为第二个状态，即：

$$\mathrm{merge}^{\mathrm{sep}}(\mathrm{e}, \mathrm{e}') = \mathrm{e}'$$

在 join 模式下，两个状态的信息都会保留下来，合并后的状态在偏序集中比第二个状态大，即：

$$\mathrm{merge}^{\mathrm{join}}(\mathrm{e}, \mathrm{e}') = e \sqcup e'$$

其中⊔即为 abstract domain 中定义的交汇运算。

由于在 sep 模式下，分叉后的路径会被单独进行处理，没有 join 的过程，因此可能存在路径条数过多导致分析时间复杂度高甚至分析不出结果的情况。但是由于分路径分析，从而对于每条路径下的特定条件分析更精确。在 join 模式下，分叉后的路径可能会出现合并，从而分析的路径条数要比 sep 模式下小，时间复杂度更低。

经过深入研究，在字符串缺陷检测这个特定的分析问题下，sep 模式要由于

join 模式，原因分析如下。

先看第一个例子。

```
int main(int argc, char const *argv[]) {
    char a[10];
    int i;
    for (i = 0; i < 10; i++) {
        a[i] = '\0';
    }
    int length = strlen(a);
}
```

图3.1　代码片段一

若在数据流汇合处对同一对象的 terminateIndex 取交，那么在 for 循环结束时，由于 10 次赋值的下标都不同，最终求交会导致 merge 之后的状态中没有'\0'，这显然与实际的运行情况不符：实际情况中数组 a 的所有位置均被赋值为'\0'。这样存在严重的误报情况。

再看第二个例子：

```
int main(int argc, char const *argv[]) {
    char a[10];
    int i = get(); //接受外界输入
    if (i % 2 == 0) {
        a[0] = '\0';
    } else {
        a[1] = '\0';
    }
    a[0] = '0';
    int length = strlen(a);
}
```

图3.2 代码片段二

第二个例子中由于 i 的取值不定，和外界输入有关，因此在做分析的时候 if 的两个分支都会考虑，且认为均可达。如果在数据流汇合处对同一对象的 terminateIndex 求并，那么在 if-else 语句汇合的地方，a 的 terminateIndex 为[0,1]，在执行 a[0]='0'之后 terminateIndex 的取值为[1]，最终调用 strlen 并没有出错。但

是如果外界输入为偶数，那么实际的执行路径并不会走 else 分支，从而在调用 strlen 的时候出现'\0'缺失，故会导致程序运行错误。因此如果求并，对带来严重的漏报的情况。

然而在偏序关系中，对于同一个对象的 terminateIndex 的操作只有交和并这两种，因此在字符串'\0'缺失这一缺陷检测的过程中，并没有一种能避免严重误报和漏报的 join 操作。

综合以上考虑，最终采取的方案为：设置 merge 和 stop 类型均为 sep，将程序中的不同分支路径分开处理。但由于 sep 模式下会出现严重的路径展开问题，这是目前程序分析领域的一个难点，在本次毕业设计中暂不对这个问题做深入研究。

3.5 组合程序分析

在第一部分中，我们对组合程序分析器做了详细的介绍，在字符串缺陷检测中，组合程序分析器可以写成如下形式：

$$\mathrm{C} = (\mathrm{L}, \mathrm{I}, \mathrm{S}, \rightarrow_{\times}, \mathrm{merge}_{\times}, stop_{\times}, \downarrow, \prec)$$

其中 L、I、S 分别为位置分析、区间分析、字符串缺陷分析对应的程序分析器。↓为 strengthening 操作符，它满足以下性质：

$$\downarrow(\mathrm{e}, \mathrm{e}') \sqsubseteq \mathrm{e}$$

其含义为根据第二个参数状态的信息对第一个参数状态做加强处理，得到一个在偏序集中比第一个参数状态更小的状态，即得到一个更加精确的状态。

≺是比较不同分析器中的状态大小的操作符。由于在大多数组合程序分析中并用到，因此在这部分不做详细介绍。

3.5.1 状态精化操作符（strengthening 操作）

在本次组合程序分析中，需要用区间分析中的状态对字符串缺陷分析中的状态做加强，因此 strengthening 操作符可以写成$\downarrow_{\mathrm{I,S}}$。

在 strengthen 阶段，需要对数组下标进行求值，这就需要区间分析的状态中的信息。根据实际需要，需要在 S、RS、FR 三种情况下的操作做处理，初始方法是利用 I 中的状态对下标进行求值，之后对后继状态的求解方法与 tranfer 阶段类似，只需要把下标表达式的值当做常数值，按照 tranfer 的流程进行处理即可。

在 strengthen 操作中通过区间分析中的状态对表达式进行求值的代码由软件学院 2014 级硕士研究生陈翔学长编写，本人在此基础上应用到了字符串下标表达式的求值，特此说明。

第 4 章　实验评价

4.1　本章引言

在字符串缺陷检测中，主要分成结构体成员的跟踪、下标表达式的求值、条件执行、指针分析、字符串函数的语义建模、函数的调用与返回这六大部分，下面精心设置了几个例子，对系统的正确性进行测试，并评估效率性能。之后通过一个实际案例对程序的正确性进行检测。

4.2　功能测试

4.2.1　结构体成员的跟踪

```
#include <string.h>

struct A{
    char a[3];
    int b;
};

struct B{
    struct A c;
    int d;
};

int main(int argc, char const *argv[]) {
    struct B z;
    int k;
    z.c.a[0] = '1';
    k = strlen(z.c.a);
    z.c.a[3] = '\0';
    k = strlen(z.c.a);
    return 0;
}
```

图4.1　测试样例一测试代码

```
[WARNING][MAY][STRING_LACK_ZERO_ERROR][/home/sunshine/Desktop/Tsmart/
TsmartCodeStandrad/cn.edu.thu.tsmart.tool.bd.cpachecker/cpachecker/testCases/string/stru
ct.c:main:17]: k = strlen(z.c.a);[variable: z.c.a]
[WARNING][MAY][STRING_LACK_ZERO_ERROR][/home/sunshine/Desktop/Tsmart/
TsmartCodeStandrad/cn.edu.thu.tsmart.tool.bd.cpachecker/cpachecker/testCases/string/stru
ct.c:main:19]: k = strlen(z.c.a);[variable: z.c.a]
```

图4.2　测试样例一运行结果

在测试样例一中，定义了两个结构体，并且存在相互嵌套的关系。针对其中字符串成员有两次操作：第一次操作中将第 0 位设置为'1'，然后调用 strlen 函数，这样由于字符串中缺少'\0'因此是非法的；第二次操作虽然设置了'\0'，但是下标超过了字符串下标的范围，因此也是无效的，之后调用 strlen 也是非法的。

根据测试样例一的测试结果，可以看出系统能正确地对结构体中字符串成员进行跟踪和分析，运行结果正确。

4.2.2　下标表达式的求值

```
#include <string.h>

int main(int argc, char const *argv[]) {
    char c = '0';
    char b[3] =    "12";
    char e[6] =    {'0', '1', '1', '\0'};

    int i = 0;
    e[i+3] = c;
    b[i+2] = e[i+3];
    int k;
    k = strlen(b);

    return 0;
}
```

图4.3　测试样例二测试代码

```
[WARNING][MAY][STRING_LACK_ZERO_ERROR][/home/sunshine/Desktop/Tsmart/TsmartCodeStandrad/cn.edu.thu.tsmart.tool.bd.cpachecker/cpachecker/testCases/string/exprIndex.c:main:12]: k = strlen(b);[variable: b]
```

图4.4　测试样例二测试结果

在测试样例二中，对字符串 e 和 b 去下标操作都是基于下标变量表达式的求值。根据样例中的操作，e 和 b 中的’\0’被先后擦除，因此调用 strlen 是非法的。

根据测试样例二的测试结果，可以看出系统能正确地进行整型下标表达式的求值，运行结果正确。

4.2.3　条件执行

```
#include <string.h>

int main(int argc, char const *argv[]) {
    char c = '0';
    char d[3] =    {'0', '1', '\0'};
    char x = '\0';
    d[2] = c;

    int i = 0;
    if (i == 0) {
        d[2] = x;
    } else {
        d[2] = c;
    }
    int k = strlen(d);

    int j;
    for (j = 0; j < 3; j++) {
        d[j] = '0';
    }
    k = strlen(d);
    return 0;
}
```

图4.5　测试样例三测试代码

```
[WARNING][MAY][STRING_LACK_ZERO_ERROR][/home/sunshine/Desktop/Tsmart/
TsmartCodeStandrad/cn.edu.thu.tsmart.tool.bd.cpachecker/cpachecker/testCases/string/fori
f.c:main:21]: k = strlen(d);[variable: d]
[ERROR][MAY][UNREACHABLE_CODE][/home/sunshine/Desktop/Tsmart/TsmartCod
eStandrad/cn.edu.thu.tsmart.tool.bd.cpachecker/cpachecker/testCases/string/forif.c:main:13
]: d[2] = c;
```

图4.6　测试样例三测试结果

在测试样例三中，if 语句中的 else 分支是不可达的，for 循环中擦除了 d 中的’\0’，因此预期的运行结果与实际运行结果吻合。同时，由于进行 sep 模式下的分析，且加入了分析器 I，因此对于 if 语句中的 else 分支不会进行分析，因为它是不可达的，程序分析器最终被报出 else 分支中的代码为 unreachable code。

4.2.4 指针分析

```
#include <string.h>
struct A{
    char a[3];
    int b;
};
struct B{
    struct A c;
    struct A* d;
    struct B** e;
};
int main(int argc, char const *argv[]) {
    struct B z;
    struct B w;
    struct A t;
    struct B* y = &z;
    struct B* u = &w;
    struct B** v;
    int k;
    v = &u;
    y->e = &v;
    (*(y->e))->c.a[0] = '0';
    k = strlen((*(y->e))->c.a);
    y->c.a[0] = '0';
    y->c.a[3] = '\0';
    k = strlen(y->c.a);
    y->d = &t;
    y->d->a[2] = '0';
    k = strlen(y->d->a);
    (*y).c.a[0] = '0';
    k = strlen((*y).c.a);
    int i = 0;
    (*y).c.a[i] = '\0';
    (*u).c.a[i] = '0';
    k = strlen((*u).c.a);
    (*v)->c.a[i] = '0';
    k = strlen((*v)->c.a);
    return 0;
}
```

图4.7　测试样例四测试代码

```
[WARNING][MAY][STRING_LACK_ZERO_ERROR][/home/sunshine/Desktop/Tsmart/TsmartCodeStandrad/cn.edu.thu.tsmart.tool.bd.cpachecker/cpachecker/testCases/string/pointer.c:main:40]: k = strlen((*v)->c.a);[variable: w.c.a]
[WARNING][MAY][STRING_LACK_ZERO_ERROR][/home/sunshine/Desktop/Tsmart/TsmartCodeStandrad/cn.edu.thu.tsmart.tool.bd.cpachecker/cpachecker/testCases/string/pointer.c:main:25]: k = strlen((*(y->e))->c.a);[variable: w.c.a]
[WARNING][MAY][STRING_LACK_ZERO_ERROR][/home/sunshine/Desktop/Tsmart/TsmartCodeStandrad/cn.edu.thu.tsmart.tool.bd.cpachecker/cpachecker/testCases/string/pointer.c:main:38]: k = strlen((*u).c.a);[variable: w.c.a]
[WARNING][MAY][STRING_LACK_ZERO_ERROR][/home/sunshine/Desktop/Tsmart/TsmartCodeStandrad/cn.edu.thu.tsmart.tool.bd.cpachecker/cpachecker/testCases/string/pointer.c:main:28]: k = strlen(y->c.a);[variable: z.c.a]
[WARNING][MAY][STRING_LACK_ZERO_ERROR][/home/sunshine/Desktop/Tsmart/TsmartCodeStandrad/cn.edu.thu.tsmart.tool.bd.cpachecker/cpachecker/testCases/string/pointer.c:main:31]: k = strlen(y->d->a);[variable: t.a]
[WARNING][MAY][STRING_LACK_ZERO_ERROR][/home/sunshine/Desktop/Tsmart/TsmartCodeStandrad/cn.edu.thu.tsmart.tool.bd.cpachecker/cpachecker/testCases/string/pointer.c:main:33]: k = strlen((*y).c.a);[variable: z.c.a]
```

图4.8　测试样例四测试结果

在测试样例四中，充分地测试了指针分析的正确性，测试的内容有：结构体与指针相互嵌套、多级指针。根据测试样例四的测试结果可以看出，实际运行结果与预期的正确结果一致。

4.2.5 字符串函数的语义建模

```
#include <string.h>

int main(int argc, char const *argv[]) {
    char a[5] = {'1', '2', '0', '3', '\0'};
    char d[2] = {'1', '\0'};
    strcpy(a, d);
    d[4] = '0';
    int len1, len2;
    len1 = strlen(d);
    d[1] = '0';
    len2 = strlen(d);
    return 0;
}
```

图4.9 测试样例五测试代码

```
[WARNING][MAY][STRING_LACK_ZERO_ERROR][/home/sunshine/Desktop/Tsmart/TsmartCodeStandrad/cn.edu.thu.tsmart.tool.bd.cpachecker/cpachecker/testCases/string/stringFunction.c:main:11]: len2 = strlen(d);[variable: d]
```

图4.10 测试样例五测试结果

由于分析的字符串函数很多，因篇幅所限，只展示 strcpy 的语义建模的结果。在测试样例五中，调用 strcpy 函数使得字符串 a 在下标为 1 和 5 这两个位置上存在'\0'，因此在第一次调用 strlen 为合法操作，第二次调用 strlen 为非法操作。根据测试样例五测试结果，实际的运行结果与预期的正确结果一致。

4.2.6 函数的调用与返回

```
struct A {
    int n;
    char a[3];
};

struct A fun(struct A* a) {
    (*a).a[0] = '0';
    return (*a);
}

int main(int argc, char const *argv[]) {
    int i;
    struct A a;
    a.a[0] = '\0';
    struct A b = fun(&a);
    i = strlen(a.a);
    i = strlen(b.a);
    return 0;
}
```

图4.11 测试样例六测试代码

```
[WARNING][MAY][STRING_LACK_ZERO_ERROR][/home/sunshine/Desktop/Tsmart/TsmartCodeStandrad/cn.edu.thu.tsmart.tool.bd.cpachecker/cpachecker/testCases/string/functioncall.c:main:16]: i = strlen(a.a);[variable: a.a]
[WARNING][MAY][STRING_LACK_ZERO_ERROR][/home/sunshine/Desktop/Tsmart/TsmartCodeStandrad/cn.edu.thu.tsmart.tool.bd.cpachecker/cpachecker/testCases/string/functioncall.c:main:17]: i = strlen(b.a);[variable: b.a]
```

图4.12 测试样例六测试结果

在测试样例六中，测试了过程间分析中最复杂的两个部分：参数为指针、返回值为结构体。根据函数 fun 的定义，a 和 b 中的字符串数组都缺少’\0’，从而两次调用 strlen 均为非法操作。

根据测试样例六的测试结果，实际运行结果与预期的正确结果相同。

4.2.7 测试指标汇总

表4.1 测试用例的运行指标

测试用例编号	代码行数/函数个数	算法时间开销（秒）	内存(MB)
测试样例一	18/1	15.675	97
测试样例二	11/1	16.120	94
测试样例三	17/1	14.318	104
测试样例四	35/1	6.661	157
测试样例五	11/1	13.926	110
测试样例六	14/2	5.153	168

4.3 检测实际项目

为了进一步检测分析器的正确性，选取了一个具有实际功能的项目代码作为测试代码。代码时间的功能为输入 20 个以内的短句，生成由这些句子组合成的长句。在长句中，短句之间由逗号隔开，长句以句号结尾。

```
#include <stdio.h>
#include <string.h>
int main()
{
   char word[20][20];
   char nullString[1] = "";

   for (int i = 0; i < 20; i++) {
      word[i] = "";
   }

   for (int i = 0; i < 20; i++) {
      gets(word);
   }

   char sentence[500] = "";
   int index = 0;
   for (int j = 0; j < 20; j++) {
      for (int i = 0; i < strlen(word[j]); i++) {
         sentence[index] = word[j][i];
         index++;
      }
      if (strcmp(word[j],nullString) != 0) {
         sentence[index] = ',';
         index++;
      }
   }
   sentence[index-1] = '.';
   puts(sentence);
   return 0;
}
```

图4.13　实际项目代码

[ERROR][MAY][UNREACHABLE_CODE][/home/sunshine/Desktop/Tsmart/TsmartCodeStandrad/cn.edu.thu.tsmart.tool.bd.cpachecker/cpachecker/testCases/string/test1.c:main:20]: sentence[index] = word[j][i];

图4.14　实际项目测试结果

运行时间 8.899 秒，占用内存 88MB。

4.4　误报与漏报分析

针对这个字符串缺陷的分析存在的误报与漏报主要由以下两个方面造成的。

第一，在处理数组时，非 char 型的一维及以上数组以及 char 型二维及以上数组视为同一个对象。

第二，在条件语句中不为整型变量相关的逻辑表达式时，会考虑所有的情况，在某些不可达的分支会被视为可达。

4.5　存在的优缺点

设计实现的算法的优点在于能够处理的程序结构较为完善，能处理常用的指针运算和函数调用与返回以及复杂的结构体嵌套，并能对常见的 C 语言字符串函数进行安全性检查，分析其中的字符串参数的缺陷。

缺点除了在 4.4 中的两个方面以外，还存在条件分支和循环展开的问题，会导致程序分析耗时增大，甚至分析不出结果。

第 5 章　结论

根据第四部分展示的结果表明：系统能正确地分析给出的测试样例。设计实现的算法能够处理的程序结构较为完善，能处理常用的指针运算和函数调用与返回以及复杂的结构体嵌套，并能对常见的 C 语言字符串函数进行安全性检查，分析其中的字符串参数的缺陷。但是由于设计结构本身的局限性，系统仍然存在以下问题。

第一个问题是对于数组的分析。除了 char 型的一维数组，其他类型的数组都被当作一个非数组元素进行分析的，即数组中的每个对象均视作同一个对象。由此会带来一定程度的漏报和误报的现象。可能的解决方法是在记录数组结构的时采用多级映射的机制进行存储，这种策略可以存储指定维数的数组。

第二个问题是当前程序分析框架下的普遍问题。在所配置的三个程序分析器中，merge 和 stop 操作符的类型均为 sep，从而存在路径展开的问题，导致分析的时间复杂度过高。可能存在的解决方法是扩展条件判断（Assume Edge）中变量的类型，将目前只针对于整型的条件判断扩展到布尔类型、字符类型等其他类型，对路径展开时不可达的路径进行剪枝。

插图索引

表格索引

参考文献

[1]. Yingfei, X.: Teaching slides of Program Analysis Technology, Peking University, 2015, http://sei.pku.edu.cn/~xiongyf04/SA/2015/main.htm

[2]. Beyer, D., Henzinger, T.A., and Theoduloz, G.: Configurable software verification: concretizing the convergence of model checking and program analysis. In Proc.CAV, LNCS 4590, pages 504-518. Springer, 2007

[3]. Beyer, D., Henzinger, T.A., Theoduloz, G.: Lazy shape analysis. In: Ball, T., Jones, R.B.(eds.) CAV 2006. LNCS, vol. 4144, pp. 532–546. Springer, Heidelberg(2006)

[4]. Beyer, D., Keremoglu, M.E.:CPAchecker: A tool for configurable software verification. Preprint arXiv:0902.0019.

[5]. Beyer, D., Henzinger, T.A., and Theoduloz, G.: Program analysis with dynamic precision adjustment. In Proc. ASE.IEEE, 2008

[6]. Ball, T., Podelski, A., Rajamani, S.K.: Boolean and cartesian abstractions for model checking C programs. In: Margaria, T., Yi, W. (eds.) ETAPS 2001 and TACAS 2001. LNCS, vol. 2031, pp. 268–283. Springer, Heidelberg (2001)

[7]. Beyer, D., Henzinger, T.A.,Theoduloz, G.: Lazy shape analysis. In: Ball, T., Jones, R.B.(eds.) CAV 2006. LNCS, vol. 4144, pp. 532–546. Springer, Heidelberg (2006)

[8]. Blanchet, B., Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Min´ e, A., Monniaux, D., Rival, X.: Design and implementation of a special-purpose static program analyzer for safetycritical real-time embedded software. In: Mogensen, T.Æ., Schmidt, D.A., Sudborough, I.H.(eds.) The Essence of Computation. LNCS, vol. 2566, pp. 85–108. Springer, Heidelberg(2002)

[9]. Codish, M., Mulkers, A., Bruynooghe, M., de la Banda, M., Hermenegildo, M.: Improving abstract interpretations by combining domains. In: Proc. PEPM, pp. 194–205. ACM Press, New York (1993)

[10]. Cousot, P., Cousot, R.: Abstract interpretation: A unified lattice model for the static analysis of programs by construction or approximation of fixpoints. In: Proc. POPL, pp. 238–252.ACM Press, New York (1977)

[11]. Cousot, P., Cousot, R.: Systematic design of program analysis frameworks. In: Proc. POPL, pp. 269–282. ACM Press, New York (1979)

[12]. Cousot, P., Cousot, R.: Compositional and inductive semantic definitions in fixpoint, equational, constraint, closure-condition, rule-based and game-theoretic form. In: Wolper, P. (ed.) CAV 1995. LNCS, vol. 939, pp. 293–308. Springer, Heidelberg (1995)

[13]. Dwyer, M.B., Clarke, L.A.: A flexible architecture for building data-flow analyzers. In: Proc.ICSE, pp. 554–564. IEEE Computer Society Press, Los Alamitos (1996)

[14]. Fischer, J., Jhala, R., Majumdar, R.: Joining data flow with predicates. In: Proc. ESEC/FSE, pp. 227–236. ACM Press, New York (2005)

[15]. Gulwani, S., Tiwari, A.: Combining abstract interpreters. In: Proc. PLDI, pp. 376–386. ACM Press, New York (2006)

[16]. Henzinger, T.A., Jhala, R., Majumdar, R., Sutre, G.: Lazy abstraction. In: Proc. POPL, pp. 58–70. ACM Press, New York (2002)

[17]. Lerner, S., Grove, D., Chambers, C.: Composing data-flow analyses and transformations. In:Proc. POPL, pp. 270–282. ACM Press, New York (2002)

[18]. Lev-Ami, T., Sagiv, M.: TVLA: A system for implementing static analyses. In: Palsberg, J.(ed.) SAS 2000. LNCS, vol. 1824, pp. 280–301. Springer, Heidelberg (2000)

[19]. Martin, F.: PAG: An efficient program analyzer generator. STTT 2, 46–67 (1998)

[20]. Mauborgne, L., Rival, X.: Trace partitioning in abstract interpretation based static analyzers.In: Sagiv, M. (ed.) ESOP 2005. LNCS, vol. 3444, pp. 5–20. Springer, Heidelberg (2005)

[21]. Necula, G., McPeak, S., Rahul, S., Weimer, W.: CIL: Intermediate language and tools for analysis and transformation of C programs. In: Horspool, R.N. (ed.) CC 2002 and ETAPS 2002. LNCS, vol. 2304, pp. 213–228. Springer, Heidelberg (2002)

[22]. Sagiv, M., Reps, T.W., Wilhelm, R.: Parametric shape analysis via 3-valued logic. ACM TOPLAS 24, 217–298 (2002)

[23]. Schmidt, D.A.: Data-flow analysis is model checking of abstract interpretations. In: Proc.POPL, pp. 38–48. ACM Press, New York (1998)

[24]. Steffen, B.: Data-flow analysis as model checking. In: Proc. TACS, pp. 346–365 (1991)

[25]. Tjiangan, S.W.K., Hennessy, J.: SHARLIT: A tool for building optimizers. In: Proc. PLDI, pp. 82–93. ACM Press, New York (1992)

致　谢

感谢孙家广老师对我工作的耐心指导，同时感谢周旻老师对我研究的帮助。软件学院系统与工程研究所 2014 级硕士研究生方镇澎、陈翔学长也给我很大的帮助，在此鸣谢。

声　明

本人郑重声明：所呈交的学位论文，是本人在导师指导下，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，本学位论文的研究成果不包含任何他人享有著作权的内容。对本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。

签　名：_____日　期：_____

附录 A　外文资料的调研阅读报告

1、Introduction

Precision and efficiency are two basic requirements in automatic program verification. In general, if a method is more precise, fewer false positives will be produced, but the method itself will be more expensive, and hence applicable to fewer programs. Two main approaches to static program verification reflect this trade-off: program analysis and model checking. Dirk Beyer, Thomas A.Henzinger, and Gregory Theoduloz extended the software model checker BLAST such that it can permit configurable program analyses in order to experiment with the trade-offs. On the purpose of customizing the execution of a program analysis, a meta engine is defined and implemented, which needs to be configured by providing and adding a **merge** operater, a **termination** check and some abstract interpreters. When several abstract interpreters are combined, the analysis can get an amazing effect. In this case, the meta engine can be configured and implemented by defining a **composite termination check** in the component termination checks; a **composite merge operator** in component merge operators; and a **composite transfer function** in the component transfer functions. It can yield dramatic results when the superiorities of various execution engines for various abstract interpreters are combined, and these results are shown in their paper published in 2007.

Dirk Beyer and M.Erkan Keremoglu illustrate the architecture and implementation of CPAchecker(a tool for configurable software verification) in their paper published in 2009.

In this paper, their work will be summarized and a brief introduction for configurable program analyses and the CPAchecker tool will be made.

2、Some Basic Concepts in Data Flow Analysis

This part is referred to the teaching slides of Program Analysis Technology in Peking University.

2.1 Mathematical Basis

Definition 2.1 Semilattice. The semilattice is a two-tuple $(S, ó)$. S is a set, ó is a join operator. For all $x, y, z \in S$, the following properties hold:

(1)Idempotence: $x ó x = x$

(2)Commutativity: $x 篌 y = y 篌 x$

(3)Associativity: $(x 篌 y) 篌 z = x 篌 (y 篌 z)$

(4)The greatest element exists, such that $x ó • = x$ for all $x \in S$.

Definition 2.2 Partial Order. Partial order is a two-tuple $(S, ô)$. S is a set, ô is a binary relation on S. Partial order has the following properties:

(1)Reflexivity: $\forall a \in S : a ô a$

(2)Transitivity: $\forall x, y, z \in S : x 赵 y \wedge y 篌 z \Rightarrow x ? z$

(3)Asymmetry: $x 赵 y \wedge y 篌 x \Rightarrow x = y$

Each semilattice defines a partial order: $x ô y$ if and only if $x ó y = x$.

Definition 2.3 Greatest lower bound.

Lower bound: For a given set S, u is a lower bound of S, if $\forall s \in S : u ô s$.

Greatest lower bound: Assume that u is a lower bound of S. For any lower bound u', if $u' ô u$, u is the greatest lower bound of S, denoted by $•_S$.

Lemma 2.4 $ó_{s \in S} s$ is the greatest lower bound of S.

Proof. According to the idempotence, commutativity and associativity of S, it is obvious that $\forall v \in S : (篌_{s \in S} s) 篌 v = ?_{s \in S} s$. So $ó_{s \in S} s$ is a lower bound of S.

For any other lower bound u, $\forall s \in S : s 篌 u = u, (_{s \in S} s) 篌 u = (_{s \in S} (s ? u)) = u$. So $ó_{s \in S} s$ is the greatest lower bound of S.

Corollary 2.5 Any subset of a semilattice has the greatest lower bound.

Definition 2.6 Monotone function. For a given Partial order $(S, ô)$, call a function defined on S a monotone function when $\forall a, b \in S, a 赵 b \Rightarrow f(a) 篌 f(b)$

2.2 Monotone Framework of Data-flow Analysis

The components of data-flow analysis are the following:[1]

(1) A control-flow graph (V, E)

(2) A semilattice $(S, ó)$ in a finite height

(3) An assigned value I for node **entry**

(4) A set of transfer functions. $\forall v \in V - entry$, there exists a node transfer function f_v.

Algorithm 2.7 Data-flow analysis. [1]

$DATA_{entry} = I$

$\forall v \in (V - entry) : DATA_v \leftarrow \bullet_V$

$ToVisit \leftarrow V - entry$

$While(ToVisit.size > 0)\{$

$v \leftarrow any\ node\ in\ ToVisit$

$ToVisit -= v$

$MEET_v \leftarrow \sqcap_{w \in pred(v)} DATA_w$

If ($DATA_v \neq f_v(MEET_v)$) $ToVisit \cup = succ(v)$

$DATA_v \leftarrow f_v(MEET_v)$.

}

Remark 2.8 Transfer functions are monotone. In most data-flow analyses, the transfer function has the form $f(D) = (D - KILL) \cup GEN$. [1]

Theorem 2.9 The safety of data-flow analysis. For any node v_i in the control-flow graph and all sets of paths from **entry** to v_i P, the following property holds:

$$DATA_{v_i} \sqsubseteq \sqcap_{v_1 v_2 v_3 \cdots v_i \in P} f_{v_i} \circ f_{v_{i-1}} \circ \cdots f_{v_1}(I_{entry})$$ [1]

Proof. For any path $v_1 v_2 v_3 \cdots v_i$, calculation for $DATA_{v_i}$ is equal to MEET join operation between all transfer functions $f_{v_i} \circ f_{v_{i-1}}$. According to the idempotence, any result after MEET join operation is smaller than the original value in the partial order. Then according to the monotonicity of transfer functions, $DATA_{v_i}$ is smaller than $f_{v_i} \circ f_{v_{i-1}} \circ \cdots f_{v_1}(I_{entry})$. Because of the randomicity of paths, $DATA_{v_i}$ is a lower bound. According to Lemma 2.4, $\sqcap_{v_1 v_2 v_3 \cdots v_i \in P} f_{v_i} \circ f_{v_{i-1}} \circ \cdots f_{v_1}(I_{entry})$ is the greatest lower bound. The proof of the theorem finishes.

Definition 2.10 Fixed point. For a given function $f : S \to S$, if f(x)=x, x is a fixed point of f.

Lemma 2.11 Fixed point theorem. For a given semilattice $(S, \sqcap)$ in a finite height and a monotone function f, the sequence $\top, f(\top), f(f(\top)), \cdots$ converges to the greatest fixed point,i.e., there exist non-negative integer n, such that $f^n(\bullet_S)$ is the greatest fixed point of f.

Proof. It is obvious that $f(\top) \sqsubseteq \top$, then apply f to two ends, we get $f(f(\top)) \sqsubseteq f(\top)$. Apply f again, we get $f(f(f(\top))) \sqsubseteq f(f(\top)) \cdots$

So the sequence is a decreasing sequence. Beacause the lattice has a finite height, there exists a position after which the elements are equal and get to a fixed point.

It is easy to get the conclusion that the sequence converges to the greatest fixed point. Assume that there exists another fixed point u, then $u \hat{o} \bullet_S$. Apply f to two ends and get the conclusion.

The proof for the lemma finishes.

Theorem 2.12 The convergence of data-flow analysis. The algorithm 2.7 can end after a finite times of executions.[1]

Proof. For a given strategy of node selection, algorithm 2.7 can be regarded as the application of the function f:

$$(DATA_{v_1}, DATA_{v_2}, \cdots, DATA_{v_n}) = f(DATA_{v_1}, DATA_{v_2}, \cdots, DATA_{v_n})$$

According to Fixed Point Theorem, the algorithm can terminate in finite steps and obtain the greatest fixed point.

3、Formalism and Algorithm

This section summarizes the main work of Dirk Beyer, Thomas A.Henzinger, and Gregory Theoduloz in their paper.[2]

We first introduce some concepts in formalism.

(1) **program**: represented by a **control-flow automaton**(CFA). A CFA consists of a set L of control locations which models the program counter pc.

(2) An initial location pc_0: it models the program entry.

(3) A set $G \subset L \times Ops \times L$ of control-flow edges: it models the operation which is executed when control flows from one location to another.

(4) **concrete state** of a program: It is a variable assignment **c** that assigns to each variable from $X \cup \{pc\}$ a value. The set of all concrete states of a program is denoted by C

(5) Each edge $g \in G$ defines a (labeled) transition relation $\xrightarrow{g} \subset C \times \{g\} \times C$. (6) The complete transition relation $\rightarrow$ is union over all edges: $\rightarrow = \cup_{g \in G} \xrightarrow{g}$

(7) A concrete state c_n is **reachable** from a region r, denoted by $c_n \in Reach(r)$, if there exists a sequence of concrete state $< c_0, c_1, ..., c_n >$ such that $c_0 \in r$ and for all $1 \le i \le n$, we have $c_{i-1} \rightarrow c_i$.

3.1 Configurable Program Analysis

A **configurable program analysis** $D = (D, \hat{I}, \textbf{merge}, \textbf{stop})$ is formed by an abstract domain D, a transfer relation I, a merge operator **merge**, and a termination check **stop**, which are introduced in the following in details. Our algorithm is configured by these

four components. The precision and cost of a program analysis are influenced by these four components..[2]

1. The **abstract domain** $D = (C, \varepsilon, \|\cdot\|)$ is composed of a set of concrete states(C), a semi-lattice ε, and a concretization function $\|\cdot\|$. The semi-lattice $\varepsilon = (E, \bullet, \perp, \sqsupseteq)$ consists a (possibly infinite) set E of elements, a bottom element $\perp \in E$, a top element $\bullet \in E$, a preorder $\sqsupseteq \subset E \times E$, and a total function $\sqcup : E \times E \to E$ (the join operator). Each lattice element from E is the abstract state. The concretization function $\|\cdot\| : E \to 2^C$ assigns to each abstract state its meaning, which means the set of concrete states that it represents. The objective of the analysis is determined by abstract domain.[2]

2. The **transfer relation** $\hat{\to} \subset E \times G \times E$ assigns to each abstract state e possible new abstract e' which are abstract successors of e. Each control-flow edge g labels a transfer relation. We write $e \xrightarrow{g} e'$ if $(e, g, e') \in \hat{\to}$, and $e \hat{\to} e'$ if there exists a g with $e \xrightarrow{g} e'$. In soundness and progress of the program analysis, the abstract domain and the corresponding transfer relation need to fulfill the following requirements:[2]

(a) $\|\bullet\| = C$ and $\|\perp\| = \varnothing$;

(b) $\forall e, e' \in E : \| e \sqcup e' \| \supseteq \| e \| \cup \| e' \|$.(the join operator is precise or over-approximates);

(c) $\forall e \in E : \exists e' \in E : e \hat{\to} e'$.(the transfer relation is total);

(d) $\forall e \in E, g \in G : \cup_{e \hat{\to} e'} \| e' \| \supseteq \cup_{c \in \| e \|} \{c' \mid c \xrightarrow{g} c'\}$.(the transfer relation over-approximates operations).

3. The merge operator **merge**: $E \times E \to E$. The information of two abstract states is combined by the merge operator. In order to guarantee soundness of our analysis, we require $e' \sqsupseteq \mathbf{merge}(e, e')$. This requirement means that the result can only be more abstract than the second state, which is dependent on the first element e. The result of **merge** can be anything between e' and $\bullet$, which means that the operator weakens the second parameter depending on the first state. Furthermore, if D is a composite analysis, some of the components dependent on other can be joined by the operator **merge**. It is apparent that the operator **merge** is not commutative. Although it is not equal to the join operator $\sqcup$ of the lattice, **merge** can be based on $\sqcup$. In the following we will use the merge operators: [2]

$\mathbf{merge}^{sep}(e, e') = e'$ and $\mathbf{merge}^{join}(e, e') = e \sqcup e'$.

4. termination check **stop**: $E \times 2^E \to \mathrm{B}$. It is used to check if the abstract state which is given as first parameter is covered by the set of abstract states which is given as second

parameter. We require for soundness of the termination check that $\mathbf{stop}(e, R) = true$ implies $\| e\| \subseteq \cup_{e' \in R} \| e'\|$. The termination check can, for example, go through the elements of the set R that is given as second parameter and search for a single element that subsumes(ô) the first parameter, or ——if D is a powerset domain(A powerset domain is an abstract domain such that $\| e_1 \grave{o} e_2\| = \| e_1\| \cup \| e_1\|$)——can join the elements of R to check if R subsumes the first parameter. It is also apparent that the termination check **stop** is not equal to the preorder ô of the lattice, but **stop** can be based on ô . In the following we will use the termination checks(the second requires a powerset domain):[2]

$\mathbf{stop}^{sep} = (\exists e' \in R : e \text{ 逖} e')$ *and* $\mathbf{stop}^{join}(e, R) = (e \ ?_{e' \in R} \ e')$.

Obviously, the precision of the analysis is not determined by the abstract domain itself. Both precision and cost are influenced by each of the four configurable components independently.

Location analysis is frequently used in practice.

Location Analysis. A configurable program analysis $\mathrm{L} = (D_{\mathrm{L}}, \hat{\mathrm{I}}_{\mathrm{L}}, \mathbf{merge}_{\mathrm{L}}, \mathbf{stop}_{\mathrm{L}})$ which tracks the reachability of CFA locations is composed of the following components: the domain D_{L} which is based on the flat lattice for the set L of CFA locations; the transfer relation $\hat{\mathrm{I}}_{\mathrm{L}}$ with $l \hat{\mathrm{I}}_{\mathrm{L}} l'$ if there exists an edge $g = (l, op, l') \in G$, and $l \hat{\mathrm{I}}_{\mathrm{L}} \perp$ otherwise(the syntactical successor in CFA without considering the semantics of the operation op); the merge operator $\mathbf{merge}_{\mathrm{L}} = \mathbf{merge}^{sep}$; and the termination check $\mathbf{stop}_{\mathrm{L}} = \mathbf{stop}^{sep}$.[2]

3.2 Execution Algorithm

For a given configurable program analysis and a given initial abstract state, the reachability algorithm CPA computes a set of reachable abstract states, which is an over-approximation of the set of reachable concrete states. The CPA is configured by the abstract domain D, the transfer relation I of the input program, the merge operator **merge**, and the termination check **stop**. The CPA algorithm keeps updating two sets of abstract states： one set named **reached** stores all abstract states that are found to be reachable, and the other set named **waitlist** stores all abstract states that are not yet processed. States are explored by the CPA algorithm. For a current abstract state e, the algorithm considers each successor e', which is obtained from the transfer relation. Now, by using the given operator **merge**, the abstract successor state is combined with

an existing abstract state from **reached**. If the operator **merge** has added new information to the new abstract state, such that the old abstract state is subsumed, then the old abstract state is replaced by the new one. If after the merge step the resulting new abstract state is not covered by the set **reached**, then it is added to the set **reached** and to the set **waitlist**.[2]

Algorithm 3.1 CPA(D, e_0**)**[2]

Input: a configurable program analysis D = (D,Î ,**merge**,**stop**), an initial abstract state $e_0 \in E$, let E denote the set of elements of the semi-lattice of D.

Output: a set of reachable abstract states.

Variables: a set **reached** of elements of E, a set **waitlist** of elements of E.

waitlist = $\{e_0\}$

reached = $\{e_0\}$

while{**waitlist** ≠ ∅}**do**

pop e from **waitlist**

for each e' with e Î e' **do**

for each $e'' \in$ **reached** **do**

//Combine with existing abstract state.

e_{new} = **merge**(e', e'')

If $e_{new} \neq e''$ **then**

waitlist = (**waitlist** ∪ {e_{new}}) , {e''}

reached = (**reached** ∪ {e_{new}}) , {e''}

if **stop**(e',**reached**) == **false** **then**

waitlist = **waitlist** ∪ {e'}

reached = **reached** ∪ {e'}

return reached

Theorem 3.2 (Soundness). In a given configurable program analysis D and for an initial abstract state e_0, a set of abstract states is computed by Algorithm CPA, which over-approximates the set of reachable concrete states: $\cup_{e \in CPA(D,e_0)}$ Õ *Reach*(‖ e_0‖) .[2]

In Dirk Beyer's paper, they also showed how model checking and data-flow analysis are instances of configurable program analysis.[2]

Combinations of Model Checking and Program Analysis. Because of the fact that the model checking algorithm never uses a join operator, the analysis is automatically path-

sensitive. In contrast, the use of a more precise data-flow lattice which can distinguishes abstract states on different paths is required by path-sensitivity in data-flow analysis. On the other way, because of the join operations, the fixed point can be reached much faster in many cases. Significant differences in precision and cost are exhibited by different abstract interpreters, depending on the way to choose the merge operator and termination check. Therefore, we need a mechanism to combine the best choices of the operators for different abstract interpreters when composing the resulting program analyses.[2]

3.3 Composite Program Analyses

A configurable program analysis can be equipped with several configurable program analyses. A **composite program analyses** $C=(D_1,D_2,?_\times,\mathbf{merge}_\times,\mathbf{stop}_\times)$ consists of two configurable program analyses D_1 and D_2, a composite transfer relation $\hat{I}_\times$, a composite merge operator $\mathbf{merge}_\times$, and a composite termination check $\mathbf{stop}_\times$. The three composites $\hat{I}_\times$, $\mathbf{merge}_\times$, $\mathbf{stop}_\times$ are expressions over components of D_1 and D_2 $(蟺_i,\mathbf{merge}_i,\mathbf{stop}_i,\|\cdot\|_i,E_i,\ _i,\perp_i,趄,\ _i)$, as well as the operators $\downarrow$ and $\prec$ (defined below). The composite operators can manipulate lacttice elements only through those components, never directly(e.g., if D_1 is already a result of a composition, then we cannot access the tuple elements of abstract states from E_1, nor redefine $\mathbf{merge}_1$). The only way of using additional information is through the operator $\downarrow$ and $\prec$.[2]

(1) The **strengthening** operator $\downarrow: E_1\times E_2\to E_1$ computes a stronger elements from the lattice set E_1 by using the information of a lattice element from E_2; it has to meet the requirement $\downarrow(e,e')$ ô e. The strengthening operator can be used to define a composite transfer relation $\hat{I}_\times$ that is stronger than a pure product relation. [2]

Example 3.3 If we combine predicate abstraction and shape analysis, the strengthening operator $\downarrow_{S,P}$ can sharpen the field predicates of the shape graphs by considering the predicate region. [2]

(2) We allow the definitions of composite operators to use the **compare** relation $\prec\subseteq E_1\times E_2$, to compare elements of different lattices.[2]

For a given composite program $C=(D_1,D_2,?_\times,\mathbf{merge}_\times,\mathbf{stop}_\times)$, we can construct a configuralbe program analysis $D_\times=(D_\times,\hat{I}_\times,\mathbf{merge}_\times,\mathbf{stop}_\times)$, where

(1) The product domain $D_\times$ is defined as the direct product of D_1 and D_2: $D_\times=D_1\times D_2=(C,\varepsilon_\times,\|\cdot\|_\times)$.

(2) The product lattice is $\varepsilon_\times=\varepsilon_1\times\varepsilon_2=(E_1\times E_2,(晻,\ _2),(\perp_1,\perp_2),趄,\ _\times)$ with

$(e_1, e_2) \hat{o}_{\times} (e_{1'}, e_{2'})$ iff

$e_1 \hat{o}_1 e_{1'}$ and $e_2 \hat{o}_2 e_{2'}$, and (e_1, e_2) 蝌 $(e_{1'}, e_{2'}) = (e_1 \ _1 e_{1'}, e_2 ? e_{2'})$.

(3) The product concretiazation function $\|\cdot\|_{\times}$ is such that $\|(d_1, d_2)\|_{\times} = \| d_1\|_1 \cap \| d_1\|_2$.

The literature agrees that this direct product itself is often not sharp enough. Even improvements over the direct product do not solve the problem completely. However, in a configurable program analysis, we can specify the desired degree of sharpness in the composite operators $\hat{I}_{\times}$, $\mathbf{merge}_{\times}$, and $\mathbf{stop}_{\times}$. For a given product domain, the definition of the three composite operators determines the precision of the resulting configurable program analysis. The composite program analysis is the nature and foundation of CPAchecker.[2]

Remark 3.4 Remark for Algorithm 3.1 When searching for the abstract successors of state e in the transfer relation, we first get the successors of e in each component CPA of composite CPA. For example, the composite CPA has two compoent CPAs: location analysis and sharp analysis. e is a composite abstract state. We first decompose e and search the successors of each component state of e. In the end, compute the cartesian product of these two sets which contains the successors of each component state of e. This cartesian product can be regarded as a rough approximation of the successors of e. Composite CPA can provide a more precise approximation by using the **strengthening** operator. In many cases, one component domain can be associated with another, so we can strengthen each component successor of e by using other component. In this way, we can get a smaller successor in the composite lattice.

4、CPAchecker: A Tool for Configurable Software Verification

CPAchecker is a platform for configurable program analysis, designed by Dirk Beyer and M.Erkan Keremoglu. In their paper, they illustrated the main concept of CPAchecker, and introduced the architecture and implementation of CPAchecker.

4.1 Overview

A conceptual basis for expressing different approaches in the same formal setting is provided by a configurable program analysis(CPA). An interface for the definition of program analyses is offered by the CPA formalism, which includes the abstract domain, the transfer relation(post operator), the merge operator, and the stop operaotr. Consequently, the tool implementation CPAchecker provides an implementation

framework which allows the seamless integration of program analyses that are expressed in the CPA framework. Developers can express their algorithms in CPA framwork, and then implement them in CPAcheckers. The main feature of the tool is that it can be regarded as a set of components that are loosely dependent on each other and that are easy to substitute.[4]

4.2 Architecture and Implementation

An overview of CPAchecker architecture in Figure 1. The central data structure is control-flow automata(CFA), which consist of control-flow locations and control-flow edges. In a CFA, a location represents a program-counter value, and an edge represents a program operation, which is either an assume operation, an assignment block, a function call, or a function return. The framework works in these several steps:[4]

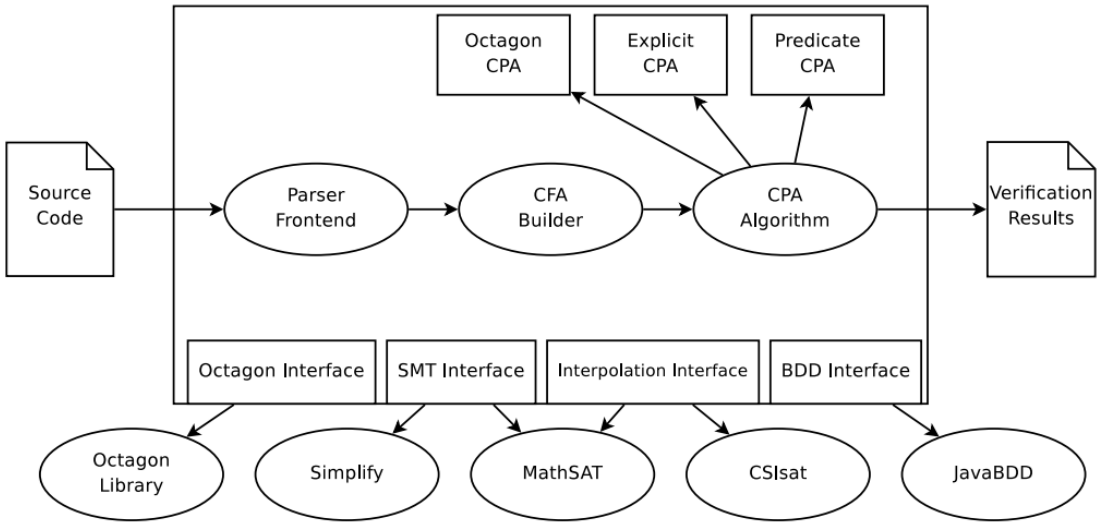

Figure 1: CPAchecker-Architecture overview[4]

(1) Before a start, transform the input program into a syntax tree, and further into CFAs.

(2) The framework provides interfaces to SMT solvers and interpolation procedures, so that developers can write the CPA operators in a concise and convenient way.

(3) Run the central algorithm which the program analysis algorithm for reachablity analysis introduced in Algorithm 3.1. When adding reached state into the reached set, analysis result is added to the state. For example, if we need to check uninitialized variables, we will add the list of uninitialized variables in each state.

(4) Fliter the reached set of states, and get the target states.

(5) Return the results and output to console.

Only two steps are necessary if developers want to extend CPAchecker by integrating an additional CPA for a new abstract domain. First, it is necessary to add an entry in the global properties file to announce the new CPA for composition. Second, implemente the interface for CPA, and provide implementations of all CPA operation interfaces.
Figure 2 shows the interaction: The CPA algorithm(shown at the top in the figure) takes as input a set of control-flow automata(CFA) representing the program, and a CPA, which is in most cases a Composite CPA. The interfaces correspond one-to-one to the formal framwork.[4]

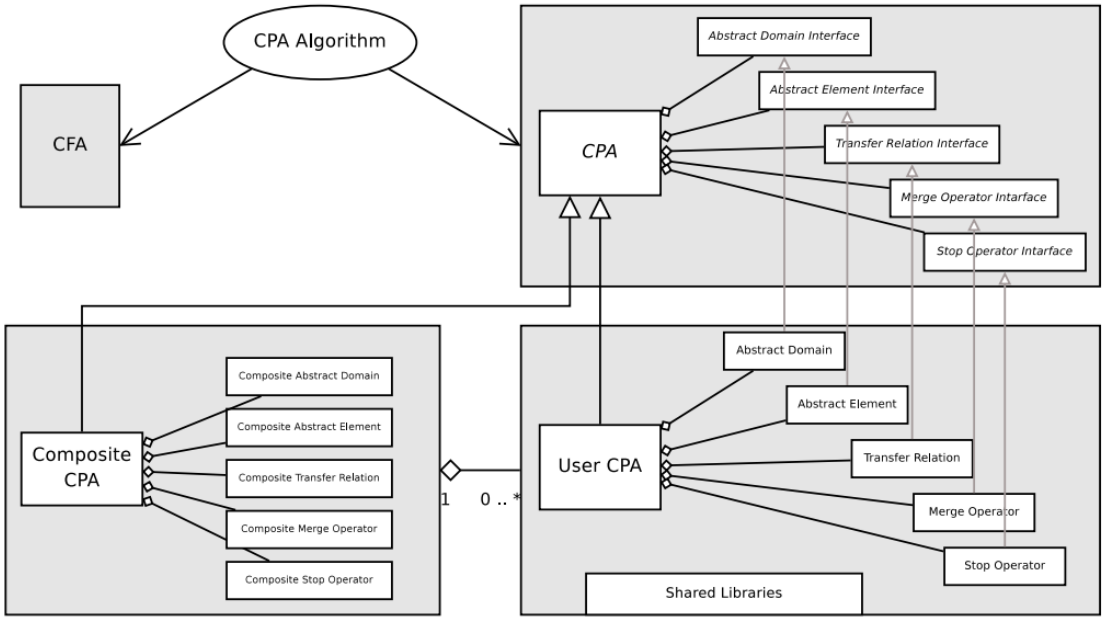

Figure 2: CPAchecker-Design for extension[4]

The elements in the gray box(top right) in Fig.2 represent the abstract interfaces of the CPA and the CPA operations. The two gray boxes at the bottom of the figure show two implementations of the CPA interfaces, one is a Composite CPA that can combine several other CPAs, and the other is a User CPA. For example, suppose we want to implement a CPA for shape analysis. We would provide an implementation for CPA, possible called ShapeCPA, and implementations for the operation interfaces on the right. If we want to experiment with several different merge operators, we would provide several different implementations of Merge Operator Interface that can be freely configured for use in various experiments.[4]

5、A Typical String Fault

In C program language, a string is typically represented by an array of char. A legal string represented by an array of char is always ended with '\0'. The length of a string is determined by the position of the first '\0'. The main target of the project is to detect the absence of '\0' in strings of the given programs. Some possible operations related to the string is following.

5.1 Initialization

A legal string represented by an array of char is always initialized in the following ways:

(1) char a[5] = "1234"; In this case, the size of the array is bigger than the length of the string.

(2) char c[5] = {'1','2','3','4','\0'};

(3) scanf("%s,&a); In this case, the IDE will detect whether input string is too long so that the array *a* has no place to store '\0'. Similiar to scanf, there are many other string input function, and they all guarantee the legality of the string.

(4) char a[5]; a[0] = '1'; a[1] = '2'; a[2] = '3'; a[3] = '\0';

5.2 Alteration

In the program, we may alter the string after initialization. For example, in the first case above, we may execute the instruction a[4] = '5'. Then the string a does not contain $'\backslash 0'$, so it becomes a illegal string.

5.3 Main difficulty

The main difficulty in the project is following:

(1) Visit struct fields. In C program, a struct can contain a char array, and maybe contains a struct which itself contains a char array, and so on. We need track each char array in each struct. So structs and struct paths need to be memorized. When a struct is defined, we need to add all char arrays in the fields of this struct to the list. When a char array is altered, we need to alter the value of the char arrays. This is the first difficulty in this project.

(2) Analyze the pointers. In the program, maybe a pointer points to a char array and erase the '\0', and we even need to consider the pointer of pointer, and pointer of struct which has the member which is a char array, and so on. When a pointer is defined, we need to add it and its pointing content to the HashMap. When a pointer is visited, we

need to search the pointer in the HashMap, find its pointing content, and do some alterations. This is the second difficulty in this project.

(3) Get the value of expressions in the index. The index of a char array may be a expression of the integer, so we need to implement the function of getting the value of the expression. This is the last difficulty in this project.

5.4 The plan of implementation

This project is based on the lab project：Tsmart. Other peers have implemented some structure of the struct fields. The first part of the project should be to extend the structure of the struct fields, in order to track the positions of '\0' in each char array.

The second part of the project should be to implement a CPA to detect the string fault, both when a string is declared and altered. This part of work is important, and the CPA is specially used in this project.

The third part of the project should be to get the value of expressions in the index. This part is based on LocationAnalysisCPA, which is defined and implemented by CPAchecker. We need to define the **strengthening** operator in the CPA defined in the second part, implement the communication between the CPA defined in the second part and LocationAnalysisCPA, and get the value of the expressions from the information provided by LocationAnalysisCPA.

The last part of the project should be to implement the pointer analysis.

Under a fair consideration, I suppose this plan of implementation is feasible, and I can implement the whole project in 12 weeks.

6、Time Arrangement of the Project

Winter vacation to week 2: Read papers and get familiar with the framework of CPA.

Week 2 to week 4: Detect the illegal initialization of string and visit struct fields.

Week 5 to week 7: Get the value of expressions in the index.

Week 8 to week 13: Analyze the pointers.

Week 14 to week 16: Write a paper.

7、Reference

[1]. Yingfei X.: Teaching slides of Program Analysis Technology, Peking University, 2015, http://sei.pku.edu.cn/~xiongyf04/SA/2015/main.htm

[2]. Beyer, D., Henzinger, T.A., and Theoduloz, G.: Configurable software verification:

concretizing the convergence of model checking and program analysis. In Proc.CAV, LNCS 4590, pages 504-518. Springer, 2007

[3]. Beyer, D., Henzinger, T.A., Theoduloz, G.: Lazy shape analysis. In: Ball, T., Jones, R.B.(eds.) CAV 2006. LNCS, vol. 4144, pp. 532–546. Springer, Heidelberg(2006)

[4]. Beyer, D., Keremoglu, M.E.: CPAchecker: A Tool for configurable software verification. Preprint arXiv:0902.0019.