清华大学学位论文 LaTeX 模板
使用示例文档 v6.0.2

(申请清华大学工学硕士学位论文)

培 养 单 位 ： 软件学院

学　　科 ： 软件工程

研 究 生 ： 李　　兀

指 导 教 师 ： 顾明教授

二〇二〇年四月

An Introduction to LaTeX Thesis Template of Tsinghua University v6.0.2

Thesis Submitted to

Tsinghua University

in partial fulfillment of the requirement

for the degree of

Master of Science

in

Software Engineering

by

Li Wu

Thesis Supervisor: Professor Gu Ming

April, 2020

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：

清华大学拥有在著作权法规定范围内学位论文的使用权，其中包括：（1）已获学位的研究生必须按学校规定提交学位论文，学校可以采用影印、缩印或其他复制手段保存研究生上交的学位论文；（2）为教学和科研目的，学校可以将公开的学位论文作为资料在图书馆、资料室等场所供校内师生阅读，或在校园网上供校内师生浏览部分内容。

本人保证遵守上述规定。

（保密的论文在解密后应遵守此规定）

作者签名：_____　　导师签名：_____

日　　期：_____　　日　　期：_____

摘 要

文章主要围绕 Range 那一套，写传统的静态分析方法

目的：整型缺陷分析：overflow, underflow, divide by zero

量化：Juliet、找一些工程

关键词：

Abstract

Key Words:

目 录

第 1 章 引言 1
1.1 研究背景与意义: 最少 1-2 页 1
1.2 国内外研究现状 1
1.2.1 抽象解释技术 1
1.2.2 数据流分析 2
1.2.3 符号化执行 3
1.2.4 整型缺陷检测技术与工具：工具最好放到后面 4
1.2.5 抽象解释技术 → 代码缺陷分析技术 4
1.2.6 常用抽象域与区间抽象域 → 整数相关，到这里加起来 3-4 页左右 ... 4
1.2.7 总结 4
1.3 研究难点与挑战 4
1.4 研究内容 5
1.5 研究方案 5
1.6 论文贡献 5
1.7 论文组织结构 5
第 2 章 基于区间算数的整数缺陷检测 6
2.1 引言 6
2.2 预备知识 7
2.2.1 控制流自动机 7
2.2.2 可配置程序分析框架 8
2.3 基于线性空间的抽象域设计 9
2.3.1 扩展的整数（RangeInteger）理论域 10
2.3.2 线性区间（Range）理论域 12
2.3.3 线性多区间（MultiRange）理论域 14
2.3.4 位敏感的线性区间（SignRange）理论域 17
2.3.5 线性区间状态（ApRangeState）理论域 21
2.3.6 精度优化 23
2.4 基于区间算数的整数缺陷检测方法 23
2.5 模块实现 24
2.6 实现方案 24

2.7 实验过程与结果 24
2.8 本章小结 24
第 3 章 基于环状区间的整型缺陷检测 → 这个不用具体实现，因为看不出来 25
3.1 基于环状区间的抽象域设计与实现 25
3.2 基于环状区间的整型缺陷检测方法 25
3.3 模块实现 25
3.4 实验过程与结果 25
3.5 本章小结 25
第 4 章 基于值流图的整型变量关系分析与缺陷检测 26
4.1 引言 26
4.2 研究动机 27
4.3 基于值流图的程序理解与需求确认方法 29
4.3.1 精确值流图的定义与构造 31
4.3.2 表达式分析算法 33
4.3.3 程序理解与需求确认 34
4.4 评估与结果 35
4.4.1 实验设计 35
4.4.2 客观评价 36
4.4.3 主观评价 37
4.5 后续工作与总结 38
第 5 章 总结与展望 40
5.1 工作总结 40
5.2 研究展望 40
参考文献 41
致 谢 43
声 明 44
个人简历、在学期间发表的学术论文与研究成果 45

主要符号对照表

HPC	高性能计算 (High Performance Computing)

第 1 章　引言

1.1　研究背景与意义: 最少 1-2 页

突出整型缺陷会造成什么重大问题，解决这个问题十分重要。

【这里需要重新找例子，更切题一点的】

1.2　国内外研究现状

主要围绕整型缺陷这边来延伸与展开。

1.2.1　抽象解释技术

论证程序正确性的方法最朴素的便是穷举程序所有可能的输入，并通过执行得到结果来判断其是否符合预期，如果运行结果符合预期，那么程序自然是正确的。然而，这种方法只是一种理论上可能的方法，在实际中，我们面对的程序输入的取值范围往往非常大，甚至无法穷举。以 C 语言的函数举例，若该函数有一个 int 型参数，由于 int 类型的表示范围是 $[-2147483648, +2147483647]$，那么单单一个 int 型参数就有 2^{32} 种取值，当输入的参数是字符串类型时，更是有无穷多中可能。因此，使用朴素的穷举来进行程序分析，其时间与空间代价在实际中是不可接受的。

当前常用的测试技术便是采用了上述思想，只不过测试技术所选择的输入是总输入空间的子集，通过边界条件分析等方法得到相对较少的输入空间。该方法的优点是大幅减少测试输入，但也带来了程序运行路径覆盖率低、需要人工参与测试输入样例的设计等问题。

相较于测试技术，抽象解释技术采用了不同的思路。抽象解释是一种对程序语义进行可靠抽象（近似）的通用理论[2]。与此同时，该理论为程序分析的设计与构建提供了一个通用的框架[3]。具体地，它是将程序语义进行不同程度的抽象，并将这种抽象及在其上的操作称为抽象域。通过将具体域中的值与抽象域中的值进行映射，从而将具体域中数量庞大甚至无穷大的取值域转化为抽象域中的有穷的取值域。并将具体域上的操作对应到抽象域上的操作，通过在抽象域上计算程序的抽象不动点来表达程序的抽象语义。

单纯通过构建在抽象域上的与操作如迁移函数来进行建模有时并不能保证在程序的迭代分析中抽象域能快速到达不动点以获得抽象语义。因此在抽象分析中

提供了加宽算子（widening），通过上近似理论来减少程序分析中的迭代次数，从而加速程序分析。由于上近似理论的可靠性，所有基于上近似抽象得到的性质，在源程序中必定成立。

抽象解释的核心问题是抽象域的设计，而如上所述，抽象解释是对程序语义的不同程度的抽象，这也就意味着抽象域并不唯一确定，针对特定问题可以设计使用特定抽象域以达到程序分析的效果。目前为止，已经出现了数十种面向不同性质的抽象域，其中，具有代表性的抽象域包括区间抽象域、八边形抽象域、多面体抽象域等数值抽象域[4]。另一方面，在开源领域出现了众多抽象域库，如 APRON[5]、ELINA[6]、PPL[7] 等。

抽象解释并不是一个已经研究成熟的课题，当下抽象解释仍然面临着很多挑战，主要包括两方面的内容：提高分析精度与拓展性。在提高分析精度方面，主要要解决的问题是基于加宽算子（widening）的不动点迭代运算的精度损失问题以及所设计的抽象域本身的表达能力具有局限性的问题。而在提高可拓展性方面，主要面临的问题是如何有效降低分析过程中抽象状态表示与计算的时空开销。

本章所涉及的整型缺陷检测技术基于抽象解释，通过设计抽象域与抽象域上的迁移函数，求得抽象域上的不动点以得到程序的抽象解释，从而进一步进行程序的缺陷分析。

1.2.2　数据流分析

通过抽象解释技术，我们能将具体域中的无穷状态问题转化为抽象域上的有穷的状态问题。而数据流分析则是在抽象解释的基础上，在控制流图上分析每个程序状态信息，从而得到每个静态程序点（语句）在运行时可能出现的状态。

数据流分析是抽象解释的一个特例，经典的数据流分析理论[8] 使用有限高度的格 $< L, \cap >$ 来表示所有可能的状态集合，其中 L 是值集，是抽象域的别名；$\cap$ 是交汇运算，是将两个状态合并成一个状态的操作。由于在程序语句中存在循环控制语句，即存在循环结构，则数据流可能从不同分支流向统一节点。因此，为了得到不同分支的信息并保证算法的可终止性，需要定义交汇运算，将不同分支状态融合到同一节点上。

数据流分析首先要确定数据流的方向，包括从 Entry 开始的正向分析和从 Exit 开始的逆向分析。数据流分析同时为每个程序语句构造一个单调的转移函数（又称变迁函数，transfer function），转移函数的输入是上一个程序点的状态信息以及程序语句，输出是程序语句执行后，下一个程序点应有的状态信息。

用伪代码写成的数据流分析算法[9] 如算法1所示：

Algorithm 1 数据流分析算法

```
1:  procedure RECURSION
2:      worklist = ∅;
3:      for i = 1 to N do
4:          initialize the value at node i;
5:          add i to the worklist;
6:      end for
7:      while worklist ≠ ∅ do
8:          remove a node i from the worklist;
9:          recompute the data flow fact at i;
10:         if new data flow fact is not equal to the old one at i then
11:             add each successor/predecessor of i to worklist uniquely;
12:         end if
13:     end while
14: end procedure
```

相比于通用的抽象分析，经典的数据流分析在使用迭代计算框架来计算程序语句的不动点时，由于单调性和格的有限高度，保证了数据流分析的收敛性。因此相较于经典抽象分析技术，加宽算子对于数据流分析并不是必须的。

由于数据流分析算法具有收敛速度快、扩展性高的特点，本章将采用数据流分析的算法作为工具研发的基本算法。

1.2.3　符号化执行

在上一小节的数据流分析中，我们讨论了数据流分析能够分析每个程序点上的状态信息，并能够保证算法的快速收敛。然而，算法的快速收敛的同时也混淆了不同路径上的信息，使得分析结果变得不明确。符号执行[10-11] 提供了一种系统遍历程序路径空间的手段，除了保持状态信息外，还同时维护路径上的约束条件。符号化执行通过以符号值来代替实际值并在遇到分支条件节点时通过调用 SMT 求解器[12] 验证分支路径是否可达来实现路径的遍历。因此，求解器的能力是制约符号执行技术的一个重要因素。

在多数情况下，静态分析方法的误报都是由于分析中不加判断的引入很多不可达路径，从而造成不合理判断。针对这种情况的误报，使用符号执行技术能够很有效的降低误报率。但是，符号执行也有它的弊端，那就是其对路径是敏感的，由此很容易产生路径爆炸问题，即，当一个程序具有 n 个条件判断语句结构时，理论

上就有可能存在多达 2^n 条路径！尤其是当程序中存在无穷路径的循环结构时，符号执行算法甚至可能是无法终止的。在实际应用中，面对这种循环语句的情况通常采取的策略是仅展开有限次，配合其他静态分析方法对循环进行分析处理。

目前，符号执行技术也面临着两方面的挑战，即提高可扩展性（scalability）与可行性（feasibility）。可扩展性指如何在有限的资源条件（内存、时间条件等因素）下提高符号执行的效率，从而更快完成分析。可行性指面对不同类型的分析目标如何应用符号执行技术，以及，如何权衡可靠性与准确性。在可扩展性方面，现有的研究主要围绕两种思路进行，其一是在具体目标下提供高效的搜索目标，其二是从约束输入范围着手，削减并合并路径从而达到减少程序路径空间的目的。

综上，由于符号化执行方法具有路径敏感、精度高的特点，将采用符号执行技术提高分析精度。

1.2.4　整型缺陷检测技术与工具：工具最好放到后面

介绍目前识别整型缺陷的常用技术与工具，阐述它们的优缺点。

【这里可能需要额外做一些调研，然后列举在这上面】

1.2.5　抽象解释技术 → 代码缺陷分析技术

1.2.6　常用抽象域与区间抽象域 → 整数相关，到这里加起来 3-4 页左右

具体展开抽象域的研究，首先介绍抽象域在静态分析方法中的角色与作用，随后剖析各个抽象域的优缺点，重点介绍区间抽象域，它能解决什么问题，为什么它比较好。

【将上面的部分分一点儿到下面来】

1.2.7　总结

接上，阐述我们为什么要做区间抽象域，期望能达到什么样的一个目标，解决了传统区间抽象域的哪些痛点。

1.3　研究难点与挑战

难点大致在抽象域的设计方面：变迁规则、抽象方法（近似手段、合并操作等）

1.4　研究内容

这里搞一张图，到时候用这个说

1. 理论研究【暂未明确】
   (a) 基于程序解释的符号敏感的区间抽象域分析方法/基于线性空间的整型缺陷检测方法
   (b)【待讨论】基于二进制串的符号敏感的区间抽象域分析方法/基于环状区间的整型缺陷检测方法
   (c) 基于区间分析的数值导向型缺陷分析组合方法
2. 工具研发

1.5　研究方案

这里同样补一张图，对应于上面的研究内容。不用像开题报告那样分小章节说，直接一段话即可。

1.6　论文贡献

最后补上。

1.7　论文组织结构

最后补上。

第 2 章　基于区间算数的整数缺陷检测

C 语言是一种面向过程的通用程序设计语言，由于其具有处理低级存储器、产生更少量的机器码以及不需要任何运行环境便可运行的特点，因而适用于操作系统等底层软件和需要高效运行的应用软件的编写。但也因为 C 语言的内存资源完全交由用户管理与控制，如果在程序设计与编写过程中考虑不全面或粗心大意，将会产生安全隐患。

其中，相当一部分的缺陷是由非法或错误的整型计算造成的：如整型溢出、除零异常、内存泄漏、缓冲区溢出等。在实际应用中，静态分析技术凭借其无需运行程序且分析路径的覆盖率高的特点，广泛应用于大规模软件的缺陷检测。通常，程序分析的精度与效率是不可兼得的：分析精度的提高往往伴随着效率的下降。平衡静态分析的精度与资源消耗，在不占用太多的资源下获得较好的分析精度一直是静态分析领域所重点关注的问题之一。

本章提出基于区间算数的整数缺陷检测技术，借助组合静态分析框架，在程序局部进行高精度的区间分析，对整体使用高效率的算法对各个局部分析结果进行组合，从而实现高效的 C 语言整数缺陷检测。本章的工作对提升现有工具的整数缺陷分析的精度和效率具有重要意义。

2.1　引言

若要判定程序中整型变量所参与的计算是否可能产生整型溢出、除零异常及缓冲区溢出等缺陷，传统的解决方案是使用抽象解释将程序中的整型变量上近似抽象为 1 个整数区间。结合数据流分析，得到整型变量在整数区间抽象域上的近似值，并以此判定整型变量的操作是否安全。

尽管该方法能够在简单逻辑代码上得到较好的分析效果，但由于抽象域的表示能力有限，在程序的逻辑分支复杂的情况下，因状态合并操作所带来的精度损失较大，通常经过数次操作即到达抽象域的上界。

考虑图2.1所示代码：函数 A 作为程序的入口，通过用户输入得到整型数值 i，并根据 i 的大小使用不同的逻辑调用函数 B 并处理函数返回值。这里为讨论方便起见，规定 int 可表示所有整数值。若使用传统的基于整数区间抽象的数据流分析算法进行程序分析，易知函数 B 的返回值范围是 $[-\infty, 3]$，则函数 A 中第 4 行 x 的取值范围是 $[-\infty, -2]$，第 6 行 x 的取值范围是 $[2, +\infty]$。在第 8 行，由于状态合并，x 的取值范围是 $[-\infty, +\infty]$，这将导致目标属性不成立。但实际上 x 的取值范围不

```
1 void funcA(int i) {
2   int x;
3   if (i > 0) {
4     x = funcB(i) - 5;
5   } else {
6     x = 5 - funcB(i);
7   }
8   assert(x != 0);
9 }
```

```
1 int funcB(int i) {
2   int x;
3   if (x > 3) {
4     x = 3 - i;
5   } else {
6     x = i;
7   }
8   return x;
9 }
```

图 2.1　基于区间算数的缺陷检测方法举例

包含 0，目标属性是安全的。

为了解决上述问题，进一步提高程序分析的精度，本章提出了基于区间算数的整数缺陷检测技术，并在基于 CPAchecker[1] 的可配置的组合静态分析工具 Tsmart 上进行了实现。经过实验，XXX。

2.2　预备知识

2.2.1　控制流自动机

控制流自动机（Control-flow Automaton，CFA）是命令式程序的一种语义等价表示方法，它是一个有向图：

定义 2.1：　CFA 可表示为有向图 $G = (N, E)$：N 是节点的集合，节点 $n \in N$ 表示程序的状态。$E \subseteq N \times Ops \times N$ 是边的集合，有向边 $e = (n, op, n') \in E$ 表示从某个程序状态到下一个程序状态的过程，其中，Ops 是状态之间所执行的指令。一般地，我们用 $pred(e)$ 表示边 e 的前驱节点 n，用 $succ(e)$ 表示边 e 的后继节点 n'，用 $act(e)$ 表示边 e 所执行的动作 op。

在本文中，若无特殊说明，CFA 是 LLVM-IR 语言上的等价表示，有向边 e 根据其表示的指令的不同，可分为如下几类：

- 空白边（BlankEdge）：表示没有执行动作的边。即 $act(e) = \varepsilon$；
- 假设边（AssumeEdge）：描述了假设条件是否成立。约定 $act(e) = (\%cmp, truth)$，其中 $\%cmp$ 为条件变量，它对应于 LLVM-IR 中 i1 类型的变量，表示假设条件。$truth \in \{0, 1\}$ 为条件取值，表示当前边上假设条件变量 $\%cmp$ 的具体取

值。一般地，我们用 $cond(e)$ 表示假设边的条件变量 $\%cmp$，用 $truth(e)$ 表示条件变量的取值 $truth$。

- Phi 边（PhiEdge）：描述了 LLVM-IR 中的 phi 指令。约定 $act(e) = (\%phi, index)$，其中 $\%phi$ 对应于 phi 指令，$index$ 指示 phi 边上的第几个操作数作为当前指令的返回结果。
- 指令边（InstructionEdge）：是 CFA 中最常见的边，描述了当前边执行了一条 LLVM-IR 指令。约定 $act(e) = (\%inst)$，其中 $\%inst$ 为所执行的指令。

为了便于描述，定义函数 $type$ 用于获取边的类型：

$$type(e) := \begin{cases} blank & e \in BlankEdge \\ assume & e \in AssumeEdge \\ phi & e \in PhiEdge \\ inst & e \in InstructionEdge \end{cases} \tag{2-1}$$

2.2.2　可配置程序分析框架

可配置程序分析（Configurable Program Analysis，CPA）是一种通用的程序分析框架，通过设计并配置相关参数，实现在同一框架下完成多种不同的静态分析任务。通常，CPA 框架将程序源代码转化为语义等价的控制流自动机并在 CFA 上多次应用静态分析算法实现程序分析。其中，CFA 是源程序的等价表示，可描述为有向图。其节点表示指令位置、边表示控制流操作，如变量声明、运算、赋值、函数调用等。在分析时，我们常将内存地址抽象为内存位置，如访问路径[13] 等。

CPA 算法的一次分析可表示为四元组 $\mathbb{D} = (D, \rightsquigarrow, merge, stop)$。其中，$D$ 为抽象域；$\rightsquigarrow$ 为转移关系，规定了在不同的控制流操作下，给定的抽象状态如何转移到新的抽象状态；$merge$ 为状态合并算子；$stop$ 为状态覆盖算子。

算法2描述了 CPA 的算法流程：算法以分析 $\mathbb{D}$、CFA 图 G 和初始状态 $e_0 \in E$ 作为输入，通过维护工作队列 $waitlist$ 和可达状态集 R 并根据转移关系 $\rightsquigarrow$ 来计算当前状态 e 的后继状态。工作队列和可达集的维护算法如算法3所示，对每个后继状态 e' 使用 $merge$ 算子来更新所有可达状态，如果可达状态 e'' 被更新，则将该状态加入到等待队列 $waitlist$ 中以更新其后继状态。如果更新后的可达状态无法覆盖 e'，则将该状态分别加入 $waitlist$ 和可达集中。

Algorithm 2 CPA 算法

Require: $\mathbb{D} = (D, \rightsquigarrow, merge, stop)$，CFA 图 G，初始状态 $e_0 \in E$

Ensure: 可达状态集 R

$waitlist \leftarrow \{e_0\}, R \leftarrow \{e_0\}$
while $waitlist \neq \varnothing$ **do**
    取出 $waitlist$ 的首元素 e;
    for all e' 满足 $e \rightsquigarrow e'$ **do**
        UpdateRW($e', R, waitlist, merge, stop$);
    end for
end while

Algorithm 3 UpdateRW 算法

function UpdateRW($e', R, waitlist, merge, stop$)
    for all $e' \in R$ **do**
        $e_{new} \leftarrow merge(e, e')$;
        if $e_{new} \neq e'$ **then**
            $waitlist \leftarrow (waitlist \cup \{e_{new}\}) \setminus \{e'\}$;
            $R \leftarrow (R \cup \{e_{new}\}) \setminus \{e'\}$;
        end if
    end for
    if $\neg stop(e, R)$ **then**
        $waitlist \leftarrow (waitlist \cup \{e\})$;
        $R \leftarrow R \cup \{e\}$;
    end if
end function

2.3　基于线性空间的抽象域设计

本节介绍基于线性空间的抽象域，抽象域的组成结构如图2.2所示。ApRangeState 为程序变量在线性空间的抽象表示，它包含变量的访问路径、取值范围、符号等信息。

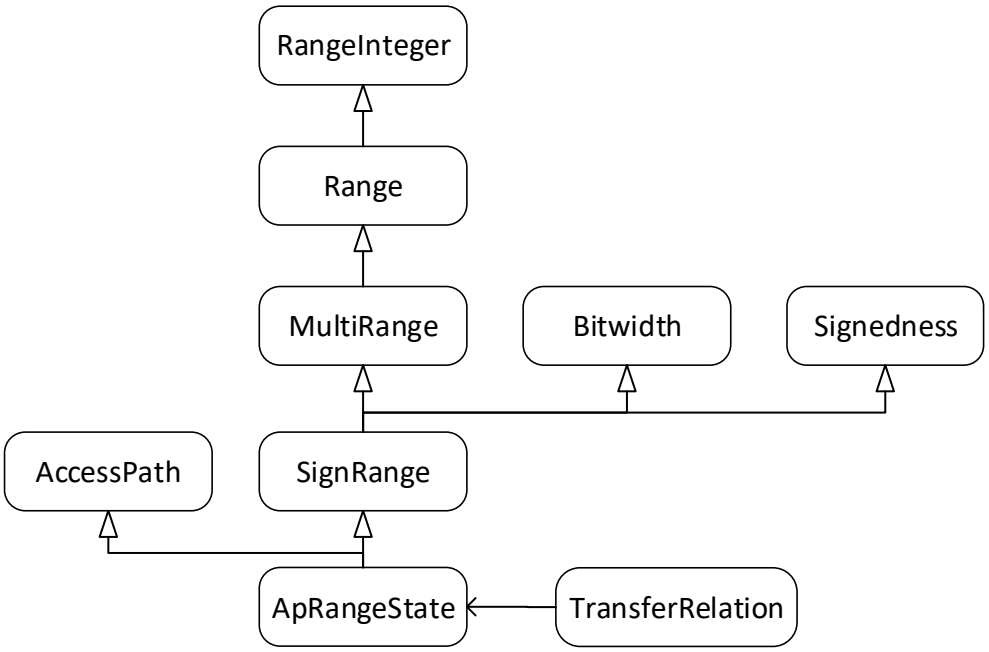

图 2.2　抽象域的组成

2.3.1　扩展的整数（RangeInteger）理论域

整型变量的可能取值是一个整数的集合，在这里我们使用扩展的整数 RangeInteger 来表示一个具体的整数，它是整数域 $\mathbf{Z}$ 的一个拓展。

定义 2.2：　记 $D_I := \mathbf{Z} \cup \{+\infty, -\infty, arb, nan\}$ 为扩展的整数理论域。

- D_I 上的最大元（$\top_I$）为 arb，它表示任意值；最小元（$\bot_I$）为 nan，它表示非数；
- D_I 上的偏序关系（$\leqslant_I$）为 $\mathbf{Z}$ 上的自然扩展，对于元素 $-\infty$ 与 $+\infty$，规定 $-\infty < +\infty$，且对任意 $x \in \mathbf{Z}$，有 $x < +\infty$ 以及 $-\infty < x$。特殊地，$-\infty$ 与 $-\infty$、$+\infty$ 与 $+\infty$ 无法比较，$\bot_I, \top_I$ 与其他任意元素无法比较。
- D_I 上的上确界操作（$\sqcup_I$）定义为：

$$a \sqcup_I b := \begin{cases} \top_I & a \neq \bot_I \wedge b \neq \bot_I \\ a & b = \bot_I \\ b & a = \bot_I \\ \bot_I & a = \bot_I \wedge b = \bot_I \end{cases}$$

对于元素 $-\infty$ 或 $+\infty$ 参与的运算，其规则如表2.1所示。其中，函数 $signum$ 为

取符号数，具体定义如下：

$$signum(x) := \begin{cases} 1 & (x \in \mathbf{Z} \wedge x > 0) \vee x = +\infty \\ 0 & x = 0 \\ -1 & (x \in \mathbf{Z} \wedge x < 0) \vee x = -\infty \end{cases} \tag{2-2}$$

特别的，元素 $+\infty$ 与 $-\infty$ 所参与的某些运算是未定义的。这样的运算在表中记录为 nan，如 $(+\infty) +_I (-\infty), (+\infty) \div_I (+\infty)$ 等。

表 2.1　RangeInteger 运算规则

运算	符号	运算规则
negate	neg_I	$neg_I(x) := \begin{cases} -x & x \neq \pm\infty \\ -\infty & x = +\infty \\ +\infty & x = -\infty \end{cases}$
add	$+_I$	$x +_I y := \begin{cases} x + y & x, y \neq \pm\infty \\ x & x = \pm\infty \wedge y \neq \pm\infty \\ y & x \neq \pm\infty \wedge y = \pm\infty \\ x & x, y = \pm\infty \wedge signum(x) = signum(y) \\ nan & \text{other cases} \end{cases}$
subtract	$-_I$	$x -_I y := \begin{cases} x - y & x, y \neq \pm\infty \\ x & x = \pm\infty \wedge y \neq \pm\infty \\ neg(y) & x \neq \pm\infty \wedge y = \pm\infty \\ x & x, y = \pm\infty \wedge signum(x) \neq signum(y) \\ nan & \text{other cases} \end{cases}$
multiply	$\times_I$	$x \times_I y := \begin{cases} x \times y & x, y \neq \pm\infty \\ +\infty & signum(x) \times signum(y) > 0 \\ -\infty & signum(x) \times signum(y) < 0 \\ nan & \text{other cases} \end{cases}$
divide ($y \neq 0$)	$\div_I$	$x \div_I y := \begin{cases} x \div y & x, y \neq \pm\infty \\ 0 & x \neq \pm\infty \wedge y = \pm\infty \\ +\infty \times_I signum(x \times_I y) & x = \pm\infty \wedge y \neq \pm\infty \\ nan & \text{other cases} \end{cases}$

续下页

续表 2.1　RangeInteger 运算规则

运算	符号	运算规则
modular ($y>0$)	mod_I	$x \bmod_I y := \begin{cases} x \bmod y & x, y \neq \pm\infty \\ x & x \geqslant 0 \wedge y = +\infty \\ y & x < 0 \wedge y = +\infty \\ nan & \text{other cases} \end{cases}$
and	and_I	$x \cdot_{op_I} y := \begin{cases} x \cdot_{op} y & x, y \neq \pm\infty \\ nan & \text{other cases} \end{cases}$
or	or_I	
xor	xor_I	
shl	$<<_I$	
shr	$>>_I$	

2.3.2　线性区间（Range）理论域

线性区间 Range 一个整数区间，表示整型变量的可能取值范围，它是由两个整数 x, y 构成的二元组。这里，x, y 为扩展的整数域 RangeInteger 上的元素，其定义与偏序关系定义如下。

定义 2.3：线性区间理论域 $D_R := \{[x, y] | x \in \mathbf{I}, y \in \mathbf{I}, x \leqslant y\} \cup \{\varnothing\}$ 是一个半格：

- D_R 上的最大元（$\top_R$）是 $[-\infty, +\infty]$，最小元（$\bot_R$）是空集 $\varnothing$；
- D_R 上的偏序关系（$\leqslant_R$）定义为：

$$[x_1, y_1] \leqslant_R [x_2, y_2] \iff x_2 \leqslant x_1 \wedge y_1 \leqslant y_2;$$

- D_R 上的上确界操作（$\sqcup_R$）定义为：

$$[x_1, y_1] \sqcup_R [x_2, y_2] := [\min(x_1, x_2), \max(y_1, y_2)],$$

此处 max 和 min 为 D_I 上的自然扩展。

我们在 Range 上定义一些基础的运算操作，其运算规则如表2.2所示。值得注意的是，最小元 $\bot_R$ 与任何元素的计算结果均为 $\bot_R$，为表示方便，在表中默认操作数均不为 $\bot_R$。

表 2.2　Range 运算规则

运算	符号	运算规则
negate	neg_R	$neg_R([x, y]) := [[neg_I(x), neg_I(y)]]_R$
add	$+_R$	$[x_1, y_1] +_R [x_2, y_2] := [[x_1 +_I x_2, y_1 +_I y_2]]_R$
subtract	$-_R$	$[x_1, y_1] -_R [x_2, y_2] := [[x_1 -_I y_2, y_1 -_I x_2]]_R$
multiply	$\times_R$	$[x_1, y_1] \times_R [x_2, y_2] := [\min(vals), \max(vals)], vals = \{x_1, y_1\} \times_{\times_I} \{x_2, y_2\}$

续下页

续表 2.2　Range 运算规则

运算	符号	运算规则
divide ($op_2 \neq 0$)	$\div_R$	$[x_1, y_1] \div_R [x_2, y_2] :=$ $\begin{cases} [\min(vals), \max(vals)] & 0 \notin [x_2, y_2],\ vals = \{x_1, y_1\} \times_{\div_I} \{x_2, y_2\} \\ [x_1, y_1] \div_R [1, y_2] & x_2 = 0 \\ [x_1, y_1] \div_R [x_2, -1] & y_2 = 0 \\ [neg(upper), upper] & x_2 < 0 < y_2,\ upper = \max(\lvert x_1 \rvert, \lvert y_1 \rvert) \end{cases}$
modular ($op_2 \neq 0$)	mod_R	$[x_1, y_1] \mod_R [x_2, y_2] :=$ $\begin{cases} op_1 -_R op_2 \times_R times_1 & x_2 > 0,\ times_1 = op_1 \div_R op_2 +_R [1, 1] \\ op_1 -_R op_2 \times_R times_2 & y_2 < 0,\ times_2 = op_1 \div_R op_2 -_R [1, 1] \end{cases}$
and	and_R	$[x_1, y_1]\ and_R\ [x_2, y_2] :=$ $\begin{cases} x_1\ and_I\ y_1 & x_1 = y_1 \wedge x_2 = y_2 \\ [0, y_1] & x_1 = y_1 \geqslant 0 \\ [0, y_2] & x_2 = y_2 \geqslant 0 \\ [0, +\infty] & x_1 \geqslant 0 \vee x_2 \geqslant 0 \\ [-\infty, -1] & y_1 < 0 \wedge y_2 < 0 \\ [-\infty, +\infty] & \text{other cases} \end{cases}$
or	or_R	$[x_1, y_1]\ or_R\ [x_2, y_2] :=$ $\begin{cases} x_1\ or_I\ y_1 & x_1 = y_1 \wedge x_2 = y_2 \\ [-1, -1] & x_1 = y_1 = -1 \vee x_2 = y_2 = -1 \\ [\max(x_1, x_2), +\infty] & x_1 \geqslant 0 \wedge x_2 \geqslant 0 \\ [-\infty, -1] & y_1 < 0 \vee y_2 < 0 \\ [-\infty, +\infty] & \text{other cases} \end{cases}$
xor	xor_R	$[x_1, y_1]\ xor_R\ [x_2, y_2] :=$ $\begin{cases} x_1\ xor_I\ y_1 & x_1 = y_1 \wedge x_2 = y_2 \\ [x_1, y_1] & x_2 = y_2 = 0 \\ [x_2, y_2] & x_1 = y_1 = 0 \\ [-\infty, +\infty] & x_1 \times_I x_2 < 0 \vee x_2 \times_I x_2 < 0 \\ [0, +\infty] & (x_1 \geqslant 0 \wedge x_2 \geqslant 0) \vee (y_1 < 0 \wedge y_2 < 0) \\ [-\infty, -1] & \text{other cases} \end{cases}$

续下页

续表 2.2　Range 运算规则

运算	符号	运算规则
shl	$<<_R$	$[x_1,y_1] <<_R [x_2,y_2] :=$ $\begin{cases}\varnothing & y_2 < 0 \\ [x_1,y_1] <<_R [0,y_2] & 0 \in [x_2,y_2] \\ [\min(vals),\max(vals)] & \text{other cases}, vals = \{x_1,y_1\} \times_{<<_I} \{x_2,y_2\}\end{cases}$
shr	$>>_R$	$[x_1,y_1] >>_R [x_2,y_2] :=$ $\begin{cases}\varnothing & y_2 < 0 \\ [x_1,y_1] >>_R [0,y_2] & 0 \in [x_2,y_2] \\ [\min(vals),\max(vals)] & \text{other cases}, vals = \{x_1,y_1\} \times_{>>_I} \{x_2,y_2\}\end{cases}$

在表格中，我们定义了构造函数 $[[\cdot]]_R$，利用它可以方便的构造 Range：

$$[[x,y]]_R := \begin{cases}[x,y] & (x,y \text{ is comparable}) \wedge x \leqslant y \\ [y,x] & (x,y \text{ is comparable}) \wedge x > y \\ \varnothing & \text{other cases}\end{cases} \tag{2-3}$$

同时，为了进一步简化书写，我们定义了集合上的操作 $\times_{\cdot_{op}}$。对集合 $setA$ 和 $setB$，有：

$$setA \times_{\cdot_{op}} setB = \{e_i \cdot_{op} e_j \mid e_i \in setA, e_j \in setB\} \tag{2-4}$$

Range 的比较规则定义如下：

$$[x_1,y_1] < [x_2,y_2] \iff y_1 < x_2 \tag{2-5}$$

$$[x_1,y_1] > [x_2,y_2] \iff x_1 > y_2 \tag{2-6}$$

2.3.3　线性多区间（MultiRange）理论域

通过分析图2.1所示的代码案例，易知在整型变量分析时，使用单独的线性区间 Range 会造成一定程度的精度损失，这时需要提供一个更为精确的理论域来对整型变量的取值范围做抽象。

本小节介绍线性多区间 MultiRange，它是由 0 到 L_{MR} 个 Range 构成的有序的区间列表，通过使用多个 Range 区间，MultiRange 能够提供更贴近实际域的变量取值区间表示。当 $L_{MR} \to \infty$ 时，MultiRange 能够精确表示整型变量的每一个可能取值，其表示能力和 RangeInteger 相同；当 $L_{MR} = 1$ 时，MultiRange 退化为 Range。

在实际应用中，由于 MultiRange 可以根据需要通过改变 L_{MR} 值的大小灵活改变理论域的表示精度，从而可以十分灵活的针对不同情况，优化程序分析的分析效率与分析精度。

定义 2.4：线性多区间理论域 $D_{MR} := \{[range_1, \cdots, range_n] \mid 0 \leqslant n \leqslant L_{MR}, range_1 < \cdots < range_n, range_i \in D_R\}$ 是一个半格：

- D_{MR} 上的最大元（$\top_{MR}$）是 $[\top_R]$，最小元（$\bot_{MR}$）是 $[\,]$；
- D_{MR} 上的偏序关系（$\leqslant_{MR}$）定义为：

$$[rangeA_1, \cdots, rangeA_m] \leqslant_{MR} [rangeB_1, \cdots, rangeB_n] \iff$$
$$\forall\, rangeA_i,\ \exists\, rangeB_j \text{ that } rangeA_i \leqslant_R rangeB_j.\ (0 \leqslant i \leqslant m, 0 \leqslant j \leqslant n);$$

- D_{MR} 上的上确界操作（$\sqcup_{MR}$）定义为：

$$[rangeA_1, \cdots, rangeA_m] \sqcup_{MR} [rangeB_1, \cdots, rangeB_n] :=$$
$$[[rangeA_1, \cdots, rangeA_m, rangeB_1, \cdots, rangeB_n]]_{MR}.\ (0 \leqslant i \leqslant m, 0 \leqslant j \leqslant n)。$$

上述的定义中使用了 MultiRange 的构造函数 $[[\cdot]]_{MR}$，其函数逻辑如算法4所示。该函数接受一个由 Range 组成的集合 $ranges$ 或直接给出待组合的 Range 列表 $range_1, range_2, \cdots, range_n$。首先根据每个区间的左值和右值进行排序，得到有序的 Range 列表 $ranges$。并将 $ranges$ 中所有相互包含或相邻的区间进行合并，得到有序的且不互相包含、连接的区间列表 $nRanges$。为了保证得到的 MultiRange 中 Range 的个数小于 L_{MR}，在算法结束前不断合并前后相邻的且距离最小的 Range 直至其个数满足要求。

Algorithm 4 MultiRange 的构造器 $[[\cdot]]_{MR}$

Require: $ranges = \{range_1, \cdots, range_n\}, L_{MR}$

Ensure: $[nRange_1, \cdots, nRange_m].\ (0 \leqslant m \leqslant L_{MR})$

```
1:  sort ranges by range_i.x then by range_i.y
2:  nRanges ← ∅
3:  [low, high] ← peek(ranges)
4:  for all range_i in ranges do
5:      if range_i.x ⩽ high +_I 1 then
6:          high ← max(range_i.y, high)
7:      else
8:          nRanges ← nRanges ∪ [low, high]
9:          [low, high] ← range_i
10:     end if
11: end for
```

12: **while** $size(nRanges) > L_{MR}$ **do**
13: 　pick range pair $(nRange_i, nRange_{i+1})$ in $nRanges$ whose distance is shortest
14: 　$nRanges \leftarrow (nRanges \setminus \{nRange_i, nRange_{i+1}\}) \cup (nRange_i \sqcup_R nRange_{i+1})$
15: 　sort $nRanges$ by $nRange_i.x$ then by $nRange_i.y$
16: **end while**
17: **return** $toList(nRanges)$

我们同样在 MultiRange 上定义一些基础运算操作，如表2.3所示。为方便起见，记 $ranges = [range_i, \cdots, range_n]$ 为一个 MultiRange，它是一个有序 Range 列表。列表 $ranges$ 沿用集合上的符号 $\times_{\cdot_{op}}$，即 $rangesA \times_{\cdot_{op}} rangesB = toList(set_{\cdot_{op}})$，其中 $set_{\cdot_{op}} = \{rangeA_i \cdot_{op} rangeB_j \mid rangeA_i \in rangesA, rangesB_j \in rangesB, 0 \leqslant i \leqslant m, 0 \leqslant j \leqslant n\}$。

表 2.3　MultiRange 运算规则

运算	符号	运算规则
negate	neg_{MR}	$neg_{MR}(ranges) := [[neg_R(range_1), \cdots, neg_R(range_n)]]_{MR}$
add	$+_{MR}$	$rangesA +_{MR} rangesB := [[rangesA \times_{+_R} rangesB]]_{MR}$
subtract	$-_{MR}$	$rangesA -_{MR} rangesB := [[rangesA \times_{-_R} rangesB]]_{MR}$
multiply	$\times_{MR}$	$rangesA \times_{MR} rangesB := [[rangesA \times_{\times_R} rangesB]]_{MR}$
divide ($op_2 \neq 0$)	$\div_{MR}$	$rangesA \div_{MR} rangesB := [[rangesA \times_{\div_R} rangesB]]_{MR}$
modular ($op_2 \neq 0$)	mod_{MR}	$rangesA \mod_{MR} rangesB := [[rangesA \times_{\mathrm{mod}_R} rangesB]]_{MR}$
and	and_{MR}	$rangesA\, and_{MR}\, rangesB := [[rangesA \times_{and_R} rangesB]]_{MR}$
or	or_{MR}	$rangesA\, or_{MR}\, rangesB := [[rangesA \times_{or_R} rangesB]]_{MR}$
xor	xor_{MR}	$rangesA\, xor_{MR}\, rangesB := [[rangesA \times_{xor_R} rangesB]]_{MR}$
shl	$<<_{MR}$	$rangesA <<_{MR} rangesB := [[rangesA \times_{<<_R} rangesB]]_{MR}$
shr	$>>_{MR}$	$rangesA >>_{MR} rangesB := [[rangesA \times_{>>_R} rangesB]]_{MR}$

由于 MultiRange 是由 Range 组成的有序列表，通过观察表2.3易知，其计算规则是 Range 计算规则的简单拓展。考虑两个只含有单独区间的 MultiRange：$rangesA = [[1,3]]$，$rangesB = [[2,3]]$。$rangesA \times_{MR} rangesB$ 按照上述计算规则的计算结果为 $[[1,3] \times_R [2,3]] = [[2,9]]$，而实际上 $rangesA \times_{MR} rangesB$ 的取值不可能为 $5, 7, 8$，计算存在精度丢失。为了解决上述问题，在后续章节提出优化方案，可进一步提升 MultiRange 的计算精度。

MultiRange 的比较规则定义如下：

$$[rangeA_1, \cdots, rangeA_m] < [rangeB_1, \cdots, rangeB_n] \iff rangeA_m < rangeB_1 \quad (2\text{-}7)$$

$$[rangeA_1, \cdots, rangeA_m] > [rangeB_1, \cdots, rangeB_n] \iff rangeA_1 > rangeB_m \quad (2\text{-}8)$$

为了后续便利，在 SignRange 上定义如下函数：

- $kill(Range)$ 用于删除 SignRange 的表示集合中参数 Range 对应的部分。例如：对于 $signRange = ([[0, 255]], 8, Unsigned)$，有 $signRange.kill([10, 100]) = ([[0, 9], [101, 255]], 8, Unsigned)$。

2.3.4　位敏感的线性区间（SignRange）理论域

在前面介绍的 RangeInteger、Range 与 MultiRange 抽象表示均未考虑整型变量在计算机上的存储格式，即对变量在计算机中所占用字节的大小是不敏感（Bitwidth Sensitive）的。当程序分析需要考虑变量的字节大小时，这类抽象表示往往不能提供整型变量实际所能表示的整数范围。

位敏感的线性区间 SignRange = (MultiRange, Bitwidth, Signedness) 是一个三元组。其中，MultiRange 是上小节所述的线性多区间；Bitwidth 表征了变量在计算机中所占用的位宽；Signedness 表征了存储于计算机中整型变量在程序中是以有符号还是无符号的形式解析的。

定义 2.5：符号性（Signedness）理论域 $D_{sgn} := \{Signed, Unsigned, Unknown\}$ 是一个半格：

- D_{sgn} 上的最大元（$\top_{sgn}$）是 $Unsigned$，最小元（$\bot_{sgn}$）是 $Unknown$；
- D_{sgn} 上的偏序关系（$\leqslant_{sgn}$）定义为：

$$Unknown \leqslant_{sgn} Signed \leqslant_{sgn} Unknown$$

- D_{sgn} 上的上确界操作（$\sqcup_{sgn}$）定义为：

$$signedness_1 \sqcup_{sgn} signedness_2 := \begin{cases} signedness_2 & signedness_1 \leqslant_{sgn} signedness_2 \\ signedness_1 & signedness_2 \leqslant_{sgn} signedness_1 \end{cases}$$

引入符号性理论域意味着一个整形变量取值范围会随着符号性的不同而映射为不同的 SignRange，尽管其在计算机内的存储内容是相同的（解释方式不同）。由于程序的这一特点，定义函数 $|_{to}(signedness)$ 用于转化同一整数取值范围的不同 SignRange 表示。方便起见，可将 $Signed, Unsigned, Unknown$ 分别记为 S, U, N。在此之前，先定义位敏感的线性区间 SignRange 的理论域：

定义 2.6：　位敏感的线性区间理论域 $D_{SR} := \{(ranges, bitwidth, signedness) \mid ranges \in D_{MR}, bitwidth \in \mathbf{Z}^+, signedness \in \{Signed, Unsigned, Unknown\}\}$ 是一个半格：

- D_{SR} 上的最大元（$\top_{SR}$）是 $(\top_{MR}, \top_{Z^+}, signedness)$，最小元（$\bot_{SR}$）是 $(\bot_{MR}, \bot_{\mathbf{Z}^+}, signedness)$；
- D_{SR} 上的偏序关系（$\leqslant_{SR}$）定义为：

$$(rangesA, bitwidthA, signednessA) \leqslant_{SR} (rangesB, bitwidthB, signednessB) \iff (rangesA|_{to}(sgn(rangesB)) \leqslant_{MR} rangesB) \wedge (bitwidthA \leqslant bitwidthB);$$

- D_{SR} 上的上确界操作（$\sqcup_{SR}$）定义为：

$$(rangesA, bitwidthA, signednessA) \sqcup_{SR} (rangesB, bitwidthB, signednessB) := (rangesA, bitwidth, signedness),$$

$$\text{其中，} \begin{cases} signedness := \max(signednessA, signednessB) \\ bitwidth := \max(bitwidthA, bitwidthB) \\ ranges := rangesA|_{to}(signedness) \sqcup_{MR} rangesB|_{to}(signedness) \end{cases} \text{。}$$

上述定义中，函数 $sgn :: SignRange$ 为取 SignRange 的 Signedness 值。类似的，有 $width :: SignRange$ 用于取 SignRange 的 bitwidth 值；$range :: SignRange$ 用于取 SignRange 的 MultiRange 值。

值得注意的是，$getRanges(\top_{SR}) := \top_{MR}$。这意味着在 SignRange 中 MultiRange 的实际可取值范围比计算机表示范围大，这一特性将用于程序分析判定整形变量是否产生整形溢出。为了表示在计算机上固定位宽的整型变量的实际取值范围，定义函数 $top(SignRange)$ 和函数 $top(bitwidth, signedness)$ 用于计算在特定位宽下的整型变量以特定符号性解释的取值范围：

$$top(SignRange) = top(bitwidth, signedness) := \begin{cases} [0, 2^{bitwidth} - 1]_R & signedness = Unsigned \\ [-2^{bitwidth-1}, 2^{bitwidth-1} - 1]_R & signedness \in \{Signed, Unknown\} \end{cases} \tag{2-9}$$

函数 $|_{to}$ 的转换规则如算法5和算法6。

Algorithm 5 Convert2Unsigned, $|_{to}(U)$

Require: $signRange = (ranges, bitwidth, signedness), signedness \in \{S, N\}$

Ensure: new SignRange with Unsigned type

1: $convertVal \leftarrow top(bitwidth, Unsigned).y + 1$

2: $nRanges \leftarrow \varnothing$

for all $range$ in $range(signRange)$ **do**
    if $range < 0$ **then**
        $nRanges \leftarrow nRanges \cup (range +_R convertVal)$
    else if $range > -1$ **then**
        $nRanges \leftarrow nRanges \cup range$
    else
        $nRanges \leftarrow nRanges \cup ([range.x, -1] +_R convertVal) \cup [0, range.y]$
    end if
end for
return $nSignRange = ([[nRanges]]_{MR}, bitwidth, Unsigned)$

Algorithm 6 Convert2Signed, $|_{to}(S)$

Require: $signRange = (ranges, bitwidth, signedness), signedness \in \{U, N\}$

Ensure: new SignRange with Signed type

$topVal \leftarrow top(bitwidth, Signed)$
$convertVal \leftarrow top(bitwidth, Unsigned).y + 1$
$nRanges \leftarrow \varnothing$
for all $range$ in $range(signRange)$ **do**
    if $range < topVal + 1$ **then**
        $nRanges \leftarrow nRanges \cup range$
    else if $range > topVal$ **then**
        $nRanges \leftarrow nRanges \cup (range -_R convertVal)$
    else
        $nRanges \leftarrow nRanges \cup [range.x, topVal]$
        $nRanges \leftarrow nRanges \cup ([topVal + 1, range.y] -_R convertVal)$
    end if
end for
return $nSignRange = ([[nRanges]]_{MR}, bitwidth, Signed)$

与 MultiRange 相比，SignRange 的计算规则着重考虑符号性。一般地，二元操作符在运算前需要统一其左右操作数的符号性。如无特殊说明，对操作数 srA, srB 参与的二元运算 $srA \cdot_{op} srB$ 将按如下规则（Unifiy Signedness）更新自己的符号性并进行运算：

（1）当 $\cdot_{op} \in \{add, subtract, multiply, divide, mod\}$ 时，按下列规则更新符号性、

位宽并运算：

$$srA \cdot_{op} srB := \begin{cases} srA|_{to}(U) \cdot_{op} srB|_{to}(U) & sgn(srA) = U \vee sgn(srB) = U \\ srA \cdot_{op} srB & sgn(srA) = N \wedge sgn(srB) = N \\ srA|_{to}(S) \cdot_{op} srB|_{to}(S) & \text{other cases} \end{cases} \tag{2-10}$$

$$width = width(srA) = width(srB) := \max(width(srA), width(srB)) \tag{2-11}$$

（2）当 $\cdot_{op} \in \{and, or, xor\}$ 时，按下列规则运算，不更新位宽与符号性：

$$srA \cdot_{op} srB := srA|_{to}(S) \cdot_{op} srB|_{to}(S) \tag{2-12}$$

（3）当 $\cdot_{op} \in \{shl, shr\}$ 时，按下列规则更新符号性并运算，不更新位宽：

$$srA \cdot_{op} srB := srA \cdot_{op} srB|_{to}(U) \tag{2-13}$$

SignRange 的计算规则如表2.4所示。方便起见，记 $signRange = (ranges, width, signedness)$ 为一个 SignRange。根据 Unify Signedness 规则，由于多数二元运算的操作数在运算前要更新符号性与位宽，因此表2.4中若如无特殊说明，计算结果中的 $width, signedness$ 均为更新后的值。

表 2.4　SignRange 运算规则

运算	符号	运算规则
negate	neg_{SR}	$neg_{SR}(signRange) := \begin{cases} (neg_{MR}(ranges), w, s) & s \in \{Signed, Unknown\} \\ (neg_{MR}(ranges|_{to}(S))|_{to}(U), w, s) & s = Unsigned \end{cases}$
add	$+_{SR}$	$srA +_{SR} srB := (range(srA) +_{MR} range(srB), width, signedness)$
subtract	$-_{SR}$	$srA -_{SR} srB := (range(srA) -_{MR} range(srB), width, signedness)$
multiply	$\times_{SR}$	$srA \times_{SR} srB := (range(srA) \times_{MR} range(srB), width, signedness)$
divide ($op_2 \neq 0$)	$\div_{SR}$	$srA \div_{SR} srB := (range(srA) \div_{MR} range(srB), width, signedness)$
modular ($op_2 \neq 0$)	mod_{SR}	$srA \ \text{mod}_{SR} srB := (range(srA) \ \text{mod}_{MR} range(srB), width, signedness)$
and	and_{SR}	$srA\, and_{SR}\, srB := (range(srA)\, and_{MR}\, range(srB), widthA, signednessA)$
or	or_{SR}	$srA\, and_{SR}\, srB := (range(srA)\, and_{MR}\, range(srB), widthA, signednessA)$
xor	xor_{SR}	$srA\, and_{SR}\, srB := (range(srA)\, and_{MR}\, range(srB), widthA, signednessA)$

续下页

续表 2.4　SignRange 运算规则

运算	符号	运算规则
shl	$<<_{SR}$	$srA <<_{SR} srB :=$ $(range(srA) <<_{MR} range(srB'), widthA, signedness)$, in which $srB' := srB.kill([widthA, +\infty]).kill([-\infty, -1])$
shr	$>>_{SR}$	$srA <<_{SR} srB :=$ $(range(srA) >>_{MR} range(srB'), widthA, signedness)$, in which $srB' := srB.kill([widthA, +\infty]).kill([-\infty, -1])$

在上一小节中，我们提到 MultiRange 在某些操作下会产生一定的精度丢失，这些精度丢失将会在后续章节进行修正。本节介绍的 SignRange 由于是 MultiRange 的扩展，其计算规则大部分调用了 MultiRange 的计算规则，因此对 MultiRange 的精度优化可以不加修改的继承到 SignRange。

需要注意的是，由于 SignRange 是位敏感的，因此在进行 shl 和 shr 操作时，其计算精度仍可优化，优化方法将在后续章节随 MultiRange 的优化一同给出。

SignRange 的比较规则定义如下：

$$srA < srB \iff range(srA') < range(srB') \tag{2-14}$$

$$srA > srB \iff range(srA') > range(srB') \tag{2-15}$$

其中，srA', srB' 为根据 Unify Signedness 规则（2-10）更新符号性和位宽后得到的 SignRange。

2.3.5　线性区间状态（ApRangeState）理论域

通过上一小节的分析，使用位敏感的线性区间 SignRange 可以对整型变量的取值范围做精确的抽象表示和计算模拟，本节介绍的线性区间状态 ApRangeState 则是对控制流自动机（CFA）中所表示的程序状态的精确抽象，它描述了 CFA 上某程序状态下各个整型变量的取值范围。

线性区间状态 ApRangeState := AccessPath $\mapsto$ SignRange 是一个从 AccessPath 到 SignRange 的映射，描述了在 CFA 节点 n 所表示的程序状态下，所有通过访问路径 AccessPath 访问得到的整型变量的可能取值范围 SignRange。其中，AccessPath 描述了一个整型变量在内存中的访问路径，通过这个访问路径，可以得到整型变量在内存中的唯一表示。

为方便起见，约定 $aps(ApRangeState)$ 为取 ApRangeState 的访问路径集合，约定 $ranges(ApRangeState)$ 为取 ApRangeState 的线性区间表示 SignRange 的集合，

约定 $ApRangeState|_{AccessPath}$ 为取 ApRangeState 上访问路径为 AccessPath 的线性区间表示 SignRange。

我们定义 Γ_n 为从访问路径 AccessPath 指向的整型变量到线性区间 SignRange 的一个映射：对位于 CFA 状态节点 n 上的整型变量的访问路径 $accessPath$，有 $\Gamma_n(accessPath) := signRange$。其中，$signRange$ 即为对应变量的取值区间的抽象表示。

定义 T 为从 CFA 上的程序状态节点 n 到线性区间状态 ApRangeState 的一个映射，$T(n) := apRangeState = \{(ap, sr) | ap \in ApSet(n), sr = \Gamma_n(ap)\}$。其中，$ApSet(n)$ 为 CFA 上节点 n 表示的程序状态中所有整型变量的访问路径集合。

对于一个待分析程序，我们在其等价表示 CFA 上定义理论域 D_S：

定义 2.7：在 CFA $= (N, E)$ 上定义的线性区间状态理论域 $D_S := \{T(n) \mid n \in N\}$ 是一个半格：

- D_S 上的最大元（$\top_S$）是 $\{(ap, sr) \mid ap \in ApSet(n_1) \cup \cdots \cup ApSet(n_n), sr = \Gamma_{n_i}(ap) \sqcup_{SR} \cdots \sqcup_{SR} \Gamma_{n_n}(ap), n_i \in N\}$，最小元（$\bot_S$）是 $\varnothing$；
- D_S 上的偏序关系（$\leqslant_S$）定义为：
$$apRStateA \leqslant_S apRStateB \iff aps(apRStateA) \subseteq aps(apRStateB) \wedge \forall ap \in aps(apRStateA), apRStateA|_{ap} \leqslant_{SR} apRStateB|_{ap};$$
- D_S 上的上确界操作（$\sqcup_S$）定义为：
$$apRStateA \sqcup_S apRStateB := \{(ap, sr) \mid ap \in aps(apRStateA) \cup aps(apRStateB), sr = apRStateA|_{ap} \sqcup_{SR} apRStateB|_{ap}\}$$
。

本节定义的线性区间状态 ApRangeState 是对控制流自动机 CFA 中节点 n 上整型变量取值信息的精确抽象。根据章节2.2.1中的定义，CFA 的边 e 表示从一个程序状态到下一个程序状态的过程，$act(e)$ 表示边 e 所执行的动作。对应的，我们定义理论域 D_S 上的变迁关系 $\rightsquigarrow_S$ 为 CFA 的边 e 所表示的程序变迁关系的抽象：

定义 2.8：对于一个控制流自动机 CFA $= (N, E), E = N \times Ops \times N$，定义线性区间状态理论域 D_S 上的变迁关系 $\rightsquigarrow_S = D_S \times Ops \times D_S$ 是 CFA 的边 $e \in E$ 的抽象表示。其中，Ops 为变迁关系上的操作。我们用函数 $act(\rightsquigarrow_S)$ 来表示 Ops 所具体执行的动作，称之为变迁函数。

记 $Y(e) :=\rightsquigarrow_S$ 为从边 e 到变迁关系 $\rightsquigarrow_S$ 的一个映射。记符号 $|_{act(\rightsquigarrow_S)}$ 为在当前线性区间状态应用变迁函数 $act(\rightsquigarrow_S)$ 得到的新的线性区间状态：$apRangeState' = apRangeState|_{act(\rightsquigarrow_S)}$。

根据 CFA 上对应的边 $e \in E$ 所表示的指令类型的不同（见公式2-1），D_S 上的变迁函数 $act(\rightsquigarrow_S)$ 随之有不同的定义：

对于空白边 $e \in BlankEdge$ 上的操作 $act(e) = \varepsilon$，有：

$$act(\rightsquigarrow_S) := \varepsilon,\ type(e) = blank \tag{2-16}$$

对于假设边 $e \in AssumeEdge$ 上的操作 $act(e) = (\%cmp, truth)$，有：

$$act(\rightsquigarrow_S) := refineVals \circ updateVal(\%cmp, truth),\ type(e) = assume \tag{2-17}$$

对于 Phi 边 $e \in PhiEdge$ 上的操作 $act(e) = (\%phi, index)$，有：

$$act(\rightsquigarrow_S) := applySelection(\%phi, index),\ type(e) = phi \tag{2-18}$$

对于指令边 $e \in InstructionEdge$ 上的操作 $act(e) = (\%inst)$，有：

$$act(\rightsquigarrow_S) := \begin{cases} handleAllocaInst(inst) & inst \in AllocaInst \\ handleBinaryOperator(inst) & inst \in BinaryOperatorInst \\ handleCallInst(inst) & inst \in CallInst \\ handleCastInst(inst) & inst \in CastInst \\ handleCmpInst(inst) & inst \in CmpInst \\ handleLoadInst(inst) & inst \in LoadInst \\ handleReturnInst(inst) & inst \in ReturnInst \\ handleStoreInst(inst) & inst \in StoreInst \end{cases} \tag{2-19}$$

【这里是老师当时写的】对于形如 $z = x \cdot_{op} y$ 这样的操作边（Instruction Edge），其变迁关系为：

$$\frac{X = \alpha_R(x), Y = \alpha_R(y)}{\alpha_R(Z) \leftarrow X \cdot_{op} Y}\ \text{T-R-OP} \tag{2-20}$$

其中，α_R 函数表示当前抽象状态中 a 和 b 的抽象表示（表示为 D_R 中的区间）。这里，在区间上的运算操作是在 I_R 和 D_R 上运算的扩展，

2.3.6　精度优化

在2.3.3和2.3.4小节中，

2.4　基于区间算数的整数缺陷检测方法

这里介绍 checker 的原理与检测规则、生成 report 的方法。

2.5　模块实现

介绍具体实现。

2.6　实现方案

这里放工具图

2.7　实验过程与结果

这里介绍在 Juliet 与选取的几个项目上的测试结果。

2.8　本章小结

水。assume, phi, instruction

第 3 章　基于环状区间的整型缺陷检测 → 这个不用具体实现，因为看不出来

为什么要使用环状区间、它能带来什么好处？→ 【主要解决第二章解决不了的问题，如 CornerCase】

【章节不要太长，主要突出差别，重点介绍环状区间】

要解决的问题：【在实现层 LLVM 有一些问题】

3.1　基于环状区间的抽象域设计与实现

【疑问】我觉得这里如果要写的话应该会被疑似抄袭那篇论文吧

【要对原文算法有一定的简化或修改】

3.2　基于环状区间的整型缺陷检测方法

类似于上一章，这里可能会重复阐述。

3.3　模块实现

介绍具体如何实现。

3.4　实验过程与结果

同样，在 Juliet 与项目上跑一跑。

3.5　本章小结

水。

第 4 章　基于值流图的整型变量关系分析与缺陷检测

4.1　引言

在软件研发活动中，软件的验证与确认[14-15] 是十分重要的步骤。验证（verification）的目的是评估软件是否如软件定义般实现，而确认（validation）的目的则是评估软件是否符合预期的需求。在具体实践时，软件确认通常会面临较大困难[16]：要判断软件是否符合预期的需求，实际上是要判断软件实现与软件需求是否匹配。然而两者的描述方式是不同的，前者是具有复杂逻辑结构的代码，后者则是抽象的文字描述，相差较大。

面对上述问题，业界常用的方法是根据需求文档，逐条测试系统是否完成了相应需求[17]。但是该方法是不完备的：在软件分支条件复杂的情况下，人们往往不能穷举出程序所有可能的输入与执行路径，造成了软件确认的不完备。另一种常用方法是在确认时尝试理解代码，随后评估代码的逻辑是否与需求相对应。这种方案仍存在两个问题：（1）代码语义的精确理解是复杂的，过程中需要较多的人工介入[18-20]，难以很好的自动化；（2）软件需求本身的描述通常不够形式化，甚至大部分情况下不够具体。

针对以上问题，本章提出基于精确值流图的程序理解与函数摘要生成算法，该方法拟解决如下问题：（1）自动抽取系统的语义，进而生成摘要信息供与用户进行需求比对；（2）根据生成的摘要内容进一步帮助用户细化需求。

该方法的核心是快速、自动化的从代码中抽取出程序语义，建立程序实现与需求之间的关系，帮助开发人员理解程序进行需求确认。程序理解[21-22] 领域以往的研究工作主要基于程序切片技术、程序标记技术以及执行可视化技术。

程序切片技术[23] 本质上是一种代码拆解技术。它通过剔除与指定变量不相关的代码语句，一定程度上减少了用户的代码阅读量。但是由于其并未能给出更上层的程序语义，用户仍需要阅读源码以获取知识。与之相反，程序标记技术[24] 通过在源码上附加辅助信息的方式来帮助用户理解代码。现有工具可提供的辅助信息多种多样，但本质上用户仍要理解代码逻辑，并不能显著提升程序理解速度。执行可视化工具使用动态分析获取程序的执行信息并将之可视化，为了达到分析目的，工具将不同执行过程信息加以融合。其缺点是当程序逻辑复杂时，可视化的效果会变差，且需要为其配置程序运行环境，具有上手难度。

在条件分支众多、代码逻辑复杂的情况下，现有工具很难帮助用户理清程序输入输出变量之间的关系。文章 [25] 指出，约 25% 的代码维护工作流程是发现

问题-修改-再验证的，同时，大量的开发者通过假设并验证程序的行为来理解软件[26]。因此，若使用一种算法（工具）帮助开发者快速准确地抽取程序语义将会大大减少相关工作时间。

本章中，作者提出了一种基于精确值流图（Value Flow Graph，简称 VFG）的自动化程序静态分析算法，该算法通过使用指针分析得到内存模型，基于内存模型构造 VFG，并进一步分析得到 VFG 上的程序语义。

本方法是纯静态分析方法，不依赖于程序的具体执行路径，拥有较好的完备性。分析结果可用于开发人员进行程序理解和需求确认，也可以用于自动化的缺陷分析。

本章的主要贡献如下：

1. 设计实现了一种基于内存模型的精准值流图分析构造方法，可处理动态内存空间、指针别名等传统数据流分析无法处理的语义关系，可区分数据流依赖、控制流依赖等依赖条件；
2. 基于值流图构造分析程序语义，该过程可自动化，并进一步可用于需求确认和缺陷分析，提升程序理解效率，提高代码质量；
3. 实现了一套工具并开展相关实验，该工具可自动生成可读的程序摘要，实验结果表明可节约 60% 的需求确认时间，并可排除代码中的隐蔽缺陷。

4.2　研究动机

程序理解往往是软件研发、软件验证与维护工作中耗时最大的工作内容。作者选取了某机动车变速器控制逻辑的部分软件需求与代码实现作为分析案例来介绍本章所述算法。

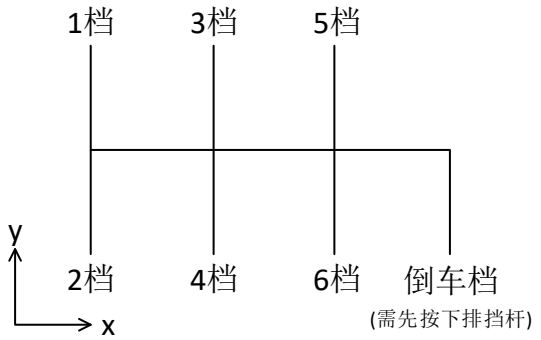

图 4.1　变速器排挡杆

该控制逻辑应用于六档变速杆，支持下压式倒挡，其档位映射如图4.1所示。现要求实现相关函数完成图4.1所示控制逻辑，代码应遵循如下要求：

1. 该函数应接受 4 个参数：x、y、*diverse* 和 *oldGear*，根据参数计算出档位值 *gear* 并返回；
2. 返回的档位值 *gear* 的取值集合为 $\{0, 1, 2, 3, 4, 5, 6, -1, -2\}$。其中 1-6 为前进档，-2 为倒车档；
3. x 为变速器挡杆在水平方向上的行程。当取值在 [0, 380) 区间时，表示 1 或 2 档；当取值在 [380, 690] 区间时，表示 3 或 4 档；当取值大于 690 时，表示 5、6 或-2 档（倒车档）；
4. y 为变速器挡杆在竖直方向上的行程。当取值在 [0, 160) 区间时，档位的可能取值为 2、4、6 档和-2 档；当取值在 [160, 780] 区间时，档位的可能取值为 0 或-1（要求下压挡杆）；当取值大于 780 时，档位的可能取值为 1、3 和 5 档；
5. *diverse* 为倒挡信号，当下压变速器挡杆时 *diverse* 值为 1，否则为 0；
6. *oldGear* 的取值为本次变速器挡杆操作前的档位取值。

代码 4.1　变速器档位控制的一个函数实现

```
1  int calculateGear(uint x, uint y, uint diverse, int
       oldGear)
2  {
3    int gear = 0;
4    if (y < 160) {
5      if (x < 380)
6        gear = 2;
7      else if (x > 690)
8        gear = 6;
9      else
10       gear = 4;
11   }
12   else if (y > 780) {
13     if (x < 380)
14       gear = 1;
15     else if (x > 690)
16       gear = 5;
17     else
```

```
18        gear = 3;
19    }
20    if (diverse) {
21      if ((gear == 6) && ((oldGear == -1 || oldGear ==
          -2)))
22        gear = -2;
23      else if (gear == 0);
24        gear = -1;
25      }
26    return gear;
27  }
```

如代码4.1所示为上述需求所对应的一个函数实现。当参数取值情况多种多样、条件分支复杂时，开发人员将难以直观的获取到输入输出之间的关系。例如，在需求文档中，档位值为 6 的条件是 $(x > 690 \wedge y < 160 \wedge (diverse = 0 \vee (oldGear \neq -1 \wedge oldGear \neq -2)))$，但开发者很难直观从代码中获取到相应的控制条件，只有通过表达式分析，才可以自动从代码中得知。试想，如果纯人工的完成这样的行为，不仅费时费力还可能出现条件遗漏和边界条件错误等情况。更重要的是，因为人工操作的不稳定，很难抓住需求和代码实现之间的微小差异。例如，在以上代码中实际存在了一个很严重的语义问题，导致在 $diverse \neq 0$ 的情况下代码执行与预期不符。即使有完整的需求文档与源代码，没有工具支持的情况下对其进行一致性确认也是十分困难的事情。本章所述工具可以帮助解决以上问题。如表4.1的 1、3 列所示为期望的返回值与控制逻辑，而实际代码的行为如表4.1的 1、2 列所示（方便起见分别将参数 $diverse$ 和 $oldGear$ 简写为 dv 和 og）。不难发现程序无法返回 $gear = -2$ 且返回值 $gear$ 与参数 $oldGear$ 无关，与预期不符。开发人员通过对比 2、3 列即可确认出该程序行为与需求之间的差异，帮助发现代码问题。该问题的根源在于程序第 23 行行尾多了一个分号致使控制流出错，经过修改后程序与需求可完全一致。

本章接下来的部分主要解释如何自动化实现以上过程。重点介绍值流图构建、语义分析、表达式分析等关键步骤。

4.3　基于值流图的程序理解与需求确认方法

本节介绍的方法是一种基于精确值流图的自动化程序分析算法。值流图是对程序的一种数据流表示，可通过基于内存模型的语义分析自动获得。基于值流图

表 4.1　根据代码4.1生成的函数摘要

返回值	实际的控制逻辑	期望的控制逻辑
-2	-	$(x > 690) \wedge (y < 160) \wedge$ $(og = -2 \vee og = -1) \wedge (dv \neq 0)$
-1	dv ≠ 0	$(160 \leq y \leq 780) \wedge (dv \neq 0)$
0	$(160 \leq y \leq 780) \wedge (dv = 0)$	$(160 \leq y \leq 780) \wedge (dv = 0)$
1	$(x < 380) \wedge (y > 780) \wedge (dv = 0)$	$(x < 380) \wedge (y > 780)$
2	$(x < 380) \wedge (y < 160) \wedge (dv = 0)$	$(x < 380) \wedge (y < 160)$
3	$(380 \leq x \leq 690) \wedge (y > 780) \wedge (dv = 0)$	$(380 \leq x \leq 690) \wedge (y > 780)$
4	$(380 \leq x \leq 690) \wedge (y < 160) \wedge (dv = 0)$	$(380 \leq x \leq 690) \wedge (y < 160)$
5	$(x > 690) \wedge (y > 780) \wedge (dv = 0)$	$(x > 690) \wedge (y > 780)$
6	$(x > 690) \wedge (y < 160) \wedge (dv = 0)$	$(x > 690) \wedge (y < 160) \wedge$ $(og \neq -2 \wedge og \neq -1)$

可执行多种静态分析算法，得到结构化表示的程序语义，进而进行需求确认。算法的工作流程如图4.2所示，首先借助现有静态分析框架将源代码转化为语义等价的控制流自动机（CFA），根据 CFA 提供的内存读写信息进一步生成精确值流图（VFG）。在得到 VFG 后，执行表达式分析算法进而得到变量间的符号表达式关系，最终根据表达式信息生成模块摘要并保存在源文件中。

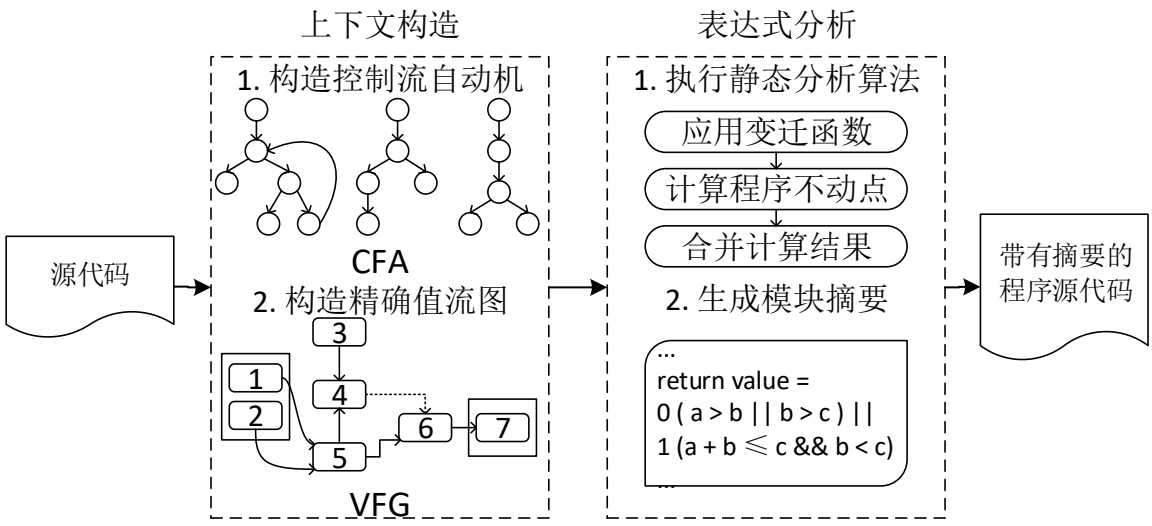

图 4.2　算法的工作流程图

4.3.1　精确值流图的定义与构造

在传统的基于控制流图的数据流分析方法中，同一变量在不同路径下的计算结果不同，为了保证算法的正确性，这类算法通常在交汇节点定义状态合并函数，具体使用上近似将变量可能的取值合并。但这将忽略路径信息导致分析精度不足。而精确值流图则是通过内存行为分析与指针别名操作分析，构造出变量取值与数据依赖和控制依赖相结合的值流图。这种精确值流图是路径敏感的，可用来解决状态计算过程中产生的路径丢失问题。

代码 4.2　代码样例

```
1  int main() {
2    int x = 0, y = 1;
3    int *a;
4    int p;
5    if (p)
6      a = &x;
7    else
8      a = &y;
9    return *a;
10 }
```

以代码4.2为例，在第 9 行的数据流关系为：$a \leftarrow \{\&x, \&y\}$。而如果使用路径敏感的数据流分析，由于有了路径可达条件信息，可以进一步将 a 的数据流表示为：$a \leftarrow \{(\&x, p \neq 0), (\&y, p = 0)\}$。

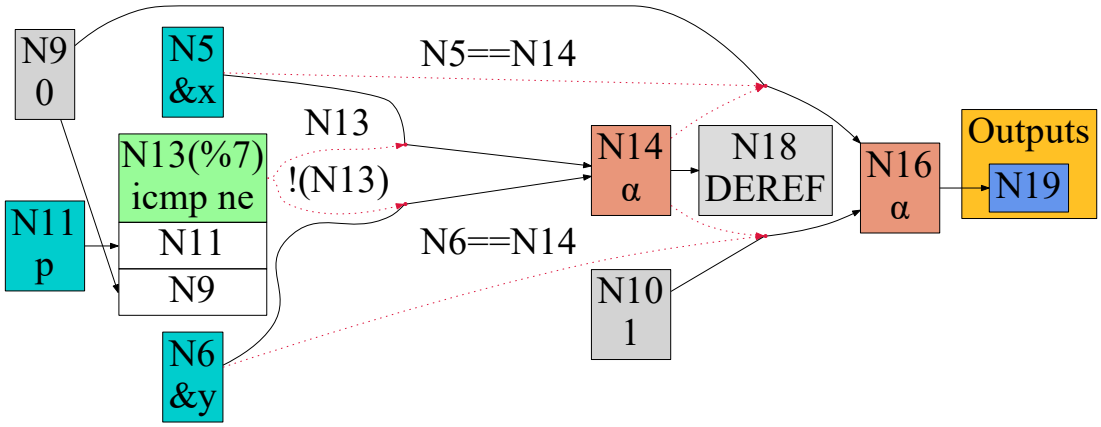

图 4.3　根据代码4.2生成的值流图

像这样，我们可以把数据流 $\bullet \leftarrow \bullet$ 作为值流图上的边，把数据流上承载的数据作为值流图的点，构造值流图如图4.3所示。

值流图按函数划分为不同的值流模块（$Module$），每个 $Module = (N, E)$ 是一个有向图，其中 N 是值流图节点集合，每个节点代表程序中的变量或常量。节点 $n \in N$ 上也可以带有条件，表示控制依赖。$E \subseteq N \times N$ 为有向边集合，表示值的流向。边 $e \in E$ 可以附加条件，表示数据依赖。

我们将 VFG 的节点按照功能分为如下几类：

常量节点 ConstNode：表示程序中的常量，如整数常量、字符串常量、函数指针常量、全局变量的地址常量等，用 $< c >$ 表示，其中 c 为字面常量。如图4.3中的节点 N_9 可以表示为 $< 0 >$。

起始节点 StartNode：表示值流图中数据初始化节点，如程序中的变量初始值、参数、函数调用的返回值等等。按照定义可知，起始节点在 VFG 图中无前驱节点，且该节点不是常量。初始节点用 $< ap >$ 表示，其中 ap 是对应的数据的内存位置的访问路径。如图4.3中的节点 N_{11} 可表示为 $< p >$。

复制节点 CopyNode：表示值的复制关系，它的值复制于前驱节点，用 $< n >$ 表示，其中 n 是 VFG 中的节点。如图4.3中的节点 N_{19} 可表示为 $< N_{16} >$。

运算节点 OperatorNode：表示由输入经过运算得到的结果，其中，运算节点以其前驱节点作为运算的输入。运算节点用 $< op, op_1, \ldots, op_n >$ 表示，其中 op 为运算符，$op_1, \ldots, op_n$ 为操作数。这里，操作数即为运算节点的前驱节点。图4.3中节点 N_{13} 可表示为 $<\neq, N_{11}, N_9 >$。

合并节点 JoinNode：合并节点具有多个前驱节点，该节点的取值为前驱节点中的某一个。合并节点用 $< (N_1, C_1), \ldots, (N_i, C_i) >$ 表示，其中 N_i 为 JoinNode 的前驱节点，C_i 为数据依赖，表示当 C_i 为真时，JoinNode 的取值为 N_i。图4.3中节点 N_{14} 可表示为 $< (N_5, N_{13}), (N_6, \neg N_{13}) >$。

对于每个模块 M，定义两个特殊节点集合 $In(M)$ 和 $Out(M)$ 分别表示模块的输入节点集合和输出节点集合。其中，输入节点包括函数的参数与全局变量，其类型总是 StartNode。而输出节点包括函数的返回值与全局变量，其类型总是 CopyNode。

为了便于后续描述算法，我们定义如下函数：

1. 定义函数 $literalVal :: ConstNode \rightarrow Value$ 用于取常量节点的具体值；
2. 定义函数 $symbolicVal :: StartNode \rightarrow Value$ 表示取起始节点的符号值。

4.3.2　表达式分析算法

定义 $ValueCase$ 为二元组 $(Value, Condition)$，其中 $Value$ 是值流图上某节点的可能取值，这个值既可以是具体的常量值，如 123、“abc” 等，又可以是符号表达式，如 “a + 3” 等；$Condition$ 是布尔表达式，它表示当值流图上某节点的 $Condition$ 条件为真时，其取值为 $Value$。

定义函数 $val :: ValueCase \rightarrow Value$ 表示取 $ValueCase$ 中 $Value$ 的值；函数 $cond :: ValueCase \rightarrow Condition$ 表示取 $ValueCase$ 中 $Condition$ 的值。

定义抽象函数 $T :: Node \rightarrow ValueSet$ 为从值流图节点到抽象域上的一个映射，其中 $ValueSet$ 为 $ValueCase$ 的集合。则基于值流图的表达式分析算法如算法7所示。

Algorithm 7 值流图分析算法

Require: $nodes$: current value flow graph's node set

1: $blockSet \leftarrow \varnothing$
2: $pending \leftarrow \{n \mid n \in nodes, n$ is ConstNode or StartNode $\}$
3: **while** $pending \neq \varnothing$ **do**
4: $cur \leftarrow$ peek($pending$)
5: $valueSet \leftarrow$ updateNodeValue(cur)
6: **if** $valueSet = \varnothing$ **then**
7: $blockSet \leftarrow blockSet \cup cur$
8: **else if** $valueSet$ is an update for cur **then**
9: $pending \leftarrow pending \cup successor(cur)$
10: move each $node$ from $blockSet$ to $pending$ which is unblocked due to $valueSet$
11: **end if**
12: **end while**

其中，函数 $updateNodeValue :: Node \rightarrow ValueSet$ 表示在迭代计算过程中每个节点的计算值，针对不同类型的节点，我们定义不同计算方法，具体计算方法如表4.2所示。

本算法属于值流图分析算法，定义了两个集合 $pending$ 和 $blockSet$，分别用于分别用于存储分析算法中待分析的节点和被阻塞的节点。

算法初始化时，将 $blockSet$ 集合置空，并将所有类型为 $StartNode$ 和 $ConstNode$ 的节点置入 $pending$ 集合中。在每次迭代计算过程中，从 $pending$ 集合

表 4.2　各类节点的值的更新算法

节点类型	更新值
ConstNode	$\{(literalVal(cur), true)\}$
StartNode	$\{(symbolicVal(cur), true)\}$
CopyNode	$\{(updateNodeValue(cur.predecessor))\}$
OperatorNode	$\left\{ \begin{pmatrix} operatorValue \\ (opr(cur), val(opd_1), ..., val(opd_i)) \\ , cond(opd_1) \wedge ... \wedge cond(opd_i) \end{pmatrix} \middle\| \begin{matrix} opd_i \in T(pre_i), \\ pre_i = pred(cur)[i] \end{matrix} \right\}$
JoinNode (Loop)	$\{(\mu, true) \mid \mu \leftarrow \mu \cup val(pred(cur))\}$
JoinNode (Normal)	$\{(val(pre_i), cond(pre_i) \wedge pcond(pre_i)) \mid pre_i \leftarrow pred_i(cur)\}$

中取出一个节点，如果其前驱节点或数据依赖条件没有计算完成，则将该节点放入 *pending* 集合中；否则，按照上表所规定的计算规则对该节点进行计算。如果之前从未对该节点进行过计算，或计算结果与之前不同，则将该节点的全部后继节点放入 *pending* 集合中，并检查 *blockSet* 中是否有依赖于本次计算结果的节点可被计算，如果有，则将相应节点重新置入 *pending* 集合中。根据上述方法，直到所有的节点均被计算完成、并且到达不动点，迭代结束。

4.3.3　程序理解与需求确认

以变速器档位控制代码（代码4.1）为例，其对应的值流图如图4.4所示。

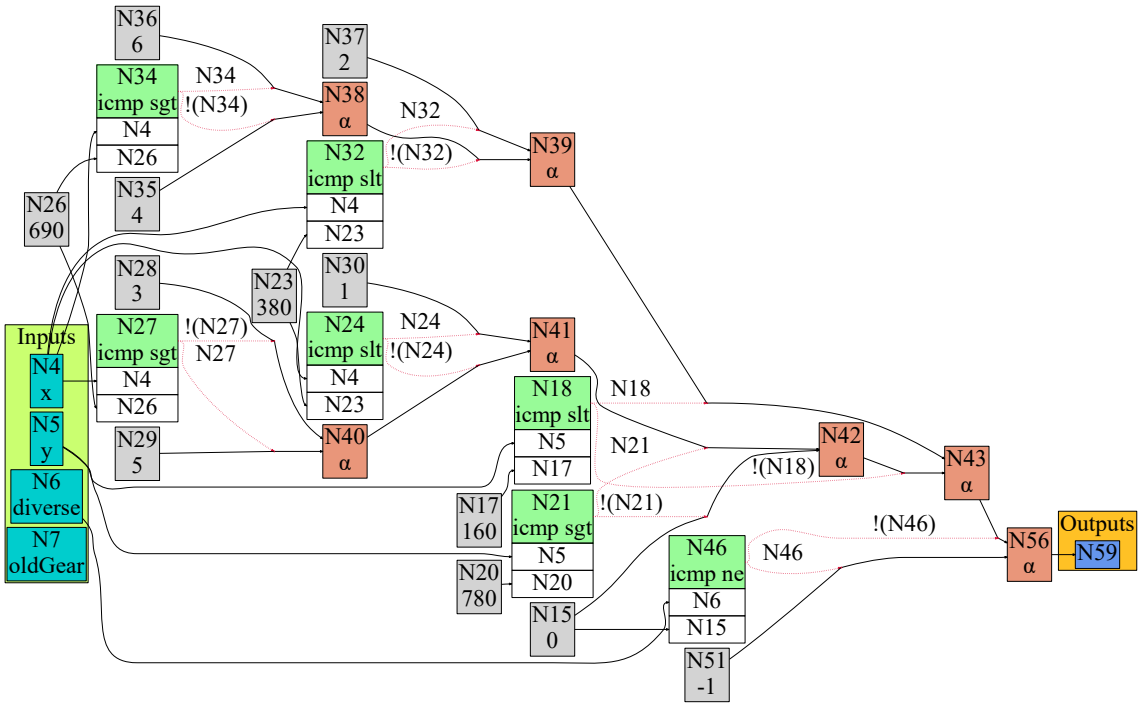

图 4.4　根据代码4.1生成的值流图

应用上述算法对所有节点进行计算，部分节点的计算结果如表4.3所示。

由于每个模块的输入节点集合 $In(M)$ 描述了模块所有可能的读入、输出节点集合 $Out(M)$ 描述了模块所有可能的输出，因此可根据输出输出节点集合构造函数摘要。如图4.4所示，由于 $Out(M)$ 中的节点 N_{59} 为复制节点，其计算结果与 N_{56} 相同，因此最终可以得到的模块摘要如表4.1第 1、2 列所示。

摘要清晰的描述了在不同分支路径条件下模块的输入与输出之间的对应关系，从而便于将软件实现与需求文档进行对比，进而提高程序理解与需求确认等工作的效率。另一方面，本算法是一种基于值流图的程序静态分析方法，因此在得到变量表达式关系的同时，可以在其上应用检查规则从而实现代码检查。典型的应用是检查每个节点在不同程序路径下的取值（表达式），判定程序是否存在如除零异常、空指针解引用、多重内存释放等缺陷，提高代码的正确率。

表 4.3　值流图上部分节点的计算结果

$Node$	$T(Node)$
N_4	$\{(x, true)\}$
N_{26}	$\{(690, true)\}$
N_{34}	$\{(x > 690, true)\}$
N_{38}	$\{(4, x \le 690), (6, x > 690)\}$
N_{41}	$\{(1, x < 380), (3, 380 \le x \le 690), (5, x > 690)\}$
N_{42}	$\left\{\begin{array}{c}(0, y \le 780), (1, x < 380 \wedge y > 780), (3, 380 \le x \le 690 \wedge y > 780),\\ (5, x > 690 \wedge y > 780)\end{array}\right\}$
N_{56}	$\left\{\begin{array}{c}(-1, dv \neq 0), (0, 160 \le y \le 780 \wedge dv = 0), (1, x < 380 \wedge y > 780 \wedge dv = 0),\\ (2, x < 380 \wedge y < 160 \wedge dv = 0), (3, 380 \le x \le 690 \wedge y > 780 \wedge dv = 0),\\ (4, 380 \le x \le 690 \wedge y < 160 \wedge dv = 0), (5, x > 690 \wedge y > 780 \wedge dv = 0),\\ (6, x > 690 \wedge y < 160 \wedge dv = 0),\end{array}\right\}$

4.4　评估与结果

4.4.1　实验设计

虑到根据源代码生成精确值流图的过程十分复杂，本算法的实现基于 Tsmart 静态分析框架，它的优点是使用静态分析的方法，高效的生成控制流自动机和精确值流图。同时该分析框架具有可配置性，能够较容易的获取到所需程序上下文信息，因此我们采用此分析框架，工具的开发语言为 Java，工具的框架图如图4.5所示：

本节从客观、主观两个方面评估论文算法（工具）在程序理解与需求确认方面的效用。客观评价重点关注算法的完整性与准确性。具体而言，我们考察算法

自动生成的摘要质量；主观评价则重点关注工具在具体的生产环境中的可用性与实用性。我们将分别从客观评价和主观评价两个方面对工具进行实验。

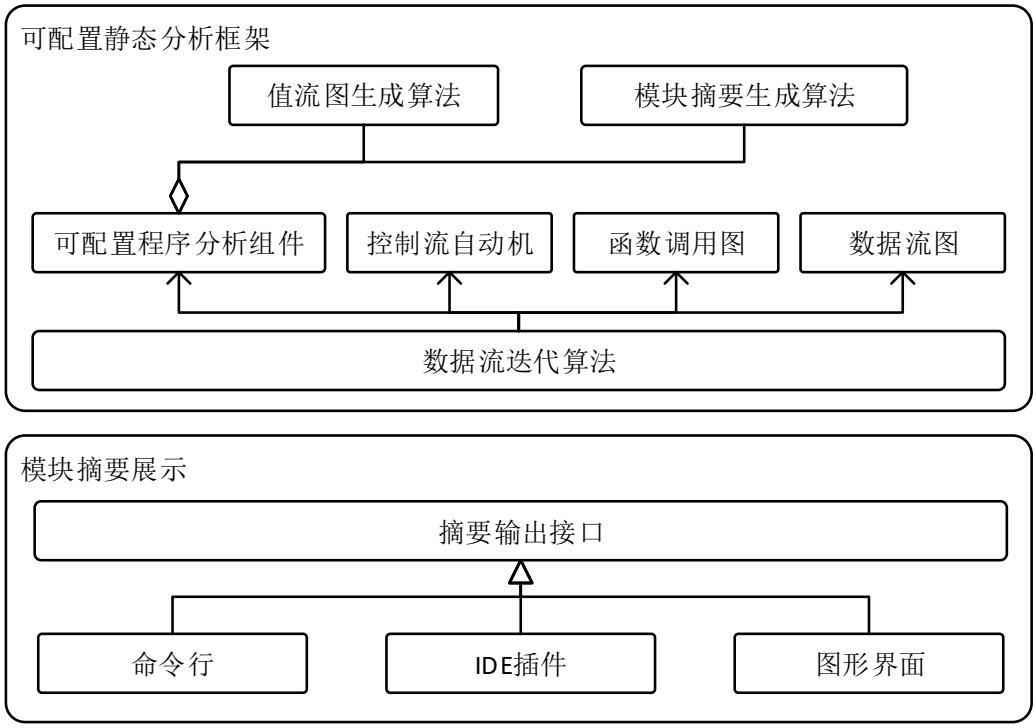

图 4.5　工具架构图

4.4.2　客观评价

客观评价方法主要包含以下几个步骤:(1)分别选取数学计算、计算机应用、工业领域和嵌入式领域中 8 个具有代表性的代码片段作为实验对象 (详见表4.4);(2)采用本文工具分别生成相应的函数摘要；(3) 判定工具自动生成的摘要与实际需求的差距。

由于摘要结果和程序的条件分支有关，因此本文分别对比两者在每个条件分支下的生成结果。同时，记录工具的时间和空间开销，包括摘要生成时间、占用内存大小以及生成的 VFG 大小，如表4.5所示。

经实验验证，本工具能够完整地生成不同类型代码的摘要，且生成摘要的条件分支数与需求一致。运行工具所消耗的资源相对较少，内存占用小于 200MB，生成的 VFG 均小于 1MB。工具可以较快的生成函数摘要，生成百行级别代码所需时间不超过 1 分钟。

表 4.4　选取的各类代码

类别	选取代码	文件名称
数学计算	三角形判定	triangle.c
	绝对值计算	abs.c
计算机应用	（加权平均）滤波算法	filter.c
	校验和算法	checksum.c
工业领域	车辆控制系统某算法	CDL_ARC429.c
	航天发动机控制系统某算法	ISP_TOOL.c
嵌入式领域	含 goto 语句的程序片段	msp430-decode.c
	带函数指针的程序片段	xen-ops.c

表 4.5　自动生成摘要和人工生成摘要的对比

程序代码		实际需求	自动摘要				正确率
类型	代码行	分支数	分支数	耗时/s	内存/MB	VFG 大小/KB	
三角形判定	24	4	4	3	134	9	1.0
绝对值	11	2	2	2	122	2	1.0
滤波算法	19	1	1	2	128	4	1.0
校验和算法	22	1	1	2	145	12	1.0
车辆控制系统	124	12	12	5	167	53	1.0
航发控制系统	178	15	15	6	171	61	1.0
Goto 语句	57	8	8	4	137	34	1.0
函数指针	36	3	3	3	132	6	1.0

4.4.3　主观评价

为保证主观评价的公平性与准确性，现从学校选取共 30 名条件相当的同学参与实验，他们均符合以下条件：（1）拥有超过 3 年的 C 语言编程经验，且均使用 C 语言通过了学校的编程水平测验；（2）对表4.4代码所在的背景了解程度相当。

实验方法包含以下几个步骤：（1）将 30 名同学分为源码组和摘要组进行对照实验，分别为其提供表 5 所示的 8 份源代码（或含摘要的源代码）；（2）令每名同学阅读代码，并令其说出代码功能，记录用时 T1；（3）令每名同学分析函数在不同输入的情况下的输出，记录用时 T2；（4）分别统计并对比两组同学在不同代码上回答问题所用时间 T1 和 T2 的平均值；（5）在源码组同学完成后，为其提供摘要。同时分别询问两组同学对摘要的看法。

如表4.6所示为两组同学在各个代码上回答实验步骤 2 和 3 中提出问题的平均用时 T1 和 T2。通过对比可知，在代码行数较小（10 行以内）且逻辑并不复杂的情况下，使用摘要对程序理解无明显帮助。但当面对几十甚至上百行代码时，使用摘要可明显减少用户理解代码所用时间。在含 goto 语句的实验样本上，可节省

60.5% 的程序理解时间。

通过实验可知，本工具可显著提高用户对代码的理解效率，能够帮助用户快速抽取程序语义并进行需求确认。同时，随着代码分支复杂度的提升，使用工具进行程序理解与需求确认的优势将越来越大。

表 4.6　各组对每类代码理解并作答所用时间

程序代码	源码组 T1/s	源码组 T2/s	摘要组 T1/s	摘要组 T2/s
三角形判定	45	32	37	14
绝对值	5	13	11	14
滤波算法	61	15	27	15
校验和算法	97	43	54	38
车辆控制系统	294	86	103	74
航发控制系统	318	92	114	91
Goto 语句	195	48	64	31
函数指针	82	33	40	29

在实验步骤 5 中，我们设计了数道问题，交予参与实验的同学回答并统计，最终得到的结果如表4.7。实验参与者对自动生成摘要工具总体持肯定态度，认为生成的摘要对理解代码有帮助。对于摘要是否可以帮助分析变更影响范围与帮助软件维护方面，一部分同学持观望和怀疑态度，认为本工具生成的摘要为函数级别的摘要，而变更影响范围分析与软件维护可能涉及全局代码，需要结合函数调用等信息进一步分析。

表 4.7　问卷设计与答复情况

问题设计	赞同人数	否定人数	其他想法
摘要是否有助于你理解代码?	28	2	0
你认为摘要是否对影响范围分析有帮助?	25	4	1
你认为摘要是否对软件维护有帮助?	22	7	1

4.5　后续工作与总结

本方法依赖于精确值流图的准确生成，当前可用工具相对较少，且在面对大型程序时消耗的时间空间较多。这会对本工具的结果造成一定影响。一个可能的解决方案是在原有的值流图算法上进行优化，针对于循环与数组，优化其结果。

另一方面，为了保证算法的收敛性与时间开销，本算法在处理循环时，引入了符号值 μ，算法的结果是带 μ 的符号表达式，这将在处理循环时带来一定的精度丢失。后续可以使用循环不变式等技术进一步提高分析精度。同时，为了增强

工具对全局影响范围的分析，后续将会结合函数调用等信息，提升工具的适用性。

软件验证与维护是软件研发中不可或缺的一环，而提高研发者对程序的理解速度则是其中的关键所在。传统的技术和工具或者难以有效帮助程序员减少代码阅读量，或者具有较高的使用门槛，难以广泛使用。本文的研究工作试图针对这一问题，提供一个程序理解算法与摘要生成工具。

本章首先通过一个变速器档位控制逻辑的案例剖析了程序理解对软件开发的重要性，并借此提出函数摘要的作用。随后介绍相关程序理解方法与摘要生成算法。我们在可配置的静态分析框架上实现了摘要生成算法，并从内外两个方面结合实验证明了摘要生成算法的有效性与实用性。

尽管本章提出的方法具有一定的局限性，但本章的研究结果仍可对程序理解、变量表达式分析和程序变更影响分析工具的设计与实现提供思路和指导。

第 5 章　总结与展望

5.1　工作总结

5.2　研究展望

参考文献

[1] Beyer D, Henzinger T A, Théoduloz G. Configurable software verification: Concretizing the convergence of model checking and program analysis // International Conference on Computer Aided Verification. Springer, 2007: 504-518.

[2] Cousot P, Cousot R. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints // Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages. 1977: 238-252.

[3] Cousot P, Cousot R. Systematic design of program analysis frameworks // Proceedings of the 6th ACM SIGACT-SIGPLAN symposium on Principles of programming languages. 1979: 269-282.

[4] 张健, 张超, 玄跻峰, 等. 程序分析研究进展. 软件学报, 2019, 30(01):80-109.

[5] Jeannet B, Miné A. Apron: A library of numerical abstract domains for static analysis // International Conference on Computer Aided Verification. Springer, 2009: 661-667.

[6] Singh G, Püschel M, Vechev M. A practical construction for decomposing numerical abstract domains. Proceedings of the ACM on Programming Languages, 2017, 2(POPL):1-28.

[7] Bagnara R, Hill P M, Zaffanella E. The parma polyhedra library: Toward a complete set of numerical abstractions for the analysis and verification of hardware and software systems. arXiv preprint cs/0612085, 2006.

[8] Aho A V, Sethi R, Ullman J D. Compilers, principles, techniques. Addison wesley, 1986, 7(8):9.

[9] Cooper K D, Harvey T J, Kennedy K. Iterative data-flow analysis. Revis ited, Department of Computer Science Rice University Houston, Texas, USA, 2004.

[10] Clarke L A. A system to generate test data and symbolically execute programs. IEEE Transactions on software engineering, 1976(3):215-222.

[11] King J C. Symbolic execution and program testing. Communications of the ACM, 1976, 19(7): 385-394.

[12] De Moura L, Bjørner N. Z3: An efficient smt solver // International conference on Tools and Algorithms for the Construction and Analysis of Systems. Springer, 2008: 337-340.

[13] Cheng B C, Hwu W M W. Modular interprocedural pointer analysis using access paths: design, implementation, and evaluation // Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation. 2000: 57-69.

[14] Wagner S, Abdulkhaleq A, Bogicevic I, et al. How are functionally similar code clones syntactically different? an empirical study and a benchmark. PeerJ Computer Science, 2016, 2:e49.

[15] Wallace D R, Fujii R U. Software verification and validation: an overview. Ieee Software, 1989, 6(3):10-17.

[16] Wagner S. Software product quality control. 2013.

[17] Ramler R, Biffl S, Grünbacher P. Value-based management of software testing // Value-based software engineering. Springer, 2006: 225-244.

[18] Ko A J, DeLine R, Venolia G. Information needs in collocated software development teams // 29th International Conference on Software Engineering (ICSE'07). IEEE, 2007: 344-353.

[19] Murphy G C, Kersten M, Findlater L. How are java software developers using the elipse ide? IEEE software, 2006, 23(4):76-83.

[20] Corbi T A. Program understanding: Challenge for the 1990s. IBM Systems Journal, 1989, 28 (2):294-306.

[21] Boysen J P. Factors affecting computer program comprehension. 1979.

[22] Sackman H, Erikson W J, Grant E E. Exploratory experimental studies comparing online and offline programming performance. Communications of the ACM, 1968, 11(1):3-11.

[23] Binkley D W, Gallagher K B. Program slicing // Advances in Computers: volume 43. Elsevier, 1996: 1-50.

[24] Sulír M, Porubän J. Labeling source code with metadata: A survey and taxonomy // 2017 Federated Conference on Computer Science and Information Systems (FedCSIS). IEEE, 2017: 721-729.

[25] Maalej W, Happel H J. Can development work describe itself? // 2010 7th IEEE working conference on mining software repositories (MSR 2010). IEEE, 2010: 191-200.

[26] Maalej W, Tiarks R, Roehm T, et al. On the comprehension of program comprehension. ACM Transactions on Software Engineering and Methodology (TOSEM), 2014, 23(4):1-37.

[27] 薛瑞尼. ThuThesis: 清华大学学位论文模板 [EB/OL]. 2017[2019-04-27]. https://github.com /xueruini/thuthesis.

致　谢

衷心感谢顾明老师，周[illegible]老师

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：_____

个人简历、在学期间发表的学术论文与研究成果

个人简历

1995 年 3 月 6 日出生于内蒙古莫力达瓦自治旗。

2013 年 9 月考入大连理工大学软件学院软件工程专业，2017 年 7 月本科毕业并获得软件工程学士学位。

2017 年 9 月考研进入清华大学软件学院攻读软件工程硕士学位至今。

发表的学术论文

[1] Yang Y, Ren T L, Zhu Y P, et al. PMUTs for handwriting recognition. In press. (已被 Integrated Ferroelectrics 录用. SCI 源刊.)

研究成果

[1] 任天令, 杨轶, 朱一平, 等. 硅基铁电微声学传感器畴极化区域控制和电极连接的方法: 中国, CN1602118A. (中国专利公开号)