基于语义分析的整型缺陷检测与程序理解

（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

二〇二〇年五月

Integer Defect Detection and Program Understanding Based on Semantic Analysis

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

May, 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

软件的验证与确认是软件研发过程中不可或缺的一环。前者关注软件是否如定义般实现，后者关注软件是否符合预期的需求。在软件实现中，开发人员常不可避免的因疏忽引入缺陷。一方面，部分具有特定模式的缺陷因现有检测工具表示精度的问题无法被及时发现与确认，这将给软件验证工作带来隐患；另一方面，代码中的深层缺陷，尤其是与需求的匹配性，则需要基于对代码的深层理解来实现。在实际工作中，这一过程通常占用较多的时间。因此，提高开发人员对程序代码的理解效率成为一个重点问题。

本文针对上述问题，试图从程序的整型缺陷检测与整型变量关系分析两个方面入手，在一定范围内解决问题。本文的工作内容如下：（1）针对当前线性区间抽象域由于表示能力不足而造成的程序分析精度损失的问题，提出了基于位敏感的区间代数的整型缺陷检测方法。具体地，本文依次设计了线性区间、线性多区间以及位敏感的线性多区间理论域与计算规则，能够较好的表示整型变量的可能取值范围；其次，为了得到程序的语义表示，本文设计了区间抽象域来描述程序的每个状态，并通过设计变迁规则，模拟程序状态的变迁，从而实现程序语义分析；最后通过设计缺陷检测规则实现程序的整型缺陷检测。（2）为了解决软件确认过程中因程序理解造成的低效问题，提出基于值流图的整型变量关系分析与缺陷检测方法。具体地，介绍了精确值流图的定义与构造，并在此基础上设计了程序语义的抽取方法，包括抽象域的设计、变迁规则的设计和表达式分析算法的设计。使用算法，可以对程序中的整型变量在不同程序路径下的取值关系进行语义抽取，并生成模块摘要。（3）基于上述两个研究工作所提出的解决方案，在 Tsmart 静态分析框架中进行编码实现，并通过实验验证了其正确性与实用性。

经过实验，本文提出的基于区间代数的整型缺陷方法在 Juliet 测试集上可达到 0% 的漏报率与低于 3% 的漏报率；本文提出的基于值流图的整型变量关系分析与缺陷检测方法能够在选用的程序代码上正确的生成程序摘要，并可有效提升研发人员对程序代码的理解效率，具有良好的应用价值。

关键词：软件验证与确认；缺陷检测；程序理解；静态分析

Abstract

Software verification and validation is an indispensable part of the software development process. The former focuses on whether the software is implemented as defined, while the latter focuses on whether the software meets the expected requirements. In software implementation, it is inevitable for developers to introduce defects due to negligence. On the one hand, some defects with specific patterns cannot be found and confirmed in time due to the representation accuracy of existing detection tools, which will bring hidden weakness to the software verification work; on the other hand, the deep defects in the code, especially the matching with the requirements, need to be realized based on the deep understanding of the code. In practice, this process usually takes more time. Therefore, it is a key problem to improve the efficiency of developers' understanding of program code.

In view of the above problems, this paper attempts to solve the problems in a certain range from the two aspects of program integer defect detection and integer variable relationship analysis. The work of this paper is as follows: (1) Aiming at the problem of the loss of program analysis precision caused by the lack of representation ability in the current linear interval abstract domain, an integer defect detection method based on bitwidth sensitive interval algebra is proposed. Specifically, this paper designs the linear interval, linear interval and bitwidth sensitive linear interval theory domain and calculation rules in turn, which can better represent the possible value range of integer variables; secondly, in order to obtain the semantic representation of the program, this paper designs the interval abstract domain to describe each state of the program, and simulates the change of the program state by designing the transition rules Finally, the integer defect detection is realized by designing defect detection rules. (2) In order to solve the problem of low efficiency caused by program understanding in the process of software validation, an integer variable relationship analysis and defect detection method based on value flow graph is proposed. Specifically, this paper introduces the definition and construction of the precise value flow graph, and designs the extraction method of program semantics, including the design of abstract domain, the design of transition rules and the design of expression analysis algorithm. Using the algorithm, we can extract the value relationship of integer variables in different program paths and generate module summary. (3) Based on the solutions proposed by the above two research works, the code is implemented in

the Tsmart static analysis framework, and its correctness and practicability are verified by experiments.

Through experiments, the algorithm based on interval arithmetic can achieve 0% false negative rate and less than 3% false positive rate in Juliet test set; the algorithm based on value flow graph can generate program summary correctly in the selected program code, and can effectively improve the understanding efficiency of developers on the program code, which has good application value.

Key Words: Software verification and validation; defect detection; program understanding; static analysis

目 录

第 1 章　引言

1.1　研究背景与意义

在软件研发活动中，软件的验证与确认[1-2]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。在实际应用中，软件验证与确认经常会遇到困难。

对于前者而言，尽管软件需求是明确的，但在开发人员实际开发过程中经常会因为疏漏而不可避免的在软件中引入一些错误，导致软件缺陷的发生。其中，因整数计算问题直接或间接造成的缺陷不占少数，典型的如整型溢出、除零异常、内存泄露等。这些缺陷如得不到及时的发现与修复将会严重影响系统的稳定性与可靠性，在一些生命攸关的领域这些缺陷将会无限放大并造成难以估量的损失。

1995 年，欧洲一研发时间超 8 年、耗资近 80 亿美元的阿丽亚娜 5 型运载火箭在发射 40 秒后发生爆炸，造成了巨大的经济损失。经事后排查，造成事故的原因是其导航系统试图将一个表示速度的浮点数转换为一个 16 位长的有符号整数。但在运行中该浮点数超过了整型变量的表示范围，导致转换失败。

时隔 22 年，人们在 Linux 内核的套接字模块中发现了一个因无符号减法造成运算溢出的问题，攻击者可以依照这个漏洞构造特殊的包套接字从而实现拒绝服务攻击和权限提升。通过这两个案例我们可以看出，整型缺陷问题一直存在且容易造成严重后果。如何在生产实践过程中及时、准确的找出整型计算造成的缺陷是十分重要的问题。

另一方面，软件确认也面临着较大困难[3]：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。

面对这一问题，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[4]。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[5-7]，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。

为了解决上述整型缺陷检测以及软件需求确认中出现的问题，本文分别提出

基于线性区间的整型缺陷检测方法与基于精确值流图的程序理解与函数摘要生成算法。

基于线性区间的整数缺陷检测方法拟解决如下问题：（1）对 C 语言整型变量进行抽象，借助静态分析对整型变量在程序运行时的取值范围做实时分析；（2）基于得到的分析结果，验证程序中是否存在整数缺陷，并生成缺陷检测报告。

基于精确值流图的程序理解与函数摘要生成算法拟解决如下问题：（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。

1.2　研究现状

本节介绍整型缺陷分析所用关键技术的研究现状，包括抽象解释技术、数据流分析和符号化执行。同时介绍有关程序理解的研究现状。

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[8]。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[9]。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[10]。另一方面，在开源领域出现了众多抽象域库，如 APRON[11]、ELINA[12]、PPL[13] 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。

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[14] 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[15] 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。

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[16-17] 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[18] 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。

1.2.4　程序理解

目前有关程序理解的研究有很多，按照技术类别，可分为程序切片技术、程序标记技术、执行可视化技术。 程序切片技术于 1979 年首次被 Weiser[19] 提出，是一种可执行的后向静态分析。所谓可执行，是指切片后的结果仍是可编译可执行的，后向是指其结果包含切片准则所依赖的所有程序语句，静态是指它只使用程序的静态分析结果，即所有可能的执行都被考虑在内。该算法使用一个迭代的过程获取程序切片，首先在控制流图中找到每个节点的直接相关语句，然后每次迭代过程中都加入间接相关语句，当结果到达不动点时，程序终止。此方法的优点是得到的切片结果仍是可编译、可执行的独立代码段。然而正因保证了程序原有特性，其切片结果并没有显著减少，这意味着开发人员的代码阅读量并未因此降低多少。

随后，由 Korel 和 Laski[20] 等人提出了动态切片技术，构造程序切片时采用了动态分析的技术，它能够根据程序的一次执行，获取程序在运行时的信息（尤其是指针、数组等），构造程序切片。由于动态切片的生成依赖于程序的执行，而程序的一次执行显然不能遍历所有执行路径，这就导致了切片结果不能包含可能相关的程序语句。优点是明显的：由于可以获取运行时的指针、数组等信息，其切片结果更加简单、准确。而缺点就是切片结果只针对特定输入，不具有兼容性。

综合以上两种切片技术，Gupta[21] 等人提出了混合切片技术，它同时使用动态分析和静态分析来获取信息，并生成程序切片。然而由于程序切片本身的特性，它只能做到减少开发人员的代码阅读量，并不能为程序员提供代码之上的抽象信息，因此在使用程序切片时，仍需阅读代码以得到知识。

程序标签技术的提出的目的是为程序员提供更多信息，以方便理解代码。目前有大量的相关文章与工具：SE-Editor[22] 是一个 Eclipse 插件，通过使用超链接注释实现在代码内展示与代码相关的图表、视频等。Stacksplorer[23]、RegViz[24] 和 DepDigger[25] 等工具为源代码提供了视觉增强，如为函数提供图形化的调用信息、

为正则表达式分组并标记组号以及根据变量的大小改变背景色等功能。还有一部分工具提供程序运行时信息，例如 In-situ[26] 会为函数创建包含运行性能信息的注释等。与本文工作较为类似的是[27]，它可以为函数提供摘要信息，然而这种摘要信息是以关键词的形式出现，仅阅读摘要并不能直接了解函数的行为。

以上工具本质上都是为程序员提供辅助信息以帮助理解，但实际在应用过程中人们仍需要阅读源代码，理解速度并没有质的提升。

执行可视化工具如 ExtraVis[28] 和 ExplorViz[29] 提供了另一种辅助理解的思路，这类工具在动态分析的基础上，基于多次执行，试图对程序的执行路径进行可视化。尽管这类工具创新性地提供了诸如 Bundle View 等视图，但由于其技术基础是动态分析，其兼容性不可避免地会减弱，不能够完整的表现程序在不同输入情况下的执行路径。同时，如果增加输入样例，则视图会变得更加复杂难以理解，违背了简化理解的初衷。另外，此类软件操作复杂，具有一定的学习成本，仍不适合快速理解代码。

1.3　研究内容与研究方案

线性区间抽象域是当前常用的用于刻画程序中整型变量取值范围的一种抽象域，但是在实际应用中，因其存在着无法刻画整型变量的符号性、且因为抽象域本身表示能力问题带来的精度丢失，本文试图设计一个新的用于刻画整型变量的取值范围的抽象域，相比于独立的线性区间抽象域，它具有更强的表示能力、更高的分析精度。

另一方面，由于理解代码一直是软件开发乃至软件确认过程中耗时最大的工作部分。在面对逻辑复杂、软件条件分支众多的情况下，理解程序变成了十分具有挑战的工作。经调研，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，因此本文试图提出一套算法（工具）来帮助开发者快速准确地抽取程序语义，加速程序的理解，从而提高人们的工作效率。

据此，本文的研究内容如下：

1. **基于位敏感的区间代数的整型缺陷检测**：为了增强现有区间抽象域表示能力和精度上的不足，本文的研究工作之一是为 C 语言整型变量设计新的抽象表示。抽象域的设计应能体现整型变量的符号性，且在保证分析效率的情况下，具有较高的分析精度。另一方面，应设计抽象域上的变迁规则，从而实现程序分析并给出判定程序中是否存在整型缺陷的方法。
2. **基于精确值流图的程序理解与需求确认**：为了帮助开发维护人员对程序的理解效率，提高程序确认的效率，本文的另一个研究工作即是提出并设计一种

程序理解方法，它可以自动化的抽取程序语义，并生成摘要，便于进一步的需求确认与缺陷分析。这种方法应能分析程序中的控制依赖与数据依赖，并准确生成不同路径条件下的变量关联信息。

3. **工具研发**：本文的研究工作应具体实现为可用工具，并在一定的测试集上分析工具的可用性与工作效率。

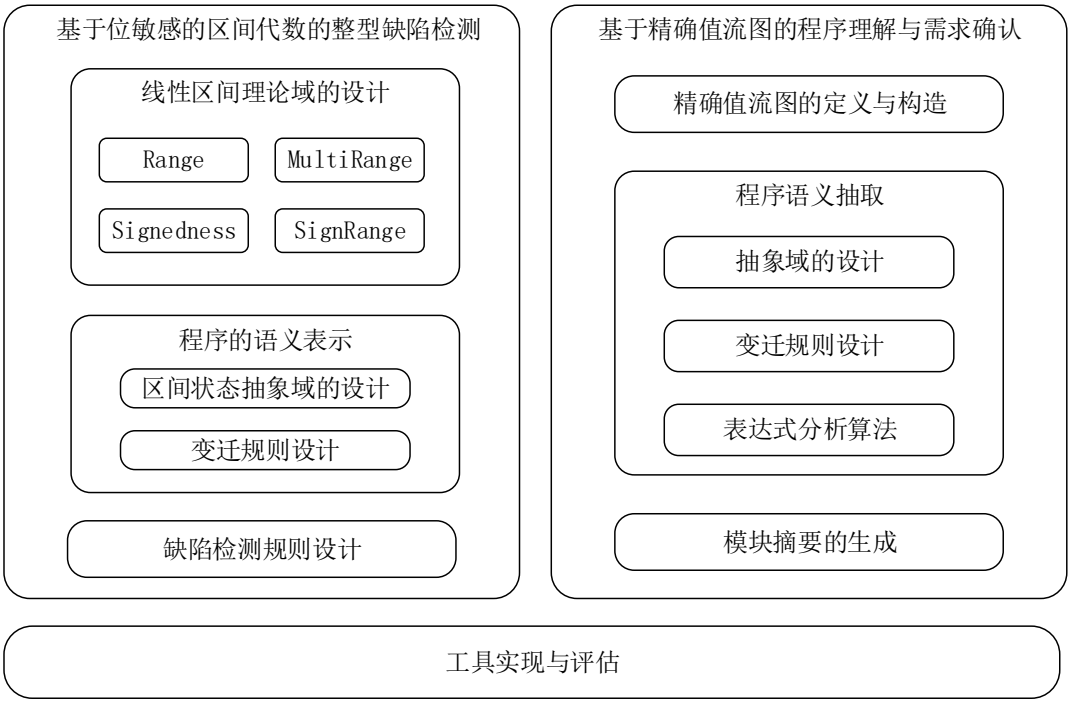

图 1.1　研究方案

基于上述三点研究内容，本文提出如图1.1所示的研究方案。

1. 针对基于位敏感的区间代数的整型缺陷检测，本文进行如下三个方面的工作：其一是线性区间理论域的设计。为了增强现有区间抽象域表示能力和精度的不足，本文依次定义了线性区间 Range、线性多区间 MultiRange 以及符号敏感的线性多区间 SignRange 并设计了其计算规则。最后定义的 SignRange 能够较好的表示程序中整型变量的可能取值范围。其二，为了实现程序的语义表示，我们设计了区间状态抽象域 RangeState 来描述程序的每个状态，并通过定义变迁规则，模拟程序状态的变迁，从而实现程序的语义分析。最后，通过在 RangeState 上设计缺陷检测规则，实现程序的整型缺陷检测。
2. 针对基于精确值流图的程序理解与需求确认，本文首先通过一个具体案例剖析使用基于值流图的程序理解方法为需求确认过程带来的好处，并以此展开，对该方法的具体内容做出了介绍。具体地，首先对精确值流图的定义与构造

做出说明，基于精确值流图，设计了程序语义抽取方法。具体内容包括抽象域的设计、变迁规则的设计和表达式分析算法的设计。在上述的基础上，我们可以对程序整型变量在不同程序路径下的取值关系进行语义抽取。最后，利用得到的信息生成模块摘要。

3. 最后，本文对上述两个研究方案中提出的方法进行编码实现，并设计相关实验，验证其正确性与实用性。

1.4　论文贡献

本文的贡献如下：

1. 设计了表示能力更强的线性区间抽象域与变迁规则，使用它可以抽象表示程序在不同状态下整型变量的整数取值范围，进一步地，实现了针对 C 语言的整数缺陷检测方法；
2. 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。同时，这一过程可自动化，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. 基于上述两点，分别编码实现了相关工具并开展相关实验，证明了其正确性与实用性。

1.5　论文组织结构

本文的组织结构如下：

第一章从软件验证与确认的角度介绍了整型缺陷分析与程序理解的相关背景知识与研究现状，并针对当前现状提出本文所研究内容与研究方案。

第二章介绍了基于区间代数的整数缺陷检测方法，首先介绍了有关控制流自动机与可配置程序分析框架的基础知识，随后依次介绍扩展的整数（RangeInteger）、线性区间（Range）、线性多区间（MultiRange）和位敏感的线性多区间（SignRange）的理论域与计算规则的设计。并介绍区间状态（RangeState）作为控制流自动机上状态节点的抽象表示，同时通过设计区间状态域上的变迁规则与整型缺陷检测规则，实现 C 语言上的整数缺陷检测。

第三章介绍了基于精确值流图的程序理解与需求确认方法，首先通过一个机动车变速器控制逻辑的软件需求与实现的具体案例，介绍了程序理解对软件开发的重要性，并提出使用摘要来降低程序理解的难度。后续内容具体介绍了基于精

确值流图的程序理解与需求确认方法的具体实现，包括抽象域、变迁规则的设计。在此基础上，介绍了变量关系分析方法、相关的程序理解方法与摘要的生成算法。

本文于第四章对上述问题的解决方法做了具体实现，并分别为两者设计了实验。通过实验验证了工具（算法）的有效性与实用性。随后，本文对实验中存在的误报、精度不足等问题做了具体分析并提出相关改进方案与技术展望。

第五章对本文所作工作进行了总结，并提出研究展望。

第 2 章　基于区间代数的整数缺陷检测

本章介绍基于区间代数的整数缺陷检测技术。旨在通过设计线性区间理论域，得到程序中整型变量的抽象表示，通过设计并借助变迁函数来模拟程序指令在抽象表示上的作用，实现程序的行为分析。在此基础上实现程序中数值导向型缺陷检测。

本章的结构安排如下：2.1小节通过一个区间分析的案例，介绍了当前基于区间分析存在的精度不足问题，引出本章研究内容；2.2小节介绍本章研究内容所需的预备知识，包括控制流自动机的概念与定义和可配置静态分析框架的概念；2.3小节依照层次结构分别介绍了具体的线性区间理论域的设计，并介绍抽象域上的变迁函数；最后，本章2.4小节利用设计的抽象域与变迁规则，给出程序的整型缺陷检测方法。我们于2.5小节通过具体例子介绍环状区间，并以此展开，介绍未来可能的研究方向。

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代数的整数缺陷检测技术，并在基于 CPAchecker[30] 的可配置的组合静态分析工具 Tsmart 上进行了实现。经过实验，基于本章工作实现的工具在 Juliet 测试集上的误报率小

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

图 2.1　基于区间代数的缺陷检测方法举例

于 3%，漏报率为 0%。

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。操作 $act(e) = (\%cmp, truth)$，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。操作 $act(e) = (\%phi, index)$，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。操作 $act(e) = (\%inst)$，其中 $\%inst$ 为所执行的指令。

为了便于后续算法介绍，我们对 LLVM-IR 中几种常见的指令做说明，现将其表示形式列举如下：

- Phi 指令：$z = \mathsf{phi}\ [y_1, x_1]\ [y_2, x_2] \cdots$。Phi 指令通常具有多个输入 $[y_i, x_i]$，其中，y_i 表示可能的取值，x_i 用于表示取值条件，当程序的执行经过 x_i 时，phi 指令的取值对应为 y_i。
- Alloca 指令：$x = \mathsf{alloca}\ type$。Alloca 指令用于为变量 x 申请一个能够存储类型为 $type$ 的内存区域，并将申请得到的内存地址返回给变量 x。
- 二元运算指令：$x = y \cdot_{op} z$，$\cdot_{op} \in \{+, -, \times, \div, and, or, xor, <<, >>\}$。二元运算指令用于对整型变量 y, z 进行运算，并将计算结果赋值给 x。
- Cast 指令：$x = \mathsf{cast}\ y\ \mathsf{to}\ type$。Cast 指令用于将一个整型变量转化为类型为 $type$ 的整型变量。通常使用它转换整型变量的位宽。
- Cmp 指令：$x = \mathsf{cmp}\ \mathsf{pre}\ y, z$。Cmp 指令用于根据条件 pre 判定变量 y 与 z 的关系，并将比较结果返回给 x。判断条件 pre 通常有如下几种：
  - eq: 判断 y 与 z 相等；
  - ne: 判断 y 与 z 不相等；
  - ugt: 无符号比较 $y > z$；
  - uge: 无符号比较 $y \geqslant z$；
  - ult: 无符号比较 $y < z$；
  - ule: 无符号比较 $y \leqslant z$；
  - sgt: 有符号比较 $y > z$；
  - sge: 有符号比较 $y \geqslant z$；
  - slt: 有符号比较 $y < z$；
  - sle: 有符号比较 $y \leqslant z$。
- Load 指令：$x = \mathsf{load}\ type\ y$。Load 指令用于从 y 所指向的内存区域中取类型为 $type$ 的数值，并将结果存放于变量 x 中。
- Store 指令：$\mathsf{store}\ type\ x, y$。Store 指令用于将类型为 $type$ 的数值 y 存放于由 x 所指向的内存区域中。

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[31] 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

1: $waitlist \leftarrow \{e_0\}, R \leftarrow \{e_0\}$
2: **while** $waitlist \neq \varnothing$ **do**
3:　　取出 *waitlist* 的首元素 e;
4:　　**for all** e' 满足 $e \rightsquigarrow e'$ **do**
5:　　UpdateRW($e', R, waitlist, merge, stop$);
6:　　**end for**
7: **end while**

Algorithm 3 UpdateRW 算法

```
1:  function UpdateRW(e′, R, waitlist, merge, stop)
2:      for all e′ ∈ R do
3:          e_new ← merge(e, e′);
4:          if e_new ≠ e′ then
5:              waitlist ← (waitlist ∪ {e_new}) \ {e′};
6:              R ← (R ∪ {e_new}) \ {e′};
7:          end if
8:      end for
9:      if ¬stop(e, R) then
10:         waitlist ← (waitlist ∪ {e});
11:         R ← R ∪ {e};
12:     end if
13: 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，其组成结构如图2.2所示。这里，位于最底层的是扩展的整数（RangeInteger）理论域，它是整数域的一个扩展，用于表示整型变量的某个可能取值。线性区间（Range）理论域表示整数区间，它是传统区间分析理论域的一个实现，以单独区间表示整型变量可能的取值范围。

在2.1小节中，我们以图2.1所示代码为例介绍了传统区间分析因区间个数问题所带来的精度丢失问题。为此，我们设计了线性多区间（MultiRange）理论域，它由若干个线性区间组成，具有更强的表示能力与更高的分析精度。更进一步地，面对具体的编程语言中整型变量所具有的位宽与符号性等属性，我们设计了位敏感的线性区间（SignRange）理论域，它是由线性多区间、位宽（Bitwidth）与符号性（Signedness）构成的三元组，使用 SignRange 可较好地刻画一个具有固定位宽的整型变量的可能取值范围，同时包含变量符号性信息。

在此基础上，我们引入区间状态（RangeState）抽象域，它描述了程序在某个状态下各个整型变量的取值信息，包括变量访问路径（AccessPath）、取值范围、符号性等。通过设计 RangeState 上的变迁关系（TransferRelation），我们可以得到程序在线性空间抽象域上的语义表示，为后续的整型缺陷分析提供支持。

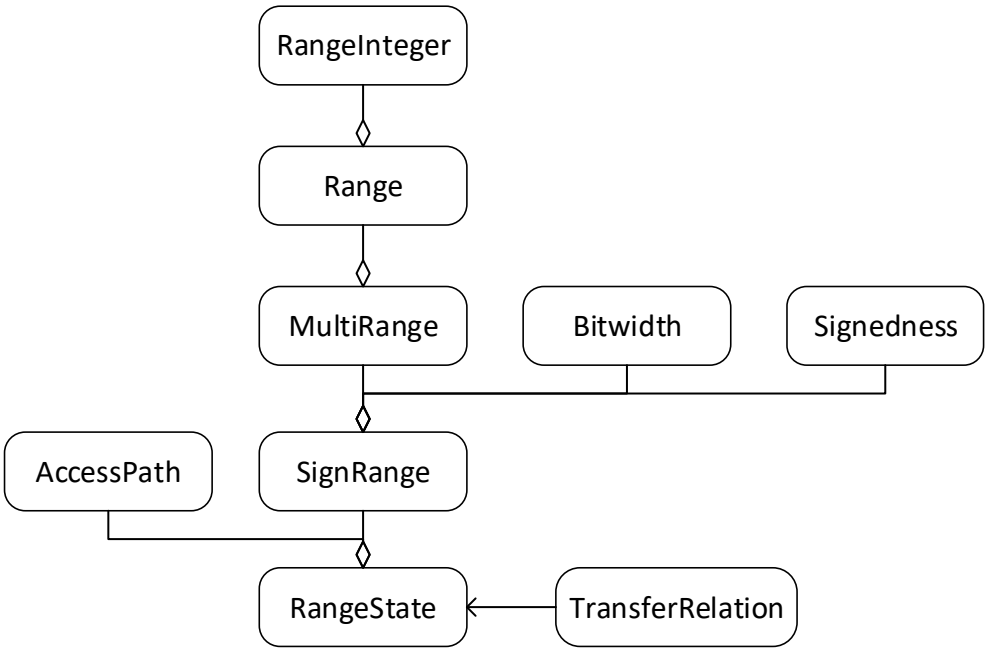

图 2.2　抽象域的组成

2.3.1　扩展的整数理论域

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

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	$x \bmod_I y := \begin{cases} x \bmod y & x, y \neq \pm\infty \\ x & x \geqslant 0 \wedge y = +\infty \\ y & x < 0 \wedge y = +\infty \\ nan & \text{其他} \end{cases}$
and	and_I	$x \cdot_{op_I} y := \begin{cases} x \cdot_{op} y & x, y \neq \pm\infty \\ nan & \text{其他} \end{cases}$
or	or_I	
xor	xor_I	
shl	$<<_I$	
shr	$>>_I$	

2.3.2　线性区间理论域

线性区间 Range 是一个整数区间，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	$[x_1, y_1]\ and_R\ [x_2, y_2] :=$ $\begin{cases} x_1\ and_I\ y_1 & x_1 = y_1 \wedge x_2 = y_2 \\ [0, y_1] & x_1 = y_1 \geqslant 0 \\ [0, y_2] & x_2 = y_2 \geqslant 0 \\ [0, +\infty] & x_1 \geqslant 0 \vee x_2 \geqslant 0 \\ [-\infty, -1] & y_1 < 0 \wedge y_2 < 0 \\ [-\infty, +\infty] & \text{其他} \end{cases}$
or	or_R	$[x_1, y_1]\ or_R\ [x_2, y_2] :=$ $\begin{cases} x_1\ or_I\ y_1 & x_1 = y_1 \wedge x_2 = y_2 \\ [-1, -1] & x_1 = y_1 = -1 \vee x_2 = y_2 = -1 \\ [\max(x_1, x_2), +\infty] & x_1 \geqslant 0 \wedge x_2 \geqslant 0 \\ [-\infty, -1] & y_1 < 0 \vee y_2 < 0 \\ [-\infty, +\infty] & \text{其他} \end{cases}$
xor	xor_R	$[x_1, y_1]\ xor_R\ [x_2, y_2] :=$ $\begin{cases} x_1\ xor_I\ y_1 & x_1 = y_1 \wedge x_2 = y_2 \\ [x_1, y_1] & x_2 = y_2 = 0 \\ [x_2, y_2] & x_1 = y_1 = 0 \\ [-\infty, +\infty] & x_1 \times_I x_2 < 0 \vee x_2 \times_I x_2 < 0 \\ [0, +\infty] & (x_1 \geqslant 0 \wedge x_2 \geqslant 0) \vee (y_1 < 0 \wedge y_2 < 0) \\ [-\infty, -1] & \text{其他} \end{cases}$

续下页

续表 2.2　Range 运算规则

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

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：

$$[[x,y]]_R := \begin{cases}[x,y] & x,y\text{是可比较的，且}x \leqslant y \\ [y,x] & x,y\text{是可比较的，且}x > y \\ \varnothing & \text{其他}\end{cases} \tag{2-3}$$

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 是可比较的，对于 $a = [x_1,y_1], b = [x_2,y_2] \in D_R$，有：

$$a < b \iff y_1 < x_2 \tag{2-5}$$

$$a > b \iff x_1 > y_2 \tag{2-6}$$

2.3.3　线性多区间理论域

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 的表示能力与 Range 相同。

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 $D_{MR} := \{[r_1, \cdots, r_n] \mid r_i \in D_R, r_1 < \cdots < r_n, 0 \leqslant n \leqslant L_{MR}\}$ 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：

$$[a_1, \cdots, a_m] \leqslant_{MR} [b_1, \cdots, b_n] \iff$$
$$\forall\, a_i,\ \exists\, b_j \text{ 使得 } a_i \leqslant_R b_j.\ (0 \leqslant i \leqslant m, 0 \leqslant j \leqslant n);$$

- D_{MR} 上的上确界操作（$\sqcup_{MR}$）定义为：

$$[a_1, \cdots, a_m] \sqcup_{MR} [b_1, \cdots, b_n] :=$$
$$[[a_1, \cdots, a_m, b_1, \cdots, b_n]]_{MR}.\ (0 \leqslant i \leqslant m, 0 \leqslant j \leqslant n)。$$

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 $r = [r_1, \cdots, r_n]$ 为一个 MultiRange，它是一个有序 Range 列表。我们在 MultiRange 上沿用定义在集合上的符号 $\times_{\cdot_{op}}$，即对于 $a, b \in D_{MR}$，有 $a \times_{\cdot_{op}} b = toList(set_{\cdot_{op}})$，其中 $set_{\cdot_{op}} = \{a_i \cdot_{op} b_j \mid a_i \in a, b_j \in b, 0 \leqslant i \leqslant m, 0 \leqslant j \leqslant n\}$。

表 2.3　MultiRange 运算规则

运算	符号	运算规则
negate	neg_{MR}	$neg_{MR}(r) := [[neg_R(r_1), \cdots, neg_R(r_n)]]_{MR}$
add	$+_{MR}$	$a +_{MR} b := [[a \times_{+_R} b]]_{MR}$
subtract	$-_{MR}$	$a -_{MR} b := [[a \times_{-_R} b]]_{MR}$
multiply	$\times_{MR}$	$a \times_{MR} b := [[a \times_{\times_R} b]]_{MR}$
divide	$\div_{MR}$	$a \div_{MR} b := [[a \times_{\div_R} b]]_{MR}$, $b \neq 0$
modular	mod_{MR}	$a \ \mathrm{mod}_{MR} b := [[a \times_{\mathrm{mod}_R} b]]_{MR}$, $b \neq 0$
and	and_{MR}	$a\, and_{MR}\, b := [[a \times_{and_R} b]]_{MR}$
or	or_{MR}	$a\, or_{MR}\, b := [[a \times_{or_R} b]]_{MR}$
xor	xor_{MR}	$a\, xor_{MR}\, b := [[a \times_{xor_R} b]]_{MR}$
shl	$<<_{MR}$	$a <<_{MR} b := [[a \times_{<<_R} b]]_{MR}$
shr	$>>_{MR}$	$a >>_{MR} b := [[a \times_{>>_R} b]]_{MR}$

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：$a = [[1, 3]]$，$b = [[2, 3]]$。$a \times_{MR} b$ 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 $a \times_{MR} b$ 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 是可比较的，对于 $a = [a_1, \cdots, a_m], b = [b_1, \cdots, b_n] \in D_{MR}$，有：

$$a < b \iff a_m < b_1 \tag{2-7}$$

$$a > b \iff a_1 > b_m \tag{2-8}$$

为了后续便利，在 MultiRange 上定义如下函数：

- $kill(mr, r) = mr \backslash r, mr \in D_{SR}, r \in D_R$ 用于删除 sr 所表示的整数范围中 r 对应的部分。例如：对于 $mr = [[0, 255]]$，有 $mr.kill([10, 100]) = [[0, 9], [101, 255]]$。
- $join(mr, r) = mr \cap r, mr \in D_{SR}, r \in D_R$ 用于得到 mr 所表示的整数范围中与 r 重合的部分。例如：对于 $mr = [[1, 10], [20, 30]]$，有 $mr.join([5, 25]) = [[5, 10], [20, 25]]$。

2.3.4　位敏感的线性区间理论域

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：

$$s_1 \sqcup_{sgn} s_2 := \begin{cases} s_2 & s_1 \leqslant_{sgn} s_2 \\ s_1 & s_2 \leqslant_{sgn} s_1 \end{cases}$$

引入符号性理论域意味着一个整型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 $|_{to}(s), s \in D_{sgn}$ 用于转化同一整数取值范围的不同 SignRange 表示，其算法在后续给出。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 $D_{SR} := \{(r, w, s) \mid r \in D_{MR}, w \in \mathbf{Z}^+, s \in \{Signed, Unsigned, Unknown\}\}$ 是一个半格：

- D_{SR} 上的最大元（$\top_{SR}$）是 $(\top_{MR}, \top_{Z^+}, s)$，最小元（$\bot_{SR}$）是 $(\bot_{MR}, \bot_{\mathbf{Z}^+}, s)$；

- D_{SR} 上的偏序关系（$\leqslant_{SR}$）定义为：

$$(r_1, w_1, s_1) \leqslant_{SR} (r_2, w_2, s_2) \iff (r_1 \,|_{to}(\mathsf{s}(r_2)) \leqslant_{MR} r_2) \wedge (w_1 \leqslant w_2);$$

- D_{SR} 上的上确界操作（$\sqcup_{SR}$）定义为：

$$(r_1, w_1, s_1) \sqcup_{SR} (r_2, w_2, s_2) := (r, w, s), \text{其中，} \begin{cases} s := \max(s_1, s_2) \\ w := \max(w_1, w_2) \\ r := r_1|_{to}(s) \sqcup_{MR} r_2|_{to}(s) \end{cases} 。$$

上述定义中，函数 $\mathsf{s}(sr), sr \in D_{SR}$ 用于取位敏感线性区间 sr 的 Signedness 值。类似的，有 $\mathsf{w}(sr), sr \in D_{SR}$ 用于取区间 sr 的 Bitwidth 值；$\mathsf{r}(sr), sr \in D_{SR}$ 用于取区间 sr 的 MultiRange 值。

值得注意的是，$\mathsf{r}(\top_{SR}) := \top_{MR}$。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型变量是否产生整型溢出。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 $top(sr), sr \in D_{SR}$ 和函数 $top(w, s), w \in \mathbf{Z}^+, s \in D_{sgn}$ 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：

$$top(sr) = top(w, s) := \begin{cases} [0, 2^w - 1]_R & s = U \\ [-2^{w-1}, 2^{w-1} - 1]_R & s \in \{S, N\} \end{cases} \tag{2-9}$$

作为补充的，定义函数 $bound(sr), sr \in D_{SR}$ 和 $bound(sr, w, s), sr \in D_S, w \in \mathbf{Z}^+, s \in D_{sgn}$ 用于得到取值范围在限定范围内的 SignRange：

$$bound(sr) := (sr \cap top(sr), \mathsf{w}(sr), \mathsf{s}(sr)) \tag{2-10}$$

$$bound(sr, w, s) := (sr|_{to}(s) \cap top(w, s), w, s) \tag{2-11}$$

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: $nRanges \leftarrow \varnothing$
3: **for all** $range$ in $range(signRange)$ **do**
4:　　**if** $range < 0$ **then**
5:　　$nRanges \leftarrow nRanges \cup (range +_R convertVal)$
6:　　**else if** $range > -1$ **then**

7: $nRanges \leftarrow nRanges \cup range$
8: **else**
9: $nRanges \leftarrow nRanges \cup ([range.x, -1] +_R convertVal) \cup [0, range.y]$
10: **end if**
11: **end for**
12: **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

1: $topVal \leftarrow top(bitwidth, Signed)$
2: $convertVal \leftarrow top(bitwidth, Unsigned).y + 1$
3: $nRanges \leftarrow \varnothing$
4: **for all** $range$ in $range(signRange)$ **do**
5: **if** $range < topVal + 1$ **then**
6: $nRanges \leftarrow nRanges \cup range$
7: **else if** $range > topVal$ **then**
8: $nRanges \leftarrow nRanges \cup (range -_R convertVal)$
9: **else**
10: $nRanges \leftarrow nRanges \cup [range.x, topVal]$
11: $nRanges \leftarrow nRanges \cup ([topVal + 1, range.y] -_R convertVal)$
12: **end if**
13: **end for**
14: **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 $a, b \in D_{SR}$ 参与的二元运算 $a \cdot_{op} b$ 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：

$$w' = \mathsf{w}(a) = \mathsf{w}(b) \leftarrow \max(\mathsf{w}(a), \mathsf{w}(b)) \tag{2-12}$$

$$a \cdot_{op} b := \begin{cases} a|_{to}(U) \cdot_{op} b|_{to}(U) & \mathsf{s}(a) = U \vee \mathsf{s}(b) = U \\ a \cdot_{op} b & \mathsf{s}(a) = N \wedge \mathsf{s}(b) = N \\ a|_{to}(S) \cdot_{op} b|_{to}(S) & \text{其他} \end{cases} \tag{2-13}$$

（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：

$$a \cdot_{op} b := a|_{to}(S) \cdot_{op} b|_{to}(S) \tag{2-14}$$

（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：

$$a \cdot_{op} b := a \cdot_{op} b|_{to}(U) \tag{2-15}$$

SignRange 的计算规则如表2.4所示。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 $w' \in \mathbf{Z}^+, s' \in \boldsymbol{D}_{sgn}$ 均为操作数按照 Unify Signedness 规则统一符号性、位宽后得到的值。

表 2.4　SignRange 运算规则

运算	符号	运算规则
negate	neg_{SR}	$neg_{SR}(r) := \begin{cases} (neg_{MR}(\mathsf{r}(r)), \mathsf{w}(r), \mathsf{s}(r)) & \mathsf{s}(r) \in \{S, N\} \\ neg_{SR}(r\,|_{to}(S))\,|_{to}(U) & s = U \end{cases}$
add	$+_{SR}$	$a +_{SR} b := (\mathsf{r}(a) +_{MR} \mathsf{r}(b), w', s')$
subtract	$-_{SR}$	$a -_{SR} b := (\mathsf{r}(a) -_{MR} \mathsf{r}(b), w', s')$
multiply	$\times_{SR}$	$a \times_{SR} b := (\mathsf{r}(a) \times_{MR} \mathsf{r}(b), s', s')$
divide	$\div_{SR}$	$a \div_{SR} b := (\mathsf{r}(a) \div_{MR} \mathsf{r}(b), w', s'), b \neq 0$
modular	mod_{SR}	$a \mod_{SR} b := (\mathsf{r}(a) \mod_{MR} \mathsf{r}(b), w', s'), b \neq 0$
and	and_{SR}	$a\ and_{SR}\ b := (\mathsf{r}(a)\ and_{MR}\ \mathsf{r}(b), \mathsf{w}(a), \mathsf{s}(a))$
or	or_{SR}	$a\ and_{SR}\ b := \mathsf{r}(a)\ or_{MR}\ \mathsf{r}(b), \mathsf{w}(a), \mathsf{s}(a))$
xor	xor_{SR}	$a\ and_{SR}\ b := (\mathsf{r}(a)\ xor_{MR}\ \mathsf{r}(b), \mathsf{w}(a), \mathsf{s}(a))$
shl	$<<_{SR}$	$a <<_{SR} b := (\mathsf{r}(a) <<_{MR} \mathsf{r}(b'), \mathsf{w}(a), s')$, in which $b' \leftarrow b.kill([\mathsf{w}(a), +\infty]).kill([-\infty, -1])$
shr	$>>_{SR}$	$a <<_{SR} b := (\mathsf{r}(a) >>_{MR} \mathsf{r}(b'), \mathsf{w}(a), s')$, in which $b' \leftarrow b.kill([\mathsf{w}(a), +\infty]).kill([-\infty, -1])$

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 是可比较的，对于 $a, b \in D_{SR}$，有：

$$a < b \iff \mathsf{r}(a') < \mathsf{r}(b') \tag{2-16}$$

$$a > b \iff \mathsf{r}(a') > \mathsf{r}(b') \tag{2-17}$$

其中，a', b' 为根据 Unify Signedness 规则（2-12）更新符号性和位宽后得到的 SignRange。

为了后续方便，在 SignRange 上定义函数 $high$ 用于获取当前线性区间中的最大值，函数 low 用于获取当前线性区间中的最小值。对于 $sr = ([[1, 5], [20, 100]], 8, S) \in D_{SR}$，有 $high(sr) = 100, low(sr) = 1$。

2.3.5　区间状态理论域

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，本节介绍的区间状态 RangeState 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。

区间状态 RangeState := AccessPath ↦ SignRange 是一个从 AccessPath 到 SignRange 的映射，用符号 Γ 表示。它描述了在某个程序状态下，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。若 AccessPath 的理论域为 D_{ap}，那么对于 $x \in D_{ap}$，记 $\Gamma(x)$ 为 x 在 Γ 上的映射，即访问路径为 x 的位敏感的线性区间表示。

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： 定义区间状态理论域 $D_S := \{\Gamma := x \mapsto y \mid x \in D_{ap}, y \in D_{SR}\}$ 是一个半格：

- D_S 上的最大元 $\top_S := x \mapsto \top_{SR}, x \in D_{ap}$，最小元 $\bot_S := x \mapsto \bot_{SR}, x \in D_{ap}$；
- D_S 上的偏序关系（$\leqslant_S$）定义为：

$$a \leqslant_S b \iff \mathsf{aps}(a) \subseteq \mathsf{aps}(b) \wedge \forall x \in \mathsf{aps}(a), a(x) \leqslant_{SR} b(x);$$

- D_S 上的上确界操作（$\sqcup_S$）定义为：

$$a \sqcup_S b := x \mapsto y, x \in \mathsf{aps}(a) \cup \mathsf{aps}(b), y = a(x) \sqcup_{SR} b(x)。$$

上述定义中，函数 $\mathsf{aps}(\Gamma) := \{x \in D_{ap} \mid \Gamma(x) \neq \bot_{SR}\}$ 表示区间状态 Γ 的访

问路径集合。类似的，定义函数 $\mathsf{ranges}(\Gamma) := \{\Gamma(x) \mid x \in \mathsf{aps}(x)\}$ 为取状态区间 Γ 的区间表示 SignRange 的集合。

区间状态 Γ 通常与 CFA 上的状态节点 n 是关联的，本节定义的区间状态 RangeState 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： 记 $\rightsquigarrow_S := D_S \times Ops \times D_S$ 为区间状态理论域 D_S 上的变迁关系。其中，Ops 表示变迁关系上的操作。我们用 $\Upsilon := act(\rightsquigarrow_S)$ 来表示 Ops 所具体执行的动作，称之为变迁函数。对于 $s \in D_S$，$s' = \Upsilon(s)$ 表示区间状态 s 在应用变迁函数 Υ 后得到的新的区间状态。

在定义区间状态理论域上的变迁关系后，我们可以使用区间状态 Γ 和区间状态上的变迁关系 $\rightsquigarrow_S$ 来抽象描述 CFA 上程序状态和程序状态之间的变迁关系。由于 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），$\rightsquigarrow_S$ 对应的变迁函数 $\Upsilon := act(\rightsquigarrow_S)$ 随之有不同的定义（下列定义中为方便起见，整型变量的 AccessPath 均由其在指令中的变量名代替）：

1. 对于空白边 $e \in BlankEdge$，$act(e) = \varepsilon$，其变迁关系为：

$$\frac{n_1 \xrightarrow{\varepsilon} n_2, S_2 = S_1}{(n_1, S_1) \rightarrow (n_2, S_2))} \quad \text{T-R-EMPTY} \tag{2-18}$$

   其中，$n_1, n_2 \in N$ 表示 CFA 上的状态节点，S_1, S_2 则为 n_1, n_2 在区间状态域 D_S 上的抽象表示。为方便起见，如无特殊说明，后续定义中的 n_1, n_2, S_1, S_2 同此含义。

2. 对于假设边 $e \in AssumeEdge$，$act(e) = (\%cmp, truth)$。其中，$truth \in \{true, false\}$。其变迁关系为：

$$\frac{n_1 \xrightarrow{[\%cmp=truth]} n_2, S_2 = \mathsf{refineVals}(S_1, \%cmp, truth)}{(n_1, S_1) \rightarrow (n_2, S_2)} \quad \text{T-R-ASSUME} \tag{2-19}$$

   其中，函数 refineVals 的操作是将区间状态 S_1 中访问路径为 $\%cmp$ 的取值范围改为 $truth$ 所代表的值，同时依照比较逻辑更改比较操作的两个操作数的范围。

3. 对于形如 $z = \mathsf{phi}\ [y_1, x_1]\ [y_2, x_2]$ 的 Phi 边（PhiEdge），其变迁关系为：

$$\frac{n_1 \xrightarrow{z=\mathsf{phi}\ [y_1,x_1]\ [y_2,x_2]} n_2, S_2 = S_1[z \,/\, Y]}{(n_1, S_1) \rightarrow (n_2, S_2)} \quad \text{T-R-PHI} \tag{2-20}$$

在上式中，当 $S_1(x_1)=0$ 时，$Y:=S_1(y_1)$ 当 $S_1(x_2)=0$ 时，$Y:=S_1(y_2)$。

4. 对于形如 $x=$ **alloca** *type* 这样的指令边（InstructionEdge），约定 w 为 *type* 类型变量在计算机上存储所用位宽，s 为 *type* 类型变量的符号性，则变迁关系可描述为：

$$\frac{n_1 \xrightarrow{x=\textbf{alloca } type} n_2, S_2 = S_1[*x \,/\, (\top_{MR}, w, s)]}{(n_1, S_1) \to (n_2, S_2)} \quad \text{T-R-ALLOCA} \qquad (2\text{-}21)$$

5. 对于形如 $x=y\cdot_{op}z$ 这样的指令边，其变迁关系为：

$$\frac{n_1 \xrightarrow{x=y\cdot_{op}z} n_2, S_2 = S_1[x \,/\, S_1(y)\cdot_{op_{SR}} S_1(z)]}{(n_1, S_1) \to (n_2, S_2)} \quad \text{T-R-OP} \qquad (2\text{-}22)$$

此处 $\cdot_{op_{SR}}$ 为 $\cdot_{op}$ 在 D_{SR} 上的自然扩展。

6. 对于形如 $x=$ cast y to *type* 这样的指令边，其变迁关系为：

$$\frac{n_1 \xrightarrow{x=\text{cast } y \text{ to } type} n_2, S_2 = S_1[x \,/\, bound(S_1(x), w, s)]}{(n_1, S_1) \to (n_2, S_2)} \quad \text{T-R-CAST} \qquad (2\text{-}23)$$

其中，w 为 *type* 类型变量在计算机上存储所用位宽，s 为 *type* 类型变量的符号性。后续定义中，若无特殊说明，对于指令中出现的 *type* 类型，将继续沿用符号 w, s。

7. 对于形如 $x=$ cmp pre y, z 这样的指令边，其变迁关系为：

$$\frac{n_1 \xrightarrow{x=\text{cmp pre } y,z} n_2, S_2 = S_1[x \,/\, \text{compare}(S_1(y), S_1(z))]}{(n_1, S_1) \to (n_2, S_2)} \quad \text{T-R-CMP} \qquad (2\text{-}24)$$

其中，函数 compare$(a,b), a, b \in D_{SR}$ 用于在 D_{SR} 上依据定义的比较规则得到的比较结果。

8. 对于形如 $x=$ load *type* y 这样的指令边，其变迁关系为：

$$\frac{n_1 \xrightarrow{x=\text{load } y} n_2, S_2 = S_1[x \,/\, bound(S_1(*y), w, s)]}{(n_1, S_1) \to (n_2, S_2)} \quad \text{T-R-LOAD} \qquad (2\text{-}25)$$

9. 对于形如 store *type* x, y 这样的指令边，其变迁关系为：

$$\frac{n_1 \xrightarrow{\text{store } type\ x,y} n_2, S_2 = S_1[*y \,/\, bound(S_1(x), w, s)]}{(n_1, S_1) \to (n_2, S_2)} \quad \text{T-R-STORE} \qquad (2\text{-}26)$$

2.3.6　精度优化

在2.3.3和2.3.4小节中，通过举例我们知道，对于两个单独的区间在进行乘法、位移等操作时会产生一定程度的精度丢失。而造成精度丢失的原因是，单独区间如 $r = [x, y]$ 只能描述从 x 到 y 的全部整数。而乘法、位移等操作会拉大区间内整数之间的距离，从而导致精度的丢失。

为了解决这一问题，我们于2.3.3小节提出了线性多区间的概念，它定义了一个由多个线性区间组成的列表，可以更加精确的表示整数的取值范围。但是在定义 MultiRange 的乘法等运算时，其运算规则仅仅是组合调用 Range 的运算规则，并没有深入优化 Range 与 Range 之间计算的精度。

借助 MultiRange 本身即可表示多个区间这一良好的性质，理想情况下我们可以将线性区间所表示的所有整数打散并分别进行运算，再重新将计算结果组合为多区间列表。例如，对于 $a = [[1,3]] \in D_{MR}, b = [[2,3]] \in D_{MR}$ 两个线性区间，我们可将其分别表示为 $a' = \{1,2,3\}, b' = \{2,3\}$。则区间上的运算 $a \times_{MR} b$ 的运算结果可以通过集合上的运算结果 $a' \times_{\times_I} b' = \{2,3,4,6,9\}$ 导出，则 $a \times_{MR} b = [[2,4],[6,6],[9,9]]$。

以上思路实际上是线性区间在具体域上的运算还原。其计算过程实际上是对整型变量的所有可能取值的枚举，这违背了抽象解释技术的初衷。在具体应用中，变量的可能取值范围通常是很大的，仅区间 $x = [[1,10],[20,100]]$ 即表示了 90 个可能的整数取值。由此可知，当区间与区间进行运算时，尽管枚举区间的所有可能取值并分别运算提高了分析的精度，但也伴随着分析效率的极大下降。

为了兼顾分析效率与分析精度，本节给出的优化方法的主要思想是在合理的区间长度下将区间打散并对区间内每个整数分别进行运算，当区间内包含的整数过多时，直接进行原始的区间运算。这样做的原因基于整型变量在程序中的使用通常符合如下几类情况：

- 作为输入变量，通过条件判断语句逐步缩小并确认整型变量的可表示范围；
- 作为普通变量参与运算，变量初始化时可能通过常量赋值也可能通过变量复制赋值。

其中，本节提出的优化的具体应用多针对于取值范围很小的变量，如上述情形中的通过常量（如常数、枚举值等）赋值初始化的整型变量、通过比较操作得到的布尔型变量、以及具体出现的小区间范围的变量等。

我们在 MultiRange 上补充定义函数 $exsits$，对于一个整数 $x \in D_R$ 和线性区间 $mr \in D_{MR}$，$mr.exsists(x)$ 用于判定 mr 中是否存在整数 x；定义 $\mathsf{cnt}(mr)$ 用于获取 mr 表示的线性区间内包含的整数个数；定义函数 $\mathsf{disperse}(mr) := \{r = [x,x] \in$

$D_R \mid mr.exsits(x)\}$ 用于得到 r 对应的整数表示集合。

则对于 MultiRange 上二元运算 $z := x \cdot_{op} y$，其中 $\cdot_{op} \in \{multiply, and, or, xor, shl, shr\}$，其优化后的运算规则可表述为：

$$x \cdot_{op} y := \begin{cases} [[\mathrm{disperse}(x) \times_{op_R} \mathrm{disperse}(y)]]_{MR} & \mathrm{cnt}(x), \mathrm{cnt}(y) \leqslant L_{MR} \\ x \cdot_{op} y & \text{其他} \end{cases} \tag{2-27}$$

其中，操作符 $\cdot_{op_R}$ 为操作符 $\cdot_{op}$ 在理论域 D_R 上的对应版本。另一方面，在位敏感的线性区间 SignRange 上，由于其计算规则的定义是 MultiRange 的扩展，因此无需修改，直接应用优化后的计算规则即可。

2.4　基于区间代数的整数缺陷检测方法

整型缺陷按照类型通常可分为两类：一类是数值型缺陷，这包括比如整型的上溢出、下溢出、整型的除零异常等；另一类是由数值型缺陷所引发的其他类型缺陷。典型的如内存泄漏、缓冲区溢出等。这类问题相较于单纯的数值型缺陷往往危害更大。以上两类缺陷可统称为数值导向型缺陷。

其中，整型的溢出缺陷是由程序变量在计算机中有限长度的二进制表示造成的。例如，在 C 语言中，一个类型为 *signed int* 的变量在计算机上通常存储为 32 位二进制数，这意味着该变量的表示是有范围的。当欲表达的整型数字超过表达的最大范围，就会出现整型溢出。由于 C 语言中对溢出的行为是未定义的，部分机器对上溢出的数字解释为负数，由此可能会出现潜在的程序逻辑紊乱。

上述数值导向型缺陷均可在标准错误枚举（Common Weakness Enumeration，简称 CWE）中找到定义，如表2.5所示。

表 2.5　CWE 定义的数值导向型缺陷

编号	CWE 类型	解释
190	Integer Overflow or Wraparound	整数运算的结果小于数学算术期望值
191	Integer Underflow (Wrap or Wraparound)	整数运算的结果大于数学算术期望值
126	Buffer Overread	缓冲区读取位置超过缓冲区上界
127	Buffer Underread	缓冲区读取位置未达到缓冲区下界
121	Stack-based Buffer Overflow	对超出栈内存对象上界的位置写入数据
122	Heap-based Buffer Overflow	对超出堆内存对象上界的位置写入数据
124	Buffer Underflow	对未达到内存对象下界的位置写入数据
369	Divide by Zero	除零异常

续下页

续表 2.5　CWE 定义的数值导向型缺陷

编号	CWE 类型	解释
401	Memory Leak	堆内存泄露

本节介绍的基于区间代数的整型缺陷检测方法可直接应用于检测数值型缺陷，即表2.5中所对应的 CWE-190、CWE-191 和 CWE-369。对于由数值型缺陷导致的其他缺陷，可使用本节提出的方法配合其他 CPA 算法加以实现。

在小节2.3.5中，我们介绍了控制流自动机 CFA 的抽象表示 RangeState 以及其上与 CFA 边对应的状态变迁关系。本节所述的缺陷检测算法实际上是区间状态 RangeState 抽象域的一个具体应用。要检测程序是否出现整型溢出、除零异常等缺陷，我们要在原有的区间状态变迁规则基础上，设计缺陷检测规则。

通过分析，易知如需检测整型的溢出和除零异常缺陷，仅需关注程序中的整型变量在二元运算或位宽转换指令时的状态即可。

对于 CFA 上通过边 e 连接的两个状态节点 n_1, n_2。当 e 是形如 $x = y \cdot_{op} z$，其中 $\cdot_{op} \in \{add, subtract, multiply, shl, shr\}$ 这样的操作边（InstructionEdge）时，我们对其应用上溢出和下溢出检测规则：

$$\mathsf{isOverflow}(n_1, e) \iff top(X) < high(X), X = S_1(y) \cdot_{op_{SR}} S_1(z) \tag{2-28}$$

$$\mathsf{isUnderflow}(n_1, e) \iff low(X) < top(X), X = S_1(y) \cdot_{op_{SR}} S_1(z) \tag{2-29}$$

其中，操作符 $\cdot_{op_{SR}}$ 是操作符 $\cdot_{op}$ 在 SignRange 上的对应版本。S_1, S_2 是状态节点 n_1, n_2 在区间状态理论域 D_S 上的抽象表示。类似的，当 $\cdot_{op} \in \{divide, mod\}$ 时，我们应用除零检测规则：

$$\mathsf{isDivideByZero}(n_1, e) \iff X.exsists(0), X = S_1(y) \cdot_{op_{SR}} S_1(z) \tag{2-30}$$

其中，函数 $exsists$ 为 MultiRange 上的自然扩展。我们将上述检测规则描述为算法7。对 CFA 上的每条边，应用 checkEdge 判定其是否存在整型缺陷，并生成缺陷报告。

Algorithm 7 checkEdge for CWE190, 191 and 369

Require: $e \in E, n_1, n_2 \in N$，其中 e 是 CFA 中连接节点 n_1, n_2 的有向边

1: **if** $type(e) = inst \wedge act(e) = \cdot_{op}$ **then**
2:　　**if** $\cdot_{op} \in \{add, subtract, multiply, shl, shr\} \wedge$ Wrapped is false **then**
3:　　　　**if** isOverflow(n_1, e) **then**
4:　　　　　　report an IntegerOverflow weakness at (n_1, e)

```
5:          else if isUnderflow(n1, e) then
6:              report an IntegerUnderflow weakness at (n1, e)
7:          end if
8:      end if
9:      if ·op ∈ {divide, mod} ∧ isDivideByZero(n1, e) then
10:         report a DivideByZero weakness at (n1, e)
11:     end if
12: end if
```

上述算法中，Wrapped 符号来自 LLVM-IR 指令，对应于二元运算中的 NSW 与 NUW 符号，用于表征运算是否允许因运算上溢出、下溢出所带来的数值翻转。

2.5　环状区间理论域简介

尽管本章提出的位敏感线性区间抽象域 SignRange 能够精确表示整型变量的可能取值范围，但在具体编码实现中会相对复杂。本节将用少量篇幅介绍环状区间 WrappedRange，它将作为本章未来的研究内容。

在2.3.4小节中，为了实现对整型变量符号性的表示，我们引入了符号性 Signedness 与位敏感区间 SignRange 来扩充现有线性区间在表示性上的不足，并在此基础上实现了对程序状态的抽象表示。但实际上这样的区间设计存在着一定程度的冗余，原因是一个整型变量不论是有符号数还是无符号数，其在计算机上的存储模式是一致的：同一个二进制串既可以按有符号数来解析，又可以按无符号数来解析。但对于 SignRange 而言，同一个整型变量对应的二进制串按其符号性解析方式不同，对应着两个不同的 SignRange。这一方面造成了表示性的冗余，增加静态分析所用的内存，另一方面也增加了编码实现的难度。

为了解决这一问题，本节介绍一种线性区间表示的替代方案——环状区间 WrappedRange[32-33]，它最早由 Navas 等人于 2012 年提出。环状区间的实现思路基于整型变量在计算机中的二进制表示与解析，它实质上模拟了以二进制串存储的整型变量在计算机中的表示与操作。它解决了 SignRange 对同一个整型变量因符号性不同而产生的不同抽象解释，一定程度上降低了编码实现的难度。

我们以一个具体例子介绍 WrappedRange 的运作方式，其形式化的定义与详细的计算规则将作为本文的未来研究工作内容，这里不做具体论述。

环状区间通常可表示为由两个长度为 w 的二进制向量 x, y 构成的二元组 $[x, y]$。其中，二进制向量 x, y 可理解为整型数字在计算机中的存储模式。例如，

对于 $x = 1001$ 表示位宽为 4 的整型数字，它既可以按有符号数解释为-7，又可按无符号数解释为 9。

那么，对于一个环状区间 $r = [x, y]$ (其中 $x, y \in \{0,1\}^w$ 且 $x \neq y + 1$)，它所表示的具体整数取值范围是：

$$r = [x, y] := \begin{cases} \{x, \cdots, y\} & x \leqslant y \\ \{0^w, \cdots, y\} \cup \{x, \cdots, 1^w\} & \text{其他} \end{cases} \tag{2-31}$$

为了便于对环状区间的理解，我们以长度 $w = 4$ 的向量构成的环状区间为例介绍环状区间的性质与运算。如图2.3所示，将 $\{0,1\}^w$ 所表示的向量以环状形式排列，则环状区间 $r = [x, y], x, y \in \{0,1\}^4, x \neq y + 1$ 所表示的取值可理解为图2.3上以 x 为起点，以 y 为终点顺时针访问得到的所有向量。

例如，对于 $r = [0110, 1001]$，其表示范围是 $\{0110, 0111, 1000, 1001\}$，对应有符号数为 $\{6, 7, -8, -7\}$，对应的无符号数为 $\{6, 7, 8, 9\}$。由于区间是环状定义的，因此允许区间的左值大于右值，对应的其区间为图2.3中超过 180° 的扇形区域。例如 $r = [1101, 0001]$ 的表示范围是 $\{1101, 1110, 1111, 0000, 0001\}$，对应的有符号数为 $\{-3, -2, -1, 0, 1\}$，对应的无符号数为 $\{13, 14, 15, 0, 1\}$。

那么，对于环状区间 $r_1 = [x_1, y_1], r_2 = [x_2, y_2]$，其加法规则可以表述如下：

$$r_1 + r_2 = \begin{cases} [x_1 + x_2, y_1 + y_2] & \mathsf{len}(r_1) + \mathsf{len}(r_2) \leqslant 2^w \\ [0^w, 1^w] & \text{其他} \end{cases} \tag{2-32}$$

类似地，我们可在未来进一步完善环状区间相关的计算规则。

通过环状区间，我们可以将有符号数、无符号数统一进行表示。但与线性区间 Range 相同地，它们均面临着无法精细刻画多段区间的情形。在后续工作中，我们将会进一步研究基于环状区间 WrappedRange 的多区间理论域，进一步提升其表示能力。

2.6　本章小结

本章介绍了基于区间代数的整型缺陷分析方法，其核心是理论域的设计。本章先介绍了整体的区间设计，随后依次对各个抽象层面进行介绍，并结合控制流自动机，设计了整型变量的缺陷检测规则。最后，在基于 Tsmart-V3 静态分析框架上做了实现，并通过实验验证了其有效性。

在2.3.1小节，我们介绍了扩展的整数 RangeInteger，相比于整数，其具体定义了 $+\infty$ 和 $-\infty$ 的概念，同时，我们介绍了有 $-\infty, +\infty$ 参与的整数运算规则，为后

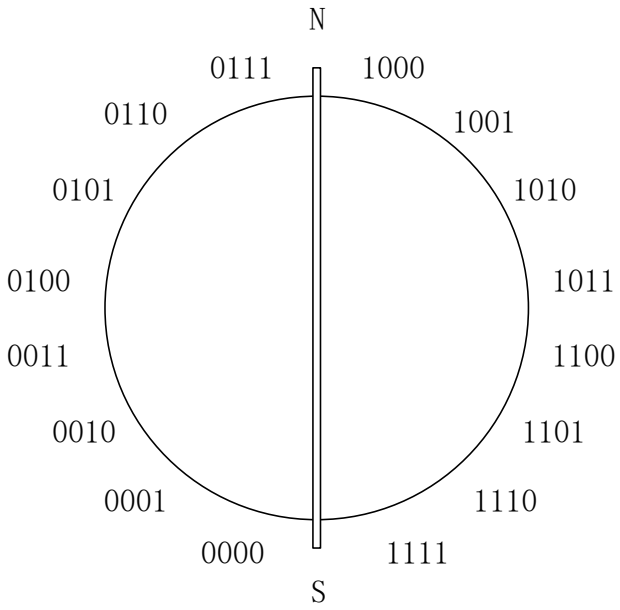

图 2.3　以环状形式列出的所有 $\{0,1\}^4$ 元素

续介绍区间抽象域奠定了基础。在2.3.2小节，介绍了基于扩展整数的区间 Range，它以单独区间的形式表示了一个整型变量的可能取值范围，并首次定义整数区间上的运算规则。

随后，在2.3.3和2.3.4小节，我们介绍了线性多区间 MultiRange 与位敏感的线性多区间 SignRange，它们的提出分别解决了单独的 Range 区间精度不足以及无法描述符号性的问题。随后为了进一步提升多区间在区间长度较小时的精度，提出了基于拆分-合并思想的精度提升方法。

最后，结合 CPA，提出了基于区间代数的缺陷检测方法。该方法基于区间状态 RangeState 抽象域，它是 CFA 上程序状态节点的抽象。通过设计抽象域与变迁规则，可以得到整型变量在程序指令执行前后的取值区间，从而实现整型缺陷分析。在第4.2章节，我们将通过实验，验证基于区间代数的缺陷检测方法的有效性。

值得一提的是，本章设计的位敏感的线性多区间 SignRange 是对整型变量的具体含义做抽象，其并未反应变量在计算机中的实际存储。实际上，针对存储于计算机上的 0-1 二进制串，我们以有符号数和无符号数对其进行解析时会得到两个不同的 SignRange 抽象表示，这会给该理论域的具体编程实现带来一定的困难，并造成一定程度的表示冗余。为了解决这一问题，本章于2.5介绍了环状区间的概念，并希望以此解决上述问题。该理论域的具体实现与优化将作为今后研究的方向。

第 3 章　基于值流图的整型变量关系分析与缺陷检测

本章介绍基于值流图的整型变量关系分析与缺陷检测方法，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。

本章结构安排如下：3.1小节简要归纳了当前程序理解技术，提出本章工作的实际意义与必要性；3.2小节通过一个需求与实现的例子具体介绍了本章工作如何帮助开发人员快速进行软件确认；3.3及随后三个小节具体介绍了算法的原理与实现方法。最后3.7小节对本章工作做了总结。

3.1　引言

软件验证与确认是软件研发过程中十分重要的步骤。其中，对软件确认过程来说，人们除了使用测试的手段逐条对软件行为进行验证以外，最常用的手段便是在理解程序代码的基础上对程序是否符合软件需求做判断。但是，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 [34] 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[35]。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程序理解相关工作时间。

程序理解[36-37] 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[38] 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[39] 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。

本章中，我们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。

3.2　案例分析

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。本节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。

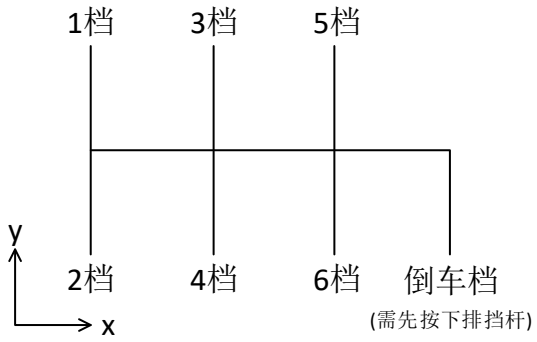

图 3.1　变速器排挡杆

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3.1所示。现要求实现相关函数完成图3.1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 3.1　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3.1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3.1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3.1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。

表 3.1　根据代码3.1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	$\mathrm{dv} \neq 0$	$(160 \leq y \leq 780) \wedge (dv \neq 0)$
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	$(x > 690) \wedge (y < 160) \wedge (dv = 0)$	$(x > 690) \wedge (y < 160) \wedge$ $(og \neq -2 \wedge og \neq -1)$

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。

3.3　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3.2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。

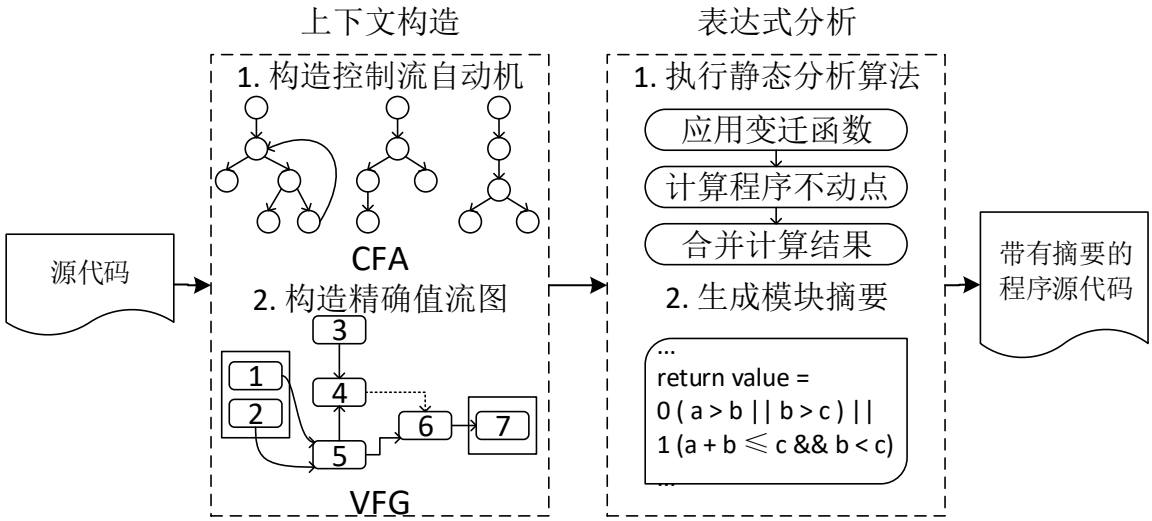

图 3.2　算法的工作流程图

3.4　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 3.2　代码样例

```
1 int main() {
2   int x = 0, y = 1;
3   int *a;
4   int p = nondet_int(); // 将p初始化为不确定的整数
5   if (p)
6     a = &x;
```

```
7     else
8       a = &y;
9     return *a;
10  }
```

以代码3.2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：$a \leftarrow \{(\&x, p \neq 0), (\&y, p = 0)\}$。

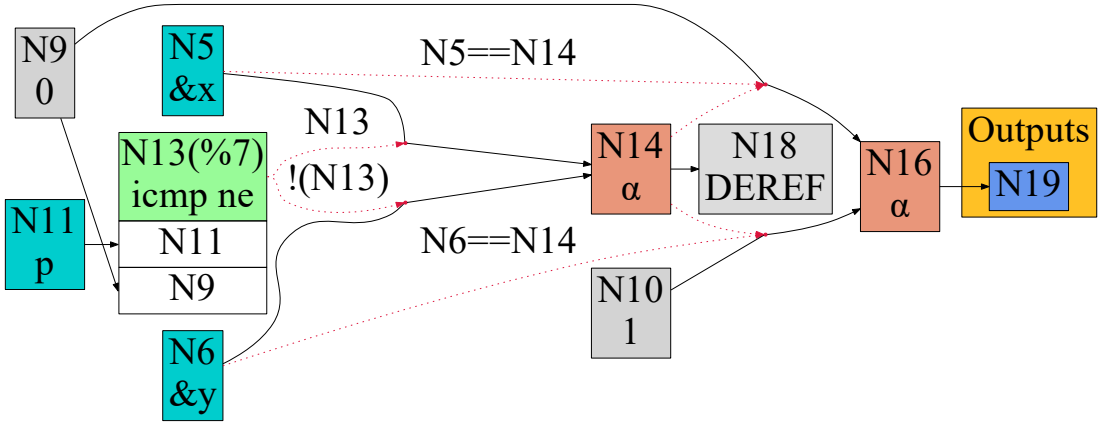

图 3.3　根据代码3.2生成的值流图

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3.3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3.3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3.3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3.3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3.3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3.3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。

3.5　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8所示。

Algorithm 8 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: $\quad cur \leftarrow$ peek($pending$)
5: $\quad valueSet \leftarrow$ updateNodeValue(cur)
6: $\quad$ **if** $valueSet = \varnothing$ **then**
7: $\quad\quad blockSet \leftarrow blockSet \cup cur$
8: $\quad$ **else if** $valueSet$ is an update for cur **then**
9: $\quad\quad pending \leftarrow pending \cup successor(cur)$
10: $\quad\quad$ move each $node$ from $blockSet$ to $pending$ which is unblocked due to $valueSet$
11: $\quad$ **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3.2所示。

表 3.2　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	$\left\{ \left(\begin{array}{l} operatorValue \\ (opr(cur), val(opd_1), ..., val(opd_i)) \\ , cond(opd_1) \wedge ... \wedge cond(opd_i) \end{array} \right) \middle\| \begin{array}{c} opd_i \in T(pre_i), \\ pre_i = pred(cur)[i] \end{array} \right\}$
JoinNode (Loop)	$\{(\mu, true) \mid \mu \leftarrow \mu \cup val(pred(cur))\}$
JoinNode (Normal)	$\{(val(pre_i), cond(pre_i) \wedge pcond(pre_i)) \mid pre_i \leftarrow pred_i(cur)\}$

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。

3.6　程序理解与需求确认

以变速器档位控制代码（代码3.1）为例，其对应的值流图如图3.4所示。

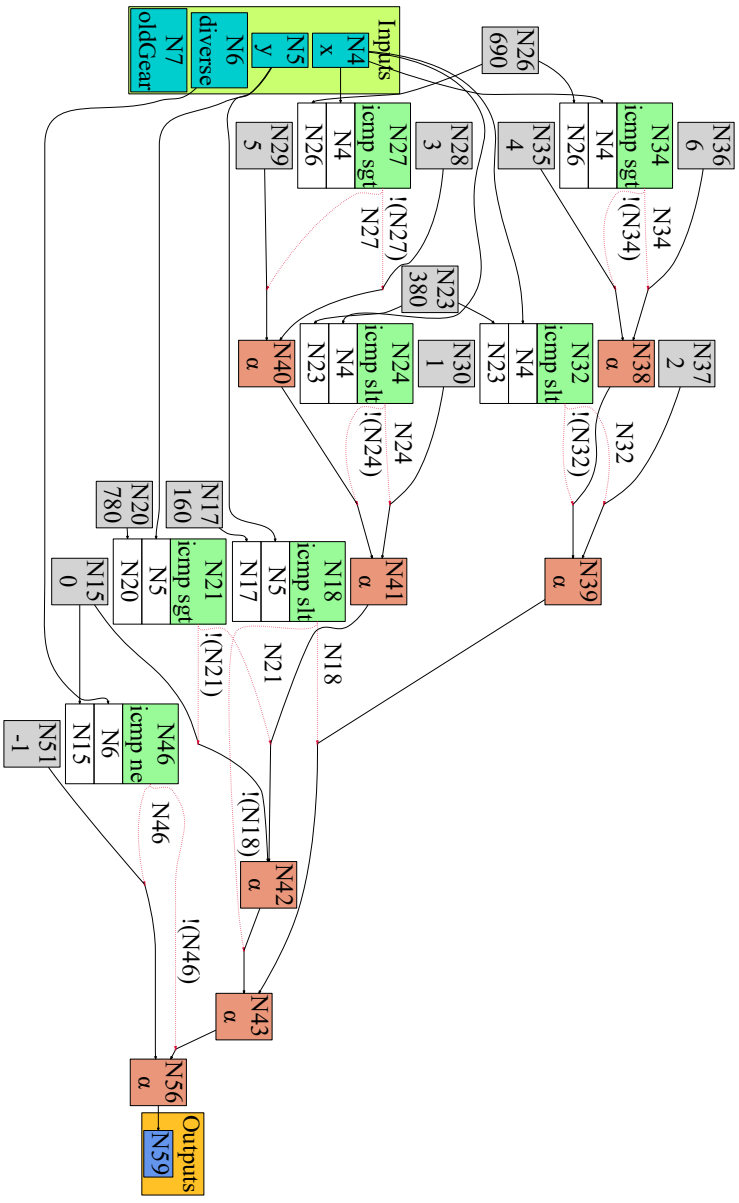

图 3.4　根据代码3.1生成的值流图

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3.3所示。

表 3.3　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}	$\left\{ \begin{array}{c} (-1, dv \ne 0), (0, 160 \le y \le 780 \wedge dv = 0), (1, x < 380 \wedge y > 780 \wedge dv = 0), \\ (2, x < 380 \wedge y < 160 \wedge dv = 0), (3, 380 \le x \le 690 \wedge y > 780 \wedge dv = 0), \\ (4, 380 \le x \le 690 \wedge y < 160 \wedge dv = 0), (5, x > 690 \wedge y > 780 \wedge dv = 0), \\ (6, x > 690 \wedge y < 160 \wedge dv = 0), \end{array} \right\}$

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3.4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3.1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。

3.7　本章小节

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。本章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，包括变量关系分析抽象域、变迁规则的设计与程序理解算法的设计等。在4.1章

节，我们将在可配置的静态分析框架上实现摘要生成算法，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 4 章　工具实现与评估

本章将分别介绍第2章和第3章所述方法的模块实现与评估方法。

4.1　基于区间代数的整数缺陷检测方法的实现与评估

4.1.1　模块实现

我们选择在静态分析框架 Tsmart-V3 上实现第2章节所述抽象域与算法。Tsmart-V3 基于开源的静态分析框架 CPAChecker，在此基础上进行大量的优化与修改，同时提供了相比 CPAChecker 更加丰富的行为接口与更为丰富的文档。工具使用的编程语言为 Java，目标检测语言是 C 语言。

整体的工具的架构设计如图4.1所示。

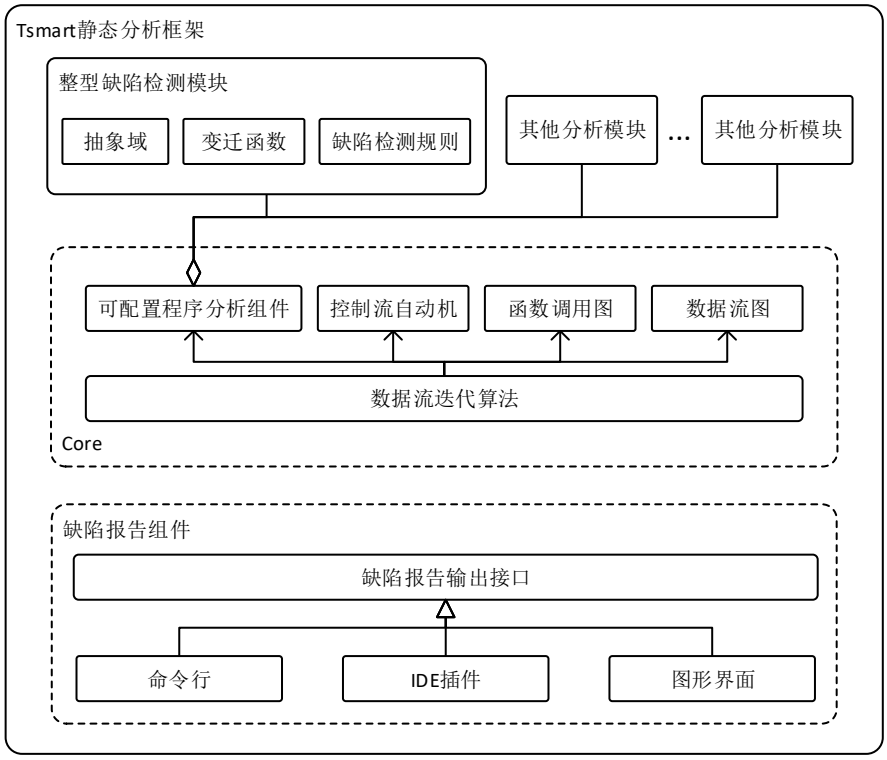

图 4.1　整型缺陷分析工具架构图

本模块的实现基于 Tsmart 静态分析框架，分析框架的输入是 C 语言源代码，工具的前端程序将 C 语言代码转化为具有静态单赋值特性的 LLVM-IR 语言，并在

得到的 LLVM-IR 上构造与之等价的控制流自动机（CFA）。

本模块借助 Tsmart 提供的接口来获取生成的 CFA 并以此为基础进行抽象域与检测算法的实现。我们按照前几个小节对理论域 RangeInteger、Range、MultiRange、SignRange 以及 RangeState 的介绍，在框架上依次实现其定义及计算规则，同时实现 RangeState 的状态变迁规则。理论域在工具上的实现结构同图2.2。

为了实现整型缺陷检测，我们按照规则2-28、2-29和2-30实现检测规则，并通过调用分析框架提供的缺陷报告输出接口实现整型缺陷的输出。

4.1.2　实验设计

为了客观评价本章提出的整型缺陷检测方法，我们选用 Juliet Test Suite 测试集作为本工具的检测样本。Juliet 测试集由美国国家标准技术研究所 (NIST) 整理出的 C 语言上包含各类标准错误的样例代码集合。针对每一类定义在 CWE 上的程序缺陷，Juliet 有对应的文件夹包含了在不同情境下出现这类错误的情况。

本次实验选用 Juliet 测试集中的 CWE190_Integer_Overflow、CWE191_Integer_Underflow 和 CWE369_Divide_by_Zero 三类缺陷样例，其中，CWE190 包含 7 个从测试集 s01 到 s07 共 3420 个测试样例；CWE191 包含 5 个从测试集 s01 到 s05 共 2622 个测试样例；CWE369 包含测试集 s01、s02 共 684 个测试样例。Juliet 测试集为同种缺陷设计了多种多样的发生环境，包括：不同字节长度、不同符号性（有符号数和无符号数）、结构体、枚举、函数调用、循环等。同时，对应的情景也不尽相同，如套接字应用场景、标准输入输出场景、随机数生成等。

如图4.3和图4.2所示代码来自 Juliet 测试集中的一个简单的测试样例，我们以此介绍如何使用 Juliet 测试集对工具进行评估。该样例用于测试缺陷检测工具对整型上溢出缺陷的检测能力。具体地，它试图对 char 类型的变量 data 进行加 1 操作。图4.2包含了正确的两个函数实现：函数 goodG2B 中 data 的取值来自常量赋值，该赋值保证变量 data 的加 1 操作不会上溢出；函数 goodB2G 中 data 的取值来自标准输入，在加法操作前检验了 data 的取值，确认加 1 操作不会产生溢出。在图4.3所示代码中，data 的取值来自标准输入，在加法运算之前并未进行溢出检验，存在整型溢出缺陷。

Juliet 测试集中，所有正确实现的函数均以“xxx_xxx_good”命名，所有包含缺陷的函数均以“xxx_xxx_bad”命名。借此，我们可以方便的对工具的检测结果进行评估：若函数实现中存在缺陷，但未在缺陷报告中出现，或缺陷定位不准确的，记为漏报；若函数实现准确，但在缺陷报告中出现，则记为误报。本次实验依

次将上述测试集的测试样例作为输入，记录工具对各个样例的测试结果并统计误报率、漏报率以及对应的测试时间，以此评估工具的准确性与可用性。实验结果将于4.1.3小节中给出。

```
/* goodG2B uses the GoodSource with the BadSink */
static void goodG2B()
{
    char data;
    data = ' ';
    /* FIX: Use a small, non-zero value that will not cause an
            overflow in the sinks */
    data = 2;
    {
        /* POTENTIAL FLAW: Adding 1 to data could cause an overflow */
        char result = data + 1;
        printHexCharLine(result);
    }
}

/* goodB2G uses the BadSource with the GoodSink */
static void goodB2G()
{
    char data;
    data = ' ';
    /* POTENTIAL FLAW: Use a value input from the console */
    fscanf (stdin, "%c", &data);
    /* FIX: Add a check to prevent an overflow from occurring */
    if (data < CHAR_MAX)
    {
        char result = data + 1;
        printHexCharLine(result);
    }
    else
    {
        printLine("data value is too large to perform arithmetic safely.");
    }
}

void CWE190_Integer_Overflow__char_fscanf_add_01_good()
{
    goodG2B();
    goodB2G();
}
```

图 4.2　Juliet 测试集测试样例举例（正例）

```
void CWE190_Integer_Overflow__char_fscanf_add_01_bad()
{
    char data;
    data = ' ';
    /* POTENTIAL FLAW: Use a value input from the console */
    fscanf (stdin, "%c", &data);
    {
        /* POTENTIAL FLAW: Adding 1 to data could cause an overflow */
        char result = data + 1;
        printHexCharLine(result);
    }
}
```

图 4.3　Juliet 测试集测试样例举例（负例）

本次实验的运行环境列举如下：

- 操作系统：Ubuntu 16.04 LTS 64 位版本
- 处理器：Intel(R) Core(TM) i7-6500U@2.50GHz 4 核心 CPU
- 内存大小：16GB

4.1.3　实验结果

本章介绍的整型缺陷检测工具在 Juliet 测试集上的测试结果如表4.1（CWE190 测试结果）、表4.2（CWE191 测试结果）和表4.3（CWE369 测试结果）所示：

表 4.1　CWE190 测试结果

测试集	总数	误报	漏报	误报率	漏报率	运行时间
S01	418	11	0	0.026315789	0	7m 49s
S02	418	11	0	0.026315789	0	7m 37s
S03	418	11	0	0.026315789	0	7m 54s
S04	418	11	0	0.026315789	0	7m 47s
S05	380	10	0	0.026315789	0	8m 31s
S06	684	18	0	0.026315789	0	15m 42s
S07	684	18	0	0.026315789	0	16m 2s

表 4.2　CWE191 测试结果

测试集	总数	误报	漏报	误报率	漏报率	运行时间
S01	418	11	0	0.026315789	0	8m 23s
S02	418	11	0	0.026315789	0	8m 8s
S03	418	11	0	0.026315789	0	8m 50s
S04	684	18	0	0.026315789	0	13m 57s

续下页

续表 4.2　CWE191 测试结果

测试集	总数	误报	漏报	误报率	漏报率	运行时间
S05	684	18	0	0.026315789	0	13m 9s

表 4.3　CWE369 测试结果

测试集	总数	误报	漏报	误报率	漏报率	运行时间
S01	418	11	0	0.026315789	0	6m 58s
S02	266	7	0	0.026315789	0	4m 35s

通过表中数据可以看出，本章提出的基于区间运算的整型缺陷分析工具在 Juliet-190、191、369 上的测试结果良好：误报率小于 3%，漏报率为 0%。其中，造成误报的原因是 Tsmart 静态分析框架在处理 for 循环时使用了循环摘要策略，导致整型变量在抽象域上的计算结果有一定的精度丢失。

在分析框架处理循环时，常常需要对循环做摘要处理。这在处理具体问题时是非常有好处的：它避免因循环展开而造成的分析时间不确定的问题，甚至规避了无限次循环展开的情形。为了得到循环摘要，常用的做法是使用循环不变式来计算各个整型变量在循环中的行为，进而计算得到结果。工具的后续研究方向是优化循环不变式的求解，从而获得更高的分析精度。

从分析效率上看，工具在各个测试集上的分析时间均小于 10 分钟，平均每个测试样例的分析时间小于 2 秒。具有实际应用价值。

4.2　基于值流图的整型变量关系分析方法的实现与评估

4.2.1　工具实现

虑到根据源代码生成精确值流图的过程十分复杂，基于值流图的整型变量关系分析的实现基于 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4.4所示。

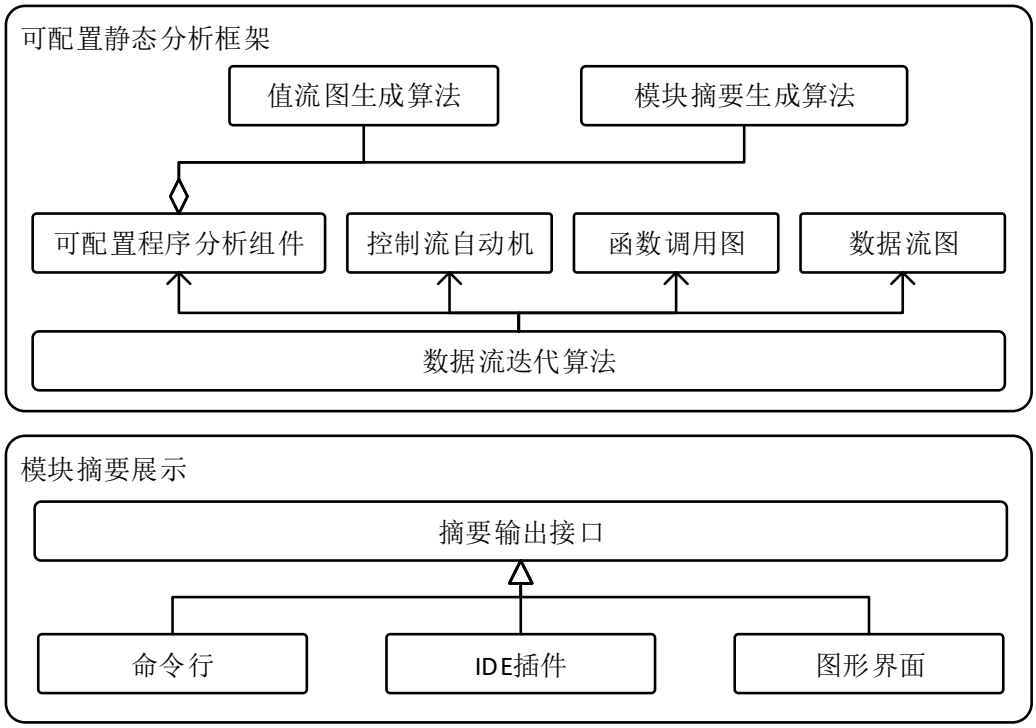

图 4.4　工具架构图

借助 Tsmart 静态分析框架，我们可以快速构建第3章节所介绍的整型变量关系的抽象域并实现其变迁关系。同时，利用 Tsmart 框架提供的相应接口，我们可以生成所需的值流图，并以此为基础实现整型变量关系分析算法。

在后两个小节，我们将从客观、主观两个方面评估算法（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。

4.2.2　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。

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

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。

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

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。

4.2.3　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

续下页

续表 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
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

4.3　本章小结

本章介绍了基于线性区间的整型缺陷检测方法与基于精确值流图的整型变量关系分析方法的具体实现方法与相关实验设计。同时，我们对得到的实验结果进行了评价并分析造成误报与精度损失的原因。

在基于线性区间的整型缺陷检测的实验中，测试用例的误报主要原自基于 for 循环的检测，而现有框架对循环的处理基于循环摘要。由于循环摘要的精度受制于循环不变式技术，因此为了进一步提升整型缺陷分析在循环中的精度，本文后续的工作重点将集中在循环不变式上。

另一方面，基于精确值流图的整型变量关系分析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。此外，为了保证算法的收敛性与时间开销，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。

第 5 章　总结与展望

5.1　工作总结

软件验证与软件确认一直是软件研发过程中不可或缺且十分重要的步骤。本文围绕软件验证与确认在生产实践中面临的两类具体问题提出了有针对性的解决方案。其一，针对当前线性区间抽象域因表示能力不足而造成的程序分析精度损失的问题，本文提出了基于位敏感的区间代数的整型缺陷检测方法。它提高了当前区间抽象域的表示能力，且在保证分析效率的情况下提升了分析精度；其二，针对当前软件确认过程中因程序理解困难而造成的工作低效问题，本文提出了基于精确值流图的整型变量关系分析与缺陷检测方法。借助生成的模块摘要，研发人员可方便的确认程序语义与需求是否相符，发现细微的语义差异，可有效提升程序理解效率并进一步应用于缺陷分析。总体上，本文完成了如下工作内容：

1. 基于位敏感的区间代数的整型缺陷检测方法。为了实现这一目标，本文首先进行了线性区间理论域的设计，设计的核心目标有两点，其一是增加表示符号性的能力，其二是将单独区间表示改换为多区间表示。为此，本文依次定义了线性区间 Range、线性多区间 MultiRange 以及位敏感的线性多区间 SignRange。通过定义理论域、设计计算规则并优化特定情况下的运算，SignRange 能够较好的表示程序中整型变量的可能取值范围。另一方面，为了实现程序语义的表示与分析，本文定义了区间状态 RangeState 来描述程序的每个状态，通过设计抽象域与变迁规则，实现了模拟程序状态的变迁，从而实现程序的语义分析。最后，通过设计 RangeState 上的缺陷检测规则，实现了程序的整型缺陷检测。
2. 基于精确值流图的程序理解与需求确认方法。为了实现这一目标，本文首先通过一个具体案例，分析了程序的摘要对程序理解的帮助。本文对精确值流图的定义和构造方法做出了说明，并基于精确值流图，设计了程序语义的抽取方法。具体地，本文设计了关于变量表达式的抽象域的设计、变迁规则的设计与表达式分析算法的设计。在上述的基础上，实现对程序整型变量在不同程序路径下的取值关系进行语义抽取。最后，借助得到的语义信息生成模块摘要。
3. 模块实现。在提出以上两点的问题解决方案后，本文对其中提到的方法进行了编码实现。基于 Tsmart-V3 静态分析框架所提供的完备的函数接口，可以方便的获取如控制流自动机、精确值流图等信息。通过编程得到相应工具后，

本文设计了一系列的实验，通过实验验证了工具、算法的有效性与实用性。

目前，本文的部分工作已集成至 Tsmart-V3 工具中，并成功应用于某研究所的生产研发环节。在其自主研发的操作系统上，应用 Tsmart 已成功帮助其确认了代码中存在的缺陷。具有较好的实际应用价值。

5.2　研究展望

在现有研究工作内容的基础上，为了进一步提高分析的效率和精度，可以针对如下两个方面展开工作：

1. 优化位敏感区间的抽象表示。在2.3.4小节中，为了实现对整型变量符号性的抽象表示，本文定义了符号性 Signedness 理论域，并结合线性多区间，对程序中具有符号性意义的整型变量提供了抽象表示。但这对理论域的编码实现带来了困难：试想，一个具有符号性的整型变量，其在计算机上的存储内容对固定值而言是不变的，但本文提出的抽象域对同一数值依据不同符号性构造了有符号和无符号两个不同的抽象表示，并设计了算法在其之间做符号性的转化。为了优化位敏感区间的抽象表示方法，我们可以设计新的理论域：这个理论域可以忠实于整型变量在计算机上的二进制串表示，并设计解析规则，对同一个表示整型数字的二进制串根据其符号性不同而使用不同的解析方式，从而得到具体的整数值。这样做所带来的好处是可以进一步减少程序分析所消耗的内存以及更加便于编码实现。
2. 优化程序分析中对循环的处理方式。在4.1小节和4.2小节中，造成工具在试验中产生误报的原因均由程序分析在循环上的处理策略造成的。为了追求静态分析精度与效率的平衡，现有技术通常采用循环不变式技术来计算程序在循环中的行为。为了进一步优化现有分析精度，应分别对整型分析与变量表达式分析这两个模块在循环不变式上的计算方式做优化。

参考文献

[1] 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.

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

[3] Wagner S. Software product quality control. Berlin, Heidelberg: Springer, 2013: 153-193.

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

[5] 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.

[6] Murphy G C, Kersten M, Findlater L. How are Java software developers using the Eclipse IDE? IEEE Software, 2006, 23(4):76-83.

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

[8] Cousot, Patrick and Cousot, Radhia. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints // Proceedings of the 4th ACM Symposium on Principles of Programming Languages. 1977: 238-252.

[9] Cousot, Patrick and Cousot, Radhia. Systematic design of program analysis frameworks // Proceedings of the 6th ACM Symposium on Principles of Programming Languages. 1979: 269-282.

[10] 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.

[11] Jeannet, Bertrand and Miné, Antoine. Apron: A library of numerical abstract domains for static analysis // International Conference on Computer Aided Verification. Berlin, Heidelberg: Springer, 2009: 661-667.

[12] 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.

[13] 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. Science of Computer Programming, 2008, 72(1-2):3-21.

[14] Aho A V, Sethi R, Ullman J D. Compilers: Principles, Techniques and Tools. Addison-Wesley, 1986.

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

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

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

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

[19] Weiser M. Program slices: formal, psychological, and practical investigations of an automatic program abstraction method. PhD thesis, University of Michigan, 1979.

[20] Korel B, Laski J. Dynamic program slicing. Information Processing Letters, 1988, 29(3):155-163.

[21] Gupta R, Soffa M L, Howard J. Hybrid slicing: integrating dynamic information with static analysis. ACM Transactions on Software Engineering and Methodology, 1997, 6(4):370-397.

[22] Schugerl P, Rilling J, Charland P. Beyond generated software documentation—A web 2.0 perspective // 2009 IEEE International Conference on Software Maintenance. IEEE, 2009: 547-550.

[23] Karrer T, Krämer J P, Diehl J, et al. Stacksplorer: call graph navigation helps increasing code maintenance efficiency // Proceedings of the 24th Annual ACM Symposium on User Interface Software and Technology. 2011: 217-224.

[24] Beck F, Gulan S, Biegel B, et al. Regviz: Visual debugging of regular expressions // Companion Proceedings of the 36th International Conference on Software Engineering. 2014: 504-507.

[25] Beyer D, Fararooy A. DepDigger: A tool for detecting complex low-level dependencies // 2010 IEEE 18th International Conference on Program Comprehension. IEEE, 2010: 40-41.

[26] Beck F, Moseler O, Diehl S, et al. In situ understanding of performance bottlenecks through visually augmented code // 2013 21st International Conference on Program Comprehension (ICPC). IEEE, 2013: 63-72.

[27] Haiduc S, Aponte J, Marcus A. Supporting program comprehension with source code summarization // 2010 ACM/IEEE 32nd International Conference on Software Engineering: volume 2. IEEE, 2010: 223-226.

[28] Cornelissen B, Holten D, Zaidman A, et al. Understanding execution traces using massive sequence and circular bundle views // 15th IEEE International Conference on Program Comprehension (ICPC'07). IEEE, 2007: 49-58.

[29] Fittkau F, Waller J, Wulf C, et al. Live trace visualization for comprehending large software landscapes: The ExplorViz approach // 2013 First IEEE Working Conference on Software Visualization (VISSOFT). IEEE, 2013: 1-4.

[30] 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. Berlin, Heidelberg: Springer, 2007: 504-518.

[31] 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.

[32] Navas J A, Schachte P, Søndergaard H, et al. Signedness-agnostic program analysis: Precise integer bounds for low-level code // Asian Symposium on Programming Languages and Systems. Berlin, Heidelberg: Springer, 2012: 115-130.

[33] Gange G, Navas J A, Schachte P, et al. Interval analysis and machine arithmetic: Why signedness ignorance is bliss. ACM Transactions on Programming Languages and Systems (TOPLAS), 2015, 37(1):1-35.

[34] Maalej W, Happel H J. Can development work describe itself? // 2010 7th IEEE International Working Conference on Mining Software Repositories (MSR 2010). IEEE, 2010: 191-200.

[35] 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.

[36] Boysen J P. Factors affecting computer program comprehension. PhD thesis, Iowa State University, 1979.

[37] 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.

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

[39] 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.

致　谢

衷心感谢顾明老师对我的指导与教诲，在我日常的学习与生活中，您始终以身作则，教会我对任何事情做到全情投入，永不放弃。您对工作与生活的热情态度，深深感染并激励着我，让我在面临困难时，勇敢前行。

我还要特别感谢周旻老师在我攻读硕士的三年里对我无微不至的关怀与帮助。在学习与生活中，每当我有疑惑之处，老师总能一针见血的点出问题的根源，并为我提供帮助。在我信心不足，没有研究热情时，是您帮我重塑信心，鼓励我继续完成科研任务。

感谢我的父母，在我求学路上一直默默支持着我。感谢实验室的陈光、谷祖兴、王岳兴、王聪、王晗、王程鹏学长在我需要帮助时为我指点迷津，感谢实验室的其他小伙伴们三年来对我的包容与陪伴。感谢我的好友姚渊、唐晓锐、宋英楠、王华清、陈高勋，是你们的陪伴让我研究生三年多姿多彩。

本课题承蒙多项自然科学基金支持，特此致谢。

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：_____

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] Li Wu, Zhou Min, Huang Hao. Program Understanding and Requirement Validation Based on Accurate Value Flow Graph. (已被 WCSE 2020 录用.)