文本复制检测报告单(全文标明引文)

№:ADBD2020R_202004101751062020041301470120308063517 8 检测时间:2020-04-13 01:47:01

检测文献: 202004130112461068_李兀_基于语义分析的整形缺陷检测与程序理解
作者: 李兀
检测范围: 中国学术期刊网络出版总库
中国博士学位论文全文数据库/中国优秀硕士学位论文全文数据库
中国重要会议论文全文数据库
中国重要报纸全文数据库
中国专利全文数据库
图书资源
优先出版文献库
学术论文联合比对库
互联网资源(包含贴吧等论坛资源)
英文数据库(涵盖期刊、博硕、会议的英文数据以及德国Springer、英国Taylor&Francis 期刊数据库等)
港澳台学术文献库
互联网文档资源
源代码库
CNKI大成编客-原创作品库
个人比对库
时间范围: 1900-01-01至2020-04-13

检测结果

去除本人已发表文献复制比：0.4% 跨语言检测结果：0%
去除引用文献复制比：0.4% 总文字复制比：0.4%
单篇最大文字复制比：0.2%（空指针解引用静态检测方法研究）

重复字数：	[184]	总段落数：	[4]
总字数：	[45214]	疑似段落数：	[2]
单篇最大重复字数：	[89]	前部重合字数：	[89]
疑似段落最大重合字数：	[95]	后部重合字数：	[95]
疑似段落最小重合字数：	[89]		

指 标：☐疑似剽窃观点 ☑疑似剽窃文字表述 ☐疑似自我剽窃 ☐疑似整体剽窃 ☐过度引用

表 格：0 公 式：没有公式 疑似文字的图片：0 脚注与尾注：0

0.8%(89)	202004130112461068_李兀_基于语义分析的整形缺陷检测与程序理解_第1部分（总10503字）
0.9%(95)	202004130112461068_李兀_基于语义分析的整形缺陷检测与程序理解_第2部分（总11130字）
0%(0)	202004130112461068_李兀_基于语义分析的整形缺陷检测与程序理解_第3部分（总11165字）
0%(0)	202004130112461068_李兀_基于语义分析的整形缺陷检测与程序理解_第4部分（总12416字）

（注释：无问题部分 文字复制部分 引用部分）

1. 202004130112461068_李兀_基于语义分析的整形缺陷检测与程序理解_第1部分 总字数：10503

相似文献列表

去除本人已发表文献复制比：0.8%(89) 文字复制比：0.8%(89) 疑似剽窃观点：(0)

1	空指针解引用静态检测方法研究 徐厚峰(导师：王戟) - 《国防科学技术大学硕士论文》- 2008-11-01	0.8%（89） 是否引证：否

原文内容

摘要

摘要

在软件开发、测试、验证与维护工作中，软件的验证与确认是十分重要的步骤。前者关注于软件是否如软件定义般实现，后者关注于软件是否符合预期的需求。但在实际应用过程中，两者均会面临重大的挑战：开发人员不可避免的会因为疏忽而

在软件中引入缺陷，这些缺陷如不及时发现并修正，将会造成巨大的损失；

另一方面，软件确认工作也面临着较大困难：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。对此，如何提高工作人员理解程序代码的效率成了重要问题。

本文依照上述问题，试图从程序的整形缺陷检测与整型变量关系分析两个方面入手，在一定范围内解决上述问题。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，提出了基于符号敏感的区间算数的整型缺陷检测方法。具体地，本文依次设计了线性区间、线性多区间以及符号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，为了实现程序的语义表示，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；最后通过设计缺陷检测规则实现程序的整形缺陷检测。（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算数的整形缺陷方法在 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，并可有效提升程序

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

Abstract

In software development, testing, verification and maintenance, software verification and validation is a very important step. The former focuses on whether the software is implemented as software definition, while the latter focuses on whether the software meets the expected requirements. However, in the practical application process, both of them will face major challenges: developers will inevitably introduce defects into the software due to negligence, and if these defects are not found and corrected in time, it will cause huge losses; on the other hand, the software confirmation work also faces great difficulties: to judge whether the software meets the expected needs, in fact, it is to judge the software implementation Match with software requirements. However, the description of the two is different. The former is code with complex logical structure, while the latter is abstract text description, which is quite different. In this regard, how to improve the efficiency of staff to understand the program code has become an important issue.

According to the above problems, this paper attempts to solve the above problems in a certain range from two aspects of the program's plastic 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 accuracy caused by the lack of representation ability in the current linear interval abstract domain, an integer defect detection method based on symbol sensitive interval arithmetic is proposed. Specifically, this paper designs the linear interval, linear interval and symbol 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 realize the semantic representation of the program, this paper designs the interval abstract domain to describe each state of the program, and sim- ulates the change of the program state by designing the change rules Finally, through the design of defect detection rules to achieve the plastic defect detection of the program. (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 detec- tion 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 tran- sition rules and the design of expression analysis algorithm. Using the algorithm, we can

Abstract 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 coding 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% and less than 3% 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 efficiency of program understanding, which has good application value.

Key Words: Software verification and validation; defect detection; program understand- ing; static analysis

目录

目录

第 1 章引言
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（verifi- cation）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 1

第 1 章引言

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。

基于精确直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 23

2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。 2

第 1 章引言

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 < , ∩ > 来表示所有可能的状态集合，其中是值集，是抽象域的别名；∩

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： 3
第 1 章引言
Algorithm 1 数据流分析算法
1: procedure Recursion 2: = ;
3: for = 1 to do
4: initialize the value at node ;
5: add to the ;
6: end for
7: while ≠ do
8: remove a node from the ;
9: recompute the data flow fact at ;
10: if new data flow fact is not equal to the old one at then
11: add each successor/predecessor of to 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
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 4
第 1 章引言
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、 5
第 1 章引言
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）基于符号敏感的区间算数的整型缺陷检测
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。

2. 202004130112461068_李兀_基于语义分析的整形缺陷检测与程序理解_第2部分

总字数：11130

相似文献列表

去除本人已发表文献复制比：0.9%(95)　文字复制比：0.9%(95)　疑似剽窃观点：(0)

1	机场地面移动目标监测及碰撞风险评估研究 赵新宇(导师：侯启真;徐焕然) -《中国民航大学硕士论文》- 2019-05-15	0.6%（67） 是否引证：否
2	基于图学习的高维数据特征提取及聚类方法研究 诸葛文章(导师：易东云;侯臣平) -《国防科技大学硕士论文》- 2017-12-01	0.3%（35） 是否引证：否

原文内容

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 6
第 1 章引言
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。
基于符号敏感的区间算数的整型缺陷检测缺陷检测规则设计
线性区间理论域的设计
Range MultiRange
SignRange Signedness
程序的语义表示区间状态抽象域的设计变迁规则设计基于精确值流图的程序理解与需求确认精确值流图的定义与构造
模块摘要的生成
程序语义抽取抽象域的设计表达式分析算法变迁规则设计工具实现与评估图 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。
针对基于符号敏感的区间算数的整型缺陷检测，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。 7

第 1 章引言

针对基于精确值流图的程序理解与需求确认，本文首先对精确值流图的定义

与构造做出说明，随后设计了程序语义抽取方法，具体共分为抽象域的设计、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。

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；

2. 基于程序在不同状态下整型变量可能取值的抽象表示，设计并实现了针对 C

语言的整数缺陷检测方法与工具；

3. 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；

4. 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；

5. 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。

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算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。 8

第 1 章引言

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。随后介绍相关的程序理解方

法与摘要的生成算法，并在最后进行了实验，从主客观两个方面证明了摘要生成

算法的有效性与实用性。 9

第 2 章基于区间算数的整数缺陷检测

第 2 章基于区间算数的整数缺陷检测

本章介绍基于区间算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的精度是不足的，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；2.4小节利用设计的抽象域与变迁规则，实现程序的整形缺陷检测方法；最后于2.5和2.6两个小节分别介绍了本章所述方法的工具实现并通过实验验证了工具的有效性。

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 [∞, 3]，则函数 A 中第 4 行 x 的

取值范围是 [∞, 2]，第 6 行 x 的取值范围是 [2, +∞]。在第 8 行，由于状态合并， x 的取值范围是 [∞, +∞]，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算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 10

第 2 章基于区间算数的整数缺陷检测

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 = (,)： 是节点的集合，节点 ∈ 表示程序的状态。 × × 是边的集合，有向边 = (, , ′) ∈ 表示从某个程序状态到下一个程序状态的过程，其中， 是状态之间所执行的指令。一般地，我们用 ( ) 表示边的前驱节点 ，用 ( ) 表示边的后继节点 ′，用

( ) 表示边所执行的动作 。

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 ( ) = ；

假设边（AssumeEdge）：描述了假设条件是否成立。操作 ( ) = (% ,) ，

其中 % 为条件变量，它对应于 LLVM-IR 中 i1 类型的变量，表示假设条件。 ∈ {0, 1} 为条件取值，表示当前边上假设条件变量 % 的具体取值。一般地，我们用 ( ) 表示假设边的条件变量 % ，用 ( ) 表示

条件变量的取值 。

Phi 边（PhiEdge）：描述了 LLVM-IR 中的 phi 指令。操作 ( ) = (% ,) ，

其中 % 对应于 phi 指令， 指示 phi 边上的第几个操作数作为当前指 11

第 2 章基于区间算数的整数缺陷检测

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 ( ) = (%)，其中 % 为所执行的指令。

为了便于描述，定义函数用于获取边的类型： ( ) := ∈ ∈ ∈ ∈ (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 = (, , ,)。其中， 为抽象域； 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； 为状态合并算子； 为状态覆盖算子。

算法2描述了 CPA 的算法流程：算法以分析 、CFA 图和初始状态 0∈

作为输入，通过维护工作队列和可达状态集并根据转移关系来计算当前状态的后继状态。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 ′使用算子来更新所有可达状态，如果可达状态

″被更新，则将该状态加入到等待队列中以更新其后继状态。如果更新后的可达状态无法覆

盖 ′，则将该状态分别加入和可达集中。

Algorithm 2 CPA 算法

Require: = (, , ,)，CFA 图 ，初始状态 0∈

Ensure: 可达状态集 1: ←{ 0} , ← { 0}

2: while ≠ do

3: 取出的首元素 ;

4: for all ′满足

′do 12

第 2 章基于区间算数的整数缺陷检测

5: UpdateRW(′, , , ,);

6: end for

7: end while

Algorithm 3 UpdateRW 算法

1: function UpdateRW(′, , , ,)

2: for all ′∈ do 3: ← (, ′);

4: if
≠ ′then 5: ← (∪ { }) { ′}; 6: ← (∪ { }) { ′};
7: end if
8: end for
9: if (,) then 10: ← (∪ { }); 11: ← ∪ { };
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，抽象域的组成结构如图2.2所示。RangeState
为程序变量在线性空间的抽象表示，它包含整型变量的访问路径、取值范围、符号等信息。 13
第 2 章基于区间算数的整数缺陷检测
RangeInteger
Range
MultiRange
SignRange
Signedness
ApRangeState TransferRelation
AccessPath
Bitwidth
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 RangeIn- teger 来表示一个具体的整数，它是整数域 Z 的一个拓展。
定义 2.2： 记
:= Z ∪ {+∞, ∞, , } 为扩展的整数理论域。
上的最大元 (
) 为 ，它表示任意值；最小元 (⊥
) 为 ，它表示非数；
上的偏序关系 (
) 为 Z 上的自然扩展，对于元素 ∞ 与 +∞，规定
∞ < +∞，且对任意 ∈ Z，有 < +∞ 以及 ∞ < 。特殊地， ∞ 与 ∞、
+∞ 与 +∞ 无法比较，⊥，
与其他任意元素无法比较。
上的上确界操作 (
) 定义为： := ≠ ⊥ ∧ ≠ ⊥ = ⊥ = ⊥ ⊥ = ⊥ ∧ = ⊥
对于元素 ∞ 或 +∞ 参与的运算，其规则如表2.1所示。其中，函数为 14
第 2 章基于区间算数的整数缺陷检测
取符号数，具体定义如下： ( ) :=
1 (∈ Z ∧ > 0) ∨ = +∞ 0 = 0
1 (∈ Z ∧ < 0) ∨ = ∞ (2-2)
特别的，元素 +∞ 与 ∞ 所参与的某些运算是未定义的。这样的运算在表中记录为 ，如 (+∞) + (∞), (+∞) ÷
(+∞) 等。
表 2.1 RangeInteger 运算规则
运算符号运算规则 negate ( ) := ≠ ±∞ ∞ = +∞ +∞ = ∞ add + + := + , ≠ ±∞ = ±∞ ∧ ≠ ±∞ ≠ ±∞ ∧ = ±∞ , = ±∞ ∧ ( ) = ( ) other cases subtract := , ≠ ±∞ = ±∞ ∧ ≠ ±∞ ( ) ≠ ±∞ ∧ = ±∞ , = ±∞ ∧ ( ) ≠ ( ) other cases multiply × × := × , ≠ ±∞ +∞ ( ) × ( ) > 0 ∞ ( ) × ( ) < 0 other cases divide (≠ 0) ÷ ÷ := ÷ , ≠ ±∞ 0 ≠ ±∞ ∧ = ±∞ +∞ × (×) = ±∞ ∧ ≠ ±∞ other cases
续下页 15
第 2 章基于区间算数的整数缺陷检测
续表 2.1 RangeInteger 运算规则
运算符号运算规则 modular (> 0) mod := mod , ≠ ±∞ 0 ∧ = +∞ < 0 ∧ = +∞ other cases and or xor shl shr << >> := , ≠ ±∞ other cases
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，构成的二元组。这里， ，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
:= {[,]| ∈ I, ∈ I, } ∪ { } 是一个半格：
上的最大元 (
) 是 [∞, +∞]，最小元 (⊥
) 是空集 ；
上的偏序关系 (
) 定义为： [1, 1] [2, 2] 2 1∧ 1 2；
上的上确界操作 (
) 定义为： [1, 1] [2,

2] := [min(1,
2), max(1, 2)] ,
此处 max 和 min 为上的自然扩展。
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 ⊥
与任何元素的计算结果均为 ⊥
，为表示方便，在表中默认操作数均不为 ⊥ 。
表 2.2 Range 运算规则
运算符号运算规则 negate ([,]) := [[( ), ( )]] add + [1, 1] + [2, 2] := [[1+ 2, 1+ 2]] subtract [1, 1] [2, 2] := [[1 2, 1 2]] multiply × [1, 1] × [2, 2] :=
[min( ), max( )], = { 1, 1} × × { 2, 2}
续下页1 6
第 2 章基于区间算数的整数缺陷检测
续表 2.2 Range 运算规则
运算符号运算规则 divide (2≠ 0) ÷ [1, 1] ÷ [2, 2] :=
[min( ), max( )] 0 [2, 2], = { 1, 1} × ÷ { 2, 2} [1, 1] ÷ [1, 2] 2= 0 [1, 1] ÷ [2, 1] 2= 0 [( ),] 2< 0 < 2,
= max(| 1|, | 1|) modular (2≠ 0) [1,
1] mod [2, 2] := 1 2× 1 2> 0 1= 1÷ 2+ [1, 1] 1 2× 2 2< 0 2= 1÷ 2 [1, 1] and [1, 1] [2, 2] := 1 1 1= 1∧ 2= 2 [0, 1] 1= 1 0 [0, 2] 2= 2 0 [0, +∞] 1 0 ∨ 2 0 [∞, 1] 1< 0 ∧ 2< 0
[∞, +∞] other cases or [1, 1] [2, 2] := 1 1 1= 1∧ 2= 2 [1, 1] 1= 1= 1 ∨ 2= 2= 1
[max(1, 2), +∞] 1 0 ∧ 2 0 [∞, 1] 1< 0 ∨ 2< 0
[∞, +∞] other cases xor [1, 1] [2, 2] := 1 1 1= 1∧ 2= 2 [1, 1] 2= 2= 0 [2, 2] 1= 1= 0 [∞, +∞] 1× 2< 0 ∨ 2× 2< 0 [0, +∞] (1 0 ∧ 2 0) ∨ (1< 0 ∧ 2< 0)
[∞, 1] other cases
续下页 17
第 2 章基于区间算数的整数缺陷检测
续表 2.2 Range 运算规则
运算符号运算规则 shl << [1, 1] << [2, 2] := 2< 0 [1, 1] << [0, 2] 0 ∈ [2, 2]
[min( ), max( )] other cases, = { 1, 1} × << { 2, 2} shr >> [1, 1] >> [2, 2] := 2< 0 [1, 1] >> [0, 2] 0 ∈ [2, 2]
[min( ), max( )] other cases, = { 1, 1} × >> { 2, 2}
在表格中，我们定义了构造函数 [[ ]]
，利用它可以方便的构造 Range： [[,]] :=
[,] (, is comparable) ∧
[,] (, is comparable) ∧ > other cases (2-3)
同时，为了进一步简化书写，我们定义了集合上的操作 ×
。对集合和
，有： × = { | ∈ , ∈ } (2-4)
Range 是可比较的，对于 = [1, 1], = [2, 2] ∈
，有： < 1< 2(2-5) > 1> 2(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
→ ∞ 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
= 1 时，MultiRange 退化为Range。 18
第 2 章基于区间算数的整数缺陷检测
在实际应用中，由于 MultiRange 可以根据需要通过改变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 := {[1, ,] | ∈ , 1< < , 0
} 是一个半格：
上的最大元 (
) 是 [
] ，最小元 (⊥
) 是 [ ] ;
上的偏序关系 (
) 定义为： [1, ,] [1, ,] , that . (0 , 0) ;
上的上确界操作 (
) 定义为： [1, ,] [1, ,] := [[1, , , 1, ,]] . (0 , 0)。
上述的定义中使用了 MultiRange 的构造函数 [[ ]]
，其函数逻辑如算法4所示。该函数接受一个由 Range 组成的集合或直接给出待组合的 Range 列表 1, 2, ,
。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 。并将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 。为了保证得到的 MultiRange 中 Range

的个数小于
，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 [[ ]]
Require: = { 1, , },
Ensure: [1, ,]. (0)
1: sort by
. then by . 2: ←
3: [,] ← peek( )
4: for all in do
5: if . +
1 then
6: ← max(. ,)
7: else 8: ← ∪ [,] 9: [,] ←
10: end if
11: end for 19
第 2 章基于区间算数的整数缺陷检测
12: while ( ) > do
13: pick range pair (,
+1) in whose distance is shortest 14: ← ({ , +1}) ∪ (+1)
15: sort by
. then by .
16: end while
17: return ( )
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 = [1, ,
] 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 ×
，即对于 , ∈
，有 × = ( ) ,
其中 = { | ∈ , ∈ , 0 , 0 }。
表 2.3 MultiRange 运算规则
运算符号运算规则 negate ( ) := [[(1), , ( )]] add + + := [[× +]] subtract := [[×]] multiply × × := [[× ×]] divide ÷ ÷ := [[× ÷]] , ≠ 0 modular mod mod
:= [[×mod]] , ≠ 0 and := [[×]] or := [[×]] xor := [[×]] shl << << := [[× <<]] shr >> >> := [[× >>]]
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： = [[1, 3]]， = [[2, 3]]。 ×
按照上述计算规则的计算结果为 [[1, 3] × [2, 3]] =
[[2, 9]]，而实际上 ×
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 = [1, ,], = [1, ,] ∈
，有： < < 1(2-7) > 1> (2-8)
为了后续便利，在 MultiRange 上定义如下函数： 20
第 2 章基于区间算数的整数缺陷检测 (,) = , ∈ , ∈
用于删除所表示的整数范围中对应的部分。例如：对于 = [[0, 255]]，有 . ([10, 100]) = [[0, 9], [101, 255]]。 (,) = ∩ , ∈ , ∈
用于得到所表示的整数范围中与重合的部分。例如：对于 = [[1, 10], [20, 30]]，有 . ([5, 25]) = [[5, 10], [20, 25]]。
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
:= { , , } 是一个半格：
上的最大元 (
) 是 ，最小元 (⊥
) 是 ；
上的偏序关系 (
) 定义为：

上的上确界操作（
）定义为： 1 2:= 2 1 2 1 2 1
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）。

指　标

疑似剽窃文字表述

1. 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。

3. 202004130112461068_李兀_基于语义分析的整形缺陷检测与程序理解_第3部分　总字数：11165

相似文献列表

去除本人已发表文献复制比：0%(0)　文字复制比：0%(0)　疑似剽窃观点：(0)

原文内容

由于程序的这一特点，定义函数 | ( ), ∈
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，，
分别记为，，。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 := {(, ,) | ∈
, ∈ Z+, ∈
{ , , }} 是一个半格：
上的最大元（
）是（，
+, ），最小元（ ⊥
）是（⊥
, ⊥Z+, ）； 21
第 2 章基于区间算数的整数缺陷检测
上的偏序关系（
）定义为：（1, 1, 1）（2, 2, 2）（1|
(s(2)) 2) ∧（1 2）；
上的上确界操作（
）定义为：（1, 1, 1）（2, 2,
2) :=（，，），其中，
:= max(1, 2)
:= max(1, 2) := 1|（）2|（）。
上述定义中，函数 s( ), ∈
为取的 Signedness 值。类似的，有 w( ), ∈
用于取的 Bitwidth 值；r( ), ∈
用于取的 MultiRange
值。
值得注意的是，r( ) :=
。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 ( ), ∈
和函数 (,), ∈ Z+, ∈
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： ( ) = (,) := [0, 2 1] = [2 1, 2 1 1] ∈ { , } (2-9)
作为补充的，定义函数 ( ), ∈
和 (, ,), ∈ , ∈
Z+, ∈
用于得到取值范围在限定范围内的 SignRange：
( ) := (∩ ( ), w( ), s( )) (2-10) (, ,) := (| ( ) ∩ (,), ,) (2-11)
函数 |
的转换规则如算法5和算法6。
Algorithm 5 Convert2Unsigned, | ( )
Require: = (, ,), ∈ { , }
Ensure: new SignRange with Unsigned type 1: ← (,). + 1 2: ←
3: for all in ( ) do
4: if < 0 then 5: ← ∪ (+)

6: else if > 1 then 22
第 2 章基于区间算数的整数缺陷检测 7: ← ∪
8: else 9: ← ∪ ([. , 1] +) ∪ [0, .]
10: end if
11: end for
12: return = ([[ ]] , ,)
Algorithm 6 Convert2Signed, | ( )
Require: = (, ,), ∈ { , }
Ensure: new SignRange with Signed type 1: ← (,) 2: ← (,). + 1 3: ←
4: for all in ( ) do
5: if < + 1 then 6: ← ∪
7: else if > then 8: ← ∪ ( )
9: else 10: ← ∪ [. ,] 11: ← ∪ ([+ 1, .])
12: end if
13: end for
14: return = ([[ ]] , ,)
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，∈
参与的二元运算将按如下规则（Unifiy Signedness）更新自己的符号性并进行运算：
（1）当
∈ { , , , , } 时，按下列规则更新符号性、
位宽并运算：
′= w( ) = w( ) ← max(w( ), w( )) (2-12) 23
第 2 章基于区间算数的整数缺陷检测 := | ( ) |
( ) s( ) = v s( ) = s( ) = ∧ s( ) = | ( ) |
( ) other cases (2-13)
（2）当
∈ { , , } 时，按下列规则运算，不更新位宽与符号性： := | ( ) | ( ) (2-14)
（3）当
∈ { , } 时，按下列规则更新符号性并运算，不更新位宽： := | ( ) (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 ′∈ Z +, ′∈
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 ( ) := (
(r( )), w( ), s( )) s( ) ∈ { , } (| ( )) | ( ) = add + +
:= (r( ) + r( ), ′, ′) subtract
:= (r( ) r( ), ′, ′) multiply × ×
:= (r( ) × r( ), ′, ′) divide ÷ ÷
:= (r( ) ÷ r( ), ′, ′), ≠ 0 modular mod mod
:= (r( ) mod r( ), ′, ′), ≠ 0 and
:= (r( ) r( ), w( ), s( )) or
:= r( ) r( ), w( ), s( )) xor
:= (r( ) r( ), w( ), s( )) shl << <<
:= (r( ) << r(′), w( ), ′), in which ′← . ([w( ), +∞]). ([∞, 1]) shr >> <<
:= (r( ) >> r(′), w( ), ′), in which ′← . ([w( ), +∞]). ([∞, 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。 24
第 2 章基于区间算数的整数缺陷检测
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，∈
，有：
< r(′) < r(′) (2-16)
> r(′) > r(′) (2-17)
其中，′,
′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 上定义函数用于获取当前线性区间中的最大值，函数用于获取当前线性区间中的最小值。对于 = ([[1, 5], [20, 100]], 8,) ∈

，有 ( ) = 100, ( ) = 1。

2.3.5 区间状态（RangeState）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 Sign-

Range 的映射，用符号表示。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

，那么对于 ∈

，记 ( )

为在上的映射，即访问路径为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 ：

定义 2.7： 定义区间状态理论域 := { := | ∈ , ∈

} 是一个半格：

上的最大元 := , ∈

，最小元 ⊥ := ⊥ , ∈ ；

上的偏序关系 (

) 定义为： aps( ) aps( ) ∧ ∈ aps( ), ( ) ( ) ；

上的上确界操作 (

) 定义为：

:= , ∈ aps( ) ∪ aps( ), = ( ) ( )。

上述定义中，函数 aps( ) := { ∈ | ( ) ≠ ⊥

} 表示区间状态的访 25

第 2 章基于区间算数的整数缺陷检测

问路径集合。类似的，定义函数 ranges( ) := { ( ) | ∈ aps( )} 为取状态区间的区间表示 SignRange 的集合。

区间状态通常与 CFA 上的状态节点是关联的，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 的边表示从一个程序状态到下一个程序状态的过程，

( ) 表示边所执行的动作。对应的，我们定义理论域上的变迁关系

为CFA 的边所表示的程序变迁关系的抽象：

定义 2.8： 记 := × ×

为区间状态理论域上的变迁关系。其中，

表示变迁关系上的操作。我们用 := (

) 来表示所具体执行的动作，称之为变迁函数。对于 ∈

， ′= ( ) 表示区间状态在应用变迁函数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和区间状态上的变迁关系来抽象描述 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 ∈ 所表示的指令类型的不同（见公式2-1），

对应的变迁函数 := (

) 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 ∈ ，( ) = ，其变迁关系为： 1 → 2, 2= 1 (1, 1) → (2, 2))

T-R-EMPTY (2-18)

其中， 1,

2∈ 表示 CFA 上的状态节点， 1,

2则为 1,

2在区间状态域

上的抽象表示。为方便起见，如无特殊说明，后续定义中的 1, 2, 1, 2

同此含义。

（2）对于假设边 ∈ ，( ) = (% ,)。其中， ∈

{ , }。其变迁关系为： 1 [% =] → 2,

2= refineVals(1, % ,) (1, 1) → (2, 2)

T-R-ASSUME (2-19)

其中，函数 refineVals 的操作是将区间状态 1中访问路径为 % 的取值范围改为所代表的值，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 = phi [1, 1] [2,

2] 的 Phi 边（PhiEdge），其变迁关系为： 1

=phi [1, 1] [2, 2] → 2, 2= 1[/] (1, 1) → (2, 2)

T-R-PHI (2-20) 26

第 2 章基于区间算数的整数缺陷检测

在上式中，当 1(

1) = 0 时， := 1(

1) 当 1(

2) = 0 时， := 1(2)。

（4）对于形如 = alloca 这样的指令边（InstructionEdge），约定为类型变量在计算机上存储所用位宽， 为类型变量的符号性，则变迁关系为： 1
=alloca → 2, 2= 1[/ (, ,)] (1, 1) → (2, 2)
T-R-ALLOCA (2-21)
（5）对于形如 =
这样的指令边，其变迁关系为： 1 = → 2, 2= 1[/ 1( ) 1( )] (1, 1) → (2, 2)
T-R-OP (2-22)
此处为在上的自然扩展。
（6）对于形如 = cast to 这样的指令边，其变迁关系为： 1
=cast to → 2, 2= 1[/ (1( ), ,)] (1, 1) → (2, 2)
T-R-CAST (2-23)
其中， 为类型变量在计算机上存储所用位宽， 为类型变量的符
号性。后续定义中，若无特殊说明，对于指令中出现的类型，将继续沿
用符号，。
（7）对于形如 = cmp pre , 这样的指令边，其变迁关系为： 1
=cmp pre , → 2, 2=
1[/ compare(1( ), 1( ))] (1, 1) → (2, 2)
T-R-CMP (2-24)
其中，函数 compare(,), , ∈
用于在上依据定义的比较规则得到的比较结果。
（8）对于形如 = load 这样的指令边，其变迁关系为： 1
=load → 2, 2= 1[/ (1( ), ,)] (1, 1) → (2, 2)
T-R-LOAD (2-25)
（9）对于形如 store , 这样的指令边，其变迁关系为： 1 store , → 2, 2= 1[/ (1( ), ,)] (1, 1) → (2, 2)
T-R-STORE (2-26) 27

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 = [,] 只能描述从到的全部整数。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 = [[1, 3]] ∈ , = [[2, 3]] ∈
两个线性区间，
我们可将其分别表示为 ′= {1, 2, 3},
′= {2, 3}。则区间上的运算 ×
的运算结果可以通过集合上的运算结果 ′××
′= {2, 3, 4, 6, 9} 导出，则 × = [[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 = [[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 ，对于一个整数 ∈
和线性区间 ∈
， . ( ) 用于判定中是否存在整数 ；定义 cnt( ) 用于获取表示的线性区间内包含的整数个数；定义函数 disperse( ) := { = [,] ∈ 28

| . ( )} 用于得到对应的整数表示集合。
则对于 MultiRange 上二元运算 :=
，其中 ∈ { , , , ,
, }，其优化后的运算规则可表述为： :=
[[disperse( ) × disperse( )]] cnt( ), cnt( ) other cases (2-27)
其中，操作符为操作符在理论域上的对应版本。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 基于区间算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 29

续表 2.5 CWE 定义的数值导向型缺陷CWE

编号CWE 类型解释

401 Memory Leak 堆内存泄露

本节介绍的基于区间算数的整形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 上通过边连接的两个状态节点 1,

2。当是形如 =

，其中

∈ { , , , , , , } 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： reportOverflow(1,) ( ) < ( ), = 1( ) 1( ) (2-28) reportUnderflow(1,) ( ) < ( ), = 1( ) 1( ) (2-29)

其中，操作符是操作符在 SignRange 上的对应版本。 1,

2是状态节

点 1,

2在区间状态理论域

上的抽象表示。类似的，当

∈ { , } 时，

我们应用除零检测规则： reportDivideByZero(1,) . (0), = 1( ) 1( ) (2-30)

其中，函数为 MultiRange 上的自然扩展。

2.5 模块实现我们选择在静态分析框架 Tsmart-V3 上实现上述抽象域与算法。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2.3所示。 30

Tsmart静态分析框架Core

数据流迭代算法可配置程序分析组件控制流自动机函数调用图数据流图其他分析模块 ... 其他分析模块整形缺陷检测模块抽象域变迁函数缺陷检测规则缺陷报告组件缺陷报告输出接口命令行 IDE插件图形界面图 2.3 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形缺陷的输出。

2.6 实验过程与结果

2.6.1 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 31

第 2 章基于区间算数的整数缺陷检测

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。

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。其中，误报是指样例本身无缺陷但测试报告指示存在缺陷的情形，漏报是指样例本身存在缺陷但测试报告指示无缺陷的情形。

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

2.6.2 实验结果本章介绍的整形缺陷检测工具在 Juliet 测试集上的测试结果如表2.6（CWE190

测试结果）、表2.7（CWE191 测试结果）和表2.8（CWE369 测试结果）所示：

表 2.6 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 32

第 2 章基于区间算数的整数缺陷检测

表 2.7 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

S05 684 18 0 0.026315789 0 13m 9s

表 2.8 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。

2.7 本章小结本章介绍了基于区间算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 +∞ 和 ∞ 的概念，同时，我们介绍了有 ∞, +∞ 参与的整数运算规则，为后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， 33
第 2 章基于区间算数的整数缺陷检测
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，提出了基于区间算数的缺陷检测方法。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，并通过实验验证了有效性。

在实验中，测试用例的误报主要原自基于 for 循环的检测。由于循环摘要的精
度受制于循环不变式技术，为了进一步提升整型缺陷分析在循环中的精度，后续的工作重点将集中在循环不变式上。 34
第 3 章基于值流图的整型变量关系分析与缺陷检测
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；随后3.3小节具体介绍了算法的原理与实现方法，
并于3.4小节进行了实验验证。3.5小节最后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 [36] 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[
37]。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[
32-33]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[
34]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[
35]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。 35
第 3 章基于值流图的整型变量关系分析与缺陷检测

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。

1档
2档
3档
4档
5档
6档倒车档
(需先按下排挡杆)x y

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：、 、 和 ，根据参数计算出档位值并返回；

2. 返回的档位值的取值集合为 {0, 1, 2, 3, 4, 5, 6, 1, 2}。其中 1-6 为前进档，-2 为倒车档；

3. 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. 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. 为倒挡信号，当下压变速器挡杆时值为 1，否则为 0；

6. 的取值为本次变速器挡杆操作前的档位取值。 36

第 3 章基于值流图的整型变量关系分析与缺陷检测

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 (> 690 ∧ < 160 ∧ (= 0 ∨ (≠ 37

第 3 章基于值流图的整型变量关系分析与缺陷检测

1 ∧ ≠ 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 ≠ 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和简写为和 ）。不难发现程序无法返回 = 2 且返回值与参数无关，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 - (> 690) ∧ (< 160)∧ (= 2 ∨ = 1) ∧ (≠ 0)

-1 dv ≠ 0 (160 ≤ ≤ 780) ∧ (≠ 0) 0 (160 ≤ ≤ 780) ∧ (= 0) (160 ≤ ≤ 780) ∧ (= 0) 1 (< 380) ∧ (> 780) ∧ (= 0) (< 380) ∧ (> 780) 2 (< 380) ∧ (< 160) ∧ (= 0) (< 380) ∧ (< 160) 3 (380 ≤ ≤ 690) ∧ (> 780) ∧ (= 0) (380 ≤ ≤ 690) ∧ (> 780) 4 (380 ≤ ≤ 690) ∧ (< 160) ∧ (= 0) (380 ≤ ≤ 690) ∧ (< 160) 5 (> 690) ∧ (> 780) ∧ (= 0) (> 690) ∧ (> 780) 6 (> 690) ∧ (< 160) ∧ (= 0) (> 690) ∧ (< 160)∧ (≠ 2 ∧ ≠ 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 38

第 3 章基于值流图的整型变量关系分析与缺陷检测

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。
源代码
1. 构造控制流自动机
2. 构造精确值流图 1 2 4 5 6 7 3
上下文构造表达式分析
1. 执行静态分析算法
2. 生成模块摘要
应用变迁函数计算程序不动点合并计算结果 ... return value =
0 (a > b || b > c) ||
1 (a + b c && b < c) ...
带有摘要的
程序源代码
CFA
VFG
图 3.2 算法的工作流程图
3.3.1 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 3.2 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 39
第 3 章基于值流图的整型变量关系分析与缺陷检测
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：←{& , & }。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：←{(& , ≠ 0), (& , = 0)}。
Outputs
N19
N18
DEREF
N5
&x
N5==N14
N6
&y
N6==N14
N9 0
N13(%7) icmp ne
N11
N9 N10 1
N11 p
N13
!(N13)
N14
αN16
α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 ← 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（ ），每个 = (,) 是一个有向图，其中是值流图节点集合，每个节点代表程序中的变量或常量。节点 ∈ 上也可以带有条件，表示控制依赖。 × 为有向边集合，表示值的流向。边 ∈ 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 < > 表

示，其中为字面常量。如图3.3中的节点 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 < > 表示，其中是对应的数据的内存位置的访问路径。如图3.3中的节点 1

1可表示为 < >。

复制节点 CopyNode：表示值的复制关系，它的值复制于前驱节点，用 < >

表示，其中是 VFG 中的节点。如图3.3中的节点 1

9可表示为 < 16>。 40

第 3 章基于值流图的整型变量关系分析与缺陷检测

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 < , 1, …,

> 表示，其中为运算符， 1, …,

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 1

3可表示为 <≠, 11, 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 < (1, 1), …, (,

) > 表示，其中为 JoinNode 的前驱节点，

为数据依赖，表示当为真时，JoinNode 的取值为

。图3.3中节点 1

4可表示为 < (5, 13), (6, 13) >。

对于每个模块 ，定义两个特殊节点集合 ( ) 和 ( ) 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 : : → 用于取常量节点的具体值；

2. 定义函数 : : → 表示取起始节点的符号值。

3.3.2 表达式分析算法定义为二元组 (,)，其中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； 是布尔表达式，它表示当值流图上某节点的条件为真时，其取值为 。

定义函数 : : → 表示取中的值；函数

: : → 表示取中的值。

定义抽象函数 : : → 为从值流图节点到抽象域上的一个映射，其中为的集合。则基于值流图的表达式分析算法如算法7所示。 41

第 3 章基于值流图的整型变量关系分析与缺陷检测

Algorithm 7 值流图分析算法Require: : current value flow graph’s node set 1: ←

2: ← { | ∈ , is ConstNode or StartNode }

3: while ≠ do

4: ← peek( )

5: ← updateNodeValue( )

6: if = then 7: ← ∪

8: else if is an update for then 9: ← ∪ ( )

10: move each from to which is unblocked due to

11: end if

12: end while

其中，函数 : : → 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 {(( ) ,)}

StartNode {(( ) ,)}

CopyNode {((.))}

OperatorNode (( ) , (1), ..., ( )) , (1)∧ ... ∧ ( ) | | | | | ∈ ( ), = ( ) [ ]

JoinNode (Loop) {(,) | ← ∪ (( ))}

JoinNode (Normal){(( ), ( )∧ ( )) | ← ( )}

本算法属于值流图分析算法，定义了两个集合和 ，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，将集合置空，并将所有类型为和的节点置入集合中。在每次迭代计算过程中，从集合 42

第 3 章基于值流图的整型变量关系分析与缺陷检测

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集合中，并检查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。

3.3.3 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。

Inputs
Outputs
N4 x
N24 icmp slt
N4
N23
N27 icmp sgt
N4
N26
N32 icmp slt
N4
N23
N34 icmp sgt
N4
N26
N5 y
N18 icmp slt
N5
N17
N21 icmp sgt
N5
N20
N6 diverse
N46 icmp ne
N6
N15
N7 oldGearN59
N15 0
N17 160
N18
!(N18)
N20 780
!(N21)
N21
N23 380
!(N24)
N24
N26 690
N27
!(N27)
N28 3
N29 5
N30 1
!(N32)
N32!(N34)
N34
N35 4
N36
6N37 2
N38
αN39
αN40
αN41
αN42
α N43
α
!(N46)
N46
N51 -1
N56
α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。

由于每个模块的输入节点集合 ( ) 描述了模块所有可能的读入、输出节点集合 ( ) 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 ( ) 中的节点 5

9为复制节点，其计算结果与 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 43

第 3 章基于值流图的整型变量关系分析与缺陷检测

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 3.3 值流图上部分节点的计算结果 ( ) 4{(,)} 2 6{(690,)} 3 4{(> 690,)} 3 8{(4, ≤ 690) , (6, > 690)} 4 1{(1, < 380) , (3, 380 ≤ ≤ 690) , (5, > 690)} 4 2 { (0, ≤ 780) , (1, < 380 ∧ > 780) , (3, 380 ≤ ≤ 690 ∧ > 780) , (5, > 690 ∧ > 780) } 5 6 (1, ≠ 0) , (0, 160 ≤ ≤ 780 ∧ = 0) , (1, < 380 ∧ > 780 ∧ = 0) , (2, < 380 ∧ < 160 ∧ = 0) , (3, 380 ≤ ≤ 690 ∧ > 780 ∧ = 0) , (4, 380 ≤ ≤ 690 ∧ < 160 ∧ = 0) , (5, > 690 ∧ > 780 ∧ = 0) , (6, > 690 ∧ < 160 ∧ = 0) ,

3.4 评估与结果

3.4.1 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3.5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。 44

第 3 章基于值流图的整型变量关系分析与缺陷检测

可配置静态分析框架

模块摘要展示

数据流迭代算法可配置程序分析组件控制流自动机函数调用图数据流图

值流图生成算法模块摘要生成算法

摘要输出接口

命令行 IDE插件图形界面图 3.5 工具架构图

3.4.2 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3.4）；（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 3.4 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 45

第 3 章基于值流图的整型变量关系分析与缺陷检测

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3.5所示。

表 3.5 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

代码行分支数分支数率类型耗时/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。
3.4.3 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3.4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3.6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。 46

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 3.6 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3.7。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 3.7 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
3.5 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 ，算法的结果是带的符号表达式，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 47

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。 48

第 4 章总结与展望

参考文献

[1] Wagner S, Abdulkhaleq A, Bogicevic I, et al. How are functionally similar code clones syntacti- cally 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. 2013.
[4] Ramler R, Biffl S, Grünbacher P. Value-based management of software testing // Value-based software engineering. 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 elipse 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 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.
[9] 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.
[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 B, Miné A. Apron: A library of numerical abstract domains for static analysis // Interna- tional Conference on Computer Aided Verification. 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. arXiv preprint cs/0612085, 2006.
[14] Aho A V, Sethi R, Ullman J D. Compilers, principles, techniques. Addison wesley, 1986, 7(8):9.
[15] Cooper K D, Harvey T J, Kennedy K. Iterative data-flow analysis. Revis ited, 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, Bj rner N. Z3: An efficient smt solver // International conference on Tools and
Algorithms for the Construction and Analysis of Systems. 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. 50
参考文献
[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 (TOSEM), 1997, 6(4): 370-397.
[22] Schugerl P, Rilling J, Charland P. Beyond generated software documentation—a web 2.0 per- spective // 2009 IEEE International Conference on Software Maintenance. IEEE, 2009: 547-550.
[23] Karrer T, Kr mer 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 visu- ally 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 summariza- tion // 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 se- quence and circular bundle views // 15th IEEE International Conference on Program Compre- hension (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 Visu- alization (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. Springer, 2007: 504-518.
[31] Cheng B C, Hwu W M W. Modular interprocedural pointer analysis using access paths: de- sign, implementation, and evaluation // Proceedings of the ACM SIGPLAN 2000 conference on
Programming language design and implementation. 2000: 57-69.
[32] Boysen J P. Factors affecting computer program comprehension. 1979.

[33] 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.
[34] Binkley D W, Gallagher K B. Program slicing // Advances in Computers: volume 43. Elsevier, 1996: 1-50.
[35] Sulír M, Porub n J. Labeling source code with metadata: A survey and taxonomy // 2017 Fed- erated Conference on Computer Science and Information Systems (FedCSIS). IEEE, 2017: 721- 729.
[36] Maalej W, Happel H J. Can development work describe itself? // 2010 7th IEEE working con- ference on mining software repositories (MSR 2010). IEEE, 2010: 191-200.
[37] 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. 51