

**高等软件工程**

**课程论文**

姓名： 李越

学号： G20178713

专业： 计算机技术

学院： 计算机与通信工程学院

成绩：

任课教师： 孙昌爱

2018年6月

软件调试是软件开发过程中一个非常重要的环节，也是一项既耗时又费力的工作。随着软件规模的不断增大和功能的不断增强，软件故障很难避免，但是未被发现的软件故障有可能会造成非常严重的后果，所以软件质量对于软件来说至关重要。

在软件开发和维护过程中，80%的人力、财力、时间等资源消耗在软件调试中，主要是为了减少软件中的故障。定位软件的故障的传统方法通常需要比较有经验的程序员，凭借个人开发经验来进行人工调试，但无论对初级程序员还是高级程序员来说，发现软件中的故障并能够进行准确的定位和修正是一项任务繁重的工作，占用了整个软件开发的大量时间。

随着人工智能的发展和应用，软件工程领域中的自动化软件故障定位技术也取得了一定的发展，大大降低软件调试和维护的成本，具有非常重要的意义。现在有许多故障定位技术已经被提出，并且在理论和实践方面都取得了进展，故而对当前最先进的故障定位技术进行概述，为这个领域的研究者提供参考是很有意义的。

《A Survey on Software Fault Localization》是发表在IEEE Transactions on Software Engineering2016年8月刊上的一篇综述性文章，这篇文章对故障定位技术做了全面的概述，并讨论了各种软件故障定位相关的关键问题和未来的展望。在这里主要介绍一下前三部分比较基础的内容，以作了解。

文章的第一部分介绍了主要工作和文章脉络，文章创建了一个出版库，其中包括从1977年到2014年11月发表的331篇与故障定位技术相关的论文，并且对近年关于故障定位技术的研究趋势做了描述。尽管文章想要尽可能全面地涵盖尽可能多的故障定位技术，但不论广度还是深度，都不能涵盖所有这些文章。所以在文章中主要关注的是定位Bohrbugs的技术。那些用于定位Mandelbugs的技术，则不包含在文章关注的范围内。

那么文章聚焦定位Bohrbugs的技术的原因是什么呢？我们先来明确一下Bohrbugs的定义，可重现故障统称为Bohrbugs。接下来我们分析一下原因，首先，绝大多数我们遇到的故障都是可以重现的，聚焦定位Bohrbugs的技术可以满足我们在软件调试时遇到的绝大数问题；其次，我们需要故障是可预知的。也就是如果让软件在相同的条件下运行，我们可以让故障再次显现。例如，如果程序遇到空指针异常会崩溃，那么，在我们设定相同的输入条件时，可以让故障重现。

文章中还提到了Mandelbugs，所以我对故障的类型做了一个简单的了解，首先关于程序bug的划分规则最早是由美国计算机科学家、图灵奖获得者詹姆斯·尼古拉·格雷(Jim Gray)，在他的著名的论文《Why do computers stop and what can be done about it?》中提出的，文章中提出了比如玻尔bug(Bohrbug)、 海森堡bug(Heisenbugs)等用著名科学家名称命名的故障。我们已经知道了什么是Bohrbug，接下来再简单介绍一下其他几种比较常见的故障。

海森堡故障(Heisenbug)：我们有可能在调试程序时遇到过这种情况：有个故障让程序崩溃了，但当程序重启后，这个故障消失了。不论你用多少的时间和精力来试图让故障重现，故障仿佛没有存在过。当我们去探测、隔离一个故障时，我们的操作会导致故障改变它们的行为表现，这些故障就叫做海森堡故障(Heisenbugs)。

薛定谔故障 (Schroedinbug)：我们有一个可以运行的程序，但在我们去浏览它的代码时，发现里面是存在故障的，并且程序完全不可能在存在这个故障的情况下运行。当我们再次运行这个程序时，程序果然是不能正确运行的，这样的故障被叫做薛定谔故障 (Schroedinbug)。

曼德博故障(Mandelbug)：当故障产生的原因过于复杂而难以理解时，故障的出现也变得没有规律，这些叫做曼德博故障(Mandelbug)。例如，操作系统中基于任务调度产生的故障，包括如性能错误，内存泄漏，软件膨胀和安全漏洞等都属于曼德博故障(Mandelbug)。文章中也提到了这种故障，并说明这不是我们聚焦的关键。

文章的第二部分简单介绍了传统直观的故障定位技术。包括程序日志记录，断言，断点和分析。程序日志记录是通过在程序中嵌入类似打印语句中以可视化变量值以及其他程序状态信息。当检测到异常程序行为时，开发人员根据保存的日志文件或打印的运行时信息检查程序日志，以诊断潜在的故障原因；断言是在程序运行期间添加到程序中的约束条件。开发人员在程序代码中将这些约束条件指定为条件语句，如果它们评估为false，则会终止执行。因此，它们可用于在程序运行时检测错误；断点用于在执行到达指定点时暂停程序，并允许用户检查当前状态。触发断点后，用户可以修改变量的值或继续执行以观察错误的发展。数据断点可以在指定表达式的值发生变化时触发，例如变量值的组合；性能分析是运行时关于指标的分析，比如对执行速度和内存使用率进行分析，通常的作用是针对程序做出优化。但是它也可以用于调试活动，比如检测不同功能的意外执行频率，识别内存泄漏或意外执行不良的代码。

传统直观的故障定位技术存在的问题是通常需要检查大量的代码，人工进行检查效率比较低，并且对程序员提出了比较高的要求，需要人工决定在何处插入打印语句，并决定对其中的哪些变量值进行打印，这需要对程序十分了解的程序员进行测试，如果交由专门的软件测试人员来进行操作的话，可能在代码理解上就要花费相当长的时间，尤其是对于代码量较大的程序，并且人工选择的主观性很大，即便选择出来可能对故障的排除也没有很大的实际意义。故而第二部分不是文章着重介绍的部分，更先进和复杂的故障定位技术是文章更关心的内容。

文章第三部分介绍了更先进的故障定位技术，并将其分为八类，包括基于切片的，基于程序谱的，基于统计的，基于程序状态的，基于机器学习的，基于数据挖掘的，基于模型的故障定位技术和其他技术。其中对于应用相对广泛的基于程序切片的故障定位方法及上面文章中提到的基于程序谱的故障定位方法做比较详细的叙述，其他定位方法简单介绍思想。

基于程序切片的技术主要介绍了静态切片技术，动态切片技术以及执行切片技术。程序切片是一种常用的调试方法，通过切片减小程序员定位程序中故障的搜索范围。一个测试用例执行结果错误是由于程序语句中执行了不正确的变量值而引起的，这种错误往往出现在与该变量语句配对的静态切片中，由于希望进一步减小可能出现故障位置的搜索范围，Lyle和Weiser通过构建一个程序模块（为两组静态切片做交集并集的操作）来解决这个问题。Korel和Laski又提出了动态切片的概念，它只考虑程序的某个特定执行情况，程序中的信息如数组、指针和循环依赖关系都可以在程序执行时动态确定。因此，动态切片与静态切片相比结果更加的准确。文章还介绍了将动态切片与基于模型的诊断相结合的故障定位技术；多点动态切片技术，将反向动态切片（BwS），前向动态切片（FwS）和双向动态切片（BiS）结合起来；面向情景的程序切片技术等动态切片定位方法以实现更精准的故障定位。关于动态切片技术的限制—不能对遗漏的错误进行识别，提出了相关切片的方法，即构造并增强动态依赖关系图，并通过在增强动态关系图上进行不正确输出的传递闭包来计算相关切片，但是，程序语句之间的错误依赖关系可能会产生过大的相关切片。为了解决过大相关切片的问题，引入了隐式依赖性的概念，使得依赖关系可以通过谓词切换获得，即使用了双重切片，动态切片和跟踪差异的组合。另一种技术是使用基于数据流测试的执行分片来定位程序错误。

用代码示例对三种切片方法的效果做了差异比较，反应出切片技术的一个问题是错误可能不在切片中，对这个问题介绍了基于块间数据依赖性的增强和细化方法去解决；对于切片内容的冗长和难以理解的问题，提出了用精细切片，以及使用屏障的概念为较小的程序切片提供一种过滤方法去优化。

基于程序谱的故障定位方法是通过分析程序的动态运行信息来进行故障定位的方法。它通常对成功运行和失败运行中程序状态特征的统计信息进行比较,直接定位到故障语句。目前常用的程序实体主要有语句和分支,它们的主要载体是程序谱信息。利用程序谱信息获取到的程序实体的特征信息来计算程序实体的可疑度,然后根据可疑度大小对被检测的程序实体进行降序排序。最开始Renieris和Reiss提出了一种基于程序谱的近邻模型，思路是比较失败的测试用例的程序谱与通过的测试用例的程序谱，找出与失败测试用例程序谱最为相似的通过测试用例程序谱，再通过比较两者的异同来定位错误。但这类方法对于通过测试用例执行的代码都是怀疑度更低，但实际操作中是有问题的。为了解决这个问题提出了著名的Tarantula相似度系数方法。该方法的指导思想是，被更多失败的测试用例覆盖到的语句，相对于被更多通过的测试用例覆盖的语句，更应被怀疑。Abreu等人之后提出了与Tarantula类似的Ochiai相似度系数方法，而后又有很多基于这种方法的研究，文章中对其中31种技术做了简单比较。除了主流的基于程序谱的方法，文章还对其他一些比较小众的方法比如程序不变量频谱（PIHS），谓词计数谱（PRC）等做了总结。

我认为这种方法应用广泛，类似于Tarantula和 Ochiai 这一类相似度系数的方法计算速度很快，通过定义好公式，将程序谱 0-1 矩阵输入，能够很快得到可疑度排名。因此这类轻量级的方法是研究的重点领域，故而基于这种相似度系数的方法比较多。但是，这类方法也有不足的地方，比如他们对于代码的针对性并不强，将所有代码等同看待，并没有对代码作一个客观的、先验的评价，以辅助错误定位。而谓词计数谱（PRC）中的一种方法SOBER，将谓词作为其研究的重点，这种方法认为谓词是程序中较为容易出错的片段，因此通过研究针对谓词的错误定位相对比较有目的性。

基于概率统计的故障定位方法的主要思想是：程序中每一个可执行实体都可能存在故障，但可疑程度不同，可以根据概率统计原理，计算每个程序实体存在故障的概率。介绍了SBI和SOBER的方法，他们在程序中使用工具和谓词的评估来产生一个可疑谓词的排序，进而检查发现出现故障的位置。但是这些方法是受限于谓词样本的，根据这种局限，等提出了一种基于交叉表的方法，它使用每条执行语句的覆盖信息和每个测试用例的执行结果。

基于程序状态的故障定位方法的主要思想是：程序状态由程序变量、系统参数、内部事件、用户输入等组成，并随程序指令执行而变化。根据测试时成功执行和失败执行的程序状态变化作为良好的故障指示器。

基于机器学习的故障定位方法的主要思想是：机器学习技术可以基于数据生成模型，可以向机器中输入大量可以作为识别或推断故障的位置的数据特征及执行结果，从而得到我们想要得到的数据特征同故障定位间的关系。

这篇文章虽然对故障定位技术做了总体性的概括，但是对于每种故障定位技术都没有做深入的探讨，对于具体解决问题的介绍比较笼统，我希望看到一个与故障定位技术相关的在实际中应用的案例，所以阅读了《A Test-suite Diagnosability Metric for Spectrum-based Fault Localization Approaches》，这篇文章是发表在2017年International Conference on Software Engineering会议上的论文，旨在通过创建完整的测试套件来提高故障定位的准确程度，文中提出了一个名为DDU的度量标准，通过量化测试套件的可诊断性对充分性测量进行补充，也就是在测试失败的情况下应用基于程序谱的故障定位方法来查明代码中的错误。

基于工程越来越大，代码量越来越多的今天，我们希望有切实有效的软件故障定位技术帮助我们进行软件调试，所以基于各种方法的故障定位技术应运而生，上面也介绍了相对主流的几种，但提高故障定位准确度的方法并不应该只关注原理或思想，我们在进行故障定位必然要用测试套件来评估其准确度，如果考虑对测试套件做出优化的思路，也能提高故障定位准确度。

我们传统的对于测试套件的度量标准大多是关注套件组涵盖的组件的数量，例如分支，路径。常用的标准有分支路径覆盖率和条件覆盖率及突变覆盖率等，但是这种度量是一个浅层次的标准，既没有明确测试套件实际应该怎么样执行这些组件，作为整体程序的测试套件也不确定是否能够提供程序中的关键信息。所以关于测试套件度量标准的优化是很有必要的，使得测试套件不仅能作为错误检测机制，而且也能作为有效的诊断辅助工具，从测试套件的角度提高故障定位技术的准确度。

之前的测试套件的可诊断性研究提出了评估基于程序谱的故障定位技术的诊断效率的标准有两种，一种是测量使用测试覆盖矩阵的密度，在密度值接近一个最佳值时，基于程序谱的方法是最有效的；另一种方法是试图减小动态基本块的大小以提高故障定位精度。但是这两种诊断方法都有缺点，首先，密度的方法的假设与实际情况中是有出入的，比如密度方法假设所有测试程序员通过代码编写不同的路径，因此产生不同的覆盖模式。但在实践中，测试覆盖相同的代码是很常见的。其次，对于减小动态基本块的大小的方法实际是侧重于检测系统中的单一故障，但实际中系统的故障是多重的。

我认为在这样的背景下，对于测试套件的优化是有意义的，文中是让测试套件包含一个测试用例，这个用例可以对系统中各种可能的组件进行组合，从而生成一个覆盖矩阵，这样做不仅可以查明单一故障，还可以查明多重故障，弥补以前诊断标准的不足，但我觉得也存在一个问题，因为在任何现实世界的系统中运行每个组件的组合都需要大量的测试用例，这样的代价是比较大的。

文中之后通过生成的覆盖矩阵得出三个关键属性矩阵：密度，多样性和唯一性，通过这三个关键属性矩阵来保证诊断准确性。这个度量标准就是DDU的度量标准，他能在处理好传统方法遇到问题的同时，也保证了易处理性。

我觉得DDU的度量标准是能够实现比较实用和有效的诊断的，从原理上讲，选取的三个度量相对比之前的标准更加有针对性也更全面，首先，密度特征可以确保组件可以经常参与测试，不会出现测试分布不均匀，从而测试的结果不充分，导致有些故障被漏检；其次，多样性特征可以确保组件之间可以以不同的组合来进行测试，这样即使使用普通测试用例不会出错，但是经过组合不同测试用例能够充分发掘程序中隐藏的故障，保证了一定数量不同粒度的测试；最后，唯一性特征可以解决两个或者两个以上的组件不明确的问题，组件如果不明确就会形成一个歧义组，如果在测试中我们不能区分出歧义组会导致诊断失败，所以对组件进行唯一性的处理还是很有必要的。DDU的度量标准并不是希望取代传统的分支覆盖等等的标准，而是希望能在基础标准上做一个补充，从而提高诊断效率。

这篇文章希望从完善测试套件的角度对故障定位技术的准确度做了提升，是一个比较新颖的思路，从测试结果上来看确实是切实有效的，诊断性能提升了大约34%。关于这篇文章我仍然有几个疑问，首先，在生成覆盖矩阵时，我们要实现对各种可能的组件进行组合，如果组件的数量多的话，对我们生成覆盖矩阵是代价比较大的，虽然在后期我们提取覆盖矩阵的三个关键属性矩阵可能会简化一定的操作，但计算代价的问题，我认为值得关注。第二，这篇文章提出的DDU度量标准是基于程序谱的故障定位方法，并且在这种定位方法下诊断性能有所提升，虽然基于程序谱的诊断方法是比较主流且有效果的方法，但是其实故障定位的方法已经发展的比较成熟，除了程序谱的定位方法还有许多高效准确的方法。对于一个好的方法我们的一个比较重要的评判标准就是泛化的能力，DDU的度量标准是一个对于其他故障定位方法依然有效的标准吗，这是我的另一个疑问。第三个疑问是，这篇文章给出的诊断性能提升的数据是基于两个代码片段和几个测试用例的运行信息，我认为实验数据比较少，从而比较有指向性，有可能并不能充分的证明这个度量对于大多数基于程序谱的故障定位方法的诊断性能都有所提升。

通过阅读《A Survey on Software Fault Localization》让我对于软件调试的背景，传统故障定位方法以及先进的故障定位方法有了宏观的认识和了解，但对于具体问题的应用，最新技术的突破并不了解，进而阅读了《A Test-suite Diagnosability Metric for Spectrum-based Fault Localization Approaches》这篇文章让我对于故障定位技术的优化和改进有了新的认识，他的思路是从测试套件的方向来改进故障定位技术，我觉得比较新颖。这两篇文章都为我带来了很大的启发，虽然我的研究方向并不是软件工程，但是在研究过程中，很多方法都是相通的，比如第二篇文章，大多数文章对于故障定位的改进方法都是从主要思想入手，在模型或方法上做一些突破。这样的方法固然很好，但是有时研究成本比较高，如果不是对于这方面十分了解并且有创新的观点，想要在这个方向上实现突破并不是一件容易的事情。如果我们换个方向想问题，从测试套件的方向去优化这个问题，也许能够以更小的研究成本达到优化故障定位方法的目的。在我们的研究过程中，如果能换个角度去思考问题，也许能得到意想不到的收获。