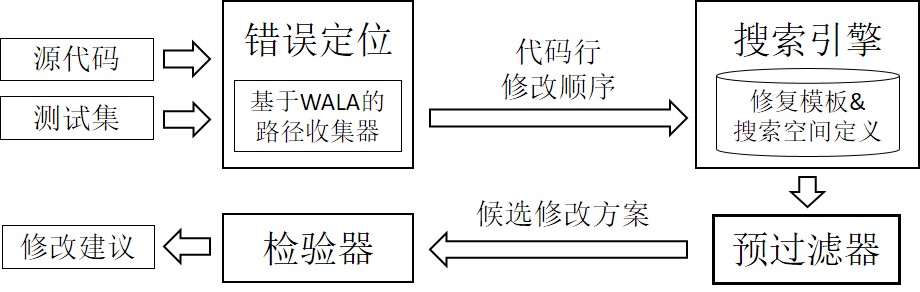
xDebug技术文档

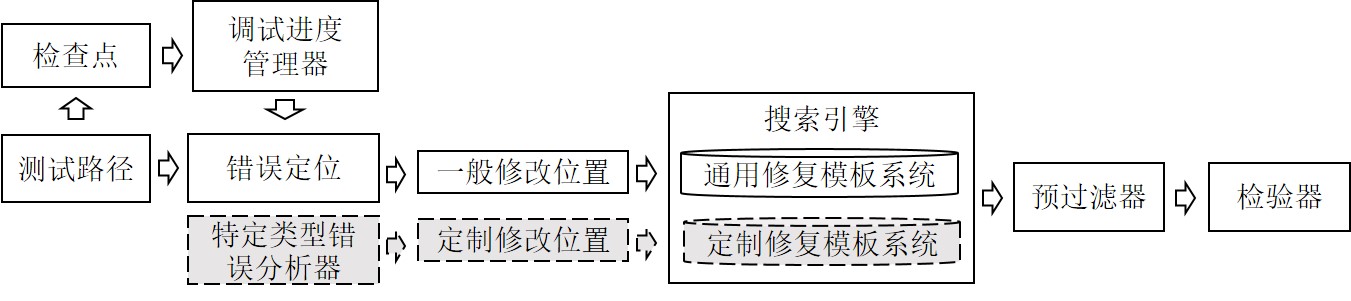
1. xDebug基本结构

xDebug是基于“生成-检验”框架的Java程序自动修复系统。其基本结构遵循“生成-检验”框架，即包含一个错误定位前端，一个搜索引擎及一个验证器。在此框架下，我们针对搜索引擎做了优化 ，形成的系统成为PFDebug，同时我们对框架进行了扩展，包括引入开发人员对程序运行状态判断的SmartDebug，以及融合针对特定类型错误修复算法的扩展框架及其实例实现NPEDebug。

PFDebug：



SmartDebug&NPEDebug：



1. 依赖库及其使用方式

xDebug的开发主要依赖于eclipse平台。在系统的全部工作流中都与eclipse JDT插件密切相关。此外在错误定位部分我们用到了静态分析库Wala的部分功能。

* 1. JDT

JDT是eclipse处理Java语言对应的插件。我们主要用到其中的两个部分，分别是处理Java代码文本的AST（抽象语法树，Abstract Syntax Tree）相关接口（主要负责处理Java语言本身），以及JDT debug组件（主要负责工程编译、运行、调试信息获取等）。具体接口说明可以参考eclipse help: https://help.eclipse.org/neon/index.jsp

* + 1. AST: JDT Plug-in Developer Guide/Programmer's Guide/JDT Core/Manipulating Java code
    2. Debug: JDT Plug-in Developer Guide/Programmer's Guide/JDT Debug
  1. Eclipse Platform

除JDT相关组件外，我们依赖了eclipse平台本身提供的功能，主要包括工程管理、文件及其他资源管理相关接口。具体说明可参考eclipse help:

* + 1. Resources: Platform Plug-in Developer Guide/Programmer’s Guide/Resources Overview
  1. Wala

Wala是Java语言静态分析工具库，我们主要利用了它提供的二进制代码处理接口和切片接口。具体内容可参见：http://wala.sourceforge.net/wiki/index.php/Main\_Page

* + 1. Trace

在错误定位模块，我们需要得到测试用例的覆盖率信息，因此需要对源代码和测试代码进行插桩。这部分依赖的接口主要在wala shrike，可参考：http://wala.sourceforge.net/wiki/index.php/Shrike\_technical\_overview

* + 1. Slicer

在开发NPEDebug时我们需要分离出与空指针异常相关的一条代码执行路径，因此利用了Wala提供的反向切片功能，这部分依赖的接口主要在wala.core中的Slicer类，其使用方法可参考：http://wala.sourceforge.net/wiki/index.php/UserGuide:Slicer

1. 通用模块
   1. 代码结构

xDebug的核心代码包括以下组件：

cn.edu.thu.tsmart.tool.da.core

cn.edu.thu.tsmart.tool.da.expr4j

cn.edu.thu.tsmart.tool.da.instr

cn.edu.thu.tsmart.tool.da.tracer

cn.edu.thu.tsmart.tool.da.ui

cn.edu.thu.tsmart.tool.da.validator.ui

cn.edu.thu.tsmart.tool.da.wala

其中，instr包含一个插桩器，在被测程序运行时被Java Runtime调用对代码进行插桩。tracer包负责分析插桩获取的信息，将其翻译成真实的执行路径，供错误定位模块分析使用。core中包含了系统核心算法，包括错误定位->搜索修改建议->验证的整体流程、修复模板系统、预过滤算法等。ui和validator.ui均是界面相关包，其中ui定义了路径展示界面、修复建议展示界面，validator.ui则定义了与SmartDebug相关的checkpoint及其展示相关的界面。expr4j是为了方便PFDebug在defects4J上做实验开发的批处理程序。wala是对我们所依赖的Wala工具包的封装。

以下我们将分别介绍几个关键包的代码组织结构和功能。

* 1. Instr与trace

我们利用java agent机制获取测试程序执行过程。其中agent机制介绍可参见：<https://docs.oracle.com/javase/7/docs/api/java/lang/instrument/package-summary.html>

Instr中包含了一个Instrumentor，它在wala对字节码分析基础上，分别对以下几种情况进行打印记录：

1. 方法进入和退出
2. 跳转
3. 方法调用

打印结果为一行字符串，内容包括时间戳、事件类型、代码位置等相关信息。Instrumentor动态将“打印”这一操作插入到这些事件之前，并且将打印的输出流重定向到一个socket中，socket的服务器端接收这一数据流并对它进行分析，在字节码的基本块层面重建执行路径（DynamicTranslator）。重建算法基本思想是在字节码callgraph上找到一条能够符合所打印的事件流的执行路径。

路径以一个嵌套的数据结构（TraceNode、InvokeTraceNode）记录，每个InvokeTraceNode包含了一个方法在执行过程中经过的多个基本块（TraceNode）以及其内部调用其他方法的信息（InvokeTraceNode）。后续错误定位将对这些路径信息做展开处理。

* 1. Core

Core包括了系统核心的数据结构和算法。其中BugFixSession定义了一个修复会话中所需用到的基本数据结构，Fault localizer、Search engine和Validator分别对应生成-检验框架的三个主要模块。

* + 1. BugFixSession

BugFixSession是一个“数据仓库”，包含了一个session进行计算所需要依赖的几乎全部数据，例如测试路径、测试结果、被测工程基本情况，同时也包含了一些全局可访问的工具类，例如SearchEngine，Validator，Filter，Expression Generator，FixSiteManager，Logger等等。

在每次计算开始前， 要初始化BugFixSession中所依赖的数据，测试路径的翻译和错误定位也在这一过程中完成。

* + 1. Fault localizer

Fault localizer采用了SFL中Ochiai这一公式进行计算。路径覆盖的基本单元是字节码级别的BasicBlock。

一般的算法实现思路是，获取所有路径，每条路径存储在某个basic block到覆盖与否的映射，最终统计所有basic block的覆盖情况，使用Ochiai进行计算。实际实现过程中我们发现，路径全部存在内存中是不现实的。例如在defects4J中，一条路径常有十万个以上节点，而若有几百个测试用例同时存在，则会出现内存不足无法计算的情况。

我们解决这一问题的办法是：尽量不在内存中存储测试路径，而只从路径中提取必要的数据。具体来说，从SFL的计算过程只需要反映覆盖情况的“统计数据”，因此我们可以通过“插桩一个测试-》获取一条路径-》统计一次数据-》扔掉这个路径”这种方式使得内存中理论上只同时保存一条路径。

考虑到后续搜索过程需要basicblock的具体信息，我们还是需要在排序列表中保存每个排位上对应的basicblock。考虑到仅被执行正确的测试用例覆盖的basicblock不在修复的考虑范围内，我们首先记录错误测试用例所覆盖的basicblock，对正确测试用例执行路径中覆盖的basicblock，如果它没有出现在错误测试覆盖的basicblock中，就将它忽略。

经过以上两个步骤的优化，实际的测试显示，在defects4J上错误定位这一步占用的内存通常不超过8G。

* + 1. Search engine
    2. Fix pattern
    3. Expression generator
    4. Validator
  1. ui与validator.ui

1. PFDebug
   1. Filter
   2. 表达式求值等价类
2. SmartDebug
   1. 断点触发数据统计
3. NPEDebug
   1. 定制化代码结构