一个可半自动化扩展的静态代码缺陷分析工具

梁广泰 孟 娜 李进辉 钟 浩 张 路 王千祥
(北京大学信息科学技术学院高可信软件技术教育部重点实验室 北京 100871)

摘 要 基于缺陷模式的代码缺陷分析技术根据预先设定的缺陷模式知识对受检代码进行缺陷分析.这种分析技术具有使用简单、查找速度快等优点,是近年来静态代码缺陷分析方法中发展比较迅速的新技术.但是目前基于这种分析技术的大多数工具并没有为用户提供足够易用、高效的扩展方式以扩充其缺陷检测能力.针对这一问题,作者提出了一个"可半自动化扩展"的代码缺陷静态分析方法,设计并实现了一个支持该方法的工具——CODA (COde Defect Analysis tool).CODA 不仅提供了"缺陷模式描述模板"以帮助用户快速地手工扩充缺陷模式库,还能在用户的指导下半自动化地挖掘新缺陷模式以快速扩充其缺陷模式库.一旦新的缺陷模式被定义并添加至缺陷模式库中,CODA 便能自动具有针对该类缺陷的检测能力.

关键词 静态分析;缺陷分析;半自动化扩展;缺陷模式
中图法分类号 TP311 **DOI 号**: 10.3724/SP.J.1016.2011.01114

A Semi-Automatic Extensible Static Defect Analysis Tool

LIANG Guang-Tai MENG Na LI Jin-Hui ZHONG Hao ZHANG Lu WANG Qian-Xiang
(*Key Laboratory for High Confidence Software Technologies of Ministry of Education, School of Electronics Engineering and Computer Science, Peking University, Beijing* 100871)

Abstract The pattern based code defect analysis approach finds defects for subject programs with the aid of predefined defect pattern knowledge. The advantages of this kind of approach lie in the simplicity of its usage and the efficiency of its analysis, which make it a new technique with more rapid development among different approaches of static code defect analysis recently. However, among the available tools based on the approach, the extension modes provided by most of them are neither friendly nor efficient enough for users to extend their defect analysis capability. In order to solve this problem, the authors proposes a semi-automatic extensible static code defect analysis approach. Based on the approach, a tool named CODA (COde Defect Analysis tool) has been designed and implemented. In order to support the efficient extension of the defect pattern library, CODA provides not only enough "defect pattern description templates" to facilitate users' manual extension, but also a "semi-automatic extension mechanism" which accelerates the process of discovering, summarizing and extending new defect patterns. Once a new defect pattern is defined and added into the defect pattern library, CODA can automatically own the detecting ability for its related defects.

Keywords static analysis; defect analysis; semi-automatic extensible; defect pattern

---

收稿日期:2009-12-21;最终修改稿收到日期:2010-07-24.本课题得到国家自然科学基金重点项目(61033006)、国家"九七三"重点基础研究发展规划项目基金(2009CB320703)、国家自然科学基金(60773160)、国家创新研究群体科学基金(60821003)资助.**梁广泰**,男,1984 年生,博士研究生,主要研究方向为静态代码分析、代码缺陷分析等. E-mail: lianggt08@sei.pku.edu.cn. **孟 娜**,女,1983 年生,博士研究生,主要研究方向为代码缺陷分析. **李进辉**,男,1983 年生,硕士,主要研究方向为代码缺陷分析. **钟 浩**,男,1979 年生,博士,助理研究员,主要研究方向为约束自动挖掘. **张 路**,男,1973 年生,博士,教授,博士生导师,主要研究领域为程序分析与理解、代码测试等. **王千祥**,男,1970 年生,博士,教授,博士生导师,主要研究领域为基于中间件的分析与调整系统等.

1 引　言

静态代码缺陷分析技术通过对代码进行静态分析来推测程序运行时的表现行为，从而发现代码中可能存在的缺陷[1]. 这类技术主要包括自动抽象解释[2]、定理证明[3]、模型检测[4]、符号执行[5]和基于缺陷模式的代码检查[6]等.

本文主要考虑基于缺陷模式匹配的代码缺陷查找方法. 基于模式匹配的代码缺陷查找方法主要包括如下两大步骤：首先，对已有代码中出现过的缺陷进行总结并提炼出“缺陷模式知识”；然后，采用静态分析的方法对受检代码进行“缺陷模式匹配”以确定受检代码是否包含相应缺陷，并把匹配结果以缺陷检测报告的形式呈现给用户[7]. 采用这种方法的代表性缺陷查找工具包括 FindBugs[7]、PMD①、Jlint②、Lint4j③、Hammurapi④、Matacompilation[6]、SABER[8]、DTS[9]等. 这类工具的基本工作原理如图 1 所示：在接收受检程序之后，分析工具的分析引擎根据已有的缺陷模式知识，应用分析技术对受检程序进行相应缺陷的检测，并将分析结果报告给用户.

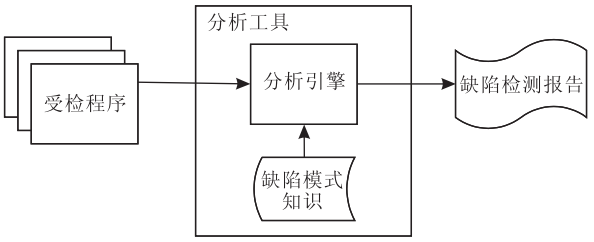

图 1　基于模式匹配的静态程序缺陷查找工具的基本工作原理图

在对现有的若干基于模式匹配的代码缺陷静态分析工具研究之后，我们发现：在使用这些工具的过程中，用户往往希望能够扩充“缺陷模式知识”，以使工具能够支持对自己所关注的“代码缺陷”进行检测. 但目前的大多数工具并没有为用户“添加新缺陷模式”提供有力的支持. 部分工具虽然支持用户扩充缺陷模式知识，但其具体的扩展方式在易用性方面还存在很大不足. 以其中最具有代表性的 FindBugs 为例：FindBugs 所关注的缺陷模式都被硬编码到工具的若干缺陷模式检查器中. 当需要使用 FindBugs 检查新的缺陷模式时，用户必须手动修改工具程序，甚至自行编写缺陷模式检查器的代码以检查新的缺陷模式. PMD 借鉴了 FindBugs 依靠“用户手工编写检测代码”的方式扩展工具能力的思想，并同时引入了 XPath 表达式来减少用户的编码量，但这种扩展方式依然需要用户花时间学习如何使用 XPath. 与 PMD 类似，Metacompilation 也同样要求用户在描述缺陷之前学习如何使用描述语言 Metal. Hammurapi 也是一个基于缺陷模式的 Java 代码缺陷查找工具，当用户需要扩展其缺陷模式时，需要基于其提供的 API 进行硬编码得以扩展. 而 Jlint、Lint4j 和 DTS 都未提供缺陷模式知识的扩充机制. SABER 通过提供“缺陷模式描述模板”在一定程度上简化了缺陷模式描述方式，但其提供的模板数量与缺陷模式描述能力非常有限.

为了探索高效易用的扩展方式，从而降低用户的学习成本、快速增强工具的错误查找能力，我们提出了一个支持“半自动化扩展”的代码缺陷静态分析方法，设计并实现了支持该方法的工具——CODA (COde Defect Analysis tool). CODA 的主要特点包括：

(1) 提供了若干不同类型的“缺陷模式描述模板”. 用户可以根据自身需要，选择适当的模板来快速增加缺陷模式. 工具能够依据用户选择的模板和填入的必要信息，自动生成符合 CODA 格式要求的“缺陷模式描述文件”. CODA 在缺陷检测过程中会自动载入这些新加的缺陷模式并在代码中检测相应缺陷. 这种扩展方式使用户不再需要手工编写代码，也不需要为描述缺陷模式而花费太多的精力学习某种语言.

(2) 提供了缺陷模式库的“半自动化扩展”机制. 利用规则挖掘器，CODA 能够从特定的库函数源代码、使用这些库函数的客户代码和库函数的 API 文档中挖掘出该库函数相关的使用规则. 利用规则转化器，CODA 能够将过滤后的有用规则转换成对应的缺陷模式并自动添加至缺陷模式库中. 缺陷模式库的半自动扩展机制不仅减轻了用户手工添加缺陷的工作负担，还为用户省去了手工收集缺陷模式的繁琐劳动，从而支持用户更快地增强 CODA 的缺陷检测能力.

本文第 2 节通过一个示例来展示 CODA 的研究动因以及 CODA 的特点与优势；第 3 节进一步阐述 CODA 的设计和实现，论述 CODA 如何实现上述特点；第 4 节在实验数据的基础上，讨论缺陷模式库的半自动化扩展机制如何帮助 CODA 快速增强

① PMD/Java. http://pmd.sourceforge.net
② Jlint. http://artho.com/jlint
③ Lint4j. http://www.jutils.com/
④ Hammurapi. http://www.hammurapi.biz/

查错能力;第 5 节总结全文,并提出对未来工作的一些设想.

2 动 因

本节首先描述 FindBugs、PMD、Hammurapi 和 SABER 等 4 个典型工具各自的缺陷模式扩展方式,然后,概要地介绍本文的解决方案.描述过程结合一个具体的缺陷模式例子进行:"代码中错误地调用了 Thread.run()方法":在 Java 程序中,只能调用 Thread.start()方法,而不能直接调用 Thread.run()方法,因为 Thread.start()会自动调用 Thread.run()方法.

(1)向 FindBugs 中添加缺陷模式

FindBugs 只支持用户通过自行编码的方式来扩充工具的错误查找能力.为添加上述缺陷模式,首先需要编写如图 2 所示的缺陷检查器(Detector)代码;然后,需要在 FindBugs 的配置文件中定义该新添缺陷模式的相关描述信息,以便于工具在检测到该类型缺陷时向用户进行报告;最后,还需要修改 FindBugs 中关于检查器的配置文件,以便工具在启动时自动载入这个新加的检查器.

```
1  package edu.pku.cs.detect;
2
3  import org.apache.bcel.Constants;
4  import edu.umd.cs.findbugs.BugInstance;
5  import edu.umd.cs.findbugs.BugReporter;
6  import edu.umd.cs.findbugs.BytecodeScanningDetector;
7
8  public class FindCallThreadRun extends BytecodeScanningDetector{
9   private BugReporter bugReporter;
10  public void sawOpcode(int seen){
11            if(seen==Constants.INVOKEVIRTUAL){
12               //To save the signature, class name and method name of the callee method.
13               String signatureCalled=getSigConstantOperand();
14               String classNameCalled=getDottedClassConstantOperand();
15               String methodNameCalled=getNameConstantOperand();
16               //To match them with the new defect pattern.
17               if(signatureCalled.equals("()V")&&classNameCalled.equals("java.lang.Thread")
18               &&methodNameCalled.equals("run")){
19               //To report the bug when found.
20               bugReporter.reportBug(new BugInstance(this,"Should_not_call_method", 3)
21                                                   .addClassAndMethod(this)
22                                                   .addCalledMethod(this)
23                                                   .addSourceLine(this));
24  }}}}
```

图 2 在 FindBugs 中需要添加的用于查找缺陷"错误地调用了 Thread.run()方法"的代码

(2)向 PMD 中添加缺陷模式

PMD 提供了两种方式来添加新的缺陷模式:编写检查代码的方式和定义 XPath 表达式的方式.但受制于 XPath 的表达能力,上面提到的具体缺陷模式只能利用编写代码的方式添加到工具中.具体地说,首先需要编写如图 3 所示的"规则"代码实现该类缺陷的查找过程,然后,为了让新添加的规则能够被工具自动加载,同样需要修改 PMD 的配置文件信息.

(3)向 Hammurapi 中添加缺陷

Hammurapi 支持用户使用其提供的 API 来扩展 Inspectors,每一个 Inspector 就是一种类型的缺陷检查器.在 Hammurapi 中添加这一新的缺陷模式,也需要通过硬编码方式实现.编码过程基本同于 FindBugs,不同的是,需要基于 Hammurapi 提供的 API 进行扩展(继承 Hammurapi 的检查器基类 biz.hammurapi.review.Inspector).

(4)本文方法概述

与 SABER 类似,CODA 也提供了若干模板供用户在描述缺陷时使用.但不同的是,CODA 提供的模板要丰富得多,几乎涵盖了大部分常见的缺陷模式类型(详见 3.2 节).为了支持用户更加快捷地添加符合模板的缺陷模式,CODA 提供了图形化界面输入方式.用户在添加缺陷模式时,只需要在选择相应模板后填入必要的信息,CODA 便能够自动生成该缺陷模式的 XML 描述文件.例如,对于上述例子,由于 CODA 提供了"代码中错误地调用 X 方法"的模板,用户在添加缺陷模式时只需要打开"图形化界面",选中相应模板并指明模板里的 X 为 Thread.run()即可.

当新缺陷模式不能通过我们提供的模板进行描述的时候,可以通过直接撰写该缺陷模式对应的"有限状态自动机 XML 描述文件"方式进行扩展(详见 3.2 节).另外,为了方便用户快速添加更为丰富的缺陷模式,CODA 还提供了一个半自动的缺陷模式挖掘器.这个缺陷模式挖掘器可以通过多种方式挖掘出大量方法调用序列相关的缺陷模式,从而支持快速扩展 CODA 的缺陷检测能力(详见 3.3 节).

```
1  package edu.pku.cs.detect;
2
3  import net.sourceforge.pmd.AbstractRule;
4  import net.sourceforge.pmd.ast.ASTPrimitiveType;
5  import net.sourceforge.pmd.ast.ASTVariableDeclaratorId;
6  import net.sourceforge.pmd.ast.SimpleNode;
7  import net.sourceforge.pmd.symboltable.NameOccurrence;
8  import java.util.List;
9
10 public class FindCallThreadRun extends AbstractRule{
11    public Object visit(ASTVariableDeclaratorId node, Object data){
12        SimpleNode nameNode=node.getTypeNameNode();
13        //if the node represents an ASTPrimitiveType variable, ignore it
14        if(nameNode instanceof ASTPrimitiveType){
15            return data;
16        }
17        //if the node's class name is not "Thread", ignore it
18        if(!appliesToClassName(node.getNameDeclaration().getTypeImage())){
19            return data;
20        }
21        List<NameOccurrence> declars=node.getUsages();
22        for(NameOccurrence occ: declars){//iterate all invocation of the thread's methods
23            if(!isTargetMethod(occ)){//if the invoked method is not "run()", ignore it
24                continue;
25            }
26            //the invoked method is "Thread.run()", report it as a violation
27            addViolation(data, node);
28        }
29        turn data;
30    }
31    private boolean appliesToClassName(String name){
32        return "Thread".equals(name);
33    }
34    private boolean isTargetMethod(NameOccurrence occ){
35        if(occ.getNameForWhichThisIsAQualifier() !=null
36        &&(occ.getNameForWhichThisIsAQualifier().getImage().indexOf("run") !=-1)){
37            return true;
38        }
39        return false; } }
```

图 3　在 PMD 中需要添加的用于查找缺陷“错误地调用了 Thread.run()方法”的代码

通过上述介绍，我们可以发现：无论是“自行编写缺陷检测代码”的方式还是“使用特殊语言描述缺陷”的方式，都对工具使用者的学习能力及应用能力提出了很高的要求. 这种来自于工具本身的实现方式妨碍了使用者根据个性需求自定义地扩展工具缺陷查找能力，降低了使用者扩充缺陷模式的效率. 与代码和描述语言相比，CODA 内置的“图形化的模板”更利于工具使用者扩充缺陷模式知识：一方面，使用自然语言定义的缺陷描述模板符合人的思维习惯，人们不需要像学习一门编程语言那样额外花时间来学习模板的使用方法；另一方面，由于每个模板含有的参数个数较少，人们在描述缺陷时需要输入的信息也较少，因此工具使用者可以较快地扩充缺陷模式知识. 除此之外，CODA 所提供的半自动化的缺陷模式扩展方式能帮助工具的使用者批量添加缺陷模式，从而加速扩展 CODA 的缺陷查找能力.

CODA 的主要优势在于其良好的易扩展性. 用户既可以使用工具提供的各种模板以“图形化方式”添加新的缺陷模式，也可以直接以“编写 XML 描述文件”的方式添加复杂的缺陷模式，还可以使用 CODA 的半自动化扩展方式批量发现、整理、添加新的缺陷模式.

3　CODA 的设计与实现

CODA 主要由缺陷查找引擎、缺陷模式库和缺陷模式挖掘器三部分组成，如图 4 所示. 缺陷查找引擎负责在受检程序中逐一应用不同类型的缺陷检查器进行缺陷检测，并将检测报告呈现给用户；缺陷模

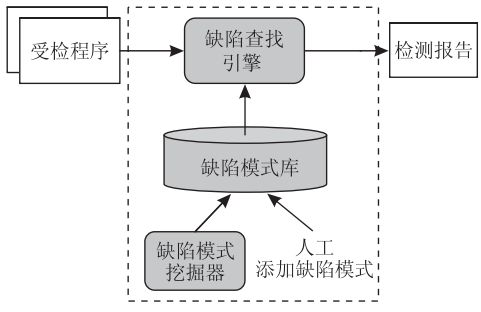

图 4　CODA 体系结构图

式库用于集中保存和管理各种缺陷模式信息;缺陷模式挖掘器负责“半自动化”地挖掘并生成新的缺陷模式信息.

3.1 缺陷查找引擎

缺陷查找引擎中包含大量检查器和分析器.“检查器”是指 CODA 中用于查找各种具体缺陷的程序模块,例如用于查找“文件流是否及时关闭”缺陷的程序模块等.“分析器”是指 CODA 中用于为检查器提供基础分析结果的各种静态分析技术,例如“控制流分析器”和“别名分析器”等.当用户提交受检程序给 CODA 时,缺陷查找引擎会逐一调用各检查器对代码进行缺陷检查,然后向用户报出所有的缺陷检测结果,如图 5 所示.缺陷查找引擎包括两大部分:分析器工厂和检查器集合.

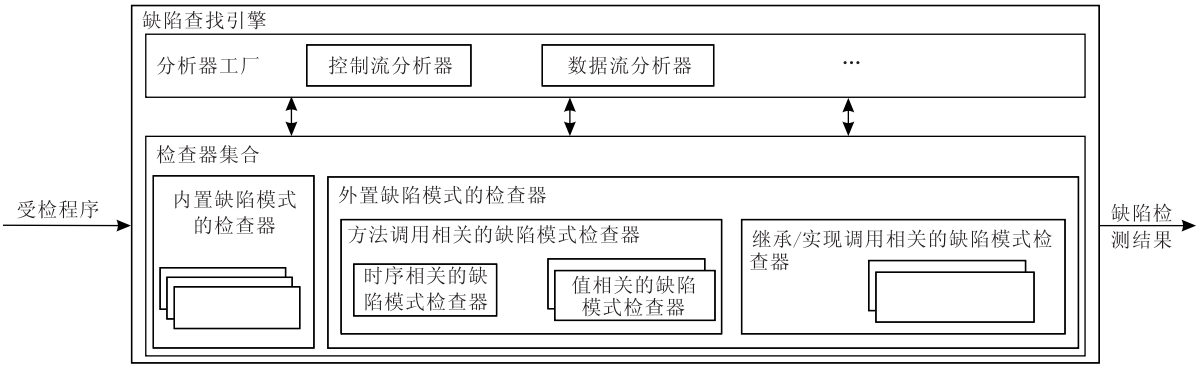

图 5 缺陷查找引擎结构图

3.1.1 分析器工厂

在缺陷模式的匹配过程中,检查器往往需要利用各种底层分析技术的分析结果来完成匹配过程或提高匹配精度,例如在检测“文件流是否及时关闭”的缺陷时,需要使用到“别名分析技术”以准确判断某一个文件流对象是否被及时关闭.CODA 中的分析器有多个,它们是对不同的静态分析技术的实现,所有的分析器组成“分析器工厂”.当某个检查器需要某个分析器的分析结果时,检查器会通过“分析器工厂”向这个分析器发出数据请求;分析器根据请求在自己的缓存中查找相应的分析结果:如果存在,则直接做出回应;否则启动分析过程生成所需结果,在结果产生后做出回应,并缓存该结果以待后续使用.

目前,CODA 中已经实现的分析器包括控制流图分析器、类型分析器、类层次结构分析、常量传播分析器、空指针分析器、锁计数分析器、活性变量分析器、调用图分析器和指向(别名)分析器.其中常量传播分析器、空指针分析器、锁计数分析器和活性变量分析器的实现属于“过程内数据流分析”,而调用图分析器和指向分析器则属于“过程间数据流分析”.

3.1.2 检查器集合

检查器是 CODA 中直接用于检查某种特定缺陷的模块.依据检查器的组织方式,我们把检查器集合中所有的检查器分成两大类:“内置缺陷模式的检查器”和“外置缺陷模式的检查器”.

(1) 内置缺陷模式的检查器

很多现有工具,如 FindBugs 和 PMD,通常只包含内置缺陷模式的检查器.这类检查器把待检测的缺陷模式信息硬编码在检查器代码中,这样使得缺陷模式信息与检查器的业务逻辑紧密关联.如果想增强该类检查器的检测能力,必须修改其源代码,或者添加自行编写的新检查器.检查器的这种实现方法大大影响了工具的可扩展性.CODA 中的内置检查器较少,主要包括 FindNullDeref、FindSleepWithLockHeld、FindWaitOrNotifyWithMultiLocksHeld、FindWaitOrNotifyWithoutLock、FindUnusedDef 和 FindWeakLoopController(参看表 1).

表 1 CODA 中“内置缺陷模式检查器”一览表

Detector 子类的名称	检查的缺陷内容	用到的分析技术
FindNullDeref	对象的空指针解引用	IsNullValueDataflowAnalysis
FindSleepWithLockHeld	在持有任何锁的情况下调用 Thread.sleep()方法	LockDataflowAnalysis
FindWaitOrNotifyWithMultiLocks	持有多个锁的情况下调用 wait()方法或者 notify()方法	LockDataflowAnalysis
FindWaitOrNotifyWithoutLock	不持有锁的情况下调用 wait()方法或者 notify()方法	LockDataflowAnalysis
FindUnusedDef	给局部变量赋值但未使用	LiveVariableDataflowAnalysis
FindWeakLoopController	多层循环结构中内部循环结构对外部循环控制变量误用	无

(2) 外置缺陷模式的检查器

外置缺陷模式的检查器是指 CODA 中那些将待检测的缺陷模式信息从错误查找逻辑中剥离出来，并集中存放在缺陷模式库中的检查器. 由于这类检查器将缺陷模式信息与错误查找逻辑进行了很好的剥离，我们可以在不修改工具代码的前提下，通过向缺陷模式库中添加新缺陷模式，达到扩展工具错误查找能力的目的.

CODA 中实现了三类外置缺陷模式的检查器："时序相关的缺陷模式检查器"、"值相关的缺陷模式检查器"和"继承/实现相关的缺陷模式检查器".

① 时序相关的缺陷模式检查器

"时序相关的缺陷模式检查器"主要负责查找"方法调用顺序"相关的缺陷模式. 这类缺陷模式的共同点在于它们几乎都可以表示成有限状态自动机的形式. 例如缺陷模式"FileInputStream 对象在使用完后需要执行 FileInputStream.close() 方法进行销毁"，其对应的有限状态自动机如图 6 所示：状态 2 为错误状态，状态 1 和 3 为正确状态. 在分析结束时，如果该状态机处于状态 2，则程序存在该类缺陷；如果处于状态 1 和 3，则说明程序未包含此类缺陷. CODA 使用 XML 描述文件对此类缺陷模式所对应的有限状态自动机进行描述(详见 3.2 节介绍).

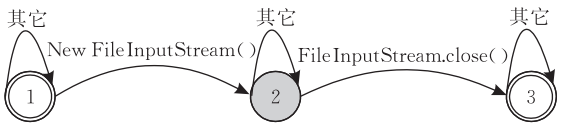

图 6　用于检查"FileInputStream 对象在使用完后需要执行 FileInputStream.close() 方法进行销毁"的有限状态机

CODA 运行时，"时序相关的缺陷模式检查器"会动态地根据这些"XML 描述"实例化有限状态自动机，并根据受检程序的实际"方法调用情况"适时驱动各个有限状态机进行状态转换. 在分析完受检程序后，若某一状态机处于错误状态，则说明受检程序中存在相应的缺陷，CODA 会将该状态机对应的"缺陷描述信息"报告给用户. 该类检查器的主要实现流程为：首先对程序中的每个方法构造控制流图，然后在控制流图上做"自动机状态"的数据流分析，最后根据自动机的所处"状态"来判断受检程序中是否存在相应缺陷.

下面对该数据流分析的要点进行介绍：

(i) 该数据流分析的对象为"不同自动机在受检程序中的所处状态". 分析过程中，CODA 会为同一类型的不同对象分别创建其各自对应的自动机. 这使得本数据流分析是对象敏感的. 例如一个方法内创建了两个文件输入流 fin1 和 fin2，CODA 会为每个输入流创建其对应的有限状态机并跟踪各个状态机的状态变化，这样当某一对象的 close() 方法未被调用时，CODA 能够定位出具体哪个对象触犯了该缺陷.

(ii) 转换函数(Transform)的定义. 为便于叙述，假设控制流图的每个块仅包含一条语句. 当遇到某一方法调用语句时，首先获得调用这个方法的对象，将这个方法调用作为转移条件，推进这个对象上的所有自动机的状态迁移. 例如，假设本方法关注的是"java.io.FileInputStream 对象上的流打开之后未关闭"的时序约束. 那么当遇到语句"FileInputStream fin = new FileInputStream()"时，就需要将"new FileInputStream()"作为状态转移条件，推进对象"fin"上的自动机到下一个状态. 同样的，如果遇到"fin.close()"语句，就将"close()"作为转移条件，推进对象"fin"上的自动机进行状态转移.

转换函数的输入是当前所有自动机的状态集合，输出是改变后的自动机状态集合.

(iii) 汇合函数(Merge)的定义. 汇合函数要决定当一个块存在多个前驱时，如何从这些前驱的"数据流输出值"得到这个块的"数据流输入值". 在该分析中，所有前驱的"数据流输出值"取并集后的自动机状态集合，会作为这个块的"数据流输入值". 采取这种策略，本检查器能够检测出特定路径上存在的缺陷. 如图 7 所示，EXIT 存在多个前驱 B2、B3，汇合后 EXIT 入口处对象 Fin 存在两个自动机状态，其中一个状态对应着路径(B1->B2->EXIT)，这个自动机状态是正确的，说明该路径不存在缺陷；另一个自动机状态对应路径(B1->B3->EXIT)，而该状态是错误状态，可以知道该路径触犯了该缺陷.

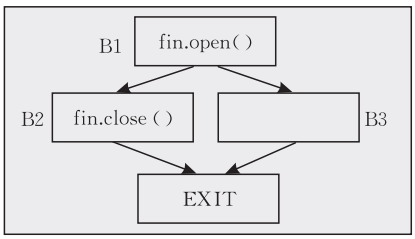

图 7　特定路径的违反时序约束的缺陷

(iv) 拷贝函数(Copy)的定义. 拷贝函数的功能就是实现从一个块的前驱的输出数据流值到这个块的输入数据流值的转换. 本方法中，直接将前驱的输出值复制到这个块的输入值即可.

② 值相关的缺陷模式检查器

“值相关的缺陷模式检查器”主要负责查找方法调用的传入传出参数值相关的缺陷模式，例如缺陷模式“java.sql.ResultSet 的 getArray()方法传入参数不能为 0”. CODA 中有多个检查器属于此类，每个检查器负责对多个缺陷模式进行查找，该类的检查器一览表详见表 2. 由于对应于同一检查器的所有缺陷模式具有相同的检测逻辑，所以我们只把能被共享的检测逻辑编写到检查器代码中，而把不能被共享的各缺陷模式信息独立定义在缺陷模式库中. CODA 会在运行时自动从缺陷模式库将这些信息加载到对应的检查器中，并使用同一检查器完成多个缺陷模式的并发匹配.

表 2 CODA 中“值相关的外置缺陷模式检查器”一览表

检查器名称	检查的缺陷类型	用到的分析器名称
FindInvalidTypeInMethodArgument	在调用某些方法时，不能传入某种类型的实例做参数	RealValueDataflowAnalysis TypeDataflowAnalysis
FindInvalidValueInMethodArgument	在调用某些方法时，传入参数取无效或非法的值	RealValueDataflowAnalysis IsNullValueDataflowAnalysis
FindVacuousComputation	无效或多余的比较运算	TypeDataflowAnalysis IsNullValueDataflowAnalysis

举例来说，缺陷模式“java.sql.ResultSet 的 getArray()方法传入参数不能为 0”与“java.sql.PreparedStatement 的 setArray()方法传入参数不能为 0”都可以表述成“A 类的 B 方法传入参数不能为 M 值”的形式，CODA 在查找它们时使用同样的匹配算法. 因此我们把匹配算法编写到检查器中，而把缺陷模式对“A”、“B”、“C”实例化的个性信息添加到相应的配置文件中.

在缺陷匹配过程中，该类检查器使用常量传播分析器和空指针分析器提供的分析结果来确定调用方法的输入输出值是否合法，使用类型分析器来确定调用方法的输入输出值的类型是否合法. 表 2 中列出了 CODA 中已经实现的“值相关的缺陷模式检查器”以及它们各自使用到的分析器名称.

③ 继承/实现相关的缺陷模式检查器

这些检查器主要负责查找在继承某个类或者实现某个接口时，类的定义、方法的定义等相关的缺陷模式，例如缺陷模式“实现 Cloneable 接口的类没重写方法 clone()”. CODA 中属于此种检查器的是 FindImplClassNoDefOrUseMethod，该检查器用于查找实现某接口或类时，未按照规范要求定义或使用某些方法. 该检查器在查找缺陷时未使用任何分析技术提供的分析结果，只使用了从字节码中提取出来的程序信息.

3.2 缺陷模式库

缺陷模式库负责保存和管理缺陷模式信息. 这些缺陷模式信息根据其所对应的检查器种类，被分为“时序相关的缺陷模式信息”、“值相关的缺陷模式信息”和“继承/实现相关的缺陷模式信息”. 它们被分别存放在不同的信息集中(如图 8 所示). 将缺陷查找能力和缺陷模式信息相分离，不仅有利于我们清晰的划分出各检查器的工作职能，还有助于用户更好地理解甚至扩展工具的缺陷模式知识. 例如，在用户发现新的缺陷模式之后，他们只需根据需要选用合适的模板来添加信息到缺陷模式库中，而无须关心缺陷查找的技术细节. CODA 为每一类的缺陷模式信息分别提供了若干模板供用户在描述缺陷时使用，这些模板几乎涵盖了大部分常见的缺陷模式类型(如表 3).

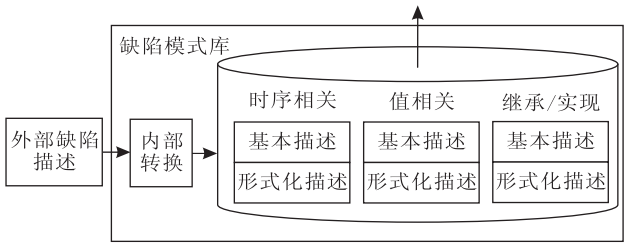

图 8 缺陷模式库结构图

表 3 CODA 提供的缺陷模式模板一览表

所属类别	模板名称
时序相关	不能调用某个方法 A 调用了方法 A 则必须调用方法 B 调用了方法 A 后不能调用方法 B 调用方法 B 之前必须调用方法 A 方法 1,2,3,…,N 必须依序调用 不要从方法 Y 中直接或间接调用方法 X 方法 Y 中必须调用方法 X
值相关	A 类的 B 方法传入参数不能为 M 值 A 类的 B 方法传入参数必须为 M 值 A 类的 B 方法传入参数不能为类型 C A 类的 B 方法传入参数必须为类型 C A 类型和 B 类型的值进行 C 类型运算为无效 A 类型和 B 类型的值进行 C 类型运行为冗余
继承/实现相关	实现接口(或类)A 的类必须实现 B 方法

为了方便用户添加，对于“时序相关”的缺陷模式，缺陷模式库还提供“图形化界面”输入方式. 用户在添加缺陷模式时，只需要在选择相应模板后填入

必要的信息并点击“添加”按钮，CODA 便能够自动生成该缺陷模式对应的自动机 XML 描述文件并存入缺陷模式库中. 对于“值相关”和“继承/实现相关”的缺陷模式，用户可以通过直接改写描述模板添加新的缺陷模式.

当新缺陷模式不能通过我们提供的模板进行描述的时候，可以通过直接撰写该缺陷模式对应的“有限状态自动机 XML 描述文件”方式进行扩展. 例如，缺陷模式“FileOutputStream 对象在使用完后需要执行 FileOutputStream.close()方法进行销毁；调用 FileOutputStream.close() 方法之后不能调用 FileOutputStream.write()方法”. 这个缺陷模式不能够直接使用上述模板进行描述(当然可以将该缺陷首先拆分成两个子缺陷“FileOutputStream 对象在使用完后需要执行 FileOutputStream.close()方法进行销毁”和“调用 FileOutputStream.close()方法之后不能调用 FileOutputStream.write()方法”，然后分别使用模板进行描述)，而 CODA 所提供的有限状态机描述方式支持对该类“复合式”缺陷模式直接进行 XML 描述. 该复合式缺陷模式对应的“有限状态自动机”和“自动机描述文件”如图 9、图 10 所示. 自动机 XML 描述文件中包括了对这个缺陷模式的描述信息(description 节点)、自动机的各个状态的描述(state 节点)、各个状态之间的转移条件(edge 节点)以及各个状态的正确与否(status 节点). 受检程序被分析后，若该状态机处于正确状态(status=right)，说明受检程序未存在该类缺陷；若当状态机处于某一错误状态(status=error)时，说明受检程序中存在相应的缺陷，CODA 会将该状态对应的缺陷描述信息(stateDescription 节点)报告给用户.

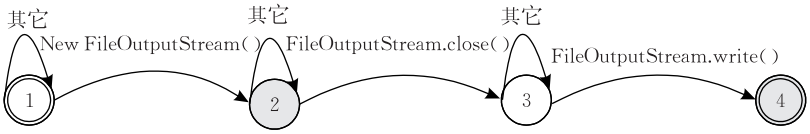

图 9　用于检查 FileOutputStream 相关“复合式”缺陷的有限状态自动机

```
〈?xml version="1.0" encoding="UTF-8"?〉
〈FSM〉
    〈description〉FileOutputStream related defects〈/description〉
    〈state〉
        〈number〉1〈/number〉
        〈status〉right〈/status〉
        〈exceptionState〉1〈/exceptionState〉
        〈edge〉
            〈class〉VisitAssignStmtEdge〈/class〉
            〈toState〉2〈/toState〉
            〈rightOp〉new java.io.FileOutputStream〈/rightOp〉
        〈/edge〉
    〈/state〉
    〈state〉
        〈number〉2〈/number〉
        〈status〉error〈/status〉
        〈stateDescription〉A FileOutputStream created and not released 〈/stateDescription〉
        〈severeLevel〉1〈/severeLevel〉
        〈exceptionState〉2〈/exceptionState〉
        〈edge〉
            〈class〉VisitInvokeStmtEdge〈/class〉
            〈toState〉3〈/toState〉
            〈invokeMethod〉java.io.FileInputStream: void close()〈/invokeMethod〉
        〈/edge〉
    〈/state〉
    〈state〉
        〈number〉3〈/number〉
        〈status〉right〈/status〉
        〈exceptionState〉3〈/exceptionState〉
        〈edge〉
            〈class〉VisitInvokeStmtEdge〈/class〉
            〈toState〉4〈/toState〉
            〈invokeMethod〉java.io.FileInputStream: void write()〈/invokeMethod〉
        〈/edge〉
    〈/state〉
    〈state〉
        〈number〉4〈/number〉
        〈status〉error 〈/status〉
        〈stateDescription〉a released FileOutputStream can't be used again 〈/stateDescription〉
        〈severeLevel〉2〈/severeLevel〉
        〈exceptionState〉4〈/exceptionState〉
    〈/state〉
〈/FSM〉
```

图 10　FileOutputStream 相关“复合式”缺陷对应的有限状态自动机描述文件

3.3 缺陷模式挖掘器

CODA 的缺陷模式库扩展工作可以是人工进行的,也可以是半自动进行的.以人工的方式扩展新缺陷模式,不仅对用户有较高的要求,而且效率较低.为了提高缺陷模式库的扩展效率,CODA 实现了缺陷模式挖掘器,从而支持以半自动化的方式扩展其缺陷模式库.缺陷模式挖掘器包括三部分:规则挖掘器、规则精化器和规则转换器,如图 11 所示.

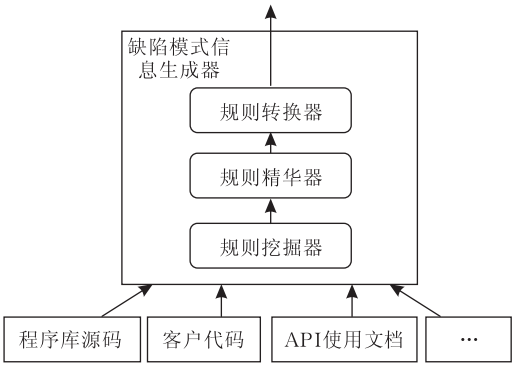

图 11 缺陷模式挖掘器结构图

(1)规则挖掘器

规则挖掘器用于自动生成客户代码调用软件库函数时所应遵循的规则.目前,我们已经实现了 3 个规则挖掘器[10-12].其中,第 1 个规则挖掘器 MATS[10] 从客户代码挖掘调用软件库函数的规则,适用于存在大量客户的情况;第 2 个规则挖掘器 JRF[11] 从软件库的源代码推导出调用软件库函数的规则,适用于能获取软件库代码的情况;第 3 个规则挖掘器 Doc2Spec[12] 从软件库的 API 文档中推导出调用该软件库函数的规则,适用于客户端代码和库函数代码都不能获取到时.以生成"打开的 FileInputStream 实例要在使用完后调用 close()方法关闭"这个规则为例,MATS 需要先对调用 FileInputStream 有关的函数插桩并收集跟踪数据,然后再从跟踪数据里挖掘出这条规则;JRF 需要分析 FileInputStream 的源代码,然后根据这个类提供的各个函数之间的关系推导出这条规则;Doc2Spec 使用自然语言处理相关技术对 API 文档中关于 FileInputStream 的描述进行剖析,从而推导出该类相关的一系列使用规则.

(2)规则精化器

规则精化器用于对规则挖掘器产生的初始规则进行筛选,以挑出可靠的库函数使用规则留待进一步处理.已有的方法[13-14] 基本上是在规则生成之后才进行筛选,例如 Yang 等人[14] 开发的 Perracotta 采用了两个启发式的方法过滤不可靠的规则.这两个过滤方法参考了函数名字的相似性和调用关系.而我们的过滤方法则尽量在跟踪数据生成时期就过滤掉对挖掘规则有负面作用的函数调用.由于滤掉了那些有负面作用的函数调用,我们的方法能有效地提高挖掘规则的准确度.并且,由于我们提出的方法在跟踪数据生成时期就过滤掉了很多有负面作用的函数调用,因而记录的跟踪数据较小,从而大大节省了挖掘规则所需要的时间.

虽然我们提出的过滤方法能有效地帮助提高生成的规则的准确度,但正如我们的实验结果所示[11]:即使引入了我们提出的过滤技术,生成的规则准确度仍然不够高.为了得到准确的缺陷模式知识,CODA 在这个环节还需要一定的人工参与,因此我们称 CODA 为具有"半自动化扩展"能力的静态缺陷分析工具.

(3)规则转换器

规则转换器的作用在于把精化后的规则转换成时序相关的缺陷模式信息,并自动添加至缺陷模式库中.

CODA 所提供的缺陷模式库的半自动化扩展机制的突出特点在于:它可以帮助用户以多种方式批量发现库函数的使用规则,并在用户的指导下自动将有效规则转换成缺陷模式信息并添加到缺陷模式库中.这种半自动化的扩展机制不仅省去了用户通过阅读大量代码搜集整理缺陷模式的繁琐劳动,还可以提高添加缺陷模式信息的工作效率.有关缺陷模式挖掘器的更多信息可以参考文献[10-12].

4 实 验

4.1 实验目的

对于共性缺陷比较确定的领域,例如某个特定的操作系统或开发环境,传统的缺陷分析工具能够基本满足要求.本文提出的方法主要适用于共性缺陷不断扩展的领域.中间件就是一个共性缺陷不断扩展的领域:随着中间件的日益增多,越来越多的企业级应用软件都是基于中间件开发得到的.随之而来的问题是,程序员往往因为对中间件的了解不够深入而错误地使用了其提供的接口,从而导致编写出的程序存在缺陷.因此,从基于中间件开发的应用程序角度来看,一个能够检查程序中是否存在中间件相关缺陷的工具是非常重要的.但就现有的各缺陷分析工具而言,它们对中间件相关的缺陷模式还基本上涉猎不多,原因主要有两个方面:(1)很多缺

陷分析工具只关注那些普遍存在于各软件中的基础性、常见性缺陷，例如空指针引用、声明的变量未被使用等等，以确保工具可以被广泛使用；(2)绝大多数工具并没有提供足够方便的扩展机制以添加新的缺陷模式，当新的中间件不断出现时，这些工具无法及时地扩展这些中间件相关的缺陷模式，因此很难查出这些中间件相关的缺陷.

在本实验中，我们分别使用 CODA、FindBugs、PMD 和 Hammurapi 对 J2EE 和 JBoss 的一些客户代码进行缺陷检测，以确认 CODA 在查找中间件相关的缺陷上是否存在优势.

4.2　实验过程

为了展示半自动化扩展机制对缺陷模式库内容的扩充以及缺陷查找能力的提升所发挥的作用，我们设计了如下实验：

(1)首先利用 CODA 的半自动化扩展机制将 J2EE、JBoss 的时序相关的缺陷模式添加到 CODA 的缺陷模式库中；

(2)选择 J2EE 客户代码(wsmg-1.76.1 和 jonas-3.0.6)和 JBOSS 客户代码(jboss-remoting-2.2.2)作为实验过程的受检程序；

(3)分别运行 CODA、FindBugs(1.3.8)、PMD(4.2.5)和 Hammurapi(5.7.0)以检测 J2EE 客户代码中的缺陷；

(4)分别运行 CODA、FindBugs(1.3.8)、PMD(4.2.5)和 Hammurapi(5.7.0)以检测 JBOSS 客户代码中的缺陷；

(5)对 4 个工具报出的 J2EE、JBOSS 平台相关的缺陷报告进行对比分析.

4.3　实验结果

在应用 4 种工具检测 J2EE 客户代码时，FindBugs、PMD 和 Hammurapi 因为缺少 J2EE 相关的缺陷检测器而都未能报出 J2EE 相关的缺陷. 而 CODA 则得益于半自动化扩展机制报出了一些与 J2EE 相关的缺陷.

CODA 在检测 wsmg-1.76.1 时报出了“javax.jms.MessageConsumer 实例在使用之后没有调用 close()方法”的缺陷. 该缺陷发生于类“wsmg.processors.PublisherThread”中. 经分析发现，该类的私有属性 consumer 属于 MessageConsumer 类型，该类的构造函数通过“session.createConsumer(destination)”语句创建了 consumer 对象，但该类的 run()等其它方法中都没有对该 consumer 执行 close()方法. 这说明 CODA 报出的该缺陷是准确的. 除此之外，wsmg-1.76.1 中还有另外两处对 MessageConsumer 进行了使用，分别位于类“wsmg.jms.TestActiveMQ40”和“wsmg.jms.DeliveryThread”中. 经分析，TestActiveMQ40 类文件的第 154 行通过“MessageConsumer consumer = session.createConsumer(destination)”语句创建了 MessageConsumer 对象，在第 167 行通过“consumer.close()”语句对该对象进行了即时关闭，该处对 MessageConsumer 的使用属于正确使用. DeliveryThread 类的构造方法中依然使用“session.createConsumer(destination)”语句创建了 consumer 对象，但该语句处于一个“永假”分支中. 在该源文件被编译成 class 文件时，编译器会自动将该分支代码作为 deadcode 剔除掉. 而 CODA 分析的是 bytecode 文件，因此并未报出这一缺陷.

CODA 在检测 jonas-3.0.6 时报出了“javax.jms.QueueSender 实例在使用之后没有调用 close()方法”的缺陷. 该缺陷发生于“sampleappli.StockHandlerBean”类文件中. 经分析，该项目的 Java 代码中只有这一处对 QueueSender 进行了使用而且 CODA 报出的该缺陷是准确的.

在应用 4 种工具检测 JBoss 客户代码(jboss-remoting-2.2.2)时，FindBugs、PMD 和 Hammurapi 也都因为缺少 JBOSS 相关的缺陷检测器而都未能报出与 Jboss 相关的缺陷. 而 CODA 却得益于半自动化扩展的方式报出了一些与 JBOSS 相关的缺陷.

CODA 在检测 jboss-remoting-2.2.2 时报出了缺陷“org.jboss.remoting.detection.Detector 实例在使用之后没有调用 stop()”. 该缺陷发生于“org.jboss.remoting.detection.util.DetectorUtil”类中. 经分析发现，该类的 start()方法中通过第 128 行的“MulticastDetector mdet = new MulticastDetector()”和第 130 行的“detector=mdet”等语句新建了一个 Detector 对象，然后在第 145 行执行了“detector.start()”方法，但在该方法结束前并未执行 stop 方法. 这说明，CODA 报出的这一缺陷是准确的. 除此之外，CODA 还报出了缺陷“org.jboss.remoting.transport.ClientInvoker 的实例在使用之后没有调用 disconnect()”，该缺陷发生于“org.jboss.remoting.Client”类中. 经人工核实，发现该类的 disconnect()方法中调用了静态类 InvokerRegistry 的 destroyClientInvoker()方法，而 destroyClientInvoker()方法中调用了 ClientInvoker 的 disconnect()方法. 该类通过跨类调用的方式“隐式地”对 ClientInvoker 实例进行了 disconnect. 由于

CODA 在跨类的方法间分析上不够完善,导致其报出的该缺陷为误报.

通过人工核实,我们发现:CODA 所报出的上述 4 种中间件相关的缺陷的准确率为 75%,漏报率约为 20%.

由于现有的分析技术、规则挖掘技术都不够成熟,我们的工具 CODA 还有很多需要完善的地方.但借助缺陷模式库的半自动化扩展机制,CODA 的缺陷检测能力能够在短时间内得到迅速扩展和提升.通过上述实验结果我们可以确认:与其它工具相比,CODA 在对中间件相关的缺陷检测方面具有较大的优势.

5 结 语

针对大多数现有静态代码缺陷查找工具都没有为用户提供足够易用的缺陷模式扩展机制,本文提出了一个支持半自动化扩展的基于缺陷模式匹配的静态缺陷分析方法,并开发了基于该方法的工具 CODA. CODA 不仅提供了很多缺陷模式描述模板,以方便用户根据自身需要,选择适当模板来描述所感兴趣的缺陷模式,还提供了缺陷模式库的半自动化扩展机制,该机制可以帮助用户半自动化地快速地发现、整理、添加新的缺陷模式.特别地,经实验表明,与同类工具 FindBugs、PMD、Hammurapi 相比,在查找 J2EE 和 JBoss 的客户程序中特定于中间件的方法调用序列相关的缺陷时,CODA 表现出了较明显的优势.

在未来工作中,我们将提供描述能力更强的缺陷描述语法,并尽可能地探索更加方便易用的描述方式以帮助用户更加快捷地添加缺陷模式.此外,我们还将努力优化和改进工具所使用的各种分析技术,因为分析技术的优劣直接决定了工具所产生的缺陷检测结果的可信度和准确度.最后,我们还将继续改进规则挖掘器和规则精化器所使用的算法,从而挖掘出更多更准确的 API 使用规则给用户,以进一步减少用户筛选规则的工作量.

参 考 文 献

[1] Nielson Flemming, Nielson Hanne Riis, Hankin Chris. Principles of program analysis. Springer (Corrected 2nd printing), 2005

[2] Cousot P, Cousot R. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints//Proceedings of the 4th ACM Symposium on Principles of Programming Languages (POPL). 1977: 238-252

[3] Detlefs D L, Nelson G, Saxe J B. A theorem prover for program checking. HP Laboratories Palo Alto, Technical Report, 2003

[4] Clarke E M, Grumberg Jr O, Peled D A. Model Checking. MIT Press, 2000

[5] Boyer Robert S, Elspas Bernard, Levitt Karl N. SELECT—A formal system for testing and debugging programs by symbolic execution//Proceedings of the International Conference on Reliable Software. 1975: 234-245

[6] Hallem S, Chelf B, Xie Y, Engler D. A system and language for building system-specific, static analyses//Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation (PLDI). 2002: 69-82

[7] Hovemeyer D, Pugh W. Finding bugs is easy//Companion to the 19th annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages and Applications (OOPSLA). 2004: 92-106

[8] Reimer D, Schonberg E, Srinivas K, Srinivasan H, Alpern B, Johnson R D, Kershenbaum A, Koved L. SABER: Smart analysis based error reduction//Proceedings of the International Symposium on Software Testing and Analysis (ISSTA). 2004: 243-251

[9] Yang Zhao-Hong, Gong Yun-Zhan, Xiao Qing, Wang Ya-Wen. A defect based model testing system. Journal of Beijing University of Posts and Communications, 2008, 31(5): 1-4 (in Chinese)
(杨朝红, 宫云战, 肖庆, 王雅文. 基于软件缺陷模型的测试系统. 北京邮电大学学报, 2008, 31(5): 1-4)

[10] Zhong Hao, Zhang Lu, Mei Hong. Early filtering of polluting method calls for mining temporal specifications//Proceedings of the Asia-Pacific Software Engineering Conference (APSEC). 2008: 9-16

[11] Zhong Hao, Zhang Lu, Mei Hong. Inferring specifications of object oriented APIs from API source code//Proceedings of the Asia-Pacific Software Engineering Conference (APSEC). 2008: 221-228

[12] Zhong Hao, Zhang Lu, Xie Tao, Mei Hong. Inferring resource Specifications from natural language API documentation//Proceedings of the International Conference on Automated Software Engineering (ASE). 2009: 307-318

[13] Weimer W, Necula G. Mining temporal specifications for error detection//Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS). 2005: 461-476

[14] Yang J, Evans D, Bhardwaj D, Bhat T, Das M. Perracotta: Mining temporal API rules from imperfect traces//Proceedings of the International Conference of Software Engineering (ICSE). 2006: 282-229

[15] David Binkley. Source code analysis: A road map//Proceedings of the 2007 Future of Software Engineering (FOSE). 2007: 104-119

LIANG Guang-Tai, born in 1984, Ph. D. candidate. His current research interests include program analysis, code defect analysis, and warning prioritization.

MENG Na, born in 1983, Ph. D. candidate. Her current research interests include program analysis, and code defect analysis.

LI Jin-Hui, born in 1983, M. S. . His research interests include program analysis, and code defect analysis.

ZHONG Hao, born in 1979, Ph. D. , assistant professor. His research interests include software engineering, and mining software repository.

ZHANG Lu, born in 1973, Ph. D. , professor, Ph. D. supervisor. His research interests include software engineering, software reuse technology, and program comprehension.

WANG Qian-Xiang, born in 1970, Ph. D. , professor, Ph. D. supervisor. His research interests include software monitoring, program analysis, and middleware.

Background

This work was sponsored by the National Basic Research Program (973 Program) of China under Grant No. 2009CB320703 and the Science Fund for Creative Research Groups of China under Grant No. 60821003, and the National Science Foundation of China under grant Nos. 60773160 and 61033006.

Lightweight static analysis tools such as FindBugs, PMD, Jlint, and Lint4j aim at detecting generic bugs by analyzing source code or bytecode against pre-defined bug patterns without executing the program. Compared with formal verification techniques such as model checking and theorem proving, these bug-pattern -pattern-based tools use lightweight analysis techniques, and they are effective in detecting generic bugs in large software. The advantages of this kind of approach lie in the simplicity of its usage and the efficiency of its analysis, which make it a new technique with more rapid development among different approaches of static code defect analysis recently. However, among the available tools based on the approach, the extension modes provided by most of them are neither friendly nor efficient enough for users to extend their defect analysis capability.

In this paper, we proposed a semi-automatic extensible static code defect analysis approach. Based on it, a tool named CODA (COde Defect Analysis tool) has been designed and implemented. In order to support the efficient extension of the defect pattern library, CODA provides not only enough "defect pattern description templates" to facilitate users' manual extension, but also a "semi-automatic extension mechanism" which accelerates the process of discovering, summarizing and extending new defect patterns. Once a new defect pattern is defined and added into the defect pattern library, CODA can automatically own the detecting ability for related defects. Our evaluation has showed that our tool was effective in detecting the middleware-related defect warnings for programs under analysis once the related defect patterns are discovered and formally specified.