

单元测试自动生成平台的分析与研究*

徐国庆¹⁺, 杨宗源¹, 黄海涛¹

¹(华东师范大学 计算机科学技术系 软件工程实验室, 上海 200062)

Analysis and Research on the Automated Generation of Unit Test

XU Guo-qing¹⁺, YANG Zong-yuan¹, HUANG Hai-tao¹

¹(Software Engineering Lab, Department of Computer Science, East China Normal University, Shanghai 200062, China)

+ Corresponding author: +86-21-6223-3654, Fax +86-21-6286-1049, E-mail: gqxu_02@cs.ecnu.edu.cn

<http://www.cs.ecnu.edu.cn/sel/~harryxu>

Abstract: Unit testing is a methodology for testing small parts of an application independently of whatever application uses them. It is time consuming and tedious to write unit tests and codes to generate test cases. In addition, it is especially difficult to write unit tests that model the pattern of usage of the application they will be used in. In this paper, we describe two unit test automated generation framework made by Software Engineering Lab of East China Normal University. The first one is called JMLAutoTest, which is built based on the formal predicates. It can generate unit test codes and test cases automatically from the JML specifications of programs. The second one is called JAOUT, which is an Aspect-oriented testing framework. In this framework testers build application-specific Testing Aspects with an Aspect-Oriented Test Description Language (AOTDL) which can be translated into generic aspects of AspectJ. JAOUT then generate unit test oracles from these testing aspects automatically.

Key words: Aspect-Oriented Programming, unit test, test oracle, test case generation

摘要: 单元测试是独立测试系统模块的一种方法。随着软件规模及其复杂度不断扩大,单元测试也越来越受到人们的关注。然而,编写单元测试代码,生成测试用例是一件非常耗时的工作;编写出能够反映应用程序在某一特定方面用途的测试模型更加困难。本文中描述了华东师范大学软件工程实验室所开发的两个自动测试生成平台:JMLAutoTest 和 JAOUT。JMLAutoTest 是基于形势规范的测试平台,它可以从程序中所指定的 JML 规范自动生成代码和测试用例。而 JAOUT 是面向方面的测试平台。在此平台中,测试者可以使用面向方面的程序描述语言(AOTDL)来建立与应用相关的高层 Testing Aspects,这些 aspects 又可以被 JAOUT 工具转化为 AspectJ 中的低级一般的 aspects。JAOUT 根据这些 Testing Aspects 来自动产生单元测试的代码和测试标准。

关键词: 面向方面的程序设计,单元测试,测试标准,测试用例生成

中图法分类号: TP311 文献标识码: A

1 单元测试概述与相关背景

随着软件规模及其复杂度的不断扩大,单元测试日益受到程序开发人员的重视。这一点可以清楚地在此

第一作者简介: 徐国庆(1981 -),男,安徽省滁州市人,硕士研究生,主要研究领域为软件可靠性与软件复用,其中包括了程序设计语言,形式化方法等方面。

限编程 (eXtreme Programming) [1] 所提倡的测试策略中反映出来。在 XP 中, 单元测试被看作是程序设计的一个重要的组成部分: 测试在编程前, 进行中, 结束后要不断地进行, 就像 Beck 和 Gammer 在文献[2]中所提到的那样, “code a little, test a little, code a little and test a little”。然而, 在实际的单元测试过程中, 程序设计人员们往往会体会到编写单元测试代码是一件复杂且耗费时间的工作: 首先得编写一个单元测试框架, 能够调用已经写好的模块, 对其进行测试; 其次, 要编写大量的代码来生成测试用例。若这些测试用例中包括的带有复杂递归结构, 则产生它们更加困难; 最后, 还得便写测试通过与否的标准, 即符合什么样标准所要测试的模块才算正确? 所有这些测试代码的人工完成阻碍了单元测试的必要实施, 降低了单元测试的效率, 而且人工完成的测试代码可靠性并没有保证。

这种情况下, 研制自动产生单元测试代码的工具就变成了一项迫切且存在着相当大实际意义的任务。那么, 这样的工具可能会在哪些方面代替人来进行自动化工作? 首先, 自动化应该体现在单元测试框架 (Test Framework) 的自动生成。这个框架可以根据所需测试的模块 (在面向对象的开发中更多以类的形式来表现), 自动被生成。框架本身必须要拥有调用被测模块中方法的接口, 装载测试用例的接口, 以及判断测试是否成功的子模块; 其次, 自动化应该体现在测试标准 (Test Oracle) 的自动生成上, 即必须存在一个严格的标准来判断什么样的测试是成功的, 什么是失败的, 而不仅仅是传统意义上程序调试中程序员按照自己的想法来编写代码判断测试结构如何; 第三, 自动化应该体现在测试用例 (Test case) 的生成上。人工编写产生大量测试输入的程序代码相当复杂, 尤其是若需产生具有复杂数据结构的测试用例 (树, 图) 则更加困难。这就需要有一个能够在测试人员指定的范围内自动生成测试输入对象的工具。

对于测试平台自动生成的研究开始于 70 年代的中期。在将近三十年对自动测试的研究和应用中, 出现了大量的方法及相应的工具。最具有代表性的为基于形式化方法的测试[3], 而这当中目前应用最广泛的是基于形式规范的自动测试[4]。形式规范是在设计时刻指定的用来验证程序正确性的一系列模型, 由于其代表了最原始并未修改过的设计思路, 与具体实现代码无关, 因此形式规范完全可以转化为测试标准 (Test Oracle) 来判断程序运行的正确性。即如果实现时的程序运行结果不符合设计时指定的形式规范的要求, 则当前程序是失败的。研究人员对如何从形式规范转化为测试标准做了大量的工作, 最著名应该是 Cheon 和 Leavens 在文献[4]中提到的将 JML 规范自动转换为 JUnit[5]框架的测试代码。JML[6]是 Iowa State University 为 Java 语言开发的行为接口规范语言 (BISL), 它可以被以注释的方式与 Java 程序混合。包含 JML 注释的 Java 程序可以被运行时断言检查器 (Runtime Assertion Checker) 编译, 如果运行时刻程序状态与 JML 注释的规范不符合, 则检查器动态抛出异常来提示谓词违反错误。JUnit 是一种为 Java 设计的单元测试框架, 此框架中包含了测试代测类中方法的方法, 建立测试用例的方法等。即在 JUnit 框架中, 测试人员可以很方便指定要测试的方法以及写代码来生成测试用例, 而运行这些需要测试的方法可以交给 JUnit 自动完成。然而在 JUnit 中, 测试标准和用例还得手工生成。文献[4]中阐述了将形式断言作为评判测试通过与否的标准, 并自动生成 JUnit 的测试代码。一旦有前置断言违反的异常被抛出, 则测试程序接收到此异常, 表明当前测试用例为无意义用例, 当前测试被忽略; 而后置断言或不变式断言违反异常被抛出, 则表明当前测试失败。

然而, 文献[4]并没有论述如何自动生成测试用例, 生成测试用例的任务还是被留给测试人员。本文将介绍的第一个工具 JMLAutoTest 框架[7]建立在文献[4]描述的 JMLUnit 框架的基础上, 可以以形势断言为依据自动生成测试用例。如果测试用例中存在着大量无意义用例 (违反方法前置谓词的用例), 则测试结果无说服力且测试效率被大大降低。JMLAutoTest 使用两阶段测试法来运行测试, 从而过滤了大部分的无意义用例。

但是, 随着软件规模加大以及人们对软件各方面行为正确性要求不断提高, 传统基于形式断言的测试中出现了新的问题。形式规范通常主要描述了程序的功能行为, 但对于程序性能以及时序都非功能性特征无法进行很好的建模。最近出现的面向方向 (Aspect-Oriented) 程序设计[9]着眼于程序中横切 (crosscutting) 的属性, 因此可以很好应付那些传统形式断言所无法解决的问题。本文描述的工具 JAOUT [9] 是面向方向的自动单元测试生成工具。它将在用面向方面测试描述语言 (AOTDL) 描述的 Testing Aspect [10] 中所规定的建议 (advice) 作为测试标准, 并自动按照某 Java 类生成测试此类方法的 JUnit 框架; 作者又将其与 JMLAutoTest 中测试用例生成模块集成起来, 从而解决了此平台中测试用例自动生成的问题。

2 JMLAutoTest-基于形式断言的自动测试平台

JMLAutoTest[7]是作者为文献[4]中描述的 JML+JUnit 平台所设计与开发的测试用例自动生成与无意义测试用例过滤工具。JMLAutoTest 作为 Iowa State University 的 Gary T. Leavens 教授所领导的 JML 项目的一部分,现在已经被集成进最新版本的 JML 工具集。JMLAutoTest 的基本设计目的在于弥补 JML+JUnit 测试平台无法生成测试用例的不足,尤其是一些带有复杂数据连接结构的测试用例(树,图等)。本章接下来的部分将对 JMLAutoTest 的基本工作原理进行描述。

2.1 测试用例的自动生成

```

public class LinkedList{
    public Node first; // 链表中的第一个点;
    protected int length; // 链表长度
    ...
}
public class Node{
    public int ID; // 节点 ID
    //指向下一个域的指针
    public Node pointer;
}
-----
public JMLObjectSequence
    makeCase_LinkedList(){
        Finitization f =
            new Finitization
                (LinkedList.class);
        /** 创建三个不同的 Node 对象, [0, 1, 2]
            为每个对象构造方法参数 */
        JMLObjectSequence nodes =
            f.createObjects(Node.class,
                new JMLPrimSequence
                    (new int[]{0,1,2}), 3);
        // 在域中加入 Null.
        f.add (nodes, null);
        //为"first"指定值域
        f.set ("first", nodes);
        f.set //为属性 "length" 指定值域
            ("length", new JMLPrimSequence
                (1,4));
        //递归产生对象集合
        f.generate("first",
            new String[]{"pointer"});
        return f.getResult();
    }

```

Fig.1. Generate the test case space for class LinkedList

图 1. 为表递归的数据结构(链表)的类 LinkedList 创建对象集合

测试用例自动生成的思想包含三个主要的过程:穷举 (Finitization), 对象合法化验证 (Object Validity Checking) 以及消除同构 (Nonisomorphism)。图 1 给出了带有递归的数据结构 LinkedList 类定义及使用 JMLAutoTest 提供接口来为创建类型为 "LinkedList" 的对象实例的全过程。JMLObjectSequence 与 JMLPrimSequence 是我们建立的工具类,作为容纳对象和基本类型(在 Java 中)的队列。我们使用类 Finitization 来建立对类属性的穷举,在符合限制的对象集合生成之后, JMLAutoTest 使用运行时断言检查器 (Runtime Assertion Checker) 来检验其是否合法并对对无意义测试用例进行过滤。稍后的章节将对图 1 做出详细的解释。

为类属性成员设置值域 JMLAutoTest 中的一个核心类就是 Finitization 类,它负责在测试人员指定的范围内穷举所有可能情况,来生成测试用例空间。因此在测试用例自动生成过程中的第一个工作就是让测试人员来指定用例生成的范围。这在 JMLAutoTest 中是通过指定对象属性成员值域来实现的。在图 1 中,因为 LinkedList 类中有两个成员, length 与 first, 因此在图中右部的测试用例生成程序中,先使用 createObject 方法创建三个 Node 对象,把空值 null 加入 Node 对象后,使用 set 方法为 LinkedList 的成员属性 first 指定值域,即将生成的这个 Node 集合。同样的方法为 length 属性指定值域为包含从整数 1 到 4 的 PrimSequence 对象。这样 JMLAutoTest 就会自动将值域中的每个值赋给对象中成员,以生成可能的测试用例。

产生测试用例空间 类 Finitization 中实现了两种类型的对象产生方法。第一种为直接调用 generate 方法,无任何参数,表示生成通常的无递归结构的对象集合。为了产生本身包含复杂递归结构的类对象集合, Finitization 类中还包含了另外一种 generate 方法。图 3 通过为一线性链表类创建对象集合来显示了这种方法的应用。可以看到这里的 generate 方法定义了两个参数,第一个参数是表示递归根的属性名称,第二个参数是一个 String 类型的数组,包含了每个节点中指针域的名称。若要创建一棵二叉树,则应该写成 f.generate("root",new String[]{"left","right"})了。这种递归用例生成的具体算法与效率分析,作者在文献[7]中做了详细介绍。

测试用例合法性验证 由于类中属性之间的关联性,因此在设置值域并穷举的时候往往会违反了这些关联要求,如上例中的链表产生了三个结点而属性 length 却为 4。JMLAutoTest 采用检验不变式(Invariant)的运行时值的方法来检验其合法性。不变式是类中所有方法都必须遵守的规范。在 JML 中,不变式即是前置状态规范也是后置状态规范。在任何方法的执行前(参数被传进但函数体还未执行)以及执行后(函数体执行后,但还未返回)都会被检测。JMLAutoTest 在一个测试用例对象生成之后,调用 JML 运行时检查器所提供的 checkInvariant 方法,检查类中 JML 不变式是否被违反,如果接收到异常,说明不变式被违反,则丢弃当前用例对象。这种通过不变式进行检验的方法不再需要开发人员在程序中提供特殊的方法来验证对象合法性。

2.2 消除无意义测试用例

在测试用例对象集产生后,用例对象并不都是有意义的(满足被测方法的前置谓词条件)。因此若产生的测试用例集合中包含太多的无意义用例,其最终的测试结果是不能说明问题的。更重要的一点,大量的时间将会被用在处理无意义的测试用例上。JMLAutoTest 采用两阶段测试法(double-phase testing)来过滤无意义用例。即将整个测试阶段分为两个不同部分:第一部分为预测试阶段,在此阶段中,测试人员建立标准(Operational Profile),将生成的测试空间划分为若干区间,从此区间中按照某种统计规则取出一小部分测试用例,并依次使用每个区间中取出的测试用例来运行程序。在所有测试用例运行结束后,观察各区间中产生的无意义测试用例数,并计算出无意义测试用例比例。在第二阶段中,测试用例被按照每个区间无意义用例的反比进行收集,并进行最终的测试。由于第一阶段的过滤,大部分的无意义测试用例都已经被丢弃,因此保证了最终的测试结果的说服力以及测试执行的效率。作者在文献[7]中详细阐述了其算法说明。

图 2 介绍了 JMLAutoTest 测试平台的建立:输入一个 Java 类 M, JMLAutoTest 将会自动生成三个测试类 JML_M_Test, JML_M_TestCase, 和 JML_M_TestClient。JML_M_Test 是 JUnit 的测试类,而 JML_M_Test 为测试用例提供者(Test case provider), JML_M_Client 为测试客户端,测试人员在其中使用以上介绍的方法自动生成指定范围,是用例自动生成。Test Case Provider 首先通过调用 test client 的 makeCase 方法,得到生成的测试用例空间。然后通过调用 makeOperationalProfile 方法将测试用例空间划分区间。TestRunner 通过 runPreTest 进行第一阶段测试;测试结束后使用 calculate 计算无意义测试用例比例,再通过 runFinalTest 来进行最终的测试。作者在文献[7]中也通过具体的用例分析来证明了这种两阶段测试法的正确性与有效性。



Fig.2. The working sequence of our testing framework
图 2. 自动测试平台的工作流程

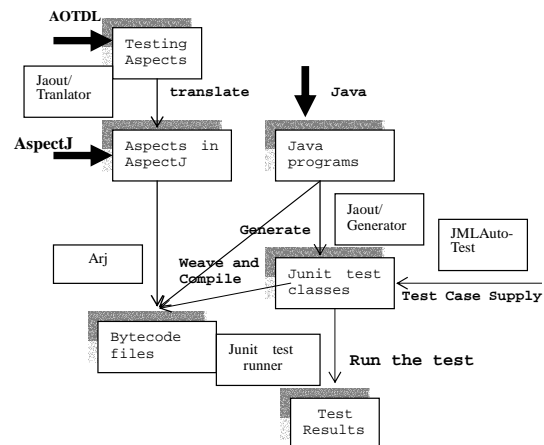


Fig.3. An Overview of the basic technique
图 3. JAOUT 工具总观

3 JAOUT-面向方面的单元测试自动生成

随着软件应用与测试范围的不断扩大,使得对于程序某些特定方面进行建模变得非常困难。这些方面尤其体现在程序中一些非功能性的特征上,如性能要求,程序中方法执行的时序逻辑等。对于这些特征的测试在传统基于形式规范的测试平台上很难完成,因为大部分的形式规范主要为保证程序的功能性表现的正确性

而设计, 如 JML, Larch 等, 而对于非功能性的特征就表现的力不从心了。由于面向方面 (Aspect-Oriented) 程序设计方法[9]关注于程序中横切的 (crosscutting) 属性, 使得程序开发方法从传统垂直方向的设计模式扩展到垂直与水平方向相结合的新型模式来, 对于程序中非功能性特征的验证就不再是一件很困难的事情。例如对程序的性能或是方法执行的时序逻辑验证在 AOP 中都得到了很好的解决。作者发觉, AOP 的这种分离对不同方面关注的思想也非常适合于考虑单元测试问题。即单元测试结果本身不仅仅是应该按照简单的功能的正确或错误来评判, 而应该更多地反映出程序各方面的问题来, 如性能, 时序逻辑等等。因此, 将 AOP 中的 aspect 中所观察的限制作为单元测试的测试标准 (test oracle) 就成为非常自然的一个想法了。

虽然有了这个想法, 但是目前的 AOP 中的 aspect 是一个低等, 语言级的模块概念: 在其中指定横切点 (pointcuts), 并且对横切点到达时, 可以提出建议 (advice); 这样的 aspect 无法被直接作为测试标准, 因为单元测试的程序无法识别它们。为了解决这个问题, 作者提出一个概念叫作应用相关的 Aspect (application-specific Aspect) [8], 即可以在这些语言级的 aspect 上抽取高级的, 更加抽象的与应用相关的 Aspect (这里特意首字母大写来区分两种不同的 aspect)。即每一个 aspect 肯定完成了一个特定的功能, 例如有的是为了跟踪程序 (tracing), 有的是为了起到前置和后置谓词的作用 (assertion), 或者有的是为了记录日志。这样每一种完成同样功能的 aspect 被抽取出来, 建立抽象的高级的应用相关的 Aspect, 这样的 Aspect 是共享许多相同的特性的。在这种思想的指导下, 我们可以对所有 AOP 中语言级的为了完成测试功能的 aspect, 建立 Testing Aspect。这种 Testing Aspect 的共同特征就是在运行时刻向测试程序发送动态的测试评判消息。这样, Testing Aspect 就可以被作为测试标准 (test oracle), 来判断最后测试在哪一方面存在问题。

在这个思路的指引下, 作者建立了第一个面向方面的单元测试自动生成平台, JAOUT [8]。图 3 描述了 JAOUT 中的基本技术框架。作者设计了 AOTDL (Aspect-Oriented Test Description Language) 语言来建立 Testing Aspect。然后, 工具 JAOUT/Tranlator 可以将 AOTDL 建立的 Testing Aspect 翻译为 AspectJ 中的语言级的 aspect。在 AspectJ 的编译器 arj 将代码混合, 并编译后, 程序的字节码 (.class) 文件被生成。JAOUT/Generator 根据程序的 Java 类文件自动生成 JUnit 的单元测试框架。由于作者将 JMLAutoTest 集成在 JAOUT 中, 测试用例也可以完全自动生成。

建立 Testing Aspect 开发人员可以使用 AOTDL 语言来建立 Testing Aspect。图 4 所示为一个名为 TempLogic 的 Testing Aspect。建立这个 Aspect 的目的在于测试 Stack 类对象是否没有被初始化 (init 方法被调用过) 就开始调用其他方法。pushReached 和 initReached 为两个横截点, 对应的条件分别是 push 方法和 init 方法被执行到。如果 init 被执行到, 就将 initialized 属性置为 true。因此如果在 push 调用到之前, isInitialized 为 false, 说明栈没有被初始化过。这里具体体现在 before 建议中。非常清晰的是 AOTDL 将 advice 和 pointcuts 的定义分在三个单元中, Utility, Meaningless Advice 和 Error Advice。由于这个 Aspect 是专门为测试建立起来的, 因此所有表示测试用例为无意义的 advice 被放置在 Meaningless Advice 单元中, 所有表示测试错误的 advice 被放置为 Error Advice 中, 而其他不直接影响到测试标准的 advice 和 pointcuts 的定义放在 Utility 中。可见此 Testing Aspect 通过对于横切的属性的规约, 显式地定义了与测试标准相关的标准。这里使用

advicetype(arguments): pointcuts: conditions: message

的语法来表示无意义测试用例或测试错误的情况。conditions 为触发条件 (布尔表达式), message 为条件触发时打印出的错误信息。图 4 中可见, 当 `s.getSize()>Max` 时, 当前测试用例是无意义的, 而当 `isInitialized` 为真, 即 `init` 方法没有调用过时, 当前测试失败。JAOUT/Translator 将图 4 所示 Testing Aspect 翻译为图 5 中的 AspectJ 中所支持的语言级 aspect。可见, Meaningless Advice 中所指定的 advice, 在条件触发时, 将抛出 `MeaninglessTestCaseException` 的异常, 而 Error Advice 中指定的 advice 条件触发时抛出的异常为 `TestErrorException`。单元测试程序只要对这两个异常进行捕捉, 就可以按照所有 Testing Aspect 中所指定的标准得出程序是否存在问题的结论。对于 JUnit 测试平台的建立与 JMLAutoTest 中相同, 只是把所捕捉的异常类型换一下即可。

```

TestingAspect TempLogic{
// all pointcuts and other utility advices are
declared
// in the Utility unit
Utility{
protected boolean isInitialized = false;
//push is reached
pointcut pushReached(Stack st):
target(st)&&call(void
Stack.push(Integer));
//init is reached
pointcut initReached(Stack st):
target(st)&&call(void Stack.init(void));
after(Stack st):initReached(st){
isInitialized = true;
}
}
MeaninglessCase Advice{
//advices for specifying criteria of
//meaningless test cases
before(Stack s):
pushReached(s):
s.getSize()>=MAX:"Overflow";
...
}
Error Advice{
//advices for specifying criteria of test
//errors
before(Stack s):
pushReached(s):
!isInitialized:"Not Initialized";
...
}
}
}

```

Fig.4. The AOTDL representation of the TempLogic Aspect
图 4. TempLogic Aspect 的 AOTDL 语言描述

```

public Aspect TempLogic{
// Definitions in Utility unit are not changed
protected boolean isInitialized = false;
//push is reached
pointcut pushReached(Stack st):
target(st)&&call(void Stack.push(Integer));
//init is reached
pointcut initReached(Stack st):
target(st)&&call(void Stack.init(void));
after(Stack st):initReached(st){
isInitialized = true;
}
//meaningless advices
before(Stack s)
throws MeaninglessTestInputException:
pushReached(s){
if(s.getSize()>=MAX){
MeaninglessTestInputException ex= new
MeaninglessTestInputException("overflow");
ex.setSource("TempLogic");
}
}
//error advices
before(Stack s)throws TestErrorException:
pushReached(s){
if(!isInitialized){
TestErrorException ex =new
TestErrorException("Not Initialized");
ex.setSource("TempLogic");
}
}
}
}

```

Fig.5. The Translation of TempLogic Testing Aspect
图 5. TempLogic Aspect 被翻译为语言级的 aspect

4 总结与结束语

本文对单元测试平台的自动化的发展及其最新研究进展进行了介绍,并主要描述了两种不同的自动化测试生成平台。JMLAutoTest 基于形式规范进行测试用例生成,使用两阶段测试法过滤无意义用例并将形式谓词作为测试标准自动生成了JUnit的单元测试框架。JAOUT为面向方面的单元测试平台,它允许测试人员使用AOTDL语言建立Testing Aspect,并在其中显式指定测试的标准。JAOUT工具可以将Testing Aspect翻译为AspectJ所识别的语言级aspect,并在运行时刻抛出测试框架可以识别的异常。这两个测试平台还存在一些问题,如AOTDL语言所支持布尔表达式的类型比较局限等。这些问题都是作者在今后研究工作所要解决的。

References:

- [1] Beck K., Extreme Programming Explained. Addison-Wesley, 2000.
- [2] Beck K. and Gamma E., Test infected: Programmers love writing tests. *Java Report*, 3(7), July 1998.
- [3] Goodenough J. and Gerhart S., Toward a theory of test data selection. *IEEE Transactions on Software Engineering*, June 1975.
- [4] Cheon Y. and Leavens G. T., A simple and practical approach to unit testing: The JML and JUnit way, *In Proc of 16th European Conference Object-Oriented Programming (ECOOP02)*, pp. 231-255., 2002.
- [5] JUnit Website, <http://www.junit.org>, May 2004.
- [6] G.T.Leavens, A.L.Baker, and C.Ruby. Preliminary design of JML: A behavioral interface specification language for Java. Technical Report TR 98-06i, Department of Computer Science, Iowa State University, June 1998. (last revision: Aug 2001).
- [7] Guoqing Xu. and Zongyuan Yang., JMLAutoTest: A Novel Automated Testing Framework Based on JML and JUnit., *In Proc. 3rd IEEE ASE Workshop on Formal Approaches to Testing of Software(FATES'03)*, pp. 118-127, Montreal, Canada, Oct.2003., also in *LNCS vol.2931*, Springer-Verlag, Jan. 2004.
- [8] Guoqing Xu, Zongyuan Yang, Haitao Huang, Ling Chen and Fengbin Xu, JAOUT: Automated Generation of Aspect-Oriented Unit Test., Submitted to *IEEE Asia-Pacific Software Engineering Conference*, May 2004.
- [9] Kiczales G., Lamping J., Mendhekar A., Maeda C., Lopes C., Loingtier J.-M., and Irwin J., Aspect-oriented programming. *In Proc. ECOOP'97, LNCS vol. 1241*, Springer-Verlag, 1997.