文章编号:1671-4598(2022)02-0084-11 DOI:10.16526/j.cnki.11—4762/tp.2022.02.013 中图分类号:TP3 文献标识码:A

单元自动化测试中类的抽象内存模型研究

杜婉莹，王雅文
（北京邮电大学 网络与交换技术国家重点实验室，北京 100876）

摘要：由于面向对象程序具有多态性等复杂特性，在软件单元测试中仅凭静态分析难以判断指针和引用指向对象的具体类型，为了解决这一问题，对类的抽象内存模型进行研究，并提出类的操作语义模拟算法；在路径分析时，通过构建和更新抽象内存模型，从而对变量所属类的范围进行限定；对于单元测试，对基于输入域的随机测试进行优化，提出基于路径的随机测试方法，得到输入变量的类型集合；实验表明，类的抽象内存模型结合操作语义模拟算法能够有效提取出路径中类相关的约束，基于路径的随机测试方法比起基于输入域的随机测试方法能够明显提高测试效率。

关键词：面向对象；单元测试；抽象内存模型；符号表；静态分析；测试用例

Research on Abstract Memory Model of Classes in Automated Unit Testing

DU Wanying, WANG Yawen
(State Key Laboratory of Networking and Switching Technology, Beijing University of Posts and Telecommunications, Beijing 100876, China)

Abstract: Due to the polymorphism and other complex characteristics of object-oriented programs, it is difficult to judge the specific types of objects pointed by pointers and references only by static analysis in software unit testing. In order to solve this problem, the abstract memory model of classes is studied, and an operation semantic simulation algorithm of classes is proposed. During path analysis, the class scope to which the variable belongs is limited by constructing and updating the abstract memory model. For unit testing, the random testing based on the input domain is optimized, and a path based random testing method is proposed to obtain the type set of input variables. Experiments show that the abstract memory model of classes combined with the operation semantic simulation algorithm can effectively extract the constraints related to classes in the path, and the path based on random test method can significantly improve the efficiency of testing compared with the input domain based on random test method.

Keywords: object-oriented; unit testing; abstract memory model; symbol table; static analysis; test case

0 引言

如今，信息产业的蓬勃发展离不开信息技术的发展，涉及到了通信设备、计算机、软件等一系列领域。随着市场对软件质量要求和软件自身复杂度的不断提高，软件测试的重要性和软件测试在软件工程中所占的比例越来越大[1]。据统计，随着对软件可靠性要求的提高，软件测试会占用40%乃至60%的总开发时间[2]，其中，单元测试作为软件测试的重要环节，其目的是检测各个单元模块的故障缺陷，会占用60%左右的测试工作所花费的时间。与手工测试相比，自动化测试能降低系统测试和维护等阶段的成本、有效提高测试的效率和质量，同时能够显著提高测试覆盖率、大大扩展测试深度[3]，自动化测试工具的研发正逐渐受到重视。

目前已有的自动化测试工具有Logiscope、PRQA、

收稿日期:2021-11-23； **修回日期**:2021-12-21。
基金项目:国家自然科学基金项目(U1736110)。
作者简介:杜婉莹(1997-)，女，湖北鄂州人，硕士，主要从事软件自动化测试和程序静态分析相关领域方向的研究。
通讯作者:王雅文(1983-)，女，陕西凤翔人，博士，副教授，博士生导师，主要从事软件自动化测试和程序静态分析相关领域方向的研究。
引用格式:杜婉莹，王雅文.单元自动化测试中类的抽象内存模型研究[J].计算机测量与控制，2022，30(2)：84-94.

Macabe、DevPartner、Purify 等[4]，这些测试工具要么分析代码的语法漏洞、要么统计程序执行时的数据，又或者是对功能的可行性和效率进行检测等。在这之中，单元测试工具有 CppUnit、C＋＋ Test、VectorCAST、Visual Unit 等，它们能对程序进行静态分析、也能生成测试框架代码，然而大部分都不支持测试用例自动生成功能，需要依赖人工来完成测试用例生成操作，因此，对测试用例生成方法的研究具有重要的理论和实践价值[5]。

面向对象程序具有封装、继承、多态三大特性。其中，继承使子类可以直接拥有父类定义的属性和操作，减少代码冗余，增强代码的复用性和可扩展性；多态能使同一操作在不同子类中有不同的具体实现，让对象以适合自己的方式响应事件[6]。当子类重写父类函数，并让父类的指针或引用指向子类对象时触发。多态的存在令程序的编写仅需指明要执行的操作，在实际执行时编译器会根据对象所属的具体类型来调用相应的方法，从而表现出不同的行为，灵活性更高[7]。对于包含存在于继承体系中的类对象的源程序，仅凭静态分析，编译器无法判断程序中类型转换语句的结果以及被调用函数所属的类[8]。

在测试用例生成时，如果需要对类对象进行实例化，对象类型的选取是用例生成效率和有效性的一个影响因素。静态生成测试用例开销更小，也具有一定的挑战性，而面向路径的测试用例生成在白盒测试中非常常见。其中，符号执行使用符号来表示程序的输入数据，模拟执行被分析程序，用符号表达式操作代替程序中对变量和参数的操作，是路径分析的一种常用手段。在这一方面，国内外都有不少研究人员参与了研发工作，例如，Euclide 定义了内存动态管理的操作语义模型[9]，许中兴等人提出了虚拟数组建模内存[10-11]，赵云山等人设计实现了针对数值型变量的符号执行系统[12]。在单元测试中，当被测函数的输入变量中含有父类引用时，如何选择类进行实例化，当被调用函数是某个基类的成员函数时，应该选择哪个类中的该函数进行摘要提取，通过静态分析来解决这些问题是本课题研究的重点。

本文第 1 节对类的抽象内存表示模型进行概述；第 2 节介绍类的操作语义模拟算法；第 3 节介绍基于抽象内存模型的单元测试用例生成方法；第 4 节通过一个实例演示路径分析中抽象内存的变化过程；第 5 节对提出的模型和算法进行实验并分析实验结果；第 6 节总结全文。

1　类的抽象内存表示模型

为了明晰函数中类对象的可取类型，以便确认单元测试的输入变量类型集合以及在函数调用点的函数摘要提取操作[13]，可以在路径生成后，通过构建类的抽象内存模型并配合符号执行技术提取出路径上与类对象有关的约束，缩小类对象实际类型的可选范围，在精简测试用例集合的同时保证覆盖率，提高单元测试的效率。

抽象内存模型是存储变量语义和约束的静态存储介质，用于记录变量在符号执行中的动态变化[14]。对程序执行自动测试，需要先通过静态分析得到测试所需的代码信息，为此需要构建符号表，它也是类的抽象内存模型的基础。

1.1　类的符号表

在静态建模中，抽象语法树是最重要的中间结构，它是源程序的一种抽象表现，也是提取程序信息的入手点。源程序的每行代码以及每个关键字都有对应的抽象语法树节点[15]。

对面向对象程序执行单元测试需要先通过静态分析得到被测函数模块的输入变量。这就需要遍历程序的抽象语法树，正确并完整地识别出程序中各个类、函数和变量的信息以及它们之间的关联关系，将其记录于符号表中，在随后生成测试用例时便能快速获取所需信息。本课题的研究重点是类，由类在组成结构方面的特点可以归纳出其基本信息，由此可得类对应符号表的结构如表 1 所示。

表 1　类对应符号表的属性

属性	说明
className	类名
parents	类的父类情况
children	类的子类情况
memVars	类的成员变量
memMethods	类的成员函数
isAbstract	是否为抽象类
scope	类对应作用域

面向对象程序中不同的类可能存在于不同的继承体系中，同一继承体系中类的属性也存在差异。如果该类存在于继承体系中，就将它的父类和子类对应的

符号表项和继承类型存储起来[16]，由于面向对象程序的继承特性，子类无需声明便可拥有父类的成员，在记录了类的继承情况后，就可以通过符号表快速获取从父类继承下来的属性和特征。C＋＋支持多继承；Java 中的类虽然只能单继承，但是可以实现多个接口，因此 parents 是一个列表[17]。

在类的成员变量中，静态变量需要单独标注，因为它们为该类的所有实例所共享，在抽象内存模型的构建和测试用例生成中都需要单独处理。不论是通过哪个实例对静态变量进行访问，实际效果和通过类名调用是相同的，影响的都是同一个成员[18]。

在类的成员函数中，构造函数、静态函数、没有函数体的函数（例如 Java 的抽象方法和 C＋＋的纯虚函数）、有函数体但可以重写的函数（例如 Java 中除构造方法、静态方法、final 和 private 修饰的方法以外的其他方法以及 C＋＋的虚函数）以及其他函数需要分开记录，可以通过这些列表的存储情况来判断类是否为抽象类。

isAbstract 用于说明该类对象能否直接实例化。因为抽象类和接口不能直接实例化，需要通过实例化子类并向上转型的方式来间接完成，所以这里需要被区分。本课题将接口与抽象类同等处理。在 C＋＋中，抽象类的子类只要没有重写父类的全部纯虚函数，就仍为抽象类；在 Java 中，抽象类通过 abstract 显示声明。

1.2 抽象内存模型分类

在面向对象程序中，针对不同的数据类型，可以将抽象内存模型分为基本抽象内存模型、数组抽象内存模型、指针抽象内存模型和类的抽象内存模型，其中类的抽象内存模型也适用于记录结构体变量的内存情况。不同类别的抽象内存模型存在一些公共属性，如符号值、抽象内存单元地址、符号表项等。除了公共属性以外，不同的数据类型还有不同的语言特性，比如数组类型需要记录数组长度、指针类型指向的是内存单元地址、类的成员在内存中离散存放[19]。每个变量都对应一个抽象内存模型，类对象也是如此，但是同一个类的多个对象之间共享一部分属性，即静态变量，任意对象都可以对这部分属性进行访问和修改。

本课题将类的抽象内存模型中与类结构相关的属性提取出来，这部分属性与类本身相对应，构成类类型的抽象内存模型；剩下的属性则与具体对象有关，构成类对象的抽象内存模型。类类型的抽象内存模型与类对象的抽象内存模型之间的关系如图 1 所示。

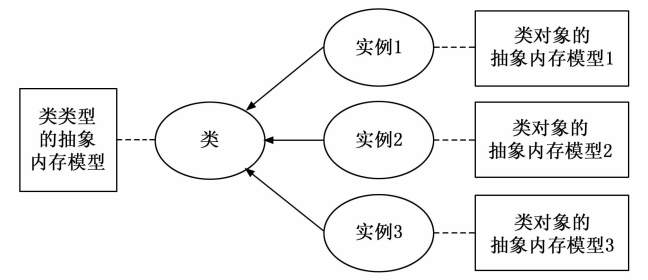

图 1 类的两种抽象内存模型之间的关系

1.3 类类型的抽象内存模型

以 Java 为例，在使用某个类时，首先要将该类加载到内存中，通过类加载器创建相应的 Class 对象。类的静态变量在内存中仅有一份，随着类的加载在方法区中分配内存，所以类的每个静态变量的抽象内存模型也应该只有一份。在路径分析中，即使是通过不同的类实例来访问和修改静态成员，影响的也只是同一个变量。因此，为每个初次遇见的类的 Class 对象构建类类型的抽象内存模型，其结构如表 2 所示，随着路径分析建立类类型的抽象内存模型与静态变量之间的关联关系。虽然 C＋＋的这一过程与 Java 不同，但是在符号执行中可以同等处理。

表 2 类类型的抽象内存模型属性

属性	说明
symbol	代表 Class 对象的符号
virtualAddr	类类型的抽象内存单元地址
nameDeclaration	类对应符号表项
name	类名
type	Class 对象的数据类型
members	类的静态变量名和各个静态变量对应抽象内存模型的映射关系

不同类别的抽象内存模型放在对应的抽象内存区中，并通过地址来定位。针对不同的抽象内存区，地址使用不同的前缀，基本抽象内存模型所在区的前缀为 Mn，数组抽象内存模型所在区的前缀为 Ma，指针抽象内存模型所在区的前缀为 Mp，类的抽象内存模型所在区的前缀为 Mc。

符号表项和抽象内存模型是一一对应的，被测函数中的每个类、变量以及复杂变量的成员变量都有这两项信息。为了提高运行效率，members 初始为空，只有在初次遇到静态变量时，才为其创建抽象内存模型，并将映射关系添加进去，下一节中类对象的抽象内存模型同理。

1.4　类对象的抽象内存模型

在路径分析中，对于首次遇到的类对象，要为其构建类对象的抽象内存模型。类对象离不开它所属的类，每个类对象都可以访问所属类的静态成员，这就需要指向对应的类类型的抽象内存模型的指针。由于面向对象程序的多态性，类对象的所属类需要进一步分为引用所属类和实际所属类，以便对类型转换语句和函数调用点进行分析，两者的意义不同，引用所属类从指向对象的指针或引用处获取，实际所属类记录于类对象的抽象内存模型中。类对象的抽象内存模型的结构如表 3 所示。

表 3　类对象的抽象内存模型属性

属性	说明
symbol	代表当前变量的符号
virtualAddr	类对象的抽象内存单元地址
nameDeclaration	变量对应符号表项
name	变量名
source	变量来源
realCType	类对象的实际所属类
members	类对象的非静态变量名和各个非静态变量对应抽象内存模型的映射关系

name 要么为变量声明时的名字，要么为复杂变量的成员变量名，如数组变量 array 的第一个成员变量的变量名为 array［0］。

source 指明该变量是输入参数、局部变量、全局变量还是类成员变量，又或者是上述某个复杂变量的成员变量。其中，输入参数、全局变量、类成员变量均是测试用例的组成部分，局部变量则不需要放在测试用例中。

realCType 是类对象的抽象内存模型中最重要的属性，这是提取类型约束的关键，每个类对象的抽象内存模型都有指向所属类对应的类类型的抽象内存模型的指针。即使存在父类引用指向子类对象的情况，类对象能访问的静态成员也只与当前引用所属类有关，不会受实际所属类影响。例如，类 Apple 继承了类 Fruit，定义变量 Apple a，将其向上转型为 Fruit，此时它的引用所属类为 Fruit，实际所属类为 Apple，可以访问 Fruit 的静态成员，而无法访问 Apple 的特有属性。然而，类对象的实际所属类在对路径的语义模拟和可达性分析、以及作为后续测试用例生成的参考中有很大作用。

2　类的操作语义模拟算法

类对象作为非数值类型变量，在符号执行中仅仅使用一个符号[20]对其进行表示是不够的，也不利于使用约束求解器[21]对其求解，需要在路径分析中通过动态地抽象内存建模来描述它的语义和约束。使用抽象内存模型对非数值型变量的约束进行处理，从中提取出数值型约束，并将剩余部分转化为抽象内存中的存储结构。操作语义模拟算法能够在符号执行时根据某个程序点之前各个变量的语义和约束信息，以及当前语句的语义信息来更新相关变量的抽象内存模型，模拟出抽象内存中的状态变化。在符号执行中，当被测函数含有类对象时，构建类的抽象内存模型，配合操作语义模拟算法，提取出路径中的约束，对类对象的具体类型进行限定，从而生成满足路径条件的测试用例或者得到函数调用点处调用方法所属的类。与类有关的操作有对象创建、成员访问和类型转换，接下来以 C++语言为例，分析每种操作对应的操作语义模拟算法，并用形式化语言进行描述。

2.1　对象创建

如表 4 所示，在 C++中，创建对象有两种方式——直接定义（见①）和通过指针创建（见②）。前者与基本数据类型变量的定义格式类似，对象在栈上分配内存，不会体现面向对象程序的多态性；后者定义一个指向对象的指针，后续可以使用指针访问对象的成员变量和成员函数，对象本身是匿名的，在堆上分配内存，可能存在父类指针指向子类对象的情况，此时会出现多态[22]。使用 new 创建的对象需要配合 delete 及时删除，以防止无用内存堆积。

表 4　对象创建的两种方式

序号	代码
①	ClassName var;
②	ClassName * p (= new ClassName);

对形如①的语句，直接为变量 var 创建类的抽象内存模型，首先判断是否已经创建了类 ClassName 对应的类类型的抽象内存模型，如果没有就进行构建；随后为变量 var 本身创建类对象的抽象内存模型，指定变量来源，变量的实际所属类为 ClassName。无论是初次创建类类型的抽象内存模型还是初次创建类对象的抽象内存模型，都需要为其指定一个符号，在类的抽象内存区中新建一块抽象内存单元，并与对应的符号表项进行关联，且 members 初始均为空。

对形如②的语句，先为指针 p 创建指针抽象内存模型，指定指针来源，定义指针状态为非空，再按上述步骤为指针 p 指向的类对象创建类的抽象内存模型，建立两者之间的联系。综上所述，对象创建的操作语义模拟算法的形式化语言描述如表 5 所示。如果只是声明 ClassName 类的指针 p，没有对其赋值，那么它的初始状态为不确定，此时暂不需要创建类的抽象内存模型；如果约束指针状态为非空，但指针指向对象的类型尚不确定，那么类对象的实际所属类为空；如果类对象的引用所属类和实际所属类不同，还要判断两者是否存在继承关系。

表 5 对象创建的操作语义模拟算法

序号	算法

①	if((model = getClassModel(ClassName))==null)
	model = createClassModel(ClassName);
	createObjectModel(var)
	.setSource(source).setRealCType(model);

②	createModel(p) .setSource(source).setPTState(NOT_NULL) .setPT(createModel(*p));

2.2 成员访问

如表 6 所示，在 C++中，根据创建对象方式的不同，访问成员的方式也有两种——直接访问（见③）和通过指针访问（见④）。

表 6 成员访问的两种方式

序号	代码
③	var. mem
④	p －＞ mem

访问对象的 mem 成员时，如果已经为其构建了抽象内存模型，则判断是否需要建立对象和成员之间的关联关系，并根据成员类型和语义信息提取约束即可，否则为其构建抽象内存模型。对形如③的语句，构建 var 和 var. mem 之间的关联关系；对形如④的语句，构建 * p 和 p－＞mem 之间的关联关系。如果 mem 是静态变量，关联关系建立在类类型的抽象内存模型中；如果 mem 是非静态变量，关联关系建立在类对象的抽象内存模型中。综上所述，以形如③的语句为例，成员访问的操作语义模拟算法的形式化语言描述如表 7 所示。

表 7 成员访问的操作语义模拟算法

序号	算法

③	if ((model = getModel(var. mem)) == null)
	model = createModel(var. mem);
	if (! isStatic(var. mem))
	getModel(var). putMember(model. getName(),model);
	if (isStatic(var. mem))
	getModel(var). getRealCType(). putMember
	(model. getName(), model);

值得注意的是，如果子类 Apple 继承了父类 Fruit 的静态成员 color，在抽象内存中同时存在类 Fruit 的抽象内存单元和类 Apple 的抽象内存单元，且两个类类型的抽象内存模型的 members 都包含 color，那么它们存储的 color 理应指向同一个抽象内存模型，为便于查找，类的静态成员的完整变量名统一为定义所在类的类名＋变量名，比如，在这个例子中，类成员 color 在抽象内存模型中存储的变量名为“Fruit:: color”。

2.3 类型转换

类型转换是类的操作语义模拟算法关注的重点，只有当路径中存在对类对象的类型判断或者类型转换语句时，才能根据不同分支或是能使程序继续执行所需的转换条件对类对象的类型进行约束。对象的向上转型会自动完成，因为向上转型一定是安全的，但是一旦转型为父类对象，就无法再调用子类原本特有的方法；对象的向下转型需要进行强制类型转换，且必须先发生过向上转型、才能成功向下转型，否则会报错。在编译时，编译器无法判断对象实例化时传递的是什么数据类型，因此不会对强制类型转换进行

检查。

C++有 4 种强制类型转换函数，分别为 const _ cast、static _ cast、dynamic _ cast 和 reinterpret _ cast。其中，static _ cast 和 dynamic _ cast 均可以用于类层次结构中基类和派生类之间指针或引用的转换。static _ cast 类似 C 语言中的强制转型，不提供运行时类型检查，因此在进行类的向下转型时具有一定的安全隐患；而 dynamic _ cast 会在运行时对类型信息进行检查，对于无法强制转型的变量会返回 nullptr，从而保证类型转换的安全性。由于这两个函数的特点，C++中的所有隐式类型转换都会调用 static _ cast；而在编码时，对于显式类型转换则常常通过 dynamic _ cast 来实现[23]。在 Java 中，通常使用 instanceof 关键字来判断某个实例是否是某个类的对象。

如果出现类型判断语句或者类型转换语句，判断类对象是否已经确定实际所属类，如果没有且待转换类型与引用所属类存在继承关系，用位于更底层的类型更新实际所属类，继续执行；如果有，判断待转换的类型是否处于引用所属类与实际所属类所在的继承体系中，如果在，用三者中位于更底层的类型更新实际所属类，继续执行，否则说明路径不可达。综上所述，类型转换的操作语义模拟算法的形式化语言描述如表 8 所示。

表 8　类型转换的操作语义模拟算法

序号	算法

⑤	addrCT = getModel(p).getAddrCType();
	realCT = getModel(p).getPT().getRealCType();
	if (realCT == null)
	if (extends(castCT, addrCT))
	getModel(p).getPT().setRealCType(lower(castCT, addrCT));
	else
	// 路径不可达
	else
	if (extends(castCT, addrCT) && extends(castCT, realCT))
	getModel(p).getPT().setRealCType(lower(castCT, realCT));
	else
	// 路径不可达

3　基于抽象内存模型的测试用例生成

对函数模块执行单元测试可以采用基于输入域的随机测试、边界值测试和基于路径的随机测试等。提取出函数的输入变量，根据每个变量的数据类型生成多个随机值并组合成测试用例。其中，基于输入域的随机测试是指在变量的取值区间内生成随机值；边界值测试指的是在变量的取值范围的边界处生成随机值，除此之外还要考虑某些特殊值，例如对于整型变量，要特别考虑取值为 0 的情况。这两种测试方式简单高效，无需考虑函数内部的实现细节，可以很快生成大量测试用例，但是也很容易造成过多的冗余测试用例，并且生成的测试用例很难能够执行到函数内部某些条件苛刻的语句[24]。此时可以根据函数的控制流图提取出未覆盖元素集合，使用基于路径的随机测试来生成覆盖到这些元素的测试用例。

当被测函数的输入变量含有基类的指针或引用且函数中出现了类型转换语句时，如果不对路径信息进行分析，就无法判断对变量进行初始化时应该传递哪种类型的实例。就算先不考虑基类为抽象类的情况，如果直接对该基类对象实例化，很有可能在类型转换时出错、或者在类型判断时无法执行到相应分支；如果对该基类的所有底层子类创建实例化对象，再以父类引用指向子类对象的方式进行赋值，很有可能会生成众多冗余的随机值，使最终得到的测试用例集合非常庞大，且测试的复杂程度会随着代码和继承体系复杂程度的提高而成倍增长，然而这其中很多都是没有必要的，会大大降低测试效率。这时，通过构建类的抽象内存模型，对类对象的具体类型进行约束，就能使生成的测试用例以较少的数量覆盖到尽可能多的语句。

分析被测函数，提取类的类型判断语句和类型转换语句对应的覆盖元素，从下往上逐个分析。对于当前覆盖元素，判断其是否为未覆盖元素，若为未覆盖元素，生成经过该覆盖元素的路径，从函数入口开始，为输入变量中的类对象构建类的抽象内存模型，通过符号执行提取出路径上类相关的约束，得到变量的类型信息。当获得的类型信息中对象的实际所属类与之前不同时，如果分析得到的类是非抽象类，直接为其生成随机值对象；如果分析得到的类是抽象类，为其距离最近的非抽象子类生成随机值对象。将各个

输入变量的随机值组合成多组测试用例并代入执行，根据执行后的插装信息更新已覆盖元素集合和未覆盖元素集合[25]，如此反复。直到全部类型判断语句及其分支和类型转换语句均已被覆盖，若覆盖率已达到100%，结束测试；否则考虑变量的实际类型为基类本身的情况，如果基类为非抽象类，直接对其实例化，不然就对距离基类最近的非抽象子类生成实例化对象。

对类对象的类型信息进行提取，除了在测试用例生成中起到了很大作用之外，在函数摘要的提取中也能派上用场。通过路径分析得到函数调用点处对象的实际所属类，从而得知动态执行时可能会调用哪些子类的方法，进而提取相应函数的函数摘要。先通过引用所属类对应的符号表项获取到方法的相关信息，如果该方法不能被重写，说明调用的就是引用所属类中的方法；如果方法可以被重写，就需要根据对象的实际所属类来判断。从实际所属类开始，由下往上查找该方法，直到找到该方法最新被重写的地方，即为后面会被调用执行的位置。

现如今，许多研究人员都在思考具有更高测试用例利用率的自动测试用例生成方法，如通过约束求解来提高测试用例命中率。在面向过程程序的单元自动化测试领域，北京邮电大学的唐荣对 C 语言中非数值类型变量的抽象内存模型和约束提取算法进行了研究和设计，实现了支持非数值型测试用例自动生成的面向路径的约束求解测试。在面向对象程序的自动化测试领域，类虽然是重点研究对象，然而大部分研究工作都局限于单个类内部一个或多个函数间的测试，没有考虑到由于类的继承和多态等特性所导致的多个类之间的相互影响，也没有对函数内部的类型转换语句进行处理。北京邮电大学的陈江南在研究面向路径的类测试方法时，提出了类成员方法扩展控制流图生成算法，根据被调用函数所属类所在的继承体系，将完整路径分为基本子路径和实例化子路径两部分，所有可能的函数调用情况都作为实例化子路径配合分支节点添加到原有的控制流图中，两部分路径分别生成后再进行组合。陈江南的研究默认基本子路径不会涉及对类对象实际类型的约束，相当于为所有派生类对象生成取值。在这一方面，中国科学技术大学的黄双玲在研究 C++程序中函数调用关系的静态分析方法时，考虑到了函数内部的类型转换语句，在记录变量的类型信息时，会分别记录变量的声明类型和动态类型，以便对后续出现的函数调用语句进行解析。

目前已有的面向对象自动化单元测试工具，如针对 C/C++语言的 Parasoft C++ Test 和针对 Java 语言的 Randoop，其研发的重心并不在函数输入变量中类对象的具体类型。当出现类的指针或引用变量时，很多都只为基类对象生成取值。只为基类对象生成取值、为所有派生类对象生成取值，以及对变量类型进行约束后生成取值，这三种测试用例生成方式的结果对比见下文中第五节。

4 实例分析

为了验证前两节中介绍的操作语义模拟算法和基于路径的随机测试算法的可行性，接下来以图 2 中的被测函数为例，演示路径分析中抽象内存的变化过程，展示如何通过类的抽象内存建模提高函数的覆盖率。

```
// Food继承于Goods
// 类Goods含有非静态整型变量weight和静态浮点型变量ratio
int totalWeight = 0;
int totalFoodWeight = 0;

void stock(Goods *goods) {
    totalWeight += goods -> weight * goods -> ratio;          // L1
    cout << goods << endl;
    if (Food *food = dynamic_cast<Food *>(goods)) {           // L2
        totalFoodWeight += food -> weight * food -> ratio;    // L3
        cout << food << endl;
    }
}
```

图 2 商店进货的代码片段

函数 stock（）的输入参数包含 Goods 类的指针 goods，且函数体内存在对变量的类型转换语句 L2，它也是一条判断语句。初始时，已覆盖元素集合为空，L2 尚未被覆盖，生成一条经过该条件表达式真分支的路径 Path：L1－>L2－>L3。在路径的起始节点处，为指针 goods 在指针抽象内存区分配抽象内存单元 Mp_0，指针来源为输入参数，此处指针取值还无法确定，所以指针状态为 NOT _ SURE，处理完毕后抽象内存的状态如表 9 所示。

表 9 指针抽象内存区状态一

virAddr	name	source	type	PT	state
Mp_0	goods	PARM	Goods *		NOT_SURE

分析 L1 语句的语义——将指针 goods 所指对象的两个成员 weight 和 ratio 相乘，并将乘积与全局变量 totalWeight 相加的结果赋值给 totalWeight。由指针的约束提取算法可知，应为指针添加非空约束，且需要为指针指向的类对象创建类的抽象内存模型。类 Goods 对应的类结构抽象内存模型尚未被创建，为其在类的抽象内存区中分配抽象内存单元 Mc_0。为类对象创建类实例抽象内存模型，分配内存单元 Mc_1，变量名为 * goods，变量来源为参数成员，其实际所属类为 Goods。为类对象 * goods 添加成员 goods －＞weight 和 goods－＞ratio，它们的数据类型分别为整型和浮点型，在基本抽象内存区中新建抽象内存单元 Mn_0 和 Mn_1。由于 weight 为实例变量，只与 * goods 这个具体实例有关，将其添加到 Mc_1 的成员域中；而 ratio 为静态变量，与类有关，且在类 Goods 中定义，因此抽象内存模型中存储的变量名为 Goods：：ratio，将其添加到 Mc_0 的成员域中。最后，为整型变量 totalWeight 新建抽象内存单元 Mn_2，变量来源为全局变量，并根据表达式信息更新其符号值。执行完这些操作后，抽象内存的状态如表 10～12 所示。

表 10　指针抽象内存区状态二

virAddr	name	source	type	PT	state
Mp_0	goods	PARM	Goods *	Mc_1	NOT_NULL

表 11　类的抽象内存区状态二

virAddr	name	source	realCType	members
Mc_0	Goods	DECL		{Mn_1}
Mc_1	* goods	MEM	Mc_0	{Mn_0}

表 12　基本抽象内存区状态二

virAddr	symbol	name	source	type
Mn_0	g_w	goods－＞weight	MEM	INT
Mn_1	G_r	Goods：：ratio	MEM	FLOAT
Mn_2	tw＋g_w * G_r	totalWeight	GLOB	INT

分析 L2 语句的语义——将 Goods 类的指针变量 goods 向下转型为 Food 类的指针并赋值给局部变量 food，判断 food 是否为空指针，不为空指针的真分支继续执行 L3。为指针变量 food 在指针抽象内存区中新建一块抽象内存单元 Mp_1，指针来源为局部变量。要使条件表达式结果为真，应使变量 goods 能够成功进行类型转换，对指针 food 添加不为空的约束。类对象 * goods 的引用所属类和实际所属类均为 Goods 类，类 Food 是类 Goods 的子类，约束 * goods 的实际所属类为位于更底层的 Food 类。为 Food 类在类的抽象内存区中分配一块抽象内存单元 Mc_2，Mc_1 的实际所属类指向它。对指针 goods 的类型转换不会改变它的取值，即所指类对象的地址，因此指针 food 应指向同一个类对象，Mp_0 和 Mp_1 的指针域均为 Mc_1。执行完这些操作后，抽象内存的状态如表 13～15 所示。

表 13　指针抽象内存区状态三

virAddr	name	source	type	PT	state
Mp_0	goods	PARM	Goods *	Mc_1	NOT_NULL
Mp_1	food	LOCAL	Food *	Mc_1	NOT_NULL

表 14　类的抽象内存区状态三

virAddr	name	source	realCType	members
Mc_0	Goods	DECL		{Mn_1}
Mc_1	* goods	MEM	Mc_2	{Mn_0}
Mc_2	Food	DECL		{}

表 15　基本抽象内存区状态三

virAddr	symbol	name	source	type
Mn_0	g_w	goods－＞weight	MEM	INT
Mn_1	G_r	Goods：：ratio	MEM	FLOAT
Mn_2	tw＋g_w * G_r	totalWeight	GLOB	INT

分析 L3 语句的语义——将指针 food 所指对象的两个成员 weight 和 ratio 相乘，并将乘积与全局变量 totalFoodWeight 相加的结果赋值给 totalFoodWeight。指针 food 指向的类对象为 * goods，检查它的两个成员 weight 和 ratio 是否已经被添加。其中，weight 为实例变量，已经被添加进类实例抽象内存模型 Mc_1 的成员域中；ratio 为静态变量，在类 Goods 中定义，完整变量名为 Goods：：ratio，在基本抽象内存区中已经创建了相应的抽象内存单元 Mn_1，然而它并没有与 * goods 的实际所属类 Food 对应的类结构抽象内存模型 Mc_2 建立关联关系，将

其添加进 Mc_2 的成员域中。最后，为整型变量 totalFoodWeight 新建抽象内存单元 Mn_3，变量来源为全局变量，并根据表达式信息更新其符号值。执行完这些操作后，抽象内存的状态如表 16～18 所示。

表 16 指针抽象内存区状态四

virAddr	name	source	type	PT	state
Mp_0	goods	PARM	Goods *	Mc_1	NOT_NULL
Mp_1	food	LOCAL	Food *	Mc_1	NOT_NULL

表 17 类的抽象内存区状态四

virAddr	name	source	realCType	members
Mc_0	Goods	DECL		{Mn_1}
Mc_1	* goods	MEM	Mc_2	{Mn_0}
Mc_2	Food	DECL		{Mn_1}

表 18 基本抽象内存区状态四

virAddr	symbol	name	source	type
Mn_0	g_w	goods－＞weight	MEM	INT
Mn_1	G_r	Goods::ratio	MEM	FLOAT
Mn_2	tw＋g_w * G_r	totalWeight	GLOB	INT
Mn_3	tfw＋g_w * G_r	totalFoodWeight	GLOB	INT

对路径 Path：L1－＞L2－＞L3 分析完毕后，得到对函数 stock（）的输入参数 goods 的约束信息，即指针所指向对象的类型应为 Food 类或其子类。此处 Food 类为非抽象类，直接对其随机生成多个实例化对象，并将对象的地址传递给指针 goods。将输入变量 goods、totalWeight 和 totalFoodWeight 的随机值组合成多组测试用例，代入并动态执行后，根据探针函数的返回值更新已覆盖元素集合和未覆盖元素集合，计算覆盖率。发现可以达到 100％，可知当前测试用例集合已满足测试需求，结束测试。

5 实验结果及分析

代码测试系统（CTS，code testing system）是一款面向 C 语言程序的自动化单元测试工具，它采用动静结合的方式，支持以函数模块为单元执行基于输入域的随机测试、边界值测试和面向路径的测试。CTS 已经完善了对 C 语言类型系统的符号表、抽象内存模型和操作语义模拟算法的设计与实现。CTS－CPP 是 CTS 的 C＋＋版本，在其基础上提供了对类的支持。本课题对 CTS 中的符号表和抽象内存模型进行扩展，将类的操作语义模拟算法和基于路径的随机测试算法应用于面向 C＋＋程序的自动化测试工具 CTS－CPP 中，使 CTS－CPP 能够提供基于输入域的随机测试和基于路径的随机测试功能。

5.1 实验环境

本课题在 CTS－CPP 中完成测试用例生成模块，其运行于 CentOS 7 系统中，JDK 版本为 1.8，使用 Java 语言在 Eclipse 平台中开发，虚拟机最大内存设置为 2G。

5.2 实验内容

为了验证本课题提出的模型和算法的可行性和有效性，本章对表 19 中的 5 个函数进行了基于路径的随机测试，记录测试过程中生成的路径和约束提取情况，并将测试结果同基于输入域的随机测试作比较。

表 19 5 个被测函数属性

工程名	函数名	输入变量个数	类对象个数	类型转换节点个数	继承体系中类个数	代码行数	代码总行数
CoffeeM	choose()	7	2	4	10	46	184
Shop	addThing()	9	3	7	11	69	202
MainA	dynamicA()	6	2	6	12	71	217
Segment	jumpTo()	7	4	9	15	91	283
Compiling	compileB()	5	2	6	13	67	230

5.3 实验结果

为了展示类的抽象内存模型结合操作语义模拟算法是否能够成功提取路径中类相关的约束，得到类对象的实际所属类，以 choose（）函数为例，执行基于路径的随机测试。choose（）函数的输入变量有函数参数 coffee、balance、time 和所属类 Shop 的成员变量 sales、bean、milk、choco，其中变量 coffee 是 Coffee 类的指针，Coffee 类存在于继承体系中，是 Instant、Latte、Mocha、White 等众多类的公共基类。在函数内部有 5 个类型转换节点，对应 4 个覆盖元素，根据 coffee 指向的子类类型不同，会执行不同分支。其路径生成和约束提取情况如表 20 所示。

同理，5 个被测函数的路径生成和约束提取情况如表 21 所示。

对程序进行静态分析，将变量、函数和类的基本

表 20　choose()函数路径生成和约束提取情况

序号	覆盖元素	路径序列	* coffee 实际所属类	* espresso 实际所属类	是否生成随机值	覆盖率是否提高
1	if_head_26_159	0－1－2－3－5－6－8－15－23－24－26－27－29－30－33－35－37－38－41－44－46－47－48－49－51－53－54	Viennese	Coffee	是	是
2	if_head_19_151	0－1－2－3－5－6－8－15－16－18－19－20－22－32－35－37－38－41－44－46－50－52－53－54	Mocha	Espresso	是	是
3	if_head_13_144	0－1－2－3－5－6－8－9－11－12－14－34－37－38－41－44－46－50－52－53－54	Latte 或 White	Espresso	是	是
4	if_head_10_140	0－1－2－3－5－6－7－36－38－41－44－46－50－52－53－54	Espresso	Espresso	是	是

表 21　5 个被测函数路径生成和约束提取情况

函数名	生成路径个数	约束是否成功提取	覆盖率是否提高
choose()	4	是	是
addThing()	7	是	是
dynamicA()	6	是	是
jumpTo()	7	是	是
compileB()	6	是	是

信息存入符号表中，生成路径后，在符号执行中根据类的操作语义模拟算法构建并更新类的抽象内存模型，对被测函数的输入变量中类对象的实际类型进行约束，根据分析结果为各个输入变量在其取值区间内生成随机值并由此得到测试用例集合，在动态执行后统计函数的覆盖情况，这就是基于路径的随机测试的主要流程。其通过对对象的实际所属类进行限定来避免生成无意义的测试用例，从而提高测试效率。如果不采取这一举措，也就是采用传统的基于输入域的随机测试的话，对于输入变量中的类对象，可以选择只为引用所属类生成实例化对象，也可以选择为引用所属类的所有子类生成随机值对象，使用这一方式虽然可以快速生成大量测试用例，但是其中的冗余对测试效率的影响不可小觑。

为了更好地说明在对类的抽象内存模型进行研究后，提出的基于路径的随机测试相比于基于输入域的随机测试在提高测试效率方面的优越性，分别采用两种方式对 5 个被测函数执行自动测试，测试结果如表 22 所示。

表 22　测试结果

函数名	测试方式					
	基于输入域的随机测试				基于路径的随机测试	
	只为基类对象生成随机值		为所有派生类对象生成随机值			
	测试用例数量	覆盖率/%	测试用例数量	覆盖率/%	测试用例数量	覆盖率/%
choose()	28	33.3	56	100	40	100
addThing()	36	18.8	132	87.3	44	87.3
dynamicA()	24	36.4	88	100	48	100
jumpTo()	28	13.2	124	75	60	75
compileB()	20	47.7	100	100	52	100

5.4　结果分析

由实验结果可知，如果执行基于输入域的随机测试，只对引用所属类的对象生成随机值得到的测试用例数量和覆盖率均不高；对引用所属类的所有子类对象生成随机值虽然能够得到较高的覆盖率，然而其生成的测试用例数量同样很高。与此相比，基于路径的随机测试通过对类对象的具体类型进行限定，能够以更少的测试用例达到更高的覆盖率。在大型程序中，测试效率的提高将会更为明显。

综上所述，本课题提出的类的符号表能够提取出测试用例生成所需的类的基本信息，类的抽象内存模型和操作语义模拟算法能够成功记录路径中类对象的语义和约束信息，将它们应用于基于路径的随机测试中，能够得到满足需求的结构和内容均正确的测试用例，并且提高了对面向对象程序单元测试的测试效率。

6 结束语

本课题基于面向对象程序特性，对类的抽象内存模型进行研究，并由此提出了类的操作语义模拟算法以及针对单元测试的基于路径的随机测试算法的概念。类的抽象内存模型能够记录类对象的类型信息及其与各个成员之间的关联关系，考虑到多个实例可能对同一个静态变量产生影响，将类的静态成员与非静态成员区别开，分别存储在类类型的抽象内存模型和类对象的抽象内存模型中。使用类的抽象内存模型，不仅能够在符号执行中提取类型约束，缩小类对象实际类型的可选范围，还能在今后配合其他类别的抽象内存模型获取其各个成员变量的约束信息[26]，结合约束求解实现更为精确的面向路径的测试。

参考文献：

[1] 龙高贵. 谈软件工程中软件测试的重要性及方法 [J]. 电脑迷，2017 (8)：215，9.

[2] 陈和平. 面向对象的自动化单元测试 [D]. 武汉：武汉理工大学，2004.

[3] 赵良福，王世签，郑科鹏. 软件自动化测试研究 [J]. 有线电视技术，2018，25 (6)：95－97.

[4] 王雅文，宫云战，杨朝红. 软件测试工具综述 [J]. 北京化工大学学报 (自然科学版)，2007，34 (s1)：1－4.

[5] 刘 芳. 面向对象自动化单元测试技术研究 [J]. 现代计算机，2020 (14)：21－26.

[6] 岳青玲. Java 面向对象编程的三大特性 [J]. 电子技术与软件工程，2019 (24)：239－240.

[7] 吴晓琴. 浅析面向对象程序设计特点 [J]. 安徽大学学报 (自然科学版)，2002 (3)：33－37.

[8] 黄双玲. 面向 C/C＋＋程序函数调用关系的静态分析方法研究 [D]. 合肥：中国科学技术大学，2015.

[9] GOTLIEB A. Euclide：a constraint － based testing framework for critical C programs [C] //Proceedings of the 2nd International Conference on Software Testing Verification and Validation. Los Alamitos：IEEE Computer Society Press，2009：151－160.

[10] XU Z X，ZHANG J. A test data generation tool for unit testing of C programs [C] //Proceedings of the 6th International Conference on Quality Software. Los Alamitos：IEEE Computer Society Press，2006：107－116.

[11] ZHANG J. Symbolic execution of program paths involving pointer and structure variables [C] //Proceedings of the 4th International Conference on Quality Software (QSIC). Piscataway，NJ：IEEE，2004：87－92.

[12] ZHAO Y S，WANG Y W，GONG Y Z，et al. STVL：improve the precision of static defect detection with symbolic three－valued logic [C] //Proceedings of the 18th Asia－Pacific Software Engineering Conference，Los Alamitos：IEEE Computer Society Press，2011：179－186.

[13] 王留帅. 基于函数摘要的 C＋＋过程间静态分析研究 [D]. 成都：电子科技大学，2017.

[14] 代子营. 面向符号执行的内存模型研究 [D]. 长沙：国防科技大学，2009.

[15] 方登辉. 基于抽象语法树的代码静态缺陷检测工具开发 [D]. 北京：北京邮电大学，2018.

[16] 庞新法. C＋＋继承性剖析 [J]. 价值工程，2014，33 (18)：207－208.

[17] 靳乔乔，王静一，郭怡冰，等. 浅析 Java 与 C＋＋的区别 [J]. 数码世界，2018 (10)：65.

[18] 谭国律，王 波，刘 萍. 程序设计中的静态元素 [J]. 电脑编程技巧与维护，2021 (10)：3－7，15.

[19] 唐 容. 支持非数值型测试用例自动生成的抽象内存建模技术研究 [D]. 北京：北京邮电大学，2013.

[20] KING J C. Symbolic execution and program testing [J]. Communications of the ACM，1976，19 (7)：385－394.

[21] SANOGO S，MESSINE，FRéDéRIc. Design of space thrusters：a topology optimization problem solved via a Branch and Bound method [J]. Journal of Global. Optimization，2016，64 (2)：273－288.

[22] 徐正威，周 琼，许 珂，等. 浅谈 Java 与 C＋＋中的内存管理 [J]. 网络安全技术与应用，2016 (3)：50－51.

[23] 侯 勇. C＋＋数据类型转换方法及其应用 [J]. 电脑与电信，2010 (4)：64－65.

[24] 王廷永，黄 松. 测试用例自动生成技术综述 [J]. 电子技术与软件工程，2021 (18)：51－53.

[25] 于 航. 单元测试中抽象内存模型优化技术研究 [D]. 北京：北京邮电大学，2019.

[26] 李飞宇. 基于内存建模的测试数据自动生成方法研究 [D]. 北京：北京邮电大学，2013.