基于角色的设计模式形式建模及演化

孙军梅[1] 缪淮扣[2]
(杭州师范大学信息科学与工程学院 杭州 310036)[1] (上海大学计算机工程与科学学院 上海 200072)[2]

摘 要 重用设计在改善软件开发过程中的作用越来越受到人们的重视。面向对象的软件设计模式概念的提出为软件设计的重用打开了新的局面。但在设计模式的具体应用中存在实现、文档和组合的问题。给出了一种基于角色的设计模式形式建模方法,将类、类属性、类之间的关系等都看作角色,并用形式建模语言 Object-Z 形式表示这些角色,可有效地解决设计模式实例化时存在的问题。基于角色的设计模式形式模型在演化时也基于角色进行演化,将模式的演化分为角色层演化和模式层演化,模式层演化是由多个角色层演化组成的。演化后的模型可以通过定理证明器验证与前模型是否保持一致。

关键词 设计模式,Object-Z,角色,演化,验证
中图法分类号 TP311 **文献标识码** A

Formal Modeling and Evolution of Design Pattern Based on Role

SUN Jun-mei[1] MIAO Huai-kou[2]
(School of Information Science and Engineering, Hangzhou Normal University, Hangzhou 310036, China)[1]
(School of Computer Engineering and Science, Shanghai University, Shanghai 200072, China)[2]

Abstract Design reuse becomes important in improving software development process. The concept of object oriented design pattern opens the situation for software design reuse. There are barriers when instantiating the design patterns, such as implementation, documentation, composition. This paper presented a formal modeling approach based on role. Class, attribute of class, the relation between class all are treated as roles and all roles are modeled with Object-Z. This effectively resolves the barriers when instantiating the design patterns. The formal model of design pattern is also evolved based on role. The evolution is divides into role layer evolution and pattern layer evolution. Pattern layer evolution is composed of role layer evolution. The model consistence can be verificated with formal theory prover.

Keywords Design pattern, Object-Z, Role, Evolution, Verification

1 引言

人们一直在致力于改善软件的开发过程(从代码的重用到设计的重用)。对于代码的重用人们提出了基于构件的软件开发等方法。如何重用设计? GOF 在文献[1]中提出的面向对象的软件设计模式(以下简称设计模式)概念,为设计的重用打开了新的局面。设计模式提供了可重用的软件设计方案,被广泛地应用于软件设计过程中。设计模式的具体应用称为模式的实例化。虽然设计模式得到了广泛的应用,但在实例化模式时还是存在很多问题,文献[2]把它们归纳为以下3个最主要的问题:实现(implementation)、文档(documentation)和组合(composition)。

设计模式的实现往往要根据使用的具体环境来编写,造成设计模式的思想可以被重用,而实现不能被重用,特别是在系统同时使用多个设计模式,模式的代码分散在业务代码中时,如有几个模式组合使用,就会出现应用程序类在模式组合中不止充当一种角色,即包含在多个模式中,这时候很难区分这些模式实例以及它们之间的关系,重用模式代码就变得比较困难。造成这种情况的根本原因是设计模式建模以对象为建模元素。本文提出以角色为单位、基于角色的模式建模。把模式的建模元素(包括类角色、关系角色、属性角色、操作角色等)所扮演的角色显式地写在模式的代码实现中,便可有效地避免用对象概念描述模式时模式的本质属性隐藏在具体实例的细节上而又无法判断到底是扮演哪个模式的哪个角色问题。

传统的模式用自然语言描述或用半形式化语言 UML 表示的比较多,用一个具体模式使用的例子来描述模式是很常见的。但这种描述方式缺乏精确性,在模式的理解上容易出现分歧。模式描述必须表示模式的一般属性,以便可以使用各种方式来实现模式。因此,用作定义模式的语义应有能力抽象地表达模式。UML 在表示模式抽象性方面有限制。例如,抽象工厂模式中扮演具体工厂和具体产品角色的数目用

到稿日期:2008-09-18 返修日期:2008-11-26 本文受国家自然科学基金项目(批准号:60673115)和上海市重点学科建设项目(项目编号:J50103)资助。

孙军梅(1974—),女,博士,副教授,主要研究方向为软件体系结构、形式化方法等,E-mail:jmei_sun@yahoo.com.cn;缪淮扣(1953—),男,教授,博士生导师,主要研究方向为软件工程、形式化方法。

UML 就很难表示。

另外，在基于模式的软件开发中，模式作为模型转换的基础，若要实现模型的自动化转换，模式描述必须精确以便能精确地使用和推理模式。传统的对象符号在这方面有一定的限制。例如，抽象工厂设计模式中具体工厂类和具体产品类之间的每个创建产品操作的同构关系（isomorphic relationships）用对象符号中的聚集和继承关系不能很好描述。本文采用形式化建模语言 Object-Z[3] 形式化表示设计模式，避免了用非形式化或半形式化建模语言表示设计模式时的不精确性。

2 基于角色的设计模式形式建模

2.1 角色元模型

（1）角色概念

角色描述模式中的一个参与者，如类、类属性（性质或行为）、类之间或类属性之间的关系。这里的角色概念独立于某种建模语言。

（2）角色元模型

角色元模型的 UML 表示如图 1 所示。定义 Role 为元类，其它类型角色都继承 Role 类，一个角色有一个名字描述它的意图和责任。通过继承 Role 类，定义一个类角色，一个类角色表示由类扮演的角色。然而，不必一个类角色必须由一个类实现。类角色可能是特征角色如属性角色和操作角色。类角色可能由超类角色来定义角色的一般属性。一个模式中的角色是有关系的。关系在这里也作为一个角色。关系角色用来定义其它角色之间的关系，表示为客户和提供者之间的一个二元关系。不像类模型中，关系定义为类之间的关系。在这里角色模型中关系定义为任何两个角色元素之间的关系，如一个操作角色与一个类角色的关系。当一个关系是类角色之间的关系时，有其它的信息去限制在元类关系上下文中的重数约束（如 clientInstanceMultiplicity 和 supplierInstanceMultiplicity）。属性为客户和提供者提供对象层多重性信息。

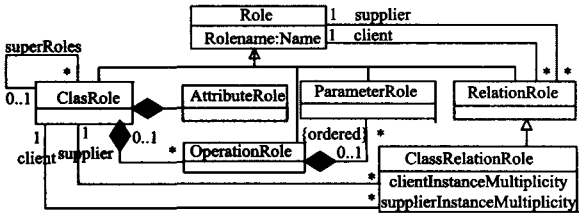


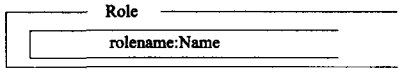
图 1 角色元模型的 UML 表示

2.2 角色概念的形式化定义

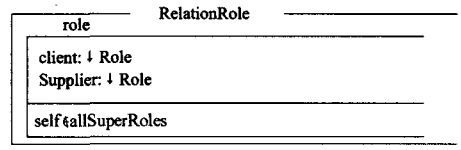
定义 给定集合[Name]，图 1 中的元类 Role 形式化为类 Role，属性 rolename 形式化为 Object-Z 中的变量。

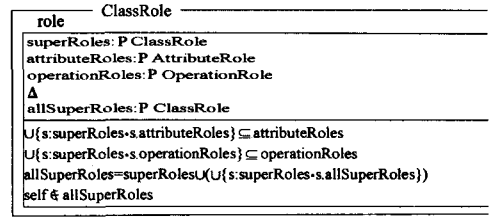
继承 Object-Z 类 Role，类角色也形式化为 Object-Z 类 ClassRole。ClassRole 和 Role 之间的继承关系通过在 ClassRole 类中包含 Role 类名来表示，Role，ClassRole，RelationRole 形式化表示为 Object-Z 模式 Role，RelationRole，ClassRole，如下所示。

形式化元类 Role

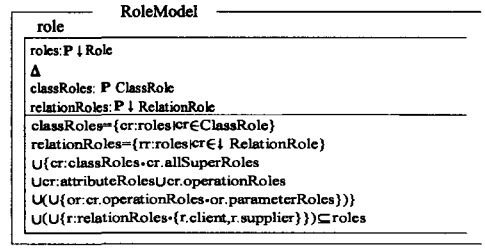


继承类 Role，形式化 ClassRole 和 RelationRole。





角色模型形式化定义为 Object-Z 模式 RoleModel，如下所示。



2.3 设计模式形式化实例

下面用形式规格说明语言 Object-Z 来对 GOF 一书中的一个设计模式：抽象工厂设计模式进行形式化说明，用 Object-Z 可以表示一些 UML 无法表示的性质，然后在后面部分给出如何基于 Object-Z 进行一致性验证及自动生成代码。

抽象工厂模式的意图是：提供一个创建一系列相关或相互依赖对象的接口，而无需指定它们具体的类。用 Object-Z 形式化表示的抽象工厂模式见 AbstractFactory 模式，有下列约束：

（1）ConcreteFactory 类角色继承自 AbstractFactory 类角色。

（2）ConcreteProduct 类角色继承自某一个 AbstractProduct 类角色。

角色之间集成一致性的约束定义如下：

（3）AbstractFactory 类角色有许多名为 CreateProduct 的操作角色，每个 CreateProduct 操作角色创建一种不同的产品。AbstractFactory 中的 CreateProduct 操作角色数与 AbstractProduct 类角色数相同。

（4）每个 AbstractProduct 类角色中有同样数量的 ConcreteProduct 类角色。抽象产品类角色 AbstractProductA 和 AbstractProductB 都有两个具体产品类角色。

角色之间的关系约束如下：

（5）每个 AbstractFactory 类角色中定义的 CreateProduct 操作类角色与一个 AbstractProduct 类角色之间是同构关系。

（6）具体工厂类 ConcreteFactory 中定义的 CreateProduct 操作类角色和具体产品类 ConcreteProduct 角色之间的关系是同构关系，即每个 CreateProduct 操作角色应该只与一个具体产品类角色有联系。

（7）给定一个具体工厂类角色，它的每个 CreateProduct 操作角色应该从不同的抽象产品类角色中创建一个具体产品。

AbstractFactory

absFactory: ClassRole
conFactories:P ClassRole
absProducts:P ClassRole
conProducts:P ClassRole
absCreateProduct:P RelationRole
conCreateProduct:P RelationRole

∀cf: conFactories · absFactory∈cf.superRoles
∀cp: conProducts · ∃ap:absProducts ap∈cp.superRoles
#{o:absFactory.operationRoles|o.rolename=CreateProduct}=#absProducts
∀ap: absProducts·#{cp:conProducts|ap∈cp.superRoles}=#conFactories
{rel:absCreateProduct·rel.client}=
{o:absFactory.operationRoles|o.rolename=CreateProduct}∧
{rel: absCreateProduct · rel.supplier}
=absProducts∧isoRel(absCreateProduct)
∀{rel:CreateProduct·rel.client}=∪{o:∪{c:conFactories
·c.operationRoles|o.rolename
=CreateProduct}∧{rel:conCreateProduct·
rel.supplier}=conProducts∧isoRel(conCreateProduct)
∀c:conFactories · absProducts
=∪{rel.conCreateProduct|rel.client∈c.operationRoles
·rel.supplier.superRoles}

定义辅助函数 isoRel 检查关系中包含的角色元素的同构关系。全入射函数保证关系中角色元素一对一映射。使用这个函数形式化模式角色模型中元素之间的同构关系。

isoRel:P RelationRole→B

∀rs:P RelationRole·

isoRel(rs)⇔{r:rs· client↦r.supplier}∈{r:rs· client}⤖{r:rs· supplier}

3 基于角色的设计模式演化

设计模式最大的好处是应对“变化”。合理使用设计模式的程序，在需求或者其它东西发生变化的时候能够比较方便地进行代码重构和设计重构。设计层演化代价比实现层代价低，这部分给出如何基于角色进行设计模式的演化。

设计模式演化是指在不改变系统设计结构属性情况下，对模式角色元素的修改，其中模式角色元素包括类、属性、操作以及类之间的关系（如继承、联系等）。在模式演化中改变最多的是增加或删除类和关系。

为了自动化演化设计模式，定义 2 层演化，转换规则定义哪个建模元素改变了（如增加或删除）以及这些元素在哪里以及怎样改变的。在这里把模式演化分为两层：

- 角色层演化
- 模式层演化

3.1 角色层演化

角色层演化描述在设计模式演化过程中能够执行的基本转换。这些基本转换包括增加或删除一个角色元素，如类、操作、属性、联系、关系。这些基本转换是进行模式演化的基本步。

标识 9 个可以增加或删除的建模角色如表 1 所列。

如对上面所述的抽象工厂模式，增加一个具体产品类角色用 Object-Z 模式 AddClassRole 来表示。相同地，删除一个角色用模式 DeleteClassRole 来表示。一个模型元素替换另一个模型元素是通过首先删除一个模型元素，然后增加一个新的模型元素完成的，可先进行删除操作，然后再进行增加操作来完成。

表 1 可增加或删除的角色

角色	参数列表	描述
Class	类名	在模式中增加或删除一个“类名”的类
Attribute	属性名、类名、类型、可访问性	从“类名”中增加或删除一个名为“属性名”，类型为“类型”，可访问性为“可访问性”的属性
Operation	操作名、类名、返回值类型、可访问性、参数 1，参数 1 类型……	从“类名”中增加或删除一个名为“操作名”，类型为“类型”，可访问性为“可访问性”，参数列表为参数 1 且为参数 1 类型的操作
Association	类名 1，类名 2	在类角色名分别为类名 1 和类名 2 间增加或删除一个一般关联关系
Generalization	父类、子类	在父类和子类之间增加或删除一个继承关系
Aggregation	整体、部分	在父类和子类之间增加或删除一个聚合关系
Composition	整体、部分	在父类和子类之间增加或删除一个组合关系
Realization	目的名，源名	在父类和子类之间增加或删除一个细化关系
Dependency	目的名，源名	在父类和子类之间增加或删除一个依赖关系

AddClassRole

RoleModel
classRoles?:ClassRole

classRoles?∉classRoles
classRoles′=classRoles∪{classRoles?}
classRoles.superRoles∈superRoles

DeleteClassRole

RoleModel
classRoles?:ClassRole

classRoles?∈classRoles
classRoles′=classRoles\{classRoles?}
classRoles.superRoles∈superRoles

3.2 模式层演化

模式层演化分成 5 类，如表 2 所列。

表 2 模式层演化类型

	演化名	描述
1	Independent	增加或删除一个独立类以及这个类和原模式中类的关系
2	Packaged	增加或删除一个有属性或操作以及对应关系的独立类
3	Class group	在几个类中一致地增加或删除一个属性或操作
4	Correlated classes	一组有相互关系的类被增加或删除
5	Correlated attributes/operations	增加删除类时对应的增加或删除一组属性或操作

有相互关系的类（Correlated classes）演化：一组有相互关系的类被增加或删除。当增加或删除某个类时，其它对应的类也要相应地增加或删除。同时对应的关系也要增加，对应的属性和操作也要增加。这些对应的关系是很重要的，一旦发生错误的转换就会引起不一致性。如抽象工厂模式就属于这种情况。如果有两个具体工厂（ConcreteFactory1 和 ConcreteFactory2）和两种具体产品（AbstractProductA 和 AbstractProductB）的抽象工厂模式，每种产品有两个具体产品。一种可能的演化是增加一种新的具体产品，这就需要相应地增加一个对应的具体工厂去创建新增加的产品。这个新增加的具体工厂也应有同样的操作（createProductA 和 createProductA），如图 2 所示。

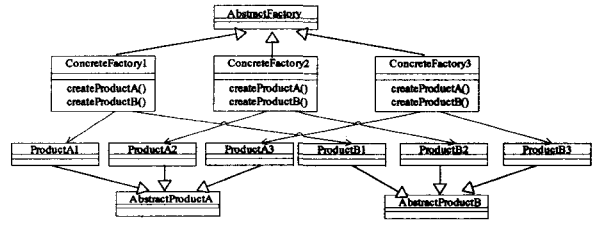


图 2 用 3 个具体工厂创建 3 种产品的抽象工厂模式

有相互关系属性或操作（Correlated attributes/operations）演化：增加删除类时对应地增加或删除一组属性或操作。仍以抽象工厂为例，当增加一种新的产品（如具有 Pro-

ductC1 和 ProductC2 的 AbstractProductC)时，需对应地在具体工厂（ConcreteFactory1 和 ConcreteFactory2）中增加 createProduct操作，如图 3 所示。

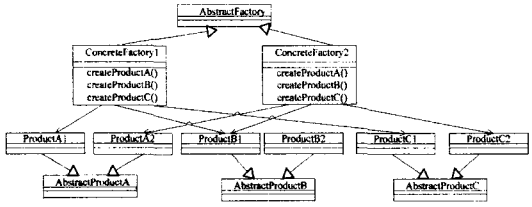


图 3　用两个具体工厂创建 3 种产品的抽象工厂模式

GOF 所述的 22 种(其中单态模式不需演化)模式演化都可由上面所述的 5 种模式层转换类型表示，如表 3 所列。

表 3　设计模式演化表

Design Pattern Name	模式层演化
Abstract Factory	4,5
Builder	4,5
Factory Method	4
Prototype	2
Singleton	——
Adapter	4,5
Bridge	2
Composite	2
Decorator	2,3
Facade	1
Flyweight	2
Proxy	4
Chain of Responsibility	2
Command	4
Interpreter	2
Iterator	4
Mediator	1
Memento	3
Observer	2,3
State	2
Strategy	2
Template Method	2,3
Visitor	2,5

3.3　模式演化过程

模式演化分为两步。首先，根据模式演化规则确定需要增加或删除的建模元素。其次，定义要在什么位置增加或删除建模元素的操作。系统演化时应该保持系统一致性和整体性。有些属性在演化后不再保持，这样需要检查演化后的系统和原系统是否保持一致。

设计者在设计中使用设计模式后，每个设计模式的演化信息允许设计者改变系统设计，但对系统的其它部分改动最小。如：Mediator 模式，设计者必须查看模式文档发现演化指导，误解或错误地改变设计模式可能会抵消使用这些模式的好处。

4　一致性验证

当系统演化时，应保持系统的一致性和完整性。一致性检查保证演化的软件有正确的性质。

因为形式规格说明不能直接被执行，故不能动态检查所写的规格说明是否一致，所以为了检查它是否一致，必须使用验证的方法。在这里使用 Object-Z 定理证明器(具体见文献[3,4])来完成演化后的设计模式的一致性验证。该定理证明器结构如图 4 所示，由 3 部分组成：Object-Z 编辑器、证明责任产生器和定理证明器 Z/EVES[5]。其中，通过 Object-Z 编辑器编辑 Object-Z 形式规格说明，进行语法分析，检查是否存在语法错误，并进行语义分析。Object-Z 编辑器编辑完毕的形式规格说明作为证明责任产生器的输入，产生相关的证明责任，并作为定理证明器 Z/EVES 的输入进行证明。

图 4　Object-Z 定理证明器结构图

下面仍然以前面用到的抽象工厂模式为例来说明设计模式一致性验证的步骤与方法。

抽象工厂模式应具有如下性质：在抽象工厂模式中每一个具体产品都会由一个具体工厂来创建。

步骤 1　对相关的性质产生证明责任。抽象工厂模式的这个性质可表示为证明责任：

PO1：Context ⊢

∀ conProduct1：conProducts • ∃ conFactory1：conFactories

• {rel：conCreateProduct • rel. client} = {o：conFactory1. operationRoles | o. rolename = CreateProduct} ∧ {rel：conCreateProduct • rel. supplier} = conProduct1 ∧ isoRel (conCreateProduct)

步骤 2　用 Z/EVES 验证证明责任。

Z/EVES 有两种风格支持定理证明：试探性(explorative)和计划性(planned)。选择计划证明来证明由设计模式性质产生的证明责任。

首先在 Z/EVES 编辑窗口编辑要证明的证明责任。证明责任编辑完毕，还要编辑证明的上下文，即前面形式化定义的抽象工厂模式的各种约束(见前面的 AbstractFactory 模式的形式化定义)。另外还要给出证明的初始化模式 InitAbstractFactory。

```
InitAbstractFactory
AbstractFactory'
absFactory'={AbstractFactory}
conFactories'={ConcreteFactory1, ConcreteFactory1}
absProducts'={AbstractProductA, AbstractProductB}
conProducts'={ProductA1, ProductA2, ProductB1,ProductB1}
absCreateProduct'={}
conCreateProduct'={createProductA, createProductB}
```

如果增加具体产品 ProductA3 和 ProductB3，而没有增加对应的具体工厂 ConcreteFactory3 去创建新产品，或者增加了对应的具体工厂，但这个新的具体工厂类没有操作 createProductA，createProductB，则采用上面的方法在 Z/EVES 中验证就会显示错误信息 false。说明演化后的抽象工厂模式与演化前的抽象工厂模式不一致。

5　相关工作介绍

有许多研究都用角色描述设计模式[6-9]，我们采用了文献[10]中对角色的定义。文献[11]提出基于角色的设计模式建模方法，该文通过对 UML 进行扩充，在实例化设计模式时，不用应用程序类替代模式中的角色，而是把角色定义成一个独立的类——角色类，使用 RoleOf 关系连接应用程序类和角色类，来实现应用程序类和角色类的分离，实现业务逻辑和模式逻辑的重用。但这种方法并没有彻底地实现应用程序类和

角色类之间的分离，因为角色类中的很多行为属性有可能是分散在多个应用程序类中的。本文中的方法使用角色建模设计模式时不仅仅只是把角色作为连接应用程序类和角色类之间关系的一个类，而是将设计模式中的各种元素都作为角色来定义，包括行为角色和属性角色以及关系角色都作为单独的类，这样将建模设计模式的元素从元层来考虑，从而实现应用程序类和角色类的彻底分离，达到设计模式不仅设计思想可以重用，代码也可以重用的目的。文献[10]没有对设计模式的演化及一致性验证问题进行探讨。文献[12]对模式的演化进行了研究，但没有给出形式化模型，也没有对演化后的一致性进行验证。也有一些对设计模式的精确建模进行研究的文献，如文献[13]。但这些研究都是从对 UML 扩充的角度来进行的。

参 考 文 献

[1] Gamma, et al. 设计模式——可复用的面向对象软件的基础[M]. 北京：机械工业出版社，2005

[2] Hannemann J, Kiczales G. Design pattern implementation in Java and AspectJ[J]. ACM SIGPLAN Notices, 2002, 37(11): 161-173

[3] 沈毅，缪淮扣，文志诚，等. 一个 Z 的证明责任产生器[J]. 上海大学学报：自然科学版，2005(11)

[4] 文志诚. 面向对象软件的形式验证技术[D]. 2006

[5] Meisels I, Saaltink M. The Z/EVES 2.0 reference manual[R]. TR-99-5493-03e. ORA Canada, Canada, 1999

[6] France R, Kim D-K, Sudipto G, et al. A UML-Based Pattern Specification Technique[J]. IEEE Transactions on Software Engineering, 2004, 30(3): 193-206

[7] Lauder A, Kent S. Precise Visual Specification of Design Patterns[C]// Proc. of ECOOP' 98, LNCS 1445. Springer-Verlag, 1998: 114-134

[8] OMG. UML2. 0superstructurespecification[OL]. http://www.omg.org/uml/

[9] Riehle D. Describing and Composing Patterns Using Role Diagrams[C]// Proc. of the Ubilab Conference' 96. 1996: 137-152

[10] Kim S K, Carrington D. Using integrated metamodeling to define OO design patterns with object-Z and UML[C]// Proc. of the 11th Asia-Pacific Software Engineering Conf. (APSEC 2004). Los Alamitos: IEEE Computer Society, 2004: 257-264

[11] 何成万，何克清. 基于角色的设计模式建模和实现方法[J]. 软件学报，2006(4)

[12] Dong J, Yang S, Zhang K. A Model Transformation Approach for Design Pattern Evolutions[C]// Proceedings of the 13th Annual IEEE International Symposium and Workshop on Engineering of Computer Based Systems. 2006: 80-92

[13] Jing Dong. UML Extensions for Design Pattern Compositions [J]. Journal of Object Technology, 1(5): 149-161. http://www.jot.fm/issues/issue_2002_11/article3

(上接第 152 页)

甚至于很大的输入空间，如何设定 α 值是提高进化搜索效率的一个重要因素。此外，当循环次数很大时，如何结合针对循环的优化方法[9]对进化搜索过程进行优化以保证测试的效率和效果；当循环体内包含多个 return 语句和循环存在嵌套时，如何改进现有方法。所有这些都会是后续研究的重点。

参 考 文 献

[1] Beizer B. Software Testing Techniques[M](2nd edition). Van Nostrand Reinhold, 1990

[2] Harman M. The automatic generation of software test data using genetic algorithms[D]. Pontyprid, Wales, Great Britain: University of Glamorgan, 1996

[3] Clark J, Dolado J J, Harman M, et al. Reformulating software engineering as a search problem[J]. IEEE Proceedings Software, 2003, 150(3): 161-175

[4] Korel B. Dynamic method for software test data generation[J]. Software Testing, Veri_cation and Reliability, 1992, 2(4): 203-213

[5] Harman M, Baresel A, Binkley D, et al. Testability Transformation-Program Transformation to Improve Testability[M]. Formal Methods and Testing, Lecture Notes in Computer Science, Springer-Verlag, 2008, 4949: 320-344

[6] Hierons R, Harman M, Fox C. Branch-coverage testability transformation for unstructured programs[J]. The Computer Journal, 2005, 48(4): 421-436

[7] Ramshaw L. Eliminating goto's while preserving program structure[J]. J. ACM, 1988, 35: 893-920

[8] Agrawal H. On Slicing Programs with Jump Statements[C]// Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation. New York, 1994

[9] Ferrante J, Ottenstein K J. The Program Dependence Graph and Its Use in Optimization[J]. ACM Transactions on Programming Languages and Systems, 1987, 9: 319-349

[10] Tracey N, Clark J, Mander K, et al. An automated frame framework for structural test-data generation[C]// Proceedings of the International Conference on Automated Software Engineering. Hawaii, USA, 1998

[11] Wegener J, Baresel A, Sthamer H. Evolutionary test environment for automatic structural testing[J]. Information and Software Technology, 2001, 43(14): 841-854

[12] Wegener J. Overview of Evolutionary Testing[C]// IEEE Seminal Workshop. Toronto, May 2001

[13] Jones B, Sthamer H, Eyres D. Automatic structural testing using genetic algorithms[J]. Software Engineering Journal, 1996, 11(5): 299-306

[14] Baresel A, Sthamer H, Schmidt M. Fitness function design to improve evolutionary structural testing[C]// Proceedings of the Genetic an Evolutionary Computation Conference (GECCO 2002). New York, USA, 2002

[15] Liu Xiyang, Lei Ning, Liu Hehui, et al. Evolutionary testing of unstructured programs in the presence of flag problems[C]// Asia-Pacific Software Engineering Conference. TaiPei, TaiWan, 2005

[16] 艾丽蓉，赵庆兰，刘西洋，等. 面向 Java 语言地进化测试中分支依赖图的构建[J]. 计算机科学，2006, 33(7): 249-252, 285