

基于耦合测试信息元数据模型的构件集成测试

马良荔 郭福亮 李永杰

(海军工程大学计算机工程系 武汉 430033)

摘 要 文中提出一个方法,由构件开发方提供有关构件内接口变量定义和使用的信息,以提高构件的可测试性和可理解性.形式化地定义了构件耦合测试准则,定义-使用属性和观察点值.在此基础上,引入包含上述两项属性的定义-使用表(DU表),给出基于该表的构件框架.最后将上述方法应用于自主开发的构件中,并生成了相应的测试用例.将文中提出的方法与 Orso 方法和 Kan 方法进行了相关的比较,结果表明文中方法无论在测试用例生成,还是在变异发现上都更有效.

关键词 构件;构件系统;构件集成测试;构件耦合测试;构件元数据

中图法分类号 TP311

Component Integration Test Based on a Metadata Model of Coupling Testing Information

MA Liang-Li GUO Fu-Liang LI Yong-Jie

(Department of Computer Engineering, Naval University of Engineering, Wuhan 430033)

Abstract This paper introduces a method to increase testability and understandability of component based on definitions and uses information of interface variables about component provided by component developers. Then formal definitions of component coupling testing criterion, definition-use attributes (DU-As) and observation-point values (OP-Vs) are given. Based on these, a definition-use table(DU-Table) is introduced, which includes DU-As and OP-Vs item. Then a framework of testable component based on above DU-Table is given. Above methods are applied to the component developed by themselves before, and related test cases are generated. Moreover above method is compared with Orso method and Kan method using same example, presenting the comparison results. The results illustrate the validity of above method, effectively generating test cases and finding more mutants.

Keywords component; component system; component-based software integration testing; component coupling testing; component metadata

1 引 言

构件测试难于实施主要源于两个方面:一是异

构性;二是缺乏相关信息.异构性是指同一规范在同一构件上有不同的实现方法,不同的构件可以用不同的程序语言实现,并用在不同的平台上.缺乏信息是指构件开发方和构件使用方之间缺乏信息,主要

是构件开发方事先不清楚所开发构件的应用环境,因此由其实施的构件测试只能假设其应用环境,会出现不一致的问题;另一方面,构件使用方在实施集成测试时,由于构件源代码未知,很难对其进行静态分析从而获得有关的数据和控制关系,构造相关的测试用例,实施测试,也很难确定实施测试所需的充分性准则。

为了解决上述问题,当前一种有效的方法就是通过由构件开发方以元数据形式提供构件的相关信息使得集成测试更易于实施. Orso 等^[1-2]引入元数据和元内容的概念,并将其应用在构件回归测试中,通过构件开发方提供的元数据和元内容来帮助构件使用方实施构件的回归测试,但是并没有讨论如何将元数据引入构件的集成测试中。

构件的一个重要属性是实现上的透明性,即具体的实现过程不可见,测试的内部信息不可见,这样就很难对构件实施白盒的集成测试,也很难分步运行软件(缺乏可控制性),很难知道执行的结果(缺乏可观察性). 鉴于此,本文提出了构件开发方根据构件源代码实施的数据流分析,抽取其中的定义和使用信息用表的形式表示,作为元数据与构件封装在一起供构件使用方使用,通过构件内方法之间的这些信息使构件使用方有效地实施集成测试。

2 构件耦合测试准则

最简单的数据流测试是由 Laski 和 Korel^[3]提出的:所有定义(all-definitions, all-defs)和所有使用(all-uses)准则. 前者是指测试必须覆盖从每个定义到至少一个使用的路径,后者是指测试必须覆盖从每个定义到所有可达使用的路径. 按照 all-defs 和 all-uses 准则,在实际测试中要生成大量的测试用例,执行时间过长,使得测试成本很高,所以很难运用在实际测试中。

Jin^[4]在软件耦合概念的基础上,提出了基于耦合的测试(Coupling-based Testing, CBT)作为集成时实施耦合测试的具体应用。

构件在集成到应用程序时,是通过构件接口与应用程序交互的,按照构件的基本特征就是可访问方法和可访问变量之间的交互,因此,基于耦合概念的测试准则必须保证集成的可访问方法之间具有访问的可能性,即保证调用方法中定义的变量可以在被调用方法中使用;从另一个角度来说,在某个方法中发现某个变量值的变化,如果在调用该方法的方法

中首次使用了该变量,则就会相应地影响该方法,类似地所有使用该变量的方法均会受到影响. 这里,首次使用不是动态需求,而是静态需求,即在某条执行路径上的首次使用很可能是另一条路径上的后续使用. 但是如果耍刻划所有这些变量的影响会使得测试成本更高,而且后续使用与首次使用相比不是很重要,所以一般在实施耦合测试都使用首次定义的概念^[5]. 本文所描述的构件测试基于方法层,因此将方法之间的调用关系以耦合形式给出. 给出如下定义。

定义 1. 假定构件 C 中方法 M_i 调用方法 M_j , 且 α 为方法 M_i 中的实参集合, $\alpha = \{x_1, x_2, \dots, x_n\}$, 映射到方法 M_j 中的形参集合 β 中, $\beta = \{y_1, y_2, \dots, y_n\}$, 其映射方法为 $y_i = \tau(x_i)$, 将方法 M_i 中调用方法 M_j 的语句称为 M_i 调用 M_j 的调用点(Call Point, CP_{ij}).

在定义 1 的基础上,给出定义 2。

定义 2. 假定对构件 C 实施构件测试, 执行测试用例集合 T , 则构件调用耦合(Component-Call-Coupling, CCC)是指 T 执行构件中所有方法调用点 $CP = \{CP_1, CP_2, \dots, CP_n\}$ 的路径集合, 可以这样来表示:

$$CCC = (CP \times CP) = \{(cp_i, cp_j) \mid \forall cp_i, cp_j \in CP\}.$$

基于定义 1 和定义 2, 给出如下定义。

定义 3. 假定对构件 C 实施构件测试, 执行测试用例集合 T , 则构件所有耦合定义(Component All-Coupling-Defs, CACD)是指 T 至少执行一条从集合 α 中的每个实参 x_i 的每个最后定义>Last-def)到集合 β 中形参 y_i 的一个或一个以上首次使用的耦合路径的集合, 可以这样表示:

假定 $D_L(V)$ 为变量 V 的最后定义集合, $U_F(V)$ 为变量 V 的首次使用集合, 则

$$\begin{aligned} CACD(V) &= (D_L(V) \times U_F(V)) \\ &= \{(d, u) \mid \forall d \in D_L(V), \exists u \in U_F(V)\}. \end{aligned}$$

定义 4. 假定对构件 C 实施构件测试, 执行测试用例集合 T , 则构件所有耦合使用(Component All-Coupling-Uses, CACU)是指 T 至少执行一条从集合 α 中的每个实参 x_i 的每个最后定义>Last-def)到集合 β 中形参 y_i 的每个首次使用的耦合路径的集合, 可以这样表示:

假定 $D_L(V)$ 为变量 V 的最后定义集合, $U_F(V)$ 为变量 V 的首次使用集合, 则

$$\begin{aligned} CACU(V) &= (D_L(V) \times U_F(V)) \\ &= \{(d, u) \mid \forall d \in D_L(V), \forall u \in U_F(V)\}. \end{aligned}$$

定义 5. 假定对构件 C 实施构件测试, 执行测试用例集合 T , 则构件所有耦合路径 (Component All-Coupling-Paths, CACP) 是指 T 执行从集合 α 中的每个实参 x_i 的每个定义到所有可达使用的所有耦合路径的集合, 可以这样来表示:

假定 $D(V)$ 为变量 V 的定义集合, $U(V)$ 为变量 V 的使用集合, 则

$$CACP(V) = \{(d, u) \mid \forall d \in D(V), \forall u \in U(V)\}.$$

3 构件方法耦合关系的图表示

在上述定义的基础上, 本文引入构件方法耦合关系图的概念来具体说明构件中的方法之间的耦合关系。

定义 6. 构件方法耦合关系图 (Method Coupling Relation Graph of a Component, MCRG) 是一个带标记的有向图, $MCRG = (M, E, A)$, 其中:

M 为有向图的顶点集合, 为构件内的方法集合, 方法名在顶点上方标明;

E 为有向边的集合, 每个边连接 M 中的两个顶点, 表示两个顶点之间有耦合关系, $E = \{\langle M_i, M_j \rangle \mid M_i, M_j \in M\}$;

A 为顶点上的注释集合, 即标明方法之间发生耦合关系的变量名称以及具体的耦合类型, 即定义 (definition) 关系和使用 (use) 关系, 分别用 $V(d; x)$ 或 $V(u; x)$ 分别表示变量 V 的定义或使用关系, x 表示变量的值。

例如: 假定有如图 1 所示的构件 A 。

```

Class A{
    private int a, b;
    ...
    public B(int x)
    {
        ...
        private C(int y)
        {
            ...
            return y;
        }
        private D(int n)
        {
            ...
            return n;
        }
        ...
        public E(int i)
        {
            ...
            B(i);
        }
    }
}

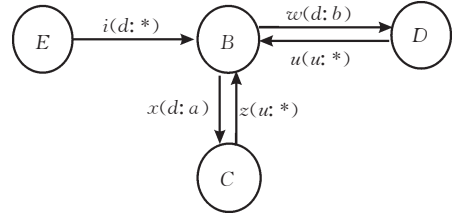
```

图 1 构件 A 的伪代码表示

图 1 中构件 A 相应的构件方法耦合图见图 2 所示。

MCRG 具有如下性质。

性质 1. 假设在方法 A 中调用方法 B , 则方法



*表示值不确定, 根据具体执行结果来确定。

图 2 构件 A 的方法耦合关系图

A 和方法 B 有耦合关系, A 和 B 之间传递的参数为耦合变量, 在 MCRG 上画节点 A 和节点 B , 且有边从节点 A 指向节点 B , 边上标记相关的参数名及其定义耦合, 标明相关的值;

性质 2. 假设方法 A 执行后返回方法 B , 则从节点 A 画一条边指向节点 B , 边上标记返回参数名及其使用耦合, 标明相关的值;

性质 3. 假设方法 A 调用多个方法, 则方法 A 有多个射出边;

性质 4. 假设方法 A 调用方法 B , 且为无参调用, 则节点 A 到节点 B 的边上无标记;

性质 5. 假设方法 A 调用方法 B , 且方法 B 有返回值, 则节点 A 和节点 B 之间有为双向边;

性质 6. 假定方法 A 调用方法 B , 方法 B 调用方法 C , 则方法 A 和方法 C 之间的耦合关系通过有向边之间的传递来实现。通过图中边的深度来度量耦合关系的数量。

4 基于耦合测试信息的元数据模型构造

4.1 定义和使用信息的表示

通过上述定义和分析, 发现给出构件的定义和使用信息对于构件使用方实施集成测试很方便, 这里将构件内部方法变量之间的定义和使用信息以元数据的形式提供给构件使用方使用, 以方便构件集成到应用程序中实施的集成测试。

本文仅仅考虑构件中每个方法的接口变量, 对于每个方法, 收集每个接口变量的定义和使用信息, 接口变量可以是可访问变量、返回变量 (其值返回给调用构件的应用程序) 以及参数 (包括实际参数和形式参数), 这些信息由构件开发方收集, 与构件封装在一起, 以元数据形式提供给构件使用方, 将这些信息以表的形式给出, 称为 DU 表, 如表 1 所示。

表 1 中, 方法名是指被测构件中包含的方法名, 变量名则是指上述方法的变量名。DU 属性给出相关变量的信息, 这里给出定义 7 (在定义语句中给出某个变量的值)。

表 1 DU 表的内容

方法名	接口变量名	DU 属性(DU-A)	首次使用	最后定义	观察点的值
方法名 1	变量名 1	标明变量名 1 定义和使用信息的元组的有限集合;	标明该变量首次使用的语句号;	标明该变量最后定义的语句号;	为该变量观察点的三元组集合,观察点的定义见定义 8;

	变量名 n	标明变量名 n 定义和使用信息的元组的有限集合;			...
...
方法名 n	变量名 1

	变量名 n

定义 7. 变量 V 的 DU 属性(简称 DU-A)是一个二元组的有限集合,表示为 $DU-A(V)=(\text{定义或使用属性,语句位置})$,其中:

- (1)定义或使用属性是变量 V 的定义或使用集合;
- (2)语句位置是指上述定义或使用在源代码中的编号.

根据上述定义,给出 DU-A 表示的相关性质.

性质 7. 假定语句是分支语句,诸如

```
l1: if 条件 A
    then { ...;
lm:   变量 b=表达式 1;
      ...}
    else { ...;
lx:   变量 b=表达式 2;
      ...}
```

则此时关于变量 B 的最后定义必须在代码运行时才能确认,因此变量 B 的 DU-A 表示可引入“ \vee ”符号来表示分支语句之间的可选择性,表示形式为 $DU-A=\{\cdots,(d,lm)\vee(d,lx)\}$.

性质 8. 假定语句是循环语句,诸如

```
while {A} 或 for( A) 或 do while
{...
ly: B=表达式 1;
...}
```

则此时关于变量 B 的最后定义取决于循环条件和循环次数,在代码运行时才能确认,这是忽略其中的

循环次数,将变量 B 的 DU-A 表示为 $DU-A=\{\cdots,(d,ly)\}$.

性质 9. 假定语句是顺序语句,则 DU-A 的计算按照语句出现的先后判别是定义还是使用属性.

图 3 给出了变量 x 的收集信息,这里 x 为可访问变量,DU-A 项的具体内容见图 3.

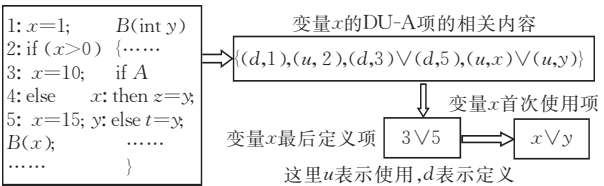


图 3 变量 X 的 DU-A 项以及最后定义项、首次使用项的表示

定义 8. 观察点(Observation Points, OPs)是指设定在构件内部,反映构件内基本数据流的语句,可以设定在构件内可访问变量和实际参数的最后定义所在的语句处、返回变量和形式参数的首次使用所在的语句处.

定义 9. 变量 V 的观察点值(OP-Vs)是一个三元组,表示为 $OP-Vs=(\text{语句位置,最后定义语句之前变量 } V \text{ 的值,最后定义语句之后变量 } V \text{ 的值})$,其中:

- (1)语句位置在定义 7 中给出;
- (2)最后定义语句之前和最后定义语句之后变量 V 的值是指如果在 DU 表中的项标明是定义属性,且为最后定义,则分别计算在最后定义语句执行前后变量 V 的值.

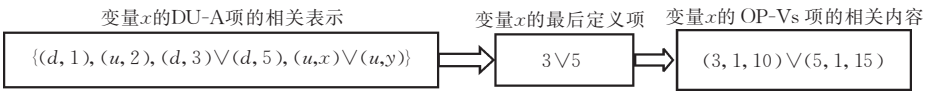


图 4 变量 x 的 OP-Vs 项的表示

OP-Vs 项仅仅从源代码中反映了接口变量的静态值,另外一些值在运行时才能知道,例如,对图

3 中的源代码进行修改,变量 x 相应的 DU-A 和 OP-Vs 项,如图 5 所示.

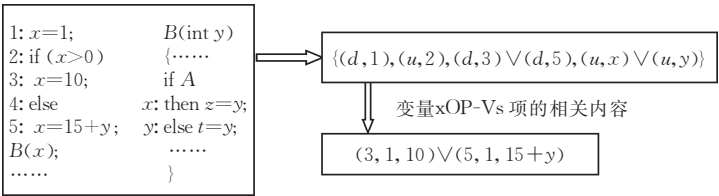


图 5 修改后代码中的变量 x 的 DU-A 和 OP-Vs 项的表示

这样,如果仅仅应用上述 OP-Vs 项还不能获得给定观察点(observation-points)的值.这里使用基于监控机制的观察点方法来获得相关的值,具体内容在下节中描述.

4.2 基于观察点的监控机制

上面章节中,提出了使用监控机制以提高构件内部数据的可观察性.为此引入了可观察点的概念,并在 DU 表中列出了可观察点的信息.一般来说,要提高构件的可观察性需要增加观察能力,主要有 3 种方法,分别是^[6-7]:① 基于框架的跟踪;② 自动的代码插入;③ 自动的构件包装.

本文在对 3 种方法进行充分比较的基础上,综合上述 3 种方法,形成观察点代码插入方法来实现监控机制.基本方法就是将观察代码插入到构件中,即将观察操作代码插入到每个观察点中.使用基于 Java 语言的事件模型实现了上述监控机制,即和构件封装在一起形成一个可监控的构件.监控构件的错误,其中包括状态监控和错误监控.为了设计上的简单性,没有涉及到事件监控、性能监控、操作监控

等内容,如图 6 所示.

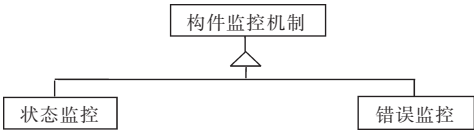


图 6 基于观察点的监控机制监控内容

① 状态监控:监控给定观察点上数据的状态和值;

② 错误监控:记录构件中的错误信息.

设计了两种方法 *stateObserve(attribute:String[])*和 *errorObserve(exception:String[])*作为嵌入式方法嵌入到构件中以帮助实现观察,限于篇幅,代码略去.

4.3 基于 DU 表的构件框架

将上述引入耦合测试信息的元数据与构件封装在一起,供构件使用方使用,帮助其在实施集成测试时有效地发现构件中的异常,这里引入异常检测机制来有效地检测错误,并引入监控机制提高构件可观察性,该框架如图 7 所示.

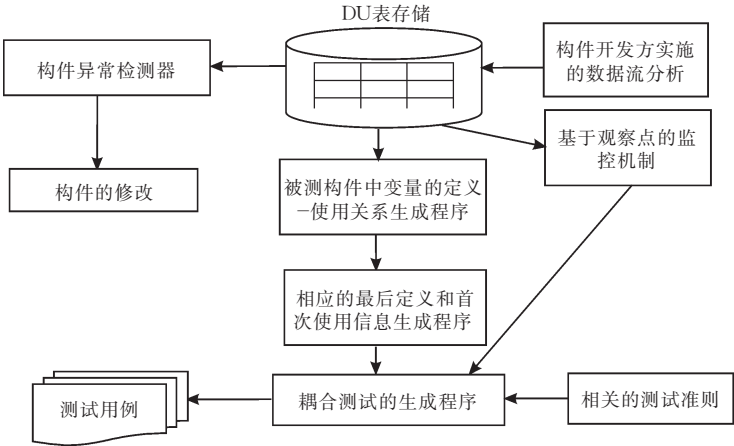


图 7 基于 DU 表的构件框架

图 7 中,构件开发方通过对构件实施数据分析,得到相应的 DU 表,将 DU 表嵌入元数据.通过构造 DU 表,就可以使构件使用方明确构件内部的数据流,在异常机制检测程序的帮助下较容易地观察到测试过程中变量的输出,进而确定故障.为了提高可观察性,这里提供了基于观察点的监控机制来监控

被测构件中变量的内部变化.

5 实例研究

在上述研究的基础上,将构造的模型应用到基于构件开发技术的教学管理系统 TeachMana 中所

实现的一个学生成绩登记构件 RegisterStuGrade 中,其中定义了 4 个类,分别是 RecordGrade 类、StudentInformation 类、CourseManagement 类和 TeacherInformation 类,有 36 个方法(有 12 个可访问方法)。

本文将 RecordGrade 类作为分析对象来进行分析。类 RecordGrade 包括 5 个公共方法,分别为 AppendGrade, ModifyGrade, DeleteGrade, DetailedGrade 以及 QueryGrade。另外还包括 6 个私有方法,分别为 AvailbleGrade, InitGrade, StatGrade, GraphicGrade, AnalyGrade 以及 ReturnGrade。为了更加客观地验证上述方法的可行性,这里假设 RecordGrade 类的源代码对于构件使用方来说不可知,仅仅依靠以构件元数据给出的构件内有关耦合测试的定义和使用信息来实施集成测试。元数据给

出的信息内容如表 1,依据前面给出的耦合测试准则所有耦合定义(CACD)生成 13 个测试用例,依据所有耦合使用(CACU)生成 20 个测试用例。

从上述测试中选择了相关的方法调用序列,形成测试序列,用以反映多个测试用例的执行,便于进行相关的分析。例如,执行测试用例 AppendGrade() 则会有测试序列 [InitGrade(), AppendGrade(), AvailbleGrade()],即调用该序列中的方法;执行测试用例 QueryGrade 则可能形成测试序列 [QueryGrade(), DetailedGrade(), ReturnGrade()]. 这样,在前面生成的测试用例基础上,根据 CACD 所生成的 13 个测试用例生成了 6 个测试序列,根据 CACU 所生成的 20 个测试用例生成了 10 个测试序列,如表 2 所示。

表 2 类 RecordGrade 的测试序列

CACU	1. InitGrade(), AppendGrade(), AvailbleGrade(), ReturnGrade()	CACD
	2. InitGrade(), AppendGrade(), AvailbleGrade(), ModifyGrade(), AvailbleGrade(), ReturnGrade()	
	3. QueryGrade(), ModifyGrade(), AvailbleGrade(), ReturnGrade()	
	4. QueryGrade(), DetailedGrade(), GraphicGrade(), AnalyGrade(), ReturnGrade()	
	5. QueryGrade(), DeleteGrade(), ReturnGrade()	
	6. QueryGrade(), DetailedGrade(), ReturnGrade()	
	7. QueryGrade(), ReturnGrade()	
	8. QueryGrade(), DetailedGrade(), GraphicGrade(), ReturnGrade()	
	9. QueryGrade(), DetailedGrade(), AnalyGrade(), ReturnGrade()	
	10. QueryGrade(), DetailedGrade(), ModifyGrade(), AvailbleGrade(), ReturnGrade()	

评估软件测试是否有效的一个重要方面是验证该测试方法发现软件中故障的能力。因此,按照 Delamaro^[8] 给出的变异实施方法,故意对类 RecordGrade 随机实施了 48 处变异,根据类 RecordGrade 功能随机生成了 43 个测试序列,并按照类 RecordGrade 对于不同输入所进行的不同处理将这些测试序列分成 6 组,分别为第 1 组,第 2 组, ..., 第 6 组。其中第 1 组包含 5 个测试序列,第 2 组包含 3 个测试序列,第 3 组包含 10 个测试序列,第 4 组包含 9 个测试序列,第 5 组和第 6 组包含 8 个测试序列。从上述测试序列发现这些变异的概率来证明本方法的有效性,如表 3 所示。

表 3 RecordGrade 类的测试结果

测试准则	测试用例数目	测试序列数目	发现的变异数目	发现变异数的概率/%
CACD	13	6	45	93.8
CACU	20	10	47	97.9
第 1 组	—	5	21	43.8
第 2 组	—	3	13	27.1
第 3 组	—	10	44	91.7
第 4 组	—	9	42	87.5
第 5 组	—	8	35	72.9
第 6 组	—	8	38	79.2

接着,对本文中的方法与 Orso 方法以及 Kan-somkeat 方法(简称为 Kan 方法,文献[6])做比较。为增加可比较性,使用了 Orso 方法中给出的例子(具体见文献[1-2]),并对构件 Dispenser 进行了 9 处修改,由于 Dispenser 仅包含一个可访问方法 dispense,因此这 9 处修改就在方法 dispense 内部完成。这些变异如表 4 所示,由于假设源代码未知,因此这些变异主要表现在与其它方法交互的接口变量上。

表 4 对构件 Dispenser 实施的变异

方法名	变量名	变异方法	变异数	语句位置
dispense	credit	初值改变、计算方法改变、参数数值改变	3	45,53,54
	sel	初值改变、计算方法改变、参数数值改变	2	47,49
available	sel	由于前面 dispense 中的 sel 修改使得这里的 sel 发生变异	2	60,62
	val	初值改变、计算方法改变	2	44,52

将修改后的构件 Dispenser 记为 Dispenser',使用本文的方法、Orso 方法和 Kan 方法来发现这些变异。本文的方法发现了全部 9 个变异,Orso 方法

仅发现了 7 个变异,而 Kan 方法发现了 8 个变异,如图 8 所示.

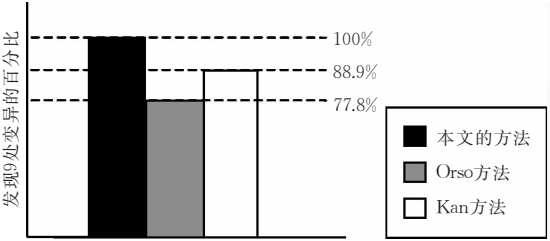


图 8 3 种方法发现的构件 Dispenser'中的变异结果

图 8 中,百分比的计算如下:

$$X\% = \frac{FMN}{MN} \times 100\% ,$$

其中 FMN 表示发现的变异数,MN 表示所有的变异数.

6 结 论

本文在概述数据流分析、形式化描述构件耦合测试准则的基础上,引入了构件方法耦合关系图的概念用以描述构件中方法之间的耦合关系.在此基础上,构造包含构件内方法变量的定义、使用信息以及最后定义、首次使用信息的 DU 表,将这些信息以元数据方式与构件一起提供给构件使用方以方便其实施集成测试,并引入可基于观察点的监控机制概念来实现构件内的可观察性,给出了基于构件的最后定义和首次使用信息的总体实现框架.在内部开发的构件 RegisterStuGrade 集成到应用程序 TeachMana 中时,实施的集成测试中使用了该方法,并将该方法与 Orso 方法、Kan 方法运用在构件 Dis-

penser 上进行了对比以体现其优越性.

参 考 文 献

[1] Orso A, Harrold M J et al. Using Component metacontent to support the regression testing of component-based software// Proceedings of the IEEE International Conference on Software Maintenance (ICSM'01). Florence, Italy, 2001: 716-725

[2] Orso A et al. Using component metadata to support the regression testing of component-based software. Technical Report GIT-CC-01-38, College of Computing, Georgia Institute of Technology, 2001: 154-163

[3] Laski J, Korel B. A data flow oriented program testing strategy. IEEE Transactions on Software Engineering, 1983, 9(3): 347-354

[4] Jin Z, Offutt A J. Coupling-based criteria for integration testing. The Journal of Software Testing, Verification and Reliability, 1998, 8(3): 133-154

[5] Offutt A J, Abdurazik A, Alexander R T. An analysis for coupling-based integration testing//Proceedings of the 6th IEEE International Conference on Engineering of Complex Computer System (ICECCS'00). Tokyo Japan, 2000: 172-178

[6] Kansomkeat S, Offutt J, Rivepiboon W. Increasing class-component testability//Proceedings of the 23rd IASTED International Multi-Conference Software Engineering. Innsbruck, Auatria, 2005: 156- 161

[7] Gao J, Zhu E Y et al. Monitoring software components and component-based software//Proceedings of the Computer Software and Applications Conference (COMPSAC). IEEE Computer Society Press, 2000: 123-129

[8] Delamaro M E, Maldonado J C et al. Interface mutation: An approach for integration testing. IEEE Transactions on Software Engineering, 2001, 27(3): 228-247



MA Liang-Li, born in 1968, Ph. D., associate professor. Her research interests include software testing and software quality assurance techniques.

GUO Fu-Liang, born in 1963, Ph. D., professor. His research interests include data mining and software reliability techniques.

LI Yong-Jie, born in 1977, master, lecturer. His research interests focus on database techniques.

Background

In recently years, component-based software technology has been increasingly considered as necessary for the vastly more complex software. The main motivation is the possibility to significantly reduce development costs and time yet sat-

isfying quality requirements. But in many cases, the drawbacks of component-based software technologies arise because of the lack of information about provided components. So software development organizations are facing unrivaled chal-

lenges. The simultaneous use of many software package developed in-house as well as (often customized) off-the-shelf software has further intensified the existing integration problems of application systems.

So how to assure component-based software systems (CBS) quality is made more challenging. It is well known that component-based software testing is one of the most effective techniques assure the quality of component-based systems. Generally, testing of COTS software is black-box because users do not have access to the source code to analyze the internal implementation. How to improve the poor testability of components is one of the issues and challenges techniques that use component testing.

To increase the testability of component, it is essential for component users to know the interior details about the component under test. Several years ago Orso presented an idea using metadata describing information about component to solve the problem. Component developers provided the de-

tail of component without revealing source code. Existing component standards, including DCOM, Enterprise JavaBeans and Microsoft .NET already supplied some additional information about a component through the use of metadata that are packaged with the component. But the metadata available so far are typically limited to information that is useful for compile-time and run-time type-checking and for design-time customization There are some related researches addressing to increase the testability of component.

In this paper, the authors propose a new analysis method based on some definition and use information extracted from source code by component developers. The information is represented in the form of table, which is encapsulated with the component as component metadata. And the information is used to increase component testability. Based on analyzing the testability of component, a method how to construct a testable component, including related data flow analysis is given.