# 基于变更模型的元数据在构件回归测试中的应用研究

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**　本文针对 Orso 元数据方法的不足，对构件可能的变更进行了充分分析和分类，并按照分类情况对相关的构件变更处理方法进行了详尽的形式化描述，构造了构件变更模型，描述了每个变更的具体表现形式，论述了方法变更到构件接口变更的映射机制和算法，给出了算法实现的框架，并将这些理论模型应用到自主开发的构件 RegisterStuGrade 中，与没有元数据情况下的回归测试用例生成技术以及 Orso 方法进行回归测试用例数和回归测试运行时间两方面的分析和对比，给出了比较结果。

**关键词**　构件，构件集成测试，构件回归测试，构件元数据，构件变更

## Application Research of Metadata Based on Change Model to Component Regression Testing

MA Liang-li　GUO Fu-liang　LI Yong-jie

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

**Abstract**　Addressed to shortages of metadata method presented by Orso, possible changes on component are analyzed and classified, and related processing methods of these changes are described formally in detail according to these classifications. Further change model of component is constructed. Then particular representation of each change is described. The mapping mechanism and algorithm from method changes mapping to interface changes is presented, and the related algorithm is given. Lastly these research models are applied to component RegisterStuGrade developed by ourselves, and the method is compared with no-metadata and Orso method. The comparison includes numbers of regression testing cases and running time. The relevant results are presented.

**Keywords**　Component, Component-based software integration testing, Component-based software regression testing, Component metadata, Component change.

## 1　引言

构件回归测试是指对构件的可选择性测试，以验证对构件实施变更所造成的影响以及修改后的构件是否仍然与其规范需求一致。对回归测试的研究主要集中在开发出不同的回归测试选择过程，使得回归测试所耗费的时间和资源最小。传统的回归测试主要依赖于源代码，所以当对构件实施回归测试时，由于构件使用方不能访问源代码，只能访问二进制可执行文件和相关文档，故提高回归测试效率成为构件测试中一个很重要的研究内容。其主要方法是无需运行全部测试用例，只运行那些受构件变更影响的测试用例，而缺乏源代码对减少和选择测试用例提出了挑战[1]。

目前 Orso 等[2,3]引入元数据和元内容的概念，应用在构件回归测试中。在具体实现中运用了两种方法：一种是基于源代码的方法，当源代码发生变更时，将变更的语句存储在元数据中，且对构件中的每一条边用插装方法记录测试用例的执行路径，这种方法是以源代码已知为前提条件的；另一种是基于规范的方法，通过由构件开发方提供与语句修改对应的测试帧的信息，使得实施回归测试时只运行受变更影响的测试帧所对应的测试用例，而无需运行所有测试用例，从而改善回归测试的效率。文中对两种方法进行了相关的实例研究，但是该方法有一定的局限性，表现在：①缺乏一定的形式化基础；②对构件中每条边都进行了插装，以记录用例的执行轨道，相对来说成本较高；③为了实施回归测试所进行的变更相对简单，只是在构件一处进行变更，实施回归测试时只运行那些导致失效的测试用例，这对于比较复杂的变更情况不适用；④文中第一种方法建立在应用程序源代码对实施回归测试的人员来说可知的基础上，缺乏通用性，而第二种方法建立在测试帧基础上，对于复杂的应用程序显然不适用。

本文在对构件变更进行充分分析的基础上，给出了描述构件变更的元数据在回归测试中的形式化描述，阐述了其在回归测试技术上的应用，通过选择可能的测试用例子集来减少回归测试成本，强调已经发生变更或可能受变更影响的构件功能，即可以这样描述：

假定一个应用程序 A 使用了一个外部构件 C，且构件 C 已经由一组测试程序 T 实施了相关测试，现有构件 C 的一个新版本 C′，如何在构件开发方提供的描述变更的元数据基础上构造出一组新的测试程序 $T' \subseteq T$，以暴露 A 由于构件 C 的变更而驱动的回归测试缺陷，而无需重新运行那些不受变更影响的测试用例。

## 2　构件变更分析与变更模型建立

由于构件发生变更，使用该构件的构件使用方就要实施回归测试。因此，要研究基于描述变更的元数据的回归测试，首先必须对构件的变更情况进行分析，对每类变更的处理方法进行描述，建立构件变更模型。这样，首先要对构件进行定义。

**定义 1**　构件是一个三元组 $C=(D,M,I)$，其中：

$D=\{d_i \mid d_i \in (D_{pub} \cup D_{pri})\}$ 表示数据集，这里 $D_{pub}$ 表示公共成员变量数据集合，即外部可访问的变量集合；$D_{pri}$ 表示私有成员变量集合方法，即外部不能访问的变量集合，只在内部使用。

$M=\{m_i \mid m_i \in (M_{pub} \cup M_{pri})\}$ 表示方法集，这里 $M_{pub}$ 表示公共方法集合，即外部可调用的方法集合；$M_{pri}$ 表示私有方法集合，即外部不能调用的方法集合，只在内部执行。

*）本文得到湖北省自然科学基金资助(2005ABA266)。**马良荔**　博士，副教授，研究方向为软件可靠性、软件测试。

$I=\{i_i \mid i_i \in (D_{pub} \cup M_{pub})\}$表示接口集，构件通过这些接口与外部交互，即为前面定义的 $D_{pub}$ 和 $M_{pub}$ 的并集。

### 2.1 构件变更分析

本文构造的构件变更模型包含两方面内容：一是变更类型(Change Type，记为 $C_t$)，二是变更描述(Change Description，记为 $C_d$)。

本文假设构件源代码对于集成构件的构件使用方是未知的，因此从构件使用方角度，无需知道构件内部具体的变更细节，构件使用方仅仅通过构件的接口与构件交互。因此，将对构件实施的所有变更通过相应的映射机制反映到构件的接口上，并且通过构件开发方提供描述变更的元数据，使得构件使用方能更好地实施回归测试，使得回归测试的测试用例集合最小[4]。从构件使用方角度，对所有的构件变更进行分类，其分类情况以及相关的处理方法如表 1 所示。

表 1 构件变更类型分类及其变更处理方法

| 变更类型 | 变更描述 | 相关的处理方法 |
|---|---|---|
| 1 | 构件(可访问变量或可访问方法)名字的修改； | 以构件(可访问变量或可访问方法)新名替代旧名； |
| 2 | 可访问方法的修改 | 通过相应的映射机制将所有这些变更映射到可访问方法和可访问变量的变更。 |

第 1 类变更是最简单的变更，只是修改构件(可访问变量或可访问方法)的名字即可。第 2 类变更由于构件本身的复杂性，分为许多情况。由于从构件使用方角度来看，所有这些变更都体现在可访问变量和可访问方法的变更，其变更的具体内容不必关心。

通过由构件开发方向构件使用方提供描述上述变更的元数据来实施有效的回归测试，即对构件实施上述变更后会使得调用该构件的应用程序的相关部分受到影响，对这些受到影响的测试用例要实施再测试，通过描述变更的元数据的使用使得所确定的受到影响的测试用例集合尽可能小。

### 2.2 构件变更模型建立

根据表 1，给出构件变更模型的定义。

**定义 2** 构件变更模型(Component Change Model，记为 CM)可描述为一个二元组：

$CC=(C_t, C_d)$，其中 $C_t$ 表示变更类型，$C_d$ 表示变更描述；

对于每个 $C_t$，有 $C_t \in \{C_1, C_2\}$，这里 $C_1$、$C_2$ 都表示构件变更类，且 $C_1 \cap C_2 = \Phi$，其中：

$C_1$ 表示构件名、可访问变量名或可访问方法名的修改操作集合；$C_2$ 表示构件内可访问方法的修改操作集合。

对于上述不同的变更类型，对应的 $m_d$ 按照下述方法给出：

(1)当 $C_t$ 为 $C_1$ 时，按照上述变更种类进一步标识为 $C_{11}$、$C_{12}$ 和 $C_{13}$ 分别表示构件名、可访问变量名和可访问方法名的修改，其 $C_d$ 表示为一个二元组 $C_{d1}=(N_{old}, N_{new})$，$N_{old}$ 表示构件(或可访问变量和可访问方法)的原有名字，$N_{new}$ 表示构件(或可访问变量和可访问方法)的现有名字；

(2)当 $C_t$ 为 $C_2$ 时，$C_d$ 表示为一个二元组 $C_{d2}=(d, ptr)$，其中 $d$ 表示说明具体变更，$ptr$ 指向构件描述性元数据的接口属性。

**定义 3** 基于变更模型的元数据(Metadata based on Change Model，记为 M-CM)为描述构件中所有变更集合的元数据：

$M\text{-}CM=\{(C_t, C_d) \mid C_t \in \{C_1, C_2\}$ 且 $C_d \in \{C_{d1}, C_{d2}\}\}$

$C_1$，$C_2$，$C_{d1}$，$C_{d2}$ 的具体含义满足定义 2。

## 3 构件内方法变更到构件接口变更的映射机制和算法描述

要将表 1 列出的变更反映在构件可访问变量和可访问方法上，就需要相应的映射机制，可以定义该映射机制。

**定义 4** 构件变更映射机制为 $f:\Delta x \rightarrow \Delta y$，其中：

$\Delta x$ 表示构件内变更，$\Delta y$ 表示构件接口的变更。

要形式化其中的映射关系，进一步引入接口－映射图的概念。

**定义 5** 变更－接口映射图(Change-Interface Mapping Graph，记为 C-IMG)为一个两层有向图 $IMG=\langle V_1, E, V_2\rangle$，其中 $V_1$ 表示构件内所有变更集合，即定义 2 中定义的所有变更模型描述；$V_2$ 表示接口顶点集，$V=M_{pub} \cup V_{pub}$，$M_{pub}$ 为可访问方法集，$V_{pub}$ 为可访问变量集；$E=\{\langle V_i, V_j\rangle \mid (V_i \in V_1$ 且 $V_j \in V_2)\}$表示构件内任一变更到任一接口之间的依赖关系。

**定义 6** 构件内任一变更到任一接口之间的依赖关系(Change-Interface Dependency Relationship)可表示为 C-ID($c_i$, con, $i_j$)，即在条件 con 为真时，变更 $c_i$ 对接口 $i_j$ 产生影响。

上述形式化定义的目的是为了实施回归测试，将构件内部的变更映射到构件接口的变更。为此要描述构件内部的某个变更对接口所产生的影响。

要实现上述映射，先要弄清构件内各方法之间的关系。这样当某个方法内出现变更时，就可以找出其所影响的方法[6]。这样，本文引入方法依赖图的概念。

**定义 7** 方法依赖图(Method Dependency Graph，记为 MDG)为一个有向图 $MDG=\langle V, E\rangle$，其中 $V$ 表示方法顶点集，$V=M_{pub} \cup M_{pri}$，$M_{pub}$ 为可访问方法集，$M_{pri}$ 为私有方法集；$E=\{\langle V_i, V_j\rangle \mid (V_i, V_j \in V)\}$表示方法 $i$ 和方法 $j$ 的依赖关系。

**定义 8** 构件内方法之间的依赖关系包括方法变量依赖(Method Varible Dependency，记为 MVD)和方法控制依赖(Method Control Dependency，记为 MCD)。

1. 方法变量依赖可表示为 MVD($m_i$, $v_i$, $m_j$)，其中包含两种情况：

(1)是方法 $m_i$ 中定义变量 $v$，在方法 $m_j$ 中引用，这种变量依赖称为直接变量依赖(Directed Method Varible Dependency，记为 DMVD)，表示为 DMVD($m_i$, $v_i$, $m_j$)；

(2)是方法 $m_i$ 和方法 $m_j$ 中都引用变量 $v_j$，这种变量依赖称为间接变量依赖(Indirected Method Varible Dependency，记为 IMVD)，表示为 IMVD($m_i$, $v_j$, $m_j$)。

2. 方法控制依赖表示为 MCD($m_i$, $c$, $m_j$)，即当条件 $c$ 为真时，方法 $m_i$ 中执行方法 $m_j$。

为了与已有方法进行更好的比较，本文也引入 Orso 等在文献[2,3]中所引用的程序。该程序的源代码如图 1 所示。

下面就对图 1 中所引入的构件 Dispenser 来进行分析。构件 Dispenser 包含一个可访问方法 dispense 和私有方法 available，应用程序 VendingMachine 中通过调用语句 expense =d. dispense(currValue, selection) 实现对构件 Dispenser 中可访问方法 dispense 的访问。当发现构件 Dispenser 中的错误并进行了相关修改，即将语句 val=0 插入语句号 16 之后，

将这些修改信息加入元数据中，使得使用该构件的构件使用方能明确构件 Dispenser 的修改所影响的测试用例，并重新运行这些测试用例，来验证修改后语句的正确性。图 1 中构件 Disenper 内的方法依赖图如图 2 所示。

```
应用程序VendingMachine（构件使用方）：
public class VendingMachine {
 final private int COIN=25;
 final private int VALUE=50;
 private int totValue;
 private int currValue;
 private Dispenser d;
 public VendingMachine() {
 totValue=0;
 currValue=0;
 d=new Dispenser(); }
public void insert() {
 currValue+=COIN;
 System.out.println( "Curr value="+currValue); }
public void return() {
 if(currValue==0)
  System.err.println("no coins to return");
 else{
  System.out.println("Take your conis");
  currValue=0; }
 public void vend(int selection) {
  int expense;
  expense= d.dispense(currValue, selection);//调用
  totValue+=expense;
  currValue-=expense;
  System.out.println("Current value="+cuuValue);
} //class VendingMachine
```

```
构件Dispenser（构件开发方）：
1.public class Dispenser{
2. final private int MAXSEL=20;
3. final private int VAL=50;
4. private int[] availSelectionVals={2,3,13};
5. public int dispense(int credit, int sel) {
6.  int val=0;
7.  if (credit==0)
8.   System.err.println("No coins inserted");
9.  else if (sel>MAXSEL)
10.  System.err.println("Wrong selection"+sel);
11. else if (!available(sel))
12.  System.err.println("Selection"+sel+"unavailable");
13. else {
14.  val=VAL;
15.  if (credit<val)
16.   System.err.println("Enter"+(val-credit)+"conis");}
17.  else
18.   System.err.println("Takeselections");}
19. return val;
20. }
21. private boolean available(int sel) {
22.  for(int i=0; i<availSectionVals.length; i++)
23.   if(availableSelectionVals[i]==sel) return true;
24. return false;
25. }
26. }//class Dispenser
```

val=0;

图 1　Orso 所引用的源程序

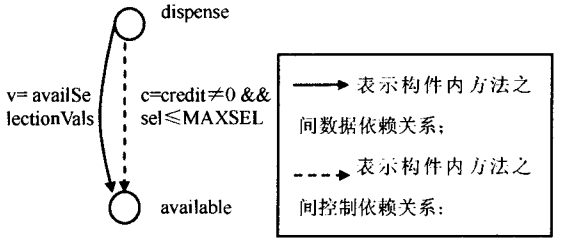


图 2　构件 Dispenser 内的方法依赖图

由图 2 所示，如果应用程序 VendingMachine 对构件 Disenper 实施方法级的回归测试，则修改方法 dispense 后只有修改变量 availSelectionVals 或在 credit≠0 && sel≤MAXSEL 条件为真时，才会影响到方法 available。而修改方法 available 不会对方法 dispense 造成影响。显然，当在构件 Dispenser 的方法 dispense 中第 16 条语句之后加入语句 val=0 后，不会对方法 available 造成影响。

按照定义 4 和定义 5 对构件内的该处变更到接口的映射，给出该变更的变更-接口映射图，如图 3 所示。

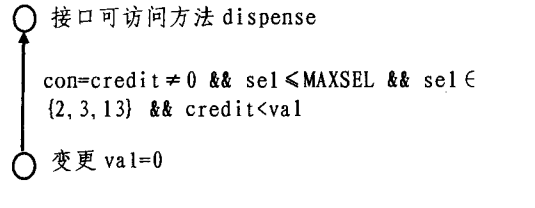


图 3　构件 Dispenser 修改后的变更-接口映射图

对构件 Dispenser 修改后，只有在 con 为真的条件下才会影响方法 dispense。根据定义可以这样来刻划该变更模型：(2,ptr)，ptr 指向该变更发生后接口的有关描述。

这里描述为一个三元组(方法名，返回值，参数列表)，因此可以这样刻划图 1 中的接口(dispense，int，(int credit，int sel))，其中(int credit，int sel)为方法 dispense 的参数列表。由于构件 Dispenser 只有一个可访问方法 dispense(一个接口)，因此接口表现为一个三元组。当一个构件有多个接口时，可以这样来定义：

**定义 9**　构件接口模型(Component Interface Model，记为 CIM)为一个三元组集合

$$CIM=\{(M_{pub}, V_r, L_{para}) \mid M_{pub}\in M\cap V_r\in\{\text{int,char,array}\cdots\}\cap L_{para}\in\{(\text{paratype}_1\ \text{paraname}_1,\cdots,\text{paratype}_n\ \text{paraname}_n)\}\}$$

其中，$M$ 为构件内方法集合，$M_{pub}$ 为可访问方法；$V_r$ 为可访问方法的返回值；$L_{para}$ 为参数列表，表示为 $(\text{paratype}_1\ \text{paraname}_1,\cdots,\text{paratype}_n\ \text{paraname}_n)$，$\text{paratype}_1,\cdots,\text{paratype}_n$ 为参数类型，$\text{paraname}_1,\cdots,\text{paraname}_n$ 为参数名。

由上所述，可以这样来构造构件内方法变更到构件接口变更的映射算法。为了方便起见，该算法用类 C 语言来进行描述。

算法：C-IMA(Change-Interface Mapping Algorithm)

输入：构件变更信息集合 $C=\{\alpha \mid \alpha$ 表示为 $CC=(C_t, C_d)$，其中 $C_t$ 表示变更类型，$C_d$ 表示变更描述$\}$；

输出：构件接口变更描述

```
for (∀α∈C){
switch(m_t)
{ c1:switch(c1)
{case c11:{ cnamemod(char oldname, char newname);//构件名的修改
break; }
case c12: vnamemod(char oldname, char newname);//构件内变量名的修改
break; }
case c13:{ mnamemod(char oldname, char newname);//构件内方法名的修改
break; }
c2:{ for ∀m_i, m_j∈{M_pub∪M_pri}
r_ij=cmethodde(m_i,m_j); //计算构件内所有方法之间的依赖关系 r;
for (i=1,j=1;i<=n,j<+n; i++,j++) //n 为构件内的方法数
{ m=caffectedm (α, r_ij); //根据方法依赖结果计算受变更 α 影
```

```
响的方法 m;
    inch=cimapping(α,m); //根据变更 α 和受 α 影响的方法 m 将方法内变更到接口变更 inch;
    moin(inch); //修改元数据中接口属性提供给构件使用方;
    } } }
```

## 4 应用实例

在上述理论研究中，使用的是 Orso 在文献[2,3]所使用的程序。由于该程序比较简单，复杂度不高，而且对程序进行的修改较为简单，仅仅在一处进行了修改，不能准确反映出本文提出的理论模型的可用性。

为验证所提出的模型在实际应用中的使用效果，本文将这些模型研究应用在自主开发的构件 RegisaStutGrade 中，为基于构件开发技术的教学管理系统 TeachMana 实现的学生成绩管理功能，篇幅所限，没有列出构件 RegisterStuGrade 和应用程序 TeachMana 的源代码。

当构件 RegisterStuGrade 发布给学校相关单位使用近两个月后，发现了该构件的 10 处错误。对这 10 处错误的修改分三次进行：

(1)第一次修改了 3 次错误，修改后对构件 RegisterStuGrade 实施第一次回归测试。在回归测试的测试用例生成过程中[6]，使用变更模型形成描述变更的元数据。并在此基础上，对上节中生成的 55 个测试用例中运用元数据中变更模型提示的信息生成了需要重新运行的 21 个回归测试用例，另外 34 个测试用例不需要重新运行。这里将以前未经修改的构件 RegisterStuGrade 称为版本 $V_1$，经过第一次修改后的构件 RegisterStuGrade 称为版本 $V_2$。

(2)第二次修改了其中的另外 4 个错误，将此时修改后的构件 RegisterStuGrade 称为版本 $V_3$。经过运用上述方法发现，55 个测试用例中有 18 个测试用例需要重新运行，另外 37 个测试用例不需要重新运行；

(3)第三次修改了其中的最后 3 个错误，将此时修改后的构件 RegisterStuGrade 称为版本 $V_4$。经过运用上述方法发现，55 个测试用例中有 12 个测试用例需要重新运行，另外 43 个测试用例不需要重新运行。

如果不使用变更的元数据来描述对构件 RegisterStuGrade 所实施的变更，则对构件 RegisterStuGrade 实施任一个变更。由于该构件的源代码对于实施回归测试的人员来说未知，因此就需要重新运行所有的 55 个测试用例。本文对引入变更模型和算法 C-IMA 的方法(记为 CM-Metadata 方法)和没有元数据的方法(记为 No-Metadata 方法)进行了分析和对比，在此基础上，也将 Orso 等人在文献[2,3]中给出的元数据框架用在本文给出的构件 RegisterStuGrade 中(记为 T-Metadata 方法)，发现第一次修改 3 个错误后使用元数据确定需重新运行的测试用例数为 29 个，第二次修改后需要重新运行的测试用例数为 24 个，第三次修改后需要重新运行的测试用例数为 25 个。这些对比和分析得出的结论如图 4 所示。

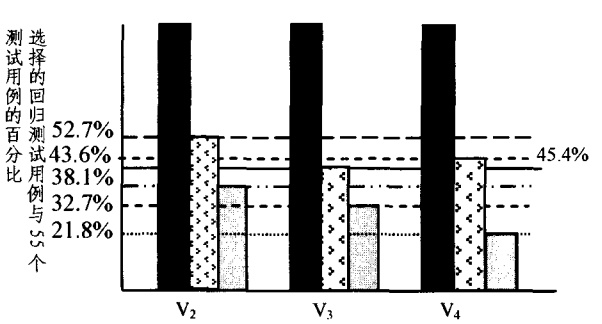


图 4 No-Metadata 方法(黑色柱形表示)、T-Metadata 方法(白色花形柱形表示)与 CM-Metadata 方法(灰色柱形表示)的回归测试用例选择比较

图 4 中，横坐标为构件版本，$V_2$，$V_3$ 和 $V_4$ 分别表示构件 RegisterStuGrade 经过三次修改后的版本号，纵坐标为回归测试用例数占初始测试用例数的比例。

从图 4 中可看出，使用本文提出的基于变更模型的元数据方法与无元数据和传统元数据方法相比，极大地减少了回归测试用例数，回归测试效率有明显的提高，使用基于变更模型的元数据生成测试用例的方法可以更好地实施软件回归测试。

另外，本文进一步用回归测试执行时间来度量基于变更模型的元数据(CM-Metadata)方法与无元数据(NO-meta)方法、传统元数据方法(T-Metadata)的比较。假设三种方法开始执行时间是一样的，表中的时间表示形式为分:秒，结果如表 2 所示。

表 2 构件 RegisterStuGrade 分别运用 NO-meta 方法、T-Metadata 方法和 CM-Metadata 方法的测试执行时间表

| 构件 RegisterStuGrade 版本 | No-Metadata 方法 | T-Metadata 方法 | CM-Metadata 方法 |
|---|---|---|---|
| $V_2$ | 28:23 | 19:15 | 15:26 |
| $V_3$ | 28:12 | 17:13 | 12:44 |
| $V_4$ | 27:56 | 18:11 | 11:23 |
| 平均时间 | 28:10 | 18:13 | 13:11 |
| 总时间 | 84:31 | 54:39 | 39:33 |

**结束语** 本文在对构件可能的变更情况进行讨论的基础上，进行构件变更分类，并根据分类情况给出相关的变更处理描述，构造了形式化的变更模型，描述了每个变更的具体表现形式，论述了方法变更到构件接口变更的映射机制和算法，给出了算法实现的框架，并进行了相关的实例研究，证明该方法的有效性。

下一步要进一步讨论如何针对不同复杂度的软件有针对性地提供变更模型，以减少模型本身的复杂性，扩展应用范畴。另外，还需逐步建立自动化的回归测试用例生成方法。

## 参考文献


[1] Zheng J. In Regression Testing Selection When Source Code is Not Available//20th IEEE/ACM International Conference on Automated Software Engineering. California, USA: November 2005:452-455

[2] Orso A, Harrold M J, et al. Using Component Metacontent to Support the Regression Testing of Component-Based Software. //Proceedings of IEEE International Conference on Software Maintenance (ICSM'01). Florence, Italy: November 2001:716-725

[3] Orso H 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, 2001:1-10

[4] Jean M, Liang D, Sinha S. An Approach To Analyzing and Testing Component-Based Systems. //ICSE'99 Workshop on Testing distributed Component-based Systems. May 1999:210-216

[5] Liang D, Harrold M J. Slicing Objects Using System Dependency Grapgs// Proceedings of the International Conference on Software Maintenance. November 1998:358-368

[6] Liu C, Richardson D. Software Components with Retrospectors//Proceeding of International Workshop on the Role of Software Architecture in Testing and Analysis. June 1998: 63-68