ISSN 1000-9825, CODEN RUXUEW
Journal of Software, Vol.19, No.10, October 2008, pp.2527−2538
DOI: 10.3724/SP.J.1001.2008.02527

E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel/Fax: +86-10-62562563

一种解决构件连接死锁问题的方法*

毛斐巧+, 齐德昱, 林伟伟

(华南理工大学 计算机科学与工程学院,广东 广州 510640)

An Approach to Solve Deadlock Problem of Component Connection

MAO Fei-Qiao+, QI De-Yu, LIN Wei-Wei

(College of Computer Science and Engineering, South China University of Technology, Guangzhou 510640, China)

+ Corresponding author: E-mail: maofeiqiao@163.com

Mao FQ, Qi DY, Lin WW. An approach to solve deadlock problem of component connection. *Journal of Software*, 2008,19(10):2527−2538. http://www.jos.org.cn/1000-9825/19/2527.htm

Abstract: Procedure call-based component connections which are latent in hard code do not only restrict the flexibility of software but also cause hidden problems to software reliability because of the existing deadlock connection loops. To solve this problem, first, a formal semantic model called call-based connector has been built which explicitly separates connection from components. Second, mapping rules used to convert call-based connectors into component the connection directed graph are proposed. Then, two algorithms, TPDCC (two phases deadlock connection check) used to find all existing deadlock connection loops, and DCEMRF (deadlock connection elimination based on maximum reuse frequency) used to find locations with the least number of connections that must be eliminated to eliminate the loops, are provided respectively. Last, its application and experimental results show that the presented approach is feasible and effective, so it can be used to enhance the reliability of software, also be fit as a basis to further design and implement adaptive connector due to its separative way of description and storage of component connection in semantic.

Key words: adaptability; reliability; connector; component; deadlock

摘 要: 隐式硬编码的基于过程调用构件连接束缚构件集成的灵活性,且存在的死锁连接造成软件可靠性隐患问题.针对该问题,首先建立基于过程调用连接器形式语义模型,显式地将连接关系从构件中分离;然后给出并通过映射规则进行连接器到构件连接有向图的转换,并设计给出两阶段死锁检查算法和基于极大复用频率死锁连接消除算法,用于找到存在的所有死锁连接回路和消除所有死锁连接需要消除的最小数目连接的位置.最后应用及实验结果表明,该解决方法可行而且有效,可以用于增强软件可靠性,同时因其从语义上分离描述和存储构件连接方式,适合以此为基础进一步设计实现适应性连接器.

关键词: 适应性;可靠性;连接器;构件;死锁

中图法分类号: TP311 **文献标识码**: A

* Supported by the Natural Science Foundation of Guangdong Province of China under Grant No.05300200 (广东省自然基金); the Guangdong-Hong Kong Technology Cooperation Funding Scheme of China under Grant No.2005A10307007 (粤港关键领域重点突破项目)

Received 2007-06-04; Accepted 2007-10-09

在构件化软件中,通过构件连接将构件集成为应用软件.根据交互方式的不同,构件间的连接有基于消息传递的连接、基于过程(也称为方法或函数)调用的连接和基于事件传递的连接等[1].在形式化软件体系结构研究中将基于各种不同交互方式的连接均抽象为非功能实体连接器.而构件是功能实体,目前软件开发中使用的主流构件模型主要有 OMG 的 CORBA,Microsoft 的 COM/DCOM/COM+和 Sun 的 JavaBean/EJB,这些功能实体间的交互以过程调用(本地或远程)为主[2].并且这种基于过程调用交互方式的连接是直接的,隐式的硬编码在构件实现体中,没有显式地与构件分离,使构件之间的连接呈现一种紧密耦合方式.这种连接方式不仅限制了构件间集成的灵活性和可扩展性,而且使构件间极易存在循环递归调用环.这种环在软件编译链接时无法发现,而在软件运行时形成死锁,造成软件稳定性和可靠性的隐患问题.所以,本文针对基于过程调用构件连接方式中存在的死锁连接问题进行研究,建立基于过程调用连接器形式语义模型,在此基础上,提出两阶段死锁连接检查算法,用于找出造成死锁的构件间连接存在的循环递归调用环;然后给出基于极大复用频率的死锁连接消除算法,用于找到存在的所有死锁连接回路和消除所有死锁连接需要消除的最小数目连接的位置.在为该问题提供一种可行而有效的解决方法的同时,为增强软件的稳定性和可靠性提供一条途径,并且为设计实现具有适应性的构件连接器奠定基础.

1 问题描述

这里,从一个简单的例子引出问题.4 个由 Java 类实现的构件体(Scone,Sctwo,SCthree 和 SCfour)连接集成的小示例应用(这里只是为了说明和描述问题而设计的一个非常简单的例子,在实际应用中构件的实现体要复杂得多),如图 1(a)~图 1(d)所示.在编译链接时没有任何问题,运行时却异常终止.其错误提示如图 2(a)所示,即发生了堆栈溢出.这是因为,如图 2(b)所示,其中 3 个构件(Scone,SCtwo 和 SCthree)的方法之间调用形成了连接环,运行时一直沿着环进行循环递归调用,而不会有递归结束条件产生,形成僵局,导致不停地有构件实例对象产生,直到存储空间不足,应用才异常终止.

因此,可将基于过程调用构件连接集成中的死锁连接问题定义为:

定义 1. 在基于过程调用构件连接集成的软件中,因多个构件间的调用交互,某一构件的功能方法(即过程)间接调用其自身,在构件的功能方法之间形成连接环,导致软件运行时构件间的循环递归调用无法返回,造成一种僵局,使软件无法向前运行,甚至异常终止.

SCone.java
```
1  package deadlockTest;
2  public class SCone {
3      public int add( int a, int b ){
4          SCtwo obj = new SCtwo();
5          a = obj.multiply(a, a);
6          a = inc(a);
7          return a+b;
8      }
9      private int inc(int a ){
10         return a+1;
11     }
12 }
```
(a)

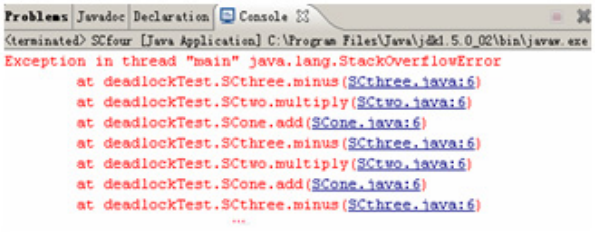
SCtwo.java
```
1  package deadlockTest;
2  public class SCtwo {
3      public int multiply(int a, int b ){
4          SCthree obj = new SCthree();
5          a = obj.minus(a);
6          a = dec(a);
7          return a*b;
8      }
9      private int dec(int a ){
10         return a-1;
11     }
12 }
```
(b)

SCthree.java
```
1  package deadlockTest;
2  public class SCthree {
3      public int minus(int a){
4          SCone obj = new SCone();
5          a = obj.add(a, 5);
6          return a-10;
7      }
8  }
```
(c)

SCfour.java
```
1  package deadlockTest;
2  public class SCfour {
3      public static void main( String[] args ){
4          SCone obj = new SCone();
5          int c = obj.add(3, 2);
6          System.out.println(c);
7      }
8  }
```
(d)

Fig.1 A simple example

图 1 一个简单的例子



(a) (b)

Fig.2 Deadlock problem

图 2 死锁问题

而要解决该问题就需要找出构件连接间存在的所有死锁连接,并消除这些死锁连接.

2 解决方法

2.1 连接器形式语义模型

从构件间的交互方式出发,建立基于过程调用连接器形式语义模型,抽象出构件间的连接关系,为死锁连接检查奠定基础.

在设计模式中,硬编码基于过程调用构件的交互语法格式为:

(1) 有返回值的,⟨resultVariable⟩=⟨objectname⟩.⟨methodname⟩(⟨arguments⟩)或

(2) 无返回值的,⟨objectname⟩.⟨methodname⟩(⟨arguments⟩).

某些情况下,arguments 也可以为空.

在这种格式的描述下,调用构件和被调用构件之间建立起连接.目前进行编译链接的软件开发工具,只能检查出其中存在的语法错误,而对于语义层次上的可能造成像死锁连接这样的逻辑错误,却不能发现.因此,建立连接器形式语义模型,以支持语义层次上像死锁连接这样的逻辑错误的自动化检查.

已存在一些具有代表性的与构件连接的形式语义建模相关的研究.

以 Allen[3]为代表的一类研究者,以 CSP(communicating sequential processes)为基础将构件间的交互方式抽象为连接器时(如文献[4−6]),连接器的形式语义模型为由 role 和 glue 描述的 CSP 进程.他们主要从参与交互的角色和交互过程的行为协议角度来建立模型.基于这种模型,可以通过描述转化预处理后,使用商业 FDR(failure divergence refinement)模型检查工具检查单个连接器本身各 role 与 glue 中是否存在等待死锁,不支持应用系统内连接器间形成的构件死锁连接检查.

以 Magee[7]为代表的另一类研究者,以π演算[8]为基础建立构件连接形式语义模型.如 Magee 以一阶π演算为基础,将连接器描述为 bind 断言,以声明相交互的构件间 provides 服务的 output 和 requires 服务的 input 间的对应关系;再如文献[9−11]以高阶π演算为基础,将连接器描述为一组端口 port 和一个路由行为 routing.这类以π演算为基础建立的连接器语义模型,支持描述 SA(software architecture)的动态行为以及 SA 的演化和精化,但不支持死锁检查和相容性检查.

与上述建模方式不同,针对基于过程调用构件交互方式,我们结合 CSP 和谓词逻辑建立基于过程调用连接器语义模型.基于这种模型,不仅能够检查出连接器间形成的构件死锁连接,而且能够找到消除死锁连接的位置,同时提供了专门的检查工具 DCCTool(deadlock connection CheckTool).

在 Wright[12,13]中,将构件形式化为 CSP[14]中的进程(process).这里,构件也按这种语义形式化,另外,以 *C* 标识,并附以下标以区别不同的构件.下面借用 CSP 中的符号,给出建立这种连接器形式语义模型的其他相关语义定义.

定义 2. 构件对外提供的任一功能操作方法(即过程),称为构件的外交互点,其形式语义为外通道(external-channel),记为 *OP*,并附以下标以区别同一构件内对外提供的不同方法.其中:

(1) 构件内部隐藏的、任一有与其他构件外交互点交互的功能操作方法(过程),称为构件的内交互点,其形式语义为内通道(internal-channel),记为\OP,并附以下标以区别同一构件内隐藏的不同方法;

(2) 返回值类型形式化为内/外通道输出类型,记为 X,并附以下标以区别不同的通道输出类型;

(3) 输入参数类型列表形式化为内/外通道输入类型,记为 Y,并附以下标以区别不同的通道输入类型;

(4) 符合 X 类型的变量或值 x,记为 $X:=x$;

(5) 符合 Y 类型的变量向量或值向量 y,记为 $Y:=y$;

(6) Y 和 X 均可以为空;

(7) $X:=x,z$,则表示变量或值 x 和 z 均符合 X 类型;

则将构件两种类型的交互点分别形式化为 $External\text{-}channel=OP!Y?X$,$Internal\text{-}channel=\backslash OP!Y?X$.

定义 3. 构件对外提供的某一功能方法 OP,其形式语义为进程 C 的某一通道 OP,记为 $C.OP$;内部隐藏的某一功能方法 OP,其形式语义为进程 C 的某一内通道,记为 $C.\backslash OP$.

定义 4. 构件的所有实例形式化为集合 L,与构件之间的关系形式化为 $L:C$;构件的某一实例形式化为集合元素 l,与构件之间的关系形式化为 $l:C$,且有 $l\in L$;实例对外提供的某一功能方法记为 $l:C.OP$.

定义 5. 构件任一功能方法内部,显式或隐式存在的所有变量元或常量元,其形式语义定义为通道状态元集,记为 $OP.Sch$;则其上任一状态元 y,有 $y\in OP.Sch$;引起状态元 y 状态变化的算法步骤,形式化为映射函数 f,变化前的状态为 y,变化后的状态为 $f(y)$,对于常量元,则是一种等价映射,有 $y=f(y)$.例如,构件一种功能方法内两条语句:int a=0;a++;状态元 a 在变化前为 a=0;经自增算法运算后 $f(a)$=1.另外,两个状态元相等用"="连接来表示.

定义 6. 硬编码的基于过程调用交互中遵循的交互协议,其形式语义为连接(join),记为 $Join=A\Rightarrow B$,其中,前件 A 描述交互协议中应满足的条件,后件 B 描述满足交互协议条件的结果.

为降低语义描述的复杂性,我们做出以下假设约束:

约束规则 1. 不允许构件中出现对外交互的内交互点,凡是内交互点,均修改为外露的.

事实上,在目前的程序设计中是没有这种设计约束的,允许构件的私有方法调用其他构件的公有方法.但是,假设约束中规定做出的修改,也是目前的程序设计所允许的,所以这种假设约束是允许的.通过约束规则 1,将两种类型的交互点统一为一种形式:$OP!Y?X$.

为了避免在死锁连接检查算法中构造构件连接有向图时生成具有重边的有向图,以降低生成的有向图的复杂性,我们做出以下假设约束:

约束规则 2. 不允许在构件的一种功能方法中,重复调用另一构件的同一种方法.

事实上,在目前的程序设计中也没有这种设计约束,在不少情况下是存在重复调用的,并且也是需要存在的.所以这种约束对于实际程序设计来说不太恰当,但这里做出这种约束,目的是为了降低后面问题解决的难度.另外,针对这种假设的不恰当性,规定这种约束是非强制性的,在必须的情况下,可以违反这种约束,出现重复调用.如果出现这种重复,则做出简化处理,认为只调用了 1 次.这种处理不会影响死锁连接回路的查找,也避免了重边的出现.

基于上述形式语义定义和约束规则,将两构件(C_1 和 C_2)之间基于过程调用的交互,形式化为基于过程调用连接器(call-based connector),其形式语义模型如下:

Call-based Connector

$Channel\ caller=C_1.OP_1!Y_1?X_1$

$Channel\ callee=C_2.OP_1!Y_2?X_2$

$Join=\exists y_1,z_1\in C_1.OP_1!Y_1?X_1.Sch\wedge Y_2:=f(y_1)\wedge X_2:=x_2,z_1\wedge x_2\in C_2.OP_1!Y_2?X_2.Sch\Rightarrow l_1:C_2.OP_1!f(y_1)?x_2\wedge z_1=x_2$

在该模型中,通道 channel 描述了连接器的两个交互点,并用 caller 和 callee 标识调用者和被调用者;连接 Join 描述了两个交互点间进行交互遵循的协议.模型将一种特殊情况也包含在内:caller 和 callee 分别标识的 channel 可以是同一个构件的两个不同通道,这种情况下,模型表示的语义为同一构件不同通道之间的交互.因为在程序设计中,为提高代码的重用性,一个构件对外提供的任一功能方法也允许被该构件对外提供的其他功

能方法所调用,这种重用方式被开发人员使用后容易间接地造成构件间的死锁连接,所以将这种特殊情况考虑在内以扩大连接器的语义范围,从而将构件内部基于过程调用造成的死锁连接也覆盖在内.该模型除了可以用于语义层次上的死锁连接检查以外,还可用于语法层次上的类型匹配检查,这里的研究主要关注前者.

2.2 两阶段死锁连接检查算法

基于第2.1节的Call-Based Connector模型,可以将各个构件中各处硬编码的过程调用交互连接抽象成为一个个的Call-Based Connector,然后进一步在其中找出存在的死锁连接.为了在这些Call-Based Connectors中找出存在的所有死锁连接,给出两阶段死锁连接检查算法(two phases deadlock connection check,简称TPDCC).

TPDCC算法第1阶段是基于Call-Based Connectors动态构造构件连接有向图,第2阶段是在有向图中查找出所有简单回路.在描述算法之前,先给出从Call-Based Connector到构件连接有向图的映射规则、判定规则以及相关定义.

映射规则1. 将caller和callee标识的Channel分别映射为构件连接有向图中的顶点.

定义7. 对于 $Channel1=C_1.OP_1!Y_1?X_1$ 和 $Channel2=C_2.OP_1!Y_2?X_2$,若 *Channel*1 与 *Channel*2 的形式一样,即 $C_1.OP_1!Y_1?X_1=C_2.OP1!Y_2?X_2$,则称 *Channel*1 和 *Channel*2 为等价通道,描述的是同一个交互点.

映射规则2. 等价通道映射为构件连接有向图中相同的顶点.

映射规则3. 将Join映射为构件连接有向图中连接代表其caller Channel和callee Channel的顶点的有向边;将Join后件中的 *li*:*Cj* 映射为构件连接有向图中代表其所在的Join的有向边的边权.

映射规则4. 将caller标识的Channel映射为构件连接有向图中代表其所在的Join的有向边的始点;将callee标识的Channel映射为构件连接有向图中代表其所在的Join的有向边的终点.

映射规则5. 将Call-Based Connectors间存在的死锁映射为构件连接有向图中的回路.

判定规则1. 若构件连接有向图中存在一个简单回路,则构件连接间存在一个死锁连接,并且死锁连接就是该简单回路.

在上述映射规则的基础上,可以应用如下所示的TPDCC算法(在算法中默认起始输入的连接器个数至少为1)动态地构造构件连接有向图,并将查找构件连接中的所有死锁连接问题,转化为查找构件连接有向图中存在的所有简单回路问题,找出构件连接中的所有死锁连接.

TPDCC算法伪码描述.

```
Step1.1: {Connector}=inputAllConnectors(connectorfile.xml);
         {Connector}.length=computeConnectorLength();i=-1;{Vertex}=∅;{Edge}=∅;go to Step1.2;
Step1.2: if ({Connector}.length>=1){i++;{Connector}.length-=1;go to Step1.3;} else {go to Step2.1;}
Step1.3: {Connector}[i]=getConnector(i);
         if ({Connector}[i].caller∉{Vertex}){{Vertex}.createV({Connector}[i].caller);}
         if ({Connector}[i].callee∉{Vertex}{{Vertex}.create({Connector}[i].callee);}
         {Edge}[j]={Edge}.createE({Connector}[i].caller,{Connector}[i].callee);
         {Edge}[j].power={Connector}[i].Join.getPowerinB();go to Step1.2;
Step2.1: computeInandOutDegreeofVertex({Vertex});
         {ZeroinDegreeVertex}=findAllZeronInDegreeVertex({Vertex});
         {Circle}=∅;{UnvisistedVertex}={Vertex};go to Step2.2;
Step2.2: if ({ZeroinDegreeVertex}!=∅){
            {ZeroinDegreeVertex}[m]=findMaxOutDegreeVertex({ZeroinDegreeVertex});go to Step2.3; }
         else {if ({UnvisistedVertex}!=∅){
                  {UnvisistedVertex}[m]=findMaxOutDegreeVertex({UnvisistedVertex});
                   startVertex={Vertex}[{UnvisistedVertex}[m].index];
                    {Presequence=∅;k=-1;go to Step2.4; }else {go to Step2.6;}}
Step2.3: startVertex={Vertex}[{ZeroinDegreeVertex}[m].index];
```

{Presequence}=∅;k=−1;{ZeroinDegreeVertex}.delete(m);go to Step2.4;
Step2.4: if ({Presequence}==∅){{Presequence}.add(startVertex);go to Step2.5;}
else{k={Presequence}.find({startVertex};}
if (k>=0){{Circle}.add({Presequence}.copySequence(k));k=−1;
startVertex={Presequence}.getLastNode();go to Step2.5;}
else {{Presequence}.add(startVertex);go to Step2.5;}}
Step2.5: if ({Vertex}[IndexofstartVertexin{Vertex}].visited==true){
If (startVertex∈vertexesIn{Circle}){
{Presequence}.deleteLastNode();startVertex={Presequence}.getLastNode();
if (startVertex==null){modify{UnvisitedVertex};go to Step2.2;} else {go to Step2.5;}
else {if({Vertex}[IndexofstartVertexin{Vertex}].outDegree!=0){
{Vertex}[IndexofstartVertexin{Vertex}].visited=false;
for (each adjacent edge i of {Vertex}[IndexofstartVertexin{Vertex}]){
{Edge}[i].visited=false;go to Step2.5;}}
else{{Presequence}.deleteLastNode();startVertex={Presequence}.getLastNode();
if ({startVertex==null}{modify{UnvisitedVertex};go to Step2.2;} else {go to Step2.5;}))
else {if ({Vertex}[IndexofstartVertexin{Vertex}].outDegree==0||all adjacent edges are visited){
{Vertex}[IndexofstartVertexin{Vertex}].visited=true;
{Presequence}.deleteLastNode();startVertex={Presequence}.getLastNode();
if (startVertex==null){modify{UnvisitedVertex};go to Step2.2;} else {go to Step2.5;}}
else {choose an edge i in its unvisited adjacent edges;
startVertex={Edge}[i].endNode;{Edge}[i].visited=true;go to Step2.4;}}
Step2.6: if ({Circle}!=∅){outputallcircles({Circle});} else {output:no deadlock circles.}

2.3 基于极大复用频率死锁连接消除算法

找出所有死锁连接后,关键问题是如何消除这些死锁连接.造成死锁连接是因为连接形成了回路,因此要找到合适的有向边作为破除边,有效地打破回路.我们给出基于极大复用频率的死锁连接消除算法(deadlock connection elimination based on maximum reuse frequency,简称 DCEMRF).该算法主要解决了消除死锁连接的关键部分,即找出要破除的有向边.而死锁连接的真正消除还要进行具体的实现工作:修改设计实现,即根据找出的要破除的有向边,对应到其所代表的构件交互连接的实现;在保证功能不变的前提下,修改这些实现,去掉这些交互连接,即进行代码重构.针对实现,为防止进行代码重构后形成新的死锁连接,这里给出修改设计实现的约束规则.

约束规则 3. 要破除构件交互连接,需修改连接实现的位置;需修改的位置,在进行代码重构后,不存在连接,或存在连接但与之连接的交互点有且仅有一个连接.

该算法的主要思想是,找出回路集内存在的不同边中复用频率最高的一个作为消除边,然后删除回路集内所有包含有此边的回路;如此反复进行,直到回路集变空,便找到了破除所有回路,要破除的最小数量的有向边.其中复用频率的定义如下:

定义 8. 在检查出的所有死锁连接回路中存在的、代表不同构件连接的有向边,在所有回路中出现的次数之和,称为该有向边(或构件连接)的复用频率.

DCEMRF 的算法描述如下所示(在算法中默认输入的死锁连接回路集非空).

DCEMRF 算法伪码描述.

Step1: inputdeadlockcircles({circle}); go to Step2;
Step2: computeEdgeReuseFrequency({circle},{Edge});
{Edge}[m]=findMaxReuseFrequency();
{EleminateEdge}.add({Edge}[m]); go to Step3;

```
Step3: for (each circle∈{circle}){
          If ({Edge}[m]∈circle.{edges})
              deletecircle(circle);}
        if ({circle}!=∅) {go to Step2;}
        else {go to Step4;}
Step4: if ({EleminateEdge}!=∅){
          outputEleminatedEdges({EleminateEdge});}
       else{output: error message;}
```

3 应用及实验结果与分析

3.1 方法的应用

在我们正在研发的软件开发平台 SmartFramework 中,应用上述方法实现了 DCCTool 子工具.使用 DCCTool,可以检查出应用系统的构件中存在的所有基于过程调用死锁连接,并找到消除这些死锁连接需要打破的构件连接.不过,目前从硬编码的构件连接到抽象语义模型连接器的建立这一步来看,DCCTool 还没有实现自动化,还要靠人工将硬编码的基于过程调用的构件连接抽象成为连接器,并将之保存在特定结构的 XML 格式文件中.在 DCCTool 中实现 TPDCC 算法时,连接器直接从 XML 格式文件中输入.第 3.4 节给出了我们在实际系统中使用 DCCTool 进行检测的结果.

3.2 实验与结果

为了检验解决方法的有效性、TPDCC 算法和 DCEMRF 算法的正确性,并测量这两种算法的性能,我们设计测试实例,使用 DCCTool 进行实验.

实验硬件环境是 P4 CPU 2.40GHz,1G 内存;实验软件环境为 DCCTool.另外,DCCTool 的运行环境 JVM 的内存空间范围为(64M~512M),版本为 jdk1.5.0_02.

测试实例设计:设计 10 组测试实例,调整每组实例中包含的交互点的个数 v,连接器个数 m 和死锁连接回路个数 n,在 DCCTool 中分别输入每组测试实例并运行该工具,观察运行结果中找到的死锁连接回路、要消除的构件连接、TPDCC 算法和 DCEMRF 算法执行的时间以及占有的内存空间.其中 1 组测试实例含 10 个交互点、12 个连接器和 2 个死锁连接回路(限于篇幅,这里省略详细实例),该实例其中 1 次测试运行结果如图 3 所示,其他测试实例都是在该实例的基础上进行增加和删除的,这里也省略描述,只给出运行结果中要消除的连接,两种算法各自的执行时间和占有的内存空间,见表 1(注:每组测试实例都至少运行 5 次,两种算法执行时间和占有的存储空间值取其中较稳定的 3 次的平均值作为最终结果).

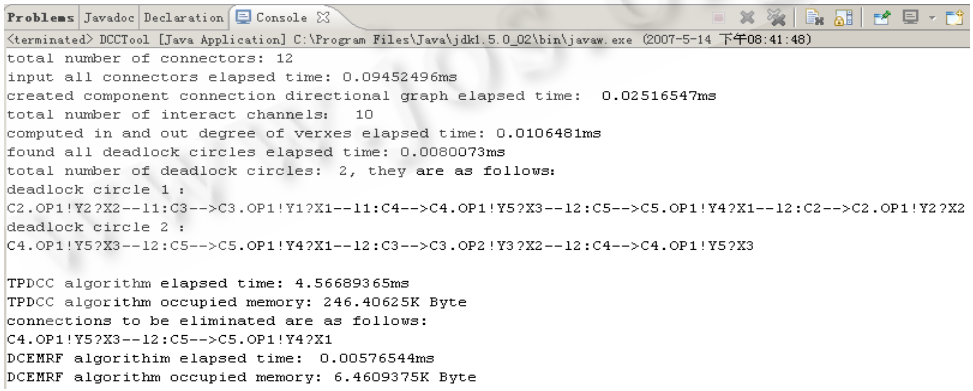


Fig.3 A testing result

图 3 一个测试结果

通过利用这 10 组测试实例进行实验,所测结果表明,应用所提出的解决方法是能够找到构件连接中存在的所有死锁连接的,并能正确给出破除所有死锁连接需要消除的连接.

Table 1 Running results of 10 groups of test cases

表 1 10 组测试实例运行结果

Group	v	m	n	TPDCC algorithm		DCEMRF algorithm		Connections to be eliminated
				Time (ms)	Space (Kbyte)	Time (ms)	Space (Kbyte)	
0	10	11	1	4.479 5	248.570	0.007 778	6.461	*C*2.*OP*1!*Y*2?*X*2--*I*1:*C*3→*C*3.*OP*1!*Y*1?*X*1
1	10	10	0	4.248 1	57.273	0.002 973	6.797	No connection to be eliminated
2	10	12	2	4.529 0	246.424	0.008 283	6.477	*C*4.*OP*1!*Y*5?*X*3--*I*2:*C*5→*C*5.*OP*1!*Y*4?*X*1
3	14	16	2	4.604 8	253.563	0.009 047	6.555	*C*4.*OP*1!*Y*5?*X*3--*I*2:*C*5→*C*5.*OP*1!*Y*4?*X*1
4	16	18	2	4.616 1	257.344	0.008 986	6.836	*C*4.*OP*1!*Y*5?*X*3--*I*2:*C*5→*C*5.*OP*1!*Y*4?*X*1
5	18	21	3	4.661 4	265.367	0.048 245	7.531	*C*4.*OP*1!*Y*5?*X*3--*I*2:*C*5→*C*5.*OP*1!*Y*4?*X*1 *C*4.*OP*2!*Y*2?*X*4--*I*2:*C*6→*C*6.*OP*1!*Y*2?*X*2
6	23	26	3	4.734 5	270.836	0.013 131	6.320	*C*4.*OP*1!*Y*5?*X*3--*I*2:*C*5→*C*5.*OP*1!*Y*4?*X*1 *C*4.*OP*2!*Y*2?*X*4--*I*2:*C*6→*C*6.*OP*1!*Y*2?*X*2
7	26	28	2	4.777 0	271.031	0.012 835	6.320	*C*2.*OP*1!*Y*2?*X*2--*I*1:*C*3→*C*3.*OP*1!*Y*1?*X*1 *C*4.*OP*2!*Y*2?*X*4--*I*2:*C*6→*C*6.*OP*1!*Y*2?*X*2
8	30	34	4	4.873 7	285.617	0.020 267	12.328	*C*4.*OP*1!*Y*5?*X*3--*I*2:*C*5→*C*5.*OP*1!*Y*4?*X*1 *C*8.*OP*1!*Y*2?*X*1--*I*1:*C*10→*C*10.*OP*1!*Y*2?*X*4 *C*4.*OP*2!*Y*2?*X*4--*I*2:*C*6→*C*6.*OP*1!*Y*2?*X*2
9	33	38	5	4.961 2	294.383	0.032 842	18.805	*C*4.*OP*1!*Y*5?*X*3--*I*2:*C*5→*C*5.*OP*1!*Y*4?*X*1 *C*8.*OP*2!*Y*3?*X*3--*I*2:*C*9→*C*9.*OP*2!*Y*2?*X*2 *C*8.*OP*1!*Y*2?*X*1--*I*1:*C*10→*C*10.*OP*1!*Y*2?*X*4 *C*4.*OP*2!*Y*2?*X*4--*I*2:*C*6→*C*6.*OP*1!*Y*2?*X*2

3.3 算法性能分析

由 10 组测试实例测得的实际复杂度仅能反映出 TPDCC 算法和 DCEMRF 算法在连接器个数 m、交互点个数 v 和死锁连接回路个数 n 在较小范围内变化时的实际性能,而针对这两种算法也很难设计出具有代表性和普遍性的测试实例.另外,测试实例数据很难自动生成,目前主要靠手工设计,进行大规模的实例测试可行性不大.因此,我们进一步从理论上对 TPDCC 算法和 DCEMRF 算法进行渐近性能分析(m,v,n 为变量,其他均为常量).

在 TPDCC 算法中与时间复杂度相关的关键操作有输入 m 个连接器的时间 t_1m(t_1 为输入一个连接器的平均时间);创建所有顶点时间 t_2v(t_2 为创建一个顶点的平均时间);创建所有顶点所进行比较操作的时间 $(v(v+1)/2)t_3$(t_3 为一次顶点比较的平均时间);创建所有边的时间 t_4m(t_4 为创建一条边的平均时间);计算所有顶点出度和入度的时间 $2t_3v^2$;查找新一轮访问起始顶点最坏情况下的时间共为 nvt_3;查找回路在访问集中的起点位置的时间共约为 $n(v/2)t_3$;取所有回路的时间共约为 nt_5(t_5 为平均一次取回路的时间);找到所有死锁回路访问顶点最好情况下共需时间约为 vt_6+nt_6(t_6 为访问一个顶点的平均时间);输出所有回路的时间共约为 t_7n(t_7 为平均输出一个死锁回路的时间).因此,TPDCC 算法的渐近时间复杂度为 $O(m+v^2+nv)$.在 TPDCC 算法中与空间复杂度相关的关键数据存储空间主要有所有连接器占用的空间 p_2m(p_2 为一个连接器需占用的字节);所有顶点占用的空间 p_1v(p_1 为一个顶点需占用的字节);所有边所占用的空间 p_3m(p_3 为一条边需占用的字节);回路集{*circle*}共占用的空间约为 $(l_1/k_1)mn$(假设每个回路中含 $(1/k_1)m$ 条边,$1\le k_1\le m$,每个对象引用占 l_1 字节);入度为 0 的顶点共占用的空间约为 $(l_1/k_2)v$(设入度为 0 的顶点共有 $(1/k_2)v$ 个,$1\le k_2\le v$);未访问过的顶点集共占用的空间约为 $(l_1/k_4)v(1\le k_4\le v)$;一轮访问前序顶点集占用的空间约为 $(l_1/k_3)v(1\le k_3\le v)$.因此,TPDCC 算法的渐近空间复杂度为 $\Theta(v+mn)$.

在 DCEMRP 算法中,与时间复杂度相关的关键操作有计算复用频率的时间 $(n/l)k^2m^2$(假设需计算 l 轮,$1\le l\le n$,则每个死锁回路中有 km 条边,$0\le k\le l$);找出复用频率最大值边的时间约为 $t_3m^2k^2(n/l)$;输出要消除的边的时间约为 qnt_0(q 为输出一条边的平均时间,$0\le q\le 1$).因此,DCEMRP 算法的渐近时间复杂度为 $O(nm^2)$.在 DCEMRP 算法中,与空间复杂度相关的关键数据存储空间有回路集{*circle*}共占用的空间约为 $(l_1/k_1)mn$;要消除的边占用的空间 $(l_1/j_1)n(1\le j_1\le n)$;暂存待删除的回路对象的引用占用的空间约为 $(l_1/j_2)n(1\le j_2\le n)$.因此,DCEMRP 算法的渐近空

间复杂度为$\Theta(mn)$.

3.4 工具的应用

我们尝试在实际系统中使用 DCCTool 进行检测,所获结果初步体现出其在增强软件可靠性上存在的价值.所检测的福利注册系统是一个典型的由EJB构件基于过程调用连接而构成的应用系统,是在Sun公司提供的用于剖析 EJB 构架和说明如何使用 EJB 构件的开源福利注册应用程序[15]基础上扩展得到的.该系统中的构件在功能和类型上均具有代表性:EJB 构件的所有类型(有状态会话型、无状态会话型、消息驱动型和实体型)均有包含,企业信息系统常用的功能(如辅助展现,数据库初始化,增、查和改等业务处理功能)也有包含.并且系统所含的交互点个数和连接器个数与实验中的测试实例相比也有数量级的提高.

检测结果如图 4 所示,存在一个死锁连接.根据所建立的连接器模型在语义上与实际构件交互的映射关系(见表 2,这里只给出了与死锁连接相关的部分映射关系,并且构件对外提供的方法只给出了名称,其他部分略),对应可知,Options 构件的 getOptionDescription 方法、OptionContent 构件的 setContent 方法、EnrollmentBean 构件的 setMedicalOption 方法、SelectionCopy 构件的 setMedicalPlan 方法与 OptionContent 构件的 isAccess 方法之间形成了循环递归调用.其中,SelectionCopy 构件和 EnrollmentBean 构件属于实体型,Options 和 OptionsContent 构件分别属于有状态会话型和无状态会话型.

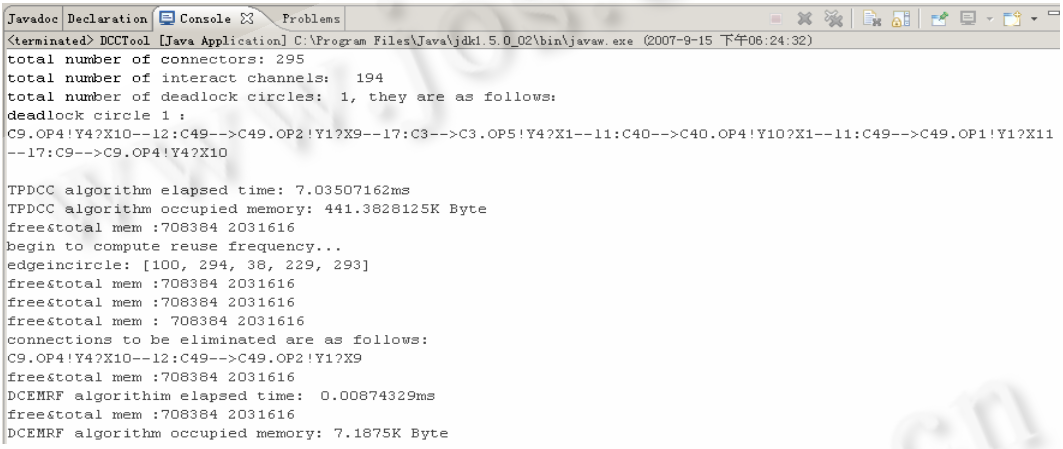


Fig.4 Checking result of welfare registration system

图 4 福利注册系统检测结果

Table 2 Partial mapping relations of connector model

表 2 连接器模型的部分映射关系

Components	Methods	Instance	Edge power	Connector channels
EnrollmentBean	setMedicalOption	eb	l7:*C*3	*C*3.*OP*5!*Y*4?*X*1
Options	getOptionDescription	ops	l7:*C*9	*C*9.*OP*4!*Y*4?*X*10
SelectionCopy	setMedicalPlan	selcopy	l1:*C*40	*C*40.*OP*4!*Y*10?*X*1
OptionContent	isAccess	opc	l1:*C*49	*C*49.*OP*1!*Y*1?*X*11
OptionContent	setContent	op	l2:*C*49	*C*49.*OP*2!*Y*1?*X*9

而且所指出的应当消除的死锁连接是 *C*9.*OP*4!*Y*4?*X*10—*l*2:*C*49→*C*49.*OP*2!*Y*1*X*9,即 Options 构件的 getOptionDescription 方法中创建了 OptionContent 构件名为 op 的实例,以 op 实例为对象调用 OptionContent 构件的 setContent 方法,按照第 2.3 节的约束规则 3,这段代码必须进行重构.我们采用的重构方式是直接删除这段代码,取而代之的是在 OptionContent 构件内部新建一种与外部无交互的私有方法,实现与 OptionContent 构件的 setContent 方法相同的功能,直接调用该私有方法.

在该系统检测过程中,TPDCC 算法耗时约 7.035 1ms,占存储空间约 441.383k;DCEMRF 算法耗时约 0.008 743ms,占存储空间约 7.188k.而该系统中包含的交互点个数为 194,连接器个数为 295,与测试实验中第 0 组测试

实例测试结果相比,当连接器个数和交互点个数有数量级提高时,两种算法所耗时间和空间都没有发生数量级的猛增,体现了算法的稳定性.

4 相关研究工作

构件死锁连接是一种逻辑错误,是构件化软件语义层次上的一个问题,是在构件集成过程中产生的一种错误,也是造成构件化软件系统产生死锁的因素之一.目前,在国内外同行的研究中,还未发现有专门解决该问题的文献,但在软件系统死锁问题研究领域存在一些较有代表性的研究工作.

Christoph[16]通过构造基于构件系统的 3-SAT 公式,每个构件的行为用一个有限标签变换系统表示,整个系统的行为由各连接器的行为和各构件的标签变换系统构成的一个全局标签变换系统表示,执行系统状态空间变换和分析,得出结论为:在基于构件的软件系统中,局部和全局死锁存在性检测都是 NP 难题.之所以成为 NP 难题,是因为检测过程涉及到系统全局状态分析.不过,Chritoph 与其同事在文献[17]中给出一种可在多项式时间内检测完毕的参数化的系统局部无死锁确认算法,但实际可用性较低.其研究针对的死锁检测不区分产生死锁的原因,构件化软件系统中导致死锁的原因有多种类型(如资源共享[18]、强同步约束[18]和多线程同步[19]等),为了追求全面,将各种原因都包括在内而使问题更加复杂,难以解决,又沿状态空间变换和分析途径解决,随着系统复杂度的增大,必然会面临状态空间爆炸问题.构件死锁连接也是导致系统产生死锁各原因中的一种类型,本文只针对这一类型进行研究,缩小了问题范围,便于解决.同时,避开了系统状态空间分析,将问题转化为简单有向图回路查找问题,从而使问题得到较好的解决.

与我们具有相似解决死锁问题方针的文献[20],针对另一种类型的死锁产生原因:即通信关系不当导致系统产生死锁,给出了检测方案,并提供了动态死锁测试工具(DDTTool),虽然其研究直接针对的是并发多任务的 ADA 程序,而非构件化软件,但运行于分布式环境中的构件化软件同样具有并发多任务特征,故其解决方案亦可以借鉴.另外,文献[19]针对因多线程同步不当而导致系统死锁给出解决方法,并提供了该类死锁的检测工具(C-checker),而其研究直接针对的是共享主存并行程序 OpenMP.另外,文献[21]给出一种基于代数变换方法,从软件设计形式规约出发,可推导系统中是否存在因构件间通信数据依赖和缓冲区对称访问等而引起的死锁.该方法虽然理论上不受软件系统复杂性的限制,但其复杂的推理公式和变换规则,没有自动推导工具支持,难以付诸实践.而文献[22]则建立了一种并发计算数据流模型,构件间采用数据托肯(token)流进行通信,通过为构件设计一种因果依赖接口,静态分析构件间是否存在因数据依赖环而产生的死锁.

对于基于状态空间分析方法检测死锁会遇到的状态空间爆炸问题,文献[23]提出一种基于启发式的搜索技术,但并不能彻底解决该问题.不过其方法可以与其他削减状态空间的方法(如基于状态等价[24,25]、局部模型检测[26])结合使用.此外,文献[27]在所建立的状态缩减理论(IOT-failure 和 IOT-state 等价理论)的基础上,也给出 7 条基本的启发式规则,用于削减状态空间.

在死锁避免方面,文献[2,18]从软件构造的角度提出一个基于构件的建模框架,指导构件正确组装,可用于从无死锁的构件构造无死锁的软件系统,并给出了无死锁系统中构件应满足的充要条件,以及由无死锁构件构造无死锁系统应满足的充要条件.文献[28]用进程代数 PA(process algebra)形式化地描述构件间的交互行为协议,提供一组交互行为协议相容性检验规则和定理,可用于防止相组装的两个构件因接口行为不一致而产生死锁的情况出现.

5 总结及今后的工作

针对构件化软件中采用基于过程调用交互方式构件连接所存在的构件间死锁连接问题,我们研究提出了一种完整的解决方法.建立了基于过程调用连接器形式语义模型 call-based connector,给出了 TPDCC 算法和 DCEMRP 算法以及消除死锁连接的设计约束规则.应用该解决方法设计实现了 DCCTool 子工具,证明该方法是可行的;10 组测试实例和福利注册系统在 DCCTool 中检测执行的结果表明,所给出的解决方法能够正确找到构件连接中存在的所有死锁连接,并能准确给出消除所有死锁连接需要消除的最小数目连接的位置.此外,构件间

的死锁连接问题属于语义层次上的问题,目前的软件开发工具在编译链接应用系统时只能检查出语法问题,无法检查出存在的构件死锁连接.因此,本文的研究工作为构件连接存在的死锁连接问题提供了一种可行而有效的解决方法,同时,为增强软件的稳定性和可靠性提供了一种途径.

在解决问题的过程中,抽象出的所有 call-based connector 保存在特定的 XML 格式文件中.这种处理方式,一方面将构件与连接关系显式分离,另一方面,将连接关系描述为与构件具有同等地位的独立实体.在设计实现软件时可以采用这种处理方式,这样可以使构件间的连接关系易于修改和扩展,从而支持构件间柔性集成,在构件连接层次上为适应性软件设计提供新思路.Call-Based Connector 目前主要描述出了构件间的交互协议以及数据类型和传递约定,在此基础上可增加支持构件动态连接的操作,以方便构件交互关系的定制和调整以及构件的替换,为设计实现适应性连接器奠定基础.这些是可继续深入研究的工作,也正是我们下一步的研究工作.而可继续开展的扩展性研究包括,设计实现相容性检查工具,以辅助进行正确的构件集成和支持构件连接可配置软件的配置部分的正确性检查,因为 call-based connector 语义模型也支持语法层次上的类型匹配检查.

References:

[1] Li G, Jing MZ. Concept, properties and functions of component connection in adaptive software architecture. Computer Engineering, 2003,29(1):265−267 (in Chinese with English abstract).

[2] Gossler G, Sifakis J. Composition for component-based modeling. Science of Computer Programming, 2005,55(1-3):161−183.

[3] Allen RJ. A formal approach to software architecture [Ph.D. Thesis]. Pittsburgh: Carnegie Mellon University, 1997.

[4] Ren HM, Qian LQ. Research on component composition and its formal reasoning. Journal of Software, 2003,14(6):1066−1074 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/14/1066.htm

[5] Xiong HM, Ying S, Yu LJ, Zhang T. A composite reuse of architectural connectors using reflection. Journal of Software, 2006, 17(6):1298−1306 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/17/1298.htm

[6] Wermelinger M, Lopes A, Fiadeiro JL. Superposing connectors. In: Proc. of the 10th Int'l Workshop on Software Specification and Design. San Diego: IEEE Computer Society Press, 2000. 87−94. http://csdl2.computer.org/persagen/DLAbsToc.jsp?resourcePath=/dl/proceedings/&toc=comp/proceedings/iwssd/2000/0884/00/0884toc.xml&DOI=10.1109/IWSSD.2000.891129

[7] Magee J, Dulay N, Eisenbach S, Kramer J. Specifying distributed software architectures. In: Schafer W, Botella P, eds. Proc. of the 5th European Software Engineering Conf. Berlin: Springer-Verlag, 1995. 137−153.

[8] Milner R, Parrow J, Walder D. A calculus of mobile processes. Information and Computation, 1992,100(1):1−40.

[9] Oquendo F. π-ADL: An architecture description language based on the higher-order typed π-calculus for specifying dynamic and mobile software architectures. ACM Software Engineering Notes, 2004,29(3):1−14.

[10] Li CY, Li GS, He PJ. A formal dynamic architecture description language. Journal of Software, 2006,17(6):1349−1359 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/17/1349.htm

[11] Cîmpan S, Leymonerie F, Oquendo F. Handling dynamic behaviour in software architectures. In: Morrison R, Oquendo F, eds. Proc. of the 2nd European Workshop on Software Architectures. LNCS 3527, Berlin: Springer-Verlag, 2005. 77−93.

[12] Allen R, Garlan D. A formal basis for architectural connection. ACM Trans. on Software Engineering and Methodology, 1997,6(3): 213−249.

[13] Allen R, Douence R, Garlan D. Specifying and analyzing dynamic software architectures. In: Astesiano E, ed. Proc. of the 1st Int'l Conf. on Fundamental Approaches to Software Engineering. LNCS 1382, Berlin: Springer-Verlag, 1998. 21−37.

[14] Hoare CAR. Communicating sequential process. 2004. http://www. usingcsp.com

[15] Matena V, Drishnan S. Apllying Enterprise JavaBeans, Component-Based Development for the J2EE Platform. 2nd ed., Beijing: Pearson Education, Inc., Tsinghua University Press, 2004. 50−207.

[16] Minnameier C. Local and global deadlock-detection in component-based systems are NP-hard. Information Proc. Letters, 2007,103(3):105−111.

[17] Majster-Cederaum M, Martens M, Minnameier C. A polynominal-time checkable sufficient condition for deadlock-freedom of component-based systems. In: van Leeuwen J, ed. Proc. of the 33rd Int'l Conf, on Current Trends in Theory and Practice of Computer Science. LNCS 4362, Berlin: Springer-Verlag, 2007. 888−899.

[18] Gossler G, Sifakis J. Compoent-Based construction of deadlock-free systems. In: Pandya PK, ed. Proc. of the Annual Conf. on Foundations of Software Technology and Theoretical Computer Science. LNCS 2914, Berlin: Springer-Verlag, 2003. 420−433.

[19] Wang ZF, Huang C. Static detection of deadlocks in OpenMP Fortran programs. Journal of Computer Research & Development, 2007,44(3):536−543 (in Chinese with English abstract).

[20] Shi XH, Gao ZY, Shao H. A dynamic deadlock testing method of a concurrent ADA program. Journal of Computer Research & Development, 1999,36(8):954−960 (in Chinese with English abstract).

[21] Compare D, Inveradi P, Wolf AL. Uncovering architectural mismatch in component behavior. Science of Computer Programming, 1999,33(2):101−131.

[22] Ye Z, Lee EA. A causality interface for deadlock analysis in dataflow. In: Proc. of the ACM & IEEE Conf. on Embedded Software. Seoul: ACM Press, 2006. 44−52. http://portal.acm.org/citation.cfm?id=1176887.1176895&coll=GUIDE&dl=ACM&CFID=2029577&CFTOKEN=10062163

[23] Gradara S, Santone A, Villani ML. DELFIN$^+$: An efficient deadlock detection tool for CCS processes. Journal of Computer and System Science, 2006,72(8):1397−1412.

[24] Bouajjani A, Fernandez JC, Halbwachs N. Minimal model generation. In: Proc. of the 2nd Int'l Workshop on Computer-Aided Verification. LNCS 531, Berlin: Springer-Verlag, 1990. 197−203. http://springerlink.lib.tsinghua.edu.cn/content/jntlcg33mv7rrwpd/?p=61814a0ed1d74ecc930363eec0f0292e&pi=4

[25] Graf S, Steffen B, Luttgen G. Compositional minimization of finite state systems using interface specifications. Formal Aspects of Computing, 1996,8(5):1−28.

[26] Stirling C, Walker D. Local model checking in the modal mu-calculus. Theoretical Computer Science, 1991,89(1):161−177.

[27] Tsai JJP, Juan EYT. Model and heuristic technique for efficient verification of component-based software systems. In: Proc. of the 1st Int'l Conf. on Cognitive Informatics. Calgary: IEEE Computer Society Press, 2002. 59−68. http://csdl2.computer.org/persagen/DLAbsToc.jsp?resourcePath=/dl/proceedings/&toc=comp/proceedings/icci/2002/1724/00/1724toc.xml&DOI=10.1109/COGINF.2002.1039283

[28] Hu HY, Lu J, Ma XX, Tao XP. Study on behavioral compatibility of component in software architecture using object-oriented paradigm. Journal of Software, 2006,17(6):1276−1286 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/17/1276.htm

附中文参考文献:

[1] 李刚,金茂忠.适应性软件体系结构中构件连接的概念、特点和作用.计算机工程,2003,29(1):265−267.

[4] 任洪敏,钱乐秋.构件组装及其形式化推导研究.软件学报,2003,14(6):1066−1074. http://www.jos.org.cn/1000-9825/14/1066.htm

[5] 熊惠民,应时,虞莉娟,张韬.基于反射的连接器组合重用方法.软件学报,2006,17(6):1298−1306. http://www.jos.org.cn/1000-9825/17/1298.htm

[10] 李长云,李赣生,何频捷.一种形式化的动态体系结构描述语言.软件学报,2006,17(6):1349−1359. http://www.jos.org.cn/1000-9825/17/1349.htm

[19] 王昭飞,黄春.OpenMP Fortran 程序中死锁的静态检测.计算机研究与发展,2007,44(3):536−543.

[20] 史晓华,高仲仪,邵晖.ADA 程序通信死锁的动态检测方法.计算机研究与发展,1999,36(8):954−960.

[28] 胡海洋,吕建,马晓星,陶先平.面向对象范型体系结构中构件行为相容性研究.软件学报,2006,17(6):1276−1286. http://www.jos.org.cn/1000-9825/17/1276.htm

毛斐巧(1980－),女,湖北老河口人,博士生,主要研究领域为软件工程,分布式计算.

齐德昱(1959－),男,博士,教授,博士生导师,主要研究领域为软件开发模式,网络安全,分布式计算.

林伟伟(1980－),男,博士,讲师,主要研究领域为计算机体系结构,网格计算.