构件系统演化一致性的判定方法

郑交交[1] 李 彤[2] 林 英[1,2] 谢仲文[1,2] 王晓芳[1] 成 蕾[1] 刘 妙[1]
(云南大学软件学院 昆明 650091)[1] (云南大学软件工程重点实验室 昆明 650091)[2]

摘 要 构件系统演化一致性是确保演化操作可靠的必要条件,若一致性得不到满足,则会致使演化后的系统达不到既定的功能目标。针对该问题,文中提出基于接口、流程结构、内部行为的构件系统演化一致性判断方法。首先,在演化后的系统中将每个构件视为判定执行者,使所有的构件协同参与一致性判定过程,从接口和流程结构出发,判断执行者和全局的一致性;其次,在满足接口、流程结构一致性的情况下,判断演化构件在演化前后的内部行为一致性;最后,通过对一个构件实例的完整分析,详细描述了该判定方法,并验证了其可行性。

关键词 构件系统,软件演化,演化一致性,接口一致性,结构一致性
中图法分类号 TP311 **文献标识码** A **DOI** 10.11896/j.issn.1002-137X.2018.10.035

Judgement Method of Evolution Consistency of Component System

ZHENG Jiao-jiao[1] LI Tong[2] LIN Ying[1,2] XIE Zhong-wen[1,2] WANG Xiao-fang[1] CHENG Lei[1] LIU Miao[1]
(College of Software, Yunnan University, Kunming 650091, China)[1]
(Key Laboratory for Software Engineering of Yunnan University, Kunming 650091, China)[2]

Abstract A necessary condition to ensure the reliability of evolution operations is the evolution consistency of component system. If this condition is not satisfied, the evolved system will miss the established functional target. In response to this problem, this paper proposed a judgement method of evolution consistency of component system based on interface, process structure and internal behavior. Firstly, in the evolved system, each component is considered as a judgement executor, so that all the components can participate in the process of consistency judgment collaboratively. Based on the interface and the process structure, the consistency between judgement executors and the global system can be judged. Secondly, in the case of satisfying consistency of interface and process structure, the internal behavior consistency of components before and after evolution is judged. Finally, the complete analysis of a component case is used to describe the judgement method in detail and verify the feasibility of the proposed method.

Keywords Component system, Software evolution, Evolution consistency, Interface consistency, Structural consistency

1 引言

软件系统有两个最基本的特性:构造性和演化性[1-2]。由于运行环境、用户需求等因素不断改变,软件系统需要不断地进行演化来满足新的要求。软件演化方式有静态和动态之分[3]:静态演化是指使整个软件系统停止工作,然后对其进行演化操作,演化完成后再重新启用系统。目前已有比较完备的方案和可行技术来实行静态演化,但这些方案在目前多变的环境中所需的资源和时间成本都太高,因此大家更多地关注于软件的动态演化。在不使系统停止运行的情形下以更小的消耗和损失对系统进行重新配置或更新,这种配置是动态的,在系统运行中对其完成更新,这就是动态演化的目的[4]。

如今,实施软件动态演化时主要面临两方面的挑战[4-5]:1)演化可靠性[5]必须得以保证,在此前提下,动态配置实施的成本要做到尽可能低,即尽可能缩小它对运行中系统的影响范畴;2)演化必须确保实施动态演化前后的系统一致性[6],只有保证了演化前后系统的一致性,演化后的系统才可以替代演化前的系统继续完成工作。在构件系统中,若要降低动态演化的成本,则必须对各个构件之间的交互关系进行分析。系统一致性关系到演化后的系统是否能够满足演化需求,从而继续完成工作,包括内部一致性和外部一致性[4]:内部一致性是指在演化前后的构件系统中演化构件的内部状态保持一

到稿日期:2017-09-27 返修日期:2017-12-09 本文受国家自然科学基金项目(61379032,61662085)资助。
郑交交(1992-),女,硕士生,主要研究方向为系统分析与集成,E-mail:506950231@qq.com;**李 彤**(1963-),男,博士,教授,CCF 会员,主要研究方向为软件工程、软件过程、软件演化等,E-mail:tli@gnu.edn.cn(通信作者);**林 英**(1973-),女,博士,副教授,CCF 会员,主要研究方向为信息安全、软件工程等;**谢仲文**(1982-),男,博士,讲师,CCF 会员,主要研究方向为软件工程、软件过程、软件演化等;**王晓芳**(1993-),女,硕士生,主要研究方向为方向软件工程;**成 蕾**(1993-),女,硕士生,主要研究方向为软件工程理论与方法;**刘 妙**(1993-),女,硕士生,主要研究方向为系统分析与集成。

致，确保系统可以继续运行以完成相应的任务；外部一致性是指在演化后的系统包括演化构件在内的各个构件之间能够正确地进行交互，即交互一致。本文基于构件系统，着重关注演化一致性的内容。

2 相关工作

在软件动态演化和演化一致性研究领域，许多科研人员已经获得了丰富的成果。例如，文献[7]和文献[8]通过在新构件和原构件之间建立联系，主要是替代性和相关性，确保部分的行为一致性，以满足整体系统的一致性，使得新构件的运行与之前的软件环境之间不会产生排斥，但是其没有考虑全局的交互问题。文献[9]和文献[10]都是通过找出演化前后理论模型之间的不一致，然后对这些不一致进行研究和修改，使得最终的演化结果保持一致。不同的是，文献[9]是研究行为概要文件来找到不一致，最后通过改变传播模型使得最终的演化满足一致性；文献[10]是在云领域通过庞大的基数集和经验找出功能模型的不一致，然后对这些不一致进行修改从而达到最终一致。这类方法的实施存在一定难度，且对相关技术和硬件条件的要求相对较高。文献[11]提出了一个基于非功能特征的软件动态演化框架，但该框架演化的准确性和有效性有待提高。

总之，在演化一致性的研究方面依然存在一些问题[12-13]。首先，现有的工作主要是替代性和相关性，没有全面地考虑构件系统演化的判定要素，例如接口、结构等要素。随着需求的变化，需要进行多次的一致性判定，因此通过接口、结构来进行判定可以减少行为一致性判定的消耗。再者，现有的工作通过对新构件和原构件进行一致性判定来完成整个构件系统的一致性判定，这种方法在构件系统的运行中会产生一些交互的时序问题，缺少整体的一致性判定。本文在已有研究的基础上，从构件系统的接口、流程结构和内部行为 3 方面出发，提出适用于演化阶段和设计阶段的一致性判定方法。主要工作如下：

1)针对已有的判定方法只考虑构件行为的一致性判定的问题，选择构件系统的接口、流程结构和内部行为作为演化一致性的判定要素。接口一致是构件间相互交互的基础条件，接口一致包括接口的对称性和消息一致性；流程结构一致是指构件的外部交互流程中包含的状态结构类型一致，共有顺序、选择、并行、循环 4 种结构；内部行为一致是指演化前后构件的内部行为一致，保证各个构件能够按照正确的时序完成消息的发送和接收，在构件系统的运行中若完成所需的功能，则满足一致性。

2)针对现有的一致性判定仅考虑新构件和原构件的一致性问题，本文还选取了演化后系统中的每一个构件成员分别作为判定执行者，将其余构件看作完整个体，判定演化构件与其余构件组合的一致性，使全局行为同时参与整个判定过程，这样可以保证新的构件系统可靠执行，并继续完成工作。

3)最后，通过一个完整的构件系统实例来描述整个一致性的判定过程，验证其是否能够完整执行完成相应的功能以及能否正确终止。

3 构件系统演化一致性判定方法

软件系统并不是由一个模块构成的，而是由多个模块或部分组成，各个部件之间通过接口交互来实现互连互通，使整个系统可以正常运作，从而完成相应的事务。文中提出了一个适用于演化阶段的从接口、流程结构、内部行为 3 方面出发的一致性判定方法。其中，接口和流程结构一致性判定分析的是演化后系统中构件间的交互一致性，内部行为一致性判定分析的是演化构件在演化前后的内部一致性。

首先对软件系统和构件进行定义，然后分析一致性判定方法中各判定要素的判定约束和算法。

定义 1(软件系统) 使用三元组 $system=(name,components,interactive)$ 描述构件系统。其中，$name$ 表示系统的名称标识；$components$ 表示构成系统的所有构件的集合；$interactive$ 表示构件间交互关系的集合。

定义 2(构件) 构件是构件系统的最基本构成单元。使用五元组 $component=(name,method,interface,structure,IM)$ 来表示构件 C。其中，$name$ 表示构件的名称标识；$interface$ 表示构件的外部接口的集合；$structure$ 是构件功能行为流程结构，显示该构件从活动开始到结束的流程；$method$ 表示构件功能实现的方法的集合。$f_a(method)$ 表示方法包含的动作；$f_m(method)$ 表示方法接收或发送的消息；$f_c(component_\alpha,component_\beta)$ 表示构件系统中 $component_\alpha$ 和 $component_\beta$ 的交互路径，为不同构件之间的交互路径都赋予一个特定的标识，用 $f_p(method)$ 表示。$IM:interface\rightarrow method$ 是一个关联函数，表示在一构件中，它的某个功能与其某个接口相关联。众所周知，接口是暴露在用户面前的，这就表明该功能会在外面有显现，是可以被观察到的。

3.1 接口一致性判定

定义 3(构件接口) 构件接口是由一组抽象方法和相关的抽象消息组成的，使用三元组 $interface=(name,message,method)$ 来表示。其中，$name$ 表示接口的名称标识；$message$ 表示接口的消息；$method$ 表示接口的方法。端口的方法包括接收和发送两种。

构件的接口一致包括进行交互的构件之间的接口操作对称和接口消息类型一致。在进行演化的构件系统中，构件间存在哪些交互是非常清楚的，这使得构件的接口一致性的判断很容易完成。其中，接口操作对称性就是指在演化构件和与其有交互的其他构件之间存在着一组对称的接口操作，它保证了若有构件发送消息就一定会有构件接收消息；接口的消息类型一致是指接口之间传递的消息是一致的，它保证了构件间能够正确地收到消息，包括消息内部结构一致和数据类型一致。

约束 1 接口的操作对称性是指演化后的系统中，各个成员构件与其余构件进行交互通信的动作匹配性，即是否能保证当一方发送消息时另一方能够正确接收到消息，具体内容应满足以下条件：

1)演化构件 $component_\alpha$ 的方法集合 $method_\alpha$ 中的每一个

操作在其与构件构成的组合 $component_\beta$ 的方法集合 $method_\beta$ 中都有对称的操作;

2)其他构件构成的组合 $component_\beta$ 的方法集合 $method_\beta$ 中的每个操作在演化构件 $component_\alpha$ 的方法集合 $method_\alpha$ 中都有对称操作。

接口的动作对称性判定算法(算法 1)首先遍历演化构件 $component_\alpha$ 方法集合中的每个操作,判定操作 m 的交互路径是否与 $component_\alpha$ 和 $component_\beta$ 的交互路径相同,若不同则表明此操作不是由 $component_\alpha$ 和 $component_\beta$ 交互所产生的,若相同则遍历 $component_\beta$ 方法集合的每个操作;判定操作 n 使用的交互路径是否与 $component_\alpha$ 和 $component_\beta$ 的交互路径相同,若不相同则当前操作不是由 $component_\alpha$ 和 $component_\beta$ 的交互所产生的,若相同则判断操作 m 与操作 n 的动作是否对称,若不对称则返回 false,若对称则继续执行遍历,直至所有操作遍历结束。

算法 1　InterfaceSym($component_\alpha$,$component_\beta$)

输入:构件 $component_\alpha$ 和 $component_\beta$

输出:布尔值

```
1. for each x∈method_α do
2.   if f_p(x)=f_c(component_α,component_β) then
3.     for each y∈method_β do
4.       if f_p(y)=f_c(component_α,component_β) then
5.         if f_a(x)=f_a(y) then
6.           执行第 3 步,遍历下一个 y
7.         else f_a(x)≠f_a(y) then
8.           return false
9.         end if
10.      else f_p(y)≠f_c(component_α,component_β) then
11.          执行第 3 步,遍历下一个 x
12.        end if
13.  else f_p(y)≠f_c(component_α,component_β) then
14.    执行第 1 步,遍历下一个 x
15.  end if
16. end for
17. return true
```

同样地,调用算法 1 分析 InterfaceSym($component_\beta$,$component_\alpha$),若两次输出均为 true,则接口对称。

定义 4(消息)　定义消息为一个四元组,即 $Message=(name,type,number,childMes)$。其中,$name$ 表示消息的名称标识;$type$ 表示消息的类型,有 complex 和 simple 两种,其中 simple 为基本数据类型;$number$ 表示消息包含的子元素的个数;$childMes$ 表示消息内容(simple 类型)或子元素的集合(complex 类型)。

约束 2　接口消息类型一致性应满足以下条件:

1)如果 $Message$ 的类型 $type$ 为 simple,则其个数 $number$ 为 0;

2)当两个消息都为简单消息时,若 $name$ 相同,$type$ 一致,$number$ 相等,则这两个消息一致;

3)当两个消息都为复杂消息时,若 $name$ 相同,$type$ 一致,$number$ 相等,$childMes$ 相同,则这两个消息一致。

消息类型一致性判定算法(算法 2)首先判断 $message_\alpha$ 和 $message_\beta$ 的名称是否一致,若不一致就返回 false,若一致则判断两个消息的类型是否一致。接下来的不一致情况均返回 false,若一致则判断 $message_\alpha$ 属于哪种类型,若为 simple,此时遍历判断两者包含的基本数据类型是否一致,若一致则返回 true;若为 complex,则判断 $message_\alpha$ 和 $message_\beta$ 包含的子元素个数是否一致,类型一致则递归检查子元素的类型是否一致,若一致则返回 true。

算法 2　MessageCon($Message_\alpha$,$Message_\beta$)

输入:由演化构件 $component_\alpha$ 发送或接收的消息 $message_\alpha$,其他构件构成的组合 $component_\beta$ 发送或接收的消息 $message_\beta$

输出:布尔值

```
1. 初始赋值:i=0
2. if message_α.name=message_β.name then
3.   if message_α.type=message_β.type then
4.     if message_α.type=simple then
5.       if message_α.childMes=message_β.childMes then
6.         return true
7.       else
8.         return false
9.       end if
10.    else message_α.number=message_β.number then
11.      while (i<GetChildMes(message_α).number) do
12.        if GetChildMes(message_α)[i].number=GetChildMes
           (message_β)[i].number
13.          if MessageCon(GetChildMes(message_α)[i],GetChi-
             ldMes(message_β)[i])=true then
14.            执行第 11 步,i++
15.            else MessageCon(GetChildMes(message_α)[i],Get-
               ChildMes(message_β)[i])=false then
16.              return false
17.            end if
18.            else GetChildMes(message_α)[i].number≠GetChi-
               ldMes(message_β)[i].number then
19.              return false
20.          end if
21.        end while
22.        return true
23.    else message_α.number≠message_β.number then
24.      return false
25.    end if
26.  else message_α.type≠message_β.type then
27.    return false
28.  else message_α.name≠message_β.name then
29. return false
```

通过以上算法即可判断出接口的消息类型是否一致。

3.2　流程结构一致性判定

整个构件系统的可观察的所有业务流程中可能出现的状态的关系集合就是该构件系统的流程结构图,也即 UML 中的状态图。一个构件系统由不同的构件交互而成,因此构件系统的流程结构是基于每一个构件的流程结构的。其中,结

构类型共有 4 种，分别是顺序结构、选择结构、并行结构和循环结构。流程结构一致性保证了包括演化构件在内的各个构件之间可以很好地进行交互，协调合作构成了一个完整的构件系统，并完成相应的功能。下面以一个订单管理构件 $component_O$ 为例来表示它的状态图，如图 1 所示。

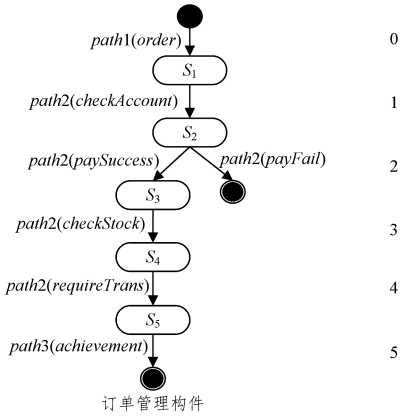


图 1 $component_O$ 的状态图

Fig. 1 State diagram of $component_O$

构件状态的改变都是由消息触发的。层数表示业务流程的进行顺序。状态路径是指一个状态到另一个状态的路径，有私有路径和交互路径之分。

顺序结构：从某一层的某一个节点出发，只有唯一一条状态迁移路径。如图 1 中的 $S_4 \xrightarrow{path2(requireTrans)} S_5$，状态节点 S_4 到状态节点 S_5 只有唯一一条迁移路径。

选择结构：从某一层的某一个节点出发，在同一条交互路径上有多条状态迁移路径。如图 1 中的 $S_2 \xrightarrow{path2(paySuccess)} S_3$ 和 $S_2 \xrightarrow{path2(payFail)} 0$ 就属于选择结构。

并行结构：从某一层的某一条节点出发，在不止一条交互路径上有状态迁移路径，表示该构件分别与其他不同的构件发生了交互。

循环结构：在某一层，到达循环条件的状态迁移就是循环结构，当满足条件时，该节点状态会循环执行，直至不满足条件。

定义 5(流程结构) 构件的流程结构就是所有状态的关系集合，即状态图。定义其为一个三元组 $structure=(length,type,constru)$。其中，$length$ 表示状态图的深度；$type$ 表示状态结构类型，$structure[i].type$ 表示某一层的状态结构类型；$constru$ 表示状态图中的状态结构，$structure[i].constru$ 表示某一层的状态结构，由具体状态 $node$ 和状态路径 $path$ 构成，$path$ 有私有路径和交互路径之分。

约束 3 结构一致性是指在构件系统的整个业务流程中，演化构件的流程结构是否与其他构件的流程结构一致，也即对于演化构件 $component_\alpha$ 的 $structure[i].type$，其对外的整个构件系统 $component_\beta$ 有与它相同的 $structure[i].type$，同时对于整个构件系统 $component_\beta$ 的 $structure[i].type$，包括演化构件 $component_\alpha$ 在内的所有组成构件存在与其相同的 $structure[i].type$。需要满足以下条件：

1)每层的状态结构 $structure[i]$ 组成了整个流程结构 $structure$；

2)只关注对外交互的状态路径 $path$，若状态结构的路径不属于对外交互路径，则判断下一个状态路径；

3)若对构件 $component_1$ 第 i 层的状态结构类型 $structure1[i].type$，在构件 $component_2$ 中有与其相同的状态结构类型 $structure2[n].type$；若对 $component_2$ 第 j 层的状态结构 $structure2[j].type$，在 $component_1$ 中存在与其相同的状态结构类型 $structure1[m].type$，完成所有遍历后，$component_1$ 和 $component_2$ 的流程结构一致。

结构一致性判定算法(算法 3)首先获取演化构件 $component_\alpha$ 和其他构件构成的组合 $component_\beta$ 的流程结构类型，则遍历演化构件 $component_\alpha$ 的每一层，若第 i 层状态结构为空，则表示不可见，继续遍历下一层；若第 i 层状态结构不为空，则遍历 $component_\beta$ 的每一层，若存在相同类型的状态结构，则停止遍历 $component_\beta$，继续比较 $component_\alpha$ 的下一层，若遍历完 $component_\beta$ 的所有层也没有找到同类型的状态结构，则返回 false，若执行结束，输出 true，表示流程结构一致。该算法首先遍历 $component$ 的状态路径，观察其是否为对外交互路径，若不是，则该层状态结构不可见，若是则判断它属于哪种类型，若同时存在多种类型，则遍历完所有层后返回其流程结构。

算法 3 StructCon($component_\alpha$,$component_\beta$)

输入：演化构件 $component_\alpha$ 与其他构件构成的组合 $component_\beta$

输出：布尔值

```
1. 初始化：result=true,i=0,j=0
2. structure1=GetStructure(component_α)
3. structure2=GetStructure(component_β)
4. for each i≤structure1.length
5.     if structure1≠null then
6.     for each j≤structure2.length
7.         if structure2[j]=structure1[i] then
8.             转到第 4 步,i++
9.         else structure2[j]≠structure1[i] then
10.            转到第 6 步,j++
11.        end if
12.    end for
13.    return false
14.    else structure1=null then
15.        转到第 4 步,i++
16.    end if
17. return true
//得到构件的流程结构
GetStructure(component)
输入：构件 component
输出：component.structure
1. 初始化：structure[i].type=∅
2. for each i≤component.structure.length
3.     for path in component.structure.constru
4.         if path=交互 path then
```

```
5.        if 分支数=1 then
6.          if 循环条件=null then
7.            structure[i].type→"顺序结构"
8.          else if 循环条件≠null then
9.            structure[i].type→"循环结构"
10.         end if
11.       else if 分支数＞1 && 路径相同 then
12.         structure[i].type→"选择结构"
13.       else if 分支数＞1 && 路径不相同 then
14.         if structure[i].type≠Ø
15.           structure[i].type→"并行结构"
16.         end if
17.       end if
18.       else path≠交互 patt then
19.     转到步骤 3，path++
20.   end for
21. end for
22. return component.structure
```

在该算法中，$structure[i]$是流程结构的第 i 层，$component.structure.constru$ 是构件 $component$ 的状态结构。

3.3　内部行为一致性判定

基于前文的接口一致性和结构一致性判定，接下来的内部行为一致性判断就要简单很多。内部行为一致是指演化构件的内部行为在演化前后保持一致，通过比较演化前后构件的内部行为来判定演化后是否能够完成演化前的功能，使整个系统可以很好地照正确的秩序进行交互，从而完成工作。

定义 6（构件的内部行为）　构件 $component_i$ 的一个完整执行可以表示为：$P_m=S_0\xrightarrow[a_1]{}S_1\cdots\xrightarrow[a_{n-1}]{}S_{n-1}\xrightarrow[a_n]{}S_n$，$S_0$ 是其开始状态，S_n 是其结束状态。一个构件的所有完整执行就是它的所有内部行为，用 $function_{all}(component_i)$ 表示，$function_{all}(component_i)=(P_1,P_2,\cdots,P_{m-1},P_m)$。

约束 4　构件演化的内部行为一致性满足以下条件：

1）内部行为一致性的参与者为演化前后的构件 $component_i$ 和 $component_i'$；

2）若构件 $component_i$ 演化为 $component_i'$，则当且仅当 $function_{all}(component_i)\subseteq function_{all}(component_i')$，满足内部行为一致性。

功能行为一致性判定算法（算法 4）首先遍历演化前后的构件 $component_i$ 和 $component_i'$，分别得到其所有功能行为 $function_{all}(component_i)$和 $function_{all}(component_i')$，然后对两者进行比较，若前者包含于后者，则返回 true，否则返回 false。

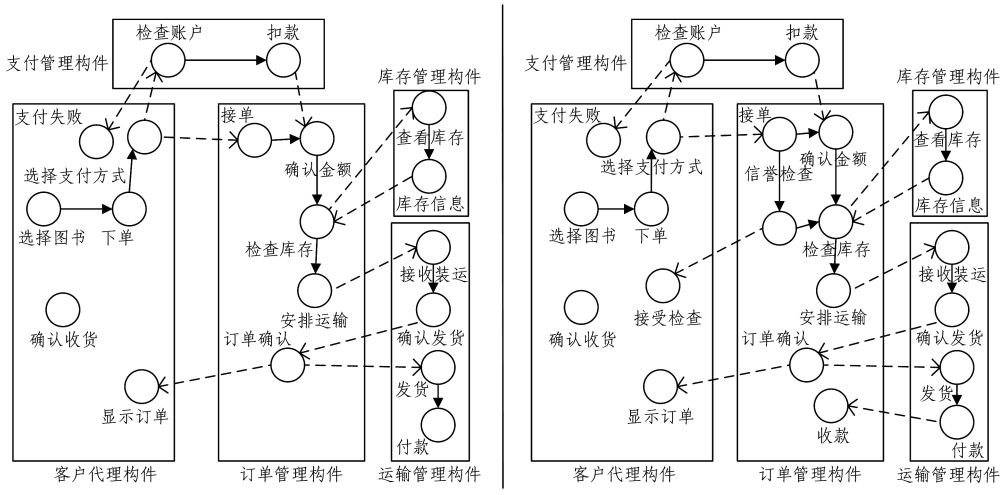
算法 4　FunctionCon($component_i$，$component_i'$)

```
输入：演化前后的构件 component_i 和 component_i'
输出：布尔值
1. 初始化：result=true，i=0，j=0
2. function_all 1=GetFun(component_i)
3. function_all 2=GetFun(component_i')
4.    if function_all 1.length ≤function_all 2.length then
5.      for each i≤function_all 1.length
6.        if function_all 1[i]≠null then
7.          for each j≤function_all 2.length
8.            if function_all 2[j]=function_all 1[i] then
9.              执行第 5 步，i++
10.           else function_all 2[j]≠function_all 1[i] then
11.             执行第 7 步，j++
12.           end if
13.         end for
14.         return false
15.       else function_all 1[i]=null then
16.         执行第 5 步，i++
17.       end if
18.     end for
19.     return true
20.   else function_all 1.length ＞function_all 2.length then
21.     return false
22.   end if
```

其中，$GetFun(component_i)$是获取构件的所有内部行为的算法，参考图的深度优先遍历算法，从初始状态 S_0 开始，层层深入，直至最后一个没有状态后继的状态结束，如此循环，直到得到构件的所有完整执行 P_m，即 $function_{all}(component_i)$。相关算法可参考文献[14]。

4　实例分析

以一个图书订购的构件系统为实例，验证所提构件系统演化一致性判定方法的可操作性。该系统由 5 个构件组成，分别为客户代理构件 $component_C$、订单管理构件 $component_O$、库存管理构件 $component_S$、运输管理构件 $component_T$ 和支付管理构件 $component_P$。顾客浏览并选择要订购的图书，然后下单，选择支付方式，再将订单信息传递给订单管理构件 $component_O$，同时通过支付管理构件 $component_P$ 向相应的第三方支付系统（支付宝、银行、微信等）发送请求付款的消息；第三方支付系统首先会检查账户，确认无误后扣款，若账户或支付出现问题（账户余额不足、密码错误等），则返回支付失败的消息，若支付一切顺利，则将支付成功的信息发送给订单管理构件 $component_O$；订单管理构件 $component_O$ 接到扣款成功的消息后，会调用库存管理构件 $component_S$ 来检查库存情况，检查完 $component_S$ 后，构件会向 $component_O$ 构件返回库存信息；然后 $component_O$ 构件向运输管理构件 $component_T$ 请求运送图书；构件 $component_T$ 收到消息后向 $component_O$ 构件确认发货单，此时 $component_O$ 构件调用 $component_C$ 构件向顾客显示最终订单；顾客确认订单后，$component_O$ 构件将此消息传递给 $component_T$ 构件；最终，顾客确认收货后，交易完成。为应对新需求的出现，在原有的先网上支付再交易的基础上增加了基于消费者信誉度检查的先消费再付款的模式。这里，订单管理构件 $component_O$ 就是演化构件，在其内部增加了信誉度检查和收款的功能。演化前后的构件系统如图 2 所示。通过分析此次演化活动，确认演化构件为订单管理构件 $component_O$，相关构件有客户代理构件 $component_C$、库存管理构件 $component_S$、运输管理构件 $component_T$ 和支付管理构件 $component_P$。



(a)演化前的构件系统 (b)演化后的构件系统

图 2 演化前后的构件系统

Fig. 2 Component system before and after evolution

4.1 接口一致性判定分析

分别以演化构件 $component_O$ 以及相关构件 $component_C$，$component_S$ 和 $component_T$ 为判定活动的执行者，并判断它们与其他构件组成的整体 $component_{others}$ 的接口动作对称性。此时，将 $component_{others}$ 中的构件间的交互看作内部行为。以订单管理构件 $component_O$ 为例，根据约束 1 和算法 1 可得到：构件 $component_O$ 的方法集合 $method_{component_O}$ 中的每一个接口动作在其他构件组成的整体 $component_{others}$ 的方法集合 $method_{component_{others}}$ 中都有对称的动作；同样地，其他构件组成的整体 $component_{others}$ 的方法集合 $method_{component_{others}}$ 中的每一个接口动作在构件 $component_O$ 的方法集合 $method_{component_O}$ 中也都有对称动作。同理，可以判定构件 $component_C$，$component_S$ 和 $component_T$ 的接口动作的对称性。根据约束 2 和算法 2 可知：此次演化发生后，由于交互消息并没有发生变化，因此构件 $component_O$ 和 $component_{others}$ 之间的交互消息与演化前相同，保持了接口消息类型的一致性。由以上分析可得，此次演化保证了接口一致性。

4.2 结构一致性判定分析

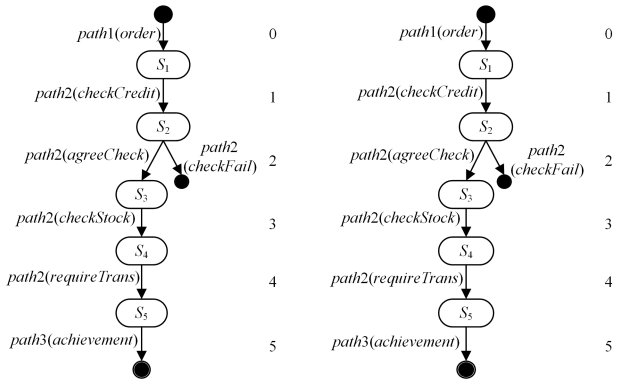
对于结构一致性判定，首先给出演化构件 $component_O$ 与 $component_{others}$ 的对外交互 UML 状态图，如图 3 所示。

以订单管理构件 $component_O$ 为例，依据约束 3 和算法 3 可得：

1)构件 $component_O$ 的状态结构集合 $structure$ 由 UML 状态图的每层状态结构 $structure[i]$ 组成，其他构件组成的整体 $component_{others}$ 的状态结构集合 $structure$ 由状态图的每层状态结构 $structure[j]$ 组成。

2)由于每一层都属于对外交互路径，没有私有路径，因此每一层都有状态结构且没有并发结构。

3)$component_O$ 的状态结构图中有顺序结构和分支结构，$component_{others}$ 的状态结构图中也只有顺序结构和分支结构，并且所有交互行为都发生在交互路径 $path1(f_c(component_O, component_T))$ 和 $path2(f_c(component_O, component_S))$ 上。



(a)订单管理构件 (b)其他构件组合

图 3 $component_O$ 与 $component_{others}$ 的对外交互状态图

Fig. 3 External interaction state diagram of $component_O$ and $component_{others}$

由以上信息可知，构件 $component_O$ 与 $component_{others}$ 的状态结构一致。使用同样的方法可以证明 $component_T$，$component_S$ 和 $component_C$ 的状态结构与 $component_{others}$ 一致，即此次演化后的结构满足一致性。

4.3 内部行为一致性判定分析

以演化构件 $component_O$ 为例，此次演化发生了增加行为，根据约束 4 和算法 4 可知：

判定执行者为 $component_O$ 和 $compenent_O'$，分别获取演化前后的订单管理构件 $component_O$ 和 $component_O'$ 的所有内部行为：

$function_{all}(component_O) = \{(orderReceiving, confirmMoney, checkStock, requireTrans), confirmOrder\}$，$function_{all}(component_O') = \{(orderReceiving, confirmMoney, checkStock, requireTrans), (orderReceiving, checkCredit, checkStock, requireTrans), confirmOrder, collectMoney\}$

根据约束 4 和算法 4 可得，$function_{all}(component_O) \subseteq function_{all}(component_O')$，即此次演化满足行为一致性。

综上，通过构件接口、流程结构和内部行为一致性 3 方面

进行演化一致性判断可知：演化前后的构件系统在构件接口、状态结构和内部行为 3 方面都满足一致性，即演化前后的系统是一致的。

结束语　在构件演化过程中，演化一致性一直是研究的重点之一，本文从接口、流程结构和内部行为 3 个判定要素出发，提出了构件演化一致性判定流程。已有方法中，文献[15]提出一种从行为角度采用层次式时间自动机对软件在线演化进行分析来判定一致性的方法，其支持对软件的时间属性和层次特征等直接建模；Miladi 等应用统一建模语言(UML)建立了构件和连接件的添加和删除规则[16]；Kacem 等扩展了 UML，通过动态元类建立 SA 的添加、删除规则和操作[17]；文献[18]基于进程代数描述构件间的交互行为，归纳交互行为一致性的定义，给出交互行为一致性所需满足的约束条件，提出一种保证交互行为一致性的方法。以上方法直接通过行为来判定一致性，对任意大小的行为变化都需要比较整体的行为，这种操作重复而又冗长。本文从 3 个要素出发判定演化一致性的方法层次分明、操作简便明确，极大地降低了时间消耗和资源消耗；且本文不再单纯地考虑新构件和原构件在行为上的替换；此外，还着重考虑了在演化后的构件系统中的各构件之间能否正确传递消息，排除了产生冗余消息的概率；将每一个构件都作为判定执行者，保证了系统全局协同参与工作，排除了交互时序问题；最后，基于一个完整构件系统实例来描述判定过程，证实了该判定方法的可行性。

本文从 3 个方面出发提出了构件系统演化一致性判定方法，主要考虑了演化前后构件系统是否一致的问题，没有涉及到演化过程的内容，下一步将对演化过程进行研究分析。

参 考 文 献

[1] LI C Y, LI Y, WU J, et al. A Service-Oriented Software Model Supporting Dynamic Evolution[J]. Chinese Journal of Computers, 2006, 29(7): 1020-1028. (in Chinese)
李长云，李莹，吴健，等. 一个面向服务的支持动态演化的软件模型[J]. 计算机学报，2006，29(7)：1020-1028.

[2] MOKNI A, URTADO C, VAUTTIER S, et al. A formal approach for managing component-based architecture evolution [J]. Science of Computer Programming, 2016, 127: 24-29.

[3] QING G, JUN L, YINGFEI X, et al. High-confidence software evolution [J]. Science China (Information Sciences), 2016, 59(7): 94-112.

[4] XIE Z W, MING L, LIN Y, et al. The consistency analysis of software dynamic evolution based on Petri net[J]. Computer Science, 2016, 43(11): 234-241. (in Chinese)
谢仲文，明利，林英，等. 基于 Petri 网的软件动态演化的一致性分析[J]. 计算机科学，2016，43(11)：234-241.

[5] ZHANG J, LEI H. Research on the reliability evolution of network software[J]. Journal of Southwest Jiaotong University, 2014, 49(2): 310-316. (in Chinese)
张靖，雷航. 网构软件可靠性演化计算研究[J]. 西南交通大学学报，2014，49(2)：310-316.

[6] JIANMEI G, YINGLIN W, TRINIDAD P, et al. Consistency maintenance for evolving feature models[J]. Expert Systems With Applications, 2011, 39(5): 4987-4998.

[7] LYTRA I, TRAN H, ZDUN U. Constraint-Based Consistency Checking between Design Decisions and Component Models for Supporting Software Architecture Evolution[C]// European Conference on Software Maintenance & Reengineering. IEEE, 2012: 287-296.

[8] RAMOS R, SAMPAIO A, MOTA A. Conformance notions for the coordination of interaction components[J]. Science of Computer Programming, 2010, 75(5): 350-373.

[9] GURUNATHAM C H, RAJARAJESWARI P, VASUMATHI D D. Consistency Evolution of Process models based on Structural Analysis and Behavioral Profiles[J]. International Journal of Modern Engineering Research, 2012, 4(2): 2568-2573.

[10] QUINTON C, PLEUSS C, BERRE D L, et al. Consistency Checking for the Evolution of Cardinality-based Feature Models[C]// International Software Product Line Conference. 2014: 122-131.

[11] WANG L, PENG X, ZHAO W Y. Framework for Software Dynamic Adaptation Evolution Based on Non-Functional Features [J]. Computer Engineering, 2008, 34(24): 74-76. (in Chinese)
王雷，彭鑫，赵文耘. 基于非功能性特征的软件动态自演化框架[J]. 计算机工程，2008，34(24)：74-76.

[12] ZHOU Y, GE J D, ZHANG P C, et al. Model based verification of dynamically evolvable service oriented systems[J]. Science China(Information Sciences), 2016, 59(3): 5-21.

[13] MA Y T, HE K Q, LI B, et al. Empirical Study on the Characteristics of Complex Networks in Networked Software[J]. Journal of Software, 2011, 22(3): 381-407. (in Chinese)
马于涛，何克清，李兵，等. 网络化软件的复杂网络特性实证[J]. 软件学报，2011，22(3)：381-407.

[14] LI L M, WANG Z J, TANG L Y. Research on the Testing of Component Functional Behavior[J]. Journal of Chinese Computer Systems, 2010, 31(4): 686-690. (in Chinese)
李良明，王志坚，唐龙业. 构件功能行为测试的研究[J]. 小型微型计算机系统，2010，31(4)：686-690.

[15] ZHOU Y, HUANG Y K, HUANG Z Q, et al. Towards an Approach of Consistency Verification for Online Software Evolution in Open Environments [J]. Journal of Software, 2015, 26(4): 747-759. (in Chinese)
周宇，黄延凯，黄志球，等. 一种开放环境下软件在线演化一致性验证方法[J]. 软件学报，2015，26(4)：747-759.

[16] MILADI M N, JMAIEL M, KACEM M H. A UML profile and a FUJABA plugin for modelling dynarnic software architectures [C]// Proceedings of the Workshop on Model-Driven Software Evolution. Washington: IEEE Press, 2007: 20-26.

[17] KACEM M H, KACEM A H, JMAIEL M, et al. Describing dynamic software architectures using an extended UML model [C]// ACM the Symposium on Applied Computing. New York: ACM Press, 2006: 1245-1249.

[18] QI X Y, WANG T, MA C. Research on interactive behavior consistency of Component Evolution[J]. Computer Engineering, 2010, 36(24): 51-53. (in Chinese)
祁晓园，王涛，马川. 构件演化的交互行为一致性研究[J]. 计算机工程，2010，36(24)：51-53.