基于构件组合运算的SA可演化性度量*)

黄万艮[1,2]　陈松乔[1]

(中南大学信息科学与工程学院　长沙 410083)[1]　(湖南大学会计学院信息管理系　长沙 410079)[2]

摘　要　在满足当前需求的众多软件体系结构(SA)中,选择适应未来发展变化的(SA),需要度量软件体系结构可演化性SAE。本文提出了基于构件组合运算的SA;从构件组合运算和SA两个层次分析了构件操作的波及效应,提出了一种新的SAE度量方法,设计了度量算法;在给出的实例中,度量了同一系统的不同SA的可演化性。最后,根据领域软件工程的特点,利用特征迹,对SAE的度量方法进行了改进。该度量方法克服了传统方法需要大量数据、过度依赖人工参与和个人经验、只适合小规模系统等不足。

关键词　构件组合运算,软件体系结构,构件操作,可演化性度量

Measuring Software Architecture Evolution Based on Component Combination Operations

HUANG Wan-Gen[1,2]　CHENG Song-Qiao[1]

(College of Information Science and Engineering, Central South University, Changsha 410083)[1]

(Information Department of Accounting College Hunan University, Changsha 410079)[2]

Abstract　In order to choose a software architecture SA which will fit with changes in the future from those satisfy with current requires, software architecture evolution SAE measure is needed. A concept of SA based on component combination operations is put forward. An analysis approach of ripple-effect of component operations is described based on component combination operations and SA, a new metric of SAE is developed, and an algorithm about SAE is designed. As an example, the evolution of different SAs about the same system is measured . Finally, the metric of SAE is improved using feature traces according to the domain software engineering. Traditional metric deficiency of too much data needed, excessive dependence on manpower participating in and individual experience, and only adaptation to small size systems, is overcome.

Keywords　Component combination operations, Software architecture, Component operations, Measuring evolution

SA是在高抽象层次上对软件系统结构的一种描述。使用SA不但可以提高软件质量,减少软件开销,提高软件生产率,而且决定了一个组织如何实现其业务目标。一个结构可能允许或限制增加用户要求的特征,允许或阻止修改,促进或约束构件重用,因此需要对SA可演化性进行度量,以方便在满足当前需求的众多SA中,选择支持未来发展变化的SA[11]。软件演化是软件生命周期中一个完整的组成部分,开发可演化软件,对软件可演化性进行度量,已经成为在IT界具有最高优先级的问题[1,7]。利用调用复杂性关系,对软件代码的可演化性进行了度量,认为函数调用关系越复杂,一个变化导致的潜在的波及效应就越大,理解代码就需要花费更多的时间,代码演化也就越难维护[12],但这种度量方法基于从具体代码获得的表示函数调用关系的优势树,是一种事后的度量方法。使用基于SA的可达矩阵,度量每一个构件的"贡献"[4],但这种方法没有明确这种"贡献"如何影响SA。其他一些SA分析评估方法,在实际应用中却很难发挥作用,因为有的方法需要较高的数学背景和大量数据,且只适用于较小规模的系统;有的方法基本通过手工完成,且过度依赖参与者的个人经验,方法本身仅仅给出了分析的流程,缺乏行之有效的支撑工具[9]。

本文根据构件组合运算,定义了SA;根据SA要素变化,定义了SA演化;根据影响SA要素的构件操作、构件演化与构件组合运算的关系,对SA可演化性进行了度量,并给出了相关算法和实例。最后,使用领域工程,确定基本需求和变化需求,利用特征和特征迹,对体系结构可演化性的度量方法进行了改进。

由于通过消息传递来实现构件之间的交互更可靠,构件更独立,一个构件可以不感知其他构件的存在和变化,更有利于系统的演化。而且,构件之间通过调用进行的交互可以转化为通过消息传递来激发构件功能的执行。构件在运行时产生的一些事件,可以转换为消息。所以我们所采用的SA是基于消息的。

本文组织如下:第1部分讨论了基于构件组合运算的SA;第2部分对SA演化进行探讨,第3部分对SA可演化性进行度量,最后对全文进行总结。

1　基于构件组合运算的SA

1.1　构件与构件组合

虽然构件还没有统一的定义,但为了讨论方便,根据Guozhen Tan[4]和ZHANG[14],给出如下定义:

定义1　构件是一个数据单元或一个计算单元,由构件接口集合和构件实现模块组成。一个接口可以定义为一个八元组,即Interface=〈RMsg, SMsg, Beha, Cons, NonFunc〉。其中,RMsg表示构件接口接收的消息集合,SMsg表示构件

*)博士点基金(20030533011)。**黄万艮**　博士生,主要研究方向:软件工程、信息安全;**陈松乔**　教授、博士生导师,主要研究方向:软件工程等。

接口发送的消息；Beha 描述构件接口行为语义；Cons 定义对构件接口行为约束；NonFunc 对构件接口非功能属性进行说明。

连接件是用于连接两个或多个构件的构件，负责消息约束性条件检查和构件之间的消息路由。根据 Medvidovic N[5]，给出如下定义：

定义 2 一个连接件可以定义为一个五元组，即 Connector=〈topIn, topOut, bottIn, bottOut, MsgRout, Cons, NonFunc〉，其中：topIn 表示连接件接受来自上部构件的消息；topOut 表示连接件发送给上部构件的消息；bottIn 表示连接件接受来自底部构件的消息；BottOut 表示连接件发送给底部构件的消息；msgRout 是一个消息路由表，说明 topIn 由 bottOut 传递到哪一个构件，bottIn 由 topOut 传递还是由 bottOut 传递。Cons 和 NonFunc 的意义与构件中的意义相同。

由连接件把一个或多个构件连接在一起，称为构件组合。设 c_1 和 c_2 由连接件 conn 连接成为一个组合构件 c，$c.RMsg$ 由 $c_1.RMsg$ 和 $c_2.RMsg$ 以及 c_1 和 c_2 的关系决定，$c.SMsg$ 由 $c_1.SMsg$ 和 $c_2.SMsg$ 以及 c_1 和 c_2 的关系决定。conn 不仅负责 c_1 和 c_2 的交互，而且还负责 c 与其他构件的交互。c_1 和 c_2 的交互不仅与 $c_1.RMsg$、$c_1.SMsg$、$c_2.RMsg$ 和 $c_2.SMsg$ 相关，而且与 c_1 和 c_2 的关系有关。

1.2 构件组合运算

ZHAO[2] 等给出了种激励和使用两种组合运算，ZHANG[6] 提出了条件、使用和协作三种组合运算，REN[3] 等提出了顺序、连接、选择、复制、中断和并行等六种组合机制。为了描述 SA，本文讨论顺序、选择、使用、并行、中断和循环六种组合运算。

定义 3 设 c 是 c_1 和 c_2 的组合，如果外界只能先执行 c_1 的功能，然后执行 c_2 的功能，并且 $c.RMsg=c_1.RMsg\cup c_2.RMsg$，$c.SMsg=c_1.SMsg\cup c_2.SMsg$，则称 c_1 和 c_2 进行了一次顺序组合运算，用 $c=c_1+c_2$ 表示。

顺序组合运算满足结合律，即对于构件 c_1、c_2 和 c_3 有 $(c_1+c_2)+c_3=c_1+(c_2+c_3)$。另外，构件顺序组合运算可以推广到有限个构件的情形，即：$c_1+c_2+\cdots+c_i+\cdots+c_n$，其意义是，只有在执行了构件 c_{i-1} 的功能之后，才能执行构件 c_i 的功能，执行了构件 c_i 的功能之后，不能再执行构件 c_{i-1} 的功能，其中 $1<i\leqslant n$。

定义 4 设 c 是 c_1 和 c_2 的组合，如果外界只有通过 c_1 的才能使用 c_2 的功能；并且 $c.RMsg=c_1.RMsg$，$c.SMsg=(c_1.SMsg-c_2.RMsg)\cup c_2.SMsg$，则称 c_1 和 c_2 进行了一次使用组合运算，用 $c=c_1\oplus c_2$ 表示。

使用组合运算不满足结合律，即 $(c_1\oplus c_2)\oplus c_3\neq c_1\oplus(c_2\oplus c_3)$。另外，使用组合运算可以推广到有限个构件的情形，即：$c_1\oplus c_2\oplus\cdots\oplus c_i\oplus\cdots\oplus c_n$，其意义是，外界只能通过构件 c_{i-1} 的才能使用构件 c_i 的功能，$1<i\leqslant n$。显然，顺序运算和使用组合运算都不满足交换律。

定义 5 设 c 是 c_1 和 c_2 的组合，如果外界只能使用 c_1 的功能或 c_2 的功能，不能同时使用 c_1 和 c_2 的功能；并且 $c.RMsg=c_1.RMsg\cup c_2.RMsg$，$c.SMsg=c_1.SMsg\cup c_2.SMsg$，则称 c_1 和 c_2 进行了一次使用选择运算，用 $c=c\odot c_2$ 表示。

选择组合运算满足结合律 $(c_1\odot c_2)\odot c_3=c_1\odot(c_2\odot c_3)$ 和交换律 $c_1\odot c_2=c_2\odot c_1$。另外，选择组合运算可以推广到有限个构件的情形，即：$c_1\odot c_2\odot\cdots\odot c_i\odot\cdots\odot c_n$，其意义是，外界依次只能使用一个构件 c_i 的功能，$1\leqslant i\leqslant n$。

定义 6 设 c 是 c_1 和 c_2 的组合，如果首先开始 c_1 的执行，一旦 c_2 开始执行，c_1 暂停执行，直到 c_2 执行完毕并收到继续执行的消息后才能继续执行；并且 $c.RMsg=c_1.RMsg\cup c_2.RMsg$，$c.SMsg=c_1.SMsg\cup c_2.SMsg$，则称 c_1 和 c_2 进行了一次中断组合运算，用 $c=c_1\wedge c_2$ 表示。

中断组合运算满足结合律 $(c_1\wedge c_2)\wedge c_3=c_1\wedge(c_2\wedge c_3)$，但不满足交换律，即 $c_1\wedge c_2\neq c_2\wedge c_1$。另外，中断组合运算可以推广到有限个构件情形，即：$c_1\wedge c_2\wedge\cdots\wedge c_n$，其意义是，构件 c_{i-1} 被构件 c_i 中断，$1<i\leqslant n$。

定义 7 设 c 是 c_1 和 c_2 的组合，如果外界即可以独立使用 c_1 的功能和 c_2 的功能，还可以同时使用 c_1 和 c_2 的功能；并且 $c.RMsg=c_1.RMsg\cup c_2.RMsg$，$c.SMsg=c_1.SMsg\cup c_2.SMsg$，则称 c_1 和 c_2 进行了一次并行组合运算，用 $c=c_1\parallel c_2$ 表示。

并行组合运算满足结合律 $(c_1\parallel c_2)\parallel c_3=c_1\parallel(c_2\parallel c_3)$ 和交换律 $c_1\parallel c_2=c_2\parallel c_1$。另外，并行组合运算可以推广到有限个构件情形，即 $c_1\parallel c_2\parallel\cdots\parallel c_n$。

定义 8 设 c 是 c_1 的组合，如果外界提供一个条件表达式 e 和 c_1 的功能使用序列，c 就能反复执行这个功能序列直到 e 的值为假；并且 $c.RMsg=c_1.RMsg$，$c.SMsg=c_1.SMsg$，则称 c_1 进行了一次循环组合运算，用 $c=\mathbb{R}c_1$ 表示。显然循环组合运算是一个单目运算。

定理 1 使用顺序、选择、循环、中断和并行四种运算可以描述两个构件之间的任何关系。

证明：设 c_1 和 c_2 是任意两个构件，它们分别与两个进程 p_1 和 p_2 相对应，并假定每一个进程有就绪、运行和休止三种状态，则：(1) $p_1.state=run\wedge p_2.state=run$，即 p_1 和 p_2 具有并行关系；(2) $p_1.state=run\wedge p_2.state=ready$，即 p_1 可中断 p_2；(3) $p_1.state=run\wedge p_2.state=close$，如果 p_2 未运行且将来也不运行，则 p_1 和 p_2 具有选择关系；否则 p_1 和 p_2 具有顺序关系。所以使用顺序、选择、中断和并行四种运算可以描述两个构件之间的任何关系。

1.3 构件组合运算的图形表示

如图 1 所示，(a)表示顺序组合运算 c_1+c_2，组合中的两个组成构件区分左和右(因为顺序合不满足交换律)，(b)表示使用组合运算 $c_1\oplus c_2$，组合中的两个组成构件区分左和右；(c)表示选择组合运算 $c_1\odot c_2$；(d)表示并行组合运算 $c_1\parallel c_2$；(e)表示中断组合运算 $c_1\wedge c_2$，组合中的两个组成构件区分左和右；(f)表示循环组合运算 $\mathbb{R}$。

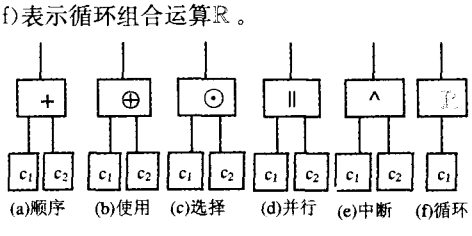

图 1 六种构件组合运算图

不同的连接件可能有相同的图形形式，如 $c_1\odot c_2$ 和 $c_3\odot c_4$ 中的 $\odot$ 连接件，它们的图形形式相同，但它们的意义是不同的，所以应视为不同的连接件，并把这些用运算符表示的连接件称为运算符连接件。

1.4 基于构件组合运算的 SA

SA包括构成系统的构件、规范构件间交互行为和约束关系的连接件、构件和连接件构成的连通图结构，反映了系统的总体结构。在基于消息的系统中，构件之间的关系隐含在连接件中，所以我们定义：

定义9 SA由构件、组合运算及其形成的拓扑结构构成。设Coms为构件集合，Conns为组合运算符连接件集合，于是SA=(Coms, Conns)。

因为任何一种复杂的SA都可以用并行、连接、管线和嵌套四种元结构组合而成，而这四种元结构又可以使用：顺序、分支、循环、嵌入式组件和并发这5类运算来描述[10]，其中嵌入式组件包括功能调用和递归，而递归可用循环、分支、顺序和调用来模拟，并发包括中断和并行，所以利用本文所讨论的六种组合运算可以描述任意的SA。其实，根据定理1，顺序、选择、中断和并行四种运算就可以描述任意的体系结构，但为了方便描述SA，我们定义了六种运算，在实际应用中，还可以定义新的运算，所以定义9中的连接件的类型不局限于本文所讨论的六种。

定义10 设sa为一给定SA，称$gsa=(V,E)$为sa的图，其中$V=sa.\text{Coms}\cup sa.\text{Conns}$，$E=\{\langle v_1,v_2\rangle \mid v_1\in V,\ v_2\in sa.\text{Conns}$且$v_1$挂接在$v_2$上；或者$v_1\in sa.\text{Conns}$，$v_2\in V$且$v_2$挂接在$v_1$上$\}$。

sa是经过构件的组合运算得到的，所以它也是一个组合构件，这不仅揭示了构件是体系结构的组成成分而SA又是构件的递归语义关系，也说明了基于构件组合运算的体系结构具有层次性。我们把最顶层的连接件称为SA图的根，用saRoot表示。挂接在连接件conn上的构件称为conn的孩子，其中最左边的孩子用first()表示，其右边的兄弟用next()表示。使用下列方法访问conn的每一个孩子：

for(child = conn. first(); child! = NULL; child = child. next())visit(child);

2 SA演化

2.1 构件演化

定义11[2] 设c_1和c_2是两个构件，且

(1)$c_2.\text{RMsg}\supseteq c_1.\text{RMsg}$

(2)$c_2.\text{SMsg}\subseteq c_1.\text{SMsg}$

(3)$c_2.\text{Beha}\Rightarrow c_1.\text{Beha}$

(4)$(c_2.\text{Cons}=c_1.\text{Cons})\vee(c_2.\text{Cons}\Rightarrow c_1.\text{Cons})$

(5)$(c_2.\text{Non-Func}=c_1.\text{Non-Func})\vee(c_2.\text{Non-Func}\Rightarrow c_1.\text{Non-Func})$

如果上述条件被满足，则称c_2是c_1的一个演化，记为$c_1\propto c_2$。

构件的演化不仅意味着演化构件可以替换体系结构中相应构件，而且味着演化构件的功能增强和非功能属性得到改善(例如结构更简单、更安全，运行效率更高等)。

2.2 SA演化

SA要素包括构件、连接件、端口、角色、角色之间的胶连、端口与角色之间的连接、子构件与父构件之间的端口绑定[8]。当且仅当这些要素变化时，SA才会发生变化，因此，我们定义：

定义12 SA要素的变化称为SA演化。

SA要素的变化可以分为两类：构件的变化和构件之间关系的变化。导致SA演化的构件变化主要是指构件接口的变化。构件之间关系的变化有两类：一类是直接改变构件之间的关系，例如$(c_1+c_3)\odot(c_2+c_3)\propto(c_1\odot c_2)+c_3$(由于SA也是构件，所以SA的演化也是构件演化)；另一类是构件接口的改变，导致构件之间的关系改变。对于前一类变化，我们已有一些研究结果，本文不再讨论；在后一类变化中，根据基于消息的软件系统的特点，可以把角色之间的胶连、端口与角色之间的连接以及子构件与父构件之间的端口绑定都归纳到连接件中。因此，可以简单地把SA要素变化视为构件变化和连接件变化。

我们将通过构件操作导致构件接口变化和连接件的变化来讨论SA演化，并对演化进行度量。

2.3 导致SA演化的操作

导致SA演化的操作有：插入、删除、替换和修改[8]。为了保持SA演化后"风格"不变，这些操作都是基于连接件的，即在一个运算符连接件上插入、删除和替换一个构件，并假定替换操作使构件的接口发生了变化。由于可以把修改视为一种特殊的替换，所以本文不讨论修改操作。

定义13 设$c_1=(\text{Coms},\text{op})$是一个由组合运算op连接Coms中的构件而成的组合构件，

(1)若$c_2=(\text{Coms}\cup\{b\},op)\wedge c_1\propto c_2$，则称通过一次插入操作，$c_1$演化为$c_2$。其中$b$是一个构件且$b\notin\text{Coms}$。

(2)若$c_2=(\text{Coms}-\{a\},Op)\wedge c_2\propto c_1$，则称通过一次删除操作，$c_2$演化为$c_1$。其中$a\in\text{Coms}$。

(3)若$\exists a\in\text{Coms}$，$a\propto b$，$c_2=((\text{Coms}-\{a\})\cup\{b\},Op)\wedge c_1\propto c_2$，则称通过一次替换操作，$c_1$演化为$c_2$。其中$b$是一个构件$b\notin\text{Coms}$。

插入、删除或替换一个构件c不仅导致其所直接挂接的连接件cnn的改变，而且可能进一步导致cnn所直接挂接的连接件的改变，这种波及效应还可能继续下去[15]。

3 SA可演化性度量

从前面的讨论可知，在SA中，对一个构件进行操作将波及到多个连接件，导致多个连接件的改变。显然，如果需要改变的连接件数目越少，则SA的可演化性就越好；如果需要改变的连接件数目越多，则SA的可演化性越差。因此，我们有理由认为一个SA的可演化性与对任意一个构件进行操作后需要改变的连接件数有关。为了度量SA的可演化性，必须解决三个问题：构件操作的影响波及到何处终止？如何计算波及到的连接件数目？这个数目与SA可演化性的具体关系是什么？

3.1 构件操作对构件组合运算的影响

设$c=c_1\oplus c_2$，由于$c.\text{RMsg}=c_1.\text{RMsg}$，$c.\text{SMsg}=(c_1.\text{SMsg}-c_2.\text{RMsg})\cup c_2.\text{SMsg}$，所以在使用关系中，$c_1$和$c_2$的改变所产生的波及效应是不同的。设$c_2$改变$c'_2$为，则$c_2\propto c'_2\wedge c'=c_1\oplus c'_2\wedge c\propto c'$。$c'.\text{RMsg}=c.\text{RMsg}=c_1.\text{RMsg}$，$c\propto c'\Rightarrow c'.\text{SMsg}\subseteq c.\text{SMsg}$，所以$c_2$的改变不会向$c$外传播，从而只需要修改$c_1$和$c_2$之间的连接件($c_2$的改变意味着将对外提供更多的功能，这时只有增加$c_1$的需求，$c_2$改变所提供的更多的功能才具有实际意义)。$c_1$的改变使$c.\text{RMsg}$发生变化，其影响将向$c$外传播，加上$c_1$的改变使$c_1.\text{SMsg}$也发生了变化，所以不仅需要修改$c_1$和$c_2$之间的连接件，而且还要修改$c$的变化所波及的连接件。

在使用关系$c_1\oplus c_3$中间插入一个构件c_2变为$c_1\oplus c_2\oplus c_3$，将导致一个连接件的变化($c_1\oplus c_2\oplus c_3$是指c_1、c_2和c_3从左到右，依次挂接在一个使用连接件上)。在使用关系$c_1\oplus c_2$

$\oplus c_3$ 中间删除一个构件 c_2 变为 $c_1 \oplus c_3$，也将导致一个连接件的变化。在 $c_1 \oplus c_2$ 中删除 c_2 后，需要修改 c_1 和 c_2 之间的连接件，最后变成 c_1。在 $c_1 \oplus c_2$ 中删除 c_1，等于删除 $c_1 \oplus c_2$，变化波及 $c_1 \oplus c_2$ 的上级连接件。

因此，如果对使用连接件 cnn 的第一个孩子进行操作，则需要修改 cnn，并且影响将向上传播；如果对使用连接件 cnn 的其他孩子进行操作，则只需要修改。

设 $c = c_1 \odot c_2$，由于 $c.RMsg = c_1.RMsg \cup c_2.RMsg$，$c.RMsg = c_1.SMsg \cup c_2.SMsg$，所以在选择组合关系中，$c_1$ 和 c_2 的改变，所导致的连接件的改变情况是相同的。由于 c_1 的改变使 $c.RMsg$ 和 $c.SMsg$ 都发生了变化，所以不仅需要修改 c_1 和 c_2 之间的连接件，而且还要修改 $c.RMsg$ 和 $c.SMsg$ 变化所波及的连接件。在 $c_1 \odot c_3$ 中间插入 c_2 变为 $c_1 \odot c_2 \odot c_3$，在 $c_1 \odot c_2 \odot c_3$ 中间删除 c_2 变为 $c_1 \odot c_3$，在 $c_1 \odot c_2$ 中删除 c_2 变为 $c_1 \odot$NULL，都将导致多个连接件的变化。

在并行和中断关系中构件变化对连接件的影响与选择关系中构件变化的影响一样，每一个构件的改变都涉及多个连接件的修改。综上所述，可以得到如下结论：

定理 2 对构件进行操作的影响，或者波及到其上层的使用连接件，或者波及到 SA 的顶层。对使用连接件 cnn 的非最左边的孩子进行操作，只对 cnn 本身产生影响。

3.2 SA 可演化性度量

我们用 $evol_{ins}(cnn)$ 表示在连接件 cnn 上插入一个构件后需要修改的连接件数目，$evol_{del}(c)$ 表示删除构件 c 后需要修改的连接件数目，$evol_{rep}(c)$ 表示替换构件 c 后需要修改的连接件数目。

定义 14 设软件体系结构 sa=(Coms,Conns)，Coms $=\{c_i \mid 1 \leqslant i \leqslant n\}$，Conns $=\{cnn_i \mid 1 \leqslant i \leqslant k\}$，$sa$ 的可插入性 $evol_{ins}(sa)$、可删除性 $evol_{del}(sa)$、可替换性 $evol_{rep}(sa)$ 和可演化性 $evol(sa)$ 分别为：

$$evol_{ins}(sa)=\frac{(evol_{ins}(cnn_1)+evol_{ins}(cnn_2)+\cdots+evol_{ins}(cnn_k))}{k}$$

$$evol_{del}(sa)=(evol_{del}(c_1)+evol_{del}(c_2)+\cdots+evol_{del}(c_n))/n$$

$$evol_{rep}(sa)=(evol_{rep}(c_1)+evol_{rep}(c_2)+\cdots+evol_{rep}(c_n))/n$$

$$evol(sa)=(evol_{ins}(sa)+evol_{del}(sa)+evol_{rep}(sa))/3$$

显然，$evol_{ins}(sa)$，$evol_{del}(sa)$，$evol_{rep}(sa)$，$evol(sa) \geqslant 1$，并且，$evol_{ins}(sa)$ 的值越小，sa 就越具有可插入性，即插入一个构件需要修改的连接件就越少；$evol_{del}(sa)$ 的值越小，sa 就越具有可删除性，即删除一个构件需要修改的连接件就越少；$evol_{rep}(sa)$ 的值越小，sa 就越具有可替换性，即替换一个构件需要修改的连接件就越少；$evol(sa)$ 的值越小，sa 就越容易演化。

3.3 SA 可演化性度量算法

设 saRoot 为基于构件运算的体系结构 sa 图的顶层连接件，每一个结点的 number 数据成员表示该结点变化所导致的连接件变化数；isConnector() 为结点的成员函数，当结点为连接件时返回真，否则返回假；first() 返回结点最左边的孩子，next 返回结点的右兄弟，stack 是一个栈，成员函数 isEmpty() 用于判断栈是否空[13]。下面的算法遵循 C++ 语言规范。用构件的 number 数据成员记录构件变化将要波及的连接件数目。

算法 1 对 *SA* 图中结点进行初始化。把连接件结点的 *number* 数据成员初始化为 1，构件的 *number* 数据成员初始化为 0。算法具体如下：

```
stack.push(saRoot);
while(! stack.isEmpty())
{node=stack.pop();
if(node.isConnector())
    node.number=1;//连接件的 number 初始化为 1
else
    node.number=0;// 构件的 number 初始化为 0
for (tmp= node.first();tmp! =NULL;tmp=tmp.next())
   stack.push(tmp);
}
```

算法 2 修改 *SA* 图中结点的 *number* 数据成员。如果接点 *child* 的父接点 *parent* 是⊕类型的连接件，且 *child* 不是 *parent* 最左边的孩子，则 *child.number*++；否则 *child.number*+= *parent.number*。具体算法如下：

```
stack.push(saRoot);
while(! stack.isEmpty())
{parent=stack.pop();
for(child= parent.first ();child! =NULL;
    child=child .next())
{ stack.push(tmp);
//child 不是⊕类型的连接件 parent 最左孩子
  if(parent.type=='⊕'||child! =node.first())
    child.number++;
else
    child.number+=parent.number;
  }
}
```

算法 3 计算 *evolInsertSA*，*evolDelSA*，*evolRepSA*，*evolSA*。*evolInsertSA* 为树中所有连接件结点的 *number* 数据成员之和，*evolDelSA*，和 *evolRepSA* 为树中所有构件结点的 *number* 数据成员之和，则 *evolSA*=(*evolInsertSA*/*k* * *k*+*evolDelSA*/*n* * *n*+*evolRepSA*/*n* * *n*)/3。具体算法如下：

```
evolInsertSA=0;evolDelSA=0;evolRepSA=0;
stack.push(saRoot);
while(! stack.isEmpty())
{ node=stack.pop();
    if(node.isConnector())
      evolInsertSA+=node.number;
    else
    { evolDelSA+=node.number;
     evolRepSA= evolDelSA;}
for(tmp= node.first();tmp! =NULL;
   tmp=tmp.next())stack.push(tmp);
}
```

3.4 实例

在下面系统组合模型图中(如图 2 所示)，(a)描述了：通过图形用户界面(GUI)构件可以使用构件 c_1 或构件 c_2，构件 c_1 先从构件 db 获得数据，然后使用构件 prnt 把数据打印出来。显然，并不能直接从(a)中看出模型所表达的含义。

(b)使用了构件组合运算符号，清楚地描述构件的组合含义，而且还可以使用代数式来描述系统 s=GUI⊕((c_1⊕(db+prnt))⊙(c_2⊕(db+prnt))，但是 SA 不是树型的。此时：

$evol_{ins}(s)=(evol_{ins}(db.+)+evol_{ins}(c_1.\oplus)+evol_{ins}(c_2.\oplus)+evol_{ins}(GUI.(\oplus))/5=(3+3+3+2+1)/5=12/5$，其中 $c_1.\oplus$ 表示 c_1 所挂接的连接件⊕。

$evol_{del}(s)=(evol_{del}(db)+evol_{del}(prnt)+evol_{del}(c_1)+evol_{del}(c_2)+evol_{del}(GUI))/5=(3+3+3+3+1)/5=13/5$，

$evol_{rep}(s)=(evol_{rep}(db)+evol_{rep}(prnt)+evol_{rep}(c_1)+evol_{rep}(c_2)+evol_{rep}(GUI))/5=(3+3+3+3+1)/5=13/5$，

$evol(s)=(evol_{ins}(s)+evol_{del}(s)+evol_{rep}(s))/3=(12/5+13/5+13/5)/3=114/45$。

(c)在(b)的基础上经过构件演化运算，系统的代数式简化为 GUI⊕(($c_1 \odot c_2$)⊕(db+prnt))，而且 SA 是树型的。此时

$evol_{ins}(s)=(2+2+1)/3=5/3$，

$evol_{del}(s)=(2+2+2+2+1)/5=9/5$，

$evol_{rep}(s)=(2+2+2+2+1)/5=9/5$，

所以 $evol(s)=(5/3+9/5+9/5)/3=79/45$。

从上面的计算可以看出，(c)比(b)具有更好的可插入性、可删除性和可替换性，因此(c)比(b)具有更好的可演化性。至于在什么情况下可以认为一个 SA 具有或不具有较好的可演化性，我们将进一步进行实证研究。

3.5 领域 SA 可演化性度量

需求变化是软件发生演化的根本原因，领域工程帮助识别可变需求。需求通常以特征的方式提出，特征迹包括一些方法以及连接这些方法所属构件的连接件[16]。图 3 给出了

(下转第 261 页)

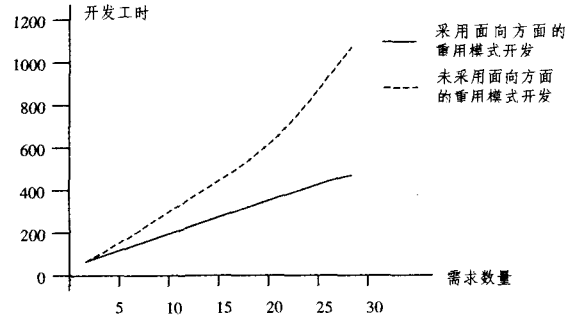

图 4　采用和未采用面向方面的重用模式开发比较

从这个简单的计算中，就可以发现通过采用面向方面的重用模式开发，能够大大增加软件的开发效率，从而达到事半功倍的效果。

参 考 文 献

1　GHEZZI C. 软件工程基础[M]. 北京：清华大学出版社，2003
2　杨雪涛. 基于 UML 的软件开发模型[J]. 重庆工商大学学报(自然科学版)，2004，21(6)：580～582
3　BUDD T. 面向对象的 JAVA 编程思想[M]. 北京：清华大学出版社，2002
4　Kurt Bittner 和 Ian Spence. 用例建模[M]. Addison-Wesley，Boston，2002

(上接第 248 页)

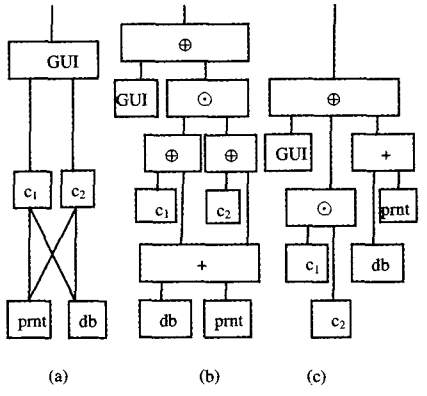

图 2　系统组合模型图

需求、特征、特征迹、方法、构件、连接件和 SA 之间的关系。对于每一个变化的需求，可以根据与特征迹相关连的构件和连接件，特别是增加一个新的需求时，可能需要插入新的构件，特征迹有利于找到这个新的构件应该挂接的连接件。

设 $Req=\{r_1, r_2, \cdots, r_n\}$ 是可变化的需求集合，并假定每一个需求对应唯一一条特征迹。下面给出基于领域的 SA 可演化性度量方法：

(1) $\forall r_i \in Req$，确定特征迹 tr_i，$1 \leqslant i \leqslant n$，

(2)确定 tr_i 经过的构件 $coms=\{c_{i1}, c_{i2}, \cdots, c_{ik1}\}$ 和连接件 $conns=\{conn_{i1}, conn_{i2}, \cdots, conni_{k2}\}$，

(3) $\forall c \in coms$，计算 $evol_{del}(c)$ 和 $evol_{rep}(c)$，$\forall conn \in conns$，计算 $evol_{ins}(conn)$，

(4)计算特征迹 tr_i 的可演化性 $evol(tr_i)$：

tr_i 的可插入性 $evol_{ins}(tr_i)=(evol_{ins}(cnn_{i1})+evol_{ins}(cnn_{i2})+\cdots+evol_{ins}(cnn_{ik2}))/k_{k_2}$，

tr_i 的可删除性 $evol_{del}(tr_i)=(evol_{del}(c_{i1})+evol_{del}(c_{i2})+\cdots+evol_{del}(c_{ik1}))/k_1$，

tr_i 的可修改性 $evol_{rep}(tr_i)=(evol_{rep}(c_{i1})+evol_{rep}(c_{i2})+\cdots+evol_{rep}(c_{ik1}))/k_1$，

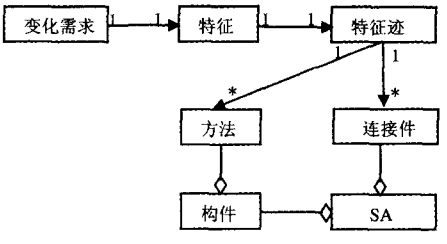

图 3　需求与 SA 和构件的关系模型

tr_i 的可演化性 $evol(tr_i)=(evol_{ins}(tr_i)+evol_{del}(tr_i)+evol_{rep}(tr_i))/3$，

(5)计算软件体系结构 sa 的可演化性 $evol(sa)=(evol(tr_1)+evol(tr_2)+\cdots+evol(tr_n))/n$。

总结　基于构件组合运算的 SA 具有层次性，构件操作对 SA 的影响，或者波及到其上层的使用连接件，或者波及到 SA 的顶层，这为计算构件操作的波及效应提供了可能；基于构件组合运算的构件操作，保证了组合构件演化的一致性。根据构件操作影响到的连接件数目来度量 SA 可演化性，具有较好的直观性；示例中应用度量算法的结果与直观结果一致。根据领域工程，区分基本需求和可变需求，利用特征迹，可以更精确地度量 SA 的可演化性，也可根据 SA 可演化性度量方法来判断 SA 对一次需求变化的适应性。本文假定了构件操作必须保持构件演化特性，并且演化扩展了构件接口，而实际中存在接口不变的情况，这只需要对算法进行适当修改。

参 考 文 献

1　Cook S, He J, Harrison R. Dynamic and static views of software evolution. In: Proc. of the Int'l Conf. of Software Maintenance, IEEE, 2001. 592～601
2　赵会群，王国仁，高远. 软件体系结构抽象模型[J]. 软件学报，2002，25(7)：730～736
3　任洪敏，钱乐秋. 构件组装及其形式化推导研究[J]. 软件学报，2003，14(6)：1077～1074
4　王映辉，张世琨，刘瑜，等. 基于可达矩阵的软件体系结构演化波及效应分析[J]. 软件学报，2004，15(08)：1000～9825
5　Taylor R, Medvidovic N, Anderson K. Acomponent- and message-based architectural style for GUI software[J]. IEEE Transactions on Software Engineering, 1996, 22(6): 390～406
6　张友生. 构件运算与软件演化研究[J]. 计算机应用，2004，4(24)：20～24
7　Mens T, Wermelinger M. Challenges in Software Evolution. In: Proceedings of the 2005 Eighth International WorkshoP on Principles of Software Evolution (IWPSE'05), IEEE, 2005
8　Sadou N, Tamzalit D, Oussalah M. A unified Approach for Software Architecture Evolution at different abstraction levels. In: Proceedings of the 2005 Eighth International Workshop on Principles of Software Evolution (IWPSE'05)
9　梅宏，申峻嵘. 软件体系结构研究进展[J]. 软件学报，2006，6(17)：1257～1275
10　Wang Yingxu. A New Mathematiccal Notation for Describing Notion and Thought in Software Design. In: Proceedings of the First IEEE International Conference on Cognitive Informatics (ICCI'02)
11　Stoermer C, Bachmann F. SACAM: The software architecture comparison analysis method: [Tech Rep]. CMU Software Engineering Institute, CMU/SEI-2003-TR2006 ESC-TR-2003-006, 2003
12　Burd E, Munro M. An Initial Approach towards Measuring and Characterising Software Evolution
13　Shaffer C A. A practical Introduction to Data Structures and Algorithm Analysis. Prentice Hall, 2001
14　张世琨，王立福，常欣，等. 基于层次消息总线的软件体系结构描述语言[J]. 电子学报，2001，5(29)：581～584
15　覃国蓉，张世琨. 基于层次消息总线的软件构架动态模拟和演化研究[J]. 计算机科学，2001，28(93)：75～77
16　Greevy O, Ducasse S. Correlating Features and Code Using A Compact Two-Sided Trace Analysis Approach. In: Proceedings of CSMR 2005 (9th European Conference on Software Maintenance and Reengineering). 314～323