功能转换——软件维护自动化之路

陈绍强 杨放春 陈俊亮

(北京邮电大学程控交换技术与通信网国家重点实验室 北京 100088)

摘要 软件维护的自动化是由维护工具根据维护人员提出的维护需求规范自动进行维护操作.这些维护操作种类很多,但最重要的是功能转换.功能转换涉及到功能识别、功能转换需求规范定义及功能生成.功能自动转换的基础在于对功能这一概念的理论剖析和建模.目前维护自动化基本上是走程序转换的道路,这种方法是建立一个概念模式及转换模式库以提供规则,程序转换就依据这套规则进行.程序转换没有从根本上解决问题,因为维护能力被限制在所提供的概念模式及转换模式中,而实际的维护要求则千变万化.这个矛盾的解决只能通过总结出功能的共性,并借以分析出具体软件系统的功能及各功能之间的约束关系,从而维护人员能针对具体的要求,提出相应的功能转换需求,进行高水平的维护.本文分析了功能概念,并在此基础上提出了功能识别及功能约束所应采取的形式,同时也给出了软件维护自动化的基本过程和需解决的关键技术问题.

关键词 软件维护,功能,程序理解,功能约束,功能转换,波动效应.

软件维护是整个软件生存周期中花费最大的一个阶段,据资料表明,美国软件维护的费用占了GNP的2%.[1]进一步分析维护活动本身的资源和人力分配,我们发现维护人员的47~62%的精力都用在分析、理解维护对象(软件)上.[2]软件理解是一个抽象、复杂的过程,目前我们对这一活动所涉及到的行为还没有完全搞清楚,因此也就对人与机器之间的分工不甚明了;另一方面,软件维护需求定义也是软件维护的薄弱环节,这是因为软件维护需求的提出建立在软件理解的基础之上,维护需求所采取的形式取决于软件理解的输出形式.这2个方面的原因使得软件维护自动化迟迟不能实现.

软件维护与软件开发有密切关系,甚至有人认为维护过程是又一个开发过程.这样说虽然不无道理,但软件维护毕竟是在一原有软件环境下进行,这一约束使得维护人员的创造力无法自由地发挥,他们必须在旧的或过时的系统中工作,即使有所创新,也必须要和原有系统保持一致,恢复这一软件的原设计者们当时的思想才是软件维护人员的主要职责,这即是理解软件的过程.软件理解的结果是导出该软件系统的各个功能及其相互间的关系.如果软件理解过程能自动实现,那么软件维护的自动化也就迎刃而解了.

本文研究得到国家自然科学基金及邮电部青年基金资助.
作者陈绍强,1971年生,博士生,主要研究领域为软件工程,软件维护,通信软件高层设计.杨放春,1957年生,教授,主要研究领域为通信软件,软件支撑环境,智能网.陈俊亮,1933年生,教授,博士导师,中国科学院院士,中国工程院院士,主要研究领域为智能网,通信软件,软件支持环境.

本文通讯联系人:陈绍强,北京 100088,北京邮电大学213信箱

本文1996-03-11收到修改稿

功能体现了软件的意图，理解了每一个功能，理解整个软件也就容易了. 软件开发技术的进步——从非结构化到结构化，再到面向对象——使得软件中的模块和功能有着越来越好的对应关系，软件维护也越来越有效，这也被实践所证明.[5]但另一方面，软件变得越来越复杂，这表现在功能的增多以及功能间关系的复杂化. 软件理解的一部分——程序理解，也就是功能内的更小功能的识别，也是没有完全解决的问题.

所以，功能分解，或者说，功能识别，是软件维护的关键问题. 第 1 节将给出功能的定义，并定义一些操作于功能上的算子. 我们认为，功能是较独立的单元，功能间的交互作用应当很小. 而且，软件中的每一操作都对 1 个或多个功能有所贡献，其表现是影响到该功能的输入输出关系. 这种关系是可逆的，可以通过分析确定的，这就为功能转换提供了可能性.

功能关系框架被建立起来后，维护人员针对功能或功能间的关系提出某种功能改变需求，功能分析工具根据需求中涉及到的输出量，可以分析出针对这些输出量，功能改变负责的操作；进而可以根据操作对功能起作用的方式而决定对组成该功能的诸操作进行改动，从而使维护工作自动化.

我们认为，软件维护的最终目标是高水平的自动化. 目前，国内外虽认识到软件维护的巨大花费及其长期性，但正如文献[1]中所指出的那样，软件维护理论基础还相当薄弱，而软件维护的自动化就更是一个崭新的课题了.

1　功能及其运算

定义 1. 功能 F 是一个函数，即确定了某种输入输出关系. 若 i 是输入，o 是输出，则按一般惯例记作 $o=F(i)$.

例如，一个过程就是一个功能.

定义 2. 功能 F 的输入(即自变量)及输出称为 F 的边界元. 边界元的个数不限. F 的输入元的个数称为 F 的入度，记作 $Indeg(F)$，输出元的个数称为 F 的出度，记作 $Outdeg(F)$.

定义 3. 每个边界元有其相关的标识符和类型，记作 $\langle Id, Type\rangle$，称为该边界元的类别. 记 $INPUT(F)$ 为 F 的输入边界元类别的集合，$OUTPUT(F)$ 为 F 的输出边界元类别的集合. 且把 $INPUT(F)\rightarrow OUTPUT(F)$ 称为 F 的类别，记作 $F: INPUT(F)\rightarrow OUTPUT(F)$.

由定义 2,3，有 $Indeg(F)=\#INPUT(F)$，$Outdeg(F)=\#OUTPUT(F)$. 其中 # 是求集合基数运算.

定义 4. $in=\{\langle Id, Val\rangle \mid \langle Id, Type\rangle \in INPUT(F), Val\in VAL(Type)\}$，称为功能 F 的一个输入，其中 $VAL(Type)$ 表示类型 $Type$ 的值集. 全体输入构成 F 的定义域，记为 $D(F)$. 值域则由 F 确定，记作 $R(F)=\{out \mid \exists\ in\in D(F), out=F(in)\}$.

显然，类别相同的 2 个功能定义域也是相同的.

定义 5. 功能上的几个算子定义如下

(1)换名：把 F 的边界元的标识符换成另一套标识符. 设有换名算子 f，如果 $\langle Id, Type\rangle$ 是 F 的一个边界元的类别，则 $\langle f(Id), Type\rangle$ 就是 $f(F)$ 的相应边界元类别.

这里，功能是指构成整个软件系统的那些子功能.

(2)"+":$F+G$的性质定义如下

$INPUT(F)\cap INPUT(G)=\varnothing$, $OUTPUT(F)\cap OUTPUT(G)=\varnothing$,

$INPUT(F+G)=INPUT(F)\cup INPUT(G)$,

$OUTPUT(F+G)=OUTPUT(F)\cup OUTPUT(G)$,

$D(F+G)=\{in_1\cup in_2 \mid in_1\in D(F), in_2\in D(G)\}$,

$R(F+G)=\{out \mid \exists\ in\in D(F+G), Prj^{out}_{OUTPUT(F)}=F(Prj^{in}_{INPUT(F)}), Prj^{out}_{OUTPUT(G)}=G(Prj^{in}_{INPUT(G)})\}$;

其中$Prjs$是投影操作:设$\tilde{x}=\{\langle Id,Val\rangle\}$, $S=\{\langle Id,Type\rangle\}$,则

$$Prj^{\tilde{x}}_{S}=\{\langle Id,Val\rangle \mid \exists\ Type, \langle Id,Type\rangle\in S\}.$$

(3)";":$F;G$的性质定义如下

$INPUT(F;G)=INPUT(F)\cup INPUT(G)$,

$OUTPUT(F;G)=OUTPUT(F)\cup OUTPUT(G)$,

$D(F;G)=\{in_1\cup in_2 \mid in_1\in D(F), in_2\in D(G)\}$,

$R(F;G)=\{out \mid \exists\ in\in D(F;G), Prj^{out}_{OUTPUT(F)}=F(Prj^{in}_{INPUT(F)}),$

$Prj^{out}_{OUTPUT(G)}=G(Prj^{F(Prj^{in}_{INPUT(F)})}_{INPUT(G)}\cup Prj^{in}_{(INPUT(G)-OUTPUT(F))})\}$

(4)"·":$F\cdot G$的性质

$INTPUT(F\cdot G)\subseteq INPUT(F)\cup INPUT(G)$,

$OUTPUT(F\cdot G)\subseteq OUTPUT(F)\cup OUTPUT(G)$,

$D(F\cdot G)=\{in_1\cup in_2 \mid in_1\in D(F), in_3<in_2, in_2\in D(G)\}$,

$R(F\cdot G)=\{out \mid \exists\ in\in D(F\cdot G), Prj^{out}_{OUTPUT(F)}=F(Prj^{in}_{INPUT(F)})\}, Prj^{out}_{OUTPUT(G)}=G(Prj^{F(Prj^{in}_{INPUT(F)})}_{INPUT(G)}\cup Prj^{in}_{(INPUT(G)-OUTPUT(F))})$.

$F;G$和$F\cdot G$的区别在于类别的定义,事实上,"·"算子隐蔽了F的某些输出元或G的某些输入元,故比起算子";","·"更象是函数的复合.区别";"和"·"是重要的,"·"粘合2个功能,使之看起来就象是一个功能,它将是下文所述的功能识别(功能组合)过程中的一个重要运算;而";"则是功能分解的重要运算.可以说,基本功能处于什么样的粒度水平基本上由这2个算子决定的.

定义 6. 功能之间的几个关系运算定义如下

(1)"=",$F=G$,当且仅当$INPUT(F)=INPUT(f(G))$,且$\forall\ \tilde{x}\in D(F)$, $F(\tilde{x})=f(G)(\tilde{x})$,其中$\tilde{x}$表示输入是一个集合,$f$是一个换名算子(如恒等算子).

(2)"≤",$F\leqslant G$,当且仅当$Indeg(F)=Indeg(G)$,且对每个$\langle Id,Type\rangle\in INPUT(F)$,都有$\langle f(Id),Type'\rangle\in INPUT(G)$,且$VAL(Type)\subseteq VAL(Type')$,即$D(F)\subseteq D(f(G))$,且$\forall\ \tilde{x}\in D(F)$, $F(\tilde{x})=f(G)(\tilde{x})$.

(3)"≈",如果$INPUT(F)=INPUT(f(G))$,且$\exists\ D_1\subseteq D(F)$, $\forall\ \tilde{x}\in D_1$, $F(\tilde{x})=f(G)(\tilde{x})$,则称$F$在$D_1$上相似于$G$,记作$F\approx_{D_1}G$.

显然,"=","≈"是等价关系,而"≤"确定了一个功能间的偏序.

为行文方便,在下文中我们将省略换名算子.

定理 1. 显然地,有

(1) $F+G=G+F$;

(2) 若 $F=G$,则 $F+H=G+H$, $F;H=G;H$, $H;F=H;G$.

定理 2. ";"、"·"满足结合律,即

$$(F;G);H=F;(G;H),(F\cdot G)\cdot H=F\cdot(G\cdot H),$$

$$(F;G)\cdot H=F;(G\cdot H),(F\cdot G);H=F\cdot(G;H).$$

这由程序设计语言的语义可得到解释.

定义 7. 若 $H=F\triangle G$, $\triangle$ 可以是"+"、";"或"·",则称 F、G 为 H 的子功能. 若 $\triangle$ ="·"或";",则称 G 依赖于 F. 说得具体点,就是 G 的输入依赖于 F 的输出.

定义 8. 对 $Out\in OUTPUT(F)$,如果$\exists\ IN\subseteq INPUT(F)$,功能 G,使 $G:IN\to\{Out\}$,则称 G 为 F 的一束,$\#IN$ 称为该束的宽度,记为 $Wide(G)$.

由定义,一个功能 F 的束的个数为 $Outdeg(F)$.

定义 9. 如果 $Indeg(F)=\max\{Wide(G),G$ 是 F 的束$\}$,则称 F 是紧凑的.

在下面代表功能 F 的图 1 中,有 2 个束 $G1$ 和 $G2$,分别用虚线标出. 功能 F 是紧凑的,因为 $Indeg(F)=\max\{Wide(G1),Wide(G2)\}=4$. 而图 2 的 2 个功能就不是紧凑的.

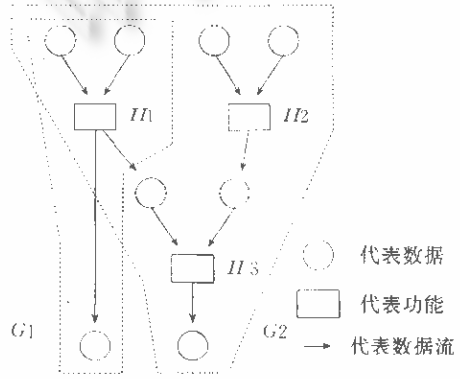

图1 紧凑的功能

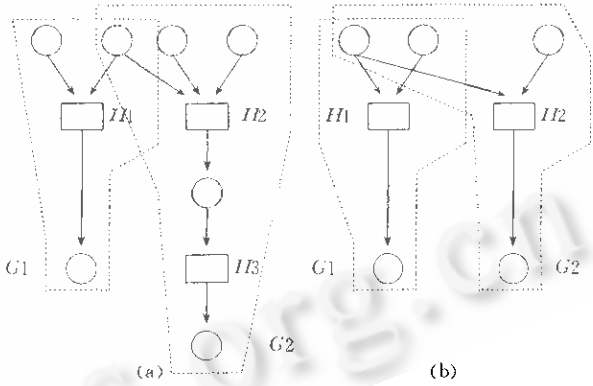

图2 不紧凑的功能

束也许包括 1 个或多个子功能,一个功能的一个束的子功能也许具有对另一个束的子功能的依赖关系,对紧凑的功能来说,尤其如此. 如图 1,有 $G2=(H1'+H2)\cdot H3$, $G1=H1''$,即有 $H3$ 依赖于 $H1$($H1'$ 和 $H1''$分别是 $H1$ 的 2 个束).

定义 10. 如果功能 F 中的束 $G1$ 的子功能对束 $G2$ 的子功能有依赖关系,则称束 $G1$ 对束 $G2$ 有依赖关系.

作为例子,我们来分析一般编程语言中的几个基本语句构件:赋值、条件和循环.

(1) 赋值语句:$A:\tilde{y}=g(\tilde{x})$,这里 A 代表功能.

则 $INPUT(A)=\{Sort(x)\mid x\in\tilde{x}\}$, $OUTPUT(A)=\{Sort(y)\mid y\in\tilde{y}\}$.

$Sort(.)$ 表示边界元的类别.

(2) 条件语句:C: if $E(\tilde{x})$ then P else Q, P、Q 代表功能.

$$INPUT(C)=\{Sort(x)\mid x\in\tilde{x}\}\cup INPUT(P)\cup INPUT(Q),$$

$$OUTPUT(C)=OUTPUT(P)\cup OUTPUT(Q).$$

(3)循环语句:W:while $E(\vec{x})$ do P,

$INPUT(W)=\{Sort(x)\mid x\in\vec{x}\}\bigcup INPUT(P)$, $OUTPUT(W)=OUTPUT(P)$.

2 功能识别

我们以函数的观点来看功能的理由是功能一般具有明晰的接口,即边界元.然而我们在对程序进行分析时,却不能仅靠输入/输出就能识别出具有一定意义的功能.我们希望识别出来的功能可以和问题域中的某个过程对应起来.这是Brooks在文献[4]中提出的软件理解理论的中心思想.

如前述,功能之间具有输入对输出的依赖性.若功能之间具有依赖关系,我们自然认为应当把它们粘合(通过"·"算子)成一个功能,除非这种依赖关系不能被屏蔽,即输入输出是外界可见的.这样,功能识别的原则是:

原则1:识别出来的最小功能(其输入输出外部可见的功能)必须是紧凑的,并且其内部是外部不可见的.

请注意"·"算子屏蔽掉了内部的数据流.显然,这条原则和软件工程中的高内聚、低耦合原则是一致的.这条原则有助于增大识别出来的功能的粒度.

原则2:如果功能H能写成$G+F$,则将H分裂成这2个功能.F和G在功能上毫不相关,它们在一起纯粹出于偶然性.这条原则有助于减小识别出来的功能的粒度.

";"算子则确定了识别出来的功能之间的关系.

定理 3.(功能分解唯一性定理)

一功能F:$INPUT\to OUTPUT$,则F能被唯一分解,即分解成只由"+",";"和"·"算子构成的唯一功能表达式.

定理的证明可由对功能施行结构归纳获得.

这个定理十分重要,它保证了功能识别乃至程序理解是确定性的,将得出唯一结果.于是我们将能够给出一个算法,该算法的任务是确定程序的唯一的功能表达式,从而功能识别甚至程序理解是可以自动化的.

功能识别算法的框架是这样的:

假设功能识别的对象是一段程序P.

(1)首先分析出P的边界元集$Bound$,根据所使用的程序设计语言的语义这是可以办到的;

(2)对$Bound$中的每一输出元,分析出其所在的束,该束的输入元都在$Bound$中;

(3)把具有依赖关系的束合并成为一个功能(用"·"或";"算子),并从该功能中区别出由";"算子结合的功能,它们是外部可见的功能;

(4)把合并后的多个(如果有的话)功能表达式用"+"算子合并成一个功能表达式,即是程序P的功能表达式.

3 功能转换

软件维护建立在程序理解的基础之上.功能分解是程序理解的必要条件,但不充分.程