用于软件维护的集成生存期模型

Stephen S. Yau, Robin A. Nicholl

Jeffrey J.-P. Tsai, Sying-Syang Liu

摘 要

本文提出一个用于软件维护环境的集成生存期模型，它有助于维护人员修改现有的软件系统，为工具自动化提供了基础。该模型用来表示软件系统开发和维护信息，主要表示软件生存期不同阶段间的关系。因为它只表示软件系统某些"基本的"语义性质：控制流、数据流和数据结构，所以与特定的规格说明、设计和程序语言无关。从软件生存期的一个阶段导出另一个阶段的软件开发过程是用图重写规则表示的，从而说明如何实现软件系统的各个成分。这种方法可以分析软件系统在整个生存期中的基本性质。文中举例说明了该模型的使用。

一、 引 言

众所周知，软件维护是大型软件费用高的一个主要因素[1]。为了遏制软件维护的这种高费用，必须有有效的软件维护方法学和有效开发易维护软件。

软件维护是一个十分广泛的活动，其中包括纠错，增强功能，删除过时功能，优化，和适应工作环境的变化。我们已经建立了一个如图1所示的维护过程模型[2,3]。为了完成软件维护过程中的一些必要任务，已经开发了各种技术和工具，如帮助程序理解的图示查询语言（GQL)[4]，方便程序修改的语法制导编辑器、漂亮打印程序和程序分片器（slicer)[5]，以及跟踪波动效应的波动效应分析器[6,7]。这些单个的技术和工具只能实现它们自己既定的狭小目标。此外，由于这些技术和工具的实现是使用现成软件，所以它们不可能以一种集成方式加以使用。

本文介绍大型软件系统在其生存期演化的一个集成模型，此模型旨在帮助实现成本合算的软件维护和易维护软件的开发。这

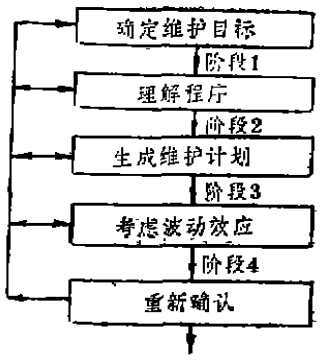


图1 软件维护的一个方法学

个模型在几个抽象级上描述任何软件系统，以便帮助维护程序员理解程序和说明程序修改计划，以及使用各种软件维护工具支持程序分析来完成程序修改。此模型不是介绍给用户的，因此它不同于各种软件开发方法，如象Jackson系统开发[8]、面向对象的开发[9]和变换开发[10]等方法中所使用的其它模型。

上述的所有分析工具[4-7]都以图分析技术为基础，所以我们也使用图和图文法[11,12]上用来表示此模型中软件开发过程的不同部分。尽管此模型并未描述软件系统的所有性质，但都表示了控制流、数据流和数据结构

的重要性质[13]。最重要的是,这些性质是在整个系统生存期与软件开发活动的记录一起进行表示的,而且借助这一开发活动记录,可以从系统的一种表示导出另一种表示,这种信息对于维护程序员是十分有价值的,因为他们有责任修改一个软件系统以便正确地反映按照其规格说明所做的改变。

二、 程序模型化技术

在程序设计环境中,通常是有几个分别定义的技术,每一种技术只对其自己的程序具体表示奏效。每一种程序表示多半适合该程序文本的自动构造,很大程度上与书写程序的语言无关,而且允许有效地使用特别的分析工具。这种表示不大适合作为一种直接用户交互的手段。图2A说明了这种方法,其中单个模型可能包括语法分析树、控制流程图和数据流程图。这一方法的缺点明显地是必须书写大量的特殊程序以便从程序中抽取必要的信息并把这种信息组织成分析工具所需要的各种形式。此外,一种分析工具产生的结果也许很难被任何其它分析工具使用。这就使我们遇到了传统的数据处理问题或数据库问题:我们怎样支持几个相互相关的(分析)程序,而这些程序的数据在概念上是类似的,在逻辑上是相关的,但它们没有共同的结构或表示呢?

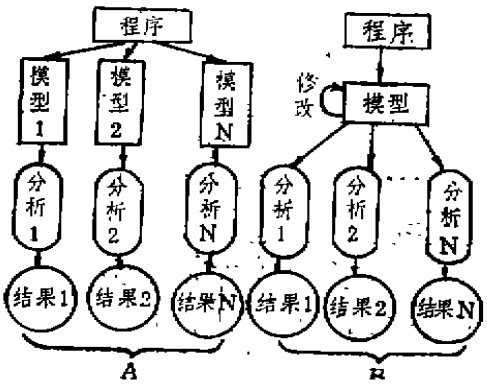


图2 A.专用的工具; B.具有一个共享程序模型的工具

努力克服软件维护中这一困难的结果,导致了一个"通用程序模型"的形成,这个模型可以提供每个单个软件分析工具所需要的信息[14]。在这种情况中,通过一个专门书写的程序,只需要构造一次程序模型。尔后,可以执行分析工具以便直接操纵模型,基本上不用知道程序的源形式。由于程序修改是软件维护的一个重要部分,所以在正在修改程序的同时,能够增量地更新模型亦是重要的。这就产生了如图2B所示的方法,这种方法为集成不同分析工具奠定了基础,提高了实现各种工具有效使用的可能性。

实际上,支持通用程序模型的增量修改以正确反映对程序的任何文本修改这一目标将使模型局限于一种程序设计语言,或者十分类似语言的一个小集合。现成程序设计语言太繁多,一个模型不可能表示所有各种语言的单个性质。一些成功的"集成"程序模型只限于一些特别的程序设计语言,例如INTERLISP[15]、PL/CS[16]和其它一些语言[17]。

三、 问题定义

在许多情况下,软件维护是在很少有或甚至没有资料(文档)可用得上的软件系统上进行的。为了对付这些情况,过去我们已钻研了能够从程序文本自动构造的模型,也钻研了基于这些程序模型的工具和技术。当程序本身是维护人员可用的唯一信息时,要保证修改正确,问题很多。在本文中,我们假定某些附加资料是可用的,并且考虑如何充分地使用这种资料来克服软件维护问题。

在这一节,我们考虑与图1中所描述的一些活动有关的具体问题。首先,我们考虑一下"理解"这个词。在这种情况下,我们指的是,我们将以一种有利于维护程序员阅读的更容易或更自然的形式表示软件系统,并且我们将允许这一表示由软件工具处置。从这个定义,显然看出,我们"理解"软件系统将依赖于我们希望找出有关它的信息类型。

也依赖于我们希望完成的活动类型。在这种特殊应用中，我们期望得到用户的更改请求，那么，我们必须充分地理解软件系统，以便能够正确地改动实际的软件部分，这些部分关系到实现此请求的维护目标。

第二个问题涉及我们只提出修改这一事实。这意味着，我们必须能够初步估计为满足具体的修改请求所需要的工作量。此外，还意味着，当增加了附加工作量时，可以得到这一工作量的更精确估计。由于软件修改的性质，其中作改变所需工作量与决策必须做什么改变和如何改变所需工作量相比是可忽略的，所以仔细分析修改的可能影响将和完成修改本身所花费的费用和时间一样。集成软件生存期模型将方便预期工作量的估计。随着进行更详细的模型分析，估计的精确度将不断增加。

第三个问题关系到软件维护工具的开发。在认可了修改计划之后，我们必须使用各种修改和分析工具来执行软件维护。建立软件生存期阶段内和阶段间基本信息的一种共用数据表示将减少各种维护工具的开发成本。

四、 软件生存期不同阶段的关系

已经描述过的问题，可以归结为确定软件系统的规格说明与构成该系统的程序之间关系的问题。照此，这个问题与软件开发过程的活动和产品密切有关。

传统的软件开发过程表示成一系列（至少三个）不同的活动：规格说明、设计和编码[1]。其中每一个活动（或阶段）都已开发了各自的技术和表示法。一个活动的结束以制备描述系统开发现状的资料为标志。因此，我们把软件开发视为产生一系列资料的过程，每一份资料都是从其前身导出的，并且从其前身出发，在更详细的（实现）程度上描述同一软件系统。

这一活动序列不是以集成方式完成的。尽管设计以规格说明为基础，但是从规格说明实现设计的设计活动还没有记录。如果不太知道规格说明、设计和程序间的一些联系，那么就不可能可靠地把用户的规格说明更改请求转变成相应的程序修改。得不到软件系统的这种信息时，应该说“理解”是困难的，并且会得出结论：对一个计划中的更改，无法得到精确的工作量估计。

幸而，有可能补救这种情形，并且描述软件开发不同阶段间的关系。然而，为达此目的，我们需要所有阶段都兼容的表示。因此，这种表示必须是与语言无关的。此外，由于每个阶段都包含一个高度相互相关的元素集，所以有必要表示不同阶段中一些复杂结构间的关系。最后，如果信息是有用的，那么它必须适当详细地描述一个阶段中的哪些元素和关系与另一阶段中的哪些元素和关系恰好相关。满足这些要求的一个模型如图3中所示的形式，其中也表明了我们的模型使用图和图文法来表示软件开发过程的不同部分。在传统的生存期模型中，实现阶段包含的信息比设计阶段多，设计阶段包含的信息又比要求阶段多。我们的模型保持了相同的特征。因此可以导出一些把要求阶段中的图映射成设计阶段中的图的重写规则和一些把设计阶段中的图映射成实现阶段中的图的重写规则。自动工具也可以用来帮助用户导出阶段间图转换的重写规则。

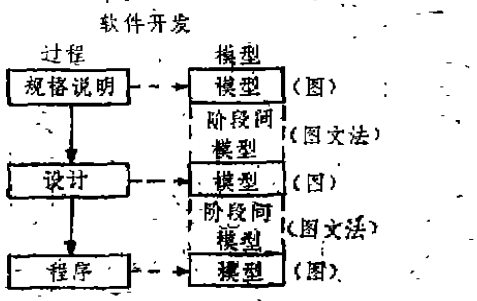


图3 软件生存期模型的形式

使用程序模型，软件生存期模型就可以从相关的软件开发资料自动地构造出来。类

似地，操纵生存期模型的工具可以向其用户隐藏基础模型的细节。尽管我们期望生存期模型应随着软件开发的进行而增量地构造，但是只有开发工作结束后才能构造出来。尽管我们知道如何写一些程序以便从单个开发资料构造模型，但我们不知道如何自动地建造阶段间的一些模型。因此，等待直到开发已经结束所需要的可能费用，取决于记录两个阶段如何相关的细节的难度。如果这些关系上保存的记录不够详细，那么维护人员可能要求花更多时间来研究与计划的改变无关的代码块。维护工作量的估计值也可能更大。

五、 图重写系统

主要由于在计算机科学中采用图和图论概念有着重大意义，致使图重写系统目前受到了极大关注[11,12]。我们想以一种抽象的方式建立软件维护过程的模型，所以自然要研究这种抽象工具的使用，特别是看到一些图表示法的优势，因为这些表示法能表达软件要求和设计，也能建立程序的模型。我们只讨论顺序的图重写，其中每次只应用一条重写规则（顺序方式），而不是一次应用所有的重写规则（并行方式）。经验告诉我们，这适合于建立软件开发过程的模型。

一个图重写系统可以定义如下：

定义1 一个（标号）图是一个八元组 $(N, A, L_N, L_A, S_N, t_N, n_L, a_L)$，其中

- N是一个结点集合；
- A是一个弧集合；
- L_N是一个结点标号集合；
- L_A是一个弧标号集合；
- S_N, t_N: $A \to N$是一些把每一个弧（分别）映射到其源点和目标结点的函数；
- $n_L: N \to L_N$是一个把每个结点映射到其标号的函数；
- $a_L: A \to L_A$是一个把每个弧映射到其标号的函数。

定义2 一个（标号、顺序）图重写系统是一个六元组 $(L_N, L_A, T_N, T_A, START, PROD)$，其中

- L_N是一个结点标号集合；
- L_A是一个弧标号集合；
- T_N是一个非终极结点标号集合；
- T_A是一个非终极弧标号集合；
- START是一个有限图，即重写规则可以作用的初始图；
- PROD 是重写规则的一个有限集合。

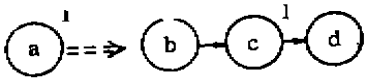
每一重写规则有一个左端和一个右端，每一端包含一个图。为了把一条具有左端L和右端R的重写规则作用到一个图G，首先必须找出图L作为G的一个子图的一个实例。如果不存在这样一个实例，那么不能运用重写规则。如果存在这样一个实例，那么应从 G中删去该子图，而把R加到G中。

图4说明了如何把一个图嵌入另一图中。由于这个例子中的每个结点被唯一地加以标号，所以我们把一个用标号L 标记的模型叫做“结点L”。图4 中所示的重写规则要求我们用一个由三个结点（b、c 和 d）和两条弧（从b到c和从c到d）组成的子图来代替结点a。显然，被重写后的图必须包含五个结点b，c，d，e和f。此外，它也必须包含从b到c和从c到d的弧。然而，在从a到e和从a到f的原图中，我们对弧应该做什么呢？也就是说，一个新的子图应怎样嵌入原图中呢？我们已经采用过的方法是把一些整数标志分配给重写规则中被选中的结点，并且约束条件出现在此规则左端的任一整数标志也必须出现在其右端。这些整数标志定义每一子图的结点间的一种关系，使得如果结点m在左端，结点n在右端，并且它们各有相同的整数标志，那么这两个结点（m和n）是相关的。在这种情况下，我们称n是m的“替换物”。标志过的结点本身叫做粘合项（gluing item），因为它们表明每一子图如何被“粘合”或“嵌入”整个图中。

这种标号分配的解释是，当运用一条规

则时，左端标记为i的任一结点被认为是由右端标记为i的结点所代替，因此进入（或离开）原路径中结点的任何弧应进入（或离开）右端图中结点的代替者。图4中，a是一个被代替的唯一结点，其代替结点是c(如由整数标志1所示)。根据嵌入规则，重写过的图是图4底部所示的图。

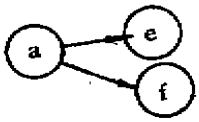
规则



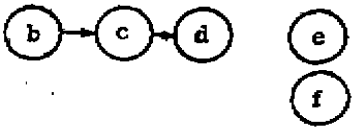
（使用具有结点b、c和d的图代替结点a）

应用

如果原图是



那么可以执行下面的规则运用。



最后（重写过的）图是

图4 例子——重写图

六、 集成生存期模型

现在，我们介绍一个集成模型，它确实能够以一种兼容的方式表示软件生存期的不同阶段，而且也可以表达阶段间的复杂关系。我们的表示法与语言无关，描述了三个“基本”语义性质：控制流、数据流和数据结构。所有这三个性质都出现在各个软件开发阶段的表示法中。我们强调这些性质，而不考虑更多的语言相关性质，如语法规则，我们根据不同阶段间的关系描述同样定义的结构之间的关系的模型。

控制流、数据流和数据结构都可以用图进行表示。为此，我们使用“图重写规则”来表达阶段间的关系。每一条重写规则表明，一个阶段的初始子图联系下一阶段的一个具体子图。我们说后一个子图“是从”前一个子图“导出”的。这种方法可以十分详细地描述阶段间的关系，当然，这种描述的价值受到表达推导规则的方式的限制。例如，所有开发步骤都可以用一条规则来描述，此规则的两端是整个资料的一些图模型。这种想法可能不理想，因为这样一条规则可能对那些跟踪一份资料的改变对另一份资料的影响的任何人没有帮助。凡有可能，由两端极复杂的规则所反映的这种不精确描述应分解成一组更具体的描述，其中每一描述定义了原复杂结构的一些小的部分间的某些关系。

这些基本关系的分离有助于使这种方法与任何具体的表示法更无关。理由是一些用来说明规格说明和设计的表示法只显式地描述其中某些关系而非全部的关系。例如，规格说明语言RSL[18]具有控制流和数据结构以及一些数据流信息的显式语句。然而，附加的数据流信息必须是通过分析导出的。由于RSL中的功能描述是用自然语言说明的，所以这些描述可能容易适合基本模型。Jackson程序设计方法[8]的设计表示法强调软件系统的数据结构，并允许从这些数据结构导出整个控制流。然而，数据流在整个方法中起次要作用。在这种方法中，对最低级功能描述没有定义表示法，尽管Jackson在他的例子中使用了类Cobol命令。

由于图模型的复杂性，我们已经开发了一种以文本形式描述这些图模型的标准化方法。对图重写规则，我们也如法泡制。这就在表示中引入了一定量的冗余，因此同一名字可以出现在模型中的几个不同位置，并且可以理解为都属于同一对象。把弧纳入到唯一命名的结点之中，图模型能够避免这种冗

余。我们发现这些描述比对应的图形更容易理解，尽管分析工具是以图表示进行工作的。图模型的文本描述往往比被模型化的资料更长这一事实，主要是模型使诸如控制流和数据流这些性质更明显化的结果，而这些性质在原资料中是隐含的。

1. 各阶段使用的模型

软件生存期任一具体阶段的模型是由软件系统的一组互连成分构成的一个图。每个成分我们使用其控制流、数据流和数据结构，及其与另一些成分的关系来表示。一个成分与另一些成分的接口说明借助了由其它成分所需要的对象和为其它成分提供的对象。一个软件系统只不过是一组具有不同初始（或主）成分的这样一些成分。让我们首先定义一个软件成分的表示法。

定义 3 一个软件成分是一个包含几个子成分的可执行对象。这些子成分是：

- 一个控制流结构。
- 一组数据结构图。
- 一组数据流三元组，其可执行对象是控制流图的一些"叶"，而其（输入和输出）数据对象是数据结构图。
- 一组不同对象名，每一个对象名都指一个数据结构图或定义此对象结构的成分。

1)接口成分：我们已经把软件成分定义成整个系统中一些分开的独立活动。为了以一种协调方式起作用，这些成分必须彼此共享信息，而且互相提供服务。我们希望以一种科学的形式表示成分间的依赖性，这种方式允许对修改进行分析，并且可以配合软件成分的表示。

我们的方法是把一个接口子成分与各个软件成分联系起来。在这个接口中，定义了在当前成分外边出现的所有对象，也定义了在当前成分内出现的且可以由其它成分使用的所有对象。这些对象进一步区分为直接链接到外部成分的对象和被间接链接的对象（作为参数），也区分为被使用但不改变的对象和可以改变的对象。接口图可以形式地定义如下：

定义 4 一个接口图是一个标号图，它具有：

- A，$N=Z^+$
- $L_N=\{INTERFACE, IMPORTS, PARAMETERS, READS, WRITES\} \cup Z^+ \cup \{e\}$
- $L_A=\{e\}$

其中Z^+表示正整数集合$\{1, 2, \cdots\}$，e代表空串。

这种图是一个有根无环有向图。由IMPORTS, PARAMETERS, READS和WRITES标记的结点叫做结构式结点。由e标记的结点和弧叫做e标号结点和弧。e标号结点叫做本原结点。由一个正整数标记的结点和弧叫做Z^+标号结点和弧。结构式结点总是树中的非终极结点。本原结点总是树的终极结点（叶）。树的所有叶都是e标号的。树的所有结点和弧都是由L_N或L_A的一个标号标记的。

结构式结点

形式：成分接口有五个不同的结点，标记为INTERFACE, IMPORTS, PARAMETERS, READS和WRITES。INTERFACE标号结点是此有向图的根。其它四个不同的结点是根结点的直接子孙结点。树的其它结点不是直接连接到根结点的。

解释：根结点子图表示这个成分与其它成分间的接口。往来于其它成分的所有引用被强迫通过这一子图。这个子图包括为完成与任何外部成分接口所需要的所有名字和结构信息。

本原结点(IMPORTS, PARAMETERS)

形式：接口有向图的本原结点被直接连接到IMPORTS，但也可以连接到PARAMETERS。

解释：这些本原结点表示接口对象。如果一个结点被连接到IMPORTS，那么由该结点所表示的对象是可以访问的。如果一个结点被连接到PARAMETERS，那么由该结点所表示的局部对象提供了对其它对象的间接

访问，并且局部对象和其它对象间的这种关系通过系统成分的执行是可以改变的。

本原结点（READS，WRITES）

形式：接口有向图的本原结点被直接连接到至少READS和WRITES结点之一（连接到这两者是可能的）。

解释：这些本原结点表示接口对象。如果一个结点被连接到READS，那么可以检查和使用由该结点表示的对象，但不可加以改变。如果一个结点被连接到WRITES，那么可以改变由该结点表示的对象。

一具体阶段的模型是一个由“接口子成分”互连的软件成分构成的图。接口子成分是一个如图5所示的接口图，它具有如下的等价文本定义：

```
INTERFACE
  IMPORTS{名表}
  PARAMETERS{名表}
  READS{名表}
  WRITES{名表}
END.
```

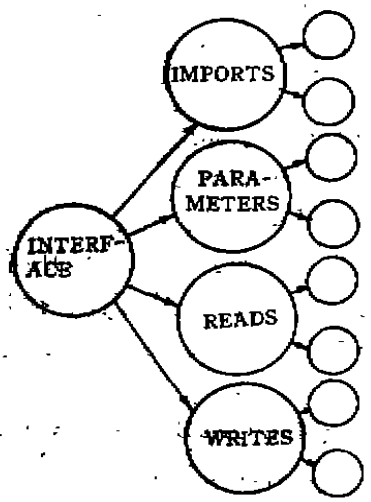


图5　成分接口

2)控制流：下面的表示法被用来描述一个软件系统成分的控制流。由于它只强调活动间相对次序，所以这种表示法与原先用来描述成分的具体表示法无关。这种表示法使用了标号图这种形式体系，它使用结点表示“活动”，使用弧表示这些活动间的控制流关系。我们首先将全面说明这一基本表示法，然后非形式地描述此表示法的其余部分。

定义5　一个基本的结构式顺序控制流的描述是一个标号图，它具有

- A，$N=Z^+$
- $L_V=\{TASK, LOOP, AND, OR\}\cup Z^+\cup\{e\}$；
- $L_A=Z^+\cup\{e\}$

此图是一个向下有向的有根树形结构。标记为LOOP，AND或OR的结点叫做结构式结点。由e标记的结点和弧叫做e标号的。e标号结点叫做本原结点。由正整数标记的结点和弧叫做Z^+的标号的。

TASK

形式：软件成分有一个标号为TASK的特异结点。这个结点是树的根。

解释：以这个结点为根的子树表示一个独立的、可执行软件成分。

LOOP

形式：LOOP结点总有一个儿子。

解释：这解释成这样的含义，即由以LOOP结点的儿子为根的子树表示的活动被执行零或多次，次数是在LOOP结点中以一个未说明的方式决定的。

AND，OR

形式：AND或OR结点总有一个儿子，它们必须是Z^+标号结点。

解释：AND结点表明其Z^+标号结点的所有孩子都必须被执行，而OR结点表明其Z^+标号结点的一个孩子必须被执行。

Z^+

形式：具有标号n的任何Z^+标号结点也必须有出度n。其树中的父亲必须由AND或OR标记。这个结点的弧是必须由集合{1，2，3，…，n}标记的源。

解释：弧标号的值指明活动执行的次序。标号为i的活动应在标号为i+1的活动之前执行。

对这些基本结构的扩充，允许表达式结

点与受OR和LOOP支配的动作相联系，如图6所示。此外，当动作的次序是未知或不确定的时候，可以忽略上面所描述的弧标号。由于在某些动作的执行中，允许不确定性，所以我们开辟了模型化并发软件系统的可能性。

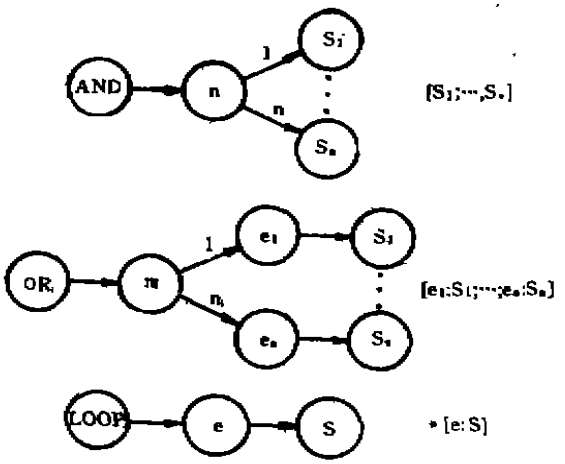


图6 基本控制流结构

总之，AND表示执行(n)个活动的序列，OR表明选择1（或n）个活动。LOOP表示一个活动的重复执行。控制流结构具有图6中所示的结构图形式。这些结构也具有如下的文本表示：

• $〔S_1; S_2; \cdots; S_n〕$表示n个结点的一个集合连接到一个AND结点，或者若无序，则指 $\{S_1; S_2; \cdots S_n\}$ 。

• $〔e_1:S_1; e_2:S_2; \cdots; e_n:S_n〕$表示n个结点的一个集合连接到一个OR结点，或者若无序，则指$\{e_1:S_1; e_2:S_2; \cdots; e_n:S_n\}$，并且由集合$\{e_i\}$表明表达式结点。

• ★〔e:S〕表示一个结点连接到一个LOOP结点，并且由"e"表明表达式结点。

我们已经证实，这种表示法可以用来描述大多数控制流性质，例如用RSL描述要求定义或用Pascal描述程序的控制流性质。此外，使用这种抽象的控制流观点，例如，有可能从RSL规格说明的控制流要求构造Pascal程序的骨架。

3）数据结构：对成分的数据结构使用同一表示被认为是很方便的。唯一的差别是在每个结构的最低级上说明的本原对象类型。图7示出了图的形式，其等价的文本表示如下：

• $〔D_1; D_2; \cdots; D_n〕$或$\{D_1; D_2; \cdots; D_n\}$

• $〔e_1:D_1; e_2:D_2; \cdots; e_n:D_n〕$或$\{e_1:D_1; e_2:D_2; \cdots; e_n:D_n\}$

• ★〔e:D〕

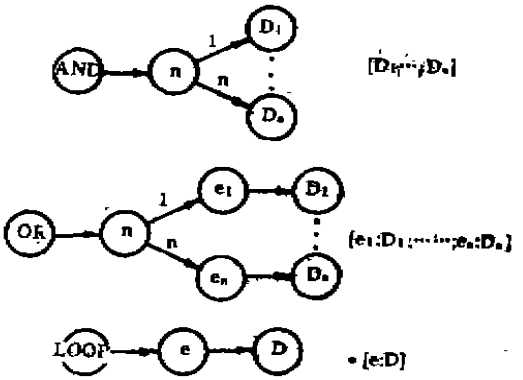


图7 基本数据结构

4）数据流：假如已经确定了数据结构和控制流，那么数据流的表示就十分简单。这种表示可以定义如下：

定义6 数据流信息用如下形式的三元组集合表示：

〈In, Ex, Out〉

其中Ex是一个可执行对象（如语句或过程），In（输入对象）和Out（输出对象）是一些数据对象（如程序变量），并且数据流信息也用如下形式的三元组集合表示：

〈Ex_1, D, Ex_2〉

其中Ex_1和Ex_2是可执行的对象，D是数据对象。

三元组〈In, Ex, Out〉具有这样的解释：Ex可以使用In的值来改变Out的值。在其图形式中，每个三元组表示从活动Ex到其它某一活动标记为Out的弧的存在和从其它某一活动到活动Ex标记为In的弧的存在。如图8所示，数据项D_1被函数S使用来产生D_2中的值。当数据流性质以这种方式进行说明时，有可能对基本动作S的行为给出一个有限的

“语义定义”。例如，在图8中，我们可以说，$D_2=S(D_1)$。三元组 $<Ex_1, D, Ex_2>$ 具有这样的解释：Ex_1 可以改变由 Ex_2 接着使用的D的值。

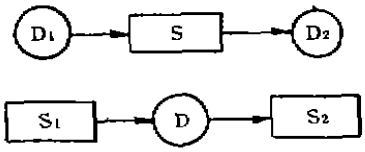


图8 数据流表示

5)数据字典：在内部，每个成分的描述是一个数据字典。习惯上，这种字典包括该成分各个元素的定义，但不包括从其它一些成分输入的那些元素。任一成分有三个元素类型：活动、数据和结构。

活动由数据项定义。活动也描述对该数据执行的操作。这些操作包括由语言，即操作环境的表示法所定义的操作和由其它系统成分完成的操作。例如，Pascal程序的“+”操作，可能指一条硬件相关的加法指令，Pascal Sin(正弦)函数，可能指系统的标准函数库的一个函数。

数据只由其结构和名字定义。数据项的结构可以通过语言，即（不太经常，但硬件相关的）操作环境的表示法来定义，或通过其它的系统成分来定义。例如，Pascal程序的标准文件“input”，指一个标准系统输入文件（通常是终端键盘），Pascal常数“maxint”，具有为具体计算机系统中Pascal程序可用的最大整数值。

结构由连接的子成分定义，这些子成分自身或者是数据项，或者是其它结构。子成分可以指由语言，即（不太经常，但硬件相关的）操作环境表示法，或系统的其它成分所定义的结构。例如，Pascal程序的标准类型“text”和“integer”，“text”指字符顺序文件的系统实现，而“integer”受变量字长和计算机系统的精度的影响。

数据字典是一个可以由内部对象名访问的子结构，表示一个具体的活动，结构，或信息块。当对象是一个命名了的活动时，内部名就产生另一软件成分。当对象是一个未命名的活动时，内部名就产生该活动的一个描述。当对象是一个结构时，内部名就产生该结构形式的一个描述。最低级结构是由计算机装置所定义的结构。当对象是一块信息时，内部名就产生信息具有的结构定义。

2. 表示阶段间的关系

使用图重写规则表示阶段间的关系十分新颖，尚未得到自动工具的支持。在这一节，我们给出一个使用图重写系统的阶段间模型。为了构造这种阶段间模型，已经存在（至少）两个阶段的图模型是必要的。我们把这些图模型叫做源阶段内和目标阶段内模型（简称为源模型和目标模型），并且我们说目标模型是从源模型导出的。由于软件系统的单个资料是用图表示的，所以阶段间模型可以简单地表示成图重写系统。阶段间模型的重写规则定义了一个可以用整个目标模型代替整个源模型这样的过程。在每条规则的左端是软件系统源资料的图模型的一个子图。在每条规则的右端是该系统目标资料的图模型的一个对应子图。现在我们对一组规则可以取的形式给出某些限制。这些限制借助基本图模型的细节来表示，可以由自动工具加以最好地控制。

阶段间模型是一个任意图重写系统，不同的是，源资料的所有结点和弧都应出现在某规则的左端，并且目标资料的所有结点和弧都必须出现在某规则的右端。阶段间模型中的每条重写规则定义软件开发活动的一个步骤。每条规则可以解释成以下求精步骤的描述：在已经研究了由一具体规则的左端表示的源模型的部分后，软件开发者定义在目标模型中起作用的规则的右端，这种作用等同于此规则左端在源模型中所起的作用。这意味着，每条规则定义软件开发活动的一个独立的推导步骤，并且可以用来划分源模型和目标模型。因此，如果对软件系统的具体部分进行改动，那么我们可以通过查找在其

（下接7页）

发者保证是正确的测试数据进行预先写明的计算得到的。

原则上，软件可靠性的估计值不光是一种计算平均无故障时间的方法，而且还考虑了统计测试期间的修改行为。当测试数据确定后，就可以在每次修改之后估计可靠性。若改正是成功的，那么通过随后的测试，可靠性估计将不断改进，从而给达到可靠性目标提供客观的定量证据。

这种客观的证据本身就是为达到可靠性目标而对软件开发进行管理控制的一个基础。例如，过程分析可以揭示未预料到的错误来源（如对所用的硬件理解不够），并为以后的增量着想适当地改进过程本身。最终认可的中间"演习"给管理以反馈，从而达到最终目标。

处理各个交付的增量也应作为开发者与用户间契约基础的一部分。也许最简单的处理方法就是独立地对待各个增量。然而，最终认可的更高统计可信度来自概括各增量间测试经验。简单的概括方法是对分别处理的增量进行管理判断。

处理各个交付增量的较复杂方法是，把软件的各个新交付部分的故障给与率模型化，并为各个交付增量逐一提出分层的估计。当后面的交付增量被测试时，早些的交付增量可能已成熟。这种可靠性改进的成熟速率可用来估计为达到预定可靠性级别所需的总测试时间。

对于项目管理来说，平均故障间隔时间及其变化速率是一个有用的决策工具。对测试中的软件，既有对平均故障间隔时间的估计，又有已知的平均无故障间隔时间的变化速率，因此对可靠性的判定可以依据生存期的费用估计，而不只考虑市场影响。

在软件提交后，发生每一故障的平均费用中必须包括修复故障所花的直接费用和给用户造成的间接费用（后者可能要大得多）。交付后的这些费用可以根据预期的故障数来估计，并同交付前额外测试所花的费用进行比较。根据使全生存期费用最少的原则来判断继续测试是否合算。

参考文献（略）

〔朱　力、张　然　译自《IEEE Software》September 1987 pp19—24仲　源校〕

（上接22页）

左端具有该系统那部分的所有规则，并识别在对应的右端的所有子图来弄清这种改动对下一阶段产生的潜在影响。

这是一种描述软件开发过程和关系的新方法，所以象"基本"性质（如阶段中的控制流）一样，阶段间不存在已确立的"基本"关系。因此，我们对构成一个阶段间模型的规则形式不做更多的假设。如果模型建造者希望阶段间任意复杂的关系，那么可以使用相应复杂的规则来描述这些关系。然而，当使用一种具体的软件开发方法时，所有规则可能表现出一些具体的模式。这种情形的最明显例子出现在"逐步求精"的功能分解过程中。这个过程由只能扩充系统最低级动作的一些推导步骤表征。这些扩充的目的是更加详细地表达构成低级动作的活动。描述这一过程的图重写规则在左端必须恰好有一个动作结点，它被右端的一个非约束图模型代替。逐步求精的所有图重写规则将适合这一模式。

（未完待续）

〔仲　源译自《IEEE Trans. Software Eng.》№8, Vol. 14, 1988, p1128-1144　声　钟校〕