形式规范与程序生成

庄庆雨（中国科学院数学所）

周力研（吉林大学计算机系）

摘　要

形式规范和程序生成是自动程序设计的两个关键问题，通过形式规范的引导可以正确地生成程序。本文首先讨论与形式规范相关的三个问题，形式规范方法，形式规范生成和形式规范正确性；然后讨论从形式规范生成目标程序的技术以及程序正确性问题。

六十年代以来，随着计算机解决的问题越来越复杂，人们面临着一个严峻的挑战——如何保证大型软件的可靠性，如何克服“软件危机”。为了解决这一问题，人们从应用和理论两个方面开展了工作。在应用领域，发展了一些系统工程的方法来开发与维护软件，以取代传统生产软件的方式；在理论方面，人们研究了形式化方法可能在软件发展过程中起到的作用。近年来，随着人工智能技术的发展，自动程序设计越来越受到人们的重视。自动程序设计就是使用变换等技术把一个程序规范（program specification）自动翻译成程序，是提高程序生产率的根本途径。

书写形式程序规范是自动程序设计的第一步，程序规范的好坏直接影响到下一阶段的工作。有三种书写程序规范的方法：非形式规范、形式规范和混合型规范。非形式规范使用类似自然语言的工具来说明程序要求，形式规范使用比自然语言抽象的工具来说明程序要求，而混合型规范是前两种的综合。程序规范是对程序要求的说明，是程序实现者的指南。因此，在交给实现者之前，必须有相应的手段来保证程序规范的正确性。由于消除程序规范中的错误能减少今后维护软件的开销，因此，人们越来越重视程序规范正确性证明的研究。

自动程序设计的第二步就是要以程序规范为指南，生成目标程序。比较常用的技术有程序变换、程序综合、速成原型。程序变换和程序综合技术比较复杂，目前只能生成一些小而简单的程序。用速成原型方法得到的原型是程序的半成品，但它展现了程序的主要功能，通过运行原型，能及时发现程序规范中的错误。

自动程序设计的最后一步是要证明目标程序的正确性，也就是说，证明目标程序满足程序规范的要求。一个程序生成后再来证明它满足程序规范有很大的困难，因此，人们认识到应该把正确性证明渗透到整个程序生产过程中，设法证明每一阶段的结果都与前一阶段的结果相一致，以减少证明的复杂性。

本文讨论了与自动程序设计相关的五个问题：形式规范方法、形式规范生成、形式规范正确性、程序生成、程序正确性证明，然后指出今后的研究方向。

一、程序规范

所谓程序规范是指对用户要求的一个精确描述，是判断程序正确与否的标准。根据形式化程度的不同，程序规范可分为非形式规范、形式规范和混合型规范三种。(1) 非形式规范：是用自然语言书写的程序规范，但由于二义性等因素，往往给实现增加不少困难。(2) 形式规范：形式规范使用各种抽象概念来描述程序应具备的功能，具有一定严格性，但较抽象，用户和实现者理解有一定的困难。(3) 混合型规范：是在一个程序规范中同时使用以上两种方法，以扬长避短。

我们下面详细介绍形式规范的两种主要技术，过程抽象和数据抽象，并通过例子给予说明。

1.1 过程抽象 过程抽象忽略了实现程序的具体细节，抽象出最本质的东西来表示程序的含义。输入/输出规范和操作规范是过程抽象的两种方法，以下将简单介绍。

1.1.1 输入/输出规范（I/O规范） 这种规范把程序抽象成一个映射，它把一个输入状态对应到一个输出状态。I/O规范可表示为{Q}P{R}。它表明，只要输入状态满足条件Q，程序的输出状态应满足输出条件R。例如：程序P求两个非零正整数的最大公约数，它的I/O规范如下：

```
Interface: gcd(integer, integer)return integer
Behavior: x>0&y>0{ged(x, y)}divide(gcd(x, y), x)&divide(gcd(x,y), y)&∀i: integer[divide(i, x) & divide(i, y) ⇒≤(i, gcd(x,y))]
Abbreviation: divide(x, y)≡∃i: integer [y=x*i]
```

1.1.2 操作规范 在I/O规范中，仅仅定义了输入和输出之间的关系，没有给出计算的详细过程，而操作规范将通过递归等方式明显给出计算的过程，但与实现不同，它仅仅是对程序的简单描述。例如：程序P求两个非零正整数的最大公约数，它的操作规范是：

```
Interface: gcd(integer, integer)return integer
Behavior: gcd(x, y)= if x≤0∨y≤0 then error else search-from(x,y,min(x, y))
Abbreviation: Search-from(x,y, z)≡ if mod(x, z)=0&mod(y, z)=0 then z else Search-from(x, y, z−1)
```

min(x, y) 得到x和y中最小值，mod(x,y) 求x和y的模。

1.2 数据抽象 数据抽象由一个数据抽象类型或一组相关的数据抽象类型组成，一个抽象数据类型是一个对象的集合，在该对象集合上只能允许有限的操作。可以认为，一个抽象数据类型的性质由这些操作的行为所确定。下面介绍两种实现数据抽象的方法：代数规范，抽象模型规范。

1.2.1 代数规范（algebraic specification） 一个数据类型的代数规范由三部分组成：语法规范、语义规范、限制规范。语法规范定义与此数据类型相关的操作名称、定义域和值域；语义规范由一组公理组成，公理用来定义操作的含义；限制规范说明各种限制条件。我们下面以“序列”为例来说明代数规范的使用。

```
Object seq(integer)
  Syntax append: seq×integer⟶seq
         leader: seq⟶seq
         trailer: seq⟶seq
         first: seq⟶integer
         last: seq⟶integer
         nullseq: ⟶seq
         length: seq⟶integer
  Semantics leader(append(s, element))=
   =s; trailer(append(s,element))=if s=
   append(nullseq, element−1)
   then append(nullseq, element)
   else append(trailer(s), element);
   first(append(s, element))=
```

```
    if s=nullseq then element else first
    (leader(s));
    last(append(s,element))=element;
    length(nullseq)=0
    length(append(s,element))=length(s)+
    1;
  Restrictions (略)
```

代数规范是由它与异型代数(heterogeneous)的关系而得名。异型代数定义为$\langle D_1, D_2, \cdots, D_n; F\rangle$，其中，$D_1, D_2, \cdots, D_n$是非空集合，F是函数集合，$\forall f \in F$，$f: D_{i1} \times D_{i2} \times \cdots \times D_{ik} \to D_{ik+1}$，$D_{ij} \in \{D_1, D_2, \cdots, D_n\}$，$1 \leqslant j \leqslant k+1$。例如：我们上面定义的序列可以认为是一个异型代数，它有两个非空集合{integer, seq}，append是一个从seq×integer到seq的函数。

1.2.2 抽象模型规范 在抽象模型规范中，要定义的数据类型由其它的数据类型来定义，而这些数据类型事先已由形式规范给出定义，因此，被定义数据类型上的操作可以根据已知数据类型上的操作来定义。我们下面定义栈，它所依赖的数据类型是序列。

```
object stack;
stack is modeled as seq;
invariant  0≤length (stack) ≤1000;
initially  stack=nullseq;
functions
  push (item: integer);
    pre 0≤length (stack) ) ≤1000;
    post stack=append (stack, item);
  pop
    pre stack≠nullseq;
    post stack=leader (stack);
  top returns (item, integer)
    pre stack≠nullseq;
    post item=last (stack);
  empty returns (flag: boolean)
    post flag= (stack=nullseq);
```

二、 形式规范的生成

为了产生与用户要求一致的形式规范，分析员起着关键作用，他使用自然语言与用户对话，了解用户的意图，然后使用形式规范语言书写程序规范。近年来，有人开始着手研究自动或半自动的工具来帮助分析员，尽管有许多研究还没有进入实用阶段，但这些研究却为形式规范的发展方向奠定了基础。

2.1智能助手 (Intelligent Assistant)

智能程序设计助手是以知识为基础的软件系统，用于帮助人们从事软件开发各个阶段的工作。迄今为止，有关智能助手的研究工作都与程序实现有关，例如，KBEmacs是一个实验系统[9]，它的主要特点是允许程序员通过组合库中的算法快速构造一个程序。随着软件发展的需要，人们的注意力已扩大到了用户需求分析上，开始研究能够帮助系统分析员产生和修改形式规范的智能助手(IA)。

用于需求分析的智能助手的作用可由下图表示：

IA并不与用户直接对话，而是作为需求分析员的助手来辅助完成需求分析工作，最后得到形式规范，交付给系统设计人员，它是沟通需求分析员和系统设计员的桥梁。用于生成形式规范的智能助手需要解决两个突出问题：①系统分析员描述问题的非形式性；②从非形式描述生成形式规范。

2.2 分析自然语言文本 使用自然语言，人们能够自然地写出对软件的要求，尽管这样的描述通常是不完整的，甚至包含着矛盾，但由于它直接记录了用户要求，其中有许多可用来推导形式规范的重要信息。在用自然语言书写的程序规范中，与软件相关的各个概念都有对应的字与它相联系，字与字在句子中的关系决定了相应软件概念之间的关系，例如，假设我们在一个规范中说明了

"为了实现功能F，首先执行功能f_1，然后执行功能f_2"，这句话蕴含地说明了这样的事实：f_1、f_2是F的两个子模块。所以，我们能够使用自然语言的语法和语义知识来分析自然语言文本，从上往下逐步分解，求精一个大的软件概念，使得它可以用多个更小的软件概念来表示，同时建立这些子概念在逻辑上的关系。

当前，从自然语言描述生成形式规范的技术还不成熟，它的发展很大程度取决于自然语言理解技术的发展。

2.3 可执行的规范语言 我们可以使用可执行的规范语言来书写形式规范，这样的形式规范是系统的一个原型，它可以被直接执行，因此，我们能够通过执行结果来判断这个原型是否满足用户的要求，并及时作出修改。可执行的规范语言是形式的，用它书写的形式规范没有二义性，因此，可以使用形式证明技术来证明它的正确性，也可以使用变换技术从它得到目标代码。

三、形式规范的正确性

它的非形式定义是：如果一个形式规范正确描述了用户的客观要求，称它是正确的，否则是错误的。形式规范是用户与实现者之间的协议书，在实现者完成这个协议书之前，必须保证它是正确的。

形式范规中的错误直观上是很容易理解的。例如：某些对象没有被定义就使用了；说明了某些对象，但从没有使用它们。很显然，这里提到的形式规范中的错误与程序中的错误十分类似，自然有人想到用分析程序错误的方法来发现形式规范中的错误，但形式规范的情况要复杂得多。例如，代数规范中的对象往往和若干个公理有联系，使用这些公理进行推理时可以产生各种结果，但我们往往很难判断一个公理的选用是否是冗余的；若一个公理被选用，也难于确定由它推导出的结果是否有用；另外，对于从两个不同的公理出发，是否会得出矛盾的结果也难于判断。我们把一个形式规范抽象成一个理论T，由对象集、公理集、规则集组成，这样一个规范的正确性可由三个性质确定。

3.1 一致性 已知一形式规范S，如果使用S的公理和规则推出true=false，那么，称S是不一致的。从理论上讲，判断一个形式规范是否满足一致性这个问题尚未解决，但实际上已有一些证明方法。比较常用的构造模型方法，例如，我们使用代数规范定义了一个数据抽象T，为了证明T满足一致性，就要实现这个数据抽象，但所依赖的基础应是一个（或若干个）能证明一致性的数据类型。

不一致的产生还往往与公理应用的次序有关，所以，为了证明形式规范是一致的，可把公理看成规则，并且证明该规则集满足Church-Rosser性质。一个规则集满足Church-Rosser性质，是说如果使用规则序列r_1，r_2，…，r_n作用于项i_0得到i_n，而且i_n的获得不依赖于应用规则的次序。Knuth和Bendix给出了证明一个规则集满足Church-Rosser性质的方法。

3.2 完整性 定义完整性的方法有许多，例如，一个理论T是完整的，若T中的每一合式公式F(或$\neg F$)都能被证明是一个定理。对于形式规范有一个较弱的定义：一个满足一致性的理论T关于一个公式集合G是足够完整的，当且仅当对G中的每一个公式都存在一个仅使用T中公理的证明。足够完整性基于这样的事实：程序往往都用来完成某一任务集，而不关心程序在该任务集之外的运行情况怎样。如何证明一个公理集合是否足够完整的问题还没有得到解决。然而，能够建立某些限制条件。当一个公理集合满足这些条件时，就能保证它是足够完整的。Guttage和Horning讨论了这样的条件。

3.3 相关性 相关性与这样的事实有关，即：形式规范中的某些内容可能是多余的。一个关于公式集合T是足够完整的理论

T满足相关性，当且仅当对T中每一公理P，至少存在G中的一个公式g，P与g的证明相关。公理P与公式g的证明相关有两种含义：①存在公式g的证明，该证明中使用了公理P；②若A是在证明g的过程中使用的公理集合，从A中去掉P后得到A′，在A′下不存在g的证明。

从理论上讲，只要形式规范语言具有形式化特征，并提供演绎能力，我们就可以建立证明器来验证形式规范是否满足一致性、完整性和相关性。ERAE就是这样的一个形式规范语言[2]。但就目前情况来看，实现自动验证还有一定的难度，因此，人们往往使用一些更为实际的方法，例如，速成原型方法。

四、程序生成

从形式规范出发，可以采用三种方式来生成程序，手工，自动和半自动。编制程序的最简单方式是由程序员来完成，程序员首先理解形式规范的含义，然后使用各种程序设计技术来编制程序。这里，我们重点讨论几种自动（半自动）生成程序的技术，从中可看出形式规范的作用。

4.1 程序综合 程序综合技术从形式规范出发来推导程序，它把定理的构造性证明过程与程序生成联系起来，证明的每一步解释为一计算步骤，通过分析证明过程来生成所需要的程序。Z.Manna和R.Waldinger给出了这种技术的一个逻辑框架，它把用逻辑公式书写的I/O规范自动翻译成等价的程序[1]。我们下面通过一个例子来说明这个框架的大意。

程序P计算i/j的商以及余数，其I/O规范是：

〔integer(i)∧integer(j)∧0≤i∧0<j〕
{div(i, j), rem(i, j)}
〔(∃y)(∃Z)(integer(y)∧integer(Z)∧i=y·j+Z∧0≤Z∧Z<j)〕

为了生成程序，Z. Manna和R. Waldinger使用了三种技术：①变换规则，它的一般形式是r⇒S if P；②归结；③结构归纳，数学归纳法是它的一个特例。已知输入断言（integer(i)∧integer(j)∧0≤i∧0<j），为了证明输出断言（integer(y)∧integer(Z)∧i=y·i+Z∧0≤Z∧Z<j）为true，从输出断言出发，以输入断言为前提，不断利用上述三种技术，记录下从输出断言到true的推导过程，最后可得到程序P：

```
div(i, j)⇐if i<j          rem(i, j)⇐if i<j
then 0                    then i
else div(i−j, j)+1        else rem(i−j, j)
```

4.2 变换 当我们要把形式规范转换成等价的可执行程序时，可以使用变换技术。变换技术的核心是变换规则，根据规则的作用，可把变换规则分为两类：①纵向变换规则，它用抽象程度较低的结构来替换抽象程度较高的结构，使得一个用抽象结构表示的程序逐步变成一个可执行的程序；②横向变换规则，它用抽象程度相同的结构来替换另一个结构，一方面为了提高程序的执行效率，另一方面为了改变结构的形式，便于使用纵向变换规则。

程序变换技术面临几个需要解决的问题：

①正确性问题。一般来说，一个程序经过变换后，它的可读性比原来差，因此，必须设法保证变换的每一步是正确的，这是变换技术成败的关键。

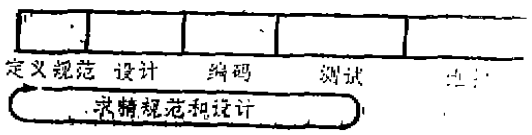
②控制问题。在程序变换过程中，对于某一个中间结果，往往有好几个变换规则可以使用，选择不同的变换规则会得到不同的结果，因此，需要一个好的控制机制。把形式规范看成初始状态，目标程序是终止状态，中间程序构成中间状态。那么，从形式规范出发，使用变换规则生成目标程序的过程等价于一个状态空间搜索问题。这样，可以把人工智能领域中有关状态空间的搜索算法应用于解决控制问题。

4.3 速成原型 对于庞大的程序系统来说，由于用户考虑不周，难免在形式规范中蕴含错误；当人们在目标系统中发现与实际要求不符合的情况时，只有回过头来修改形式规范，重新设计，重新编码，这造成人

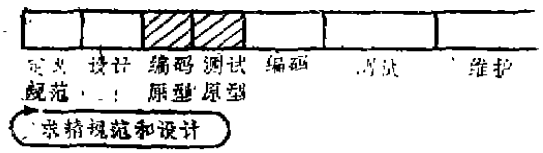
力物力的极大浪费。为了减少浪费，我们可以从形式规范出发，先快速生成目标系统的原型，尽管原型离实际要求还有一段距离，但它展现了目标系统的主要功能。通过执行系统的原型，能及时发现形式规范以及设计过程中的错误，并及时对这些错误进行修改，从而减少了重新设计、重新编码的工作。有了系统原型后，我可以对原型施行变换和求精，逐步得到目标系统，而不必从设计开始。

传统软件生存周期包括生成形式规范、设计、编码、测试和维护几个阶段，由于需要重复修改形式规范错误和设计错误，编码和测试所需时间较长。速成原型不同于传统的软件生成过程，增加了编码和测试原型的工作，由于许多形式规范错误和设计错误都可以通过原型发现和修改，编码和测试时间大大减少。速成原型与传统软件生成方式在开销方面的差别可用下面两个图来表示。

传统方式：



速成原型：



五、程序正确性证明

程序正确性证明的目标是要证明程序满足它的形式规范说明。我们用Requirements-set表示形式规范例举的要求集，用Functions-set表示程序所实现的功能集，那么，程序满足形式规范有两种解释：①Requirements-set⊆Functions-set。即：只要求程序实现了形式规范所指定的功能。并不关心程序是否实现其它功能；②Requirements-set=Functions-set。即：严格要求程序实现的功能均为形式规范所指定。

怎样证明程序的正确性呢？这是人们一直关注的问题。对一些小程序来说，已有一些形式化的方法可以使用，但对大型程序来说，这些形式化的方法却无济于事。目前，人们只能使用测试技术来尽量发现程序中所含的错误。如果程序运行是确定的，程序在一数据集上的运行结果就决定了程序在此数据集上的正确与否，然而，程序在有限测试情况下执行正确并不能保证程序完全正确。

我们下面从形式化的角度来讨论程序正确性证明，尽管这些方法离实用还有一段距离，但它们是研究应用系统的基础，我们重点讨论基于I/O规范的程序正确性证明。

5.1 基于I/O规范的证明 一个程序P的I/O规范可表示为{Q}P{R}，只要程序的输入状态满足Q，那么，输出状态应满足R。令O_1，O_2，…，O_n是P中的全体对象，那么，P的一个合法输入状态$d_0=(a_1,a_2,\cdots,a_n)$，a_i是对象O_i的取值，$1\leqslant i\leqslant n$，并且$\vdash Q(d_0)$；同样，可以定义P的合法输出状态。从效果上看，P定义了一个执行函数E(P，d_0)，它把程序的输入状态d_0映射到输出状态d_f，这里要求P终止，否则E(P，d_0)没有定义。

定义1：程序P是完全正确的，当且仅当$(\forall d_0)(\vdash Q(d_0))\rightarrow\vdash$(P终止)$\wedge\vdash R$(E(P，$d_0$))。记为：$\vdash\{Q\}P\{R\}$。

定义2：程序P是部分正确的，当且仅当$(\forall d_0)(\vdash Q(d_0)\wedge\vdash$(P终止))$\rightarrow\vdash R$(E(P，$d_0$))。记为：$\vdash\{Q\}P\{R\}^+$。

程序P由多个语法单位（一个语法单位可以是一条或多条指令）组成，P的执行效果通过多个语法单位的执行效果来实现，因此，语法单位执行的正确性决定了程序P的正确性。令$\overline{Q}$和$\overline{R}$是语法单位S_i的输入条件和输出条件，那么，$\vdash\{\overline{Q}\}S_i\{\overline{R}\}$ iff $(\forall d_0)(\vdash\overline{Q}(d_0))\rightarrow\vdash(S_i$终止)$\wedge\vdash\overline{R}$(E($S_i$，$d_0$))，$\vdash\{\overline{Q}\}S_i\{\overline{R}\}^+$ iff $(\forall d_0)(\vdash\overline{Q}(d_0)\wedge\vdash(S_i$终止))$\rightarrow\vdash\overline{R}$(E($S_i$，$d_0$))。一个程序

的执行效果不仅依赖于语法单位的执行效果，而且也依赖于语法单位执行的顺序。已知程序$P=S_1; S_2; \cdots; S_n$，d_0是初始状态，一个依赖于d_0的程序执行踪迹T由下面的序列定义，$T(P, d_0)=S_{i_1}, S_{i_2}, \cdots, S_{i_j} \in \{S_1, S_2, \cdots, S_n\}$。如果程序P在$d_0$下终止，$T(P, d_0)$是一个有穷序列。否则，$T(P, d_0)$是一无穷序列。一个依赖于$T(P, d_0)$的计算是$C(P, d_0)=d_1, d_2, \cdots$，这里$d_j$是执行$S_{i_j}$后到达的状态。

从程序正确性的定义看出，执行函数$E(P, d_0)$在证明过程中将起到主导作用，而执行函数的定义又依赖于程序的语义。有三种定义程序设计语言语义的方法：①操作语义，②指称语义，③断言语义。当用操作语义方法来定义程序P的语义时，首先定义一个抽象机器M，使得P中的语法单位都可以使用M上的状态和操作来定义，然后把P翻译成M上的等价程序M(P)，执行M(P)。当使用指称语义方法来定义程序P的语义时，P中语法单位的语义通过语义评价函数(Semantic Valuation function)来定义，语义评价函数把语法单位映射到相应的值（可以是数，真假值，函数等等）；指称语义与操作语义的区别在于操作语义要求考虑一个语法单位的实现细节，而指称语义则不必。它把注意力放在输入和输出的关系上。当使用断言语义方法来描述程序P的语义时，引入输入断言和输出断言，输入断言描述程序输入应满足的条件，输出断言描述程序输出应满足的条件，即：$\{Q\}P\{R\}$。我们下面针对执行函数$E(P, d_0)$的不同定义来讨论程序正确性证明的方法。

方法1：执行函数的操作语义定义

定义执行函数$E(P, d_0)$的一个简单方法是符号执行，在程序P中，用符号化的对象值作为对象的输入，而不使用实际的对象值，这样，程序的执行就符号化了。例如，用$V(x)$标明对象x的符号值，并给对象A, B分别赋于符号化的值a, b，那么：

$C:=A+2\times B$　符号执行后产生符号值$V(c)=a+2\times b$

$D:=C-A$　符号执行后产生符号值$V(b)=a+2\times b-a=2\times b$

很显然，对象符号化后，程序状态d可表示$(V(n_1), V(n_2), \cdots, V(n_i))$，$V(n_i)$是对象$n_i$的符号值。

一个语法单位S的执行效果$E(S, d)$可由一个函数$f(d)$定义，即：$E(S, d)=f(d)$，这样，程序的一个计算可表示为：

$$C(P, d_0)=d_1, d_2, d_3, \cdots=f_1(d_0), f_2(d_1), f_3(d_2), \cdots=f_1(d_0), f_2(f_1(d_0)), f_3(f_2(f_1(d_0))), \cdots)$$

若程序P从初始状态d_0出发能到达终止状态，则程序P的执行函数可由下面的等式定义：

$$E(P, d_0)=f_{|C(P, d_0)|}(f_{|C(P, d_0)|-1}(\cdots f_1(d_0))\cdots)$$

$|C(P, d_0)|$表示计算的长度。对象符号化后，程序P的语义可使用$R(V(n_1), V(n_2), \cdots, V(n_n))$来定义，所以，为了证明程序P在初始状态$d_0$下是正确的，只需证明下面的等式：

$$R(V(n_1), V(n_2), \cdots, V(n_n))=E(P, (V(n_1), V(n_2), \cdots, V(n_n)))$$
$$=f_{|C(P, (V(n_1), \cdots, V(n_n)))|}(\cdots f_1((V(n_1), \cdots, V(n_n)))\cdots), \quad (5\text{-}1)$$

由于公式（5-1）的定义依赖于执行踪迹$T(P, d_0)$，为了证明程序P是正确的，我们必须证明对不同的执行踪迹T，公式（5-1）成立。这是符号执行的最大弱点。

方法2：执行函数的指称语义定义

我们可以把执行函数$E(P, d_0)$定义为一个二元关系R，$R\subseteq D\times D$，D是P的全体状态组成的集合。为了建立关系R，把程序P看成由语法单位$S_1, S_2, \cdots, S_m$构成，每个语法单位S_i也由一个二元关系$R_i\subseteq D\times D$定义，然后，关系R由一个语义评价函数$f(R_1, R_2, \cdots, R_m)$定义。

已知可以用$R(V(n_1), \cdots, V(n_o))$定义程序P的语义，为了证明程序的正确性，需要证明下面的等式：

$$R(V(n_1), V(n_2)\cdots, V(n_n))$$
$$=E(P, (V(n_1), V(n_2), \cdots, V(n_n)))$$
$$=f(R_1(V(n_1), \cdots, V(n_n)), \cdots, R_m(V(n_1), \cdots, V(n_n)))$$

初看起来，该方法类似于符号执行方法，因为它们都使用$R(V(n_1), \cdots, V(n_n))$，然而，指称语义方法不依赖程序执行踪迹$T(P, d_0)$来定义执行函数

$E(P, (V(n_1), \cdots, V(n_n)))$，关系$R_i$蕴含地定义了$S_i$的符号值。

方法3：执行函数的断言语义定义

在这种方法中，执行函数使用谓词变换来定义，一个谓词变换是一个函数，它把由断言和语法单位构成的二元组映射到另一个断言。用G和H分别表示语法单位S的输入断言和输出断言，那么，S的语义可通过谓词变换TR(S, G)=H来定义。考虑程序P在输入状态d_0情况下的踪迹$T(P, d_0)=S_{i1}, S_{i2}, S_{i3}, \cdots$，让$TR(S_{i1}, Q)=Q_1$，$TR(S_{i2}, Q)=Q_2$，…，若P终止，那么$K=|T(P, d_0)|$是一个有限数，这个变换过程可表示为：$\{Q\}S_{i1}\{Q_1\}S_{i2}\{Q_2\}S_{i3}\cdots S_{ik}\{Q_k\}$。所以，为了证明$\vdash Q(d_0)\rightarrow\vdash R(E(T(P, d_0), d_0))$，我们只需证明$TR(S_{ik}, Q_{k-1})=Q_k=R$，即：$TR(T(P, d_0)d_0)=R$。Floyd给出了归纳断言方法[4]，Hoare给出了公理方法[5]，由于这两个方法比较复杂，这里不给予介绍。

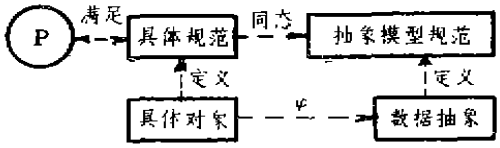
5.2 基于操作规范的证明 假设一个操作规范OS正确表达了用户的要求，P是实现OS的一个程序，当要证明程序P满足OS时，我们只需证明P与OS是等价的。Mecarty和Greif在这方面作了一些工作，当OS与P都可用递归描述时，有一些证明方法。

5.3 基于代数规范的证明

假设AS是一个代数规范，P是AS的一个实现，为了证明P是正确的，我们可以证明AS中的每一个公理被程序P所满足[6]。由于程序P一般使用具体对象来实现代数规范所确定的功能，所以，在证明程序正确性之前，需要作两件预备工作：①使用代数方法形式描述具体对象；②建立抽象对象与具体对象间的关系。

5.4 基于抽象模型规范的证明

假设AMS是一个抽象模型规范，它所依赖的数据抽象是DA，P是AMS的一个实现，为了证明P是正确的，我们从P中推导出一个代数形式CMS，CMS所依赖的对象是具体的，然后建立一个从CMS到AMS的同态映射[8]。例如，我们有一个关于栈的抽象模型规范，所依赖的数据抽象是序列，而实现栈时使用了数组，为了证明实现栈的程序P满足抽象模型规范AMS，可以建立一个映射：map(a, pointer=seq(a, 1, pointer)，它表明一个用数组表示的栈可用序列来表示，然后证明AMS中的每一个公理被P所满足。我们可以用下图表示这种方法的大意。



本文讨论了形式规范和程序生成技术，书写形式规范的目的是为了正确生成程序，而正确生成程序需要形式规范的引导。自动程序设计是软件工程的最终目标，而生成形式规范是自动程序设计过程的第一步，形式规范的表示方式以及生成技术直接影响到下一阶段的工作，目前，形式规范技术的发展还远远达不到自动程序设计的要求，因此，我们必须加紧这方面的研究工作。自动程序设计的第二步是从形式规范生成目标程序，变换技术是人们关注的重点。怎样保证目标程序满足形式规范？这是自动程序设计成败的关键，目前的技术还难于做到这一点。因此，还需我们付出艰苦努力。

作者在完成本文的过程中，得到了陆汝钤、周龙骧、陆伟明三位教授的指导和帮助，在此表示感谢。

参 考 资 料

[1] Readings in Artificial Intelligence and Software Engineering, 1986, Morgan Kaufmann Publishers, Inc.

[2] Proceedings, Fourth International Workshop on Software Specification and Design, April 3—4, 1987, Monterey, California, USA, Computer Society Press of IEEE.

[3] Goodenough, J.B. & Gerhart, S.: "Toward a Theory of Test Data Selection", IEEE Trans. on Software Engineering, Vol.1, No.2, 1975.

[4] Floyd, R.W. "Assigning Meaning to Programs" Proc. Symposium on Applied Mathematics, American Mathematical Society, Vol.19, 1967.

[5] Hoare, C.A.R, "An Axiomatic Approach to Computer Programming" Comm.

（下转27页）

有的软件测试技术和工具从理论上进行归纳总结，并在此基础上从理论的角度出发对软件测试方法学进行有益的探索.与此同时还要从工具的角度出发，在实践中验证方法的可行性和有效性。

参考文献

1 Beizer, B., Software Testing Techniques.

2 Ramamoorthy, C. V., and Ho, S. F., Testing large Software with automated Software evaluation Systems, IEEE Trans. Software Eng., Vol. SE-1, NO.1, March 1975.

3 Clarke, L. A., A System to Generate Test Data and Symbolically Execute Programs, IEEE Trans. Software Eng., Vol. SE-11, No.3, March 1985.

4 Miller, E., Program testing, Art Meets Theory, Computer, Vol. 10, No. 1, January 1977.

5 King, J.C., Symbolic execution and program testing, CACM July 1976.

6 Howden, W.E., Experiments with a symbolic evaluation System, Proc.1976 Nat. Computer Conf., June 1976.

7 Osterweil, L. J. and Fosdick, L. D., DAVE—A Validation Error Detection and Documentation System for Fortran Program, Software Practice and Experience, Vol.6, Oct.—Dec. 1976.

8 Osterweil, L. J. and Fosdick, L. D., Some Experience with DAVE—A Fortran Program Analyzer, in AFIPS Conf. Proc., Vol. 45, Montvale. NJ, AFIPS Press, 1976.

9 wilson, C. and Osterweil, L. J., Omega—A dataflow analysis tool for the C programming language,IEEE Trans. Software Eng., Vol. SE—11, No. 9, September 1985.

10 Richard,N.T. and Osterweil, L. J., Anomaly Detection in Concurrent Software by Static Data Flow Analysis IEEE Trans. Software Eng., Vol. SE—6, No.3,May 1980

11 Voges,U., Gmeiner, L. and Mayrhauser, A. A. V, SADAT—An Automated Testing Tool, IEEE Trans. Softwre Eng., Vol. SE—6,No. 3,May 1980.

12 Senn, E. H., Ames, R. R., and Smith, K. A., Integrated Verification and Testing System(IVTS)for HAL/S Program, in 1983 SOFTFAIR.

13 Andrews, C. L. and Dehaan, W.R., RXVP80 The Verification and Validation System for Fortran and COBOL,in 1983 SOFTFAIR.

14 Tischler, R., Schaufler, R. and Payne, C.,Static Analysis of Programs as an Aid to Debugging. SIGPLAN Notices Vol. 18, No. 8, August, 1983.

15 With, N., Programming in Modula-2, Springer—Verlag, 1984.

16 王来强 ITEM：一个集成的软件测试系统 南京大学硕士论文 1987.

（上接52页）

of the ACM, Vol.12, №.10, 1969.

〔6〕 Guttage, J.U. & Horowitz, E. &Musser, D.R.: "Abstract Data Types and Software Validation", Comm.of the ACM, Vol. 21, №.12, 1978.

〔7〕 Flon, L. & Misra, J.: "A Unified Approach to the Specification and Verification of Abstract Data Types." Proc.Specification of Reliable Software Conf., Cambridge, Mass., 1979

〔8〕 Hoare,C.A.R., "Proofs of Correctness of Data Representations", Acta Informatica, Vol.1, №.4, 1972.

〔9〕 Formal Methods of Program Verification and Specification, PRENTICE-HALL, INC., Englewood Cliff, New Jersey 07632.

〔10〕 Proceedings: 9th International conference on Software Engineering March 30-April2, 1987, Monterey, California, USA. Computer Society Press of IEEE.

〔11〕 Algebraic Approaches to Program Semantics, 1986, Springer-Verlag New York Inc.