程序验证 模块法 (18)

计算机科学 1994Vol. 21 №. 1

75-79

支持程序验证的模块方法 TP31

刘定飞 钟 珞
(武汉工业大学自动化系 武汉 430070)

摘要 本文给出一种支持程序验证的模块方法，并讨论了基于函数语义的模块验证。程序(特别是大型复杂的程序)可划分为若干个模块，模块本身又可划分为若干个更小的子模块，其验证独立进行。将独立验证过的各子模块复合，完成上级模块或程序本身的验证。

关键词 抽象数据类型，模块，程序验证，函数语义，模块方法。

一、引言

程序正确性证明(即程序验证)是从理论上研究和保证程序的正确性。这方面的研究早在五十年代就为 Turning 等人所注意。六十年代后半期 Floyd、Hoare 等人提出了不变式断言和公理化方法等。七十年代以来，出现了许多不同的证明方法，以此产生了许多"证明了"的程序。目前，程序验证是软件工程中一个十分活跃且富有生命力的课题。

但是，程序验证方法有其局限性，有效地使用这些方法需要相应的系统支持，而且对于实际中常见的大型复杂的程序，使用目前所研究的这些方法和技术还存在许多困难。

二、模块与程序验证

模块在软件开发中起着十分重要的角色：首先它有着抽象作用，程序员可实现过程抽象和数据抽象，有助于人们集中去考虑做什么；其次，它将程序段相对独立起来，避免其余部分的干扰，使人们以最小代价可靠地对程序进行修改。

数据抽象的本质可用具体世界和抽象世界的关联图表示。具体世界由传统的编程语言直接操纵的对象组成；抽象世界由程序员从宏观考虑的对象组成。在各个世界中，对象之间存在着映射。具体世界与抽象世界可由图 1 所示的一个表示函数的映射联接起来。

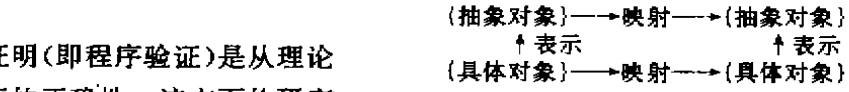

图 1 具体世界与抽象世界关联图

数据抽象须验证其正确性，即从具体到抽象的映射是正确的。抽象世界也可由具体世界中的程序必须做什么来定义。但是，程序规格说明并不总是函数型的，这对验证是一个较大的障碍。本文提出的验证理论对非函数型的规格说明不必作修改，同时对函数型的情况将显得更简单。

三、模块的函数语义

函数语义是关于程序含义的语义，在数学上其符号定义是精确的。我们这里只讨论一个本文所需的子集。

状态作为最基本的语义，将程序标志符与值联接在一起，状态是变量名到变量当前值的一个函数映射。

表达式是状态到值的映射。状态 S 的整数常量的语义是常量的整数表示。状态 S 的标志符 V 的语义是 S 赋给 V 的值，记为 $S_{(v)}$。基于变量与常量，可归纳得出整数表达式的语义。如果表达式为 x+y，那么状态 S 中，它的值是状态 S 中 x 的值加上状态 S 中 y 的值。我们采用类似于 Kleene 法的形式来表示语义函数。

定义：〖C〗，整数常量C；

〖V〗$_{(S)}$=$S_{(V)}$，标志符V，状态S；

〖X+Y〗$_{(S)}$=〖X〗$_{(S)}$+〖Y〗$_{(S)}$，表达式X、Y，状态S。

同样可以定义减法、乘法等。

对布尔表达式有类似如下定义：

若〖X〗$_{(S)}\geqslant$〖Y〗$_{(S)}$，则X>=Y为真；

若〖X〗$_{(S)}<$〖Y〗$_{(S)}$，则X>=Y为假。

将程序语句看成状态到状态之间的映射，令V是变量，E是表达式，V=E赋值语义为：

〖V=E〗=｛(S,T)：T=S除〖V〗$_{(T)}$=〖E〗$_{(S)}$外｝

即除了表达式值已赋给变量外，赋值语句的输入与输出状态将相同。

也可归纳定义其它程序复合的语义，如：〖{A;B;}〗=〖A〗∘〖B〗，其中A、B为语句，∘为函数复合。

更复杂的定义有：

〖if(B)S〗=｛(U，〖S〗$_{(U)}$)：〖B〗$_{(U)}$｝∪｛(U,U)：¬(〖B〗$_{(U)}$)｝

这是一个有布尔表达式B及嵌套语句S的条件语句，两个集合分别表示输入为“真”及“假”的情况。

再看循环语句的定义：

〖while(B)D〗=｛(T,U)：∃ k≥0，使得

∀ 0≤i≤k

(〖B〗(〖D〗$^{i}_{(T)}$)∧¬〖B〗(〖D〗$^{k}_{(T)}$)∧〖D〗$^{k}_{(T)}$=U)｝

即存在一个自然数k，使得循环k后，T变成T状态中D的k重复合。我们引入以下定理，以便进行while语句验证。

定理1(while语句验证) 设W是程序段while(B)D，那么：f=〖W〗当且仅当：

1)f的域=〖W〗的域，记为：dom(f)=dom(〖W〗)；

2)f$_{(T)}$=T，当¬〖B〗$_{(T)}$时；

3)f=〖if(B)D〗∘f

这个定理隐含着证明循环W的一个方法：首先，推测或计算出一个判别函数f。然后，用定理中的三个条件来检验判别函数是否为真。

过程调用语句是语句语义定义的难点。调用的语义函数是经正文代替实现参数传递后的被说明程序对象的函数。当在过程P(其程序对象为T)的说明中有一个参数x，通过参数A对P的调用的语义为：

〖$P_{(A)}$〗=〖T←A\x〗

T←A\x表示x的每一次在T中出现都用A代替的T。

过程调用的语义函数为数据抽象关联图中的具体对象提供了精确形式。具体对象间的映射是过程调用的语义函数。但是，抽象对象及其变换很少有数学方法能描述它们。然而，我们使用了标志符和状态。图中最重要的元素是一个典型具体对象与抽象对象之间的关联性，即表示函数。这个映射常常是多对一的，因为具体实现并非唯一。在数据抽象关联图中：

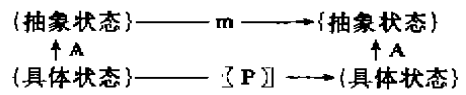

图2 数据抽象关联图

抽象映射为m，表示映射为A，具体映射为过程P。我们说抽象图2是可交换的，当且仅当从左下角开始，经两条可能的路径可得到同一结果，即A∘m⊆〈P〉∘A。从抽象函数为一个规格说明的观点来看，可交换图对应于程序的一个正确实现。

四、程序函数

在具体世界中，对每一初始状态X，经过程P，映射到最终状态Y。如果对每一给定的X，状态Y是唯一的，那么｛(X,Y)｝便定义了程序函数。程序函数是对程序功能的精确描述。

我们给出一个例子，来计算其程序函数。

```
void ExpRat(R,N)
Rational *R;
int N;
```

```
/*abs:(N>=1→R=R**N|N<1→)
   con:((N>=1→R.Num,R.Den=R.Num**N,
        R.Den**N)|(N<1→))
*/
{
    Rational T;
    int I;
    T.Num=R->Num;
    T.Den=R->Den;
    I=1;
    while(I<N)
      {
      I=I+1;
      T.Num=T.Num*R->Num;
      T.Den=T.Den*R->Den;
      }
    R->Num=T.Num;
    R->Den=T.Den;
    }
```

其中类型 Rational 的定义如下：

```
typedef struct Retional{
        int Num;
        int Den;
};
```

为方便计算，我们用“**”表示乘方，将“->”和“·”符号看成一样。通过 While 语句函数 F 和两个赋值语句函数来构造函数：

```
    (I,T.Num,T.Den=1,R.Num,R.Den)·
  ((I<N→I,T.Num,T.Den=N,T.Num*R.Num
  **(N-I),T.Den*R.Den**(N-I))|(I>=N
  →))·
  (R.Num,R.Den=T.Num,T.Den)
```

结果为：

```
((I<N→I,T.Num,T.Den,R.Num,R.Den=N,R.
Num*R.Num**(N-1),R.Den*R.Den**(N-
1),R.Num*R.Num**(N-1),R.Den*R.Den**
(N-1))|
  (I>=N→1,T.Num,T.Den=1,R.Num,R.Den))
```

忽略对局部变量的影响，化简为：

```
  ((I<N→R.Num,R.Den=R.Num**N,R.Den
  **N)|(I>=N→))
```

这和“con”注释中给出的函数是相同的。这就是程序函数。下面我们进行证明。

赋值语句序列的两个函数显然是正确的，我们仅须对 while 语句进行证明。

设循环体为 S，运用第三节的定理 1，预期函数 F 和 〚while(I<N)S〛 是一致的，如果以下成立：

1)dom(F)=dom(〚while(I<N)S〛)；

2)$F_{(T)}$=T，当¬〚I<N〛$_{(T)}$时；

3)F=〚if(I<N)S〛·F

F 的域 I<N OR I>=N 为真。如果 I>=N，循环结束。如果 I<N，执行 while，I 加 1，循环逼近结束。这样，第一条件得到满足。

第二个条件要求当 while 条件不再满足时，F 成立。这恰恰是 F 定义中的最终情况。

对第三个条件，if(I<N)S 的函数为：

```
((I<N→I,T.Num,T.Den=I+1,T.Num*R.Num,
    T.Den*R.Den)|(I>=N→))
```

〚if(I<N)S〛·F 的复合为：

```
((I<N→I,T.Num,T.Den=I+1,T.Num*R.
Num,T.Den*R.Den)|(I>=N→))·
((I<N→I,T.Num,T.Den=N,T.Num*R.Num
**(N-I),T.Den*R.Den**(N-I))|(I>=N→))
```

对于所存在的四种情况，我们构造四个跟踪表，不难得出四个函数(有些情况无意义)。然后将四个函数复合到一起，可得：

```
((I+1<=N→I,T.Num,T.Den=N,T.Num*R.
Num**(N-I),T.Den*R.Den**(N-I))|(I>=N
→))
```

因为 I+1<=N 与 I<N 是等价的，因此上式与 F 是一致的。while 验证定理的第三个条件得以满足。

以上所得到的程序函数将用于第六节。

五、支持程序验证的模块方法

以上讨论了函数语义及其对程序函数的计算。这一节，我们将讨论模块方法。

模块的调用程序向模块传递外部信息，在模块内作为隐蔽的内部形式，并作相应的操作。最后，被内部隐蔽的值又必须和模块外面世界通讯，转变为外部有用的形式。

例如，一个实现复数的模块，其原始数据可能以两个实数(浮点数)的形式存在，一个表示模，另一个表示角度。COMPLEX 模块的输入转换子程序可如下说明：

```
void InComplex(Mag,Ang,Val)
float Mag,Ang;
COMPLEX * Val;
```

程序员可从读入一对浮点数值开始，得到一个 COMPLEX 类型的值。类似地：

```
Void OutComplex(Mag,Ang,Val)
float *Mag,*Ang;
COMPLEX Val;
```

可用来输出结果。而：

```
void AddComplex (A,B,Result)
COMPLEX A,B,*Result;
```

将实现复数相加。

应用模块时，程序员将其抽象动作进行推理。设想计算机中并不存在的对象以及模块所实现的直观对象之间的映射。设将一对浮点值映射到 COMPLEX 型的抽象函数是 C，那么输入转换图为：

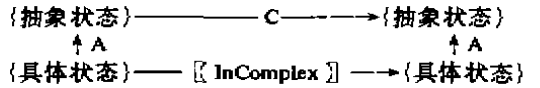

图 3 输入转换图

构成抽象值的原始数据取决于具体状态，抽象值由表示映射生成。这样，在输入转换图的左边，映射 C 和 〖InComplex〗 只取从具体状态输入的值。因此输入图可简化为有公共域的三角形：

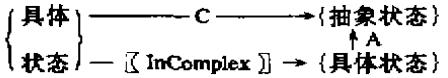

图 4 输入转换简化图

程序员大脑中对模块内每一操作都有其抽象函数，使用这些函数，可对程序进行推理。在操作图中，先考虑顶层的一个操作序列。从不包含模型类型 T 的对象的三角形开始，由抽象操作 InT 创建 T 的对象，然后经抽象操作 m_1T，m_2T，…等，最后由 OutT 变回到已知的值（另一个三角形）。这些序列的抽象含义是模块外面的值经函数：InT。m_1T。m_2T。…。OutT 进行转换，其中间值是模块类型的抽象值。

实际上，计算是由图的底层实现的。从值开始，经连续变换，其间总是以语言的内部形式存在。如果函数的过程分别为：PInT，Pm_1，Pm_2，…，POutT，那么实际上计算如下：

〖PInT〗。〈Pm_1〉。〖Pm_2〗。…。
〖POutT〗

我们将这一复合叫扩展图：

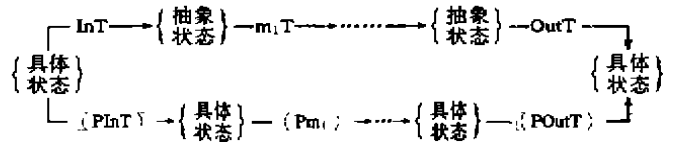

图 5 扩展图

程序的正确性就表示为扩展图是可交换的。即实现是正确的，当且仅当：

InT。m_1T。m_2T。…。OutT⊆〖PInT〗。〈Pm_1〉。〖Pm_2〗。…。〈POutT〉

但是，软件开发中正确性证明必须从每一操作的独立证明，而不是从操作序列着手的。下面，我们给出证明定理。

定理 2 一个模块的实现是正确的，当满足条件：存在一个表示函数 A，使得每一操作图都经 A 是可交换的；且 A 的内部类型是一致的。

证明 （从略）

因此，实现一个模块的验证是：选择合适的表示函数；计算每一过程的语义；证明对预期的抽象函数，已计算的语义和已选的表示函数每一操作图都是可交换的。

六、一个验证实例：有理数

第四节我们给出了 Rational（有理数）的类型说明及过程（函数）ExpRat，说明了有理数的抽象世界，而具体状态是由整数值（N，D）对偶表示的，相应的值是一个分子为 N、分母为 D 的分数。当定义了 N 和 D≠0 时，有理数有定义。从具体状态到抽象状态的表示映射 Arat 为：

Arat＝{(S.T)：T＝S 除非 X.Num 和 X.Den 形式的所有标志符被 x 所代替，〖X〗$_{(T)}$＝〖X.Num〗$_{(S)}$/〖X.Den〗$_{(S)}$，〖X.Den〗$_{(S)}$≠0}

这个表示较麻烦。可由一个保护赋值语句表示状态映射的条件赋值来取代。设抽象状态的唯一变量为 R：

Arat＝(R.Den＜＞0→R＝R.Num/R.Den)

第四节的 ExpRat 函数将实现有理数 R 的 N 次幂。抽象（"abs"）和具体（"con"）世界的注释描述了这种目的。"abs"注释所描述的抽象函数为：

ExpRatabs＝{(S,T)：〖N〗$_{(S)}$≥1∧T＝S 除非〖R〗$_{(T)}$＝〖R〗$_{(S)}$＊＊〖N〗$_{(S)}$}∪{(S,S)：〖N〗$_{(S)}$＜1}

利用第四节已计算出的程序函

数，只要证明以下图可交换，便可证明 ExpRat 的正确性：

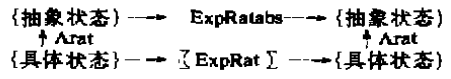

图 6　ExpRat 状态转换图

即：Arat ∘ ExpRatabs⊆〖ExpRat〗∘ Arat。

Arat 和 ExpRatabs 的复合为：

(R. Den<>0→R=R. Num/R. Den) ∘ ((N>=1→R=R * * N)|(N<1→))

相应复合的两种情况，我们使用两个跟踪表来计算：

表 1

部　分	条　件	R	R. Num	R. Den
Arat	R. Den ( )0	R. Num/R. Den		
ExpRatabs	N>=1	(R. Num/R. Den) * * N		

表 2

部　分	条　件	R	R. Num	R. Den
Arat	R. Den( )0	R. Num/R. Den		
ExpRatabs	N<1			

结果函数为：

((R. Den<>0 AND N>=1→R=(R. Num/R. Den) * * N)|

(R. Den<>0 AND N<1→R=R. Num/R. Den))

〖ExpRat〗与 Arat 的复合为：

((N>=1→R. Num, R. Den=R. Num * * N, R. Den * * N)|(N<1→)) ∘

(R. Den<>0→R=R. Num/R. Den)

与上述类似，通过构造两个跟踪表可得出它们的复合函数：

((N>=1 AND R. Den<>0→R=R. Num * * N/R. Den * * N)|

(N<1 AND R. Den<>0→R=R. Num/R. Den))

可见，它和第一个复合函数的结果是等价的。因此，图 6 是可交换的，即 ExpRat 是正确的。

七、结束语

我们将程序划分为若干模块，对每一模块独立进行验证。每一模块又划分为若干子模块，各子模块又划分为更小的子模块。如此反复，直至显而易证的基本操作或是已经证明的子模块。这样，我们能够利用已取得的结果，利用已经产生了的"证明"和基本操作，通过复合来构造大型复杂软件的验证。

从更长远设想，我们可基于这种思想及模块方法，研究软件验证的辅助工具和自动验证系统。

参考文献

[1]Z. Manna, Mathematical Theory of Computation, McGraw-Hill, New York, 1974

[2]C. A. R. Hoare, Proof of Correctness of Data Representation, Acta Inform., Vol. 1, pp. 271-281, 1972

[3]D. Dries, The Science of Programming, Spring Ger-Veriag, 1981

[4]仲萃豪等，程序设计方法学，北京科学技术出版社，1985 年

[5]吴锡琪，钟珞，刘定飞，程序结构分解及复杂性度量，小型微型计算机系统，Vol. 12 №1, pp. 39-46, 1991 年

[6]钟珞，抽象数据类型及其代数描述的直观讨论，软件产业，№11, pp. 16-18, 1987 年

[7]钟珞，基于抽象数据类型的程序设计语言的设计，计算机应用研究，Vol. 8 №1, pp. 1-5, 1991 年

[8]赵恳，钟珞，背景关联世界中抽象数据类型的关联性，四川轻村学院学报，Vol. 7 №1, pp. 76-82, 1992 年

[9]钟珞，用队列计算表达式的值，上海微型计算机，№4, pp. 55-57, 1986 年

[10]钟珞，吴锡琪，抽象数据类型形式变换的研究，小型微型计算机系统，Vol. 14 №7, 1993 年

[11]钟珞，吴锡琪，函数型程序设计中的推测计算，武汉工业大学学报，Vol. 15 №4, 1993 年