第 43 卷 第 6A 期
2016 年 6 月

计 算 机 科 学
Computer Science

Vol. 43 No. 6A
June 2016

基于抽象状态的类的行为规格化方法

王 伟 丁二玉 骆 斌

(南京大学软件学院 南京 210093) (计算机软件新技术国家重点实验室 南京 210093)

摘 要 为独立方法定义严谨的规格可以保证程序的正确性。但是在面向对象的程序中,方法之间因为共享属性而相互影响,这就需要能够反映方法间影响的规格化方法。研究者们使用抽象变量、状态抽象、堆、查询等多种方法进行了尝试。文中给出一种基于抽象状态的类的行为规格方法,该方法基于抽象状态解决了类方法间的共享依赖和相互影响,同时实现了规格与实现的独立描述与运行时自动化验证。

关键词 抽象状态,共享依赖,规格化方法

中图法分类号 TP311.1 文献标识码 A

Behavior Specification Method of Class Based on Abstract State

WANG Wei DING Er-yu LUO Bin

(Software Institute, Nanjing University, Nanjing 210093, China)

(State Key Laboratory for Novel Software Technology, Nanjing 210093, China)

Abstract Defining the specifications of the method can reduce the software error, and ensure the correctness of the program. But in object-oriented programs, methods influence each other, so better specifications methods are needed . The researchers try a variety of methods, such as abstract variable, state abstraction, heap, inspector methods and so on. In this paper, we gave a behavior specification method of class based on the abstract state. The method depends on abstract state to solute the shared dependency and influence between the class specification method, and realizes the independent description and validation runtime cohesion between specification and implementation.

Keywords Abstract state, Sharing dependency, Specification method

1 引言

保证程序的正确性及减少软件错误一直是研究者们关注的问题。Hoare[1]提出了基于“前置后置条件”的接口规格方法。但现在的软件系统越来越复杂,在规模和功能上都有大幅扩展,因此需要发展 Hoare 的方法以适应新的大规模复杂系统开发。

扩展 Hoare 方法的一个关键是将应用于函数的规格机制扩展到为类建立形式化规格,即利用前置条件、后置条件描述类的各个方法,并集成为一体表现类的行为。相比于单个方法的规格,为类建立规格的困难在于:1)类的不同方法之间是相互联系、相互依赖的,类的规格化方法必须明确描述这种联系——其背后的基础是被共享的属性数据;2)规格是抽象机制,不应该涉及类的内部实现,包括属性在内。

为了建立正确的类规格,近些年来,研究者们进行了抽象变量、接口规格、状态抽象、查询方法等不同方面的尝试,并取得了很大的进展。

本文提出了一种以状态抽象方法为基础的基于抽象状态的类的行为规格化方法。该方法的基础是使用抽象状态抽象类的属性,使用状态依赖描述类方法间的共享依赖。与其它状态抽象方法相比,本文方法的扩展是明确了状态的主体细分,通过区分相对状态与绝对状态,将一直被忽略的“属性与接口输入/输出参数”之间的联系纳入了规格范畴。

本文第 2 节介绍相关工作,它们为本研究提供了启发和理论依据;第 3 节给出了形式化方法的工作思路;第 4 节给出了所提规格方法的完整形式化定义;第 5 节用例子来说明形式化方法的应用;最后是进一步工作的展望。

2 相关工作

为保证程序的正确性,Hoare[1]提出了关于接口的规格方法,即若一种方法在执行前满足前置条件,运行后满足后置条件,那么这种方法就是正确的。Hoare 提出的这个结论为规格化发展起到了奠基的作用,也为类中接口的规格化提供了理论基础。

针对面向对象方法,人们需要发展 Hoare 的理论以适应面向对象的特殊性。Guttag[2]使用抽象的数据类型来降低设计和实现大型软件的复杂度,用代数公理证明规格的正确性。Goguen[3]和 Zilles[4]都着重强调代数化方法,使用代数方法验证抽象数据类型的规格化的正确性。使用抽象数据类型的代数规格法虽然可以描述方法之间的相关关系,但是却失去了对单个方法的准确规格定义。

针对各方法的精确规格定义,Meyer[5]提出契约式编程延续了对方法的精确定义,但其不足是依赖内部状态来体现方法之间的相互依赖。Robert[6]也引入契约来表达接口的前

王 伟(1991—),男,硕士生,主要研究方向为软件需求工程、软件规格化;丁二玉(1980—),男,博士,讲师,主要研究方向为软件需求工程、软件规格化;骆 斌(1967—),男,教授,博士生导师,主要研究方向为数据库、操作系统、人工智能。

置条件和后置条件,在调用这个接口时可以用前置条件和后置条件验证正确性。Robert 在文献[7]中进一步介绍了一种带有断言的 lambda 计算来检测系统运行,确保当契约被违背时能够及时标记错误。但契约式以可被编译的形式将前置、后置条件断言写在程序中,使得断言和程序语句很难区分,阅读难以辨别,还影响了程序的运行效率。

Yoonsik[8]尝试使用模型方法和模型变量来解决上述问题,并建立了规格方法 JML。JML 不需要直接引用具体的程序状态,而是使用抽象的模型变量来代表内部数据状态,模型方法使用模型变量进行规格,从而避免了实际代码和抽象规格的混乱。但这种方式也有缺陷:模型变量难以定义,如果将模型变量和代码实现过度脱离,则会难以实现运行时断言检查;如果将模型变量直接映射到内部数据,又会间接造成规格与代码的混淆。

也有研究者使用状态抽象的方法进行类的规格定义。Hoffman[9]给出了一个状态抽象模块,它提供了一个好的数学模型,并对状态抽象有新的理解。它和数据精炼有很多相似之处,与代数规格语言有很大的联系,使得设计决策更加抽象化,不必关心具体状态的值和顺序,提高了抽象规格的清晰性和易用性。Hatcliff[10]提供两种分析方法(理论证明和数据流)来证明规格内部允许状态改变,将状态封装到内部不会影响规格。虽然这些研究考虑到了状态的抽象,但是未明确方法和状态的依赖问题,也未提供抽象状态与实现问题的关联机制。状态的抽象提高了接口的抽象程度,但是状态的变化使得模块的接口难以定义;有很多种类型的接口定义,如基于性质、基于模块、方法规格说明等,但是状态抽象是完全独立于这 3 个而存在的。

堆方法是基于抽象状态方法扩充出来的另一种思路。堆方法是在对象引用与对象状态之间建立映射,堆中可以找到由区域引用的对象的状态。通过在程序中检验堆状态,可以验证程序的正确性和完备性。Jacobs[11]详细叙述了堆方法的相关表示法和词语句法,并引入多个状态变量证明了正确有效的程序不会在执行过程中异常停止。然而堆方法也有缺点,它定义的对象过多,容易造成空间爆炸。

面向对象规格的难点在于方法间的相互影响,人们也尝试界定这种影响。在面向对象的语言中,类总是把状态和能改变这些状态的行为放在一起,而实际上这些行为中只有一部分行为会改变这些状态,那么对状态没有影响的行为就被界定为是没有副作用的方法,即查询方法[12]。查询方法只给出了方法间依赖关系的界定方法,没有给出存在依赖的那些方法的规格解决方案。

3 基于抽象状态的类的规格方法

在分析总结上文提到的各种方法之后,本文提出了基于抽象状态的类的规格方法。

抽象状态是对类数据概念上的约定,不是直接基于类数据实现的反映。如果类的某个方法 M 会因为其数据概念 X 上的差异而表现出行为差异:$X=st_i \Rightarrow M=f_i(params)$,$X=st_j \Rightarrow M=f_j(params)$,那么就认为类存在一个相应的抽象状态变量 $X=\{st_1,st_2,\cdots,st_n\}$,它影响了该方法的行为表现,需要参与类方法的规格。

在 Hoare[1]定义 $P\{C\}Q$ 时,约定 P 是定义在 Code 输入上的断言,Q 是定义在输出上的断言。在面向对象方法中,方法的可能输入包括方法参数和属性,可能输出包括输出值和属性,这要求在 P、Q 的定义中考虑类的属性实现:定义完全基于属性的断言以及综合了属性与输入/输出的断言。抽象状态变量实质上就是在 P、Q 定义上对类属性实现的一种概念约定。如果抽象状态变量 X 只约定了类的属性实现 CImp,那么称之为绝对状态变量 $AbsoluteStateX ::= \{st_1,st_2,\cdots,st_n \mid CImp \mapsto st_i, i=1,2,\cdots,n\}$。如果抽象状态变量 X 约定了类的属性实现 $CImp$ 与方法输入/输出值的综合,那么称之为相对状态变量 $RelativeStateX[p] ::= \{st_1,st_2,\cdots,st_n \mid (CImp,p) \mapsto st_i, i=1,2,\cdots,n, p=\{param_1,param_2,\cdots,param_m\}\}$。

一个类可以同时有多个抽象状态变量。作为一种设计规格特征,每个抽象状态变量及其状态都由设计师定义,由实现人员映射到类实现之中。为衔接抽象规格与具体实现,可以要求实现人员为每一个抽象状态变量 X 都必须实现一个查询方法 $getX()$ 或 $getX(p)$ 来反映类在任意时刻的状态特征,这样在代码实时运行时,将 P、Q 断言中的状态变量替换为查询方法,就能够实现断言的自动化检查。

为保证状态查询方法本身不会带来副作用,要求每个状态查询方法必须是[12]所定义的无副作用的查询方法。

如果类的某个方法规格定义中需要使用抽象状态变量,就说明该方法除了影响输入/输出之外,还可能产生改变数据实现的副作用。如果类的两个或多个方法都在规格定义时使用了相同的抽象状态变量,就说明它们之间可能存在着数据实现上的相互依赖关系。

综合所有方法的定义,可以为类的任一个抽象状态变量 X 建立状态机:

$STM_x=\{States, Transitions, Initial, Final\}$, $States=\{st_1,st_2,\cdots,st_n\}$, $Initial=\{st_i \mid st_i \text{ defined on } Q \text{ of Constructors}\}$, $Final=\{st_j \mid st_j \text{ defined on } Q \text{ of Destructors}\}$。检查该状态机的正确性就可以发现类在该抽象状态变量所关联的方法间规格定义存在的缺陷。

因为类自身的复杂性是受控的,所以可以设想一个类的抽象状态变量及其状态的数量都是有限的,不会造成状态空间爆炸的问题。

4 基于抽象状态的类的规格语法

类的规格可以定义为由抽象状态变量、方法组成,表示如下:

$C ::= \{\{X\}\{interface\}\}$

$X ::= AbsoluteStateX \parallel RelativeStateX[p]$

$AbsoluteStateX ::= \{st_1,st_2,\cdots,st_n \mid CImp \mapsto st_i, i=1,2,\cdots,n\}$

$RelativeStateX[p] ::= \{st_1,st_2,\cdots,st_n \mid (CImp,p) \mapsto st_i, i=1,2,\cdots,n\}$

$p=\{param_1,param_2,\cdots,param_m\}$

$CImp$ 是类的实现

$Interface ::= \{Pre\}\text{Method}\{Post\}$

$Pre ::= \text{Axioms } \underline{\text{define on}} \{\{X\} \cup In\ Params\}$

$In\ Params$ 是方法的输入值

$Post ::= \text{Axioms } \underline{\text{define on}} \{\{X\} \cup Out\ Params\}$

$Out\ Params$ 是方法的输出值。

方法由前置条件、方法主体和后置条件组成。前置条件是可能包含绝对状态或相对状态的断言，后置条件和前置条件一样，也是可能包含绝对状态或相对状态的断言。

如果一个 $Interface$ 的 Pre 与 $Post$ 定义都不需要引用任何 X，那么该 $Interface$ 属于 $Indicator$ 类型。

对每一个 X，都必须存在一个 $Indicator$ 类型的方法：

$$\forall X, \exists X\text{-}Indicator \subset Indicator ::= (CImp \mapsto st_i \mid st_i \in X)$$

在程序运行时，将实际的输入/输出值替代断言中的相对状态变量的 p，就可以使用状态查询方法替代断言中的抽象状态变量，实时进行断言的自动化检查。

5 基于方法的规格及验证示例

在 Eclipse 上开发扩展的插件工具，实现了基于本方法的类规格定义、提取与解析工作。

插件工具的主要思路如下：

1)在程序的备注中使用特定标记定义方法的前置与后置条件。

2)读取目标程序，整理分析规格语言和程序源文件，建立规格模型。

3)将规格生成以目标语言为格式的检测语句。

4)使用插入法，将检测语句添加到适当的执行路径中，生成编译文件。

5)当程序运行时，根据其执行时的情况，记录规格检查结果。

该工具已经开发完成，但因篇幅有限，这里就不详细描述工具的实现细节　下面是使用该工具的一个表现示例

Storage. java

```
public class Storage{
    enum state{empty,notfull,full}
    enum storelock{open,close}
    enum getlock{open,close}
    List<Object> o=new ArrayList<Object>(10);
    private state indicatorState(){
        if(o. size()==0){
            return state. empty;
        }else if(o. size()==10){
          return state. full;
        }else{
          return state. notfull;
        }
    }
private storelock indicatorStorelock(){
    if(o. size()==10){
        return storelock. close;
    else{
        storelock. open;
    }
}
    private getlock indicatorGetlock(){
        if(o. size==0){
        return getlock. close;
    }else{
        return getlock. open;
    }
}
/*@pre1:indicatorState()  !=state. full;
@pre2:indicatorStorelock()==storelock. open;
@post1:indicatorState()  !=state. empty */
public void promulgator(Object x){
    o. add(x);
}
/*@pre1:indicatorState()  !=state. empty;
@pre2:indicatorGetlock()==getlock. open;
@post1:indicatorState()  !=state. full*/
public Object collector(){
    Object x=o. remove(o. size() - 1);
    return x;
  }
}
```

图 1　eclipse 插件使用

如图 1 是 eclipse 规格检查扩展插件使用时所需要的代码格式。

首先定义 Storage 类，在 Storage 中抽象状态有 3 个状态，分别为空状态、有数据但不满状态及满状态，storelock 状态代表存储锁，getlock 状态代表取出锁，分别定义取出状态的方法；在 Storage 中定义取出和存储的方法，方法包括前置条件和后置条件。

```
public class Demo{
  public static void main(String[] args){
    Storage s=new Storage();
    s. collector();//(1)
    Object x;
    s. promulgator(x);//(2)
    s. promulgator(x);//(3)
  }
}
```

在上述方法(1)执行时，因为初始化时 Storage 的 state 状态为空，所以不满足前置条件，此方法无法执行。当 Storage 中差一个就是满的状态且连续执行方法(2)和(3)时，方法(3)的前置条件不满足，方法(3)无法执行。

结束语　在编写类的规格时，我们暂时没有考虑类中的不变量，因为不变量在类中是很难追踪的，需要跟踪类在内存中的变化，即使实现了也会花费较多的时间。后续工作中将尝试引入不变量断言。

在为类定义了基于抽象状态的方法规格之后，可以在后续工作中将它扩展为类的状态图，并据此检查类在规格定义上的错误。

本文的方法是适用于类粒度的，如果将粒度进一步提升为模块、部件甚至需求元素，方法就会出现局限性，我们在后续工作中也将沿着状态分解与合并的思路探讨更大粒度的程序规格定义。

参 考 文 献

[1] Hoare C A R. An Axiomatic Basis for Computer Programming [J]. Communications of the ACM, 1959, 12(10): 576-580

[2] Guttag J V, Horowitz E, Musser D R. Abstract Data Types and

Software Validation[J]. Communication of the ACM, 1978:21(1):1048-1064

[3] Goguen J A, Thatcher J W, Wagner E G, et al. Abstract data-types as initial algebras and correctness of data representations [C]// Proc. Conf. on Comptr. Graphics, Pattern Recognition and Data Structure. 1975

[4] Zilles S N. Abstract specifications for data types[R]. IBM Res. Lab., San Jose, Calif., 1975

[5] Meyer B. Applying "Design by Contract"[J]. Computer, 1992, 25(10):40-51

[6] Findler R B, Felleisen M. Behavioral Interface Contracts for Java [OL]. http://www.researchtate.net/publication/2245179—Behavioral—Interface—Contracts—for—Java

[7] Findle R B, Felleisen M. Contracts for Higher-Order Functions [J]. ACM Sigplan. Notices, 2002, 37(9):48-59

[8] Cheon Y, Leavens G T, Sitaraman M, et al. Model Variables: Cleanly Supporting Abstraction in Design By Contract[J]. Software-practice & Experience, 2003, 35(6):583-599

[9] Hoffman D, Strooper P. State Abstraction and Modular Software Development[M]// SIGSOFT 95. Washington, DC, USA, 1995

[10] Hatcliff J, Leavens G T. Behavioral Interface Specification Language[J]. ACM Computing Surverys, 2012, 44(3):1-58

[11] Jacobs B, Piessens F. Inspector Methods for State Abstraction: Soundness Proof[J]. Journal of Object Technology, 2007, 6(5): 55-75

[12] Dallmeier V, Wasylkowski A, Bettenburg N. Identifying Inspectors to Mine Models of Object Behavior[C]// ICFEM. 2004

[13] Jacobs B, Piessens F. Inspector Methods for State Abstraction: Soundness Proof[R]. CW Reports, 2007

[14] Grunwald D, Gladisch C. Generating JML Specifications from Alloy Expressions[M]// Hardware and Software: Verification and Testing. 2014

[15] Agostinho S, Moreira A. Contracts for Aspect-Oriented Design [C]// SPLAT 2008. ACM, 2008

[16] Kumar A, Bandyopadhyay. Modeling of State Transition Rules and its Application[J]. ACM SIGSOFT Software Engineering Notes, 2010, 35(2):1-7

[17] Polikarpova N, Furia C A. What Good Are StrongSpecifications? [C]// ICSE 2013. San Francisco, CA, USA, 2013

[18] Polikarpova N, Furia C A, Meyer B. Specifying reusable components[C]// VSTTE. LNCS, vol. 6217, 2010:127-141

[19] Wei Y, Furia C A, Kazmi N, et al. Inferring better contracts[C]// ICSE. 2011:191-200

[20] Wei Y, Roth H, Furia C A, et al. Stateful testing: Finding more errors in codeand contracts[C]// ASE. 2011:440-443

(上接第 453 页)

进,即以一定概率选择函数集和终止符集中的元素来组成头部。具体地,若依概率 P 挑选函数集中的元素,则依概率 $(1-P)$ 挑选终止符集中的元素。实验表明,当挑选函数集中元素的概率略大于挑选终止符集中元素的概率时,群体的整体质量较好。本文中 p 取 0.65。

采用的适应度函数公式如下:

$$fitness=w_1 * \frac{N_i}{N}+w_2 * \frac{1}{KEL_i}$$

其中,w_1、w_2、KEL_i 等具体含义见第 2 节。

每个算法运行 25 次,最后得到的结果为 25 次运行结果的平均值。

3.2 实验结果

因 Iris 数据集是 3 类别分类问题,故对数据集进行相应处理,得到 Iris-setosa、Iris-versicolor、Iris-virginica 这 3 个数据集。通过反复实验,对 w_1 和 w_2 分别进行不同的组合,得到的实验结果如表 1 所列。

表 1 改进 GEP 分类结果

数据集		$w_1=1.0$ $w_2=0.0$	$w_1=0.9$ $w_2=0.1$	$w_1=0.8$ $w_2=0.2$	$w_1=0.7$ $w_2=0.3$
Iris-setosa	分类精度(%)	100	100	100	100
	KEL 平均值	26	7	6	6
Iris-versicolor	分类精度(%)	97.4	97.3	96.7	95.8
	KEL 平均值	28	22	18	14
Iris-virginica	分类精度(%)	97.6	97.4	97.3	96.7
	KEL 平均值	20	16	12	10

从表 1 可以看出,随着 w_1 逐渐变小,各数据集的分类精度有所下降,但染色体的有效长度 KEL 相应也变小,即分类精度下降而规则的易理解性上升。当 $w_1=1.0$ 时,适应度函数等于分类规则的准确度,故所产生的规则分类精度很高,但由于未考虑染色体的有效长度,导致所产生的规则的 KEL 平均值偏高;当 $w_1=0.9$ 时,适应度函数综合考虑分类准确度和 KEL 长度,但通过给予分类准确度更高的权重来实现准确度优先考虑而 KEL 长度次之考虑,所产生的规则准确度有一定程度的下降,KEL 平均值有明显的下降;依此类推,w_1 取 0.8、0.7 时,分类准确度逐渐降低,而 KEL 平均值逐渐减小即复杂度降低。之所以没有考虑 $w_1=0.1$、0.2、0.3、0.4 等情况,是由于对于分类规则挖掘而言,应该更强调分类准确度而非规则的简单性,如果 w_1 取以上值,会导致规则简短但分类精度达不到要求,故不予考虑。

但通过分析可以看出,w_1 取值 1.0 和 0.9 的情况下,分类准确度的下降微乎其微,而 KEL 平均值的下降却显而易见,对于表中 3 个数据集,如果分类准确度只考虑 2 位数,则 w_1 取值 1.0 和 0.9 是没有区别的。故在对分类准确度的精度要求不是特别高的情况下,可以通过设置 w_1 的值来兼顾准确度和规则简单性。

综上所述,可以看出,在进行分类规则挖掘时,采用本文所提出的适应度函数,可以通过对 w_1 和 w_2 的值进行相应设置,达到综合考虑分类准确度和易于理解性的目的。

参 考 文 献

[1] Peilikan M. A Simple Implementation of Bayesian Optimization Algorithm[OL]. http://core.ac.uk/display/22715852

[2] 付红伟,刘园园,陈金鑫,等. 改进的 GEP 算法在演化建模中的应用[J]. 重庆工学院学报,2009(3):167-170

[3] 付红伟,毛亚梅,罗炜,等. 混合决策树/遗传算法的数据挖掘[J]. 软件导刊,2009,4(1):157-159

[4] 姜大志,吴志健,康立山,等. 基因表达式程序设计的 GRCM 方法[J]. 系统仿真学报,2006,6(18):1466-1468