

一种基于分支条件的对象状态机自动提取方法

吴 浩, 彭 鑫, 赵文耘

(复旦大学计算机科学与工程系 软件工程实验室, 上海 200433)

E-mail: wuhao@fudan.edu.cn, pengxin@fudan.edu.cn, wyzhao@fudan.edu.cn

摘 要: 对象行为协议对于辅助其他开发者理解并正确使用对象所提供的外部行为具有十分重要的意义。然而相关文档却常常缺失或存在不一致, 需要通过逆向分析的方法进行恢复。针对这一问题, 本文提出了一种基于驱动执行和动态分析的对象状态机 (Object State Machine, OSM) 提取方法。该方法从源代码中提取包含类属性的条件表达式, 以其在运行时刻的取值情况及程序异常信息作为状态标识, 并通过驱动执行的方式获取运行时的状态转换信息, 然后分析运行时信息逐步构造对象状态机。该方法已实现为相应的原型系统, 初步实验结果表明通过该方法可以高效、准确地恢复对象行为协议。

关键词: 对象状态机; 行为协议; 条件提取; 状态分析; 驱动式调用

An Automatic Extraction Method of Object State Machine Base on Condition Expressions

WU Hao, PENG Xin, ZHAO Wen-yun

(Department of Computer Science and Engineering, Fudan University, Shanghai 200433, China)

Abstract: Object behavior protocols are of great significance in guiding developers to understand and properly use the external functions provided by the object. Nevertheless, the related documents are often missed or inconsistent. Therefore, reverse analysis is used to recover the protocols. To achieve this goal, this paper presents a new method to extract Object State Machine (OSM) based on intentional method invoking and dynamic analysis. This method firstly extracts condition expressions containing class attributes from source code and defines states to be the sets of values of expressions at run-time as well as the exceptions; then gets states transition information by intentional method invoking; and finally uses the sates and the states transition information to construct object state machine. The corresponding prototype system has been realized and the preliminary experimental results show that this method can efficiently and accurately recovers behavior protocols.

Key words: Object State Machine, Object behavior protocols, Conditions extraction, State analysis, intentional method invoking

1 引言

在面向对象的软件系统中, 对象作为状态和行为的封装体存在。对于有状态的对象, 其对外提供的多个公共方法在使用上并不是完全独立的, 而是具有一定的约束关系, 这种关系称为对象行为协议。接口行为协议描述了类接口在调用顺序上的约束关系, 对接口的正确使用有着重要意义。接口行为协议逆向恢复的相关研究 (如[1][2][3][4][5]) 主要分为静态分析和动态分析两种。静态分析 (如数据依赖分析方法[3]) 通过对源代码中的数据流和控制流进行分析, 提取方法间的依赖关系并根据依赖关系、断言和异常处理等静

态信息分析合法的调用序列。动态方法 (如 K-TAILS 方法[7]) 则在源程序代码插装的基础上, 通过系统的动态运行获取相应的执行轨迹并通过分析和抽象程序运行轨迹得到接口的行为协议。接口行为协议的表示则主要有正则表达式和对象状态图两种方法。相对于正则表达式而言, 对象状态图更加直观、容易理解, 能明确描述各个状态下对象可能的行为。因此, 我们采用对象状态图作为行为协议的描述手段。

本文中, 我们基于静态分析与动态分析相结合的方式, 采用对象状态机来描述模块接口的行为协议, 提出了一个自动提取对象状态机的新方法。我们采用了一种新的方法表示抽象状态, 从判断条件中提取用于标识类状态的条件表达式, 并利用这些表达式运行时的值来表示对象状态, 减少了

状态提取和状态比较的复杂性。同时,只用修改类属性的方法来表示状态变迁避免了状态无关方法的影响。另外,采用驱动式的方法,动态的提取信息,根据运行时的对象状态,判断可能导致新状态的变迁,并进行相应的方法调用。最后,利用这些对象的状态信息和变迁信息,生成状态机。得到的状态机有助于理解对象的业务功能,正确的使用对象和对程序进行自动验证。

接下来的第二部分将对对象行为协议逆向恢复的相关工作进行介绍和分析。第三部分介绍本文方法的主要过程以及所使用的堆栈类代码示例,其中的一些关键步骤将在第四部分进行讨论。第五部分对所实现的工具原型进行了介绍并对相关问题进行了讨论。最后,第六部分总结全文并对下一步的工作进行了展望。

2 相关工作

我们的想法主要启发于 Hai Yuan 等[1]利用分支覆盖信息自动提取抽象对象状态机的工作。他们利用所有方法执行时所经过的分支信息的集合来表示状态,方法调用表示变迁,通过自动生成的测试用例来获得动态信息,提取状态机。我们的方法与[1]的根本区别在于状态表示的不同。尽管都有对条件表达式的分析,我们的方法从分支条件中提取用于标识类状态的条件表达式,并利用这些表达式运行时的值的集合来表示对象状态。同时,我们只用修改类属性的方法来表示状态变迁,并采用驱动式的方法自动进行方法调用。相比而言,我们的方法更容易进行状态提取和状态合并,同时获得类似精确的状态机表示。

Tao Xie 等[2]还利用观察者(接口中含有返回值的方法)抽象的方法来提取对象状态机。这种方法使用当前对象的所有观察者方法的返回值来表示状态。由于使用具体的值的集合来表示状态,得到的对象状态机具有很高的复杂性,不利于理解。在我们的方法中,使用提取用于标识类状态的条件表达式,提高了抽象的程度,减少了产生的状态。

Whaley[3]等通过系统执行的轨迹来获取 java 组件的接口序列信息。提取出的接口信息被表示为多个有限自动机的形式,每个自动机中包含读写同一个类属性的所有方法的信息。在每个自动机上,Whaley 把所有调用同一个修改类属性的方法后得到的具体状态映射为一个抽象状态。Whaley 方法依赖于测试用例的选择,过少的测试用例不能包括所有的方法调用序列,而利用自动生成的测试用例,又将得到由修改同一类属性的方法构成的完全图。我们的方法中,由于调用序列采用驱动生成的方法,从而避免了对测试用例选取得依赖,可以自动的得到对象状态机信息。

Ernst[4]等通过测试用例执行,利用 Daikon 发现程序中的不变点。不变点以原子的规格说明的形式出现。不变点用于描述方法的前置后置条件,类属性、参数和方法返回值之间的关系。Ernst 的方法更注重对数据流的分析,而我们的方法侧重于对类行为协议的提取,调用序列约束的获得。Henkel 和 Diwan[5]通过测试用例执行,从 java 类中提取代数规约。他们获得低层次的两个函数之间的关系。我们则以对象状态机的形式,展示出全局的多个函数之间的相互调用关系。

我们采用了一种新的方法表示抽象状态,即从程序的判断条件中提取用于标识类状态的条件表达式,并利用这些表达式运行时的值来表示对象状态,减少了状态提取和状态比较的复杂性。同时,只用修改类属性的方法来表示状态变迁避免了状态无关方法的影响。另外,采用驱动式的方法,动态的提取状态信息,生成接口方法调用序列消除了测试用例选择不足对结果的影响。

3 OSM 提取方法概述

有关对象状态和状态图, UML 和 OMT 有着类似的定义。

在 UML 中,对象状态和状态图定义如下[8]:状态表示对象在其生命周期中的一种状况,状态图用来描述一个特定对象的所有可能状态及其引起状态转移的事件。大多数面向对象技术都用状态图表示单个对象在其生命周期中的行为。

OMT 的定义为[9]:状态是对象的一组特殊的属性及其相互关联关系的组合。基于这些影响对象行为的属性,一系列相关的值组成了对象状态;状态图是用节点表示状态,带有事件名称的有向边表示变迁的模型。

在面向对象的状态图中,变迁通常是由于方法调用引起的。对于类中的方法,我们使用[6]中对类方法的分类,即一个类由构造方法,析构方法,修改方法和访问方法所组成。其中,构造方法用来初始化该类的一个新的对象;析构方法用来在对对象进行垃圾收集前释放系统资源;修改方法是类中那些会修改类属性值的方法;访问方法则是仅对类变量进行读操作的方法。

一般情况下,出现在类的判断条件中的属性或方法,常常与对象状态相关联,决定方法的执行情况。在没有条件判断或断言的情况下,错误的函数调用往往在运行时会有异常产生。因此,我们从判断条件中提取出仅与类属性相关的条件表达式,利用这些表达式运行时的值和系统异常来描述对象状态。这种表示法提取了可能的类状态的条件表达式,有着较好的抽象结果。变迁方面,我们区分构造方法,析构方法,修改方法和访问方法,并只把构造方法和修改方法放入变迁集合。对于访问方法,我们把它作为一种特殊的事件,仅在每个对象状态上检查是否可进行相应的方法调用。在信息提取和代码插装之后,驱动式的进行方法调用,逐步构造和精化对象状态机。下面给出对象状态机(OSM)的定义以及本文的对象状态机抽取方法的基本步骤。

定义1. 类 c 的一个对象自动机模型(OSM) M 表示为如下六元组: $M = (Q, \Sigma, \Sigma', \delta, q_0, Q_e)$ 其中

- Q 是对象状态的集合,对象状态包括正常状态和异常状态两种情况。其中,正常状态由所有仅关联到类属性的条件表达式和为检查空引用附加的判断表达式在运行时的值的集合组成;异常状态,作为 Q 中的一种特殊状态,即程序运行时,非法方法调用导致的结果;
- Σ 是类的所有构造方法和修改方法的集合;
- Σ' 集合由所有类的访问方法组成,在某一状态下调用 Σ' 集合中的方法,只会产生以下两种执行结果:1) 不改变当前状态,2) 抛出异常;
- δ 表示状态之间变迁的函数, $\delta(q_i, m_i) = q_j$

表明在状态 q_1 调用方法 m_i , 将会导致 OSM 变迁到一个新状态 q_2 ;

- q_0 表示 OSM 的初始状态, 即在构造方法执行之前的状态; Q_E 表示OSM终结状态的结合, 由异常状态和执行结束的状态组成。

本文所提出的对象状态机抽取方法的基本步骤如图 1 所示:

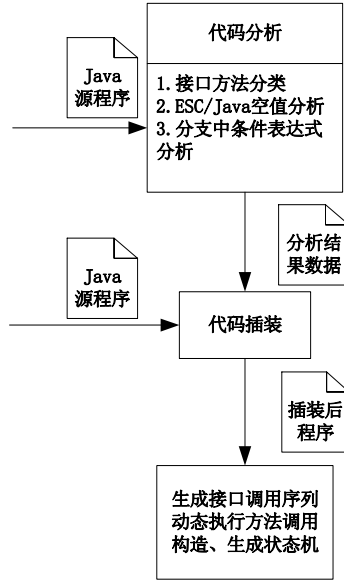


图 1: OSM 提取流程图

Fig 1:Flow Chart of OSM Extraction

OSM 提取方法的第一步是对 java 程序源代码(字节码)进行分析。在第一步的分析中主要完成三个工作: 接口方法分类, 可能的未判断空引用查找和从分支语句中抽取条件表达式。第二步中, 根据分析提取出的数据, 对源代码进行插装, 以获得运行时分支条件表达时运行时值的信息。最后, 根据插装后程序运行时的信息, 动态的决定对象下一步调用的方法, 生成状态机。

本文以下部分, 将结合 Mark Allen Weiss 的数据结构与算法分析中的堆栈程序[16] (图 2) 逐步介绍 OSM 的提取方法。

此程序同样使用在 Daikon 的发布版本中作为示例程序, 包含了堆栈的基本操作 (修改方法) push, pop, makeEmpty 和 topAndPop; 以及判断堆栈状态和获得栈顶元素的方法 (访问方法) isEmpty, isFull 和 top。push 操作首先检查栈是否已满, 若已满则抛出 Overflow 异常, 否则向栈内压入一个元素; pop 操作则先检查栈是否为空, 若为空便抛出 Underflow 异常, 否则从栈内弹出一个元素; topAndPop 操作类似于 pop 操作, 在完成 pop 操作的同时, 返回栈顶元素; makeEmpty 清空栈内的所有元素。方法 isEmpty, isFull 分别用于判断栈为空或满的情况。方法 top 返回栈顶元素。可用栈的构造方法初始化类的大小。

4 关键步骤

4.1 从代码中提取条件表达式

上面已经提到在面向对象的程序设计中, 一个类的方法一般分为构造方法, 析构方法, 修改方法和访问方法四类。

```

public class StackAr
{
    static final int DEFAULT_CAPACITY = 10;
    public StackAr()
    {
        this( DEFAULT_CAPACITY );
    }
    public StackAr( int capacity )
    {
        theArray = new Object[ capacity ];
        topOfStack = -1;
    }
    public boolean isEmpty()
    {
        if(topOfStack == -1)
            return true;
        return false;
    }
    public boolean isFull()
    {
        if(topOfStack == theArray.length - 1)
            return true;
        return false;
    }
    public void makeEmpty()
    {
        java.util.Arrays.fill(theArray, 0, topOfStack + 1, null);
        topOfStack = -1;
    }
    public Object top()
    {
        if( isEmpty() )
            return null;
        return theArray[ topOfStack ];
    }
    public void pop() throws Underflow
    {
        if( isEmpty() )
            throw new Underflow();
        theArray[ topOfStack-- ] = null;
    }
    public void push( Object x ) throws Overflow
    {
        if( isFull() )
            throw new Overflow();
        theArray[ ++topOfStack ] = x;
    }
    public Object topAndPop()
    {
        if( isEmpty() )
            return null;
        Object topltem = top();
        theArray[ topOfStack-- ] = null;
        return topltem;
    }

    private Object [ ] theArray;
    private int topOfStack;
}
  
```

图 2: 堆栈程序示例

Fig 2: Stack Example

由于析构方法用于在对对象进行垃圾收集前释放系统资源, 由系统完成, 不会影响到对象的状态变化, 我们这里并不对析构方法进行讨论。通过对程序中赋值语句的分析, 不难区分以上几种类型的方法。访问方法只会读取类的相关属性值, 所以不会导致类的状态发生改变。但在不正确的状态下,

调用访问方法则会导致异常的抛出。因此, 在我们的 OSM 模型中把访问方法作为一种特殊的输入符号。我们只会在一状态下, 去检查可以调用那些访问方法, 而不会把它们做为变迁函数的参数去处理。例如, 图 2 中的 top 方法, 仅仅返回栈顶元素不会改变对象的状态, 但在空栈时调用则可能返回非预期结果。把访问方法同其他改变类属性值的方法隔离开来, 降低了状态机中变迁的数量, 使其易于理解和构造。在对类的方法进行分类之后, 我们进行提取相关条件表达式的工作。对象的状态主要是由一组特殊的属性及其相互关联所决定的, 为此我们所要提取的条件表达式中暂时不包括依赖于局部变量的情况。

定义 2. OSM 的状态表达式集(SCE, State-Represented Conditional Expressions), 定义为类中出现的条件表达式(CE)的集合, 其中 CE 由 *field*, *const*, *m()*, *ro* 所成, *field* 表示类的属性; *const* 表示常量, 包括 true, false 和 null 值; *m()* 表示类的方法, 且此方法不使用任何局部变量作为参数; *ro* 表示关系运算符。例如, CE: *topOfStack == -1* 中, *topOfStack* 为类属性 *field*, *==* 是关系运算符 *ro*, *-1* 则是常量 *const*。

```

public class Assignment
{
    String var;
    public void setVar(String s) {
        var = s;
    }
    public String checkVar() {
        if(var.equals("var"))
            var += "v";
        return var;
    }
}

```

Java 源代码

```

Assignment ...
Assignment: setVar(java.lang.String) ...
[0.078 s 4895568 bytes] passed
Assignment: checkVar() ...

Assignment.java:10: Warning:
Possible null dereference (Null)
    if(var.equals("var"))
        ^
[0.157 s 4787320 bytes] failed
Assignment: Assignment() ...
[0.031 s 4933744 bytes] passed

```

ESC/Java 分析结果

图 3: ESC/Java 示例

Fig 3: Example of Using ESC/Java

为了检测由于类属性值依赖关系导致的调用顺序约束, 我们首先向 SCE 中加入一些空引用的判断的条件表达式。ESC/Java[10]作为一种 java 程序的检测器, 通过静态分析的方式发现可能的程序执行异常。ESC/Java 能够提示多种可能的出错信息并发出警告, 如除数为 0, 下标值过大, 下标值为负以及空引用等。这里, 我们只利用其中的空引用警告。如图 3 所示, 对于类 Assignment, 调用方法 checkVar 时, 类属性 var 的值可能为 Null, ESC/Java 运行得出如右图所示的结果。对于每条警告信息, 如果 var 不是类属性或

警告信息不是可能的空引用, 则忽略这条信息。否则, 若 *field==Null* 不在 SCE 中, 即添加 *field==Null* 至 SCE。对于图 3 中的示例, 我们会把 *var==Null* 表达式加入到对应的 SCE 中。利用 ESC/Java 对图 2 所示的堆栈程序进行分析发现: 类属性 *theArray* 在 push, pop, top 等方法中可能为空的警告, 我们把 *theArray==Null* 加入 SCE 中。尽管 StackAr 类在初始化之后, *theArray* 都不等于 Null, 但这类信息在类似于 Assignment 的程序中, 有着重要的意义。例如, 在 Assignment 中, 向 SCE 中加入 *var==Null*, 提取出的 OSM 即可发现, 在可调用 checkVar 方法的合法状态上, (*var==Null*) 值为 false。

本部分的第三个工作为从分支语句中抽取条件表达式。通过对源代码的扫描, 我们不难提取出所有不依赖于局部变量的条件表达式。在我们的原型中, 首先利用 JTB (Java Tree Builder) 构建出抽象语法树 (AST)。然后, 对 AST 进行深度优先遍历。在遍历的过程中, 通过 Visitor 模式, 抽取条件表达式, 同时排除那些依赖于局部变量的条件表达式。

4.2 驱动式的接口调用

在驱动式的进行接口调用, 生成状态机之前, 我们对代码进行插装, 以获得动态运行时的信息。关于代码插装, 现在有更多的工具可以完成 (BCEL, JavaAssist 等), 这里就不详细叙述。

图 4 展示了我们驱动式的调用接口方法, 生成状态的算法。目前, 我们手动设置了一些变量值作为输入, 今后考虑使用 JTest 或 JCrasher 等自动的测试工具生成测试数据。首先, 解释一下算法中进行状态比较的方法: *absEquals* 和 *subOf*。

定义 3. 对于对象状态 *s1*, *s2*, 如果 *s1.absEquals(s2)*, 那么对于 SCE 中的每个条件表达式 *ce* 的运行时的值, 都存在 *s1.ce.value = s2.ce.value*。如果 *s1.absEquals(s2)*, 我们说 *s1* 和 *s2* 处于同一个抽象状态。

为了表示同一抽象状态下的具体状态变化, 我们记录每次方法调用后改变的状态的值。在我们堆栈的例子中, 在非栈满的状态下, 都有 (*topOfStack == theArray.length - 1*). *vaule = false*。但每次执行 push 操作后, *topOfStack* 的值都会加 1, 我们在相应的状态中, 利用四元组 (方法名, 变量名, 前值, 后值) 记录每一次类变量值的变化。其中, 前值和后值分别表示方法调用前后对于变量的值。若某一状态 *s* 下, 包含 (push, *topOfStack*, 5, 6), (push, *topOfStack*, 6, 7), 而另一状态 *s'* 仅包含 (push, *topOfStack*, 6, 7); 我们认为 *s' . subOf(s)*。当 *s' . absEquals(s)* 且 (*not s' . subOf(s)*) 时, *s'* 中包含 *s* 中所没有的类变量值改变, 此时我们取 *s* 和 *s'* 中所有类变量值改变的并集得到新的状态 *news*。此时, *s.absEquals(news)* And *s' . absEquals(news)* And *s.subOf(news)* And *s' . subOf(news)*。通过进行 *subOf* 的比较, 可以判断程序执行是否到达已存在的具体状态。在图 2 的例子中, 由于关系运算符的两端都是数值型变量 (或常量), 我们可以判断方法调用后类变量值变化的趋势, 来决定是否继续驱动的调用相关方法。而对于非数值型的关系运算, 我们目前设定一个阈值 *t*, 若方法调用次数超过 *t* 且

未返回已有具体状态，则停止对此方法的进一步调用。

```

1.  set states, set proceededStates;
2.  intiState si;
3.  states.add(si, si.construct1... si.constructN);
4.  proceededStates.add(si);
   //在每个还没有处理的状态上，调用修改方法
5.  for(every s in (states - proceededStates)) {
6.      for(every mi in  $\Sigma$  and not(mi, constructor)) {
7.          tmpstate = clone(s); //临时变量tmpstate保存原状态
           //得到新状态newstate
8.          newstate = tmpstate.invoke(mi);
9.          count = 0; //记录循环调用的次数
           //新状态与原状态处于同一个抽象状态的处理，t为阈值
10.         while(newstate.absEquals(tmpstate) && count < t) {
           //若到达一个已处理过的情况，跳出;
           if(newstate.subOf(tmpstate))
12.             break;
           //否则合并s和newstate的信息，并在新状态上继续
           //调用mi，直至返回已有状态或到达新状态
13.         else {
           // newstate添加s中的具体状态信息，并保存为
           //tmpstate
14.             tmpstate = merge(newstate, s);
15.             states.replace(s, tmpstate);
16.             newstate = tmpstate.invoke(mi);
17.         }
18.         count++;
19.     }
20.     if(!(newstate.subOf( any state in states)))
21.         states.add(newstate);
22. }
23. proceededStates.add(s);
24. }

```

图 4：状态机动态提取算法

Fig 4: Algorithm of Dynamic Extraction of OSM

通过以上方法，提取出堆栈示例的状态图(图 5)与期望结果基本一致。

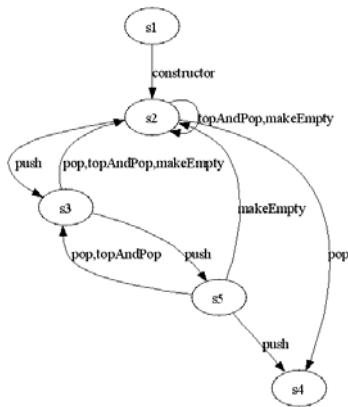


图 5：堆栈状态图

Fig 5: State Chart of the Stack

其中，s1 为初始状态，s4 为异常状态，s2, s3, s5 对应的条件表达式运行时值如表 1 所示：

条件表达式 状态	isFull (topOfStack == theArray.length - 1)	isEmpty (topOfStack == -1)
S2	TRUE	FALSE
S3	FALSE	FALSE
S5	FALSE	TRUE

表 1：堆栈状态与条件表达式对应关系表

Table 1: Mapping of States and CEs of the Stack

5 原型实现

为了实现基于判断条件的对象状态机自动提取方法，我们开发了一个源代码分析工具的原型。它的实现基于 Java Tree Builder (JTB) [13] 框架。JTB 是一个开源的语法树生成框架，通过读入 JavaCC [14] (Java 开发的语法分析生成器) 生成的语法文件，自动生成语法树结构。它可以通过访问者模式对生成的语法树结构单元进行操作。我们通过开发一个深度优先的抽象语法树访问者类来遍历目标源代码的语法树，提取源代码中的信息，本文中主要是利用抽象语法树中的信息来区分方法的种类并提取出出现在 “If Statement”，“For Statement”，“Switch Statement” 和 “While Statement” 条件表达式。我们把这些用于下一步状态机提取的信息存放在相关文件中。对于 ESC/Java 的分析结果文件，由于其结构比较简单，利用对 AST 遍历时已提取的类属性信息，我们通过程序读取文件，处理空引用，并生成条件表达式，也加入到对应文件中。

在对代码进行插装的过程中，我们采用 JBOSS AOP 项目的 JavaAssist [17] 工具。其主要用于对二进制代码的插装。尽管 BCEL 库提供了更加强大的二进制代码操作方法，但 JavaAssist 已经足以帮助我们获得动态运行时的信息。

对于驱动式的状态生成，我们采用序列化的方法来保存每一次方法调用后的接口状态。利用 java 语言的 Serialization 机制，无需考虑对象的内部结构。在实现过程中，通过 ObjectOutputStream 的 writeObject 方法，把对象写入内存的区域中；然后，使用 ObjectInputStream 的 readObject 方法，获得相应对象的拷贝。同时，由于 Serializable 接口没有方法或字段，仅用于标识可序列化的语义，所以我们进行代码插装时，在类声明中加入实现序列化接口的声明，从而保证保存每个对象的正确状态。

最后，利用贝尔实验室的 GraphViz [15] 图形显示工具，绘出状态图。我们的程序生成 GraphViz 可识别格式的文件。这样，通过 GraphViz 提供的 Grappa 或者 Dot 工具，使用者即可编辑状态图或生产 bmp 等格式的图片。

对于文中提到的 StackAr 堆栈程序，在默认构造函数下，程序运行结果图 6 所示：

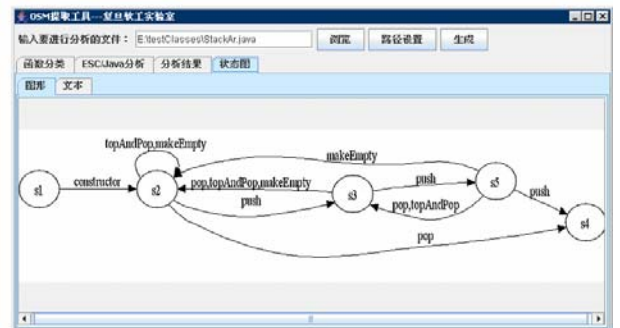


图 6：程序运行结果图

Fig 6: Program Demo

6 实验及讨论

为表明方法的效率和准确性，我们以数据结构中常用的堆栈，基于数组的队列，循环队列，二叉树和哈希表进行了

初步的实验。利用文中的方法对状态进行抽象表示,假设提取出的可能的状态相关条件表达式的数量位 n ,则实际产生的状态数量小于等于 2^n 。通过堆栈的例子(表1)可以看出,除去初始状态和异常状态外,产生的状态数量为3。(这里,条件表达式 $isfull()$ 与 $topOfStack == theArray.length - 1$ 是同值的,可作为一个处理)。对队列,环形队列,二叉树和哈希表的实验得到的状态数量分别为4,3,2,3。实验比较发现数组实现普通队列相较循环队列,存在同时为队空和队满的状态($tail == Array.length - 1$ and $head == tail$)。实验结果表明,在程序运行的过程中产生的状态数量在一个较小的数量级,因此有着好的处理效率。对于堆栈、队列、循环队列和二叉树得到的状态图和手工建模得到的状态图,基本一致。对于哈希表,产生的状态图不能区分是否存在对应元素时结果的不同,但仍然反应了主要的状态信息,可以指导调用者的正确使用。

[1]中以堆栈为例进行分析,对于堆栈的同一个状态,[1]中由于针对具体的运行时值进行分析需要5至10个分支信息来表示,而本文的方法精简到2个条件表达式的值。[2]以链表来说明其方法的运行,[2]中通过对链表的11个不同状态和161个变迁进行抽象,来提取它的观察者抽象表示;本文的方法仅产生出两个抽象的状态,并且降低了状态合并的复杂度。[7]的状态提取方法更多的是基于程序执行轨迹的K-Tails方法,其复杂度随方法个数的增加而指数级增长,本文的方法则更多依赖于类本身所包含的状态的数量。

我们采用了一种新的方法表示抽象状态,从判断条件中提取用于标识类状态的条件表达式,并利用这些表达式运行时的值来表示对象状态,减少了状态提取和状态比较的复杂性。同时,只用修改类属性的方法来表示状态变迁避免了状态无关方法的影响。另外,采用驱动式的方法,动态的提取状态信息,生成接口方法调用序列消除了测试用例选择对结果的影响。

由于采用条件表达式来表示类状态,因此可能会丢失一些类状态的表示信息。在构造和精化状态机的过程中,Daikon等工具所提供信息有助于更好描述状态变化的方向,降低状态生成算法的时间复杂性。在我们的当前原型系统中,动态生成状态机的部分的输入变量值由手工完成,今后我们将考虑利用自动测试工具完成。

在下一步的工作中,我们将尝试加入其它可能的类状态表示方法及动态运行信息来完善状态机生成过程。

7 总结

本文提出了一种基于判断条件的对象状态机(OSM)提取方法,并实现了其原型系统。我们的方法通过进行静态信息提取和代码插装,利用驱动式的方法,动态的提取状态信息,生成接口方法调用序列,逐步构造和精化对象状态机。自动提取的状态机模型可用于补充程序文档,进行程序验证和指导测试用例生成等方面。我们的方法中,通过条件表达式表示类状态,提高了抽象的程度,减少了产生的状态;同时,采用动态驱动式的状态生成方法,使产生的状态更加完备。初步实验结果表明,上述方法可以较为准确的获得对象状态机模型,对于系统的理解有着重要的帮助。

References:

- [1]. Hai Yuan and Tao Xie. Automatic extraction of abstract-object-state machines based on branch coverage. In *1st International Workshop on Reverse Engineering To Requirements (RETR 2005)*, (Pittsburgh, PA, USA), Nov. 2005, pp. 5-11.
- [2]. T. Xie and D. Notkin. Automatic extraction of object oriented observer abstractions from unit-test executions. In *Proc. 6th International Conference on Formal Engineering Methods*, Nov. 2004.
- [3]. J. Whaley, M. C. Martin, and M. S. Lam. Automatic extraction of object-oriented component interfaces. In *Proc. International Symposium on Software Testing and Analysis*, pages 218-228, 2002.
- [4]. M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Trans. Softw. Eng.*, 27(2):99-123, 2001.
- [5]. J. Henkel and A. Diwan. Discovering algebraic specifications from Java classes. In *Proc. the 17th European Conference on Object-Oriented Programming*, pages 431-456, 2003.
- [6]. Rainer Koschke and Yan Zhang. Component Recovery, Protocol Recovery and Validation. In: *3. Workshop Software-Reengineering, Bad Honnef (10./11.Mai 2001)*, Fachberichte Informatik, Universit t Koblenz-Landau, Nr. 1/2002, pages 73-76
- [7]. Davide Lorenzoli, Leonardo Mariani and Mauro Pezzè, Inferring State-Based Behavior Models, in *proceedings of the 4th International Workshop on Dynamic Analysis (WODA 2006) co-located with the 28th International Conference on Software Engineering (ICSE 2006)*, ACM, Shanghai (China), 23 May, 2006
- [8]. Grady Booch, James Rumbaugh, Ivar Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesley, 1999.
- [9]. J. Rumbaugh. *Object-Oriented Modeling and Design [M]*. Prentice Hall. 1991: pp. 165~168.
- [10]. Jeremy W. Nimmer and Michael D. Ernst. Static verification of dynamically detected program invariants: Integrating Daikon and ESC/Java. In *Proceedings of RV'01, First Workshop on Runtime Verification*, Paris, France, July 23, 2001.
- [11]. K. Rustan M. Leino, Greg Nelson, and James B. Saxe. ESC/Java user's manual. Technical Report 2000-002,

Compaq Systems Research Center, Palo Alto, California,
October 12, 2000.

- [12]. Jeremy W. Nimmer and Michael D. Ernst. Automatic generation of program specifications. In *ISSTA 2002, Proceedings of the 2002 International Symposium on Software Testing and Analysis*, (Rome, Italy), July 22-24, 2002, pp. 232-242
- [13]. JTB, <http://compilers.cs.ucla.edu/jtb/jtb-2003/>. Accessed Sept 2006
- [14]. JavaCC, <http://javacc.dev.java.net/>. Accessed Sept 2006
- [15]. Graphviz, <http://www.graphviz.org/>. Accessed Sept 2006
- [16]. Daikon, <http://pag.csail.mit.edu/daikon/>. Accessed Sept 2006
- [17]. JavaAssist, <http://www.csg.is.titech.ac.jp/~chiba/javassist/>. Accessed Sept 2006