文章编号:1001 - 9081(2006)10 - 2483 - 03

基于 AOP 的程序状态可视化方法研究与实现

邵付东,刘晓东,杨保明
(西安交通大学 电子与信息工程学院,陕西 西安 710049)
(fudongshao@126.com)

摘 要:论述了面向方面的编程(AOP)概念及其在处理横切关注点方面的优势,研究了其在程序状态可视化方面的应用,并利用 AOP 技术实现了一个表达式求值时的程序状态可视化的实例。程序状态可视化功能常常表现为一种横切关注点,相比利用面向对象的编程(OOP)来实现程序状态可视化功能,AOP 技术提供了一种更加松耦合且更具模块性的实现方式。

关键词:面向方面的编程;横切关注点;程序可视化;面向对象的编程;AspectJ
中图分类号: TP311.52 **文献标识码**:A

Research and implementation of program condition visualization based on AOP

SHAO Fu-dong, LIU Xiao-dong, YANG Bao-ming
(*Department of Computer Science and Engineering, Xi'an Jiaotong University, Xi'an Shaanxi* 710049, *China*)

Abstract: This paper described the concept of Aspect Oriented Programming (AOP) and its advantages of solving crosscutting concerns, discussed its application in the field of program condition visualization and presented an example of program condition visualization in the program of Evaluation of Expression by using AOP technology. The function of program condition visualization usually appears as crosscutting concern. Compared with Object Oriented Programming (OOP) technology, AOP technology provides a more loose - couple approach to modularize crosscutting concerns.

Key words: Aspect Oriented Programming (AOP); crosscutting concern; program condition visualization; Object Oriented Programming(OOP); AspectJ

0 引言

在软件开发的过程中,为了方便调试程序或者帮助程序理解等需求,常常需要将系统中的一段程序运行时的状态或结果进行可视化。一般的做法是,将可视化的程序代码做成模块,然后直接嵌入到相关的系统中。对于按照结构化程序设计思想建立的系统,这段可视化代码通常采用函数的形式嵌入原系统。而对于按照面向对象思想建立的系统,这些可视化代码一般会被包装成对象加入原系统。在系统中如何处理这段代码使得系统易于扩展,以适应不断变化的需求,人们提出了多种解决方法,如 MVC 模型、Observer 模式等等。

然而,这些解决方法也不能完全的将可视化代码模块化,可视化代码依然作为系统不可分割的一部分,从而使得可视化部分与系统的其他部分紧耦合。随着系统应用和核心功能的不断扩展,相应的可视化代码与系统的核心功能部分掺和在一起,造成系统中的代码混乱、分散,导致系统程序的可读性差、难于扩展、可重用性低。

面向方面的编程(Aspect Oriented Programming, AOP)是一种关注点分离技术,它模块实现系统的关注点,再将这些实现松耦合的组织在一起来建立最终的系统。利用 AOP 技术,可将类似可视化这样横切整个系统的关注点封装在方面(Aspect)里,在不干扰原系统的情况下,将可视化的代码与原系统松耦合的结合在一起,使可视化和原系统的核心功能相分离,使得程序易读,方便扩展,并提高了代码的可重用性。

1 面向方面的编程

一般来说,编程语言决定了程序的基本单位,这种基本单位体现了一个关注。按照程序基本单位所体现的关注来进行的关注划分,称为主流划分,程序基本单位称为主流模块。按照 Dijkstra 的观点,在对系统进行分割时往往存在着多种关注,然而一般的程序语言不可能在不同的模块中实现不同的关注,必然导致同一模块实现多种关注,如在类中可能既夹杂着对同步的处理也夹杂着对错误的处理,这种对关注的实现相互交叉的现象称为横断现象。横断现象是导致代码复杂难懂、难维护、适应性和可复用度降低等的直接原因[1]。常常把同步、安全、日志等这些横断整个系统的非功能性需求称为横切关注点或者系统级关注点,而把体现系统业务性能的功能性需求称为一般关注点或者核心级关注点。

AOP 是一种关注点分离技术,通过运用方面这种程序设计单元,允许开发者用一种松散耦合的方式来实现独立的关注点,再组合这些实现来建立最终的系统。方面是 AOP 提供的一种程序设计单元,它将横切关注点封装实现为独立的模块;方面之于 AOP 如同类之于 OOP 一样,不过方面关注的是系统的横切关注点,而类更多关注的是系统的一般关注点。AOP 技术使设计和代码更加模块化和更具结构性,使关注点局部化而不是分散于整个系统中,同时还和系统其他部分保持良好定义的接口,从而真正达到"分离关注点,分而治之"的目的[2]。

收稿日期:2006 - 04 - 03;**修订日期**:2006 - 06 - 12 **基金项目**:国家 863 计划项目(2003AA209021)

作者简介:邵付东(1983 -),男,云南昭通人,硕士,主要研究方向:软件工程与虚拟现实技术应用; 刘晓东(1954 -),男,陕西西安人,副教授,主要研究方向:人工智能与虚拟现实技术的研究与应用; 杨保明(1977 -),男,湖北孝感人,硕士,主要研究方向:虚拟现实技术应用与人工智能.

使用 AOP 技术来开发系统有如下一些优点：

1）模块化横切关注点：AOP 用最小的耦合处理每个关注点，使得横切关注点也是模块化的。这样产生的系统，其代码的冗余小。模块化的实现还使得系统容易理解和维护。

2）系统容易扩展：由于方面模块根本不知道横切关注点，所以很容易通过建立新的方面加入新的功能。另外，当系统中加入新的模块时，已有的方面自动横切进来，使系统易于扩展。

3）设计决定的迟绑定：使用 AOP，设计者可以推迟为将来的需求作决定，因为他可以把这种需求作为独立的方面很容易地实现。

4）更好的代码重用性：AOP 把每个方面实现为独立的模块，模块之间是松散耦合的。松散耦合的实现通常意味着更好的代码重用性，AOP 在使系统实现松散耦合这一点上比 OOP 做得更好。

AOP 构建在现有的技术之上，它利用方面来实现系统的横切关注点，利用现有技术来实现系统的一般关注点，并提供一套机制来将两者组合在一起实现最终的系统。图 1 显示一个 AOP 系统结合 OOP 来开发软件的过程的简单抽象。

2 基于 AOP 的开发步骤

AOP 能够利用“方面”来模块化系统的横切关注点，OOP 利用类来模块化实现系统的一般关注点。如果将关注一般关注点的 OOP 方法和关注横切关注点的 AOP 方法结合，分别从纵向（OOP 方法）和横向（AOP 方法）来构建系统，这样得到的系统将具有更好的可扩展性、复用性和系统的稳定性[3]。利用 AOP 技术来开发系统的一般步骤如图 1 所示。

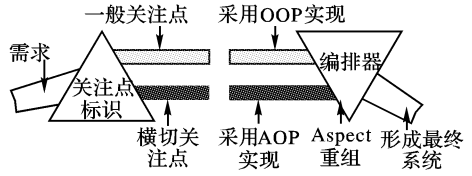

图 1 采用 AOP 的开发步骤

开发步骤可分为如下三个过程：

1）Aspect 分解：分解需求提取出一般关注点和横切关注点，即将一个系统的核心模块关注点和横切关注点分离开来。

2）关注点实现：对于一般关注点，采用 OOP 技术来实现；而对于横切关注点，将采用 AOP 技术。

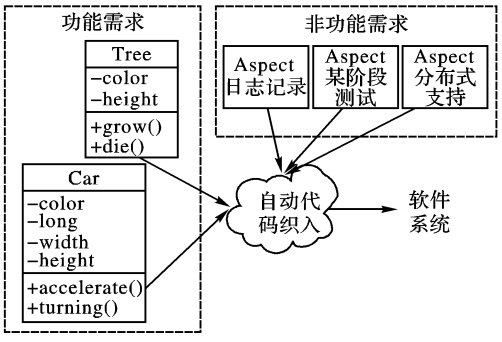

图 2 方面织入过程

3）Aspect 重新组合：方面编排器通过创建的 Aspect 来指定重组的规则，该规则规定了实现的 Aspect 如何与 OOP 实现的基础代码结合来构建最终系统的，这个重组过程被称为织入，织入过程如图 2 所示。通过采用 AOP 自动代码织入技术，将表达功能需求的对象与表达非功能需求的方面结合，从而实现横切关注点与一般关注点在实现上的彻底分离，进而生成结构良好的软件系统。

3 使用方面技术可视化程序运行时状态

程序运行时状态的可视化需求常常属于横切性需求，如果按照传统的编程方式，一方面很难在系统中找到一个合适的位置“插入”这段代码；另一方面，将实现这段可视化需求的程序代码嵌入到系统中，将使得系统的代码混乱，分散，导致系统程序可读性差、难于扩展、可重用性低。通过 AOP 技术，利用方面来模块化实现独立的关注点，上面的问题将会得到较好的解决。下面以表达式求值过程中堆栈的可视化一例来说明如何应用 AOP 技术来实现程序运行时的状态可视化。本例采用面向方面的 AspectJ 语言和 Java 语言来实现（有关 AspectJ 的具体语法请参照 AspectJ 文档[5]）。

表达式求值是程序设计语言编译中的一个最基本的问题，它的实现是栈应用的一个典型例子，利用“算符优先法”可以实现表达式求值[4]。程序执行过程中的主要操作对象是操作数栈和运算符栈两个堆栈。通过这两个堆栈程序可以很容易地完成简单表达式（诸如 2 + 3 * 5）的运算。但是，在设计这个程序时，要很清晰地想象这两个堆栈的变化却是一件很不容易的事。在设计程序时，如果能有一个辅助程序来显示这两个堆栈的变化状况，会有效的帮助准确地把握程序运行的状态。

我们假定已经有一个使用算符优先法对表达式求值的 Java 程序，它共有 4 个类，其基本类结构如图 3 所示。

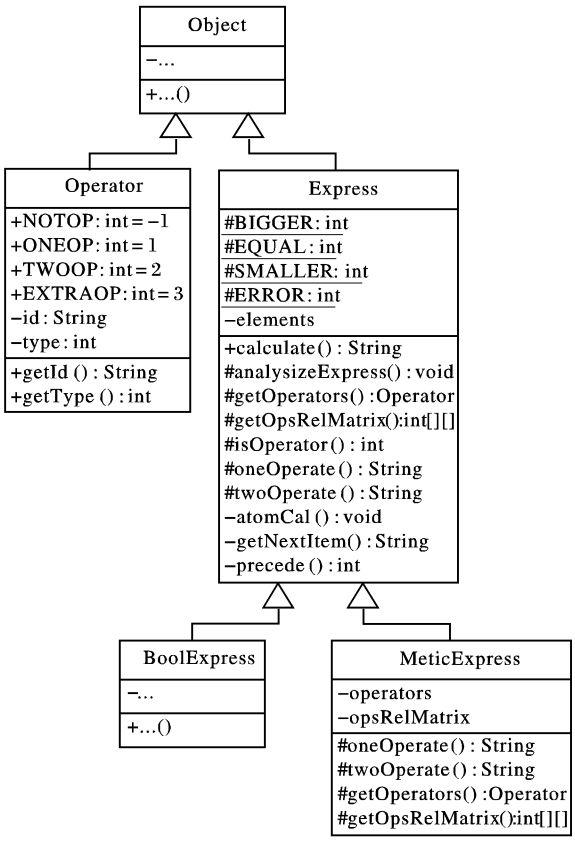

图 3 表达式计算程序的类图

运用算符优先法操作两个堆栈的代码集中在 Express 类的 calculate 方法中：

```
public String calculate( ) throws ExpressException {
  //构造堆栈对象
  Stack OPTR = new Stack( );          //运算符栈(内为 String)
```

```
Stack OPND = new Stack();          //操作数栈(内为 String)
//执行算符优先算法
String data = getNextItem();                    //接收一个子项
while (data != null || !OPTR.empty()) {
  //数据处理完毕,完成剩下的运算
  if (data == null) {
    atomCal(OPTR, OPND);                        //执行一步运算
    continue;
  }
  if (isOperator(data) == Operator.NOTOP) {
    //是操作数
    OPND.push(data);
    data = getNextItem();
  }
  else {
    //比较优先级
    if (!OPTR.empty() && data != null) {
      String aOp = (String) OPTR.peek();
      String bOp = data;
      switch (precede(aOp, bOp)) {
        case SMALLER:
          OPTR.push(data);
          data = getNextItem();
          break;
        case BIGGER:
          atomCal(OPTR, OPND);
          break;
        case EQUAL:
          int opType = isOperator(data);
          switch (opType) {
            case Operator.TWOOP:        //执行某种二元运算
              atomCal(OPTR, OPND);
              break;
            case Operator.EXTRAOP:                //取掉括弧
              OPTR.pop();
              break;
            default:
            throw new ExpressException("不能识别的相等关
            系运算符:" + data);
          }
          data = getNextItem();
          break;
        default:
        throw new ExpressException("无法判断运算符\"" +
          aOp + "\"与\"" + bOp + "\"的关系.");
      }
    }
    else if (OPTR.empty()) {
      OPTR.push(data);
      data = getNextItem();
    }
  }
}
//返回计算结果
return (String) OPND.pop();
}
```

针对这段代码,可以编写一个方面来模块化堆栈的运行时状态的可视化需求。这个方面捕获堆栈对象(Stack)的构造事件,以便在程序构造堆栈时将该堆栈放入定义的监视器(InspectFrm)中。同时,这个方面还要捕获堆栈对象的 push()方法和 pop()方法的调用事件,从而堆栈数据发生改变时通知监视器显示这一变化。方面的代码如下:

```
import java.util.Stack;
import java.awt.Graphics;
aspect InspectStack{
  private int stopTime = 1000;
  static InspectFrm inspectFrm = new InspectFrm();
  //Stack 创建切入点
  pointcut create(): call(Stack.new(..));
  pointcut in(Object obj): call(public Object Stack.push(Object))
    &&args(obj);
  pointcut out(): call(public Object Stack.pop());
  /*在创建切入点处用 returning 捕获新创建的对象*/
  after() returning(Stack s): create(){
    inspectFrm.addInspectStack(thisJoinPoint.toString(), s);
  }
  /*捕获入栈操作(暂停的目的是便于观察)*/
  after(Object obj): in(obj){
    inspectFrm.update();
    try{
      Thread.currentThread().sleep(stopTime);
    }catch(InterruptedException e){
    }
  }
  /*捕获出栈操作*/
  after(): out(){
    inspectFrm.update();
    try{ Thread.currentThread().sleep(stopTime);
    }catch(InterruptedException e){
    }
  }
}
```

至此,由于采用了“方面”来实现堆栈运行时状态的可视化需求,程序的结构演化为图 4 所示的形式。可以看出,方面的加入不会影响程序纵向结构,而且还很好的保证了功能性概念与非功能概念的正交分解,从而达到可视化需求和表达式求值之间松耦合。

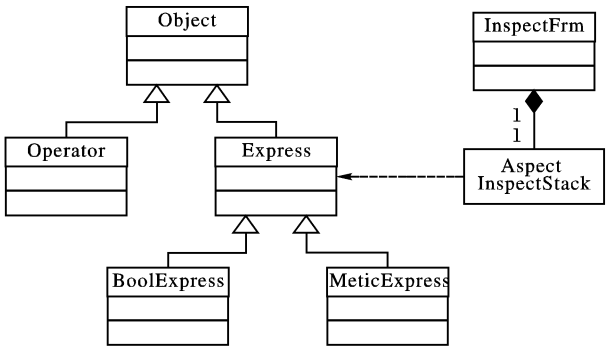

图 4　加入方面后表达式计算程序的基本结构

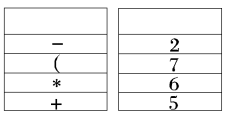

图 5　堆栈状态示意

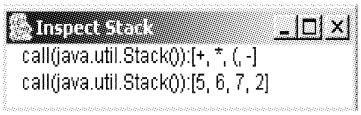

图 6　堆栈运行时状态

通过使用 AspectJ 语言提供的方面织入机制,可以很方便的将堆栈可视化这个“方面”和表达式求值的程序松耦合的织入在一起。图 5 显示了在计算表达式 5 + 6 * (7 - 2)/3 时,开发人员经过分析得到的存在于脑海中的映像,是开发人员对程序运行状态“想象”的结果。图 6 显示了利用方面实现可视化功能后的效果。

4　结语

程序运行时状态的可视化常常被用于调试程序、结果显示、过程演示及帮助程序理解,是实现人机交互友好的重要方式。本文通过一个实际例子介绍了使用AOP技术解决程序

(下转第 2493 页)

欲验证该模型是无死锁的和公平的,需要验证以下性质:

A:该算法是无死锁的,即银行家没有足够的资金了,但客户的要求仍无法满足;

B:该算法是公平的,即每个客户的贷款申请最终都会得到满足;

根据系统模型定义原子命题,在 LTL&PN 中原子命题是根据 Petri 网上的 Token 数来定义的,我们可以得到以下原子命题(其中 $BANK = 10$ 表示 $BANK$ 中有 10 个 Token)。

$P_{00} :: BANK = 10$;

$P_{10} :: CREDIT_1 = 8$;$P_{11} :: CLAIM_1 = 8$;

$P_{20} :: CREDIT_2 = 6$;$P_{21} :: CLAIM_2 = 6$;

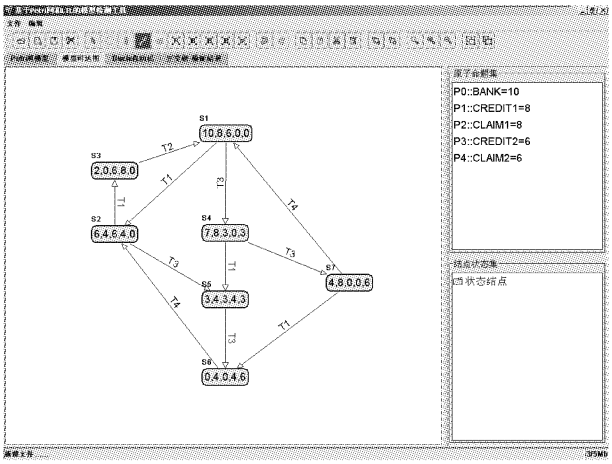

图 3　银行家问题的可达图

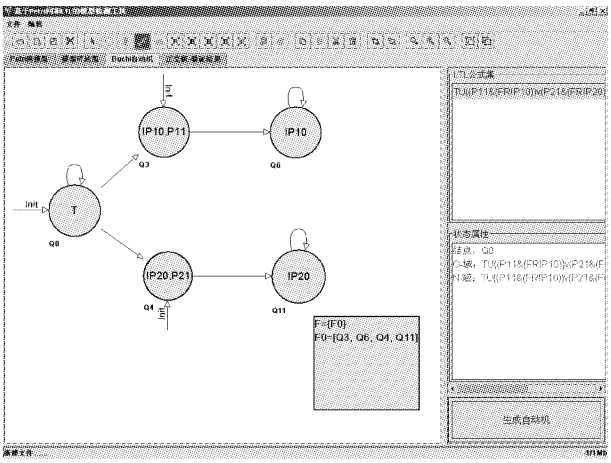

图 4　性质 B 取反后的等价 BA

考虑对性质 B 进行验证,可得到系统应满足的性质:

$$\varphi = \bigwedge_{i \in \{1,2\}} G(P_{i1} \wedge F(P_{i0}))$$

对其取反并化为 NNF 式为:$true \cup ((P_{11} \wedge (falseR\neg P_{10})) \vee (P_{21} \wedge (falseR\neg P_{20})))$,可得到如图 4 所示的等价 GBA,由于其只有一个可接受状态集,所以其实际为一个 BA,Q_3、Q_4、Q_6、Q_{11} 为其可接受状态。

这样,可以通过 3.2 节的算法来计算模型可达图与性质 B 取反后的等价自动机 $A_{\neg\varphi}$ 的正交积,通过检验正交积的可接受语言即正交积中是否有从初使状态可达的包含可接受状态的强连通分量来进行模型检测,由 LTL&PN 得到的正交积如图 5 所示。由图可知,存在着一条由初始状态 $< S_1, Q_0 >$ 经过 $< S_2, Q_4 >$ 到达的回路($< S_3, Q_{11} >$,$< S_1, Q_{11} >$,$< S_2, Q_{11} >$),由 $A_{\neg\varphi}$ 可知,Q_{11} 为可接受状态,所以性质 B 不为真。这条回路表明客户 1 在无限的等待银行的贷款而始终没能得到满足,所以系统模型不满足公平性原则。

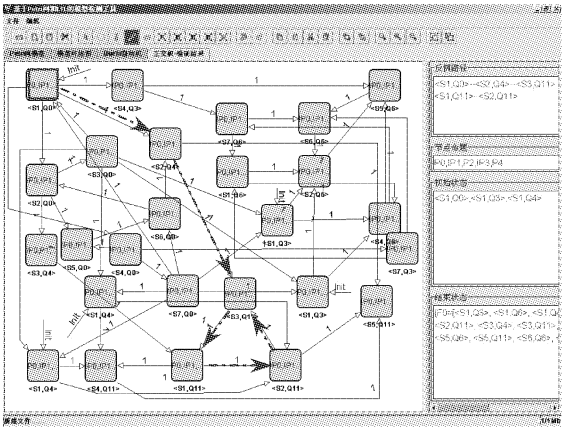

图 5　系统模型与 $A_{\neg\varphi}$ 的正交积

5　结语

本文给出了一种实际可行的模型检测方法,并通过对 Büchi 自动机的化简,在一定程度上提高了模型检测的效率,可有效的避免状态空间爆炸的问题。根据文中所提出的模型检测方法,课题组开发了基于 LTL 和 Petri 网的模型检测工具,结果表明,本工具可以很好的对 Petri 网所建立的系统模型进行检测。

参考文献:

[1] LAMPORT L. The Temporal Logic of Actions [J]. ACM transaction on Programming Language and Systems, 1994, 16(3): 872 - 923.

[2] PNUELI A. The Temporal Semantics of Concurent Programs [J]. Theoretical Computer Science, 1981, 13: 45 - 60.

[3] GERTH R, PELED D, VARDI MY, *et al.* Simple on-the-fly automatic verification of linear temporal logic[A]. In PSTV 1995, Warsaw, Poland, Chapman Hall[C]. 1995. 3 - 18.

[4] SOMENZI F, BLOEM R. Efficient Büchi Automata from LTL Formulae[D]. Department of Electrical and Computer Engineering University of Colorado, Boulder, CO, 80309 - 0425.

[5] GIANNAKOPOLOU D, LERDA F. Efficient translation of LTL formulae into Büchi automata[R]. RIACS Technical Report 01. 29 June 2001.

[6] PELED DA. Software Reliability Methods[Z]. Springer, 2001.

[7] PENUELI P. The Temporal Semantics of Concurent Programs [J]. Theoretical Computer Science, 1981, 13: 45 - 60.

(上接第 2485 页)

状态可视化问题的一种方法。只要对例子稍加修改,该方法就可以满足很多同类型的问题。如通过对多线程的调用关系的可视化可以随时监测到线程死锁,通过对遗留系统中的调用关系进行可视化可以在遗留系统的现代化改造过程中帮助程序理解等等。在此过程中采用 AOP 技术使得编程人员可以不用更改原系统的任何代码,就能够让这些非功能性需求以一种松耦合的形式织入到系统中。

参考文献:

[1] 邓阿群,厉小军,俞欢军,等. 一种新型软件设计方法 AOP 的研究[J]. 系统工程与电子技术,2004,(7):970 - 975.

[2] 曹东刚,梅宏. 面向 Aspect 的程序设计——一种新的编程范型[J]. 计算机科学,2003,(9):5 - 10.

[3] 魏巍,徐全生. 沈阳工程学院学报(自然科学版)[J]. 2005,(1):75 - 78.

[4] 严蔚敏,吴伟民. 数据结构[M]. 北京:清华大学出版社,1997.

[5] GRADECKI JD, *et al.* 精通 AspectJ[M]. 王欣轩,吴东升,等译. 北京:清华大学出版社,2005.