基于依赖性分析的对象行为协议逆向恢复*)

黄 洲 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)

摘 要 对象行为协议对于理解对象行为语义、对象行为验证、测试以及指导其他开发者正确使用对象所提供的外部行为都有十分重要的意义。然而在很多遗产系统中,对象行为协议常常缺失或随着长期的代码维护而出现不一致。针对这一问题,本文提出了一种静态的对象行为协议逆向恢复方法。该方法首先通过源代码分析获取对象(类)内部各方法之间直接和间接的依赖关系,然后在对象(类)内部依赖关系的基础上构建行为协议状态机。由于对象(类)内部的依赖关系是对象行为约束的主要根源,而静态分析具有全面、准确的优点,因此该方法获得的行为协议具有较好的准确性,而相关的实验结果也很好地验证了这一点。

关键词 行为协议,抽象状态图,方法依赖,静态分析,再工程,逆向工程

Behavior Protocols Recovery Based on Dependency Analysis

HUANG Zhou PENG Xin ZHAO Wen-yun

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

Abstract The behavior protocol of object has the critical meaning in understanding and integrating the object into development. Unfortunately, protocol is rarely maintained during the development of lagecy systems. To handle the problem, this paper proposes a technical to recovery the behavior protocol of object in the form of Finite State Machine. It extracts dependency of functions through a static source-code analyse procedure and reconstruct the relations between states. The method offers an accurate and understandable result to instruct the developers in reengineering procedure.

Keywords Behavior protocol, Abstract state graph, Function dependence, Static analysis, Reengineering, Reverse engineering

1 引言

面向对象系统中的对象通过 public 方法对外提供服务接口,体现对象的外部行为。由于对象是状态、数据和内部及外部行为的封装体,因此对象的各个 public 方法的调用之间往往存在着某种序列关系。这种对象外部行为之间的序列关系一般称为行为协议,通常可以用有限状态机进行描述。正确的协议描述在对程序进行理解、测试和重用等方面都有十分关键的作用。

UML 状态图可以描述一个类或者一组类单元的状态转移行为。类对象的行为被描述为一组状态变迁的组合,每个变迁都导致程序从一个状态迁移到另一个状态。状态图显式地定义了程序中对象所能进行的行为以及行为造成的结果,能够让开发人员从行为的角度来描述程序的协议规约,用来指导开发和测试维护过程[1,2]。不过,在相当一部分的遗产系统中,由于时间的关系或者人为的疏忽,经常会导致程序协议的缺失或随着长期的代码维护而出现不一致的情况。如何从遗产系统中恢复出与之匹配的行为协议已经成为一个十分活跃的研究课题,相关的方法也大量出现[3-8]。这些方法大致可以分为基于动态分析的方法和基于静态分析的方法两大类。动态方法关注的是程序运行时的执行情况。它的一般步骤是先对程序源代码进行插装(或称为注入),然后执行一定数量的测试用例以获取程序的实际运行轨迹,再从提取到的运行信息中挖掘出程序的行为协议。动态分析有两个最大的限制,首先动态分析往往需要对源程序进行一定的修改工作,以能够跟踪程序的运行情况,这可能要求分析人员对程序有一定的理解。其次,也是更重要的一点,就是动态测试受限于测试用例的质量,分析结果可能因为测试用例的不同而有明显的不同,甚至因为测试用例设计的欠妥,而导致错误的分析结果出现。静态分析的方法可以避免动态方法的上述弱点。静态分析的方法直接从静态源代码中提取隐含的协议信息。本文中以接口方法之间的依赖关系作为观察对象,尝试采用一种静态的基于源代码解析的行为协议提取工作。在研究过程中,我们开发了一个辅助的原型工具。借助这个工具可以从程序的源代码中自动提取出我们感兴趣的协议信息。

本文剩下的部分按以下的结构进行组织:第 2 部分介绍了一些背景知识和相关技术,并给出了一个实验例子;第 3 部分是本文的主体,介绍了我们理论的方法;第 4 部分介绍了我们原型工具实现的一些细节和研究过程中的一些经验讨论;最后是总结。

2 相关背景及方法用例

2.1 方法依赖

对象(类)内部的依赖关系是对象行为约束的主要根源,由于对象是状态、数据和内部及外部行为的封装体,因此对象的各个方法的调用之间往往存在着某种依赖关系,称为方法

*)国家 863 计划(2007AA01Z125)、国家自然科学基金(60473061,60703092)。黄 洲 硕士生,主要研究方向为软件再工程;彭 鑫 博士,讲师,主要研究方向为软件体系结构、软件产品线、软件再工程;赵文耘 教授,博士生导师,CCF 高级会员,主要研究方向为软件工程及电子商务、企业应用集成。

的行为依赖。方法的依赖关系可以通过方法的前后置条件进行描述:方法的前置条件就是为了正常调用该方法所需要满足的一组属性,前置条件满足了,方法才能正常被执行。方法的后置条件是接口调用后所产生的属性。一个方法依赖于另一个方法就可以表示为这个方法的前置条件依赖于另一个方法的后置条件。在程序中,方法之间的行为依赖具有两层的关系。一层我们称之为语法级的依赖关系;另一层称为语义级的依赖关系。语义级的依赖关系一般都涉及到需求相关的程序规约,比如领域规则、业务逻辑等等,在本文中将暂不作考虑。而语法级的依赖关系,是一种代码级的依赖关系,主要是基于变量共享而产生的,如果方法 $F1$ 使用到了在 $F2$ 中定义的变量,那么就称 $F1$ 依赖于 $F2$[9]。方法中如果直接使用了一个共享变量的值(直接引用或者调用了这个变量所具有的接口),那么必定要求这个变量在某个地方要被初始化(定义的地方,构造函数,或者某个初始化方法)等等。这就是方法之间隐含存在的约束。这种约束可以通过静态分析有效地提取出来。本文中暂不考虑通过外部介质形成的共享,例如外部文件、数据库标记等。

2.2 抽象状态图

作为描述行为协议的状态图,可以用一个三元组 $P=\langle V,A,T\rangle$ 进行描述。V 是状态的有穷集合;A 是活动的集合;$T\subseteq V\times A\times V$ 是状态迁移的集合。其中,程序状态 V 可以是具体的,也可以是抽象的。具体的状态通常是根据程序中状态变量的具体取值进行描述,状态变量的值不同,就表示了程序所处的状态不同。而抽象的程序状态则是通过程序的一些抽象性质对程序的状态进行描述[10]。具体状态图由于状态变量值的复杂性,进行状态描述时的难度较大,并且产生的状态图中,冗余的状态相当多,不利于对系统行为进行观察和使用。相对而言,抽象状态图往往关注于程序中的某个方面的抽象属性,体现出这个属性在程序运行过程中变化的情况,这令它的状态空间比较小,使建模的难度和模型的复杂性都大为降低。在文献[4]中,取的是程序的分支覆盖情况作为观察的抽象属性,分支覆盖的变化形成状态的迁移。文献[6]中提出了一种抽象状态的通用描述方法和恢复工具。在我们的方法中,将程序在运行过程中的行为能力定义为程序的状态。我们认为,在程序执行过程中,一个状态之所以与另一个状态不同,实际上应该是它们使能的活动集合存在着不同[11]。一个活动可以被称为使能的当且仅当它在当前状态下,可以被执行。在具体程序中,活动就被映射为 j 接口调用。程序的状态在本质上就是程序所能进行的行为能力的一种抽象,程序状态的改变,其实就是程序所能进行的行为发生了改变。我们的方法尝试直接对程序的行为能力进行状态建模,恢复出程序的状态图。

2.3 例子

我们的原型工具是面向 Java 语言的一个分析工具,这里我们引入一个简单的 Java 例子作为处理对象,来演示我们方法的流程。它的主要功能是建立数据库连接,并且验证用户输入的用户名和密码。这种处理流程可以在相当多的现有程序中找到。为了简便,我们省略了业务相关的处理流程,只保留了简单的框架结构。程序中包含了 5 个类变量(共享变量),四个公共接口方法。setParam()方法用于设置数据库连接参数;connect()方法建立数据库连接;connectByParam()方法根据输入参数建立数据库连接;最后 login()方法使用数据库连接验证登录参数。可以看到,当这个类被创建后,并不是所有的接口方法都可以马上被调用,例如 login()方法。因为此时,方法中使用到的变量都还未被赋过值,此时它们是无意义的(connected),甚至包含错误的(conn)。只有随着其他方法的执行,比如 connectByParam(),login()方法才能被正常调用。这说明调用 connectByParam()前后程序的状态是有所不同的。这种状态的不同从表面上看是 conn,connected 等变量的值发生了变化,不过归根结底是程序所能进行的行为发生了变化:login()从不能被调用变成能够被调用。我们将把这种行为能力定位为我们所要恢复的抽象状态。

```
public class Login {
        String url;
        String dbuser;
        String dbpass;
        boolean connected;
        Connection conn;
        String userInfo;
        public void setParam(String aurl, String auser, String apass) {
                url = aurl;
                dbuser = auser;
                dbpass = apass;
        }
        public void connect() throws Exception {
                conn = DriverManager.getConnection(url, dbuser, dbpass);
                connected = true;
        }
        public void connectByParam(String aurl, String auser, String apass) throws Exception {
                setParam(aurl, auser, apass);
                connect();
        }
        public void login(String username, String password) {
                if (connected) {
//...do something with conn
userInfo = "some info";
                }
        }
public void disconnect() {
                conn.close();
        }
public void reset() {
                url = null;
                dbuser = null;
                dbpass = null;
          conn = null;
        }
}
```

图 1 一个简单的登录程序

3 行为协议抽取方法

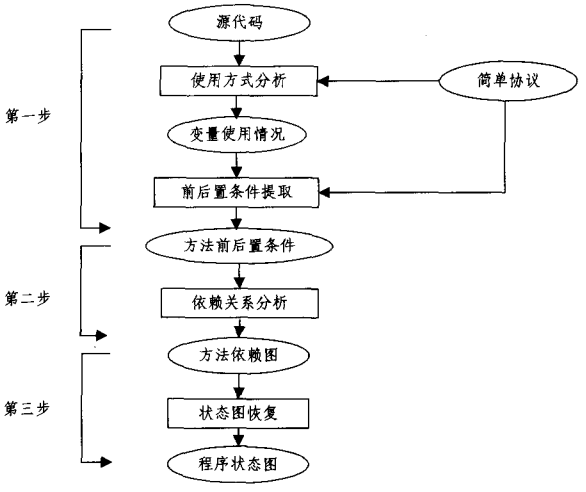

图 2 方法基本步骤

本文的状态机恢复方法总共可分为三个步骤:第一步是对源代码进行分析,获取方法中变量的使用情况,目的是通过自动分析提取出方法的前置条件和后置条件;第二步的工作是根据第一步中得到的前后置条件,建立方法之间的依赖关系。最后,通过这些依赖关系,就可以提取出方法之间隐含的调用协议,从而恢复出最终的状态图表示,这将在第三步中进行。图 2 中展示了我们方法的一个简单的流程图。

3.1 提取方法的前置条件和后置条件

本文中接口的前后置条件,是根据方法中涉及的全局变量的状态进行定义(通常只有全局变量才能在方法之间产生联系,所以本文中我们只考虑全局变量在方法中的使用情况)。变量根据在方法中的使用方式可分为三类:作为赋值属性,作为取值属性,以及特殊属性。

定义 1

1. 赋值属性：变量作为赋值属性使用即变量在赋值语句的左边出现(除特殊属性情况外)。

2. 取值属性：其余的情况下变量都作为取值属性出现(除特殊属性情况外)。

3. 特殊属性：为对象赋 null 值以及执行一些对象自带的析构方法的情况。

(例如 Connection 对象的 close()方法)。

作为赋值属性使用时变量体现为一种地址的含义，其使用目的是为该地址所引用的变量进行赋值。任何状态下，对象都可以作为赋值属性而被使用；作为取值变量的时候，变量作为具体的对象值来进行使用，这要求对象此时的状态满足其使用的条件；否则，对于变量的使用将是无意义的，甚至错误的。作为特殊属性使用后，变量将自身重置为一种无意义的状态，在状态被修改前，无法作为取值使用。

方法的前后置条件和方法使用的变量的状态有相当大的关系。一部分对象(比如：String 类对象)的状态通常只分为未初始化和已初始化。另一部分对象(比如：Connection 对象)，除了初始化情况外，还存在一些行为协议造成的状态(比如调用 Connection 对象的 close()方法后将无法继续对该对象进行使用)。简单对象的状态可以被直接指定，或认为只具有初始化相关状态。而更复杂的对象可以由它所包含的简单对象的状态组合而来。因此，提取复杂对象的状态是一个递推的过程。最终，方法的前后置条件由方法中使用到的变量对象的状态条件组合而成。方法的前置条件表示为对各个变量对象状态的要求，后置条件表示为对各个变量对象状态的改变结果。

例如，上例中的 Login 对象用到了 conn 变量执行操作。根据 Connection 的协议知道 conn 变量在执行 close()方法后将不能继续使用。disconnect()方法里面包含了 conn. close()的执行，那么根据 Connection 的协议知道 disconnect()的执行结果是 conn 变为 closed 状态。

下文中对对象状态进行了简化，将对象所具有的状态分为两类：

定义 2

1. 不可使用状态(NotAvaiable)：当变量为 null 或变量执行了析构方法。

2. 可使用状态(Avaiable)：其他情况。

使用方式和变量状态之间存在以下的关系：变量作为特殊属性使用后，一定处于不可使用状态；只有处于可使用状态的变量才可以被作为取值属性使用。

定义 3 全局变量 x 在方法 A 中未出现，但是方法 A 调用了方法 B，B 中引用到了 x，则 x 的使用方式和状态修改也将在方法 A 中被定义(即使它不在方法 A 中出现)，这称为间接引用。

有了变量的使用方法和使用状态后，就可以定义方法的前后置条件了。

定义 4 F 为程序中的一个方法，设方法的前置条件集合为 $Pre(F)$，方法的后置条件为 $Post(F)$，定义为：

$Pre(F)=\{x \mid x$ 为全局变量 && x 在 F 中作为取值属性使用 $\}$

$Post(F)=\{(x,y) \mid x$ 为全局变量，y 为 x 在方法执行后的状态 $\}$

从语法的角度来讲就是，在调用方法 F 时，$Pre(F)$ 中的全局变量都必须出于可使用状态。只要这个属性条件满足，那么方法的执行在语法方面就不会有问题，能被正常执行的。此时，方法的后置条件就对应于方法的执行对全局变量的初始化状况的影响，即 $Post(F)$ 中得全局变量在执行方法 F 后的状态。有些方法没有前置条件，例如类的某些构造函数以及一些初始化的方法，它们可以在任意状态下被调用。而另外一些方法的前置条件必须通过执行其他方法来进行满足。

同一个变量 x 可能在一个方法中多次出现，所以需要谨慎判断 x 在方法中的使用方式和最终的状态变化。

准则 1 x 为全局变量，x 在方法 F 中先后以不同的使用方式出现，则

(1)x 的使用方式以它在方法中最先出现的使用方式为准；

(2)x 的使用状态以它在方法中最后被改变的使用状态为准。

例如，x 先被赋值，然后又作为值被使用。或者先作为取值属性被使用，然后又被重新赋值。这样，前者的使用方式被定义为赋值属性，后者被定义为取值属性。

最后要提到的一个情况是分支的问题。在分支语句和循环语句中，根据分支条件的不同，全局变量的使用方式可能会不同(或出现未使用的情况)。由于这涉及到运行时的信息，在我们的实现中暂时未考虑这种分支的区别处理。

定义好方法的前后置条件以后，我们通过一个基于抽象语法树的原型工具，对源代码进行解析处理，来自动提取每个方法的前后置条件。表 1 就是前面例子经过我们工具分析后的方法条件表，要注意的是方法 connectByParam，在例子中，这个方法没有直接引用到任何全局变量，但是通过方法 setParam 和 connect 间接引用到。

3.2 根据方法的前后置条件构建方法的依赖图

根据第一步中提取出的方法的前后置条件信息，我们就可以进行方法之间依赖关系的构建。

两个方法之间的依赖关系是由于在方法体中引用了同一个全局变量而产生的。对于同一个全局变量来说，对其进行赋值的方法和对其进行使用的方法之间存在依赖关系。具体定义如下：

定义 5 设 F 和 G 为程序中的两个方法，x 为变量集合，如果 $x \subseteq Pre(F)$ 并且 $(x,A) \subseteq Post(G)$，则称方法 F 依赖于方法 G(方法 F 中使用到的部分变量集合 x 会被方法 G 置为可使用 Avaiable)。

特别地，如果 $(Pre(F),A) \subseteq Post(G)$，则称方法 F 全依赖于方法 G(方法 F 中使用到的变量全部会被方法 G 置为可使用)。

表 1 方法前后置条件表

	setParam	connect	connectBy Param	login	disconnect	reset
Pre-Condition	None	urldbpass dbuser	None	conn connected	conn (A)	None
Post-Condition	url (A) dbpass (A) dbuser (A)	conn (A) connected (A)	url (A) dbpass (A) dbuser (A) conn (A)	userinfo (A)	conn (NA)	url (NA) dbpass (NA) dbuser (A) conn (NA)

需要注意的是，这种依赖关系同样是建立在语法级别上的。同一个全局变量 x 可能在多个方法内部作为赋值属性使用，同时也可能在多个方法内部作为取值属性使用，由 x 产生的依赖关系是一种多对多的关系。如果存在一个方法 F 通过一个全局变量 x 依赖于方法 G，并不意味着在语义的调用顺序上，方法 F 只能在调用方法 G 后才能被调用。

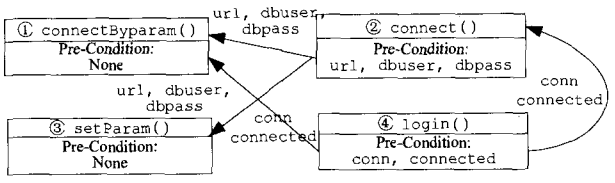

图 2　方法依赖图

图 2 是根据表 1 中提取出的信息构建出的方法依赖图。其中，disconnect 方法和 reset 方法各自独立于其它方法，图中就没有标出。图中的箭头表示了依赖的方向，而边上的变量名表示了依赖的内容。这样，方法之间隐含的依赖关系，就已经被显示地表示出来。

3.3　根据方法的依赖图生成程序的抽象状态图

前面提到过，程序状态所代表的真正含义是程序使能活动的集合。所以，第三步中恢复的抽象状态，以程序当前所能调用的方法集合作为状态变量。只有程序中所能调用的方法集合发生变化，才认定程序的状态发生了迁移。而方法能不能被调用，又取决于它的前置条件是否满足，本文中即变量的赋值情况。因此，与状态相关的上下文环境 $C=(F,V)$，F 是方法的状态空间，它的每一项都是一个名、值对，(function, status)$\in F$，指明了方法和其所处的状态。*Status* 的取值为 Y(可执行)或 N(不可执行)；V 是变量的初始化集合，变量 $v\in V$ 表示它已经被初始化。

F 中包含所有方法的明值对，初始状态下标记都置为 N，变量空间 V 为空集。然后，将所有前置条件为空的方法的标记，置为 Y。接下来的状态图构建过程是一个深度优先的图生成过程，具体如算法 1。算法 1 实际上同时执行了第二步和第三步的工作。

算法 1　状态图生成算法

```
// 输入：一个状态 s 的上下文环境 s:=(F,V),//当前状态集合 S 为全局变量
// 输出：通过状态 s 能作出的所有变迁结果
procedure StateTransfer( s )
var F′:Functions;
var V′:Variables;
var s′:State;
var transferred:Boolean;
for all (f,statf) ∈ F′ do
    create F′ as s.F;
    create V′ as s.V;
    if statf = Y then
        V′ <− update( V′,Post(f) );
        for all (g,statg) ∈ F′ do
            if (statg = N) ∧ (Pre(g) ( V′) then
                if s′ exist then
                    statg:= Y;
                else
                    create s′ as (F′,V′);
                    statg:= Y;
                end if
            end if
        end for
        if s′ <> null then
            for all s″ ∈ S do
                if s″ = s′ then
//创建当前状态 s 到已有状态 s″的变迁，变迁方法为 f;
                    transferred:= true;
                    break;
                end if
            end for
            if not transferred then
//创建当前状态 s 到新状态 s′的变迁，变迁方法为 f;
                S <− S ∪ {s′};
              StateTransfer(s′); //深度优先递归调用算法
            end if
                else
            //创建当前状态 s 到自身的变迁，变迁方法为 f;
        end if
    end if
end for
```

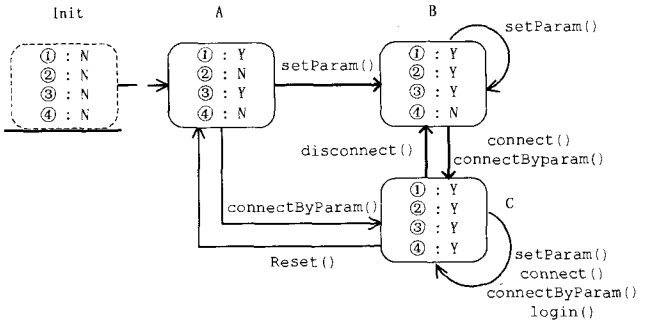

图 3　例子程序所生成的抽象状态图

图 3 显示了上面例子生成的最终状态图，这里用图 2 中方法所对应的标号来代替方法名。其中的 update 方法是用于根据方法的后置条件，对当前变量的空间进行更新。更新包括将不可使用的变量置为可使用，或将可使用的变量置为不可使用。初始状态 Init 下，所有方法的执行标记都为 N。经过初始化后，无前置条件的 setParam 和 connectByParam 的标记被修改为 Y，然后以状态 A 作为当前状态执行算法，此时变量空间为空。首先查找到可执行方法 setParam；将 setParam 的后置条件 Post(setParam)加入到变量空间；然后查找状态为 N 的方法，发现方法 connect 的前置条件被满足，当前还没有创建临时状态，所以创建与状态 A 相同的状态 A'，将 A'中方法 connect 的状态改为 Y。方法 login 的前置条件仍然未被满足。因为创建了 A'，且当前没有状态与 A'相同，所以 A'生成为新状态 B，并以方法 setParam 作为状态迁移。然后以状态 B 作为当前状态，深度优先递归执行算法，此时变量空间为{ url,dbuser,dbpass}。查找到方法 connect 的时候，发现方法 login 的前置条件被满足，所以生成了新状态 C。随着新状态的生成和递归的返回，最终形成图 3 的抽象状态图。

4　实现和讨论

4.1　实现

为了实现对于方法前后置条件的自动提取和方法依赖关系的自动生成，我们开发了一个基于 Java 的原型分析工具。其中，用于第一步提取源代码中方法的前后置条件的部分，我们集成了 Java Tree Builder (JTB)[12]框架。JTB 是一个开源的语法树生成框架，它读入 JavaCC(是一个用 Java 开发的语法分析生成器)[13]生成的语法文件，自动构建语法结构。我们开发了一个深度优先的语法树访问者类，来自动遍历目标源代码的语法树，提取源代码中的信息，本文中主要是提取了全局变量的使用信息。第二步根据第一步中提取出的变量的信息用算法 1 进行计算，获得方法的依赖关系。最终，我们根据方法的前后置条件，使用 Graphviz 来进行程序状态图生成的步骤。Graphviz 是一个开源的图形绘制工具，可以很方便地绘制结构化的图形网络。通过分析方法的执行对前后置条件的影响，逐步生成 Graphviz 的图形描述文件，最终生成的状态图效果如图 4 所示。整个流程基本上是一个全自动的过程，不需要人工的干预。

图 5 中给出了一个更加复杂的例子，可以看出方法之间由于数据依赖关系的存在形成的隐含调用规约。而这种规约可以用我们的方法恢复为状态图的形式。

(下转第 276 页)

tional requirements// 2nd International Conference on Dependability of Computer Systems (DepCoS-RELCOMEX'07)
[2] Heise M, Souquières J. A Method for Requirements E-licitation and Formal Specification// Proceedings of the 18th International Conference on Conceptual Modeling. Lecture Notes In Computer Science, 1999: 309-324
[3] Wilson P, Filho P. Quality Gates in Use-Case Driven Development// International Conference on Software Engineering, Proceedings of the 2006 International Workshop on Software Quality. Shanghai, China, 2006: 33-38
[4] Seffah A, Djouab R, Antunes H. Comparing and reconciling usability-centered and use case-driven requirements engineering processes// User Interface Conference, 2001. AUIC 2001. Proceedings. Second Australasian, 2001: 132-139
[5] Behrens T. Capturing business requirements using use cases. Chief Technology Officer, Alpheus Solutions, Dec. 2004. http://www.ibm.com/developerworks/rational/library/dec04/behrens/index.html? S_TACT=105AGX52&S_CMP=cn-a-r
[6] Li Xiaoshan, Liu Zhiming, He Jifeng. Formal and Use-Case Driven Requirement Analysis in UML// 25th Annual International Computer Software and Applications Conference (COMPSAC'01). 2001
[7] Hamilton K, Miles R. Learning UML 2.0. O'Reilly, April 2006
[8] Li Liwu. A Semi-Automatic Approach to Translating Use Cases to Sequence Diagrams// 29th International Conference on Technology of Object-Oriented Languages and Systems. Nancy, France. IEEE Computer Society, June 1999
[9] Sommerville I. Software Engineering (6th Edition). Ian Sommerville. , China Machine Press: 79

(上接第 268 页)

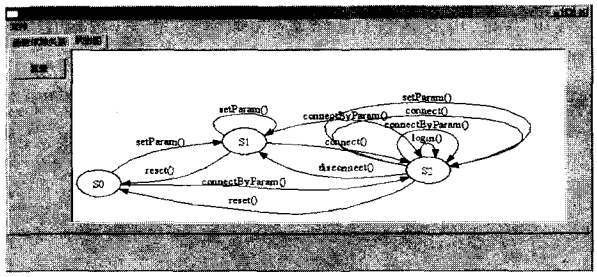

图 4 最终生成的状态图

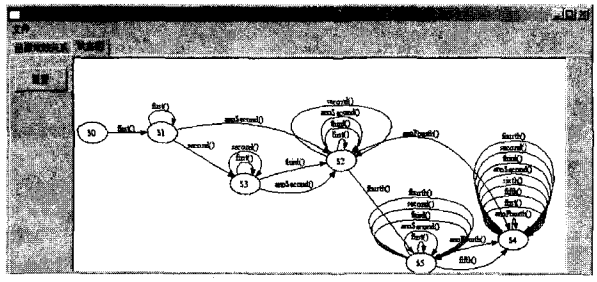

图 5 另一个状态图的例子

4.2 讨论

静态的分析方法对于提取语法级别的依赖来说是十分有效而且准确的。但是我们在实践中也可以明显感觉到静态分析存在的一些弱点。主要的弱点就是静态分析所能提取的信息比较有限，无法对运行时相关的依赖关系进行提取。例如，通过输入参数和一些由分支变量进行控制的状态迁移等语义相关的依赖关系，以及动态绑定等运行时相关的情况，就无法使用静态分析进行处理。因为这些都涉及到运行时变量当时的具体取值，而这些值在编译期是未定义的。但是取值的不同，很可能就导致了程序所采用的行为会有所不同。这些信息将是我们接下来所要关注的。为了弥补静态方法的这些不足，接下来我们会尝试引入一些动态方法进行辅助，比如符号执行(Symbolic Execution)[14]以及程序不变量(Program Invariants)[15]等。这些动态方法也可以提取出前后置条件相关的一些信息。通过动态的执行信息和静态的结构信息相结合，辅助我们的方法，可能提供更强的依赖分析功能。

结束语 本文提出了一种从遗产系统中恢复出程序接口行为协议的方法。最终的接口约束以抽象状态图的形式表示出来。我们的方法通过考察程序中全局变量的共享使用来推测方法之间隐含的依赖关系，恢复出方法调用的行为序列。这是一个自动的静态分析过程，避免了动态方法对测试用例的依赖性。恢复出的抽象状态图保留了协议的约束行为，并减少了复杂度和冗余性。通过实验证明，我们的方法可以有效地对接口方法的协议进行恢复，这对于遗产系统的理解有着重要的帮助。

参 考 文 献

[1] Kim Y, Hong H, Bae D, et al. Test cases generation from UML state diagrams// IEEE Proceedings- Software. 1999, 146(4): 197-192
[2] Briand Y L L C, Penta M D. Assessing and improving state-based class testing: A series of experiments. IEEE Transactions on Software Engineering, 2003, 30(11)
[3] Yang Jinlin, Evans D. Dynamically Inferring Temporal Properties// Proc. the ACM-SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering. 2004: 23-28
[4] Yuan Hai, Xie Tao. Automatic Extraction of Abstract-Object-State Machines Based on Branch Coverage// Proceedings of the 1st International Workshop on Reverse Engineering to Requirements at WCRE 2005 (RETR 2005). November 2005: 5-11
[5] Xie Tao, Notkin D. Automatic Extraction of Sliced Object State Machines for Component Interfaces// Proc. 3rd Workshop on Specification and Verification of omponent-Based Systems at ACM SIGSOFT 2004/FSE-12 (SAVCBS 2004). October 2004: 39-46
[6] Corbett J, Dwyer M, Hatcliff J, et al. Bandera: extracting finite-s-tate models from Java source code// 22nd International Conference on Software Engineering. June 2000
[7] Whaley J, Martin M C, Lam M S. Automatic extraction of object-oriented component interfaces// International Symposium on Software Testing and Analysis. July 2002
[8] Olender K M, Osterweil L J. Interprocedural Static Analysis of Sequencing Constraints. ACM Transactions on Software Engineering and Methodology, 1992, 1(1): 21-52
[9] Tang M H, Wang W L, Chen M H. A UML Approach for Software Change Modeling. cs. albany. edu
[10] Abstract state machines and high-level system design and analysis. Theor. Comput. Sci, 2205, 336 (2-3): 205-207
[11] Zhang Yan, Hu Jun, Yu Xiao-feng, et al. 场景驱动的构件行为抽取. Journal of Software, 2007, 18(1)
[12] http://compilers. cs. ucla. edu/jtb/jtb-2003/
[13] https://javacc. dev. java. net/
[14] King J. Symbolic Execution and Program Testing. Communications of the ACM, 1976, 19(7)
[15] Ernst M D, Cockrell J, Griswold W G, et al. Dynamically discovering likely program invariants to support program evolution// IEEE Transactions on Software Engineering. February 2001
[16] Nimmer J W, Ernst M D. Automatic Generation of Program Specifications. ACM SIGSOFT Software Engineering Notes, 2002