协议软件实现的面向对象的方法

肖 斐 朱崇湘 孙永学 赵文耘
(复旦大学计算机系, 上海, 200433)

摘 要 提出一种用面向对象的方法来实现协议。协议由 *FSM* 说明, 然后 *FSM* 由一组相关的对象实现。对象的成员函数是接口事件, 它们激发状态变迁。与状态变迁相联系的行为构成成员函数的主体。当状态变迁产生时, 一个对象成为另一个对象。文中给出了一个实例, 还提出一个软件工具, 使设计者可以用图形方式编辑状态机, 并且自动产生 *C*+ + 的类定义。

关键词 有限状态机 类 对象 继承 重载

分类号 TP391.72 TP391.75

1 介绍

由于分布式系统日益增长的应用, 通用协议软件的设计和实现成为确保系统正常运行的关键。由于并发性、复杂性和性能的需求, 通用软件的设计与实现十分困难。需要设计技术和工具来辅助设计者完成协议软件。

一种协议软件的设计方法应该满足以下要求: 首先, 由它实现的系统不应有严重的性能问题, 昂贵的 IPC(内部进程通讯)和不必要的数据拷贝应尽可能避免; 第 2, 设计方法既适用于开发新的协议, 又能实现符合现有标准的协议; 第 3, 可扩充性; 第 4, 支持全部或部分的自动实现。还应使设计者采用可视化工具或图形界面来辅助开发协议软件。

本文中我们描述实现协议软件的面向对象方法, 并提出用于产生协议实现所需对象类的工具。在我们的方法中, 协议由扩展的有限状态机(FSM)建模。然后 FSM 被转换成一组相关的对象, 在实现过程中为 FSM 的一个状态定义一个对象。在一个给定的状态, 对应于每个激发状态变迁(从一个到另一个状态或者回到原状态)的接口事件, 为状态对象定义一个成员函数(或称为方法)。与状态变迁相联的行为根据引发变迁的接口事件组成成员函数的主体。当接口事件被识别并被发送到协议实体, 相应的对象成员函数被调用。当状态变迁被激发时, 一个对象变成代表另一状态的对象。

我们设计了一种让设计者能以图形方式修改状态机并且自动生成 C+ + 对象类和它们相关的成员函数的工具。这一工具实际上是具有图示化修改能力的程序生成器。对象类和成员函数以框架形式形成, 用户可以把事先写好的例程加入到成员函数中。目前内含的例程是手写的。但是它们可以用规范的编译器生成。扩充性实现方法允许我们的工具也支持用户可修改 FSM, 增加新的状态和变迁, 提供的工具会生成一组新的对象类和成员函

1997-07-16 收稿

肖 斐: 博士生, 研究方向为计算机网络与通讯, 分布式系统。 朱崇湘: 硕士生, 研究方向为软件工程。
孙永学: 硕士生, 研究方向为软件工程。 赵文耘: 复旦大学计算机科学系副教授, 主要从事软件工程方面研究。

数。同样,亦能处理状态和变迁的删除。为了指明更详细的操作,一个状态被分解成多个子状态。此时,相应的状态对象将会被重新定义为一组子对象。子状态对象组成了原对象子类的对象。设计者可以用工具打开一个状态,然后把它重新定义成子状态和局部接口事件。子类的定义也能用工具自动生成。

2 协议对象的设计和实现

在本节中,我们将展示如何用对象来实现一个协议软件,同时提出一个让用户以图形方式修改状态机和自动生成 C++ 对象类的工具。协议由 FSM 说明,然后 FSM 由一组相关的对象实现。我们将引用实际的协议 CCITTT.70 的简化版本来表明上述概念。

2.1 对象设计

某一特定层的操作可以依据扩展的有限状态形式化地说明。协议规范包括:

1) 描述有限状态机的一组可能的状态和变迁;

2) 一组精细定义的接口事件,分别对应于层交界处的服务原语;

3) 由程序段定义的行为在状态变迁时执行。

在我们的方法中,协议说明之后进行对象设计。对象设计分成以下 3 个步骤:

1) 标识对象。FSM 的每一个状态定义由对象来实现,这些对象称为状态对象。

2) 定义对象的成员函数。成员函数是激发状态变迁的接口事件。

3) 完成对象的成员函数。成员函数由被相关接口事件激发的状态变迁所关联的行为构成,变迁的激活状态也是成员函数的一部分。

目前主要的面向对象语言,如 C++,Smalltalk,要求每个对象是一个类的实例。这样,一个状态必须定义成某个类的实例。

当 FSM 被初始化,一个初始状态对象实例被创建。当状态变迁激发,from_state 实例变成 to_state 实例,状态变迁由接口事件激活,接口事件可能是被上层实体调用的本层服务原语,可能是被下层实体指明的下层服务原语,也可能是系统产生的事件。比如:超时。这些接口事件均为状态对象的成员函数。若状态不支持执行某一特定的接口事件,它的状态对象或忽略该事件或标识该事件为意外。状态对象类的声明如图 1 所示。

```
class pm_14{        // This is a transport layer object
  public:
    pm_14( ) { }    // Instance creation
    // Interface events of this layer
    virtual void t_conn_req(pud*, pm*){}
    virtual void t_conn_ind(pud*, pm*){}
    virtual void t_conn_resp(pud*, pm*){}
    virtual void t_conn_conf(pud*, pm*){}
    virtual void t_data_req(pud*, pm*){}
    virtual void t_data_ind(pud*, pm*){}
    virtual void t_disc_req(pud*, pm*){}
    virtual void t_disc_ind(pud*, pm*){}
    // Interface events from lower layer entity
    virtual void n_conn_req(pud*, pm*){}
    virtual void n_conn_ind(pud*, pm*){}
    virtual void n_conn_resp(pud*, pm*){}
    virtual void n_conn_conf(pud*, pm*){}
    virtual void n_data_ind(pud*, pm*){}
    virtual void n_disc_req(pud*, pm*){}
};
```

图 *1* 状态对象类的声明

由规范声明层实体映射到实现层实体的设计步骤归纳如下:

1) 为由 FSM 建模的协议设计对象类。FSM 的每一个状态对象对应一个对象类。

2) 为每个对象类设计成员函数。由于每个接口事件触发一个特定状态的变迁, 所以把它定义成对象类的成员函数, 用以实现该状态。

3) 状态变迁动作对应函数体。当状态变迁条件满足时, 一状态变成另一状态。

2.2 一个例子

此处我们显示如何根据 FSM 的说明实现一个协议。采用的例子是 TP0 远输协议的发送者 T.70, 它是 G4FAX 的一部分。图 2 所示的 FSM 是一个简化版本, 省略了一些协议的细节。下一节介绍工具时将采用完整的 T.70。同样, FSM 只显示了点对点的协议状态和操作。

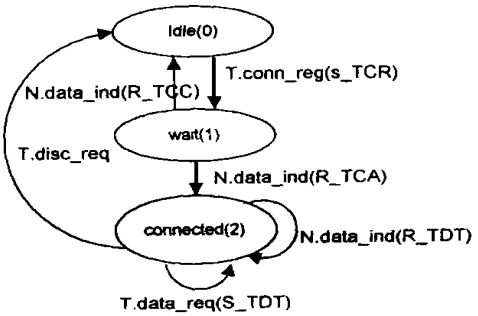


图 *2* 减化的 T.*70 FSM* 模块图

在我们目前的实现中, 如图 3 所示。完整的 FSM 定义为它组成状态的基类。使 FSM 的每一个状态对象对于上层实体或下层实体有一致的接口。协议实体可以只须询问一个对象 Pm_14, 而不是一组对象集。同时, Pm_14 可以作为对于其它状态对象通用的发送者。例如, 当 wait_1 对象的成员函数 N_data_ind 被调用时, Pm_14 的过程被请求, 因为 N_data_ind 在 Pn_14 中定义。该过程首先决定到达的信息是哪一类 TPDU, 然后它调用相应的 Wait 中的成员函数。例如, 当到达的 TPDU 是 TCA(Transport Connection Accept), 于是 wait_1 的 R_TA 被调用。完整的 FSM 定义为基类的另一优点是利于增长式的实现, 这会在后面说明。缺省操作可以在对象状态类重定义。如果状态不支持某一接口事件的执行, 将忽略它或标识为意外。

```
class Idle_0: public pm_14{
    // Declared as subclass of pm_14
    // This will be explained later
public:
    Idle_0(): pm_14(){}
    // Instance creation
    void t_conn_req(pdu*, pm_14*)
    // defined the function in superclass
      pm_14
    { …// Send TCR and becomes a wait_1
        object
    }
};
class wait_1: public pm_14{
public:
    wait_1(): pm_14()   {}
    // Instance creation
    void R_TCA()
    { …// Receives a positive response
    // T_conn_conf to upper layer
    // Becomes a connected_2 object
    }
    void R_TCC()
    { …// Becomes an Idle_0 object
    }
class connected_2: public pm_14{
public:
    connected_2(): pm_14(){}
        // Instance creation
    void t_data_req(pdu*, pm_14*)
    { …// Send TDT
    }
    void t_disc_req(pdu*, pm_14*)
    { …// Becomes an Idle_0 object
    }
    void R_TDT(pdu*, pm_14*)
```

```
{ …// Data_ind to upper layer entity                    }
 …// at the end of re_assembly                        };
```

图 3　T.70FSM 状态对象的类定义

3　设计工具

本节我们提出一个辅助设计者实现用 FSM 说明协议的软件工具, 如图 4 所示。状态机编辑器是一个图形方式编辑器, 允许设计者在屏幕上直接修改状态和接口事件。它记录状态机的状态和接口事件, 生成 C+ + 的对象类和它们的成员函数。图 5 是由 FSM 建模的 T. 70 发送协议和它的类定义。FSM 是用状态机编辑器编辑的, 自动生成对象类, 同时生成了成员函数声明。设计者可用工具加入手写子程序到成员函数。这些子程序可用规范编译器生成。设计者可用状态机编辑器分解一个状态, 增加更多的细节, 如子状态和其他状态接口函数。

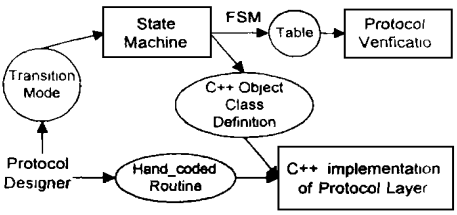
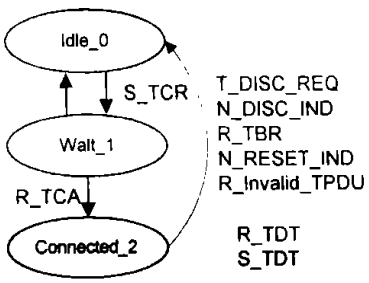


图 4　用软件工具实现协议

```
R_n_TPDU
    R_TCC
    R_TCR
    R_TBR
    R_Invalid_TPDU
    N_RESET_IND
    N_DISC_IND
    I_DISC_REQ
    timeout
class Idle_0: public T70 {
public:
  Idle_0(): T70() {}
  void S_TCR() {}
  void R_any_TPDU() {}
  ~ Idle_0() {}
};

class Wait_1: public T70 {
public:
  Wait_1(): T70() {}
  void R_TCC() {}
  void R_TCR() {}
  void R_TBR() {}
  void R_Invalid_TPDU() {}
```




```
  void N_RESET_IND() {}
  void T1_1timeout() {}
  void N_DISC_IND() {}
  void T_DISC_REQ() {}
  void R_TCA() {}
  ~ Wait_1() {}
};
class Connected_2: public T70 {
  Connected_2(): T70() {}
void T_DISC_REQ() {}
void N_DISC_IND() {}
void R_TBR() {}
void N_RESET_IND() {}
void R_TDT() {}
void S_TDT() {}
```

```
void R_Invalid_TPDU() {}
~ Connected_2() {}
```

图 5　描述 T.70 发送协议的 *FSM* 以及用工具产生的 C++ 对象类

用户可用状态机编辑器打开一个状态, 用以增加细节, 如子状态和其他接口事件。

4　增长式的实现和继承

这节中我们讨论如何增长式地实现一个协议。增长式实现协议是指通过增减 FSM 的接口事件与修改已有协议的模型, 通过分解用某些只对子状态可见的接口函数分解状态成子状态来重新定义状态。前者可以用状态编辑直接处理, 如前一节所示, 后者也可用状态机编辑器处理, 如图 6 所示。

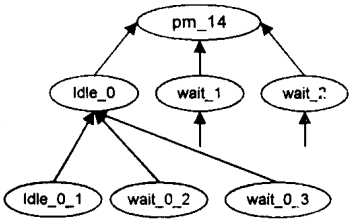


图 6　协议机的类层次

4.1　方法和原则

在 ISO 或 CCITT 推荐书中, 协议限定了 3 种不同类型的行为: 点对点, 层内和局部行为。协议规范分成 2 个层次。协议层包含点对点的行为, 细节层增加层内和局部行为。在图 2 中所示的 FSM 只包含了点对点行为。

在我们的方法中, 协议的软件实现可以分解成 3 个层次: 功能层, 抽象的状态机层和细节层。每一层被视为实现的一个阶段。在功能层, 协议机被看做响应接口事件需求完成功能的黑盒子。这一层将会协议的服务规范。在抽象的状态机层, 它符合 OSI 框架中的协议层, 点对点的行为如图 2 的简单的 FSM 表示, 细节层与 OSI 框架结构一致。

为了扩充式地实现协议, 我们可以从功能层开始, 然后进行到抽象状态层。最后到达细节层。在功能层, 协议只有一个状态, 即整个协议机, 它可以用一个对象类实现, 所有的服务原语作为它的成员函数, 功能层协议类的定义如图 1 所示。

当从功能到抽象状态层, 协议机的唯一状态被分解, 如图所示。亦有了个状态。这样抽象状态机可以被 3 个对象类实现, 分别对立于一个状态, 而且每个对象类定义为实现功能层协议机对象类的子类。

现在, 我们得出一个分解状态的的原则。

原则 **1**: 如果由对象 O 实现的状态 X 被分解为 X1, X2 ……Xn, 它们分别被对象 O1, O2, ……O3 实现, 则 O1, O2 ……On 是 O 的子类。

从抽象状态机层到细节层的协议实现可以用相似的方法处理。在细节层引起更多的状态和增加更加层内与局部的行为。这样新增状态的对象类作为分解状态对象类的子类。

实际上, 状态有时需要合并。下面是状态合并的原则。

原则 **2**: 若一组状态 X1, X2, …, Xn 分解由对象类 X1, X2, …, Xn 实现, X1, X2, …, Xn 合并成一由对象类 O 实现的状态 X, 则 O1, O2, …, On 是 O 的子类。

通常, 功能层对象定义了对应接口事件的缺省操作, 它们位于构成整个协议机通用接口的成员函数中。因为子类对象可继承它基类中的成员函数, 抽象机状态的对象可以重用在功能层对象中定义的成员函数。然而, 抽象机状态的对象为了适合自身的操作需要亦可重定义成员函数。子类中被重定义函数将会重载在基类中的定义。例如, 成员函数 t_conn_

req 在类 idle0 中的定义(图 3), 将会重载在类 pm_14 中的定义, pm_14 是 idle0 的基类。细节层的对象也有同样性质。这种方法的优点在于, 当从功能层到抽象状态机层, 再到细节进行实现时, 已经完成的部分不必全部更改。可通过子类重用或重载。

4.2 例子

图6 所示是状态 Idle 的详细设计, 是由状态机编辑器生成的。我们有 3 个子类, Idle_0_1, Idle_0_2, 与 wait_0_3。为了细化一个状态, 设计者用状态机编辑器从屏幕上打开 FSM 的一个状态, 按需要增加状态和事件。类的继承关系和定义如图 6, 图 7 所示。子类自动生成。

```
class Idle_0_1: public Idle_0 {
public:
   Idle_0_1(): Idle_0() {}
   void T_CONN_REQ() {}
   ~ Idle_0_1() {}
};
class Wait_0_2: public Idle_0 {
public:
   Wait_0_2(): Idle_0() {}
   void N_DISC_IND() {}
   void S_TCR() {}
   void T_DISC_REQ() {}
   ~ Wait_0_2() {}
}
class Wait_0_3: public Idle_0 {
public:
   Wait_0_3(): Idle_0() {}
   void N_DISC_IND() {}
   void To_3timeout() {}
   void N_RESET_IND() {}
   void R_any_TPDU() {}
   void T_DISC_REQ() {}
   ~ Wait_0_3() {}
};
```

图 7 *Idle_0* 的子类定义

5 结束语

一般一个协议可以按照下列方法实现和访问: 1) 作为操作系统的一部分, 并通过系统调用来询问; 2) 作为用户进程, 通过内部进程通讯来访问; 3) 作为与用户进程的不同过程, 通过过程调用访问。正如前文所述, 内部进程通讯会带来性能的惩罚, 所以第 2 种方法应排除。在第 1 种方法中, 存在全局的上下文切换。由于在系统与用户之间位于不同的地址空间, 这种全局的切换是不可避免的。设计者决定在协议找中何处是系统与用户的边界。因为我们考虑作为用户过程的协议实现, 所以系统采用第 3 种方法。

在我们的方法中, 发生状态变迁时, 一个对象变成另一个对象, 在大多数现有的语言中, 为实现对象转换。必须首先生成 to_state 对象, 然后删除 from_state 对象, 生成与删除对象体数量通常是耗时的操作。目前, 对象转换操作如下进行, 当开始会话时, 生成一组对象, 分别对应整个状态机的一个状态。这样, 当状态变迁时, 不再进行对象的生成和删除, 当对象指针指向 to_state 对象, 该对象将处理即将到来的事件和消息。

参 考 文 献

1 曾华桑等译. 计算机网络(第 2 版). 成都出版社, 1989

2 张根度等. 开放系统互连标准. 电子科学技术编辑部, 1987

3 Myers J., Rose M. Post Office Protocol- Version 3. RFC 1725, DS, 1994

An Object- Oriented- Based Approach to Protocol Software Implementation

Xiao Wen Zhu Chongxiang Sun Yongxue Zhao Wenyun
(Fudan University, Shanghai, 200433)

ABSTRACT In this paper, an object- oriented- based approach to protocol software implementation is presented. A protocol is specified by an FSM, and the FSM is implemented by a group of related object. In our method, each state is implemented by an object. The member functions of an object are the interface events that trigger state transitions, and actions associated with state transitions constitute the body of the member functions. An object becomes another object if a state transition is enabled. A real example is given for illustration. We also present a software tool that lets a designer edit a state machine graphically, and generates C++ class definitions automatically.

KEY WORDS FSM Class Object Inheritance Override

(上接第 7 页)

Modeling and Simulation of Space Robots

Wu Wei Hong Bingrong
(Depart. of Comp. Sci. and Engin. Harbin Institute of Technology, 150001)

ABSTRACT Space robots are expected to accomplish various missions instead of astronauts in the future space investigation. Modeling and simulation of space robot are an important part in designing and developing space robots. This paper discusses the methods of modeling of space robots in detail, including building geometric and kinematics models, the modules and functions of the space robot simulation system. And finally, we give the simulation results which show that the system is reliable and can work in real- time environment

KEY WORDS Space stations Robots Simulation Modeling