ISSN 1000-9825, CODEN RUXUEW
Journal of Software, Vol.19, No.10, October 2008, pp.2539−2549
DOI: 10.3724/SP.J.1001.2008.02539

E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel/Fax: +86-10-62562563

基于设计演算的形式化用例分析建模框架*

陈 鑫[1,2+], 李宣东[1,2]

[1](南京大学 计算机科学与技术系,江苏 南京 210093)

[2](计算机软件新技术国家重点实验室(南京大学),江苏 南京 210093)

Design Calculus Based Approach to Modeling Use Case

CHEN Xin[1,2+], LI Xuan-Dong[1,2]

[1](Department of Computer Science and Technology, Nanjing University, Nanjing 210093, China)

[2](State Key Laboratory for Novel Software Technology (Nanjing University), Nanjing 210093, China)

+ Corresponding author: E-mail: chenxin@seg.nju.edu.cn

Chen X, Li XD. Design calculus based approach to modeling use case. *Journal of Software*, 2008, 19(10):2539−2549. http://www.jos.org.cn/1000-9825/19/2539.htm

Abstract: This paper proposes a formal approach to modeling use case which captures requirements from multi-angle views: The class diagrams, the use case sequence diagrams, the use case state diagrams, the specification mapping and the system invariant. By defining formal semantics of those views, each aspect of requirements is given exact formal descriptions. As a result, integrated specification of one method can be built by integrating formal descriptions of its interaction specification and its functional specification. At the same time, properties of use case models can be specified and analyzed through the proof in design calculus. As an application, rules for checking the consistence of use case models are studied. An example to illustrate the feasibility of the proposed method is given.

Key words: use case model; semantics; multi-view; consistency checking; weakest precondition

摘 要: 提出一种形式化用例分析建模框架,引入类图、用例顺序图、用例状态图、功能规约函数和系统不变式从多个角度为需求建模.通过定义这些视图的形式化语义,为需求的各个方面定义了准确的形式化描述.利用该框架,可以从方法的交互行为规约和功能规约合成描述方法全部行为的全规约;也可以定义用例模型的性质,并通过设计演算中的证明来分析验证这些性质.作为应用,研究了检查用例模型一致性的规则.给出一个实例说明建模框架的可行性.

关键词: 用例模型;语义;多视图;一致性检验;最弱前提条件

中图法分类号: TP311 **文献标识码**: A

如何准确地描述系统的需求,是软件开发方法研究的关键问题之一.基于 UML 的软件开发过程,广泛地使用用例模型描述目标系统的需求规约.经典用例模型包含两个主要部分[1−3]——问题域概念模型和用例.问题域

* Supported by the National Natural Science Foundation of China under Grant Nos.60425204, 60721002 (国家自然科学基金); the National Basic Research Program of China under Grant No.2002CB312001 (国家重大基础研究发展计划(973)); the National High-Tech Research and Development Plan of China under Grant No.2007AA010302 (国家高技术研究发展计划(863)); the Natural Science Foundation of Jiangsu Province of China under Grant No.BK2007714 (江苏省自然科学基金)

Received 2007-06-26; Accepted 2007-12-24

概念模型着重描述问题领域中存在的概念和概念间的关联关系,常常用概念类图来表示.用例是描述需求的主体部分,它通常使用用例图和自然语言从多个方面描述系统的需求.UML 标准并不包含类图、用例图等标准视图的形式化定义,因而借助用例模型定义的需求规约缺乏明确的形式语义,存在着模糊和不明确之处.通过用例模型定义的需求规约是后续软件开发活动的主要依据,任何不准确的描述都可能导致系统的实现出现错误.因此,如何向用例模型中引入形式化的描述方法,从而可以准确地描述系统的需求规约,并应用形式化的分析和验证技术分析用例模型的性质,成为一个亟待解决的问题.

设计演算[4,5]是一种基于指称语义的演算体系.它使用设计(design)来描述研究对象是否正常启动与终止,以及启动时需要满足的前置条件和正常终止时需满足的后置条件.这种建模方法只考虑被研究对象启动前和结束后的状态变化,不考虑中间的活动过程,非常适合描述规约.

本文提出了一种形式化用例分析建模框架,它引入类图、用例顺序图、用例状态图、功能规约函数和系统不变式从多个角度为需求建模.基于设计演算理论,此框架定义了各类视图的形式化语义,从而为通过这些视图描述的需求的各个方面赋予了准确的形式化定义.利用该框架,可以从方法的交互行为规约和功能规约合成描述方法全部行为的全规约;也可以定义用例模型的性质,并通过设计演算中的证明来分析验证这些性质.用例模型视图间的一致性是用例模型正确性的基本要求,本文在上述形式框架中研究了用例模型的一致性检查规则.静态一致性把类图作为用例模型的类型定义部分,它检查各视图中对类、类关联、类方法和类属性的使用是否与类图中的类型定义产生冲突.动态一致性检测同一用例各个视图分别描述的功能和动态行为规约是否有冲突.用例模型动态一致性的检验可以通过基于设计演算的证明过程来完成.

本文第 1 节介绍设计演算的有关理论.第 2 节引入本文的用例模型.第 3 节给出用例模型各视图的形式化定义.第 4 节研究用例模型视图间的一致性并给出一致性检验的方法.第 5 节比较相关的工作.第 6 节进行总结并对未来的工作进行展望.此外,一个关于零售终端的实例研究贯穿全文,以说明本文形式化途径的可用性.

1 设计演算

设计演算[4,5]是一种基于指称语义的演算体系.设计不仅定义被研究对象成功启动时需要满足的前置条件和正常终止时满足的后置条件,也包含了正常启动与终止的描述.设计既可以用来定义规约,同时也可以定义各种命令的语义.文献[4]利用设计演算定义了一种类 Java 程序设计语言的语义.文献[5]用设计研究了几种进程代数的语义.设计演算包含了程序构造算子作用于设计的运算规则、设计的最弱前提条件的计算规则以及设计的精化方法等演算方法.本节介绍用例模型形式化的数学系统——设计演算方法,下面的定义及定理大都来源于文献[4,5].

定义 1(设计 design). 一个定义在输入变量集合 $in\alpha$ 和输出变量集合 $out\alpha$ 上的设计是一个谓词,形如 $p(in\alpha)\vdash R(in\alpha,out\alpha')$,它被定义为 $(p\wedge ok)\Rightarrow(R\wedge ok')$.这里,$p(in\alpha)$ 是一个关于输入变量集合 $in\alpha$ 的谓词,$R(in\alpha,out\alpha')$ 是一个关于输入变量集合 $in\alpha$ 和输出变量集合 $out\alpha'$ 的谓词.一个设计规定,如果该设计被成功启动,即 $ok=\text{true}$,并且初始状态满足前置条件 p,则其一定会终止,即 $ok'=\text{true}$,且终止状态满足后置条 件 R.

为区分输出变量的起始和终止状态,对属于输出变量集合 $out\alpha$ 的变量,添加上标′来表示其终止状态.辅助布尔变量 ok 和 ok'的值描述设计的正常启动和终止,它不能被任何程序命令所操纵,无论是表达式还是赋值语句都不能改变它的值.

文献[5]中指出,设计既可以作为定义规约语言的数学对象,也可以定义程序设计语言中各种命令的语义.文献[4,5]中同时证明了,设计对于程序构造算子是封闭的.我们以定理的形式给出它们的演算方法.

定理 1. 设计对于程序构造算子是封闭的.这里,$Cmd_1\sqcap Cmd_2$ 表示非确定选择结构,程序即可以选择执行 Cmd_1,也可以选择执行 Cmd_2,且选择的过程是程序外部无法干预的.$Cmd_1\lhd b\rhd Cmd_2$ 表示条件选择结构,条件 b 满足则执行 Cmd_1,否则执行 Cmd_2.$Cmd_1;Cmd_2$ 表示顺序组合结构,先执行 Cmd_1,并以 Cmd_1 的结束状态为起始状态,执行 Cmd_2.

$$(p_1\vdash R_1)\sqcap(p_2\vdash R_2)\equiv(p_1\wedge p_2)\vdash(R_1\vee R_2),$$

$$(p_1 \vdash R_1) \lhd b \rhd (p_2 \vdash R_2) \equiv (p_1 \lhd b \rhd p_2) \vdash (R_1 \lhd b \rhd R_2),$$
$$(p_1 \vdash R_1);(p_2 \vdash R_2) \equiv (p_1 \wedge \neg(R_1;\neg p_2)) \vdash (R_1;R_2),$$

其中,谓词的顺序组合 $P(s');Q(s)$ 定义为 $\exists m \bullet P(m) \wedge Q(m)$.

考虑几个特殊的设计.最强的设计 false,记作 $\top$,最弱的设计 true,记作 $\bot$.设计 *skip* 是这样的一个设计: $skip \overset{\text{def}}{=} \text{true} \vdash \wedge_{x \in \alpha} x' = x$,它保持字符表中所有变量的值不变.一个卫式条件是定义在变量集合 $in\alpha$ 上的一个谓词,记作 $g_\top$,并且有如下的定义: $g_\top \overset{\text{def}}{=} skip \lhd g \rhd \text{false}$,它表示如果卫式条件得到满足,则将遵照设计 *skip* 定义的行为,否则进入死锁.

最弱前置条件的演算方法是程序分析的经典技术.可以将设计与最弱前提条件用下面的定义连接起来.

定义 2(设计的最弱前提条件).

$$wp(p \vdash R, q) \overset{\text{def}}{=} p \wedge \neg(R;\neg q).$$

下面的定理给出了在程序构造算子的作用下,计算最弱前提条件的规则.

定理 2. 使用程序构造算子连接的设计的最弱前提条件可以由如下的规则得到,这里 $f(v)$是一元函数,b 是布尔表达式,q 是一个谓词, $D_1 = p_1 \vdash R_1$ 和 $D_2 = p_2 \vdash R_2$ 是设计, $q[f(v)/v]$ 表示用 $f(v)$替换变量 v 在谓词 q 中所有的自由出现.

$$wp(\text{true} \vdash v' = f(v), q(v)) = q[f(v)/v],$$
$$wp(D_1 \vee D_2, q) = wp(D_1, q) \wedge wp(D_2, q),$$
$$wp(D_1 \lhd b \rhd D_2, q) = wp(D_1, q) \lhd b \rhd wp(D_2, q),$$
$$wp(D_1; D_2, q) = wp(D_1, wp(D_2, q)).$$

2 用例分析建模框架

基于 UML 的软件开发过程通过用例模型从多个不同的角度分析目标系统的需求[3].首先是识别问题领域中的概念,明确概念之间的相互关联关系,并以此为基础构造问题领域的概念模型.概念模型通常表示为概念类图.接着,用例逐个描述系统的需求.通常用例从这样的几个方面来描述需求:系统应提供哪些方法?这些方法的名称和输入输出是什么?每种方法的功能是什么?系统的这些方法按照怎样的顺序和用户进行交互?系统必须遵守哪些业务规则和约束[2,3].据此,本文的用例分析建模框架引入了类图、用例序列图、用例状态图、功能规约函数和系统不变式分别对需求的这些方面进行建模.类图中的概念类图表示问题域的概念模型;类图中的用例控制类定义了用例操作的对象和用例方法的型构;用例序列图和用例状态图分别用序列和控制状态转换的方式描述用例与环境进行交互的场景;功能规约函数定义了方法的功能规约;系统不变式刻画了目标领域的业务规则和相关约束.

通过用例模型为目标系统建立需求规约是一个由局部分析到整体集成的过程[5].局部分析过程逐个分析系统的各个需求以及每个需求所包含的各个方面.在整体集成过程中,关于同一需求的各个方面被整合起来,形成关于该需求的一个完整描述.例如:通常,用例对系统提供的方法的行为是从两个方面进行描述的.交互行为规约描述系统方法与用户进行交互时先后发生的顺序;功能规约描述每种方法的功能.要描述每种方法的完整行为,就必须整合这两个方面的描述.本文中,用例分析建模框架引入用例序列图、用例状态图和功能规约分别描述方法行为的这两个方面,并通过集成的方法,整合成关于方法行为的完整描述——全规约.

本文将用例分析建模框架 M 定义为一个五元组 $(\Gamma, \Delta, \Omega, \Phi, \theta)$,其中:

- Γ是一个类图.类图包含了概念类图和用例控制类图两个部分.概念类图描述概念及概念间的关联关系.概念类只定义属性而不定义方法,概念间的关联关系也是通过定义概念类来描述的.每个用例控制类对应于一个用例,用例控制类需定义属性和方法,类属性定义用例操作的对象,方法定义用例可与外部进行交互的操作.
- Δ是一族序列图的集合.每个用例对应于该族中的一个集合,此集合中的每个序列图描述了该用例允

许的系统和环境之间交互的一个场景.这里,交互场景被定义为一个事件的序列,序列中的每个事件对应着对用例控制类方法的一个调用.

- Ω是一个状态图的集合,集合中的每个元素描述了一个用例内部的控制状态.为描述控制状态的信息,为每个用例控制类增加一个特殊的变量 *state*,并约定变量 *state* 的值为当前的控制状态.
- Φ是一个映射,它从用例控制类声明的每种方法映射到该方法的功能规约说明.规定,对于型如 $C::m(T_1\ in\ T_2\ out)$的一种方法,它的功能被定义为一对前置条件和后置条件,并表示成这样的形式: $C::m(T_1\ in\ T_2\ out)\{pre_m \vdash post_m\}$.这里,$C$ 是用例控制类的名字,m 是方法名,T_1,T_2 是类型名,*in*,*out* 分别是输入和输出参数.
- θ是系统的不变式,它是一个关于类、类的属性、类的方法的逻辑表达式,记录了系统必须要满足的约束条件.

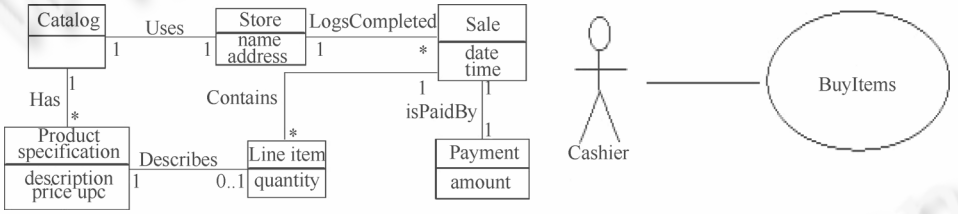
下面用一个实例来说明经典用例模型描述的需求如何通过本文的用例分析建模框架来定义的.

例 1:图 1 描述了一个零售系统中零售商品业务的需求.

经典用例模型将此需求描述为:

问题域概念模型:每个商店都保存一份产品的目录(catalog),目录里包含了对其销售的所有产品的规格说明(product specification).商店与顾客的每一笔交易(sale)都被记录下来,每一笔交易包含交易产品的名称和数量信息(line item),以及最后付款(payment)的记录.图 1(a)中用概念类图表示了此概念模型.

用例 BuyItems-Controller 描述了发生一笔交易的操作过程:收银员需要逐笔输入顾客购买的每项产品(EnterItem),系统随即生成一个新的交易,逐项记录每项产品的数量,并计算总价.收银员调用操作 endSale 通知系统产品输入结束,最后调用操作 makePayment 付款找零结束本次交易.只有产品目录里有记录的产品才能被出售.用例图如图 1(b)所示.



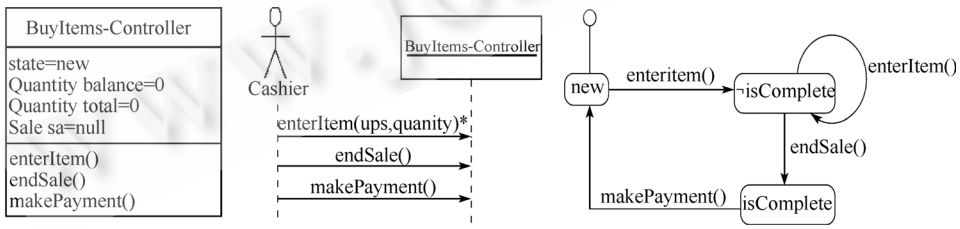
(a) Conceptual class diagram (b) Use case diagram

(a) 概念类图 (b) 用例图

Fig.1 Use case model of a point-of-sale system

图 1 一个零售系统的用例模型

本文的用例分析建模框架,通过用例控制类图、用例序列图、用例状态图对用例的各个方面分别加以描述.如图 2 所示.



(a) Use case controller class diagram (b) Use case sequence diagram (c) Use case state diagram

(a) 用例控制类图 (b) 用例序列图 (c) 用例状态图

Fig.2 Three views illustrating the use case BuyItems

图 2 说明用例 BuyItems 的 3 个视图

3 用例分析建模框架的形式化定义

本节研究用例分析建模框架 M 中各个视图的形式化定义.

3.1 类 图

类图Γ是整个用例模型的类型定义部分,它同时描述了系统的静态体系结构.关注类图中以下几个部分:

cname 是一个集合,包含了Γ中定义的所有类的名字.

aname 是一个集合,包含了Γ中定义的所有关联的名字.在概念类图中,关联关系被定义成关联类,每个关联类的定义包含了两个属性的声明,各自表示此关联中的角色.

superclass 是一个函数,*superclass*(C)返回概念类图中类 C 的直接父类.把概念类图上的一般继承关系 $\succ$ 定义为直接继承关系的传递闭包,并有 $N \succeq M$ iff $N \succ M \vee N = M$.

attr 是一个函数,对于类名 $C \in cname \cup aname$, 函数 *attr*(C) 返回类 C 中声明的属性的集合.用 *dtype*(a) 表示属性 a 的类型.用函数 *Attr*(C)表示对函数 *attr*(C)的扩展,它返回的集合还包含了类 C 继承的公有和保护属性的声明.

meth 是一个函数,对于类名 $C \in cname \cup aname$, 函数 *meth*(C)返回类 C 中声明的方法的集合.集合中的每个元素是一个型如 $m(T_1\ x_1,...,T_k\ x_k)$ 的方法型构声明,其中 m 是方法名,T_i 和 x_i 分别代表参数的类型和名称.

例 2:对于例 1 中的类图,有:

$$cname = \{Store, Sale, Catalog, ProductSpecification, LineItem, Payment\},$$
$$aname = \{Has, Uses, Describes, Contains, LogCompleted, isPaidBy\},$$
$$meth(BuyItems - Controller) = \{enterItem(int\ upc, Quantity\ qty), endSale(), makePayment()\}.$$

关系 *superclass* 为空,函数 *attr* 可以很容易被定义出来.

3.2 对象图以及面向对象的命令作为设计

考虑基本数据类型集合 T 和对象引用的无穷集合 R,并令空引用 $null \in R$.一个值是一个基本类型或者是一个引用.一个对象 o 是一个空引用 *null* 或者是一个三元组 $\langle r, C, \sigma \rangle$,其中引用 r,记作 *ref*(o),属于集合 R;类名 C,记作 *type*(o) ; σ 是对象 o 的状态,记作 *state*(o),它为每个属性 $a \in Attr(C)$ 赋予一个类型为 *dtype*(a)的值,该值要么属于基本数据类型集合,要么是一个对象的引用,或者是空引用.使用符号 $o.a$ 表示 a 的值 $\sigma(a)$.

文献[4]使用设计来定义面向对象语言中命令的语义.这里考虑两个命令的语义:修改对象属性的值和对象的创建.需要引入新的状态变量Π来定义系统当前的状态,即系统当前的堆.定义函数 $\Pi(C)$ 返回当前在堆Π存在的所有类型为 C 的对象.

令变量 le 的类型是一个类,它要么是一个简单变量,其类型名 C 满足 $C \in cname \cup aname$, 要么是某个类变量的属性 $le_1.b$. 属性赋值语句 $le.a := e$ 改变系统堆上由引用 le 指向的对象 o 的属性 a 的值为表达式 e 的计算结果,如果此赋值语句是良定义的,则它的形式化定义如下:

$$le.a := e \overset{\text{def}}{=} D(le.a := e) \wedge a \in Attr(type(le)) \vdash \begin{pmatrix} \prod(dtype(le))' = \prod(dtype(le)) \wedge \\ \{o \oplus \{a \mapsto value(e) \mid o \in \prod \wedge ref(o) = le\}\} \end{pmatrix}.$$

这里,用 $D(le.a := e)$ 表示表达式 $le.a := e$ 是良定义的,它要求 $le.a$ 和 e 都是良定义的,而且它们的类型相容[1].

对象创建命令 $C.new(le)$ 是良定义的,如果 $C \in cname \wedge D(le) \wedge dtype(le) \succeq C$. 此命令执行的结果是创建一个新的对象,使引用 le 指向该对象,并为该对象的所有属性赋初始值.这一系列动作可以形式化地定义为

$[\![C.new(le)]\!] \overset{\text{def}}{=} D(C.new(le)) \vdash \exists r \notin ref(\prod) \bullet (AddNew(C, r) \wedge Modify(le))$,其中:

$$AddNew(C, r) \overset{\text{def}}{=} \prod(C)' = \prod(C) \cup \{\langle r, C, \{a_i \mapsto init(C.a_i) \mid a_i \in Attr(C)\}\rangle\},$$
$$Modify(le) \overset{\text{def}}{=} le' = r.$$

对于属性的初始化,若 $dtype(C.a_i) = M$, 且 M 是一个类名,则 $a_i \mapsto init(C.a_i) \overset{\text{def}}{=} M.new(C.a_i)$.

按上述方法形式化后的类图Γ,构成了此用例模型的类型定义部分.类图中关于概念类、概念类关联关系和用例控制类的定义,构成了整个用例模型的类型定义部分.它明确了概念类的类型、概念类属性的类型、表示关联关系的关联类的类型、关联类属性的类型、用例控制类属性的类型以及用例控制类方法的型构.其余视图的形式化描述,需要对照类图定义的类型,进行类型检查,以确定没有和类型相关的错误.

3.3 用例序列图

用例序列图描述了系统与环境进行交互的过程,此过程中,环境通过调用系统提供的方法与系统进行交互.如前所述,每个用例被定义成一个用例控制类,与该用例相关的操作成为类的方法.这样,每个业务流程就表示为基于这些方法的一个调用序列.据此,定义一个序列图的语义是对前缀封闭的有限长序列的集合,序列的每个事件是对用例 *controller* 定义的方法 *m* 的一个调用 $?controller.m(u;v)$.其中,u,v 各自代表输入输出参数序列.集合中的每个元素对应该用例一个合法执行的前缀.称这个集合是该用例的交互协议.

例 3:对于例 1 中描述的系统序列图,有:

$$Tr(BuyItems)=?enterItem()^{*}+?enterItem()^{+};?endsale();?enterItem()^{+};?endsale();?makepayment().$$

这里用符号 ; 连接同一序列中的相邻事件,符号+分隔各个事件序列.

对于一个用例允许多个交互序列的情形,把该用例的交互协议定义为所有这些集合的并集.

用例之间的包含关系允许用例之间共享相同的部分.可以使用函数调用来定义用例间的包含关系.约定,如果用例 U_1 包含了用例 U_2,那么环境仍然只和用例 U_1 交互,用例 U_1 负责收到请求后再调用用例 U_2 提供的方法,并且只有在用例 U_1 对用例 U_2 的调用返回以后,用例 U_1 才继续响应环境其他调用.

3.4 用例状态图

一个用例的状态图 $S_C \overset{\text{def}}{=}(S,s_0,\varsigma)$ 包含了控制状态集合 S,一个初始状态 $s_0\in S$ 和一个跃迁关系 $\varsigma\in S\times Label\times S$. 一个跃迁 $t\in\varsigma$ 从控制状态 s 转移到控制状态 s' 被标识为一个三元组 $l=\langle m(in;out),g,a\rangle\in Label$,其中 $m()$是一个触发事件,表示环境对用例提供操作的一个调用,a 是由事件 m 触发的行为,a 可以是一个 OO 命令,如赋值、创建新对象、函数调用等,也可以是用设计定义的形式化规约.g 是关于用例控制类属性和 m 输入参数的谓词.把跃迁 t 记作 $s\xrightarrow{l}s'$.

本文约定,用例状态图所有的跃迁都是由对其方法的调用引起的,方法开始执行以后不会被内部的事件所打断.如图 1(d)所示,对于用例控制类 *BuyItems-controller*, 只关心外部调用事件和这些事件所引起的控制状态跃迁.

为了定义控制状态跃迁,为每一个用例控制类引入表示控制状态的变量 *state*.这样,跃迁 $t=s\xrightarrow{l}s'$ 可以被定义成一个卫式设计: $cs(t)\overset{\text{def}}{=}(state=s)_{\top};(state'=s')$.

对于每一个触发事件 $m()$,考虑一个跃迁的集合 $\varepsilon(m())$, 集合中的每个跃迁都以 $m()$为触发事件.对一个用例控制类来说,它的状态图对于事件 $m()$的所有可能的响应就定义了 m 的行为: $m()\{\sqcap_{t\in\varepsilon(m())}cs(t)\}$.

这里,使用内部选择符连接响应函数调用事件 $m()$所有可能的跃迁,在函数调用事件发生以后,用例控制类从可能响应该事件的所有跃迁中,随机选择执行一个卫式条件为真的跃迁.

例 4:定义用例控制类 BuyItems-Controllers 的状态控制图为

BuyItems-Controller ::

$enterItem()\{(state=new)_{\top};(state'=\neg IsComplete)\sqcap(state=\neg IsComplete)_{\top};skip\}$

$endSale()\{(state=\neg IsComplete)_{\top};(state'=IsComplete)\}$

$makePayment()\{(state=IsComplete)_{\top};(state'=new)\}$

3.5 功能规约

本小节处理跃迁 $t=s\xrightarrow{l}s'$,其中, $l=\langle m(in;out),g,a\rangle\in Label$,在控制状态由 s 迁移到 s'的同时,动作 a 对数据状态的改变.把动作 a 的规约记作 $m@s\xrightarrow{l}s'$,在不引起歧义的情形下简单记作 $m@s$,并令 a 的规约为跃迁 t

的功能规约.同样地,功能规约是利用设计来定义的.

功能规约函数Φ用内部选则符$\sqcap$连接由调用 m 而引发的所有跃迁的功能规约,作为方法 m 完整的功能规约.

例 5:用例控制类 *BuyItems-Controllers* 中各种方法的功能规约可定义如下:

$$known(upc)=\exists sp\in \Pi(ProductSpecification)\bullet sp.upc=upc$$

$$newLine(li)=\exists li\in \Pi'(LineItem)\bullet LineItem.New(li);(li.quantity'=qty)$$

$$addDescribes(it,upc)=\exists d\in \Pi'(Describes)\bullet Describes.New(d);(d.item=it\wedge d.upc=upc)$$

$$addLineToSale(x,li)=\exists y\in \Pi'(Contains)\bullet Contains.New(y);(y.sale'=x\wedge y.line'=li)$$

$$addTotal()=total'=total+qty\times sp.price$$

$$enterItem@new=know(upc)\Rightarrow (Sale.New(sa);\bigvee_{sp.upc=upc}(newLine(li);addLineToSale(sa,li);addDescribes(li,upc))\wedge addTotal)$$

$$enterItem@\neg isComplete=know(upc)\Rightarrow \bigvee_{sp.upc=upc}(newLine(li);addLineToSale(sa,li);addDescribes(li,upc))\wedge addTotal)$$

$$endSale@\neg isComplete=skip$$

$$makePayment@isComplete=\exists p\in \Pi'(Payment)\bullet Payment.New(p);p.amount'=amount;amount\geq total\vdash balance'=amount-total$$

3.6 全规约

跃迁 $t=s\xrightarrow{l}s'$,其中 $l=\langle m(in;out),g,a\rangle\in Label$ 的全规约记为 $Spec(t)$,它集成了跃迁在控制状态上的变化和功能规约,定义它为

$$Spec(t)\overset{\text{def}}{=}(g\wedge state=s)_{\top};m()@s\xrightarrow{l}s'\wedge(state'=s').$$

上述定义表示一个跃迁只有当其功能规约中的卫式条件 g 被满足且当前的控制状态为 s 的情况下,才能发生,它会跃迁到控制状态 s'并且遵照功能规约 $m@s\xrightarrow{l}s'$ 的描述修改数据状态空间.

同样地,类中每种方法的全规约定义为通过内部选则符$\sqcap$连接由该方法调用引起的所有跃迁的全规约.令跃迁 $t_i=s_i\xrightarrow{l_i}s_i'$,且 $l_i=\langle m(in;out),g_i,a_i\rangle\in Label$, $i\in 1..k$, 是用例控制类 C 的状态图中,由对方法 m 的调用引起的所有的跃迁,则方法 m 的全规约定义为

$$Spec(C::m())\overset{\text{def}}{=}\{Spec(t_1)\sqcap\ldots\sqcap Spec(t_k)\}.$$

为使用方便,用 $body(C::m())$ 表示 $Spec(t_1)\sqcap\ldots\sqcap Spec(t_k)$,用 $g(C::m())$ 表示 $\exists i\bullet(1\leq i\leq k\wedge g_i\wedge(state=s_i))$.

4 一致性与检验方法

应用形式化分析建模框架,本节研究用例模型的一致性问题.一致性问题的研究包括两个方面:静态一致性把类图作为用例模型的类型定义部分,它检查各视图中对类、类关联、类方法和类属性的使用是否与类图中的类型定义有冲突.动态一致性则检查各个视图刻画的功能和动态行为规约是否一致.

4.1 静态一致性检查

类图Γ构成了整个模型的类型定义部分,因而其余视图中出现的类、类属性和类方法都应在类图Γ中有明确的定义,并且对它们的使用不能和类图中的类型定义相冲突.静态一致性要求:

(1) 系统序列图Δ中出现的用例控制类必须在类图Γ中有明确的定义,而且组成每个序列的事件,必须是对在类图Γ中定义的对应用例控制类方法的调用.

(2) 状态转换图Ω中出现的用例控制类必须在类图Γ中有明确的定义,而且引起状态跃迁的事件必须是对已在类图Γ定义的该用例控制类方法的调用.

(3) 被功能规约映射Φ说明的方法必须在类图Γ中有明确的定义,规约说明中引用的类、类属性、类的方

法必须在类图Γ中有明确的定义,且规约说明参照类图Γ中的类型没有类型错误.

(4) 系统不变式θ中引用的类、类属性、类的方法必须在类图Γ中有明确的定义,且该不变式参照类图Γ中的类型定义没有类型错误.

4.2 动态行为一致性检查

用例分析建模框架关于用例动态行为的描述包含几个方面:全规约是对用例中各种方法功能的完整定义,序列图定义了系统与环境交互时,方法调用应遵循的协议,状态图描述了方法调用引起的系统内部控制状态的变化,系统不变式是系统动态行为必须满足的约束.相应地,用例模型的动态一致性定义包含两个方面.定义全规约和系统不变式θ是一致的,如果所有用例控制类中所有方法的全规约都保持系统不变式;序列图、状态图和全规约都描述了系统与方法调用有关的行为,它们之间的一致性定义为:如果对于用例序列图Δ中每个用例控制类的所有函数调用序列,它可以被用例状态图Ω中关于同一控制用例的状态图和功能规约映射Φ中定义的相关函数功能规约所完全实现.

首先考虑全规约和系统不变式θ的一致性,它要求所有用例控制类中所有方法的全规约都保持系统不变式,可以形式化定义为:

若用例控制类U的方法m有型如$\{U::m()\{Spec(t_1)\sqcap Spec(t_2)...\sqcap Spect(t_k)\}\}$的全规约,其中$Spec(t_i)$是跃迁$t_i=s_i\xrightarrow{l_i}s_i'$, $l_i=\langle m(in;out),g_i,a_i\rangle\in Label$的全规约,那么对所有的$i\in 1..k$一定有:

$(\theta\wedge g_i\wedge state=s_i\wedge state'=s_i'\wedge a_i)\Rightarrow\theta'$,其中$\theta'$表示对$\theta$中所有的变量在其所有的自由出现处用其加撇号的形式予以替换后得到的谓词.

上述定义表明,对于用例控制类U的方法m可能引起的每个跃迁,如果该跃迁的起始状态能够保持系统不变式,那么其终止状态也一定能够保持系统不变式.

然后考虑关于交互行为的一致性问题.令v_0是用例控制类C的属性v的初始值,全规约$\{U::m()\{Spec(t_1)\sqcap Spec(t_2)...\sqcap Spect(t_k)\}\}$定义用例控制类$U$状态图的完整语义为一个跃迁序列的集合,集合中的元素型如:$s_0\xrightarrow{l_0}s_1\xrightarrow{l_1}s_2...$, $l_i=\langle m(in;out),g_i,a_i\rangle\in Label$,且对该序列有最弱前提条件:$wp(v=v_0;a_1;a_2...;a_i,g_{i+1}[in_{i+1}])=\text{true}$成立,对$i\geq 0$.它意味着,序列前缀的每个执行都建立了紧接着的下一跃迁的卫式条件.

据此,定义用例序列图Δ,用例状态图Ω,功能规约说明映射Φ是一致的,如果对于用例序列图Δ中每个用例控制类的所有函数调用序列,它可以被用例状态图Ω中关于同一控制用例的状态图和功能规约映射Φ中定义的相关函数功能规约所完全实现.形式化定义为:

$\forall tr=?m_0();...?m_k()\in Tr(U)$, U是一个用例控制类,型如$\overline{tr}=?m_0();?m_1()...?m_i(),i\leq k$的$tr$的任意前缀$\overline{tr}$一定满足:$wp(v'=v_0\wedge state'=s_0;body(m_0());body(m_1())...;body(m_j()),g(m_{j+1}()))=\text{true}$.

上述定义要求,用例控制类协议中出现的每个调用序列都是被该用例的状态转换图所允许的跃迁序列的一个前缀.这样就保证了,如果环境遵照用例控制类协议来调用方法,那么环境和用例控制类之间一定不会发生死锁.

对于每一个跃迁序列,用例控制类属性的初始值v_0、类方法的功能规约以及卫式条件需相互协调,从而保证一系列卫式条件和前置条件都能得到满足,才有这一跃迁序列的成功发生.卫式条件和前置条件在定义方法功能规约时发挥着不同的作用,卫式条件保证对该方法的调用不会引起死锁,而前置条件保证对该方法的调用不会导致系统发散.

例 6:检查例 1 中通过用例分析建模框架建立的用例模型是否满足本章定义的一致性.

首先,对照静态一致性检验规则,检查例 2~例 5 中用例模型各视图的形式化描述,可知其满足本章定义的静态一致性.

然后,检查用例模型是否满足动态一致性.先分析其方法的全规约是否和系统不变式相一致.系统的需求包含这样一个约束:只有产品目录里有记录的产品才能被出售.此约束θ可表示为

$\forall sa \in \Pi(Sale) \bullet (\forall ct \in \Pi(Contains) \bullet (ct.sale = sa \Rightarrow$

$\exists item \in \Pi(LineItem), d \in \Pi(Describes), ps \in \Pi(ProductSpecification) \bullet ct.li = item \wedge d.it = item \wedge d.upc = ps.upc))$

对于用例控制类 *BuyItems-Controller* 的方法 *endSale* 和 *makePayment* 有:

$\Pi'(Sale) = \Pi(Sale) \wedge \Pi'(Contains) = \Pi(Contains) \wedge \Pi'(LineItem) = \Pi(LineItem) \wedge \Pi'(Describes) = \Pi(Describes) \wedge \Pi'(ProductSpecification) = \Pi(ProductSpecification)$, 即$\theta$中出现的类的对象在方法执行前后保持不变,故它们的动态行为可以保持约束θ.

对于方法 *enterItem* ,则需要对 $Spec(enterItem@new)$ 和 $Spec(enterItem@\neg IsComplete)$ 两种情形分别加以证明.对 $Spec(enterItem@new)$ 需要证明:

$$(\theta \wedge g(enterItem@new) \wedge state = new \wedge state' = \neg isComplete \wedge a(enterItem@new)) \Rightarrow \theta'.$$

相应地,对 $Spec(enterItem@\neg IsComplete)$ 需要证明:

$$(\theta \wedge g(enterItem@\neg isComplete) \wedge state = \neg isComplete \wedge state' = isComplete \wedge a(enterItem@\neg isComplete)) \Rightarrow \theta',$$

其中的证明过程并不复杂,这里从略.

对关于交互行为的一致性,可以通过证明序列 $?enterItem()^+;?endsale();?makepayment()$ 的任意前缀 $?enterItem()^*,?enterItem()^+;?endsale(), ?enterItem()^+;?endsale();?makepayment()$,一定有:

$$wp(init; g(enterItem)) = \text{true}, wp(init; body(enterItem), ...body(enterItem), g(enterItem)) = \text{true},$$

$$wp(init; body(enterItem), ...body(enterItem), g(endSale)) = \text{true},$$

$$wp(init; body(enterItem), ...body(enterItem) body(endSale), g(makePayment)) = \text{true},$$

其中 $init \stackrel{def}{=} state' = new \wedge balance' = 0 \wedge total' = 0 \wedge sa' = null$.证明过程从略.

由上述证明可知,此用例模型满足本章定义的一致性.

5 相关工作的比较

已有很多的工作尝试用例图结合其他类型视图来描述系统的需求规约.在文献[6]中,一个需求规约包含了用例定义、类图和用例活动图 3 个部分.用例定义描述了用例的功能,活动图描述每个用例的行为模式,类图描述了用例模型的结构信息.3 个视图之间的一致性定义为类图可以实现每个用例的功能规约和行为规约.文献[7]引入问题域模型来定义系统的需求规约,一个问题域模型包含用类图表示的概念视图,描述用例交互活动的协作图和描述用例内部行为的活动图.问题域模型没有给出检查视图间一致性的方法.上述两种方法描述的需求规约可以直接在本文的用例模型中加以描述.这两个工作的最大问题在于视图缺少严格的形式化定义,模型中功能规约的定义采用了自然语言或者是半自然语言半形式化的描述,因而,无法对用例模型描述的规约作语义检查.文献[8]研究了用例模型的形式规约,它用合同定义用例模型的功能规约,应用最弱前提条件演算分析模型的性质.与本文的工作相比,它只给出了形式化用例功能规约的方法,而未考虑对问题领域的概念模型和用例与环境交互行为的形式化建模.

与本文紧密相关的工作还包括对 UML 模型的形式化建模和语义一致性检验.文献[9]使用规约语言 *Z* 对类图、对象图和类状态图进行形式化.类被处理成规约语言 *Z* 中一个抽象数据类型(abstract data type,简称 ADT),类方法的功能规约使用前后置条件定义.在此形式化框架下,类图和类状态图间的一致性被定义为状态图定义的方法功能规约应满足类图中的不变式约束.文献[10]研究了用 *B* 语言形式化 UML-RSDS 模型的方法,它包含对类图、类状态图和 OCL(object constraint language)约束的形式化.每个类被翻译成一个 *B* 机器,类方法的功能规约被定义为一对前后置条件,利用 OCL 描述的约束被处理成相应 *B* 机器的不变式.与上述工作一样,本文使用成熟的形式化理论体系定义 UML 的语义.成熟的形式化理论体系不仅有完整而严密的推理体系,更有成熟的工具支持.文献[9]中提到可利用工具 Z/Eves 辅助证明的过程.文献[10]研究了从 UML-RSDS(unified modeling language-reactive systems design support)模型的 *B* 语言描述转换到模型检验工具 SMV 的输入,从而利用后者检验模型的时序性质.本文的形式化工作也很容易获得工具的支持.根据设计演算的相应规则,可以开发直接支持设计演算推理过程的证明工具,也可以采用转换的方法,利用现有的工具辅助证明过程.例如,根据设计的定义

和设计最弱前提条件的定义,可以方便地把最弱前提条件的演算过程转换到谓词演算上,并使用 PVS(prototype verification system)等定理证明工具辅助证明的过程.也可以直接将使用设计描述的系统转化成 *B* 机器或者是 *Z* 的模式(schema),并借助它们的支持工具完成分析和证明.

文献[11]研究类图和类状态机的形式化建模.它应用规约语言 Object-*Z* 定义类图,用协议状态机定义类状态机,这些定义可被进一步翻译成 CSP(communicating sequential processes)进程,类图和类状态机之间的一致性可以利用模型检验工具 FDR(failures divergence refinement)来检验类图中每个函数的功能规约是否支持状态机描述的函数调用序列,即它们之间是否有精化关系.文献[12]研究序列图和状态图的一致性,序列图和状态图都被翻译成一组 CCS(calculus of communicating systems)进程,它们之间的一致性被定义为开放进程间的互仿真关系.文献[13]用描述逻辑来形式化定义类图、序列图和状态图,并用逻辑推理来证明它们之间的一致性.文献[14]定义了一种新的图形化表示——属性图,用它定义类图和序列图的语义,在此基础上,由研究类图和序列图到属性图的转换方法,类图和序列图是一致的仅当它们对应的属性图是一致的.

与上述工作相比,本文的形式化工作处理了更多类型的视图,包含了描述问题域概念结构的概念类图,描述交互行为的序列图和状态图,描述系统方法功能的功能规约和描述全局性质的系统不变式,它们被统一在基于设计演算的语义框架里.引入多种类型的视图,可以方便用户描述需求的多个方面.各类视图形式化语义的引入,使得需求的各个方面有了严格而准确的描述,而且也可以应用形式化的证明和验证技术分析需求的各个方面.特别地,在面对大规模软件系统时,需求的各个方面可以分别被描述和分析,降低了用例分析和建模过程的复杂度.同时,用例分析建模框架提供了集成需求各个方面和检查用例模型一致性的形式化方法,可以有效地保证大规模用例模型的正确性.

6 总结以及工作展望

如何准确地描述系统的需求,是软件开发方法研究的关键问题之一.本文提出了一个用例分析建模框架,它引入了多种类型的视图,支持用户从多个角度为需求建模.此框架定义了各类视图的形式化语义,从而为多种视图所描述的需求的各个方面赋予了准确的形式化定义.框架提供了集成方法,支持从方法的交互行为规约和功能规约生成描述方法完整行为的全规约.用例模型中引入形式化的描述,不仅可以准确地描述系统的需求,而且为应用形式化分析和验证技术分析用例模型的性质准备了条件.基于此形式化框架,本文研究了用例模型的一致性问题,给出了检验一致性的规则和演算方法.

进一步的工作包括开发支撑工具,工具接受输入的用例模型后,可以辅助用户应用设计演算规则,检查用例模型的一致性.工具也可用于证明系统的某些性质.工具通过集成其他定理证明工具,将需要证明的命题转换成其他定理证明工具的标准形式,从而自动化或者是半自动化地完成证明.更深入的研究包括为基于 UML 的设计模型和实现模型建立形式化途径,并研究用例模型、设计模型和实现模型之间的精化方法.

致谢 本文的主要工作是作者在 UNU/IIST 访问学习期间完成的,文中的很多建模思想和技术方法都来源于与刘志明研究员的讨论,在此向他表示衷心的感谢.

References:

[1] Ivar J, Grady B, James R. The unified process. IEEE Software, 1999,16(3):96−102.

[2] Philippe K. The Rational Unified Process—An Introduction. 2nd ed., Boston: Addison-Wesley, 2000. 155−169.

[3] Craig L. Applying UML and Patterns. 2nd ed., London: Prentice-Hall Int'l, 2001. 45−191.

[4] He JF, Li XS, Liu ZM. rCOS: A refinement calculus of object systems. Theoretical Computer Science, 2006, 365(2):109−142.

[5] Tony H, Jifeng H. Unifying Theories of Programming. London: Prentice-Hall, 1998. 74−82.

[6] Georg K, Hans-Werner S, Mario W. Coupling use cases and class models as a means for validation and verification of requirements specifications. Requirements Engineering, 2001,6(1):3−17.

[7] Egidio A, Gianna R. Tight structuring for precise UML-based requirement specifications. In: Martin W, Alexander K, Simonetta B, eds. Proc. of the 9th Int'l Workshop on Radical Innovations of Software and Systems Engineering in the Future. LNCS 2941, Berlin: Springer-Verlag, 2002. 16−34.

[8] Luigia P, Ralph-Johan B, Ivan P. Analysing UML use cases as contracts. In: Robert BF, Bernhard R, eds. Proc. of the 2nd Int'l Conf. on the Unified Modeling Language (UML'99). LNCS 1723, Berlin: Springer-Verlag, 1999. 518−533.

[9] Nuno A, Susan S, Fiona P. Formal proof from UML models. In: Jim D, Wolfram S, Michael B, eds. Proc. of the 6th Int'l Conf. on Formal Engineering Methods (ICFEM 2004). LNCS 3308, Berlin: Springer-Verlag, 2004. 418−433.

[10] Kevin L, David C, Kelly A. UML to B: Formal verification of object-oriented models. In: Eerke AB, John D, Graeme S, eds. Proc. of the 4th Int'l Conf. on Integrated Formal Methods (IFM 2004). LNCS 2999, Berlin: Springer-Verlag, 2004. 187−206.

[11] Holger R, Heike W. Checking consistency in UML diagrams: Classes and state machines. In: Elie N, Uwe N, Perdita S, eds. Proc. of the 6th IFIP WG 6.1 Int'l Conf. on the Formal Methods for Open Object-Based Distributed Systems. LNCS 2884, Berlin: Springer-Verlag, 2003. 229−243.

[12] Vitus SWL, Julian P. Consistency checking of sequence diagrams and statechart diagrams using the π-calculus. In: Judi R, Graeme S, Jaco VDP, eds. Proc. of the 5th Int'l Conf. on Integrated Formal Methods (IFM 2005). LNCS 3771, Berlin: Springer-Verlag, 2005. 347−365.

[13] Ragnhild VDS, Tom M, Jocelyn S, Viviane J. Using description logic to maintain consistency between UML models. In: Perdita S, Jon W, Grady B, eds. Proc. of the 6th Int'l Conf. on the Unified Modeling Language, Modeling Languages and Applications (UML 2003). LNCS 2863, Berlin: Springer-Verlag, 2003. 326−340.

[14] Aliki T, Hartmut E. Consistency analysis between UML class and sequence diagrams using attributed graph grammars. In: Ehrig H, Taentzer G, eds. Proc. of the GraTra 2000—Joint APPLIGRAPH and GETGRATS Workshop on Graph Transformation Systems. Berlin: Technische University Berlin, 2000. 77−86.

陈鑫(1975－),男,辽宁大连人,博士,主要研究领域为软件工程,面向对象的技术,构件技术,形式化方法.

李宣东(1963－),男,博士,教授,博士生导师,CCF 高级会员,主要研究领域为软件工程,形式化方法,软件验证.