UML的形式化及其应用*)

周 瑾[1] 马应龙[1] 李 巍[1] 吴志林[2]
(中国科学院软件研究所软件工程中心 北京100080)[1]
(中国科学院软件研究所计算机科学重点实验室 北京100080)[2]

摘 要 本文介绍一个面向度量的UML的代数表达框架。这个框架可以作为设计模型检测的基础,并为设计人员提供一个在软件设计进化过程中检测一些设计错误和提出可能的优化方案的理论基础。本文给出了UML类图、序列图和状态图的代数表达并用例子说明了这个UML的代数表达框架的应用及它是如何检测设计错误和提供可能的优化建议的。

关键词 UML,形式化

A Formal Framework of UML and its Application

ZHOU Jin[1] MA Ying-Long[1] LI Wei[1] WU Zhi-Lin[2]
(Technology Center of Software Engineering, Institute of Software, Chinese Academy of Sciences, Beijing 100080)[1]
(Key Labaratory of Computer Science, Institute of Software, Chinese Academy of Sciences, Beijing 100080)[2]

Abstract This paper describes a metrics-oriented algebraic representation for UML Class, Sequence, and State Transition diagrams. The paper uses an example to illustrate how to transform between UML graphical and algebraic representations; the transformation serves as the basis of automated generation of algebraic UML during design. The algebraic representation of object-oriented design provides a foundation for design model checking.

Keywords UML, Formal framework

1 引言

统一建模语言UML(Unified Modeling Language)是OMG(Object Management Group)组织提出的一个通用的可视化语言,是面向对象分析与设计的一种标准表示[1,2]。UML用图形符号描述模型,用模型来表示软件系统的结构(或静态特征)以及行为(或动态特征)。UML包括九种图,其中,类图、序列图和状态图是UML的核心子集——类图描述系统的静态结构;序列图描述系统中对象之间通过消息进行的交互;状态图描述系统中个体对象内部可能的行为及状态。

运用UML可以图形化地进行面向对象的系统设计,但是,其非形式化特征使开发者无法从中看出设计的优劣,不利于对系统优劣的度量,不能帮助开发人员评估和改进其设计。对UML进行形式化,给出其形式化的语法和语义,就可以用数学的方法对设计做正确性检查,并且能够用算法实现机器自动检测,发现设计中的错误,帮助对设计进行优化;通过对提取出的度量数据进行统计和评估,可以反映出设计的优劣,能够指导开发者将来设计的方向、维护测试、资源分配等种种工作,提高整个软件产品和开发过程的质量。

目前,已经有一些组织和研究者从事这方面的工作。这些工作从形式化描述方法分有两种:一种是基于UML的元模型的形式描述,另外一种是基于数学理论基础的形式化描述。pUML组织在UML的元模型的研究方面进行了很多的工作:A. S. Evans等人的策略是明确定义UML的核心元素指称语义[5,6];J. M. Alvarez等人定义了元模型语言的活动语义[7]。使用数学方法进行UML形式化是很多研究者采用的方法:M. Shroff使用Z语言对类图进行形式化描述[8];Xuandong Li和Johan Liliu等对UML的序列图进行了形式化[9],将序列图是否满足时间约束一致性归结为一组不等式方程组是否有解。

但是,目前大多数基于元模型或数学基础的UML形式化描述方法都没有与度量联系起来,都不适合度量,从中提取度量数据比较困难。本文提出的面向度量的UML形式化框架为这个问题提供了一个解决方案。该框架针对OO度量的需要,将冗余信息删除,突出与度量相关的信息,从中可以方便地提取度量数据。这样,在软件开发的早期——系统设计阶段就可以对软件的质量进行度量、分析、监控,达到事半功倍的作用。

本文着重描述对UML的形式化及其应用,关于其在度量方面的应用将于另一篇文章中描述。本文内容组织如下:第1节简单介绍本文目的及内容;第2节描述UML形式化框架;第3节通过一个字查找系统的例子讲述UML形式化的应用。

2 UML形式化框架

这一部分介绍面向度量的UML形式化框架。目前只对UML的核心子集——类图、序列图和状态图进行了形式化,其它图的形式化有待日后补充,该形式化框架将逐步完善。

2.1 UML类、对象及信号的形式描述

为了对UML进行形式化,首先要对其中的几个核心概念:数据类型、类、对象和信号进行形式化。

在一个设计模型中,数据类型被用来声明属性、变量和参数,分为基本数据类型和用户自定义的数据类型两种。用户自定义类型是用户自定义的类,而基本数据类型实际上也可以看成类。例如在Java中整数类型Integer就是一个类,只是由于Integer类型太通用和常见了,因此大家将其看成一个不可分割的整体(这实际上也是基本数据类型和用户自定义类型

周 瑾 硕士研究生,研究方向为软件工程技术。

的划分依据)。本质上这两种数据类型是一致的,都可以看成是一个类,因此将数据类型组成的集合定义为一个类的集合。

定义1(数据类型) 数据类型组成的集合 T 是一个类的集合,即在 UML 设计模型中的属性、变量和参数的声明中用到的所有类的集合。其中,基本数据类型组成的类集合记为 BT,用户自定义数据类型组成的类的集合记为 UT。显然,$T=BT\cup UT$。

由于类名唯一标识了一个类,而在声明属性、变量或参数时通常使用类名来引用类,因此有些时候可以将数据类型集合 T 看成是设计模型中所有类的名称组成的集合,BT 和 UT 也同样可以看成是由类名组成的集合。为了后面定义的方便,假定 BT 中包含一个空类型 *void*。

定义2(类) 一个类 c 定义为三元组 $c=(name,A,M)$,其中:

·*name* 表示类名。

·A 是 c 的属性的集合;每个属性 a 可以定义为一个四元组 $a=(name,type,scope,visibility)$,其中:*name* 表示属性的名称;$type\in T$ 表示 a 的类型,无定义时默认为 *void*;*scope* 表示属性是类级别的(*static*)还是对象级别的:$scope\in\{$"*class*","*object*"$\}$;*visibility* 表示属性的可见性,$visibility\in\{$"*public*","*private*","*protected*"$\}$。目前没有考虑包(package)。

类 c 中所有类级别的属性组成的集合记为 $CA=\{a\in A\mid ascope=$"*class*"$\}$,对象级别的属性组成的集合记为 $OA=\{a\in A\mid ascope=$"*object*"$\}$,显然 $A=CA\cup OA$。

由于在一个类中不能出现同名的属性,为了表述的方便,引用属性的时候直接使用属性名称。

·M 为类 c 的方法的集合;每个方法 m 可以定义为一个四元组 $m=(name,fct,scope,visibility)$,其中:*name* 表示方法的名称;*fct* 描述方法的输入参数和返回值的类型,$fct\in T^*\times T$。如果 $fct=(t_1,t_2,\cdots,t_n,t)$,$n$ 是输入参数的个数(可以为 0),$t_i(1\leqslant i\leqslant n)\in T$ 表示方法的第 i 个输入参数的类型,t 表示返回值的类型。假定方法均有返回值,没有返回值的默认其返回值为 *void*。

为了后面表述的方便,将 *fct* 的最后一个分量记为 *ran*,而从 *fct* 的元组分量中除去 *ran* 之后剩下的部分记为 *dom*。如果把方法看成函数,则 *dom* 代表方法的定义域,而 *ran* 代表方法的值域。

·*scope* 表示方法是类级别的(static)还是对象级别的:$scope\in\{$"*class*","*object*"$\}$;*visibility* 表示方法的可见性,$visibility\in\{$"*public*","*private*","*protected*"$\}$。目前没有考虑包(package)。

可能存在同名的方法(重载),但 *name* 和 *fct* 组成的二元组(*name*,*fct*)对于每一个方法来说都是唯一的,因此在引用方法的时候使用二元组(*name*,*fct*);而在不引起混淆的情况下,也可以用方法名标识方法。

对类 c 的引用可以直接采用其第一个分量 *name*,因为在同一个命名空间内类不会重名。而对类 c 的各个元组分量的引用采用 *name*.* 的形式,例如 *name*.A 等。对于属性和方法元组中的分量的引用也采用类似的方法。

将类形式化后,就可以给出数据类型取值的定义,如下:

定义3(数据类型的取值范围) 基本数据类型 $bt\in BT$ 的取值范围遵循通常的理解和定义,例如 integer 类型的取值范围集合是$\{0,\pm1,\pm2,\cdots,\pm n,\cdots\}$;用户自定义类型 $ut\in UT$ 的取值范围集合递归定义如下:假定 ut 的属性集合 $ut.A=\{(a_1,t_1,scope_1,visibility_1),\cdots,(a_k,t_k,scope_k,visibility_k)\}$,类型 $t_i\in T(1\leqslant i\leqslant k)$ 的取值范围集合为 $VS_i(1\leqslant i\leqslant k)$,则 ut 取值范围集合为 $VS_1\times\cdots\times VS_k$。例如,类 c 包括类型分别为 Integer 和 Char 的两个属性,则 c 的取值范围 $VS_c=VS_{Integer}\times VS_{Char}$,$VS_{Integer}=\{0,\pm1,\pm2,\cdots,\pm n,\cdots\}$,$VS_{Char}$ 为所有的 ASCII 字符组成的集合。

用 $t.VS$ 来表示类型 t 的取值范围集合。

定义4(对象) 对象 o 定义为 $o=(name,t,V)$。

·*name* 表示对象的名字。由于 UML 中允许存在匿名对象,因此 *name* 允许为空。

·$t\in T$ 表示对象声明时所使用的类型。

·V 表示对象的实例变量的集合,每个实例变量 v 定义为 $v=(name,type)$,其中 *name* 是实例变量的名字,而 $type\in T$ 表示实例变量的类型。V 满足 $|V|=|t.OA|$ 并且 $\forall v\in V(\exists a\in t.OA(v.name=a.name\wedge v.type=a.type))$。

定义5(对象的状态) 对象 o 的状态 s 定义为 o 的属性的取值,即 $o.s\in o.t.VS$。

对一个对象的各个属性进行赋值之后就得到对象的一个状态。显然,对象可以有很多个状态,但在某一具体时刻对象只能处于一个状态。

另外,因为序列图和状态图都牵涉到了信号(signal)这一概念,所以我们对 UML 中的信号概念也进行了形式化。在 UML 中,信号被定义成一种特殊的类,它没有方法,其属性就是信号的参数,但没有对属性的可见性做出规定。在信号之间可以存在继承(generalization)关系,因此所有的信号可以放到一个特殊的类图——信号图(Signal Diagram)中进行定义,图中只有信号这种特殊的类,而且类之间除了继承关系之外不存在其它的关系。

在这里我们默认一个信号的所有属性的可见性都是"*public*",其定义如下:

定义6(信号,signal) 一个信号 $sig=(name,A)$,其中 *name* 和 A 的定义与在类中的定义相同,而且 A 满足 $\forall a\in A$,$a.visibility=$"*public*"。

将所有信号组成的集合记为 Sig。

2.2 类图的形式描述

定义7(类图) 类图 cd 的形式化为 $cd=(name,CS,ASS,GEN,AGG,DEP)$ 的五元组,其中:

·*name* 代表 cd 的名称,可以为空。

·CS 是 cd 描述的类的集合。

·ASS 是 cd 中类间直接关联关系(association)$ass=(name,c_1,c_2,dir)$的集合。

其中:*name* 是 *ass* 的名字。由于允许存在匿名关联,*name* 允许为空;$c_1,c_2\in CS$ 是 *ass* 所关联的两个类;$dir\in\{$"*bidirectional*","*unidirectional*"$\}$代表关联的方向类型:$dir=$"*bidirectional*"时,关联为双向的;$dir=$"*unidirectional*"时,关联的方向是由 c_1 指向 c_2。

·$GEN\subseteq CS\times CS$ 是 cd 中类间直接继承关系(generalization)的集合。令 $c_1,c_2\in CS$,则 $(c_1,c_2)\in GEN$ 当且仅当 c_1 是 c_2 的泛化,即 c_2 是 c_1 的子类。

·$AGG\subseteq CS\times CS$ 是 cd 中类间直接聚合关系(aggregation)的集合。令 $c_1,c_2\in CS$,则 $(c_1,c_2)\in AGG$ 当且仅当 c_1 聚合 c_2,即 c_2 是 c_1 的一部分。

·$DEP\subseteq CS\times CS$ 是 cd 中类间直接依赖关系(dependency)的集合。令 $c_1,c_2\in CS$,则 $(c_1,c_2)\in DEP$ 当且仅当 c_1 依赖于 c_2。

一个设计模型中所有类图的集合记为 CD,类图中所有类的集合即为 UT,表示所有的用户自定义类型。为了后面讨论的方便,将各个类图中包含的所有类的集合记为 $CD.CS$,

将的各个类图中的所有直接关联关系的集合、所有直接继承关系的集合、所有直接聚合关系的集合以及所有直接依赖关系的集合分别记为 $CD.ASS$、$CD.GEN$、$CD.AGG$ 和 $CD.DEP$。

特殊地，我们将进行信号及信号之间关系定义的图称为信号图(Signal Diagram)，可以认为信号只放在这一张图中进行定义，即信号图在一个设计模型中是唯一的。其定义如下：

定义8(信号图，Signal Diagram) 信号图 $sigDiag=(Sig,GEN)$，其中：

·Sig 表示所有的信号组成的集合。

·$GEN\in Sig\times Sig$ 表示信号之间的继承(Generalization)的关系，其定义与类图中 GEN 的定义完全类似。

对类图进行定义后，类之间的关系由类图集合完全确定。下面就数据类型间的关系进行讨论。

定义9(数据类型之间的 Is 关系) 即数据类型中子类型和父类型之间的关系。用户数据类型分为基本数据类型和用户自定义类型，因此数据类型之间的 Is 关系也分为基本数据类型之间的 BIs 关系和用户自定义类型之间的 UIs 关系(假定基本数据类型和用户自定义类型之间不存在子类型和父类型的关系)。显然 $Is=BIs\cup UIs$。基本数据类型之间 BIs 的关系遵循常规定义。用户自定义数据类型之间的 UIs 关系定义为 $UIs\subseteq UT\times UT$：令 $sut,ut\in UT$ 则 $(sut,ut)\in Is$ 当且仅当 $\exists\ ut_1,\cdots,ut_n\in UT$ 使得 $(ut,ut_1)\in CD.GEN$，$(ut_1,ut_2)\in CD.GEN,\cdots,(ut_n,sut)\in CD.GEN$($n$ 可以为0)。

2.3 序列图的形式描述

定义10(序列图) 一个序列图 $sed=(OS,Mes)$是一个二元组，其中：

·OS 是 sed 中所有对象的集合。

·Mes 是 sed 中所有消息的集合，每个消息 mes 定义为 $mes=(sn,name,para,sender,peceiver,type)$。其中：$sn$ 是一个序号，描述消息 mes 在序列图的时间序列中的相对位置，sn 不可省略；$name$ 是 mes 的名称；$para=((name_1,t_1),\cdots,(name_n,t_n))$，$t_i(1\leqslant i\leqslant n)\in T$，是 mes 所带参数的名称和类型组成的元组，n 是参数的个数(n 为0表示没有参数或参数省略)，其中 $name_i$ 和 t_i 分别表示第 i 个参数的名称和类型，没有定义时分别默认为空串和 $void$；$sender\in OS$ 是 mes 的发送者，$receiver\in OS$ 是 mes 的接收者；$type\in$ {"$syncall$","$asyncall$","$send$","$return$"}，描述 mes 的类型："$syncall$"代表方法的同步调用；"$asyncall$"表示方法的异步调用；"$send$"表示信号(signal)的发送；"$return$"表示方法同步调用结束后的返回。

这里 $name,para,receiver,type$ 之间必须满足以下约束：

①如果 $type=$"$syncall$"或者 $type=$"$asyncall$"，则$\exists\ m\in receiver.t.M$ 使得 $name=m.name$ 且 $para=((name_1,t_1),\cdots,(name_n,t_n))$满足 $n=0$或者 $n=|m.dom|$(记 $m.dom=(pt_1,\cdots,pt_n)$)且$\forall\ 1\leqslant i\leqslant n(t_i=pt_i\vee(t_i,pt_i)\in Is)$(即 $para$ 中每个参数的类型应该和 m 的输入参数类型一致)。

②如果 $type=$"$send$"则$\exists\ sig\in Sig$ 使得 $name=sig.name$ 且 $para=((name_1,t_1),\cdots,(name_n,t_n))$满足 $n=|sig.A|$且 $sig.A$ 中的属性存在一种排列$(a_1,\cdots,a_n)$使得$\forall\ 1\leqslant i\leqslant n(t_i=a_i.type\vee(t_i,a_i.type)\in Is)$。

因为返回消息的作用不大，这里暂时不考虑返回消息。

将设计模型中所有序列图的集合记为 SED。

2.4 状态图的形式描述

由于在 UML 中没有对动作(Action)的表达方式进行精确的定义[1,2]，因此在 UML 的形式化框架里面也没有给出动作的确切定义，而仅仅将动作的表达当作一种无格式的字符串。

定义11(状态图) 状态图 std 形式化为 $std=(context,SV,COM,ES,TS,Init,End)$，其中：

·$context$ 表示状态图的上下文(目前只考虑类或方法)，即 $context\in$ {"$class$","$method$"}。

·SV 表示状态图中所有状态结点的集合，每个状态结点 sv 定义为 $sv=\begin{pmatrix}name,type,entry,exit,\\doActivity,isConcurrent,isRegion\end{pmatrix}$

其中：$name$ 代表状态结点的名称；

$type\in\left\{\begin{matrix}"SynchState","StubState","PseudoState",\\"SimpleState","CompositeState",\\"SubmachineState","FinalState"\end{matrix}\right\}$

代表状态结点的类型；

$entry$ 表示状态结点的入口动作，可以为空；

$exit$ 表示状态结点的出口动作，可以为空；

$doActivity$ 表示处于状态结点 sv 时会执行的动作，可以为空；

$isConcurrent\in\{0,1\}$，$isConcurrent=1$表示状态结点 sv 是并发状态，否则不是；

$isRegion\in\{0,1\}$，$isRegion=1$表示状态结点是某个并发状态的子区域，否则不是。

·$COM\subseteq SV\times SV$ 表示状态结点间直接包含关系，$com=(sv_1,sv_2)\in COM$ 表示 sv_1直接包含 sv_2。

·ES 代表状态图中所有事件的集合，每个事件 e 定义为 $e=(name,para,type)$。

其中：$name$ 表示事件的名称；$para=((name_1,t_1),\cdots,(name_n,t_n))$表示事件的参数，$n$ 表示事件参数的个数(可以为0，表示没有参数或参数省略)，其中 $name_i$ 和 $t_i\in T(1\leqslant i\leqslant n)$分别是第 i 个参数的名称和类型；$type\in$ {"$call$","$send$","$time$","$change$"}表示事件的类型："$call$"代表方法调用事件；"$send$"代表信号(signal)发送事件；"$time$"代表时间事件；"$change$"代表变化事件。

·TS 表示状态之间的转换关系的集合，每一个转换关系 $trans$ 定义为 $trans=(source,target,trig,guard,action)$。

其中：$source,target\in SV$，分别代表 $trans$ 的转出和转入状态；$trig\in ES$ 表示 $trans$ 的触发事件；$guard$ 表示进行状态转换 $trans$ 时需要满足的条件的集合，这里也仅将其看成是字符串；$action$ 表示进行状态转 $trans$ 时需要执行的动作的集合。

·$Init\in\left\{\begin{matrix}sv\in SV\mid(sv.type="PseudoState")\wedge\\(\forall\ (sv_1,sv_2)\in COM(sv_2\neq sv))\end{matrix}\right\}$表示状态图的初始状态。

·$End=\left\{\begin{matrix}sv\in SV\mid(sv.type="FinalState")\wedge\\(\forall\ (sv_1,sv_2)\in COM(sv_2\neq sv))\end{matrix}\right\}$表示状态图的终结状态组成的集合(由于状态图中存在 concurrent 的状态，因此除了 End 中的 FinalState 之外，还有可能存在其它 FinalState)。

SV 和 COM 应该满足：

$$\forall\ sv\in SV\left(\begin{matrix}\left(\begin{matrix}sv.isConcurrent=1\Rightarrow\\\left(\begin{matrix}sv.type="CompositeState"\wedge\\\forall\ ssv\in SV\left(\begin{matrix}(sv,ssv)\in COM\\\Rightarrow ssv.isRegion=1\end{matrix}\right)\end{matrix}\right)\end{matrix}\right)\wedge\\\left(\begin{matrix}(sv.isRegion=1)\Rightarrow\\\left(\begin{matrix}\forall\ ssv\in SV\left(\begin{matrix}(ssv,sv)\in COM\Rightarrow\\ssv.isConcurrent=1\wedge\\ssv.type="CompositeState"\end{matrix}\right)\end{matrix}\right)\end{matrix}\right)\end{matrix}\right)$$

即并发状态直接包含的一定是区域(Region)状态结点，而直接包含区域状态结点的状态一定是复合的(composite)和并

发的(concurrent)。

将设计模型中所有状态图的集合记为 STD。

另外,为了方便指称,本文以自然数为下标来标识集合里的不同元素,例如,以 $O_1,O_2,\cdots$ 来标识不同的对象。

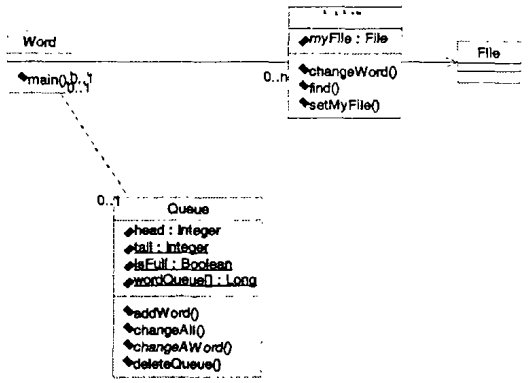
3 UML 形式化的应用

如前文所述,将 UML 形式化后,能够采用数学的方法对 UML 设计模型进行正确性检测,用算法实现机器自动检查。主要的思想是建立一个规则库,存储 UML 图应该遵循的规则,这些规则可以是 UML 语言本身的规定(如不可递归继承),也可以是开发者为了优化 UML 设计而作的规定(如继承不可超过20层等);用规则库中的规则检验 UML 设计,实现机器自动检查。

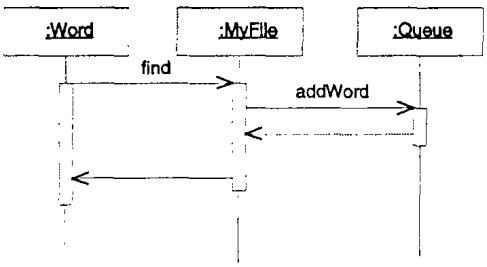
下面就以一个字查找系统 UML 设计为例(该系统在文档中查找指定的词,并且可以将查找到的词进行替换),说明第2介绍的 UML 形式化框架的应用,并通过几个较常见的错误说明其在 UML 设计正确性的机器自动检查方面的意义:

错误检测:图间不一致错误。

例如,在序列图中,类 MyFile 与 Queue 间有消息传递,但是在类图中它们之间无关联关系,如下所示。



(a) 类图



(b) 序列图

图1 错误示例——图间不一致

图中,MyFile 类为要进行处理的文件,Queue 类实现一个循环先进先出的队列,储存在 myFile 中查找到的指定词的位置,以便进行替换。该图的形式化如下:

·类图:

$$cd_1=\begin{pmatrix}\text{“”},\{Word,MyFile,Queue,File\},\\ \{ass_1,ass_2\},\Phi,\Phi,\{dep_1\}\end{pmatrix}$$

$ass_1=(\text{“”},Word,MyFile,\text{“unidirectional”})$

$ass_2=(\text{“”},Word,Queue,\text{“unidirectional”})$

$dep_1=(MyFile,File)$

·序列图:

$sed_1=(\{o_1,o_2,o_3\},\{mes_1,mes_2,mes_3,mes_4\})$

objects:

$o_1:o_1=(\text{“”},Word,\Phi)$

$o_2:o_2=(\text{“”},MyFile,\{v_1\})$

$v_1=(\text{“myFile”},File)$

$o_3:o_3=(\text{“”},Queue,\{v_1,v_2,v_3,v_4\})$

$v_1=(\text{“head”},Integer)$

$v_2=(\text{“tail”},Integer)$

$v_3=(\text{“isFull”},Boolean)$

$v_4=(\text{“wordQueue[]”},Long)$

messages:

$mes_1=(1,\text{“find”},(\text{“”},void),o_1,o_2,\text{“syncall”})$

$mes_2=(2,\text{“addWord”},(\text{“”},void),o_2,o_3,\text{“syncall”})$

$mes_3=(3,\text{“”},(\text{“”},void),o_3,o_2,\text{“return”})$

$mes_4=(4,\text{“”},(\text{“”},void),o_2,o_1,\text{“returen”})$

由 UML 的定义可知,有关系传递的对象分别所属的类之间必定存在关系,即如下规则:

$$if\exists\ sed_i\in SED,mes_i\in sed_i.Mes,mes_i.sender=o_1,$$
$$mes_i.receiver=o_2,o_1.t=c_1,o_2.t=c_2$$
$$\Rightarrow\begin{pmatrix}\exists\ ass_i\in CD.ASS,(ass_i.c_1=c_1\wedge ass_i.c_2=c_2)\\ \vee(ass_i.c_1=c_2\wedge ass_i.c_2=c_1)\end{pmatrix}\text{由}$$
$$\vee\begin{pmatrix}\exists\ gen_i\in CD.GEN,(gen_i.c_1=c_1\wedge gen_i.c_2=c_2)\\ \vee(gen_i.c_1=c_2\wedge gen_i.c_2=c_1)\end{pmatrix}$$
$$\vee\begin{pmatrix}\exists\ agg_i\in CD.AGG,(agg_i.c_1=c_1\wedge agg_i.c_2=c_2)\\ \vee(agg_i.c_1=c_2\wedge agg_i.c_2=c_1)\end{pmatrix}$$
$$\vee\begin{pmatrix}\exists\ dep_i\in CD.DEP,(dep_i.c_1=c_1\wedge dep_i.c_2=c_2)\\ \vee(dep_i.c_1=c_2\wedge dep_i.c_2=c_1)\end{pmatrix}$$

序列图的形式化定义可知 $mes_2\in sed_1.Mes,mes_2.sender=o_2$,$mes_2.receiver=o_3,o_2.t=MyFile,o_3.t=Queue$,类 MyFile 的对象 o_2 与类 Queue 的对象 o_3 间有消息传递,但由类图的形式化定义可以得出:

$$!\begin{pmatrix}\exists\ ass_i\in CD.ASS,(ass_i.c_1=MyFile\wedge ass_i.c_2=Queue)\\ \vee(ass_i.c_1=Queue\wedge ass_i.c_2=MyFile)\end{pmatrix}$$
$$\wedge!\begin{pmatrix}\exists\ gen_i\in CD.GEN,(gen_i.c_1=MyFile\wedge gen_i.c_2=Queue)\\ \vee(gen_i.c_1=Queue\wedge gen_i.c_2=MyFile)\end{pmatrix}$$
$$\wedge!\begin{pmatrix}\exists\ agg_i\in CD.AGG,(agg_i.c_1=MyFile\wedge agg_i.c_2=Queue)\\ \vee(\ agg_i.c_1=Queue\wedge agg_i.c_2=MyFile)\end{pmatrix}$$
$$\wedge!\begin{pmatrix}\exists\ dep_i\in CD.DEP,(dep_i.c_1=MyFile\wedge dep_i.c_2=Queue)\\ \vee(dep_i.c_1=Queue\wedge dep_i.c_2=MyFile)\end{pmatrix}$$

MyFile、Queue 间无关系相连,由此可知存在类图与序列图间不一致的错误。

除了可以发现 UML 设计中的错误外,UML 形式化还可以查出设计中的不精确的地方,提出改进方案:

1) *定义不精确* 例如,若类图中 MyFile 与 File 间定义为 dependency 关系,但 MyFile 的属性变量 myFile 的类型是 File 类,这时更精确的定义应该是 MyFile 聚合 File,它们之间是聚合(aggregation)关系。如图2所示。

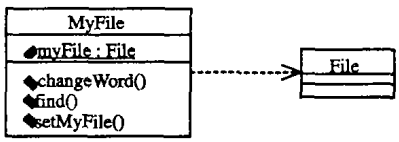


图2 优化示例——类间关系定义不精确

该图的形式化为:

·类 *MyFile*:

$MyFile=(\text{“MyFile”},\{a_1\},\{m_1,m_2,m_3\})$

$a_1=(\text{“myFile”},File,\text{“object”},public)$

$m_1=(\text{“changeWord”},(void),\text{“object”},public)$

$m_2=(\text{“find”},(void),\text{“object”},public)$

$m_3=(\text{“setMyFile”},(void),\text{“object”},public)$

·类图的形式化为:

$cd_2=(\text{“”},\{MyFile,File\},\Phi,\Phi,\Phi,\{dep_1\})$

$dep_1=(MyFile,File)$

由 UML 的定义可以得到如下规则:$\forall\ c_1,c_2\in CD.CS$,$\exists\ a_1,(a_1\in c_1\cdot A)\wedge(a_1.type=c_2)\Rightarrow\exists\ agg_i\in CD.AGG,(agg_i\cdot c_1=c_1)\wedge(agg_i.c_2=c_2)$即若类 A 包含类型为类 B 的属性,则 A 与 B 应为 aggregation 关系。

由上述形式化可知$(a_1\in MyFile.A)\wedge(a_1.type=File)$,MyFile 应与 File 是 aggregation 的关系,而类图的形式化表

明 MyFile 与 File 是 dependency 关系，由于 dependency 关系较 aggregation 关系弱，因此这个定义不是错误但不够精确。

2）设计中待完善的漏洞　例如，在类图中，如果类 MyFile 与 InWord 是聚合关系（MyFile 聚合 InWord），但是，MyFile 中没有包含任何 InWord 类型的属性变量，如图3所示，那么这里就有待设计者进一步地完善。

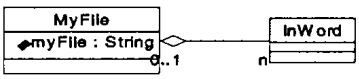


图3　优化示例——关系定义漏洞

则类图的形式化如下：

·类：

$InWord: InWord=(\text{“}InWord\text{”},\Phi,\Phi)$

$MyFile: MyFile=(\text{“}MyFile\text{”},\{a_1\},\Phi)$

$a_1=(\text{“}myFile\text{”},String,\text{“}object\text{”},public)$

·类图：

$cd_2=(\text{“”},\{InWord,MyFile\},\Phi,\Phi,\{agg_1\},\Phi)$

$agg_1=(MyFile,InWord)$

根据 UML 的定义，可以得出如下规则：$\forall\ c_1,c_2\in CS, if\ \exists\ agg_i\in CD.AGG, agg_i.c_1=c_1$，即如果 $agg_i.c_2=c_2\Rightarrow\exists\ a_i\in c_1.A, a_i.type=c_2$

c_1与 c_2是 aggregation 的关系，那么 c_1应该包含类型为 c_2 的属性变量。

由类图的形式化中关于关系 agg_1的定义可知 $MyFile, InWord\in CS, agg_1\in CD.AGG, agg_1.c_1=MyFile, agg_2.c_2=InWord$。而类的形式化中属性集合 $MyFile.A=\{a_1\}, a_1.type=String$，不包含类型为 InWord 的属性变量，由此可知设计存在不完善之处。

3）基于设计的优劣考虑，可对 UML 图形的设计做一些限制　可以通过加入规则对 UML 图形设计进行限制，以防止一些不良设计的出现。例如，通过规则：$\forall\ c_i\in CD.CS, if\ \exists\ c_i=c_1,\cdots,c_n\in T(n\geqslant 1)$，并且$(c_1,c_2),\cdots,(c_n,c)\in CD.GEN$ $!\exists\ c_0,(c_0,c)\in CD.GEN\Rightarrow n\leqslant 20$可以保证类图中不会出现继承深度过大的情况。

另外，状态图的形式化如下：

MyFile 类有三个状态——Initial（初始）、Find（查找）和 Change（替换），其状态图见图4。

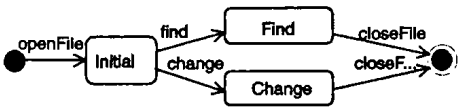


图4　类 MyFile 的状态图

形式化如下：

$STD=\{std_1\}$

$std_1=\left(\begin{array}{l}\text{“}class\text{”},\{sv_1,sv_2,sv_3,sv_4,sv_5\},\Phi,\{e_1,e_2,e_3,e_4\},\\ \{trans_1,trans_2,trans_3,trans_4,trans_5\},sv_1,sv_5\end{array}\right)$

$sv_1=(\text{“”},\text{“}PseudoState\text{”},\Phi,\Phi,\Phi,0,0)$

$sv_2=(\text{“}Initial\text{”},\text{“}SimpleState\text{”},\Phi,\Phi,\Phi,0,0)$

$sv_3=(\text{“}Find\text{”},\text{“}SimpleState\text{”},\Phi,\Phi,\Phi,0,0)$

$sv_4=(\text{“}Change\text{”},\text{“}SimpleState\text{”},\Phi,\Phi,\Phi,0,0)$

$sv_5=(\text{“”},\text{“}FinalState\text{”},\Phi,\Phi,\Phi,0,0)$

$e_1=(openFile,(void),\text{“}call\text{”})$

$e_2=(find,(void),\text{“}call\text{”})$

$e_3=(change,(void),\text{“}call\text{”})$

$e_4=(closeFile,(void),\text{“}call\text{”})$

$trans_1=(sv_1,sv_2,e_1,\Phi,\Phi)$

$trans_2=(sv_2,sv_3,e_2,\Phi,\Phi)$

$trans_3=(sv_2,sv_4,e_3,\Phi,\Phi)$

$trans_4=(sv_3,sv_5,e_4,\Phi,\Phi)$

$trans_5=(sv_4,sv_5,e_4,\Phi,\Phi)$

最后，需要注意的是：在形式化一个 UML 系统设计时，会有多个序列图，因为每个图都是独立的，所以形式化时，每个图中的对象和消息均从1开始编号。

总结　本文中我们介绍了一个面向对象设计的形式化框架，给出了 UML 类图、序列图和状态图的代数表达，同时举例说明了如何将 UML 图形转换成上述 OO 模型的代数表示。上述 OO 模型的代数表示以及 OO 设计模型到其代数表达的自动转换为 OO 设计过程中的设计模型的自动检测错误和提供设计优化的建议提供了一个理论基础。

参 考 文 献

1 Unified Modeling Language: Infrastructure version 2.0 3rd revised submission to OMG RFP ad/00-09-01, 2003. http://www.omg.org/cgi-bin/doc?ad/2003-03-01

2 Unified Modeling Language: Superstructure version 2.0 3rd revised submission to OMG RFP ad/00-09-02, 2003. http://www.omg.org/cgi-bin/doc?ptc/2003-08-2

3 Booch G, Rumbaugh J, Jocobson I. The Unified Modeling Language User Guide. Massachusetts, Addison-Wesley, Reading, 1998

4 Rumbaugh J. Unified modeling language reference manual. Massachusetts, Addison-Wesley, Reading, 1999

5 Evans A S, Kent S. Meta-modelling semantics of UML: the pUML approach. In: 2nd Intl. Conf. on the Unified Modeling Language, B. Rumpe and R. B. France, eds. Colorado, LNCS 1723, 1999. http://www.cs.york.ac.uk/puml/publications.html

6 Evans A S, France R B, Lano K C, Rumpe B. Meta-modelling semantics of UML. In: Behavioural Specifications for Businesses and Systems, Kluwer, Haim Kilov, ed. Chapter 4, 1999. http://www.cs.york.ac.uk/puml/publications.html

7 Alvarez J M, et al. An Action Semantics for MML. In the fourth workshop on Rigorous Object Oriented Methods, King's College, March 2002. http://www.dcs.kcl.ac.uk/staff/tony/docs/ActionSemanticsForMML.pdf

8 Shroff M, France R B. Towards a Formalization of UML Class Structures in Z. In: Proc. COMPSAC'97, Los Alamitos, CA, Aug. 1997. 646～651

9 Li Xuandong, Lilius J. Timing Analysis of UML Sequence Diagrams. In: UML'99, The Unified Modeling Language. Beyond the Standard. The Second International Conference, Fort Collins, CO, USA, 1999