维普资讯 http://www.cqvip.com

面向方面软件重构等价性形式化证明方法*)

陈生庆[1] 张立臣[2] 陈广明[1]

(嘉应学院计算机系 梅州 514015)[1] (广东工业大学计算机学院 广州 510090)[1]

摘 要 软件重构在不改变程序行为的情况下通过对代码进行小的改进以提升设计,使之更容易理解和维护,面向方面的程序设计是软件开发的新技术,为了有效实施面向方面的软件重构,需要开发者识别面向方面程序的转化规则。然而,由于使用的AOP语言没有形式化的语义定义,难以确认转化和重构的程序运行行为。本文对MCI操作语义的面向方面的扩展使之支持程序的方面特征的描述,定义了两个程序的观测等价,讨论了AspectJ的形式语义模型的建立,在MCI的语义下形式化地精确证明了Add Before-executing编程规则的观测等价性,其基本原理和方法可以适用于其他规则的证明,通过上述工作提出了面向方面重构的程序和它的面向对象程序原型等价性的证明方法。

关键词 重构,面向方面编程,形式化方法,MCI,等价性,AspectJ

An Equivalence Proving in Formal Method for Aspect-Oriented Refactory

CHEN Sheng-Qing[1] ZHANG Li- Cheng[2] CHEN Guang-Ming[1]

(Computer Department of Jiaying College, Meizhou 514015)[1]

(Computer College, GuangDong University of Technology, Guangzhou 510090)[2]

Abstract Refactoring is that you make small changes to your code to improve your design, making it easier to understand and maintrained without to change program behaviours. Aspect Oriented Programming(AOP) is a new technology for software development. In order to explore the benefits of refactoring aspect-oriented developers are identifying common transformations for aspect-oriented programs the laws proof. However, they lack support for assuring that the transformations observation behaviour and are indeed refactorings because there is not formal semantics in the AOP language that we have used. In this paper, an operational semantics for Method Call Interception(MCI) is extented to fit in with represent AOP featurs. An equivalence relation stating that two programs have the same observation behaviour is defined; the paper presents a formal model for the semantics of AspectJ. We use these concepts and discuss exactness for its laws in formal mothed MCI. The observation equivalence proving of Add Before-executing law can be used in other law. In conclusion, we show how it is possible to prove that an new aspect-oriented program by refactoring is equivalent its Object Oriented prototype

Keywords Rafactoring, Aspect oriented programming, Formal method, MIC, Equivalence, AspectJ

1 前言

面向方面的编程(Aspect Oriented Programming, AOP)是在对面向对象基础上进行扩展的新的软件开发方法,它定义了横切需求的统一表达方式,通过方面建模解决了需求横切问题,通过特征需求解决方案的剥离,使开发者能够集中于整体结构的搭建,简化了开发的难度,提高了代码的可读性,为软件资源的复用打下了良好的基础[1,2]。

软件重构是遗留系统再工程以及代码优化和维护的重要手段,它在不改变软件可观测行为的基础上对软件内部的代码、结构进行重组,使之更为清晰、具有更高的效率。随着面向方面软件开发的逐步成熟,利用方面语言的特征实现对面向对象软件中横切代码的重构成为引人关注的问题。在重构过程中,需要确定重构前后程序可观测运行结果的等价性,而对编程语言规则的形式化表达将有利于对实际的状态转化进行精确的描述,并以此建立重构的等价性判别基础[3~5]。

面向方面语义一直是研究的热点,然而目前使用的面向方面语言仍然没有形式化的语义描述[6],因此无法对程序进行有效的精确推理,这对程序等价性判别造成很大的困难。利用形式化语义定义面向对象语言及与之对应的方面语言的编程规则,利用其操作语义的推导规则,是对程序可观测运行结果等价性证明的有效途径。由于典型的面向对象语言Java及其经方面扩展而成的AspectJ具有丰富表现力,其编程规则也非常丰富,构造或扩展某种形式化语义对其完整描述并非易事。由于方面重构的等价性证明仅仅涉及其中的部分规则,因此对这部分规则的形式化语义描述可以起到事半功倍的效果。我们的目标是通过选择对面向对象概念具有良好表现能力的形式化方法,对其进行结构、语义进行扩展,使其支持方面概念的描述,从而建立重构前后代码可观测等价性的定义和推导方法,最终提出证明实际编程语言编程规则等价性的基本方法,并以此为基础检验重构代码的观测等价。

形式化方法MCI(Method Call Interception)的优点在于其利用了简洁的形式对面向对象各种结构和操作进行了精确表达,同时其清晰的语言特征有利于实现面向方面的扩展。针对MCI的面向方面扩展类似于AspectJ对Java的扩展,这些语义能很好地表现面向方面的核心概念advice,为从面向对象语言到面向方面语言语义转化提供了良好的基础。

本文的第2部分讨论MCI的语义和提出程序观测等价,这个部分的主要工作是定义了观测等价的概念,它为MCI语义的方面扩展之后的等价性判别提供了统一的基础;在第3

*)本文受国家自然科学基金(No. 60474072、No. 60174050),广东省自然科学基金(No. 04009465、No. 010059),广东省高校自然科学研究项目(No. Z03024)基金资助。**陈生庆** 讲师,硕士,主要研究方向:软件工程技术、实时系统;**张立臣** 教授,主要研究方向:分布式、实时系统;**陈广明** 副教授,硕士,主要研究方向:软件工程技术、形式化方法。

维普资讯 http://www.cqvip.com

个部分讨论 MCI 的方面扩展，定义了新的语言元素和操作语义；第 4 部分讨论了方面重构所必须的基本编程规则观测等价性的证明方法；最后讨论了与此有关的工作并对本文的工作进行了总结。

2 MCI 的语义和程序等价性

2.1 MCI 的语义

文[6,7]定义了类 Java 的形式语言 MCI，描述了其操作语义、语言的求值和命令规则。由于静态语义被用于按照类型系统检验程序的结构合法性，而不涉及程序的行为，因而本文只考虑其动态语义。MCI 语言的构造规则如图 1 所示（虚线后为扩展的 MCI，后文另述）：

```
prog=cdef* cn. mn
cdef=class cn extends cn {field* mdef* }
field=type fn
mdef=type mn (arg* ) body
type=cn|void
arg=type vn
body=exp|abstract
cn=class names
fn=field names
mn=method names
vn=variable names
exp =null|this|vn|view type exp|exp. fn|exp. vn
    = exp|exp. mn (exp* )|super. mn (exp* )
-----------------------------------------------------
|let vn: type=exp in exp|exp;exp|while (exp) exp
    |caller|callee|superimpose exp on eve
eve=mci loc|eve within loc
mci=dispatch|enter|exit
loc=* |object exp|class cn|subclass cn|method mn|
     result type|argument type vn
     |loc && loc|loc||loc|! loc
```

图 1 MCI 构造规则

prog 表示程序，cdef 表示类定义，field 表示类中的成员，mdef 表示方法定义，arg 表示方法参数，body 表示方法体，cn、fn、mn 和 vn 分别表示类名、成员名、方法名和变量名，exp 表示表达式，加粗标识为保留字。

MCI 运算规则包括四个域：方法代码表(T)，它将方法名及参数映射到对应的方法体；对象存储(Σ)，是对象到它的成员(field)值的完全函数；执行对象的引用标识(θ)；程序变量环境标识(η)。本文采用符号 $\Pi_i(t)$ 表示多元组的第 i 维投影。运算规则的实例化表示了表达式的运算结果和系统状态的变化。整个程序的求值可以依赖于由规则的实例构成的一棵求值推导树。

如前文所言，选择 MCI 的重要原因是它对面向对象的良好表达能力，MCI 经面向方面扩展所引入的结构和语义只影响方法调用的求值规则，因而对该求值规则的理解将使我们能精确地了解方面扩展后的语义特征是如何变化的。有关 MCI 的其余求值规则的可以参考文[7,8]。

$$T,\Sigma_0,\theta,\eta \vdash \exp \Rightarrow \rho,\Sigma_1 \quad (1)$$

$$\Pi_1(T)\cdot(\rho,mn)=((vn_1,\cdots,vn_n),\exp') \quad (2)$$

$$T,\Sigma_1,\theta,\eta \vdash exp_1 \Rightarrow vn_1,\Sigma_2 \quad (3)$$

$$\cdots \quad (4)$$

$$T,\Sigma_n,\theta,\eta \vdash exp_n \Rightarrow vn_n,\Sigma_{n+1} \quad (5)$$

$$\eta' = \perp[vn_1 \to v_1,\cdots,vn_1 \to vn] \quad (6)$$

$$T,\Sigma_{n+1},\theta,\eta' \vdash exp' \Rightarrow v,\Sigma_{n+2}' \quad (7)$$

$$T,\Sigma_0,\theta,\eta \vdash exp.\ mn(exp_1,\cdots exp_n) \Rightarrow v,\Sigma_{n+2}' \quad (8)$$

图 2 面向对象调用求值规则

ρ 表示通用对象的引用。$f.x$ 表示完全函数，$\perp$ 表示不完全函数，方法调用的求值产生了返回值 v'，和一个更新的对象存储(Σ_{n+2}')。

赋予方法调用规则的语义解释是：在初始环境(T,Σ_0,θ,η)下求值表达式 exp，该表达式的求值返回了实例对象 ρ 并用 Σ_1 表示新的对象存储；(3)～(5)为方法参数的顺序求值的条件和结果，如果满足(2)和(6)的要求，装配了经求值参数而形成的运行环境将执行方法体。

2.2 MCI 程序的等价

使用 MCI 语义去推理面向方面的程序，检测两个程序的可观测行为是否相同，需要定义等价关系。一个合适的等价关系体现了定义者对程序等价的理解[6,7]。

思考两种堆栈的执行：使用数组表示堆栈和使用链表。两个程序实现的都是栈的行为，但由于数据结构不同，最终的状态也不相同。程序执行结果包括对象存储状态和变量状态，链表和栈的对象存储状态是相同的。仅由变量状态不同而判定两个程序不等价与本文讨论的目标不相符合。由于可将影响运行观测结果的变量转化为对象描述，而具有 MCI 语义的编程规则不改变数据对象的结构，因此可以通过比较对象存储状态来建立一种被称之为观测等价的等价关系。

定义 1 对象存储 $\Sigma=(\delta \to_{fin} v)\times(\rho \to_{fin} cn)$。其中，$\delta$ 为成员位置，ρ 为对象的引用，$_{fin}\ v$ 为合法的成员值，$_{fin}\ cn$ 为类型。

定义 2 P 和 Q 是两个 *MCI* 程序，其中 ΣPB、ΣQB 和 ΣPE、ΣQE 分别表示程序 P、Q 执行前后的对象存储，$P\langle I\rangle$、$Q\langle I\rangle$。表示程序 P、Q 的合法输入集合，$P\langle\rho\rangle$、$Q\langle\rho\rangle$ 表示 P、Q 程序运行中对象引用的集合，$\mathcal{R}$：$P\langle\rho\rangle \leftrightarrow Q\langle\rho\rangle$ 表示一一映射，P 观测等价于 $Q(P \equiv_O Q)$ 当且仅当：

$$\forall pi,qi,pq,qq \cdot pi \in P\langle I\rangle \wedge qi \in Q\langle I\rangle \wedge pq \in P\langle\rho\rangle \wedge qq \in Q\langle\rho\rangle \wedge pi = qi \wedge \Sigma_{PB} = \Sigma_{QB} \wedge pq \leftrightarrow qq \in \mathcal{R} \Rightarrow \Pi_1(\Sigma_{PE}) = \Pi_1(\Sigma_{QE})$$

通过定义 2 可以发现，观测等价表示的是程序运行结束后 P 的对象存储结果中的各个对象成员值同 Q 中的相同。比较状态等价，观测等价的概念更弱，但它更为灵活，在等价关系的推导中可以将足够表示程序观测行为的状态同其他次要状态相分离，只要前者满足定义 2 的要求，即认为两个程序是观测等价的。由数组和链表构成的堆栈执行是不同的，但是按照定义 2 忽略了次要变量和对象结构上的差异，在建立了数组对象和链表对象引用的映射后，它们是观测等价的。这个结果更符合应用的需要。另外由于观测等价对程序的外部行为感兴趣，定义 2 采用的是对象而非类的等价。如果存在一个从未被程序调用的方法，它不会影响观测等价的推导构成。

3 MCI 的语义的面向方面扩展

3.1 语言元素的扩展

文[9]在 MCI 中定义了新的结构 superimpose，赋予该结构的语义可以使 superimpose 在某个事件发生时截取一个正在执行的方法，转而求值给定的表达式，这种结构可以作为 MCI 面向方面扩展的基础。由于 superimpose 过于简单，需要赋予它更丰富的涵义：首先方法执行环境中的变量对 superimpose 可见；其次容许 superimpose 截取多个方法。

对 MCI 的面向方面扩展后的语法结构如图 1，虚线后为方面扩展的语言元素，开始于 caller 定义。Superimose 结构定义了在事件(eve)发生时表达式(exp)将被求值。eve 可以被看作是 AspectJ 中的 pointcut 表达式，事件描述定义了何时何处方法拦截发生，一个方法可能在三个点(mci)被拦截：dispatch，此时表达式 exp 在参数前求值；enter，此时表达式 exp 在参数求值后方法执行前求值；exit，此时表达式 exp 在方法执行后被求值。这些 mci 点类似于 AspectJ 中的 before-

维普资讯 http://www.cqvip.com

call，before-execution 和 after-returning-execution 。事件的另一个组成部分(loc)基于类、方法、参数及返回类型等描述了方法拦截的具体位置。

由于 MCI 的方面扩展引入了 advice 概念，需要在对象存储(Σ)中增加了一个表达式登记(Δ)域，它类似于 AspectJ 中的 advice 的引用。

$$T,\Sigma_0,\theta,\eta \vdash eve \Rightarrow k,\Sigma' \quad (9)$$

$$\tau_\alpha=((\Pi_{cn}(\theta),\Pi_{mn}(\theta))=\exp) \quad (10)$$

$$Register(k,\Sigma',\alpha)\Rightarrow\Sigma'' \quad (11)$$

$$T,\Sigma_0,\theta,\eta \vdash superimpose\ exp\ on\ eve \Rightarrow 0,\Sigma'' \quad (12)$$

图 3　superimpose 求值规则

图 3 展示了 superimpose 结构的求值规则。这条规则规定求值 superimpose 声明返回一个空引用(0 是空表达式的涵义)和一个更新的对象存储(Σ'')。superimpose 求值包括三个条件：首先事件表达式的求值，该求值产生事件描述(k)和一个更新的对象存储(Σ')(9)；其次建立由 α 表示的类似于 AspectJ 中 advice 的执行体(10)；最后 register 助手函数通过登记更新对象存储为Σ''(11)。

一个事件可以通过 superimpose 结构登记。在 MCI 的 call 规则中查询函数将搜索运行环境，识别被登记的事件是否同执行的方法相匹配，如果匹配将对登记的表达式求值，superimpose 必须在方法调用前被求值以使 advice 生效。在 superimpose 求值前的任何方法调用行为将遵照 MCI 原有的规则。该特征实际上可以支持 advice 的动态引入。

3.2　方面扩展的 MCI 语义

虽然增加了 MCI 的结构，但是只有方法调用求值规则需要改变，比较图 2 表示的调用规则，图 4 描述的经面向方面扩展的方法调用求值规则，增加了(15)(20)(22)的求值，定义的助手函数 dispatch，enter，exit 将分别在参数表达式求值前、参数表达式求值后和方法体执行前以及方法体执行后进行检测，寻找匹配方法的被登记事件并求值相应的表达式。

$$T,\Sigma_0,\theta,\eta \vdash \exp \Rightarrow \rho,\Sigma_1 \quad (13)$$

$$\Pi_1(T)\cdot(\rho,mn)=((vn_1,\cdots,vn_n),\exp') \quad (14)$$

$$dispatch(T,\Sigma_1,\theta,(\rho,mn))\Rightarrow\Sigma_1' \quad (15)$$

$$T,\Sigma_1',\theta,\eta \vdash \exp_1 \Rightarrow vn_1,\Sigma_2 \quad (16)$$

$$\cdots \quad (17)$$

$$T,\Sigma_n,\theta,\eta \vdash \exp_n \Rightarrow vn_n,\Sigma_{n+1} \quad (18)$$

$$\eta'=\perp[vn_1\to v_1,\cdots,vn_1\to v_n] \quad (19)$$

$$enter(T,\Sigma_{n+1}',\theta,(\rho,mn),\eta')\Rightarrow\Sigma_{n+1}' \quad (20)$$

$$T,\Sigma_{n+1},\theta,\eta' \vdash \exp'\Rightarrow v,\Sigma_{n+2}' \quad (21)$$

$$exit(T,\Sigma_{n+2}',\theta,(\rho,mn),\eta',v)\Rightarrow v',\Sigma_{n+2}' \quad (22)$$

$$T,\Sigma_0,\theta,\eta \vdash \exp.mn(\exp_1,\cdots\exp_n)\Rightarrow v',\Sigma_{n+2}' \quad (23)$$

图 4　面向方面的调用求值规则

为确保所有的 *superimpose* 表达式在程序开始执行前被求值，*superimpose* 只被定义在主方法开始的位置。为了更好地适应对编程语言程序等价性的推导，需要将 *MCI* 语义映射到类似 *AspectJ* 这样编程语言中，可以参照具体语言的的编程规则，对 *MCI* 的语义进行进一步的约束。文[9]中给出了 30 条 *AspectJ* 的编程规则，方面 *MCI* 可以映射其中的主要部分。由于 *superimpose* 求值规则仅涉及增加的表达式登记(Δ)域，因此在没有方法拦截时面向方面扩展的 *MCI* 在语义上同 *MCI* 一致，这是使用方面 *MCI* 证明面向方面重构程序的观测等价性的重要基础。

4　类 AspectJ 程序等价性证明

利用方面 MCI 规则对类 AspectJ 的重构程序的观测等价性进行严格证明的基本方法是：(1)比较类 Java 和类 AspectJ 中的编程规则，抽象出它们的操作语义；(2)用方面 MCI 语义对这些编程规则进行精确描述；(3)利用 MCI 操作语义建立求值推导树，证明编程规则具有观测等价性。由于类 Java 和类 AspectJ 语言的编程规则大部分只涉及本地代码，而且每一个规则都集中于一个程序的结构，根据结构归纳法的基本原理，对基本编程规则的观测等价性证明工作解决了重构程序等价性形式化证明问题。

4.1　编程规则的操作语义

在 AspectJ 中，Add Before-executing 规则表示的是在方法执行前，方面定义的 advice 体的联结过程(weaving)，规则将部分方法的代码移进 advice，这些 advice 在方法参数求值后、方法体执行前被触发执行。图 5 的 A、B 部分是结合语言特性对 Add Before-executing 规则进行抽象后的结构表示，其余的编程规则可以按相同的方法加以处理。

为方便使用面向方面 MCI 操作语义对规则进行形式化证明，图 5 的 A、B 部分采用了一种简洁的伪代码表示，如有需要也可以将其改写为具体语言的代码表示。在抽象过程中需要详细考察语言的特性，下面进行一些必要的说明。

在编程规则的描述中，ts 表示类的和方面定义的集合，fs 表示 f 成员定义的集合，ms 表示方法定义的集合，pcs 表示 pointcut 的集合。需要注意的是 advice 不能表示为集合，因为定义的顺序决定了 advice 的优先权。

在 advice 内，可以在被捕获的 join point 上下文中存取变量。表达式 bind(context)包括那些被披露的上下文的 pointcut 标志符。检查 Add Before-executing 规则左侧的结构，body′在加入点(join point)定义的具有 before 性质的 advice 之后执行，这意味着规则右边的新的 advice 应该最后一个被执行以保持两侧代码执行顺序。因此，新加入的具有 before 性质的 advice 应定义在 before 和 around 性质的 advice 之后、after 性质的 advice 之前。另外在规则 Add Before-executing 中新的 advice 最后执行，这需要有一个前置条件规定 aspect A 对在 ts 中定义的其他 advice 具有低的优先级，图 5 的 A、B 部分中通过语言元素定义的顺序保证了该条件成立。

在规则右侧将 body′移入方面，需要约束上下文依赖以保证规则关联合法的 AspectJ 程序。通过 this 指针可以进行私有成员的存取操作，在表达式 body′[cthis/this]中，所有 this 出现的场合将其替换为 body′中的变量 cthis。

通过上面的讨论可以得出图 5 的 A、B 部分描述完整，准确地体现了 Add Before-executing 规则，通过左右两侧的比较，明确了方面联结的语义和过程，可以作为使用 MCI 语义形式化证明规则观测等价性的基础。该规则的证明，可以保证任何符合该结构左右两侧的程序实例，是观测等价的。

4.2　观测等价性的证明

根据图 5 的 A、B 部分的语义描述，用 MCI 语言对 Add Before-execution 编程规则进行形式化的描述，结果如图 5 的 C、D 部分所示。

在图 5 的 C、D 部分中，将 AspectJ 的 before-execution advice 映射到 MCI 语言的 superimpose on enter 结构中。同时规定只能在主方法开始定义处定义 superimpose。在前文讨论的 advice 顺序问题，对 Add Before-executing 规则的 MCI 形式化表示不会产生语义上的影响，无需对 advice 按性质进行分离，只需要将 enter 处新定义的 superimpose 放在所有其它的 superimpose 之前，以确保新的 superimpose 最后被执行即可。

维普资讯 http://www.cqvip.com

```
ts A
class C {
  fs
  ms
  T m(ps) {
    body';
    body
      }
  }

  paspect A {
    pcs
    bars
    afs
    }
```

```
ts B
class C {
  fs
  ms
  T m(ps) {
    body
      }
  }
  paspect A {
    pcs
    bars
    before(context):
    exec( δ (C.m))&&bind(context)
    bind(context){body'[cthis/this]}
  afs}
```

```
ts C
class C extends T{
  fs
  ms
  Type m(ps) {
    body';
    body
      }
  }
class M extends T {
    void main(){
      sis
      mainBody
  }  }
```

```
ts D
class C  extends {
  fs
  ms
  Type m(ps) {
    body
      }
  }
  class M extends{
   void main{
    superimpose body'
     on enter class C
     &&method m
     &&atgument ps
     sis;
mainBody}}
```

图 5　Add Before-executing(A、B) 编程规则和 Add Before-executing (MCI)(C、D)编程规则

前文已经说明在 MCI 的方面扩展中只有方法调用 call 的求值规则发生了改变，因而可以保留除 call 之外所有其它的 MCI 语义。mainbody 表示一个对类 C 中方法 m 的单独调用，因此需要通过 let c:C=new C in c. m(ps)命令建立一个对象以实现对方法的调用。由于 superimpose 仅仅影响方法调用的语义，而 mainbody 中任何其它的简单结构无须顾及方法调用，因此其他语言结构不受 superimpose 定义的影响。

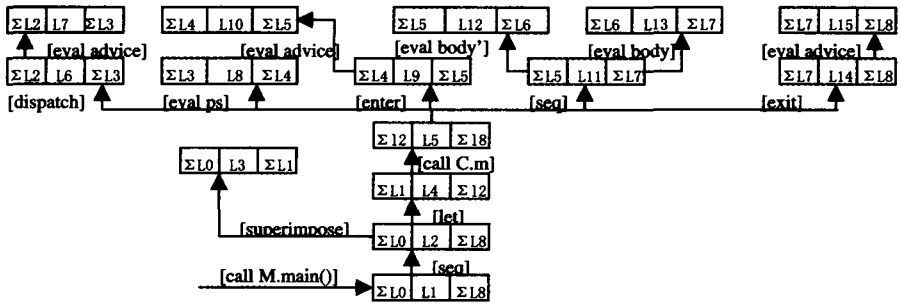

图 6　左侧的求值树

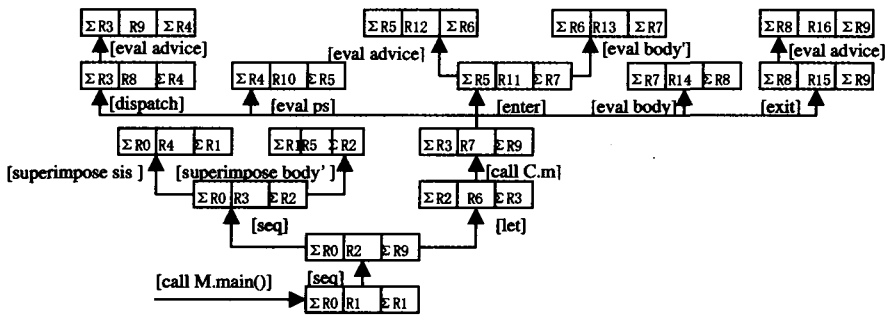

图 7　右侧的求值树

在图 6、7 中是根据方面扩展的 MCI 语义构造的 Add Before-executing 规则的求值推导树，图 6 展示了规则左侧的求值推导树，它由表示程序运行状态的各个节点组成。箭头表示按照 MCI 操作语义的求值。节点左边的域表示每一个转化的输入对象存储，右边的正方形表示每一个输出对象存储。节点按照执行的顺序被编号，第一个节点被标记为 L1。

首先求值一个连续的语言成分(L2)，然后是 sis 中的 superimpose 定义求值(L3)和 let 命令的求值(L4)。Let 命令更新了存储并且调用类 C 中的方法 m(L5)。方法调用求值规则由图 2 说明，首先，执行对 dispatch MCI 登记的事件(L7)，然后求值方法的参数(L8)，接着执行 enter MCI 登记的事件(L10)，随后是求值连续语言成分构成的方法体(L11)，包括 body′(L12)和 boby(L13)两个部分。最后执行 exit MCI 登记的事件(L15)。由于只关心两个程序的执行的比较，对两个程序中的相同部分，如 body，ps，dispatch 和 exit advice 无需进一步展开的求值。

图 7 中表示右侧的求值树。连续的语言成分(R2)先于另一个(R3)求值，在这个语言成分中包括图 7 中显式定义的 superimpose(R4)和其他没有显式描述的 superimpose(R5)。同左侧类似，开始调用方法 C. m。求值 superimpose 命令在进入类 C 的方法 m(其参数为 ps)时，通过对将要执行的 body′登记更新位于对象存储的 registry。enter 求值帮助函数(R11)在存储对象的 registry 中对相关于该方法的被登记事件进行搜索，从而发现应该被执行的 body′(R12)。同左侧

维普资讯 http://www.cqvip.com

相比另一个不同是方法体的求值只包括 body(R14)。

定理 1 Add Before-executing 规则具有观测等价性。

证明:按照在观测等价性的定义,我们只对那些可以更新对象存储对象成员的节点感兴趣。首先,let 命令可以通过增加一个新的对象和对象的 field 值来更新对象存储。另外更新对象存储对象成员的方法是赋值。赋值可能出现在任何表达式中。

据此寻找能够对表达式求值的节点。

在左边,同求值相关的表达式是:let(L4),dispatch(L7),ps(L8),enter(L10),body′(L12),body′(L13)和 exit(L15)。类似地,在右侧同求值相关的表达式:let(R6),dispatch(R9),ps(R10),enter(R12),body′(R13),body(R14)和 exit(R16)。

建立一个关系 $\mathcal{R}=\{(L4,R6),(L7,R9),(L8,R10),(L10,R12),(L12,R13),(L13,R14),(L15,R16)\}$,分析对应节点的求值链:

(1)L4 和 R6 的求值链:Σ_{R0}[seq][let]→L4,Σ_{L0}[seq][let]→R6;

(2)L7 和 R9 的求值链:L4→[call C. m][dispatch][eval advice]→L7,R6→[call C. m][dispatch][eval advice]→R9;

(3)L8 和 R10 的求值链:L7→[eval ps]→L8,R9→[eval ps]→R10;

(4)L10 和 R12 的求值链:L8→[eval advice]→L10,R10→[eval advice]→R12;

(5)L12 和 R13 的求值链:L10→[eval body′]→L12,R12→[eval body′]→R13;

(6)L13 和 R14 的求值链:L12→[eval body]→L13,R13→[eval body]→R14;

(7)L15 和 R16 的求值链:L13→[eval advice]→L15,R14→[eval advice]→R16;

显然,如果两个程序的初始存储结构 Σ_{R0} 和 Σ_{L0} 是相等的,那么(1)到(7)的求值过程的初始值和命令都是相同的,因而其存储结构 Σ 中的对象成员的值是相同的。根据定义 2,定理 1 得证。

这个证明可以简单地推广到规则 Add before-call 和规则 Add after returning successfully。它们唯一不同是使用的 advice,规则 Add before-call 在 dispatch 结构上使用 superimpose,规则 Add after returning successfully 将在 exit 结构上使用 superimpose。规则 Add before-call 的右侧推导树中 body′在其他方法调用前被求值。这意味着 body′甚至先于被调用的方法参数而求值,因此左边的求值树应该将 body′放在 dispatch 节点之上,以保证它在被考虑方法的参数前被求值。规则 Add after returning successfully 的证明也基本类似,唯一的不同在左边的求值树中 body′显示在 exit 节点的上面,以保证在规则的两边 body′在 body 后被求值。由于每一个规则在一个时间只处理一个结构,因此其余的绝大部分规则的证明并不复杂。

相关工作和结论 有几种其它的方法推理面向方面的程序,文[10]中定义了领域专用语言,定义了基于执行监控的横切表示方法。文[11,12]给出了以进程代数为形式化基础的面向对象语言。它使用了 CSP 的子集,将加入点(join points)作为同步集合,定义了两个程序之间的等价表示,并且展示了联结进程的正确性,然而由于它使用的是一种强制语言,缺乏对面向方面语言的适应能力。文[13]定义了动态加入点的语义模型,能表示 AspectJ 的特征,同时使用了简约的策略将联结过程转化为一个隐含的调用。这个转化容许使用已经隐含的调用语义去推理相关的程序,由于转化非常复杂,限制了其使用规模。文[14]将一些先置条件应用于面向对象重构,提出了一些新的从 Java 到 AspectJ 的重构条件。然而这些讨论仅仅将重构以及相关的条件应用作为全部内容,没有证明重构转化的等价性。

程序等价性的形式化证明并非易事。面向方面的重构仅对原型程序进行有限的改变,目的是使用面向方面的编程规则进行程序的优化,因此对重构前后程序等价性证明仅涉及编程规则的形式化描述,因而在此基础上的等价性证明无需针对整个编程语言语义及结构进行形式化处理,降低了描述和推导过程的复杂性。对 MCI 操作语义的面向方面扩展能简洁明确地表示类 AspectJ 中 advice 定义和联结过程,其基本方法能应用于其他面向方面语言的处理。观测等价性定义能满足重构目标评测的要求,可以作为编程规则等价性的形式化证明的基础,由于面向方面的重构可以看做是已证明规则的实例化,本文的工作为解决重构语义等价性形式化证明问题提供了可行的途径。

由于 MCI 语义缺乏模块化结构,本文对其进行的方面扩展工作也没有涉及这个问题,局限了语言的表达能力,因此对 MCI 的进一步扩展以支持对所有 AspectJ 结构的覆盖,是今后需要改进的重要工作。

参 考 文 献

1 任洪敏,钱乐秋. 构件组装及其形式化推导研究[J]. 软件学报,2003,14(6):1669~1677

2 陈广明,张立臣,陈生庆. 面向方面的实时软件开发方法[J]. 计算机科学,2005,32(7):189~192

3 Monteiro M,Fernandes J. Towards a Catalog of Aspect-Oriented Refactorings. In:4th Intl. Conf. on Aspect-Oriented Software Development, Chicago, USA,ACM PressMar. 2005

4 Opdyke W. Refactoring Object-Oriented Frameworks:[PhD thesis]. Urbana-Champaign, IL, USA, 1992

5 Roberts D. Practical Analysis for Refactoring:[PhD thesis]. Urbana-Champaign, IL, USA, 1999

6 Wand M,Kiczales G, Dutchyn C. A semantics for advice and dynamic join points in aspect-oriented programming. In: G. T. Leavens and R. Cytron, eds. FOAL 2002 Proceedings: Foundations of Aspect-Oriented Langauges Workshop at AOSD 2002, number 02-06 in Technical Report, Department of Computer Science, Iowa StateUniversity, Apr. 2002. 1~8

7 Aldrich J. Open Modules: A proposal for Modular Reasoning In: Aspect-Oriented Programming. In:C. Clifton,R. Lämmel, and G. T. Leavens, eds. FOAL'04 Proceedings: Foundations of Aspect-OrientedLanguages Workshop at AOSD 2004; Technical ReportCS Dept., Iowa State Univ., Mar. 2004

8 Barzilay O,Feldman Y,Tyszberowicz S,Yehudai A. Call and Execution Semantics in AspectJ. In:C. Clifton,R. Lammel, and G. T. Leavens, eds. FOAL'04 Proceedings: Foundations of Aspect-Oriented Languages Workshop at AOSD 2004;Technical Report CS Dept.,Iowa State Univ.,Mar. 2004

9 Borba P H M,Sampaio A C A,Cavalcanti A L C, Cornelio M L. Algebraic reasoning for object-oriented programming. Science of ComputerProgramming,January 2004

10 Cole L,Borba P. Deriving Refactorings for AspectJ. In:Proc. of the 4th International Conference on Aspect-Oriented Software Development (AOSD 2005), Chicago, USA,ACM Press,Mar. 2005

11 Andrews J H. Process-algebraic foundations ofaspect-oriented programming. In: REFLECTION '01: Proceedings of the Third International Conference onMetalevel Architectures and Separation of CrosscuttingConcerns, volume 2192. Springer-Verlag, Sept. 2001. 187~209

12 Gamma E,Helm R,Johnson R. Vlissides J. Design Patterns:Elements of Reusable Object-Oriented Software. Addison-Wesley, 1994

13 Douence R, Motelet O,Sudholt M. A formal definition of crosscuts. In: REFLECTION '01: Proceedings of the Third International Conference on Metalevel Architectures and Separation of Crosscutting Concerns, volume 2192, Springer-Verlag, Sept. 2001. 170~186

14 Fowler M,Beck K,Brant B,Opdyke W, Roberts D. Refactoring: Improving the Design of Existing Code. Addison-Wesley,1999