对象描述语言编译器的设计和实现*

张 波 黄 涛 傅远彬 邵丹华

(中国科学院软件研究所计算机科学开放研究实验室 北京 100080)

摘要 对象描述语言ODL(object description language)是组合框架软件SCOP(software construction=object+process control)环境的核心,ODL将类的抽象规范和具体实现相分离.ODL编译器支持这种分离,并在此基础上实现了语言独立;另外,在SCOP对象管理系统OMS(object management system)的支持下,在不同站点上开发的类及其所提供的服务可以共享,从而实现了位置独立.编译器还实现了动态联编的分离编译、基于对象库的复用和自动链接以及含参类的快速实例化.编译器以C语言为中间代码,生成的代码具有良好的可移植性.该文详细介绍了ODL编译器的关键设计技术和实现方法.

关键词 编译器,面向对象,继承,子类,类规范,动态联编,含参类.

中图法分类号 TP314

随着计算机系统结构的发展,面向对象技术的发展经历了语言为中心、数据库为中心到今天"操作系统"为中心的研究[1],即以互操作为主要目标的、语言独立、平台独立和位置独立的"操作系统"级支持.SCOP(software construction=object+process control)[2,3]组合框架软件即是为此而研制的.SCOP系统是一个支持应用软件系统开发的面向对象的组合软件构造环境.作为SCOP环境核心的对象描述语言ODL(object description language),将对象的抽象规范和具体实现分离开来,从而实现了对象实现的语言独立,即对象实现可以采用不同的语言或算法,这一点对client是透明的,client是通过对象的规范理解和使用server对象所提供的服务.另外,在SCOP对象管理系统OMS(object management system)的支持下,实现了位置独立,即通过OMS在不同站点上开发的server类可以共享.进一步,使用服务(操作)的client对象与提供服务的server对象可以位于不同的站点机,而这种位置的不同对client是透明的.ODL编译器将上述目标付诸实现.

1 ODL语言简介

对象描述语言ODL是在充分研究现有面向对象的程序设计语言OOPL(object-oriented programming language),特别是Smalltalk,C++,Eiffel等[4~8]的优缺点及OOPL特性的研究成果[9]而设计,是SCOP组合框架软件系统中的核心语言.它将对象规范和对象实现分离开来,为此提供了用于描述对象规范接口的描述语言和用于体实现的体实现框架,为方便起见,我们统称为对象描述语言ODL.ODL是C语言的扩充,与C语言完全兼容.ODL中,对象的静态结构和动态行为都封装在一个对象描述中.面向对象的程序设计强调的是对象的操作界面;面向对象的数据库强调的是对象的观察结构.ODL将两者有机地结合起来,即对象的静态结构可以通过对象的属性来观察;而对象的动态行为则由该对象所提供的操作来展示.ODL的类定义分为类规范(Specification)及体实现(Body)两部分:类规范用于对象的理解,即提供对象的对外语法接口和语义描述,从而使用对象无需了解对象的实现语言和实现算法.体是程序员通过体实现框架给出的类规范的具体实现;同一类规范可有多个体;体可用不同的语言、算法和数据结构来实现.ODL支持语言独立性,一方面通过语言映射,在其他语言编写的程序中可以操作ODL对象;另一方面,通过实现框架可以用其它语言实现ODL类规范.此外,ODL支持含参类、多态性、操作重载、动态联编等特性.有关ODL语言的细节,请参见文献[10].ODL编译器完全实现了ODL语言的特性.

---

* 本文研究得到国家自然科学基金和国家863高科技项目基金资助.作者张波,1972年生,博士生,主要研究领域为面向对象软件构造,软件设计方法学,软件工程环境.黄涛,1965年生,博士,副研究员,主要研究领域为软件工程,程序设计方法学,对象语义理论.傅远彬,1972年生,硕士,主要研究领域为面向对象软件构造,组合软件工程.邵丹华,1971年生,硕士,主要研究领域为面向对象的软件构造,软件工程环境.

本文通讯联系人:张波,北京 100080,中国科学院软件研究所计算机科学开放研究实验室

本文1997-05-15收到原稿,1997-06-20收到修改稿

ODL编译器概述

ODL编译器以C语言为中间语言,即ODL编译器将ODL源程序翻译成C代码,然后调用C编译器产生目标代码,在系统提供的运行库支撑下运行.采用这种方法的优点有两个:首先,C语言是一种使用广泛的语言,很多计算机平台都支持它,因此,用C语言作为中间代码使ODL编译器具有极好的可移植性;其次,面向平台的编译优化由C编译器完成,可使构造ODL编译器时精力集中于更高层的问题.

ODL语言是C语言的超集,其宏定义和文件嵌入与C语言一致.因此,对ODL程序的预处理,编译器是调用C编译的预处理程序实现的.为节省大量构造例行的词法分析和语法分析程序时间,我们采用LEX和YACC来生成词法分析和语法分析代码.同时,我们采用了对象管理系统(OMS)来管理编译器产生的结果;在此基础上,我们实现了分离编译、软件复用及自动连接等功能.

ODL编译器的主要工作有:词法分析、语法分析、类型表、符号表管理及C目标代码生成等.ODL编译器的结构如图1及图2所示.

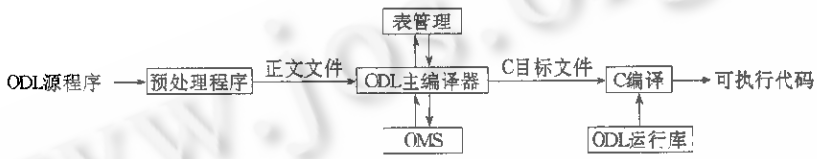


图1 ODL编译器结构

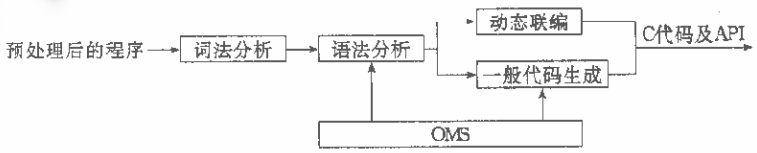


图2 ODL主编译器结构

ODL编译器的关键技术

3.1 类规范和多体

3.1.1 对象的表示及其内存映象

如图3所示,在ODL中,每一对象都有其唯一的标识,即对象标识.我们通过对象标识可以得到相应对象的对象描述符:对象类(及其范畴)id值,标识该对象所属对象类;对象体id值,表示该对象所对应的体,对象体在对象被创建时指定;数据区首址,表示该对象的内存映象中数据区的首址(其中实参编号的含义和作用见下文).

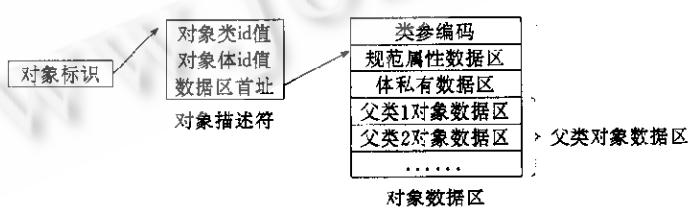


图3 对象的表示及其内存映象

3.1.2 属性和操作的翻译

类规范的核心部分是属性和操作.我们将属性集合翻译为一个结构,结构名和类名相关,而将属性翻译为该结构的成员.通过数据区首址指针和偏移量的计算,我们可以获得该类及其父类在内存映象中所对应的数据区的首址,因此可以很方便地通过类型强制对属性进行翻译.由于多体及多态性,偏移量的计算有时是动态的,即在运行时计算偏移量.

类的规范中只给出了操作的描述,而并未加以实现,操作由体来实现.ODL编译器将类的操作实现为C语言中的

函数,由于 ODL 提供多体以及操作重载,如直接使用操作名为相应 C 语言函数名,则同一个类的同一个操作在该类不同体中的实现之间,以及操作名相同而参数不同的操作之间存在名字冲突.为解决这一冲突,我们采用编码方法,对操作名进行编码,从而生成相应函数名,借助于编码后的函数名的差异区分不同的操作及解决冲突.

为使编码简洁有效,我们采用如下的编码方案:

* 系统给每个类分配一个 id 值,以区别不同的类,类 id 值的分配由 OMS 进行管理;
* 为同一类的每个实现体分配一个体号(体号 0 和 1 保留,以供实现动态联编使用),以区别同一类的不同体;
* 为同一类的每个操作分配一个操作号(这样,重载的操作之间有着不同的操作号),以区别同一类的不同操作.

这样,根据类 id 值、体号和操作号,生成对应于该操作的函数名.由操作翻译而来的函数具有如下同一的格式:

fname(this 指针,char **data*,*parameter_list*);

fname 由编码而来,this 指针指示调用该操作的对象实例,*parameter_list* 为用户定义的参数链.由于可能是子类对象调用该操作,我们用 *data* 表示在子类对象的数据区中相应于定义该操作的类的数据区首址.这样,只需在调用之前计算 1 次数据区首址,而不必根据 this 指针进行多次的计算,从而简化了翻译,并且提高了目标代码的质量.

我们采用动态链接技术把体的实现代码作成动态链接库.这样,当修改一个体时,引用到该体的代码就不必重新编译,这为对象库的修改和升级提供了简洁有效的途径.

3.2 动态联编

动态联编是实现多态性的基础,是面向对象的程序设计语言所必须面临的问题和关键技术之一.ODL 编译器生成动态联编函数以实现多态性,这项工作是在分析完用户源程序后进行的.因为这时我们才可以获得动态联编所必须依赖的两种信息:

* 完整的父子类关系,给出一个类,必须能找到其所有的子类,因为每个子类对象都能当作父类对象来使用;
* 父子类关系中的偏移量.

编译器将上述两种信息链入目标代码以支持多态性的实现.

3.2.1 多体给动态联编带来的问题

在 ODL 中,除了需要解决传统面向对象语言中动态联编所要解决的问题之外,还要解决由于引进多体而带来的新问题.ODL 中,体是作为对象实例的数据信息而不是它的类型信息而出现的.虽然每个对象实例必须属于唯一确定的体,但这些信息有时是不为编译器所知的,一个简单的例子:设 $f()$为类 A 的一个操作,有如下一段程序:

```
A a[2];
a[0]=creat(@A,@@body1);
a[1]=creat(@A,@@body2);
for(i=0; i<=1; i++) a[i].f( );
```

在循环内部,编译器无法确定所要调用的操作是在体 body1 中实现的操作 $f()$,还是在体 body2 中实现的操作 $f()$.

因此,在 ODL 中,动态联编必须考虑到多体的存在,即在操作调用的过程中,不仅需要根据对象实例的类 id,而且还要根据对象实例的体 id 以及父子类关系来动态地确定应该调用的操作.

3.2.2 ODL 操作调用和动态联编的实现

ODL 中,操作调用在缺省的情况下都是动态的,我们采用动态联编函数来实现这一功能.下面就 ODL 中的 3 种调用方式分别加以介绍.

ODL 提供以下 3 种操作调用方式:

(1) a.A::body::f(parameter_list);

(2) a.A::f(parameter_list);

(3) a.f(parameter_list).

第 1 种方式为限定了类和体的操作调用,对象实例 a 调用的是类 A 中定义的,在体 body 中实现的(也有可能是通过父子关系继承而来的)操作 $f()$,类 A 必须是实例 a 的静态类型的祖先.这种调用方式不具有多态性,不属于动态联编的范畴.

第 2 种方式为限定了类的操作调用,对象实例调用的是类 A 中的操作 $f()$,同样 A 也必须是 a 的静态类型的祖先,至于是类 A 的哪个体中实现的操作,则要在运行时动态地根据对象实例 a 的对象标识而确定.我们将该操作调用翻译为如下格式的动态联编函数的调用:

fname(a,parameter_list);

fname 根据类 A 的 id 值、体号 0(体号 0 和 1 被保留,实际的体号从 2 开始)及操作 f 在类 A 中的操作编号等生成.该动态联编函数的实现是在分析完用户源代码后生成的,其功能为根据 a 的对象描述符以及记录下来的父子类继承关系,确定所应实际调用的是类 A 的哪个体中的操作,并计算相应的偏移量.

第 3 种调用方式既没有类的限制也没有体的限制,因此所有信息都要在运行时动态地确定.我们将其翻译为如下的动态联编函数的调用:

fname(a,parameter_list);

其中 fname 根据作为实例 a 静态类型的类(设为类 A)的 id 值、体号 1 及操作 f 在类 A 中的操作编号生成.实例 a 所在的类必为类 A 的子类.该动态联编函数的实现也是在分析完用户源代码后生成,其功能为根据实例 a 的对象描述符,找到其祖先中最近定义操作 f 的类以及该类相应的体,计算偏移量,并调用相应的操作.

3.2.3 动态联编函数与目标代码的分离

动态联编函数的生成是在分析完用户源程序后进行的.我们把动态联编函数和目标代码的生成分离开来,当增加新的子类时,我们只需重新生成动态联编函数,而不需要重新编译已经编译过的源代码.这体现了增量编译的原则,对基于对象库的复用提供了支持,提高了效率.

3.3 含参类

3.3.1 ODL 中的含参类

在 ODL 中,含参类为一组具有相似特性的类提供了模板,通过不同实参进行实例化,即得到一组类定义.含参类提供了另一种代码复用的机制.

ODL 编译器在分析含参类的过程中,与分析非含参类一样,进行语法检查和类型检查并生成相应的"目标代码".此"目标代码"并不是真正的 C 代码,其中记录有形参的信息,在实例化该含参类的时候,根据记录下来的形参信息和实际类参作相应的处理,转换为实际的目标代码.

因此,ODL 编译器对含参类的处理经历了分析和实例化两个过程,尽可能抽取共性的部分在分析时处理,如语法检查、类型检查以及"目标代码"的生成等,实例化时,再根据实际类参进行转换,这样避免了对含参类的多次分析,使编译的效率得以提高.

3.3.2 类参编码方案

为了区别含参类的不同实例化,我们对类参进行编码,即将实参链按照编码方案转换成字符串,再根据该字符串产生唯一的编号.这样,以不同的实参对同一含参类进行实例化时,它们之间虽然具有相同的类 id 值,但具有不同的实参编号,我们可以根据实参编号区分同一含参类的以不同实参进行的实例化.为了形式上的统一和处理上的方便,我们保留了 0 作为非含参类的"实参"编号,即实参编号为 0 时,表明该类为非含参类.

3.3.3 含参类的实现

前面几节已经介绍了对非含参类的处理,但上述方法对含参类不够用,对前面所介绍的方法作如下的改进.

1) 对象实例的内存映象

我们用实参编号来区分同一含参类的不同实例化,如图 3 所示.我们为对象实例的内存数据区多分配 4 个字节,在数据区的头 4 个字节,记录实参编号,以此来区分不同实例化的对象实例.在创建对象实例的时候,为对象实例分配相应的内存,计算出实参编号,并将其写入内存区的头 4 个字节.

2) 属性和操作的翻译

前面我们将类规范中的属性集翻译为 C 中的结构,结构名和类名相关,引入含参类后,结构名不仅应该和类名相关,而且还应该和实参编号相关,并且是一一对应的.

与属性类似,对于规范所定义的操作,前面介绍过根据类 id 值、体 id 值和操作编号编码而成相应的 C 函数名.由于含参类的引入,在生成相应的 C 函数名的编码过程中,除了根据前面所述的 3 项外,还要根据实参编号.这样,就可以区别同一含参类的不同实例化的操作.

3) 动态联编

同一含参类的不同实例化之间可以有共同的父类,在这种情况下,我们必须能够动态地区分一个对象实例是哪一个实例化的对象实例,由于我们在内存映象中记录了实参编号,使这种区分成为可能.由此可见,在引入含参类后,我们不仅根据对象实例的类 id 值、体 id 值,还要根据对象实例的实参编号,来动态地确定所应该调用的函数.

3.4 基于对象库的复用和自动链接

在SCOP环境中,对象管理系统OMS用来存储、管理所有系统产生和需要的信息,所有信息均以对象的形式由OMS管理.ODL编译器通过OMS,使用户可以向对象库中添加对象类和体,也可以从对象库中取出对象类和体加以复用;另一方面,ODL编译器提供了基于对象库的代码自动链接.同时,OMS屏蔽了位置信息,使在不同的站点上开发的类可以共享,从而实现了位置独立.

3.4.1 对象类和体的复用

ODL编译器的动态联编技术、含参类实例化技术为基于对象库的类和体的复用提供了有力的支持,当用户要求将类和体入库时,ODL编译器将编译过程中记录下来的类和体的信息按照一定的格式存储在对象库中,并将类和体所对应的目标代码文件存放在OMS指定的位置.当用户需要复用对象库中的类和体时,只需提供类和体的名字,编译器通过和对象库的接口将指定的类和体的信息取出,建立相应的类和体的父子及继承关系,作为计算偏移量计算和生成动态联编函数的基础.同时,自动将相应的目标代码取出,自动链接到现有代码中.由于动态联编函数与类的目标代码是分离开来的,所以复用类时无需修改其目标代码,使自动链接能顺利实现.

对于含参类,由于可以用任何类型将其实例化,如果将每次实例化后的类都存入对象库中,会占用大量的库空间,这是行不通的.为了节省对象库的空间,我们仅保留其带有含参类信息的"目标代码",用户需要复用时,再根据实际参数进行相应的转换.

3.4.2 自动链接

在面向对象的程序设计过程中,实现一个类通常需要引用其他的类,有时甚至需要引用全局变量和全局函数等,而且有的引用是递归的,即A的实现直接或间接地引用了B,而B的实现又直接或间接地引用了A.

ODL在编译的过程中记录下了类和体、全局变量、全局函数等的引用和被引用关系.当某一个类及该类的体需要入库时,ODL根据记录下的引用和被引用关系,将该类和体所直接或间接引用到的类和体、全局变量、全局函数等(如果有的话)一起入库,并在OMS中建立起相应的引用和被引用关系,当用户复用该类和体时,编译器根据对象库中所记录的引用和被引用关系自动链进相关代码.

3.5 语言独立性

不同的程序员有其熟悉、擅长的语言,用熟悉的语言编程可以提高程序的质量和产量.基于抽象规范和实现体的分离,ODL实现了语言独立:通过语言映射,使用其它语言可以操作ODL对象;通过实现框架,ODL对象可以用不同的语言来实现.

3.5.1 ODL的语言映射

语言映射需要在所映射的语言中定义以下内容:ODL基本类型及构造类型、对象的引用、属性的访问以及操作的调用等.下面以C语言为例,简要介绍ODL的C语言映射.

1) 基本类型的映射

ODL类型	C类型
int	ODL_int
char	ODL_char
……	……

2) 构造类型的映射

ODL的构造类型如结构、联合、数组等直接映射为C语言的相应类型.

3) 对象的引用

假设要在C语言中使用ODL的类*A*,实现体为*B*,则其C语言映射为

```
typedef    ODL_Object    T_A_B;
```

C程序员可用类型T_A_B声明一个对象变量T_A_B *Obj*;

4) 属性的访问

假设类*A*有一属性:*char attr*,将其映射为C语言中的get和set函数

```
ODL_char * T_A_B_Get_attr(T_A_B p);
ODL_void T A B Set_attr(T_A_B p, ODL_char value);
```

则在C程序中用如下方法引用对象*Obj*的属性*Attr*及对其赋值

```
T_A_B_Get_attr(Obj);
```

T_A_B_Set_attr(Obj,a');

5) 操作的调用

假设类 *A* 有一操作:int op (short p1,char *p2),将其映射为 C 语言中的函数

ODL_int T_A_B_op(T_A_B p,ODL_short p1,ODL_char *p2);

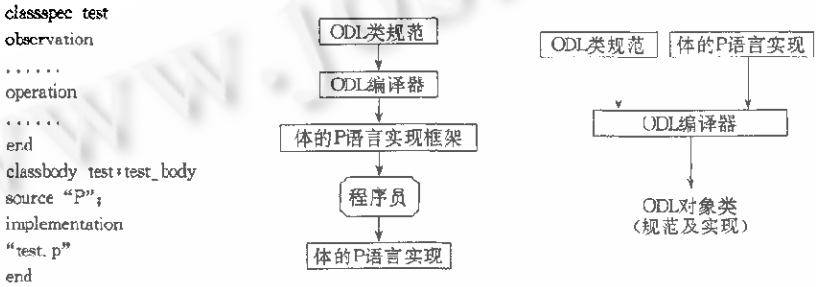
则在 C 程序中用如下方法调用对象 *Obj* 的操作 *op*

T_A_B_op(Obj,10,pCharValue2);

3.5.2 ODL 类规范的实现框架

ODL 类规范不仅可以有多个体,而且可以用不同的语言实现.ODL 编译器采用实现框架方式来完成类规范的多语言实现,即通过 ODL 编译器生成相应语言的实现框架,再由程序员填写符合规范的实现,然后再通过编译器将规范和实现合成一体.这种方式具有实现简单、不用扩充实现语言等优点.

实现过程如图 4 所示.首先写一类规范,在类规范的体实现部分用 source 字段指明实现所用语言,在 implementation 语句之后指明实现的文件名.然后用 ODL 编译器编译规范,编译器分析规范及实现体描述之后将规范映射成实现语言的实现框架.接着程序员完成相应的编程,最后由 ODL 编译器完成两者的合成.



(a) 类规范的内容 (b) ODL类规范到实现的映射 (c) 规范与实现合成过程

图4 ODL类规范的多语言实现

4 结束语

作为组合框架软件 SCOP 环境的核心,ODL 及其编译系统实现了以下两个主要的设计目标:(1) 支持抽象规范和具体实现分离的原则,抽象规范用来描述对象界面,具体实现可以使用不同的语言,在此基础上实现了语言独立;(2) 在 SCOP 对象管理系统 OMS 的支持下,在不同站点上开发的类及其所提供的服务可以共享,从而实现了位置独立.

另外,与同类工作相比,ODL 编译器在关键技术方面有以下主要特色:(1) 编译器采用 C 语言为中间代码,生成的代码具有良好的可移植性;(2) 采用动态联编函数实现动态联编,动态联编函数和目标代码相分离,这样,当添加子类时,无需重编译原有代码;(3) 把体的实现代码做成动态连接库,这样,当修改某一个体时,不用重新编译与该体相关的其他的类、体或全局函数;(4) 含参类是作为类型而出现的.编译器在分析含参类的过程中,与分析非含参类一样,进行语法检查和类型检查,并生成相应含有参数信息的"目标代码".在实例化含参类的时候,根据记录下来的形参信息和实际类参作相应的处理,转换为实际的目标代码.同 C++相比,ODL 对含参类的处理方法比较简洁,避免了对含参类的多次编译,提高了编译效率;(5) ODL 提供了基于对象库的自动链接及复用,用户只需在源程序中指出要复用的类的名字,由 ODL 编译器自动从对象库中取出该类的结构信息进行编译和链接,不需要提供该类的定义文本.而 C++是通过包含类定义的头文件进行复用的.与 C++相比,ODL 基于对象库的自动链接及复用既提高了效率,又给用户提供了极大的方便.

同时,ODL 编译器存在着一些问题,需在进一步的工作中加以改进.例如:由于 ODL 编译器将代码生成交给 C 编译去实现,给符号调试的实现带来了一定的难度.目前,我们的符号调试是借用 C 的符号调试器,但由于我们在翻译过程中采用了编码方案,因此,符号调试器的使用不太方便.

参考文献

1 Lewis Ted G. Where is client/server software headed. Computer, 1995,28(4):49～55

2 冯玉琳,黄涛,武小鹏.面向对象的组合软件工程研究.计算机学报,1996,19(3):237～240
(Feng Yu-lin, Huang Tao, Wu Xiao-peng. Research on component software engineering in object orientation. Chinese Journal of Computers, 1996,19(3): 237～240)

3 冯玉琳,黄涛,李京.面向对象的软件构造.软件学报,1996,7(8):129～136
(Feng Yu-lin, Huang Tao, Li Jing. Object oriented software construction. Journal of Software, 1996,7(8):129～136)

4 Goldberg A, Robson D. Smalltalk-80: The Language and Its Implementation. Addison-Wesley, Reading, MA, 1989

5 Bjarne Stroustrup. An overview of C++. ACM Sigplan Notices, 1986,21(10):7～18

6 Bjarne Stroustrup. The C++ programming language, 2nd Edition. Prentice Hall, Englewood Cliffs, NJ, 1991

7 Bertrand Meyer. Object-oriented software construction. Prentice Hall, Englewood Cliffs, NJ, 1998

8 Bertrand Meyer. Eiffel: the language. Prentice Hall, Englewood Cliffs, NJ, 1992

9 Peter Gronogo. Issues in the design of an object-oriented programming language. Structured Programming, 1991,12(1):1～15

10 中国科学院软件研究所计算机科学开放实验室技术报告,Dec. 1995 (ISCAS-LCS-95-16).
(Technical Report ISCAS-LCS-95-16. Laboratory of Computer Science, Institute of Software, The Chinese Academy of Sciences, Dec. 1995)

Design and Implementation of Object Description Language Compiler

ZHANG Bo　HUANG Tao　FU Yuan-bin　SHAO Dan-hua

(*Laboratory of Computer Science　Institute of Software　The Chinese Academy of Sciences　Beijing*　100080)

Abstract　The ODL (object description language) is one of the kernel parts of SCOP (software construction = object + process control) component software. Being language-neutral, ODL separate class specification form body implementation. Furthermore, with the help of the SCOP Object Management System (OMS), object class and their services could be shared across computers. That is, details about server location, the choice of implementation language and algorithm are transparent to programmers. The ODL compiler brings all such transparency into reality. Using C language as the target, the generated code is very portable. The authors outline the design of the ODL compiler and some key issues in implementing the compiler in this paper, including dynamic binding, auto-linking, quick-instantiation of template, language-mapping and implementation-skeleton etc.

Key words　Compiler, object-oriented, inheritance, subclass, class specification, dynamic-binding, template.