WCET分析中面向对象程序多态性问题的解决方法*)

姬孟洛[1] 李书浩[1] 秦 杰[2] 齐治昌[1]

(国防科技大学计算机学院 长沙 410073)[1] (河南工业大学信息科学与工程学院 450052)[2]

摘 要 用面向对象建模语言(如统一建模语言UML)设计并用面向对象程序设计语言(如C++)实现实时系统是实时系统开发领域的一个趋势,但面向对象的主要特征(如多态性)却使程序最差情况执行时间(Worst-Case Execution Time,WCET)更加难以分析。本文通过把UML设计信息引入WCET分析来解决此问题。考虑到UML关联关系描述了两个或多个具体类之间的对应关系,因此本文要求指定关联角色的多重性,并假定能够建立关联关系与其在程序中表示的对应关系。在已知关联角色多重性的基础上,本文计算特定循环的执行计数并确定在超(虚)类调用位置上每个具体类的对象个数,该循环使用超类变量遍历统一表示的关联角色对象。通过和Corti等人方法的结合,本文方法能够自动计算具有多态性特征的面向对象程序的WCET。实验结果表明,本文研究的情形在面向对象程序中普遍存在。

关键词 多态性,面向对象程序,WCET分析,实时系统,软件工程

A Solution to the Polymorphism of Object-Oriented Programs in WCET Analysis

JI Meng-Luo[1] LI Shu-Hao[1] QIN Jie[2] QI Zhi-Chang[1]

(Department of Computer Science, National University of Defense Technology, Changsha 410073)[1]

(School of Information Science and Engineering, Henan University of Technology, Zhengzhou 450052)[2]

Abstract It is a tendency for real-time systems to be designed with object-oriented modeling language, such as Unified Modeling Language (UML), and implemented with object-oriented languages such as C++. However, some object-oriented features like polymorphism bring about challenges for Worst-Case Execution Time (WCET) analysis. This paper presents a solution to cope with a kind of widely used dynamic binding by introducing UML design information into WCET analysis. Realizing that the association in UML is a relationship between two or more concrete classes, we emphasize the specification of the multiplicity of the roles of an association, and suppose the ease of establishing the mapping between an association and its representation in source code. Based on the multiplicity of the roles of association, we compute the execution count of a loop structure that traverses all the objects of the sub-classes a super class variable represents in the loop, and determine the number of objects of each concrete class in a super (virtual) class call site. By combining our method with the method of Corti and Gross, we can calculate the WCET of a program written in object-oriented languages with polymorphism characteristics. Empirical study shows that the cases we explore are broadly used in object-oriented programs.

Keywords Polymorphism, Object-oriented program, Worst-case execution time analysis, Real-time system, Software engineering

1 引言

实时系统与非实时系统的不同之处在于前者的正确性不仅要求输出正确,而且要求输出及时(既不太晚也不太早)。对实时系统而言,保证所有的任务在规定的时限完成非常重要。因此,实时系统必须具有可预测性,尤其是时间的可预测性。一般说来,大多数实时调度算法[1]都假定:在作调度分析之前,所有任务最差情况下的执行时间(Worst-Case Execution Time,WCET)是已知的。

时间正确性并不是实时系统开发中遇到的唯一问题。大型动态实时系统还需要复杂的嵌入式软件和硬件,它们难以理解、维护和修改。因为增加了依赖时间的行为,构建大型系统中遇到的问题在实时系统领域会更加严重。

另外人们逐渐认识到,为了评估实时系统的正确性和质量,需要精确地描述实时系统,并且需要在系统整个生命周期中支持严格的方法。在近几年学术界关于实时系统开发方法的几个提案中,面向对象方法被公认为最适合开发大型实时系统[2]。

虽然面向对象方法具有诸多优点,但是在实时系统中使用面向对象方法会导致系统的不可预测性和低效性[3],对于继承和多态尤其如此。

本文提出一个用于解决WCET分析中多态性(动态联编)问题的新方法。我们假定实时系统设计语言为标准面向对象建模语言——统一建模语言UML,实现语言为C++。我们强调在UML设计模型中使用关联关系(association relation)表示具体类之间的对象数目。通过从系统设计模型中抽取关联关系,并把此关系映射到程序中的相关表示,我们可以界定相关循环的迭代次数以及其中虚函数调用位置上各个具体对象的执行次数。这里,我们假定关联关系用统一结构的形式表示(比如数组和指针列表)且使用循环遍历该结构的

*)国家自然科学基金(No. 60303013)资助。**姬孟洛** 博士生,研究方向:实时系统分析、面向对象设计;**李书浩** 硕士,研究方向为实时系统;**秦 杰** 博士,研究方向为网络安全,Web信息处理技术;**齐治昌** 教授,研究方向:软件工程、计算机教育。

对象。通过把上述界限应用到只分析函数内程序的 Corti 和 Gross 方法[4](简称 CG 方法),我们可以分析具有多态性特征的面向对象程序。

2 UML 中的关联关系及其程序表示

2.1 UML 中的关联关系

UML 是标准建模语言。"UML Profile for Schedulability, Performance, and Time"[5]是 UML 在实时系统建模领域的外围扩展语言,它在使用 UML 已有的标准图元(diagram)基础上,提供用于实时系统中和时间、可调度性以及性能有关的标准图元,以便构建和这些方面有关的定量模型[5]。

关联是一种结构关系,它说明了一种事物的对象与另一种事物对象的连接。一般说来,给定连接两个类的关联,我们可以从一个类的一个对象导航到另一个类的另一个对象,反之亦然。

两个类之间的一般关联表示的是同等类之间的结构关系。"同等"意为两个类在概念上处于同一层次,同等重要。但对于表达"整体/部分"或者"具有"的聚集(aggregation)关系,较大的类是由较小的类组成的。聚集是一种特殊类型的关联关系。

一个类的对象是该类的实例。实例是具体事物的模型。类之间的关联关系表示了可以实例化的类之间的关系。关联关系中类的多重性(multiplicity)用于标注该类的直接实例。关联关系中角色(role)的多重性指明了有多少对象通过关联的实例连接,并表述为值的范围或者值。本文要求系统设计人员指明关联中角色的多重性,并建议系统设计人员把多重性说明为准确的数字或者范围。

举例来说,在图 1 和图 2 中,一个 picture 由 2 个 square、4 个 rectangle、3 个 circle 和 2 个 polygon 组成。类 rectangle、circle 和 shape 是类的子类,因此在类 picture 和类 rectangle 之间建立关联。类 picture 和类 shape 之间没有关联,因为类 shape 没有实例。同时,类 rectangle 在关联中的多重性是 4,不是 6,它只是表示了类 rectangle 的直接实例,因此不包括子类 square 的实例。

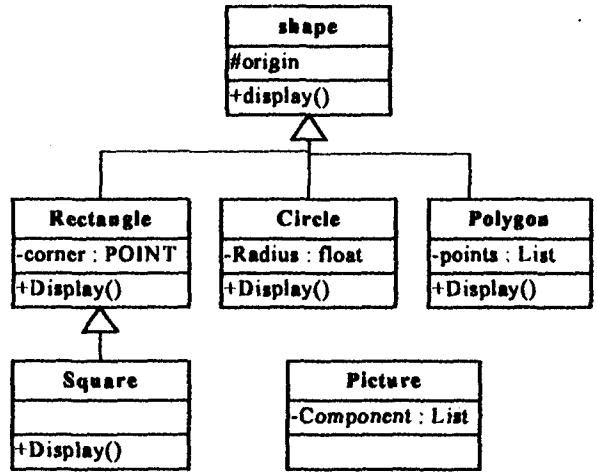

图 1 一个例子的类层次图

2.2 程序中关联的表示和访问

关联关系同样意味着从一种类型的对象导航到另一种类型。两个类之间普通的关联关系通常用对象的指针或引用(reference)实现[6]。但对于表示整体与多个部分之间的聚集关系,其实现就不那么简单。

在程序中,超类(尤其是抽象类)变量可以"表示"其子类的具体对象。因此,当访问这样的变量时,可以认为访问的是其子类的任意一个对象。

对于关联关系,尤其是聚集关系,部分类有可能具有共同的特性,从而它们会有一个共同的超(抽象)类,此超类抽象表示它们的共同特性。为了方便地从一个整体类对象导航到部分类对象,很自然的选择是以一种统一的方式(比如超类变量的数组或者指针列表)组织部分类对象。这种关联组织往往放置在"大"类变量中,当然也可以全局声明,此时把访问此组织的类看作是关联关系中的"大"(整体)类变量。举例来说,在图 2 中,Picture 的属性 Component 是 shape 的列表,它包含于 Picture 之中,从 Picture 通过 Component 可以访问任意一个 shape 对象。

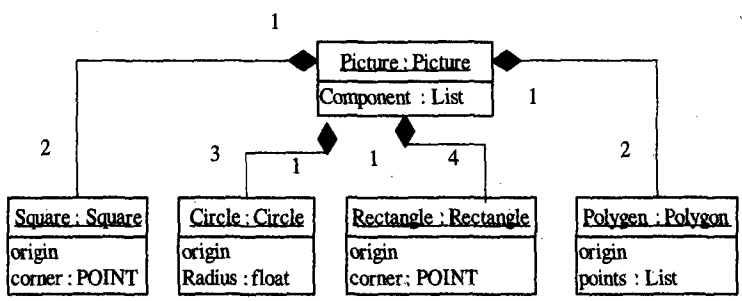

图 2 图 1 中类的关联关系

当需要从整体对象利用关联组织访问所有的部分对象时,自然会选择使用循环结构遍历所有的部分对象。循环结构的特征取决于关联组织的构造。举例来说,当 Component 是 shape 的数组时,Picture 会使用以下代码显示其组件:

```
① for (int i=0;i<Component. GetCount();i++) {
②    Component[i]. Display();}
```

当 Component 是 shape 的指针列表时,Picture 会使用以下代码显示其组件:

```
①Shape * pShape = Component. GetFirst();
②while (pShape->IsNotNull()) {
③    pShape->Display();
④  pShape = Component. GetNext(pShape);}
```

此时,仅仅通过分析源程序代码很难知道循环将迭代多少次以及在循环中每类对象将访问多少次。但是,当能够建立遍历循环和 UML 模型中关联关系之间的对应关系时,此信息可以从 UML 设计的系统模型中获得。

换句话说,可以认为"能够识别出来"的循环结构中的超(抽象)类变量用于遍历所有的子类对象。子类对象的个数由 UML 设计模型中关联关系的角色多重性说明。在使用超类变量作为循环变量的循环和 UML 模型中的关联关系之间有

一个对应关系。

由软件工程师负责建立以超类变量作为循环变量的循环和 UML 模型中关联关系之间的对应关系。为此，有可能会对 UML 模型中关联关系的实现有所限制。需要注意的是，时间可预测性是实时系统的基本要求，而不确定性是实时行为难以分析的一个主要原因，因此有所限制也是合理的。

3 利用 UML 中的关联关系分析程序结构

3.1 界定循环迭代次数遇到的问题

从 UML 设计模型中的关联关系可以得到两个类之间的多重性关系。假定两个类分别为 Class1 和 Class2，则它们的对应关系为 $m:n$，这里 m 是 Class1 的对象个数，n 是 Class2 的对象个数。为简单起见，限制 m 为 1。

当通过超类变量从一个对象遍历到另一个对象时，程序相应循环结构(称之为关联循环)的迭代次数是超类变量所代表的所有对象多重性的累加。但是，当导航路径包括多个关联关系通过一组嵌套循环结构的几个超类变量时，则很难确定嵌套中循环的迭代次数。下面举例说明。

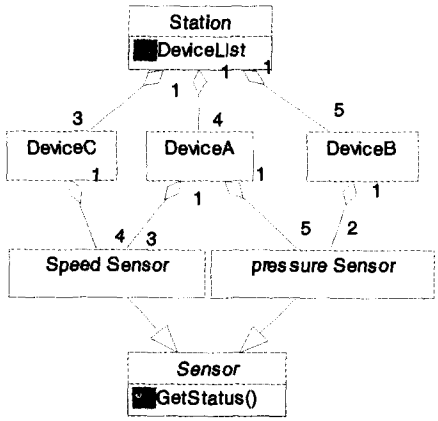

图 3 实时控制系统的类关系图

对于图 3 所示的实时控制系统，Station 对象包括 4 个 DeviceA 对象、5 个 DeviceB 对象、3 个 DeviceC 对象。一个 DeviceA 对象包括 3 个 SpeedSensor 对象、5 个 PressureSensor 对象；一个 DeviceB 对象包括 2 个 PressureSensor 对象；一个 DeviceC 对象包括 4 个 SpeedSensor 对象。类 SpeedSensor 和 PressureSensor 是 Sensor 类的子类。类 DeviceA、DeviceB 和 DeviceC 是类 Device 的子类，如图 4 所示。

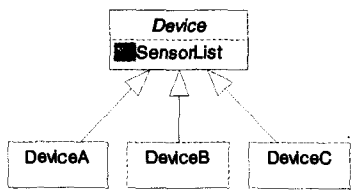

图 4 实时控制系统的类继承关系图

当通过遍历 Station 与 Device 之间以及 Device 与 Sensor 之间的关联从 Station 对象导航到 Sensor 对象时，比如说为了检查传感器的状态，相应的程序代码段如图 5 所示。

```
// pStation is pointer to object of class Station
①Device* pDevice= pStation->DeviceList.GetFirst();
②while (pDevice->IsNotNull()) { // iterate 12 times
③ Sensor* pSensor= pDevice->SensorList.GetFirst();
④ while (pSensor->IsNotNull()) {// iterate 54 times
⑤ pSensor->GetStatus();
⑥ pSensor=pDevice->SensorList.GetNext(pSensor);
⑦ }
⑧pDevice= pStation->DeviceList.GetNext(pDevice); }
```

图 5 通过使用两层关联遍历对象

此时，内层循环“while (pSensor->IsNotNull())”的迭代次数无法准确界定。当 pDevice 表示 DeviceA 时，其迭代次数为 8；当 pDevice 表示 DeviceB 时，其迭代次数为 2；而当 pDevice 表示 DeviceC 时，其迭代次数为 4；各不相同。

3.2 界定多重循环的迭代次数

虽然难以单独精确界定多重循环结构中一个内层关联循环的迭代次数，但完全有可能在整个多重循环结构之上界定一个循环的执行计数以及其中每条语句的执行计数，循环中每条语句的执行计数等于直接包含它们的循环的执行计数。举例来说，虽然无法精确界定图 5 代码段中内层循环的迭代次数，但能够确定在整个多重循环中内层循环的执行计数以及其中每条语句的执行计数。内层循环的执行计数为 Station 中所有传感器的总和，它是 Station 中 DeviceA、DeviceB 和 DeviceC 中的传感器的总和，该值为：$3\times4+4\times(3+5)+5\times2=54$。

为了计算每个嵌套的关联循环及其语句的执行计数，需要定义数据结构存储必要的信息，如图 6 所示。为每个(超)类循环变量定义 struct variable_info。对于多重循环结构，variable_info 中的 outerVar 是一个指向直接外层循环变量的指针。类似地，innerVar 是一个指向直接内层循环变量的指针。当没有这样的循环变量时，outerVar 和 innerVar 为 NULL。variable_info 中的 count 是该循环变量在循环中表示的对象数。ClassList 是该类变量所表示的所有类的列表(有可能包括它自己)，此列表通过索引访问。如果该类不是一个超类，或者在循环结构不是作为一个超类使用的，则 ClassList 只包括自己，此时其长度为 1。

为关联关系中的每个类定义 struct class_info。class_info 中的 id 是该类的标识。exeCount 是该类对象在循环中的执行计数。对于循环结构外面的对象，其 exeCount 为 1。sonCount 是该类直接包含的对象数。如果该类没有部分类，则 sonCount 为 0。

```
struct variable_info {
    int count; // the number of objects it represents
    struct variable_info *outerVar;
    struct variable_info *innerVar;
    struct class_info *classList; };
struct class_info {
    class_id id; // class identification
    int exeCount; // execution count of object itself
    int sonCount}; // count of objects it contains
// multiplicity of two classes
int Assoc[class_id_number][ class_id_number];
```

图 6 计算循环执行计数的数据结构

如前所述，根据 UML 设计模型中的关联关系，可以得到两个类之间的多重性对应关系。此多重性定义了一个列表{whole, part: multiplicity}，这里 whole 是指整体类(大的)，part 是指部分类(小的)，一个整体类对象对应了 multiplicity

个部分类对象。此类信息存储在数组 Assoc[classId1][classId2]定义的数据结构中，其中 classId1 和 classId2 分别是两个类的标识。如果两个类没有关联关系，则 Assoc[classId1][classId2]=0。

每个嵌套关联循环执行计数的计算包括两步。首先，通过分析程序标识出程序中的关联循环和相关变量(包括超类变量)，并建立每个超类变量的 variable_info 列表。通过程序分析或者直接使用 UML 设计模型可以得到类之间的继承信息，从而建立每个超类变量的 ClassList 列表。

第二步，计算每个嵌套关联循环的执行计数。嵌套关联循环最外层的类变量一定是非超类变量或者是没有作为超类使用的类变量，它也可以是“this”变量，即指向被分析的程序代码段所在的类，它还可以是关联循环绑定的从导航路径中截取的类变量。举例来说，图 5 代码段中最外层的类变量是 pStation，其 ClassList 只包括自己，其执行计数为 1。

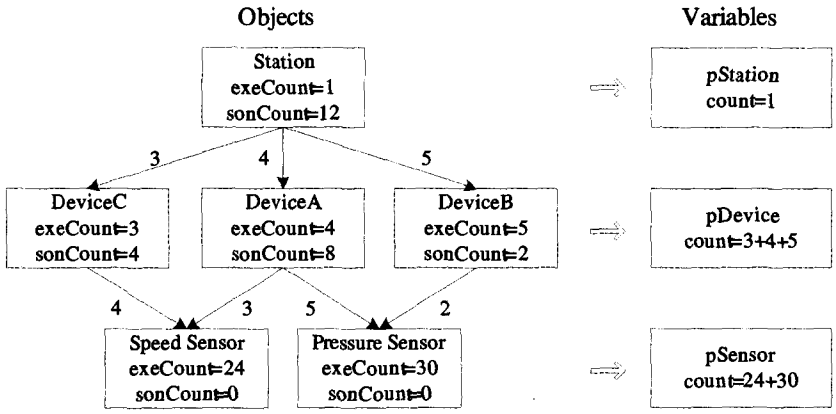

图 7　图 3、图 4 和图 5 中例子的计算

对于嵌套循环及其相应的类变量 cv，有：

$$cv.\text{classList}[i].\text{exeCount}=\sum_{j=1}^{n1}\text{Assoc}[cv.\text{outerVar}->\text{classList}[j].\text{id}][cv.\text{classList}[i].\text{id}]*cv.\text{outerVar}->\text{classList}[j].\text{exeCount} \quad (1)$$

$$cv.\text{classList}[i].\text{sonCount}=\sum_{j=1}^{n2}\text{Assoc}[cv.\text{classList}[i]][cv.\text{innerVar}->\text{classList}[j].\text{id}] \quad (2)$$

这里，$i=1,\cdots,m$，m 是 cv 可以表示的类的数量。$n1$ 和 $n2$ 分别是 cv 的直接外层和内层类变量 ClassList 的计数。

$$cv.count=\sum_{i=1}^{m}\text{classList}[i].\text{exeCount} \quad (3)$$

图 3、图 4 和图 5 中例子的计算如图 7 所示。

如前所述，最外层的类变量为 pStation，其 ClassList 只包括一个类，该类的 exeCount 为 1，因为其 outerVar 为 NULL。根据公式(2)，该类的 sonCount 为 3+4+5=12。因此，根据公式(3)，pStation 的 Count 为 1。

对于类变量 pDevice，它表示类 DeviceA、DeviceB 和 DeviceC。根据公式(1)，ClassList 中类 DeviceA、DeviceB 和 DeviceC 的对象执计数分别为 1×4、1×5 和 1×3，这里 1 是类 Station 对象的执行计数，4、5 和 3 分别是 Station 与类 DeviceA、DeviceB 和 DeviceC 的多重性。根据公式(2)，类 DeviceA、DeviceB 和 DeviceC 的 sonCount 分别为 4、8、2。根据公式(3)，变量 pDevice 的执行计数为 1×4+1×5+1×3=12，这正是图 5 中第一个循环的迭代次数。

对于类变量 pSensor，它表示类 SpeedSensor 和 PressureSensor。根据公式(1)，类 SpeedSensor 的对象执行计数为 4×3+3×4=24，这里类 DeviceA 和 DeviceC 对象的执行计数为 4 和 3，类 DeviceA 和 DeviceC 与 SpeedSensor 的多重性分别为 3 和 4。类 PressureSensor 的对象执行计数计算类似。根据公式(3)，变量 pSensor 的执行计数为(4×3+3×4)+(4×5+5×2)=54，这正是图 5 中内层循环的执行计数。类 SpeedSensor 和 PressureSensor 的 sonCount 为 0。

4　实验评估

这里给出 9 个任意选择的 C++ 程序的实验研究结果，如表 1 所示。表 1 中大多数源程序代码是公开的，只有最后一个来自于我们以前开发的实时通信设备监控项目，该项目用统一的组织方式和操作方法监控 15 个不同类型的设备。

表 1 中，project 是 Come from 所指示的源程序代码的名字，Classes、Super classes 和 Their sub classes 是它们对应的类的数量。sourceforge 是指可公开访问网站 http://www.sourceforge.net，Pande 是指文[7]中的源程序代码，这些代码也是公开的。

表 1 说明继承关系是面向对象开发中广泛使用的关系。表 2 把表 1 中的超类变量分为两种形式：Unified form(统一形式)和 Other form(其它形式)。Unified form 是指超类变量用统一的方式组织(比如数组和指针列表)并用来遍历其子类对象，它正是本文关注的形式。Other form 是指其它情况。Unified form 和 Other form 的总平均值用公式 $average=(\sum \text{Per.}\%)/n$ 计算，这里 n 是表 1 和表 2 所列项目数。

表 1　实验程序描述

No.	Project name	Classes	Super classes	Their sub classes	Come from
1	ffdshow	47	2	24	sourceforge
2	ariannexp	35	1	18	sourceforge
3	jobprocessor	2	1	0	sourceforge
4	bochs-2. 2. pre3\ iodev	74	1	26	sourceforge
5	vislcg	21	1	6	sourceforge
6	ocean	7	1	3	Pande
7	simul	7	3	4	Pande
8	Drawcli	22	1	3	VC samples
9	ComManager	45	1	15	self developed
	total	260	12+99=111		

表 2　实验程序中关联关系的表示

No.	Project name	Occurrence of superclass variable	Unified form		Other form	
			Num	Per. %	Num.	Per. %
1	ffdshow	3	2	66%	1	33%
2	ariannexp	1	1	100%	0	0
3	jobprocessor	1	1	100%	0	0
4	bochs-2. 2. pre3 iodev	9	0	0	9	100%
5	vislcg	13	1	8%	12	92%
6	ocean	3	3	100%	0	0
7	simul	1	1	100%	0	0
8	Drawcli	1	1	100%	0	0
9	ComManager	1	1	100%	0	0
	average			75%		25%

从表 2 可知，程序中的超类变量常常用统一的形式来表示其子类对象，虽然也有用其它形式表示其子类对象。从表 2 还可知，关联关系是类关系中重要关系，探究关联关系在设计和实现之间的关系是有意义的。

当用其它形式调用超类方法时，自动计算其 WCET 的方法可以采用取所有相应子类方法的最大时间，或者使用手工标注[7]。这两种方法要么不太精确，要么比较繁琐。

5　WCET 分析

本节说明如何应用上述方法在结合 CG 方法[4]基础上计算面向对象程序的 WCET。WCET 分析的目标是为一个程序段在指定的处理器上运行之前事先给出其最差情况的执行时间信息。WCET 分析通常包括两部分：程序流分析和处理器特征分析。程序流分析用于界定循环的迭代次数以及标识程序流等与支撑硬件无关的程序流信息。处理器特征分析用于为目标系统上目标代码的执行特征分类，比如指令缓命中或者流水。

CG 方法只用于函数内。通过把本文方法应用于 CG 方法，能够分析面向对象程序。CG 方法包括三个部分：结构分析、基本块迭代和指令执行时间计算。下面按照这三个部分说明如何把本文方法应用于 CG 方法。

5.1　结构分析

CG 方法把程序控制结构分解为彼此嵌套的层次子树。这些子树（也称为区域）表示代码模式或者语义构造，分为：blocks、if-then、if-then-else、while、natural-loop、repeat 等。结构分析算法在部分简化的图上不断识别区域。图中节点与预先定义好的区域进行比较，如果发现匹配，该区域就被看作是一个新的节点。识别过程持续到只剩下新发现的节点为止。图 8 说明了一个简单 C 代码的结构化分析。

我们扩展了 CG 方法使其包括方法调用和虚类方法调用。一般的方法调用很直接。虚类方法调用称为 V-Call。对于虚类方法调用，比如说图 5 代码段中的“pSensor->GetStatus()”，它有两个区域：每个具体类方法的调用和调用之前与之后的语句。V-Call 表示为 3 个部分：被调用的方法、具体类和 return。

“pSensor->GetStatus()”的 V-Call 如图 9 所示。需要注意的是，图 9 是过程间调用图，该表示方法受到文[9]的启发。

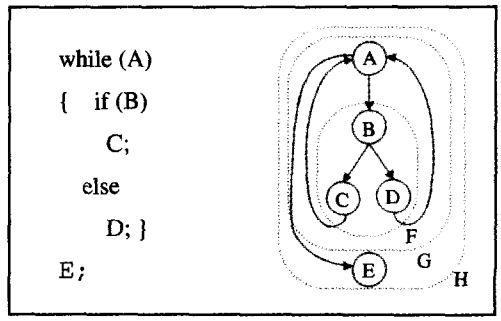

图 8　简单 C 代码的结构化分析

其中 F：=If-then-else(B,C,D)，G：=While(A,F)，H：=Block(G,E)

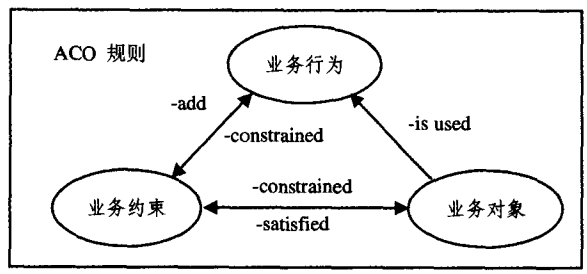

图 9　图 5 中 pSensor->GetStatus()的 V-Call

5.2　基本块迭代

在 Healy 方法[10]的基础上，CG 方法使用结构分析从循环头向每个程序块传播循环界定信息。通过把循环迭代次数和循环体内各个路径的不同迭代次数结合起来，CG 方法能够比较精确计算每个块的界定信息。

V-Call 的计算基于第 3 节的方法。一个语句的迭代次数是直接包含它的循环结构的迭代次数。对于超类变量 v 的 V-Call，如果 v 是关联循环用来遍历子类对象的变量，比如图 5 中语句 3 和 5，那么每个具体类执行 exeCount 次，exeCount 由 class_info 的相应域说明。如果 v 是 v 包含的关联循环外层的关联循环变量，换句话说 v 位于更深层的关联循环，比如说图 5 中语句 6，那么每个具体类 i 执行 subCount 次，subCount 由以下公式计算：

$$\text{sub Count}_i = \text{classList}[i].\text{exe Count} * \text{classList}[i].\text{son Count} \quad (4)$$

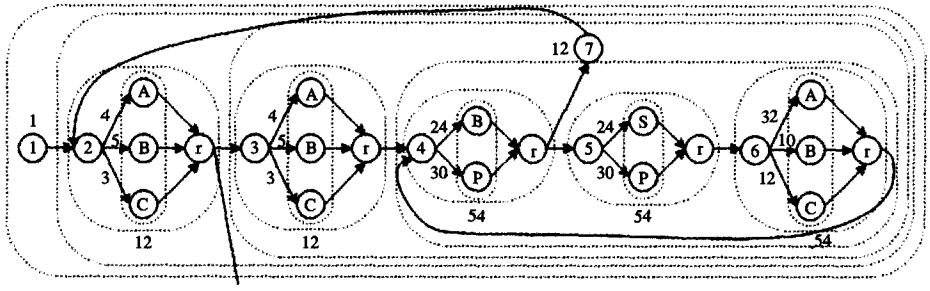

图 10　用块迭代次数标注的图 5 代码段的区域

这里 $i=1,\cdots,m$，m 为 v 可能表示的类的数量。

图 5 中的区域及其块的迭代次数可由图 10 部分说明。其中没有标注的块的迭代次数可直接得到。

在图 10 中，圈中的数字表示相应的语句序号，r 表示 return，A、B、C 分别指 DeviceA、DeviceB、DeviceC，S 和 P 分别指 SpeedSensor 和 PressureSensor。图 10 只标注了具体类的块和对应于语句的块的迭代次数。

5.3 指令执行时间

一旦每个基本块的迭代次数确定之后，就需要计算其中每条指令的执行时间（周期数）。CG 方法近似估算缓存（指令和数据）和流水线的行为。

为了精确估算一条指令执行之前的 CPU（流水和缓存）状态并精确计算其时间，应当模拟到达此指令的所有路径。因为完全模拟每一条可能的轨迹显然是不可能的，所以，根据一条指定指令在硬件（流水和缓存）上所具有的局部效应原理，减少被模拟代码的时间。其假设为：经过一定时间之后，一条指令的流水和缓存影响就不明显了。

对每个基本块 b 的第一条指令，只模拟进入路径的最后 n 条指令以便近似计算流水线和 CPU 执行单元的内容。n 的值说明为参数，通常在 50 到 100 条指令之间。试验显示，把 n 的值增大到超过此范围，应用程序的 WCET 也不会有变化[11]。

在模拟过程中，我们使用了比 CG 方法更简单的 CPU 模型以近似计算更新每个处理器单元的流水执行。这里使用 200MHz 的 Alpha 21064 微处理器，CG 方法使用的是 Pentium III 处理器。

按上述方法能够有效计算每条可能路径中每条指令的实际执行时间。每条路径根据其最大可能的执行次数加权。然后根据所有可能进入路径的加权累加和计算一条指令的执行时间。基本块中所有指令时间的累加就定义了基本块的执行时间。需要注意的是，V-Call 块的执行时间是其所有组件执行时间的累加。

在我们的实验中，在我们以前开发的 WCET 分析工具[12]的基础上，结合 CG 方法实现了前述方法。其结果是，对图 5 所示的程序代码，在 Alpha 21064 下的执行时间为 10，268 个周期。

6 相关工作

有关导出 WCET 分析的程序流信息已经做了很多工作[8,10,12~18]，其中有些是手工标柱[13,15,16]，有些是自动导出标注信息[8,10,12,14,17,18]。文[18]还处理面向对象语言，但没有涉及动态绑定。

Gustafsson[19]综述了面向对象程序的 WCET 分析现状并描述了计算面向对象程序的 WCET 时存在的问题，他建议取消或者限制动态绑定的使用。

Hu[8]第一次研究了 Java 语言的动态绑定问题。他使用标注系统确定动态绑定的不确定性：用户说明循环结构中虚类调用位置上各个子类方法的调用数目。其中最重要的标注是 A3 和 A4：

```
//@DefineScope(Scope Name)                    A3
//@ScopeWCET(Scope Name;nCount *
        UseWCET(…)+…)                         A4
```

DefineScope()标注即 A3 用于定义一个简单的或者嵌套的循环以建立 WCET。而 *Scope* WCET()标注即 A4 用来标注整个指定范围的 WCET 估算。

很显然，对于单重关联循环，比如 2.2 节的代码段，能够用此标注系统标注。但对于多重关联循环，比如图 5 中的代码段，使用此标注系统则难以标注。另外，手工标注工作量大且容易出错。

结论和未来工作 本文提出一种解决面向对象实时程序 WCET 分析问题的新方法，重点解决多态性的不可预测问题。我们强调两个具体类之间关联关系（尤其是聚集关系）中角色的多重性，并假定能够方便地建立设计中的关联关系与实现关联关系的程序代码之间的对应关系。我们考察了关联关系的角色对象用超类变量统一表示（如数组和指针）的情形。计算了使用循环中超类变量遍历其所表示的所有子类对象的循环结构及其中语句的执行计数，确定了超类调用位置中每个具体类对象的执行计数。通过和 CG 方法[4]结合，本文方法能够自动计算具有多态性特征的面向对象程序的 WCET。实验结果表明，本文所研究的情况在面向对象程序中是普遍存在的。

受 CG 方法[4]限制，本文方法目前只能应用于软实时系统。但我们相信本文方法能够应用到硬实时系统，这正是我们下一步要做的工作。下一步工作还包括放宽一些限制条件如关联关系中角色多重性之间的对应关系以及进一步研究多态性与模型及程序之间的对应关系。

参 考 文 献

1 Audsley N, Burns A, Davis R, et al. Fixed priority pre-emptive scheduling: an historical perspective. Real-Time Systems, 1995, 8(2-3): 129~154

2 Pereira C E. Are Object-Oriented Concepts Useful to Real-Time Systems Development? The International Journal of Time-Critical Computing Systems, 2000, 18: 89~94

3 Wedde H F. Critical Issues in Object-Oriented Real-Time Systems- A Guided Panel Discussion. The International Journal of Time-Critical Computing Systems, 2000, 18: 69~70

4 Corti M, Gross T. Approximation of the Worst-Case Execution Time Using Structural Analysis. In: 4th ACM International Conference on Embedded Software, Pisa, Sep. 2004

5 Response to the OMG RFP for a UML Profile for Schedulability, Performance, and Time-Revised Submission. OMG document number: ad/2001-06-14, OMG, Framingham, MA, USA, June2001

6 Douglass B P. Real-Time UML: Developing Efficient Object for Embedded System. Addison-Wesley Publishing Company, 2000

7 Pande H D, Ryder B G. Data-flow Based Virtual Function Resolution. In: Proc of the Third Intrnational Symposium on Static Analysis, 1996

8 Hu Yu-Shing, Bernat G, Wellings A. Addressing Dynamic Dispatching Issues in WCET Analysis for Object-Oriented Hard Real-Time Systems. In: Proc. 5th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing. Washington D. C., USA, April 2002. 109~116

9 Schordan M, Amme W. Virtual Method Resolution with Typed Alias Graphs. In: Proc. 8th International Workshop on Compilers for Parallel Computers. Aussois France, 2000. 151~162

10 Healy C A, Sjödin M, Rustagi V, et al. Supporting timing analysis by automatic bounding of loop iterations. Real-Time Systems, 2000. 121~148

11 Corti M, Gross T. Approximation of the Worst-Case Execution Time Using Structural Analysis. In: Proc. 4th ACM International Conference on Embedded Software (EMSOFT'04), Pisa, Italy, Sep. 2004

12 Ji Meng-Luo, Li Jun, Wang Xin, et al. An Automatic WCET Analysis Tool Based on Abstract Interpretation. Computer Engineer, to appear

13 Shaw A C. Reasoning about time in higher level language software. IEEE Transactions on Software Engineering, 1989, 15(7): 875~889

14 Blieberger J. Discrete loops and worst case performance. Computer Languages, 1994,20(3):193～212
15 Kirner R. The Programming Language wcet C:[Technical report]. Vienna, Austria: Technische Universität Wien, 2002
16 Stappert F, Ermedahl A, Engblom J. Efficient Longest Executable Path Search for Programs with Complex Flows and Pipeline Effects. In: Proc. Int Conf. on Compilers, Architectures and Synthesis for Embedded Systems, Atlanta, Georgia, USA, 2001
17 Healy C A, Whalley D B. Automatic Detection and Exploitation of Branch Constraints for Timing Analysis. IEEE Transactions on Software Engineering, 2002. 763～781
18 Gustafsson J, Ermedahl A. Automatic derivation of path and loop annotations in object-oriented real-time programs. Parallel and Distributed Computing Practices, 1998, 1(2)
19 Gustafsson J. Worst Case Execution Time analysis of Object-Oriented Programs. In: Proc. 7th IEEE International Workshop on Object-Oriented Real-Time Dependable Systems (WORDS'02). January2002. 58～63

(上接第 239 页)

我们分别分析图像中水平相邻、垂直相邻和对角线相邻的两个像素相关性，因此，我们选择 1000 对相邻的像素点，然后用(13)，(14)公式来计算其相关性：

$$\mathrm{cov}(x,y)=E(x-E(x)(y-E(y)) \tag{13}$$

$$r_{xy}=\frac{\mathrm{cov}(x,y)}{\sqrt{D(x)D(y)}} \tag{14}$$

上两式中，x,y 代表图像中相邻的两个像素点的灰度值。且有，在数字计算中，用到以下离散公式：

$$E(x)=\frac{1}{N}\sum_{i=1}^{N}x_i \tag{15}$$

$$D(x)=\frac{1}{N}\sum_{i=1}^{N}(x_i-E(x)(y_i-E(y))$$

$$C(x,y)=\frac{1}{N}\sum_{i=1}^{N}(x_i-E(x))(y_i-E(y)) \tag{16}$$

表 1 为图像 2(a)加密前后相邻的像素灰度值的计算结果，从计算结果可知，图像加密后相关性相差很大。

表 1 图像加密前后的相邻像素灰度值的相关性比较

	加密前图像灰度值相关性	加密后图像灰度值相关性
水平方向	0.9674	0.00187
垂直方向	0.945	0.00097
对角线	0.9158	0.00082

从表 2 可以看出，置换操作之后，加密图像和原图像的相邻像素相关性发生了极大的变化，密文图像有很好的抵抗差分分析和线性分析的能力，保密性高。

4.3 混乱性能分析

混乱，是指密文和明文之间的统计特性的关系尽可能地复杂化，这也就是混沌映射通过迭代，将初始域扩散到整个相空间。灰度替换设计(11)能有效地对明文进行混乱。

灰度替代操作(11)实质上是对像素点进行混沌的加密操作，改变该点的像素值，从而打破明文中各点像素值之间的统计关系。图 4 是初始值的微小改变。将混沌系统的初值改变量 10^{-4} 时，对图像 2(a)加密后其密文间差值的分布情况。可以看出，灰度替换设计(11)能够使得图像中几乎所有的像素点的灰度值发生变化，是个比较理想的灰度替代操作。

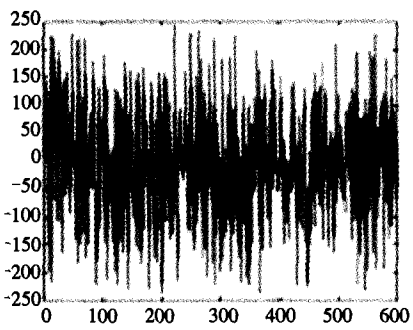

图 4 不同初值加密后的密文之间的差值

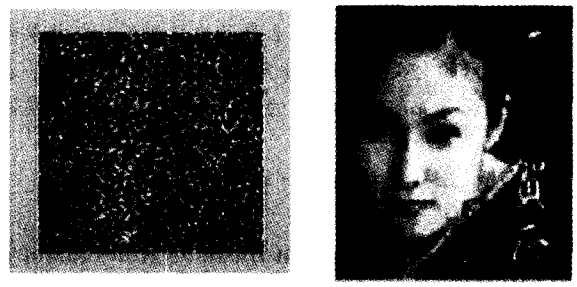

图 5 原图 2(a)的加密和解密图

4.4 算法正确性分析

由于算法中的像素置换操作和灰度替代操作都是可逆的，隐私算法具有可逆性，能够正确解密。因此在掌握正确密钥的情况下，能够对密文图像进行正确解密。图 5(a)和图 5(b)是图 2(a)的加密图和解密图。

结论 图像本身具有数据量大、像素点之间高相关性和高冗余性等特点，针对这些特点，本文应用离散混沌动力系统，设计了一种基于共轭混沌映射 Logistic 映射和 Tent 映射的图像加密算法。这是一种把灰度值替代和像素置换相结合的方法，实验表明，置换设计能有效地对明文进行扩散，灰度替换设计能有效地对明文进行混乱。算法符合密码学中加密算法准则，使得效率提高，便于实现。

参 考 文 献

1 Schneier B, 著. 吴世忠, 祝世雄, 张文政, 等译. 应用密码学. 机械工业出版社, 2001
2 Fridrich J. Image encryption based on chaotic maps. IEEE Int. Conf. Systm, Man&Cybernet ics Computational Cybernetics, Simulations, 1997. 1117～1120
3 Chuang T J, Lin J C. A new multiresolutiona pproach to still image encryption. Patern Recognition Image Anal , 1999, 9(3): 431～436
4 李昌刚, 韩正之, 张浩然. 一种随机密钥及"类标准映射"的图像加密. 计算机学报, 2003, 26 (4): 465～470
5 彭军. 混沌在网络信息安全中的应用研究:[博士论文]. 2003, 6: 27～40
6 樊春霞. 混沌保密通信系统的研究:[博士论文]. 2004, 10: 97～99
7 韦鹏程, 张伟, 杨华千. 一种多级混沌图像加密算法研究. 计算机科学, 2005, 32 (7): 172～175
8 Chang HKC, Liu JL. A linear quadtree compression scheme for image encryption. Signal Process Image Communation, 1997, 10 (4): 279～290
9 Chang C C, Hwang M S, Chen T S. A new encryption algorithm for image cryptosystems. J Syst Software, 2001, 58: 83～91
10 Cheng H, Li X B. Partial encryption of compressed images and videos. IEEE Trans Signal Process, 2000, 48(8): 2439～2451