基于文法简化和语句深度的静态结构模型嵌入式软件分析

李祯祥[1] 刘崇伟[1] 杨广益[2] 刘金硕[2]
(国网天津市电力公司电力科学研究院 天津 300000)[1] (武汉大学计算机学院 武汉 430079)[2]

摘 要 提出了一种基于文法简化和配合语句深度的静态结构模型的嵌入式软件分析方法。该方法设计了文法简化的词法分析和配合语句深度的语法分析,结合控制流/数据流分析,对嵌入式软件进行分析。以智能电能表开源软件作为案例,进行了 30 次实验,将人为插入的错误代码作为验证对象,同 PC-Lint 和 Splint 测试工具进行对比,本方法能够正确分析的概率为 91%,介于 PC-Lint 的 95%和 Splint 的 90%之间。该方法在解决了编译器对嵌入式平台不兼容问题以及保障正确率的情况下,提高了测试的效率。实验结果证明本方法适用于通过编译的 C(含嵌入式)程序。

关键词 嵌入式系统,静态结构模型,软件分析
中图法分类号 TP305 **文献标识码** A

Analysis of Embedded Software Based on Static Model with Simplified Grammar and Sentence Depth

LI Zhen-xiang[1] LIU Chong-wei[1] YANG Guang-yi[2] LIU Jin-shuo[2]
(Tianjin Electric Power Research Institute, Tianjin 300000, China)[1]
(Computer School, Wuhan University, Wuhan 430079, China)[2]

Abstract In order to solve the problem that the embedded software has the shortcoming of the platform dependence, this paper presented an embedded software analysis method based on the static structure model. Before control flow and data flow analysis, a lexical analysis/syntax analysis method with simplified grammar and sentence depth was designed to analyze the embedded software. This paper used the open source software of smart meters as a case, and used the artificial errors as the test objects, repeated 30 times. Compared with the popular static analyzing tools PC-Lint and Splint, the method can accurately orient 91% errors, which is between PC-Lint's 95% and Splint's 90%. The result indicates that the correct rate of our method is acceptable. Meanwhile, by removing the platform-dependent operation with simplified syntax analysis, our method is independent of development environment. It also shows that the method is applicable to the compiled C (including embedded software) program.

Keywords Embedded system, Static structure model, Software analysis

1 引言

随着嵌入式技术的发展和嵌入式设备的普及,基于嵌入式设备的软件编程已经成为了计算机软件开发的重要工作。然而嵌入式系统由于采用不同的核心硬件,例如 MCU 的型号差异等,导致与之对应的软件在开发细节上更是千差万别。因此,对嵌入系统软件的代码质量进行科学的管理更困难,同时由于嵌入式软件的真实使用环境也十分复杂,因此对嵌入式软件进行方便快捷的测试非常重要。综上所述,嵌入式软件的开发、测试与分析已成为了研究的热点。

软件测试可以使用人工或自动手段来运行或测定某个系统的过程,其目的在于检验它是否满足规定的需求或弄清预期结果与实际结果之间的差别[1]。通过软件测试,可以发现程序中的错误和执行程序的过程。通过分析错误产生的原因和错误的发生趋势,还可以帮助项目管理者发现当前软件开发过程中的缺陷,以便及时改进。

软件错误中有相当部分的错误是程序结构错误和变量定义使用错误。结构错误由程序中非正常的分支、死代码等导致,而变量的定义使用错误则是由变量的定义、使用以及其他操作等导致。对程序采用基于静态结构模型的控制流进行分析,可以较好地分析出程序的非正常的分支等结构错误。对程序进行数据流分析可以分析出程序变量的定义和使用错误。对嵌入式软件的静态结构模型分析,可以使用户在不运行程序的情况下快速定位潜在的软件故障问题,避免嵌入式软件运行的环境影响因素[2]。

目前,常用的软件静态检测工具有 PC-lint 和 Splint[3]。PC-Lint 是 GIMPEL SOFTWARE 公司开发的 C/C++软件代码静态分析工具,不仅能够对程序进行全局分析、识别没有被适当检验的数组下标、报告未被初始化的变量、警告使用空指针以及冗余的代码,还能够有效地指出程序在空间利用、运行效率上的不足。Splint 是一个检查 C 语言程序安全弱点和编写错误的程序,能对 C 语言程序进行多种常规检查。然而

本文受国网天津电力公司项目(KJ15-1-32)资助。
李祯祥(1983-),男,工程师,主要研究方向为计量技术,E-mail:13752288360@139.com。

它们都依赖于具体的开发环境：PC-lint 需要在其配置文件 std.lnt 中指明待测程序所使用的函数库、编译器、文件路径等；Splint 在带有 C 编译器的系统（如 UNIX）下能发挥较好的作用，而在 Windows 系统下需要集成到具体的开发环境中。

由此可见，上述静态测试软件均依赖于具体开发环境，存在源码编写与检测互相耦合的问题，为嵌入式软件的独立测试带来诸多不便。本文提出了一种独立于开发环境的基于静态结构模型的嵌入式软件分析方法，简化了词法和语法分析，并且针对简化的词法分析和语法分析，该方法构建了相应的控制流与数据流模型。

本文第 2 节为研究背景；第 3 节为基于模型的智能电能表软件分析的方法描述；第 4 节为实验；最后总结全文。

2 研究背景

2.1 词法分析和语法分析

词法分析为源程序形成目标程序的第一阶段，主要作用是读取源程序中的字符，利用词法分析器将它们形成类型为关键字、标识符、分界符、运算符和常数等的词法单元，输出一组词法单元序列。通常还需要对这 5 种类型进行进一步划分，形成具体的子类型，并用符号表记录。

语法分析是根据某种给定的形式文法，通过词法分析阶段产生的词法单元序列构建数据结构（通常为语法分析树），并进行语法检查。

2.2 控制流分析

控制流中的一个“节点”通常是一个条件、一个简单的语句或者一块语句（多个连续语句）；一个程序的控制流关系（Control Flow Relation）叙述了程序元素和它们执行的次序之间的联系[4]；对应于控制流关系的图被称为控制流图[5]。

控制流分析模型可以检测出如下的程序错误：

①死代码，即从程序入口进入后无法达到的语句；

②程序中含有未调用的函数；

③程序中因某个路径无法达到停机语句而产生的异常中断故障。

控制流分析步骤：

①确定所有程序元素；

②根据程序元素之间的相互关系得到控制流图；

③借助算法对控制流进行分析，找出存在的问题。

2.3 数据流分析

除了结构错误外，程序错误中往往存在着一定数量的数据错误。这些错误通常能通过编译，然而因变量的使用不当导致程序产生各种意想不到的故障。数据流分析以代数运算的方式对检查程序全局范围内的变量错误进行检查。程序中对变量的操作有定值与引用两种，对应的变量状态有未被定值状态、被定值状态和被引用状态[6]。

数据流分析的基本流程如下：

①确定程序中变量的定值与使用信息；

②基于控制流图，利用数据流方程，计算到达每个节点的入口和出口的变量集；

③通过变量集的迭代，借助算法对数据流进行分析，找出存在的问题。

数据流分析可以检测出如下错误：

①变量未定值而引用造成的故障；

②变量定值后未引用；

③变量重复定值。

3 基于文法简化和语句深度的静态结构模型的嵌入式软件分析

不同的嵌入式设备因采用不同的 MCU 而具有不同的指令集，编译时需要使用特定的编译器。常规的静态分析工具配合自身实现或外部提供的编译器，对通用 C 语言程序有较好的检测分析效果。然而，这些工具对嵌入式的 C 语言程序的检测能力有所下降。本方案在嵌入式软件编译通过的前提下，跳过常规的编译阶段，设计了基于文法简化和语句深度的软件静态分析模型，对嵌入式软件进行分析。由于本方法是基于源码能够编译通过这个前提，即被测源码不存在语法问题，因此可跳过语法检错阶段，采用逐行词法分析和语法分析生成程序控制流/数据流分析所需的词法序列，从而在降低对编译平台的依赖程度的同时提高分析效率。模型的处理过程如下。

步骤 1　逐行词法和文法简化的语法分析：

①对源码进行扫描、读取和预处理，产生代码序列；

②对代码序列逐行进行词法分析，产生一组词法单元序列；

③利用经过简化的文法对词法单元序列进行整理、识别和分类，同时为语句标记深度，产生语句序列。

步骤 2　控制流分析：

①提取语句序列中各语句的关键字与函数声明、调用信息，形成控制流节点；

②根据语句深度，为控制流节点确定控制流关系，产生控制流节点树；

③检测结构错误。

步骤 3　数据流分析：

①识别语句序列中变量的定值与引用信息；

②根据变量定值-到达和活跃性方程的迭代算法，对数据流问题进行求解；

③检测变量定值-引用错误。

模型的处理流程如图 1 所示。

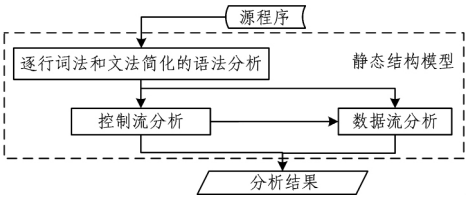

图 1　基于文法简化和语句深度的静态结构模型程序分析流程

3.1 逐行词法分析和文法简化的语法分析

本文所提出的模型采用逐行词法分析和文法简化的语法分析，即在词法分析阶段，对程序进行逐行词法分析，产生一组词法单元序列；在语法分析阶段，采用简化的文法识别程序

中所有符合 C 语法的语句，并对单行多语句进行拆分和对跨行语句进行合并，产生语句序列。最后，为每一条语句设置语句深度，如图 2 所示。

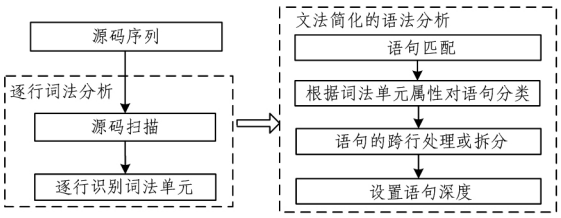

图 2　逐行词法分析和文法简化的语法分析

3.1.1　逐行词法分析

源码经过预处理中的头文件包含、宏定义替换和条件编译后，进入词法分析阶段。读取经过预处理的源码字符序列，扫描阶段对不需要生成词法单元的内容进行分词处理，即删除注释与压缩连续空白字符。

考虑到词法分析的主要作用是产生符合程序编写语言规则的语句，对于能通过编译的程序，大部分的语句是各自形成一行。对此，设计有穷自动机词法分析器（Finite Automaton Analyzer，FAA），逐行进行词法分析，输出一组词法单元序列。对于特殊情况，如跨行语句和一行含多条语句的情况，可在语法分析阶段进行语句的跨行合并、拆分。逐行词法分析识别过程如图 3 所示。

图 3　文法简化的词法分析过程

将上述过程用一个三元组表示：

$N=\langle IN, FFA, OUT\rangle$

其中，IN 为输入集，$IN=\{c \mid c\in \text{ASCII}\}$，读取时以字符的形式逐个传入 FAA，OUT 为终结状态，接收到一个词法单元，并将其输出为词法单元：

$OUT=\{ID, KEY, OD, UC\}$

其中，ID 为标识符（Identifier）接收状态；KEY 为关键字（Keyword）接收状态；OD 为运算符（Operator）和分界符（Delimiter）接收状态；UC 为无符号常数（Unsigned Constant）接收状态。

通过词法分析，接收上述类型的词法单元，并将其存入符号表。词法单元为一个二元组，用〈类型，标号〉表示，如关键字 if 可用〈KEY，if〉表示。每接收到一个新的词法单元，则在符号表中新建一条记录，并用新的标号来标记它。

3.1.2　文法简化的语法分析

通过遍历逐行词法分析所产生的词法单元序列，采用简化的文法匹配语句，形成语句序列。该过程所形成的语句整体上可分为两大类型：1）无需借助文法即可识别的语句：与控制流相关的分支语句（if、else、switch、case、default）、循环语句（for、while、do）、跳转语句（break、continue、return、goto）和大括号（'{'、'{'）、函数调用语句；2）通过文法识别的语句：顺序语句（变量的声明、赋值和使用、表达式等）、函数声明和函数定义语句。针对这两种类型，进行文法设计，构建语法分析树。定义数据类型集合如表 1 所列。

表 1　数据类型集合

集合名	集合含义	元素
B	基本类型关键字	void，int，float，doduble，char，idata，xdata，pdata，sbit，sfr
M	类型修饰关键字	unsigned，union，enum，auto，extern，static，const，volatile

对于函数语句，函数声明的一般形式为（括号内的参数名可缺省）：

返回值类型 函数名（类型 参数 1，类型 参数 2，…）

在不进行语法检错的前提下，使用如下简化的函数声明文法：

F→(M|ε)B(ε|＊)id(…

对于顺序语句，变量的声明文法如下所示（其中"ε"为空，"id"为标识符，"expr"为表达式）：

D→TV

T→AB

A→M(ε|A)

V→K(ε|,V)

K→id(ε|=expr)

以语句 static const u8 DBit=DS_MthComUse 为例，该变量声明语句对应的语法分析树如图 4 所示。

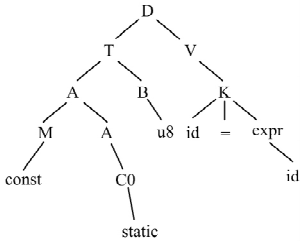

图 4　由变量声明语句生成的词法分析树

程序编写者出于代码的美观性或使程序更利于阅读，通常会将部分与变量操作相关的语句（如字符数量较长的数组声明语句、函数调用语句）分成连续的多行。对此，通过跨行文法，将满足上述情况的跨越多行的语句合并形成一行。将该过程称为语句的跨行合并处理。

根据 C 语言编写规范[7]，代码中每个左大括号应为单独的一行，右大括号所在行除跟随一个条件判断外，不应跟随其他内容，修改代码中不符合该规范的内容。另外，流程控制语句可能出现不带有大括号的现象。这种现象源自两种情况：1）C 语言规定的语法，如 case 和 default 等之后无需添加大括号，可跟随一个语句（块）；2）因代码编写者的编写习惯不规范而导致的分支语句之后未带大括号，并跟随一个语句（块）。针对这两种情况，对于简单的情况，即语句后跟随单挑语句，直接进行语句拆分；将另一种相对复杂的情形，即语句之后跟随的是一个语句块，延至控制流分析阶段进行处理。

上述的语句合并或拆分处理保证了每条语句的独立性。为语句设置深度，处理过程如下所示。

```
List〈Sentence〉senList;//语句序列
int deep=0;//当前语句深度
for(遍历词法单元序列){
    Sentence s;//建立一条语句类型变量
    if(token->name==='{')
```

```
        deep++;//语句深度递增
    else if(token->name== '}')
        deep--;//语句深度递减
    s.depth=deep;
    senList.add(s);
}
```

3.2 基于语句深度的控制流分析

基于上一节输出的语句，建立控制流节点，基于语句的深度对其进行遍历，产生控制流节点树[8]。算法实现的伪代码如下。

算法 1 控制流分析——控制流节点及树的生成

输入：语句序列：$s=\{s_0,s_1,\cdots,s_{n-1}\}$

输出：控制流节点

1. 建立并初始化基本块序列：$b=\{b_0,b_1,\cdots,b_{l-1}\}$
2. 基本块合成
3. 建立并初始化控制流节点序列：$n=\{N_0,N_1,\cdots,N_{l-1}\}$
4. for i=0,1,…,l−1 do
   if(N_i 为分支或循环语句类型) && (N_i→depth = N_{i+1}→depth) then
   将子语句节点深度+1
   end if
   for j=i−1,i−2,…,0 do
   if N_j→depth= N_i→depth−1 then
   N_i→parent=N_j
   break
   end if
   for k=i+1,i+2,…,l−1 do
   if N_k→depth+1= N_i→depth then
   N_i→children.add(N_k)
   else if N_k→depth≤N_i→depth then
   break
   end if
5. 为每个节点添加前驱和后继节点

通过上述算法，得到了一组控制流节点。通过分析每个节点的前驱和后继，可找出程序结构错误。

3.3 数据流分析

通过 3.1 节对变量的识别，可得到如下集合：

$def[N]$：在节点 N 内被定值，且在 N 内部后续无重复定值的变量集合。

$kill[N]$：节点 N 内被注销的所有变量定值集合。

$use[N]$：节点 N 内被引用的变量集合。

结合 3.2 节控制流得到的程序结构，利用数据流方程[9,10]，计算出变量的定值和使用异常。分析方法的伪代码如下。

算法 2 数据流分析——变量定值与引用错误的检测

1. change=1,N=N_0
2. while change=1 do
   change=0
   while N→successor !=null do
   temp=unionAll(N→precursor→out)
   if temp !=N→in then
   change=1
   N→out=union (N→def,diff_set (N→in,N→kill));
   N→in=unionAll(N→previous→out);
   N→inlive=union(N→use,diff_set (N→outlive,N→kill));
   N→outlive=unionAll(N→successor→inlive);
   end if
   N=N→successor
   end while
   end while
3. N=N_0
4. while N→successor !=null do
   N→UDchain=N→in (each assigned variable in N)
   N→DUchain =N→outlive(each referred variable in N)
   if N→UDchain=null then
   N→errorType=reference
   end if
   if N→DUchain=null then
   N→errorType=assignment
   end if
   N=N→successor
   end while

算法 2 中与集合相关的运算如下：

并集运算：$\mathrm{union}(A,B)=A\cup B$

差集预算：$\mathrm{diff\_set}(A,B)=A-B$

多重并集运算：$\mathrm{unionAll}(S)=\bigcup_{n\in s} n$

4 实验

为了验证方法的有效性，实验对模型进行了代码的实现，并以智能电表程序为例(其中一个电表的 MCU 类型是 C51，其文件结构如表 2 所列)。将人为制造的故障插入程序中(如表 3 所列)，利用本文提出的方法对故障进行定位，其 MCU 为 C51 的电表程序的测试结果，如表 4 所列。

为了验证程序的鲁棒性，对程序测试了 30 次(具体测试结果见表 5)。例如，表 5 的第 4 行，插入程序中的故障有 7 个，方法检测出的故障数有 6 个，包括 2 个结构错误和 4 个数据错误，其正确率为 86%。同时为了验证测试所提方法和效能，将方法和常用的静态分析工具 PC-lint 和 Splint 进行了对比(见表 6)。表 6 表明，在 3 种工具中，PC-lint 依靠调用外部的编译器有最高的正确率；Splint 内部集成了编译器，对由常规 C 语言编写的程序也有较好的测试效果，但在嵌入式 C 程序的测试中，正确率是三者中最低的，原因是 Splint 所集成的编译器无法识别特定的嵌入式指令和数据类型。由所提方法所实现的静态工具不对程序进行语法检测；而在嵌入式软件检测上，本方法采用特定的简化文法，对嵌入式 C 程序的检测正确率介于 PC-lint 的正确率和 Splint 的正确率之间。表中所列数据为分别进行 30 次实验后的平均正确率，以及对开发环境的依赖性的排查结果。

表 2 C51 程序的文件结构

文件名	文件类型	文件名	文件类型
mainproc.c	源文件	uartx.c	源文件
init.c	源文件	interrupt.c	源文件
LCD.c	源文件	event.c	源文件
verify.c	源文件	ATT7051.c	源文件
iokey.c	源文件	cpu_card.h	头文件
ex_func.h	头文件	variable4.h	头文件
ex_var.h	头文件	define4.h	头文件

表 3　插入程序中的错误信息

错误序号	文件名	文件类型	错误描述	错误类型	所在行号
1	mainproc. c	源文件	函数调用语句 Emu_Para_Verify()被注释	结构错误	78
2	init. c	源文件	变量 GPIO4 的赋值被注释	数据错误	13
3	init. c	源文件	变量 CKCFGWP 的赋值被注释	数据错误	113
4	init. c	源文件	else 分支无法到达(死代码)	结构错误	160
5	verify. c	源文件	函数调用语句 Veri_Etmr1_Io()被注释	结构错误	356
6	iokey. c	源文件	变量 DispErrorFlag 重复赋值	数据错误	81
7	iokey. c	源文件	循环变量 i 未初始化	数据错误	260
8	ex_var. h	头文件	声明变量 ADCFG1 但未进行初始化	数据错误	35
9	define4. h	头文件	数组 Ascii[]声明但未初始化	数据错误	19
10	ex_func. h	头文件	删除一条函数声明	函数声明	58

表 4　C51 程序的测试结果

结构错误	序号	数据错误	序号	故障总数	注入错误总数	正确率(%)
3	1,4,5	6	2,3,6,7,8,9	9	10	90

表 5　实验测试结果

实验号	测得数据				注入故障数	正确率(%)
	故障总数	结构错误	数据错误	正确故障数		
1	9	3	6	9	10	90
2	4	2	2	3	3	100
3	2	0	2	1	2	50
4	6	2	4	6	7	86
5	7	2	5	7	8	87.5
6	5	2	3	5	5	100
7	9	3	6	8	10	80
…	…	…	…	…	…	…
28	2	0	2	2	2	100
29	6	3	3	6	6	100
30	3	2	1	3	3	100
总计	149	57	92	138	151	91

表 6　测试效果对比

测试工具	正确率(%)	依赖的编译器	运行环境	是否画控制流图	是否检查语法错误	是否依赖环境
本方案	91	内部集成(简化的)	JVM	是	否	否
PC-lint	95	调用外部的编译器	Windows	否	是	是
Splint	90	内部集成	Windows/UNIX	否	是	是

实验结果显示,分析测得的故障总数为 149,实际存在的故障总数为 151,正确测得的故障总数为 138,正确率为 91%。对比静态检测软件 PC-lint 的 95%和 Splint 的 90%,本方法的正确率介于两者之间,但是本方法具有不依赖于嵌入式软件开发环境的优势。由此可见,本文提出的基于静态结构模型的嵌入式软件分析方法在得出被测程序结构的同时,能有效检测出程序中大部分的故障,简化了测试操作,为程序检测人员定位出了程序中潜在的代码隐患。

结束语　通过对嵌入式软件的静态结构模型的研究,在保证静态分析正确率的基础上,本文提出的基于文法简化和语句深度的静态结构模型嵌入式软件分析方法简化了分析过程,降低了分析工具对具体开发环境的依赖,提高了嵌入式软件分析的效率。基于该静态结构模型的嵌入式软件分析方法,能够提高嵌入式软件分析的自动化程度,减少嵌入式设备投入使用后因潜在隐患爆发而造成的损失。

参 考 文 献

[1] 兰雨晴. 软件测试[J]. 计算机系统应用,2003,24(12):66-68

[2] 刘佳欣. 嵌入式软件静态检测及自动化路径测试工具的研究与设计[D]. 广州:华南理工大学,2012

[3] 邓世伟. 嵌入式软件的测试方法和工具[J]. 单片机与嵌入式系统应用,2001,2(4):140-142

[4] 赵立平. 控制流分析[J]. 计算机工程与应用,1979(2)

[5] 周希. 基于静态分析的程序控制流图生成工具的设计与实现[D]. 广州:中山大学,2013

[6] 张广梅. 软件测试与可靠性评估[D]. 北京:中国科学院计算技术研究所,2006

[7] American National Standards Institute. ANSI C[EB/OL]. [2016-3-10]. https://zh. wikipedia. org/wiki/ANSI_C

[8] Kiczales G, lamping J, Mendhekar A. An Overview of Aspect[C]// J. Proc. 13th European Conference on Object-Oriented Programming, LNCS, Vol. 1241, Springer-Verlag, 2000:220-242

[9] 张广梅. 数据流相关软件故障的静态检测[J]. 计算机辅助设计与图形学学报,2005,17(11):2477-2482

[10] 王胜文. 采用数据流图的故障模型生成算法与应用[J]. 哈尔滨工业大学学报,2009,1(41):118-121

(上接第 489 页)

[11] Wedderburn R W M. Quasi-likelihood function, generalized linear models and the Gauss-Newton method[J]. Biometrika., 1974,61(3):439-447

[12] Xia T, Jiang X, Wang X. Strong consistency of the maximum quasi-likelihood estimator in quasi-likelihood nonlinear models with stochastic regression[J]. Statistics & Probability Letters, 2015,103(4):391-400

[13] Su F, Chan K S. Quasi-likelihood estimation of a threshold diffusion process[J]. Journal of Econometrics, 2015,189(2):473-484

[14] Hu H C, Song L. Quasi-Maximum Likelihood Estimators in Generalized Linear Models with Autoregressive Processes[J]. 数学学报(英文版),2014,30(12):2085-2102

[15] Liang K Y, Zeger S L. Logitudinal data analysis using generalized linear models[J]. Biometrika, 1986,73(1):13-22

[16] 陈夏,陈希孺. 广义线性模型极大拟似然估计的强相合性和渐进正态性[J]. 应用概率统计,2005,21(3):251-263

[17] 张三国,廖源. 关于广义线性模型拟似然估计弱相合性的几个问题[J]. 中国科学 A 辑:数学,2007,37(11):1369-1377

[18] 朱仲义. 异方差非线性模型中的拟似然估计的渐进性质[J]. 河海大学学报,1999,24(2):110-115

[19] Ahmed S E, Fallahpour S. Shrinkage estimation strategy in quasi-likelihood models[J]. Statistics & Probability Letters, 2012, 82(12):2170-2179

[20] 高启兵,吴耀华. 广义线性回归拟似然估计的渐进正态性[J]. 系统科学与数学,2005,25(6):738-745