一种过程间单子切片方法

张迎周[1,2,3,4],符 炜[1]

(1.南京邮电大学计算机学院,江苏南京 210003;2.广西可信软件重点实验室,桂林电子科技大学,广西桂林 541004;
3.江苏省无线传感网高技术研究重点实验室,江苏南京 210003;4.宽带无线通信与传感网技术教育部重点实验室,江苏南京 210003)

摘 要: 在现有的过程内单子切片算法基础上,提出基于回填待定标号的过程间单子切片算法:先以待定标号初始化子过程中开始处参数变量的切片;再对其进行过程内单子切片分析,据此可得相应参数间依赖关系;最后回填切片表中相应的待定标号,从而获得所需的过程间单子切片.算法充分利用了过程内单子切片的结果,相当程度上避免了重复计算,无需进一步构造诸如特征子图、连接语法等中间形式,同时通过参数间依赖避免了调用上下文问题.此外,文中算法保留了过程内单子切片算法的强语言适应性和组合性.

关键词: 单子切片方法; 模块单子语义; 过程间程序; 参数间依赖; 组合性

中图分类号: TP311 **文献标识码:** A **文章编号:** 0372-2112 (2013) 08-1457-05

电子学报 URL: http://www.ejournal.org.cn **DOI:** 10.3969/j.issn.0372-2112.2013.08.001

An Approach of Monadic Slicing for Interprocedural Programs

ZHANG Ying-zhou[1,2,3,4], FU Wei[1]

(1. *College of Computer, Nanjing University of Posts and Telecommunications, Nanjing, Jiangsu* 210003, *China*; 2. *Guangxi Key Laboratory of Trusted Software, Guilin University of Electronic Technology, Guilin, Guangxi* 541004, *China*; 3. *Jiangsu High Technology Research Key Laboratory for Wireless Sensor Networks, Nanjing, Jiangsu* 210003, *China*; 4. *Key Lab of Broadband Wireless Communication and Sensor Network Technology, Nanjing, Jiangsu* 210003, *China*)

Abstract: Program slicing is a technique for simplifying programs by focusing on selected aspects of semantics. This paper extends our previously presented intraprocedural monadic slicing to handle procedures. It presents backfilling labels based monadic approach to compute static slices of a program with call-by-value-result procedures. It first uses some given labels to initialize the monadic slices of the corresponding formal parameters at the beginning of a procedure; then analyzes this procedure through intraprocedural monadic slicing methods, whose slice result can be used to obtain the dependences among the parameters. It lastly obtains the final slice tables by backfilling the corresponding given labels. The algorithms in this paper make the best of the slice result from intraprocedural monadic slicing algorithms, without the need of a characteristic graph, a system dependence graph, or similar intermediate structure. They can also address calling-context problem through the dependence relations among the corresponding parameters. Furthermore, they reserve the excellent properties of compositionality and language-flexibility from intraprocedural monadic slicing algorithms.

Key words: monadic slicing methods; modular monadic semantics; interprocedural programs; dependences among the parameters; compositionality

1 引言

Weiser 等人曾发现,程序的某一个输出只与源程序中部分语句有关,删除其它的语句并不影响该输出的结果.他们把这种只与某个输出有关的语句构成的程序称为源程序的一种静态切片[1],并提出了基于控制流图(CFG)的程序切片算法.Ottenstein 等人[2]引入了基于程序依赖图(PDG)的图形可达性算法,以此来计算过程内程序切片.Horwitz 等人[3]随后提出了基于系统依赖图(SDG)的两阶段图形可达性算法(简记为 HRB 算法).二十多年来,人们对程序切片进行了广泛而深入的研究,取得了许多研究成果,使得它在软件调试、测试、维护、度量、程序并行化、软件逆向工程与再工程等方面得到广泛应用,因而也受到了广大软件研究、开发人员的高度重视[4,5,13,14].

虽然人们已提出了多种程序切片方法,但是,目前程序切片的算法还比较单一,基本上还是采用基于依赖图的可达性算法.为此,我们曾提出了一种新的语义切片方法:模块单子切片[6~8],它基于程序的模块单子语

收稿日期:2012-05-29;修回日期:2012-11-12;责任编辑:蓝红杰
基金项目:国家自然科学基金(No.60973046,No.61100135);江苏省普通高校研究生科研创新计划(No.CX10B_195Z,No.CXLX12_0479);南京邮电大学攀登计划(No.NY210009);广西可信软件重点实验室开放基金和江苏省"青蓝工程"优秀青年骨干教师项目

义.模块单子切片方法丰富了现有的程序切片算法,同时具有了较强的可扩展性和可重用性,支持组合式程序设计模式,反映现代程序设计语言(如 C ++ 、Java 等)中模块化设计特性.本文在此工作基础上,修订并扩展文[8]中的单子切片方法,使之能求过程间程序的单子切片,并以此展示我们单子切片算法的易扩展性.

2 实例语言 W 的单子语义

单子(monad)概念来源于哲学,是构成物质世界存在的最基本单位.1950 年,单子已被作为范畴论里的一种函子,但直到 1989 年才由 Moggi 将之引入到语义描述框架中[10].一般地,单子可形式化地表示成三元组(m, return m,bind m),其中 m 是类型构子,return m 和 bind m 是其两个基本操作,且满足三个规律(即左幺元、右幺元和结合律).为了使用单子来描述程序语义,常使用单子转换器来结合多个单子,并给一个单子增加新的计算信息.目前人们已设计了不少单子转换器[9,12,13],如下面的环境单子转换器 *EnvT* 等.在文[8]中,我们给出了如下的切片单子转换器 SliceT,以此来统一描述程序切片这一类计算.为讨论方便,本文仍考虑文[8]中的简单命令式实例语言 W,其抽象语法和模块单子语义见图 1.图 1 中,符号 *Fix* 表示不动点算子;*lkpEnv* 是环境 *Env* 的查找操作(函数);*updSto* 是存储 Loc 的更新操作;*getValue* 和 *putValue* 是 I/O 单子的基本操作[10,11].最终单子 *ComptM* 可由基本单子(如输入/输出单子 IO 或恒等单子 Id)和作用其上的单子转换器构成.为了从切片表中标号集合 L 获得一个符合 W 文法的程序切片,我们给出 W 语言的 $Syn(s,L)$定义(见图 2),其中 s 表示被分析的 W 源程序;$Refs(l.e)$表示所有出现在表达式 $l.e$ 中变量的集合;ε 表示空操作.

3 过程内单子切片

我们在文[6 ~ 8]中将程序切片这类计算抽象成切片单子转换器 SliceT,它独立于具体语言.通过切片单子转换器 SliceT,其它的单子(如状态单子 StateMonad 和环境单子 EvnMonad[14,15])可很容易地被转换成切片单子 SliceMonad.同样地,其它单子转换器 t 也可将 SliceMonad 转换成其它相应的单子.在文[8]中详细叙述了过程内程序的单子切片技术.单子静态切片算法(见算法 1)考虑程序最终点的单变量程序切片,对应切片标准为:$<p,v>$,其中 p 为程序最后语句点,v 为程序中某个变量.算法基本思想为:先将切片单子转换器组合到语义模块描述中,如 *ComptM*((*SliceT*(*StateT*(*EnvT*)IO,使其包含程序切片计算;然后按此语义描述逐句分析源代码并计算相应的静态切片,最后得到所要求的切片.

论域:ide:*Ide*(标识符);l:*Label*(标号);e:*Exp*(表达式);loc:*Loc*(存储);v:*Value*(值域)

抽象语法:S::= $ide := l.e \mid S_1; S_2 \mid skip \mid$ read $ide \mid$ write $l, e \mid$ if $l.e$ then S_1 else S_2 endif | while $l.e$ do S endwhile

语义方程:S::= *Stmt*→*ComptM*()　　E::= *Exp*→*ComptM Value*

$S[ide := l.e] = \{v \leftarrow E[l.e]; loc \leftarrow lkpEnv(ide, rdEnv); updSto(loc, return\ v)\}$

$S[S_1; S_2] = \{S[S_1]; S[S_2]\}$

$S[skip] = return()$

$S[\text{if } i.e \text{ then } S_1 \text{ else } S_2 \text{ endif}] = \{v \leftarrow E[l.e]; \text{ease } v \text{ of } tt \rightarrow S[S_1]; ff \rightarrow S[S_2]\}$

$S[\text{while } l.e \text{ do } S \text{ endwhile}] = Fix(\lambda f.\{v \leftarrow E[l.e]; \text{ease } v \text{ of } tt \rightarrow f \cdot S[S]; ff \rightarrow return()\})$

$S[\text{read } ide] = \{loc \leftarrow lkpEnv(ide, rdEnv); v \leftarrow getValue; updSto(loc, return\ v)\}$

$S[\text{write } l.e] = \{v \leftarrow E[l.e]; putValue(return\ v)\}$

图 1　实例语言 W 的模块单子语义

算法 1 中的切片数据结构 Slices 是由变量及其切片(标号集合)所构成的一张表(Hash 表),并包含三个基本操作函数:*lkpSli*、*updSli* 和 *mrgSli*,分别用来查找 Slices 中某变量对应的切片(标号集合)、更新 Slices 中数据和合并两个 Slices.此外,算法 1 中需要按式(1)计算加标表达式的 L',以此来捕获相应的控制依赖和数据依赖[7,8].

算法 1　过程内单子静态切片算法

输入:静态切片标准 $<p,v>$,**输出**:静态切片

1. 初始化标号集合 L 和 Slices 中标号集合.2. 将切片单子转换器组合到程序语义模块中.

3. 按模块单子方法逐条语句计算相应变量切片.得到包含所有变量切片的 Slices.

4. 由 $Syn(s, lkpSli(v, \text{Slices}))$返回最终静态切片结果.

$Syn(s,L) = \text{case } s \text{ of}$

"$ide := l.e$";	if $l \in L$ then "$ide := l.e$" else ε
"$S_1; S_2$";	$Syn(S_1, L); Syn(S_2, L)$
"*skip*":	ε
"read ide":	if $ide \in \{v\} \cup \bigcup_{i \in L} Refs(l.e)$ then "read ide" else ε
"write $l.e$":	if $v \in Refs(l.e)$ then "write $l.e$" else ε
"if $l.e$ then S_1 else S_2 endif":	if $(Syn(S_1, L) = Syn(S_2, L) = \varepsilon) \wedge (l \in L)$ then ε else "if $l.e$ then $Syn(S_1, L)$ else $Syn(S_2, L)$ endif"
"while $l.e$ do S endwhile":	if $(Syn(S, L) = \varepsilon) \wedge (l \notin L)$ then ε else "while $l.e$ do $Syn(S, L)$ endwhile"

图 2　$Syn(s,L)$的定义

$$L' = \{1\} \cup L \cup \left(\bigcup_{r \in Refs(l,e)} lkpSli(r, getSli) \right) \quad (1)$$

通过与西班牙奥维耶多大学 Labra 博士合作[12],我们开发了一个过程内单子切片器原型系统.该原型以程序的模块单子语义为基础,结合共代数(coalgebraic)和类属(generic)概念,支持增量式开发,其实现语言为 Haskell.

4　参数间依赖分析

影响子程序间依赖关系的主要因素有:①由于子程序的调用和参数传递而引起的数据依赖;②由于全局变量的作用而引起的数据依赖.子程序调用实质就是把控制转移到相应的子程序,在转入子程序之前必须把实际参数传递给被调用的子程序,即有一个子程序形式参数与实际参数匹配的过程.这里假设子程序的参数传递方式有三种:in、out 和 in out,分别对应于按值、结果、值 – 结果的调用方式.按 out 或 in out 方式传递参数时,允许在子程序内对形参进行修改,引起相应实参的变化,从而影响调用程序中相关变量的切片.全局变量可作为一个形参与实参同名的参数处理,如果在执行过程中此变量的值可能被改变,则作为一个 out 或 in out 参数,否则为 in 参数.为将函数和过程统一处理,把函数的返回值作为一个 out 参数处理.本文主要考虑 in out 参数传递方式.HRB 算法[3]中,为了构建 SDG 图,也需要计算参数间的依赖关系,主要是找出子过程的 actual_out 参数和 actual_in 参数间依赖关系.

5　过程间单子切片算法

本节将根据过程内的单子切片结果直接给出相应的参数(in out 型)依赖关系.具体地,我们先对调用语句 s 定义如下集合:①定义集 $Def(s) = \{x \mid x$ 是语句 s 中值被改变了的变量$\}$;②引用集 $Ref(s) = \{x \mid x$ 是语句 s 中引用的变量$\}$;③变量依赖集 $Dep(s, x) = \{y \mid y \in Ref(s) \wedge x \in Def(s)$且 x 的定义引用了 y 的值$\}$.

具体地,对某个调用语句 s,其相应的 $Def(s)$和 $Def(s, a)$可通过算法 2 获得.算法 2 中的 $Tproc$ 为由过程内单子切片方法首次获得的 P 最终点处单子切片表:即先以待定标号(如 $lx, ly, \cdots$)初始化 P 开始处的所有形参(如 $x, y, \cdots$)的单子切片,再对 P 进行过程内单子切片,然后得到最终单子切片表 $Tproc$(含待定标号).

算法 3 详细描述了基于回填待定标号的子程序间单子切片算法.算法 3 可得到某个子程序的首次被调后切片表 $Tproc$(含待定标号),这有利于算法 2 计算参数间依赖关系.从而,子程序调用的切片就变成子程序关于参数的切片,并进行相应的待定标号回填.

算法 2　参数间依赖算法

输入:对子程序 P 的调用语句 s,P 单次被调用后的切片表 $Tproc$(含用来初始化 P 参数切片的待定标号).

输出:s 的定义集 Def 和依赖集 Dep.

算法:　令 $Def(P) = \{x \mid lkpSli(x, Tproc) \neq \{l_x\}, x$ 为 P 中形参$\}$;--//所有被修改的形参

　if $x \in Def(P)$ then 设 a 为 x 对应的实参;

　$Def(s) = Def(s) \cup \{a\}$;令 $Dep(P, x) = \{y \mid ly \in lkpSli(x, Tproc), y$ 为 P 中形参$\}$;--//形参间依赖关系

　　if $y \in Dep(P, x)$ then 设 b 为 y 对应的实参;

　　$Dep(s, a) = Dep(s, a) \cup \{b\}$;

　　end if;

　end if;

算法 3　子程序间单子切片算法

输入:子程序 P　**输出**:P 单次被调用后的切片表 $Tproc\_P$,或者 P 被多次调用后的切片表 Tp_P

算法:if $Tproc\_P$ 为空 then　　--//P 为首次调用

　for P 的每个形参 x do 令 lx 为 x 的待定标号;$updSli(x, \{l_x\}, getSli)$;end for;

　　for P 的每条语句 s do if s 为对程序 Q 的施调语句 then 令 $Tproc\_Q$ 为 Q 单次被调用后切片表;

　　据 $Tproc\_Q$ 和算法 2 获取 s 处的定义集 Def 和依赖集 Dep;

　　if $a \in Def(s)$ then 设 x 为 a 对应的形参;for all $b \in Dep(s, a)$ do 设 y 为 b 对应的形参;

　　按公式(1)计算 $L_b = lkpSli(b, getSli) \cup \{l\} \cup L$;

　　$updSli(a, (lkpSli(x, Tproc\_Q) - \{l_y\}) \cup L_b, getSli)$;　　--//回填待定标号

　　end for; end if;

　　else 按算法 1 对 s 进行过程内单子切片分析;end if;end for;

　$Tproc\_P \leftarrow getSli$; else　　--//$P$ 被多次调用

　　$T \leftarrow getSli$;　　--//临时保存再次进入 P 时的切片表

　for P 的每个形参 x do $updSli(x, lkpSli(x, Tproc\_P) \cup lkpSli(x, T), getSli)$;

　if $l_y \in lkpSli(x, getSli)$且 y 为 P 形参 then

设 b 为 y 对应的形参,$Tcall$ 为 P 的施调点处切片表;按公式(1)计算 $L_b = lkpSli(b, Tcall) \cup \{l\} \cup L$;

　　$updSli(x, (lkpSli(x, getSli) - \{l_y\}) \cup L_b, getSli)$;　　--//回填待定标号

　end if;end for;$Tp\_P \leftarrow getSli$;

　end if;

为了扩展过程内的单子切片算法求含 in out 参数过程的 W 语言程序切片,我们需先引入按值 – 结果调用过程的抽象语法及其模块单子语义,如图 3 所示.然后给出该类过程的如下单子静态切片语义描述,其中"--"后面的为注释说明部分.切片算法已被证明能够捕获 PDG 图中的控制依赖和数据依赖,且单子切片结果与依赖图切片定义吻合[11].

6 实例分析

为进一步说明我们的过程间算法,下面来分析一个具体的程序(见图 4).对于图 4 所示程序,其调用图及分析次序见图 5.在对每个程序进行单子切片之前,要将其中所有子过程的切片表 *Tproc* 以给定(待定)的临时标号初始化,即要将 *Tproc_Add*、*Tproc_Inc* 和 *Tproc_A* 分别初始化为[(*a*,{l_a}),(*b*,{l_b})]、[(*z*,{l_z})]和[(*x*,{l_x}),(*y*,{l_y})].因子程序 *Add* 中没有任何调用语句,故可直接利用过程内单子切片算法,求得其切片表为图 6 的 *Tproc_Add* 部分.

类似地,可分析子过程 *A*,得到其单次调用后的切片表 *Tproc_A*(见图 6).在分析完所有子程序,开始分析主程序 *Main* 之前,当前的切片表为表 1 中所示.由 *Tproc_A* 及算法 3 对 *Main* 程序进行分析,并对 while 循环前后的切片表进行相应合并,直到循环稳定.本例中,while 语句处迭代分析 3 次后就稳定了,最终所得的切片表见表 2,它包含了程序中所有单变量的切片结果.

表 1 在分析 Main 程序前的切片表

Var	Labels
a	{*lx*, *ly*, 17, 18, 25, 31}
b	{*ly*, 17, 31}
x	{*lx*, *ly*, 17, 25}
y	{*ly*, 18, 25, 31}
z	{*ly*, 18, 25, 31}

论域:*arg*:*Arg*(参数);*proc*:*Proc*(过程)抽象语法:*D*::= procedure *ide*(*arg*1,⋯,*argn*) is *S*

S::= call *l*. *ide* (*ide*$_1$,⋯,*ide*$_n$)|⋯ *as before* ⋯ 语义方程:*D*:: *Dec*→*ComptM Env*

D [procedure *ide*(*arg*1,⋯,*argn*) is *S*]

= {let *proc* = λ *loc*$_1$⋯ *loc*$_n$. {ρ←*rdEnv*; ρ′←{*xtdEnv*(*arg*$_1$, *lkpSto*(*loc*$_1$), ρ);

⋯; *xtdEnv*(*arg*$_n$, *lkpSto*(*loc*$_n$), ρ)}; *inEnv* ρ′ *S*[*S*]; *updSto*(*loc*$_1$, *lkpSto*(*lkpEnv*(*arg*$_1$, *rdEnv*)));

⋯; *updSto*(*loc*$_n$, *lkpSto*(*lkpEnv*(*arg*$_n$, *rdEnv*)))}; *xtdEnv* (*ide*, *return proc*, ρ′}

S[call *l*. *ide* (*ide*$_1$, ⋯, *ide*$_n$)] = {*loc*$_1$←*lkpEnv* (*ide*$_1$, *rdEnv*);⋯; *loc*$_n$←*lkpEnv* (*ide*$_n$, *rdEnv*);

proc←*lkpEnv* (*l*. *ide*, *rdEnv*); *proc* (*loc*$_1$,⋯, *loc*$_n$)}

图 3 以按值-结果调用过程扩展 W 语言后的单子语义

```
program Main is          procedure A (x, y: in out) is   procedure Add (a, b: in out) is   procedure Inc (z: in out) is
    var sum: Int;            var x: Int;                     var a: Int;                      var z: Int
    var i: Int               var y: Int                      var b: Int                       begin
    begin                    begin                           begin                        31  call Add (z, 1);
4   sum := 0;            17  call Add (x, y);            25  a := a + b;                     skip
5   i := 1;              18  call Inc (y);                   skip                             end
6   while i < 11 do          skip                            end
7   call A (sum, i)          end
    endwhile
    end
```

图 4 一个过程间实例程序

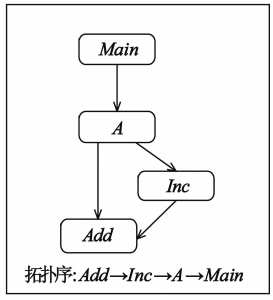

图5 图4程序的调用图及其拓扑序

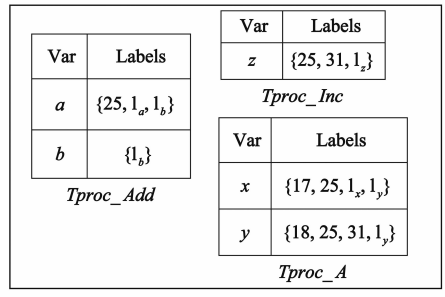

图6 子程序的单次调用后的切片表(含待定标号)

表 2　分析完所有程序后最终切片表

Var	Labels
sum	{4, 5, 6, 7, 17, 18, 25, 31}
i	{5, 6, 7, 18, 25, 31}
a	{4, 5, 6, 7, 17, 18, 25, 31}
b	{5, 6, 7, 17, 31}
x	{4, 5, 6, 7, 17, 18, 25, 31}
y	{5, 6, 7, 18, 25, 31}
z	{5, 6, 7, 18, 25, 31}

7　结论

本文提出了基于回填待定标号的过程间单子切片算法,该算法通过参数间依赖避免了调用上下文问题.与基于 SDG 两阶段扫描算法相比,本文算法因其无需构造 PDG 和 SDG,故可避免由于程序复杂、SDG 过于庞大而难以理解、实现时内存空间紧张等问题.另外,由于我们算法在很大程度上避免了重复计算,过程间单子切片的计算充分利用了已有的切片结果,从而大大提高了单子切片的计算效率,且最终可返回每个程序中所有单变量的单子切片.此外,由于我们的算法基于模块单子语义,因而有较强的语言适应性.

由于模块单子语义的可执行性保证了我们切片方法的可行性.从而模块单子切片技术能在一定程度上解决目前在动态切片、面向对象程序切片及并发程序切片等方面存在的问题.这也是我们下一步研究工作重点,并打算针对某个具体实际语言(如 Java)进行单子切片研究及其切片工具的开发.

参考文献

[1] Weiser M. Program slicing [J]. IEEE Transactions on Software Engineering, 1984, 16(5): 498 – 509.

[2] Ottenstein K J, Ottenstein L M. The program dependence graph in a software development environment [J]. ACM Sigplan Notices, 1984, 19(5): 177 – 184.

[3] Horwitz S, Reps T, Binkley D. Interprocedural slicing using dependence graphs [J]. ACM Transactions on Programming Languages and Systems. 1990, 12(1): 26 – 60.

[4] Tip F. A survey of program slicing techniques [J]. Journal of Programming Languages, 1995, 3(3): 121 – 189.

[5] Binkley D, Gallagher K B. Program slicing [J]. Advances in Computers, 1996, 43: 1 – 50.

[6] Zhang Y Z, Xu B W, Shi L, Li B X, Yang H J. Modular monadic program slicing [A]. 28th Annual International Computer Software and Applications Conference, COMPSAC 2004 [C]. Hong Kong, China: IEEE CS Press, 2004. 66 – 71.

[7] 张迎周,徐宝文.一种基于模块单子语义的动态程序切片方法[J].计算机学报,2006,29(4):526 – 534.
Zhang Y Z, Xu B W. An approach to dynamic program slicing based on modular monadic semantics [J]. Chinese Journal of Computers, 2006, 29(4): 526 – 534. (in Chinese)

[8] 张迎周,徐宝文.一种新型形式化程序切片方法[J].中国科学,E 辑:信息科学,2008,38(2):161 – 176.
Zhang Y Z, Xu B W. A novel formal approach to program slicing [J]. Science in China, Series E: Information Sciences, 2008, 38(2): 161 – 176. (in Chinese)

[9] Moggi E. An abstract view of programming languages [R]. University of Edinburgh, LFCS Report: ECS-LFCS-90-113, 1989.

[10] Wansbrough K. A modular monadic action semantics [D]. University of Auckland, Auckland, 1997.

[11] Liang S. Modular monadic semantics and compilation [D]. University of Yale, Yale, 1998.

[12] Zhang Y Z, Labra J, del Río A. A monadic program slicer [J]. ACM Sigplan Notices, 2006, 41 (5): 30 – 38.

[13] 陈平,韩浩,沈晓斌等.基于动静态程序分析的整形漏洞检测工具[J].电子学报,2010,38(8):1741 – 1747.
Chen P, Han H, Shen X B, et al. Detecting integer bugs based on static and dynamic program analysis [J]. Acta Electronica Sinica, 2010, 38(8): 1741 – 1747. (in Chinese)

[14] 戚晓芳,徐宝文,周晓宇.一种基于程序可达图的并发程序依赖性分析方法[J].电子学报,2007,35(2):287 – 291.
Qi X F, Xu B W, Zhou X Y. An approach to analyzing dependence of concurrent programs [J]. Acta Electronica Sinica, 2007, 35(2): 287 – 291. (in Chinese)

作者简介

张迎周　男,1978 年生于安徽巢湖.南京邮电大学计算机学院教授,博士,主要研究方向为软件工程、程序切片、Web 服务、函数式编程等.
Email: zhangyz@njupt.edu.cn

符　炜　男,1989 年出生于江苏建湖.南京邮电大学计算机学院信息安全系硕士研究生,主要研究方向为程序切片、Web 服务、函数式编程等.