反编译中数据类型自动重构技术研究

何 东 尹 青 谢耀宾 井 静
(信息工程大学信息工程学院 郑州 450002)

摘 要 类型重构作为反编译的关键问题,对程序的可读性及可理解性具有重要的作用。给出了汇编基础上数据类型自动重构的算法。对于简单类型,通过基于格的类型属性操作,用迭代算法来实现类型恢复;对于复杂类型,通过构建标记等价类来恢复结构化类型的框架,而后通过收集框架内可访问的偏移集合并利用简单类型恢复的算法对偏移对象类型进行恢复,从而推导出复杂结构类型。该算法是目前正在开发的类型重构工具的关键技术,它不仅能够准确地重构简单类型,而且能够准确地解析复杂类型,且准确率较高。

关键词 反编译,类型重构,类型依赖方程,等价类
中图法分类号 TP311.5 **文献标识码** A

Automatic Data Type Reconstruction in Decompilation

HE Dong YIN Qing XIE Yao-bin JING Jing
(Institute of Information Engineering, Information Engineering University, Zhengzhou 450002, China)

Abstract As one of the most significant modules of decompilation, data type reconstruction has an important role in readability and intelligibility. This paper proposed an algorithm for automatic type reconstruction from assembly code obtained from the MinGW GCC 3.4.5 compiler. The basic types are reconstructed using an iterative algorithm, which uses a lattice over the types' properties. The composite types' skeletons are recovered by establishing label equivalence classes, and the member variables by constructing the set of offsets for each composite type. The algorithm is the essential part of the tool being developed by authors, which not only reconstructs the basic type exactly, but also makes an active research into the hot issue aimed by all researchers currently and it has a favorable outcome.

Keywords Decompilation, Type reconstruction, Type dependence equation, Equivalence class

1 概述

反编译是编译的逆过程,其目标是将二进制代码转换成与之逻辑和功能等价的高级语言形式[1]。源程序中变量的结构类型决定了变量的存储空间和使用规则,然而源代码经过编译优化之后,有关程序的类型及调试信息都不复存在,转而以匿名的字节块信息代之,通过内存的分配访问形式及其相互间的依赖关系体现变量的类型信息。随着第三方软件的大量使用,出于安全的考虑,越来越多的软件需要进行安全分析,检测软件中是否含有漏洞以及恶意代码,而常规手段就是对比分析程序中的数据结构类型[2]。类型重构作为"源代码再现"的过程,能够有效增强代码的可读性,提高程序分析的效率。

高级C程序中变量的类型分为基本类型和复杂类型。其中基本类型包括整型、字符型、浮点型及指针型等;复杂类型包括数组、结构体及联合体等。类型重构的原则通常是先恢复基本类型,而后通过综合的分析方法恢复出复杂数据类型[3]。

本文给出了数据类型的自动重构算法,其分析对象是高级C程序经MinGW GCC 3.4.5编译后的汇编代码,目标是具有高级类型结构的程序代码。其大体流程如图1所示,即由汇编程序获取分析对象并送入基本类型重构模块,经过该模块处理以后,产生3种可能的结果:1)已识别的类型;2)未识别的类型;3)指针类型(间接类型)。已识别的类型对象对应于源程序中能够唯一确定的简单变量类型;未识别的类型对象将在下次迭代中继续送入基本类型重构模块进行求解;指针类型所对应的间接对象会被送入复杂类型重构模块,通过该模块处理后会产生复杂类型的框架以及复杂类型内部成员变量对象(结构体成员变量或数组元素),对于新生成的对象继续以迭代的方式送入基本类型重构模块进行求解。该算法最终结束于对象集不再发生变化的条件,即达到一个固定点。若该对象集不为空,则表示产生了类型冲突,需要以显示的类型转换对结果进行表示。但是通常情况下,算法会以空集结束,因为所有的类型都是基于有穷格的单调函数操作。

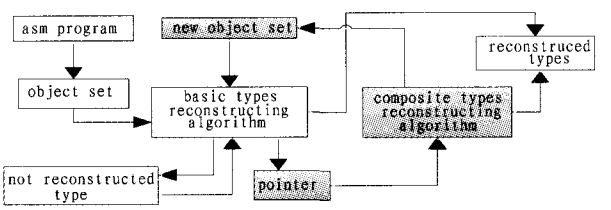


图1 数据类型重构算法流程图

到稿日期:2011-06-29 返修日期:2011-09-28 本文受国家"863"计划基金(2007AA01Z483),河南省科技攻关(092101210503)资助。
何 东(1984-),男,硕士生,主要研究方向为软件逆向工程,E-mail:preperdong@sina.com;尹 青(1967-),女,博士,副教授,硕士生导师,主要研究方向为软件逆向工程;谢耀宾 男,讲师,主要研究方向为软件逆向工程;井 静 女,博士生,主要研究方向为软件逆向工程。

2 基本数据类型重构

研究的基本类型如下：设 $\Omega_0=\{$unsigned char, char, unsigned short, short, int, unsigned int, long, float, unsigned long, double, long double, void$\}$, $\Omega_1=\{$void*$\}$。其中 void 作为特殊符号表示空类型，void* 表示通用的指针类型。则欲重构的基本数据类型集 $\Omega=\Omega_0\cup\Omega_1$。用如下的 3 个属性分别对 Ω 中每一个数据类型 T 进行刻画：*core*(*core*∈{integer, float, pointer})，*size*(*size*∈{1, 2, 4, 8}) 和 *sign*(*sign*∈{signed, unsigned})。则每个给定的类型可用三元组表示为 $T=\langle\tau^{core},\tau^{size},\tau^{sign}\rangle$。比如，上述的空类型 void=$\langle\Phi,\Phi,\Phi\rangle$，unsigned short=$\langle\{$integer$\},\{2\},\{$unsigned$\}\rangle$等。因此，基本数据类型重构的主要任务就是：1）判断变量的 *core* 属性(pointer, float, integer)；2）判断变量所占内存的大小；3）对于整型变量，判断其是有符号数还是无符号数。

2.1 类型分析对象及约束条件

汇编程序中含有潜在数据类型信息的对象包括寄存器、固定内存单元(全局变量)、堆栈内存单元(局部变量及子程序参数)、间接对象引用等，将所有这些对象统一用 OBJ 表示。因为变量之间存在相互依赖的关系，通过某一个(某些)变量的类型能够推导出另外一些变量的类型，所以可以建立用于类型求解和推导的类型依赖方程，其通用形式如下：$obj_p:t_{n,1}\odot obj_q:t_{n,2}\Rightarrow obj_r:t_{n,0}$，其中⊙对应于汇编指令的操作。同时针对方程中出现的每个 *obj* 指定一个相应的类型元组 t，写成 $obj:t=\langle\tau^{core},\tau^{size},\tau^{sign}\rangle$，即 *obj* 所对应的重构类型为 t。

为了更加准确地推导变量的类型，需要为对象添加适当的约束，这些约束受处理该变量的指令、存储寄存器及程序环境等的影响，在后续类型推导时将作为流函数使用[4]：C_{reg} 表示寄存器约束，影响变量类型的 *core* 和 *size* 属性。比如，以 dx 为基础的导出类型 T，其属性 $\tau^{core}=\{$short$\}$, $\tau^{size}=\{1,2\}$；C_{cmd} 表示指令约束，影响整个类型的属性。比如，*movswl*%*dx*,%*eax*，则对于 dx 其隐含的信息是 $\tau^{core}=\{$short$\}$, $\tau^{size}=\{2\}$, $\tau^{sign}=\{$signed$\}$；C_{flag} 表示标志位约束，影响 *sign* 属性。比如，*cmp eax*，0 与 *jae Label*，则根据跳转指令可得 $\tau^{sign}=\{$unsigned$\}$；C_{env} 是环境约束，影响整个类型的属性。另外为了增强类型综合推导能力，定义类型运算 join 函数'⊔'[5]，对于不同类型 $t_1=\langle\tau_1^{core},\tau_1^{size},\tau_1^{sign}\rangle$ 和 $t_2=\langle\tau_2^{core},\tau_2^{size},\tau_2^{sign}\rangle$，其运算规则为 $t_1\sqcup t_2=\langle\tau_1^{core}\sqcup\tau_2^{core},\tau_1^{size}\sqcup\tau_2^{size},\tau_1^{sign}\sqcup\tau_2^{sign}\rangle$。

2.2 类型推导规则

根据指令的不同形式，可按照如下 4 类指令操作确定类型依赖方程：一元操作、二元操作、拷贝操作和间接引用操作。对于不同的方程，其各自的类型推导规则如下：

对于一元操作，其类型依赖方程为 $\Delta e_1:t_{e_1}\Rightarrow e_{res}:t_{res}$，其中 Δ 为一元操作，则类型推导规则为 $t_{res}=t_{e_1}$。

对于二元操作，其类型依赖方程为 $e_1:t_{e_1}\odot e_2:t_{e_2}\Rightarrow e_{res}:t_{res}$，则类型推导规则如下：

$$e_1:t_{e_1}\odot e_2:t_{e_2}\Rightarrow e_{res}:t_{res}\vdash$$

$$\frac{\text{pointer}\in\tau_{e_1}^{core},\text{pointer}\in\tau_{e_2}^{core}}{\text{integer}\in\tau_{res}^{core}},$$

$$\frac{\text{pointer}\in\tau_{e_1}^{core},\text{integer}\in\tau_{e_2}^{core}}{\text{pointer}\in\tau_{res}^{core}},$$

$$\frac{\text{float}\in\tau_{e_1}^{core},\text{float}\in\tau_{e_2}^{core}}{\tau_{res}^{core}\in\text{float}}$$

定义最大上界交运算⊓：设 S_1 和 S_2 是两个整数集，$S=S_1\sqcap S_2=\{s\mid s=\max(s_1,s_2),\forall(s_1,s_2)\in S_1\times S_2\}$。比如集合 $S_1=\{1,2\}$, $S_2=\{2,4\}$，则 $S=\{2,4\}$，所以 $\tau_{res}^{size}=\tau_{e_1}^{size}\sqcap\tau_{e_2}^{size}$。同样 τ^{sign} 的推导过程如下：

$$\frac{\text{unsigned}\in\tau_{e_1}^{sign},\text{unsigned}\in\tau_{e_2}^{sign}}{\text{unsigned}\in\tau_{res}^{sign}},$$

$$\frac{\text{unsigned}\in\tau_{e_1}^{sign},\text{signed}\in\tau_{e_2}^{sign}}{\text{unsigned}\in\tau_{res}^{sign}},$$

$$\frac{\text{signed}\in\tau_{e_1}^{sign},\text{unsigned}\in\tau_{e_2}^{sign}}{\text{unsigned}\in\tau_{res}^{sign}},$$

$$\frac{\text{signed}\in\tau_{e_1}^{sign},\text{signed}\in\tau_{e_2}^{sign}}{\text{signed}\in\tau_{res}^{sign}}$$

对于拷贝操作，其类型依赖方程为 $e_1:t_{e_1}\rightarrow e_2:t_{e_2}\Rightarrow e_{res}:t_{res}$，则 $t_{res}=t_{e_1}\sqcup t_{e_2}$。

对于间接引用操作，其类型推导规则为 $\tau_{e_1}^{core}=\tau_{e_1}^{core}\cup\{$pointer$\}$，并且 τ_{res}^{size} 和 τ_{res}^{sign} 保持不变。

2.3 类型重构算法

算法以 OBJ $=\{obj_1,\cdots,obj_n\}$ 为输入，通过 make_equationtree, OBJ)建立汇编指令的类型依赖方程；用 init(equation)将系统中所有的对象类型初始化为 void；spread_info_lr(equation)和 spread_info_rl(equation)实现类型信息从左到右(操作数到操作结果)和从右到左(操作结果到操作数)的传播；Type(*obj*)返回系统方程中对象 *obj* 的类型；*_join(Type(*obj*))执行 join 操作，当重构类型发生变化时返回 true，否则返回 false；match_type(Type(*obj*))则在类型推导完毕后，从重构的类型中按照类型属性的关系匹配对象 *obj* 的最佳类型。

由于该算法是基于有限对象的推导过程且每一个对象的属性都是有穷格的单调描述，因此整体的算法过程如图 2 所示。

```
Tree=make_tree(program);
Equation=make_equation(Tree,OBJ)
init(Equation);
set_constraint(Equation);
flag=true;
while(flag)
    flag=false;
    for all equation Equation
        spread_info_lr(equation);
    for all objOBJ:
        flag|=left_join(Type(obj));
    for all equation Equation
        spread_info_rl(equation);
    for all objOBJ:
        flag|=right_join(Type(obj));
    for all objOBJ:
        flag|=full_join(Type(obj));
for all objOBJ:
    Tobj=match_type(Type(obj));
```

图 2 基本数据类型重构算法

3 复杂数据类型重构

假设所有内存访问均可以用 $base+offset+\sum_{j=0}^{n}C_jx_j$ 的形式表示[6]，其中 *base* 表示基地址，*offset* 表示相对于基址的偏移量，且与 C_j 均为常量。不失一般性，假设 $0<C_j<C_{j+1}$，且 $m=\min_{j=1}^{n}C_j=C_1$, $M=\max_{j=1}^{n}C_j=C_n$。则后续主要是对内存访问表达式进行化简，转换成对象表达式(*obj*+*offset*)。

3.1 基本思想

复杂类型重构的目标是最大限地从汇编指令列表当中提取可能的类型信息。考虑如下的变量类型：

结构体　结构体是内存中不同类型变量的集合，其大小由其内部变量决定。编译器出于体系结构的要求可能会在结构体中间或结尾处加入额外的字节来对齐内存空间，同时由于变量符号名在编译过程中被删除了而以数值偏移量代之，因此对于不同的结构体，其成员变量可能会产生相同的数值偏移量。为了得到完整的结构体类型，需要用某些全局信息以变量大小作为特征，将数值偏移量映射成具有准确含义的符号名。

根据结构体内存布局特点，通过对具有相同基地址的内存偏移量进行收集，可以获得整个结构体的框架结构。而后利用简单类型的重构思想对框架内不同的偏移量进行迭代求解来获取偏移对象的类型，若偏移结构仍然是一个复合类型，则可再次利用复杂类型重构的思想对该偏移结构进行递归计算，直至求得结构中所有变量的类型。

数组　数组是内存中相同类型元素的集合。与上述相比，结构体由基址通过常偏移量进行访问，而数组则通过索引表达式进行访问。比如，对于数组 int $m^{[32]}$ 和表达式 $m[j]$，其对应的索引表达式可能是 $m+j*4$。由于数组中各元素类型相同，且索引项不提供附加的类型重构信息，因此可以将其去掉。当去掉地址表达式中的索引项之后，仅剩下基址和常偏移量，这与结构体地址表达式相似，组合 $base+offset$ 将作为进一步类型分析的依据。

地址表达式的乘法部分提供了数组元素大小的信息，通过上面的例子，可以推断'4'就是数组元素的大小。如果是多维数组，地址表达式可能会更加复杂。但根据实验结果，依旧可以推断地址表达式中的最小常乘数就是数组元素的大小。

如果高级语言当中使用带有乘法的索引表达式，元素大小的计算会变得复杂。比如 $m[2*j]$，其对应的地址表达式是 $m+j*8$，若单用一个地址表达式则很难正确地判断该数组元素的大小，很可能将其错估为 8；若通过对指向同一数组的多个地址表达式进行综合判断，将表达式中多个乘数的最大公约数 $m'=GCD(m_1,\cdots,m_n)$ 作为数组元素的大小，该歧义就可消除。

3.2 复杂类型重构过程

根据以上信息得出复杂类型的重构步骤如下：

赋值标记(Assign_labels)　标记为类型重构提供原始数据，被标记的对象表示其为指针类型(复杂类型)的可能性，因此需要为程序中的对象指定标记作为分析的对象。令 l 表示一个标记，则需要赋予标记的元素包括：

1)内存访问：$l_i(b_i,o_i,m_i,\cdots,M_i)$

2)子程序的返回值

构建标记集合(Build_obj_label_sets)　为了判断不同的标记是否指向同一个指针类型，需要建立标记等价类，因此首先要建立推导链，分析标记的可达性。令 $LS_{in}(reg,n)$ 为汇编指令第 n 行之前的可达标记的集合，$LS_{out}(reg,n)$ 为第 n 行之后的可达标记的集合。则该函数主要是对上述的赋值标记进行分析，确定 LS 集合。

构建标记等价集合(Build_label_equiv_sets)　基于已建立的标记集合，定义如下的等价关系：

$$\frac{\exists l_1,l_2,reg,n:\{l_1,l_2\}\subseteq LS(res,n)}{l_1\equiv l_2}$$

$$\frac{l_1:(b_1,o_1,m_1,\cdots,M_1),l_2:(b_2,o_2,m_2,\cdots,M_2),b_1\equiv b_2}{l_1\equiv l_2}$$

按照该规则，所有的标记都将会被划分成等价类，且每个等价类对应一个确定的复杂数据类型。

构建聚集集合(Build_aggreg_sets)　对于等价集合中的任意标记，根据如下规则定义聚集关系，记为 $AA(l_1,l_2)$：

$$\frac{l_1:(b_1,o_1,m_1,\cdots,M_1),l_2:(b_2,o_2,m_2,\cdots,M_2),|o_1-o_2|<m_1}{AA(l_1,l_2)}$$

则所有等价标记依据偏移量之间差值的不同，可筛选出嵌套的数据结构的标记集合，也即所有具有该关系的等价标记将会被进一步划分成新的等价类。由于聚合操作没有改变标记的基地址，因此如果 l_1 和 l_2 在操作之前含有相同的基址，则聚合后基址依旧相同。

结构化类型的重构(Reconstruct_structs)　通过以上收集的信息可以对结构化的类型进行重构。设 S 是所有标记的集合，其划分的等价类为 $S_1,S_2,\cdots,S_m$，则每个等价类中所含的标记都有相同的基址。再设 Q_i 为等价类 S_i 中所有元素相对于该等价类基址的偏移集合，则该集合可看成是一个新的结构化类型的变量的偏移集合。

更新对象集合(Update_object_sets)　对于每个新的变量类型都会有新的对象产生，且该对象会被加入到图 1 所示的新对象集进入下一轮基本类型重构。至此，复杂类型的类型框架恢复过程结束，后续将对框架内的变量进行基本类型的重构，直至对象集合为空。复杂类型重构的大体算法过程如图 3 所示。

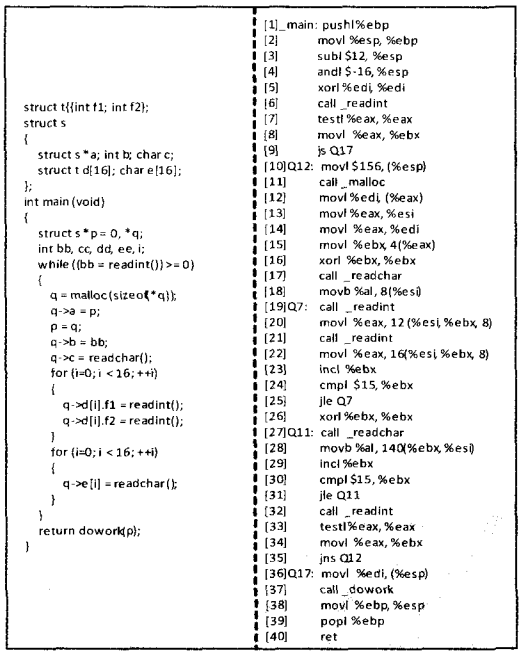
```
def composite_reconstruction()
    assign_labels()
    build_reg_label_sets()
    build_label_equiv_sets()
    build_aggreg_sets()
    reconstruct_structs()
    update_object_sets()
```

图 3　复杂数据类型重构算法

3.3 实例分析

为了说明复杂类型的重构过程，考虑图 4(a)给出的程序。其中 readint，readchar 及 dowork 是为阻止编译器进行代码优化而附加的函数。图 4(b)给出了图 4(a)中程序经编译器 MinGW GCC 3.4.5 编译后所对应的汇编代码。

由图 4 可见，在聚集步骤中检测到了一个等价类 $\{L_{20}:(L_{11},12,8),L_{22}:(L_{11},16,8)\}$，其中 L_{11} 是第 11 行指令 malloc 函数返回值的标记。经过聚集步骤后产生了一个新的等价类 $S_1:\langle L_{11},12\rangle$，则前面的标记 L_{20} 和 L_{22} 可以重新表示为 $L_{20}:\langle S_1[],0\rangle$ 和 $L_{22}:\langle S_1[],4\rangle$，'[]'代表嵌套标识符。在结构体重构步骤中，标记 S_1，L_{12}，L_{15}，L_{18} 和 L_{28} 由于含有相同的基地址 L_{11}，因此其成员变量的偏移集合是 $\{0,4,8,12,140\}$，按照前述聚集关系经过迭代操作后可得偏移为 12 和 140 处的标记结构是数组类型。则按照以上信息复杂类型的重构结果如图 5 所示，其中 t_1,t_2,t_3,t_4,t_5 及 t_6 表示待识别的基本类型，$f_1,f_2,f_3,f_4.f_1,f_4.f_2$ 和 f_5 表示新产生的对象，它们将送入简单类型重构模块进行类型恢复。



```
struct t{int f1; int f2};
struct s
{
   struct s *a; int b; char c;
   struct t d[16]; char e[16];
};
int main(void)
{
   struct s *p = 0, *q;
   int bb, cc, dd, ee, i;
   while((bb = readint()) >= 0)
   {
      q = malloc(sizeof(*q));
      q->a = p;
      p = q;
      q->b = bb;
      q->c = readchar();
      for(i=0; i < 16; ++i)
      {
         q->d[i].f1 = readint();
         q->d[i].f2 = readint();
      }
      for(i=0; i < 16; ++i)
      {
         q->e[i] = readchar();
      }
   }
   return dowork(p);
}
```

```
[1]_main: pushl %ebp
[2]       movl %esp, %ebp
[3]       subl $12, %esp
[4]       andl $-16, %esp
[5]       xorl %edi, %edi
[6]       call _readint
[7]       testl %eax, %eax
[8]       movl %eax, %ebx
[9]       js Q17
[10]Q12:  movl $156, (%esp)
[11]      call _malloc
[12]      movl %edi, (%eax)
[13]      movl %eax, %esi
[14]      movl %eax, %edi
[15]      movl %ebx, 4(%eax)
[16]      xorl %ebx, %ebx
[17]      call _readchar
[18]      movb %al, 8(%esi)
[19]Q7:   call _readint
[20]      movl %eax, 12(%esi,%ebx, 8)
[21]      call _readint
[22]      movl %eax, 16(%esi,%ebx, 8)
[23]      incl %ebx
[24]      cmpl $15, %ebx
[25]      jle Q7
[26]      xorl %ebx, %ebx
[27]Q11:  call _readchar
[28]      movb %al, 140(%ebx,%esi)
[29]      incl %ebx
[30]      cmpl $15, %ebx
[31]      jle Q11
[32]      call _readint
[33]      testl %eax, %eax
[34]      movl %eax, %ebx
[35]      jns Q12
[36]Q17:  movl %edi, (%esp)
[37]      call _dowork
[38]      movl %ebp, %esp
[39]      popl %ebp
[40]      ret
```

(a) 源程序　　(b) 汇编程序

图 4

```
struct s2
{
   t1 f1;  /at offset 0 */
   t2 f2;  /at offset 4 */
   t3 f3;  /at offset 8 */
   struct s1  /* sizeof(struct s1) == 8 */
   {
      t4 f1;  /at offset 0 */
      t5 f2;  /at offset 4 */
   } f4[];  /at offset 12 */
   t6 f5[];  /at offset 140 */
};
```

图 5　复杂类型重构的中间结果

结束语　本文给出了汇编算法基础上数据类型自动重构的算法，该算法能够对简单及复杂数据类型进行准确的恢复。由于算法是基于简单类型的有穷格及有限内存访问标记集合上的操作，因此其能够在有限循环内结束。该算法是目前正在开发的类型重构工具的关键技术，下一步将尝试采用动态的分析方法对二进制代码进行分析，形成动、静结合的分析方法，争取在复杂数据类型重构的准确度上有进一步的提高。

参 考 文 献

[1] 陈凯明，刘宗田. 反编译研究现状及进展 [J]. 计算机科学，2001，28(5)：113-115

[2] 肖海，陈平. 基于运行时类型分析的整形漏洞二进制检测和定位系统 [J]. 计算机科学，2011，38(1)：140-143

[3] Lin Z, Zhang X, Xu D. Automatic reverse engineering of data structures from binary execution[C]// Proceedings of the Network and Distributed System Security Symposium. 2010

[4] Cifuentes C. Reverse Compilation Techniques, Queensland University of Technology[D]. Department of Computer Science, July 1994

[5] Mycroft A. Type-Based Decompilation[C]// Proceedings of the 8th European Symposium on Programming Languages and Systems. March 1999：208-223

[6] Emmerik M V. Static Single Assignment for Decompilation[D]. Queensland：The University of Queensland，2007

(上接第 119 页)

化测试策略产生的总折扣和检测到的缺陷数均明显多于随机测试策略，因此本文的优化测试策略优于随机测试策略。拟在以后的研究中进一步扩展制约条件，以提高其效率和适用性。

参 考 文 献

[1] 陈明. 软件测试技术[M]. 北京：清华大学出版社，2011：11-61

[2] Dalley J L. The art of software testing[C]// Dayton. Proceedings of the IEEE 1991 National Aerospace and Electronics Conference. Dayton，1991：757-760

[3] Cai K Y, Cangussu J W, DeCarlo R A, et al. An overview of software cybernetics[C]// Proc. the 11th International Workshop on Software Technology and Engineering Practice. Chicago：IEEE Computer Society Press，2004：77-86

[4] Cai K Y. Optimal software testing and adaptive software testing in the context of software cybernetics[J]. Information and Software Technology，2002，44(02)：841-855

[5] Chen T Y, Li Y C, Ning W Y, et al. Adaptive testing of software components[J]. ACM Symposium on Applied Computing，2005，17(5)：1463-1469

[6] Wong W E, Hu Hai. Optimal and adaptive testing with cost constraints[C]// Proceedings of the 2006 International Workshop on Automation of Software Test. Shanghai：ACM，2006：71-77

[7] Okamura H, Furumura H, Dohi T. On the effect of fault removal in software testing Bayesian reliability estimation approach[C]// Washington. Proceedings of the 17th International Symposium on Software Reliability Engineering. Washington：IEEE Computer Society，2006：247-255

[8] Derman C. Finite State Markovian Decision Processes[M]. New York：Academic Press，1970：234-305

[9] Loo P S, Tsai W K. Random testing revisited [J]. Information and Software Technology，1988，30(7)：402-417

[10] Chen T Y, Kuo F C, Liu Huai, et al. Does Adaptive Random Testing Deliver a Higher Confidence than Random Testing [C]// IEEE the 18th International Conference on Quality Software. Chicago：IEEE Computer Society，2008：145-154

[11] Cai K Y. A controlled Markov chains approach to software testing [J]. IEEE Transactions on Software Engineering，2002，21(6)：312-320

[12] Hu Hai, Jiang Chang-hai. Adaptive Software Testing in the Context of an Improved Controlled Markov[C]// Annual IEEE International Computer Software and Applications Conference. Chicago：IEEE Computer Society，2008：853-858

[13] 殷脂，曹渠江. 构件软件的自适应测试[J]. 计算机应用，2005，25：417-420

[14] 江韩斌，崔小乐，周豪. 自适应软件测试过程的稳定性模型[J]. 微电子学与计算机，2008，25，(8)：98-102