# 一种基于CIL静态分析的C#程序缺陷检测方法

边 攀 梁 彬 石文昌
（中国人民大学信息学院 北京 100872）

**摘 要** 通过静态检测发现源程序中的潜在缺陷，可以帮助程序员在软件发布之前发现并修补程序缺陷，提高软件的安全性。提出一种通过静态分析CIL代码来检测C#程序代码缺陷的方法。采用改进的深度优先搜索算法遍历目标程序的控制流程图，结合历史状态缓存机制，能够大幅度提高检测效率；另外，为便于实施别名分析，还提出一种基于内存区域的变量表示方法。基于所述分析方法，开发了一个C#源代码缺陷静态检测系统，并对实际开源项目进行了检测。实验结果表明，本系统能够高效、准确地检测C#程序中常见类型的缺陷。

**关键词** 静态分析，缺陷检测，别名分析，CIL，C#

**中图法分类号** TP311 **文献标识码** A

## CIL Static Analysis Method for C# Program Defect Detection

BIAN Pan LIANG Bin SHI Wen-chang
(School of Information, Renmin University of China, Beijing 100872, China)

**Abstract** Finding potential defects by statically detecting source code can help programmers find and fix the defects before the software is released, and thus can improve the security of the software. This paper provided a CIL static analysis method to detect defects in C# programs. We adopted an improved depth-first search algorithm to traverse the control flow graph of the target program, and combining with the strategy of caching history states, the performance of the detection can be greatly improved. In addition, to be convenient for alias analysis, we proposed a method based on Memory Region to represent variables. Based on the analysis method described in this paper, we developed a system for detecting defects in C# programs. We applied the system on real C# projects, and the detecting result shows that it can detect common kinds of defects in C# programs efficiently and accurately.

**Keywords** Static analysis, Defect detection, Alias analysis, CIL, C#

## 1 概述

C#在编程语言排行中名列前茅，在各个方面，尤其是Web编程方面具有广泛的应用。而Web应用又常常是攻击者的攻击目标，因此迫切需要开发一种能够有效检测C#软件中代码缺陷的方法，以提高软件的安全性。一般地，缺陷检测分为静态检测和动态检测，相比而言，静态检测[3,5-7,10]不需要执行源程序，能够帮助程序员在软件发布之前发现程序缺陷，并及时修补，从而提高软件的安全性。

静态检测的一般做法是遍历目标程序可执行路径，与检测规则相匹配，并将违反规则的代码作为潜在缺陷报告给用户[5,7]。静态检测的首要任务是获取目标程序并进行预处理。在实际应用中，某些C#软件可能难以获得源代码，因此直接在C#源代码层次进行缺陷检测并不是最佳选择。调研发现，C#语言是基于.NET框架的，源程序在编译后会生成由CIL(Common Intermediate Language，通用中间语言)指令组成的托管程序集(Managed Assembly)。因此，可以通过分析CIL程序集，实现对C#代码缺陷的静态检测。

选择在CIL层次进行缺陷检测，除了考虑在缺少源码时也可以实施检测外，还基于如下考虑：(1)CIL指令相比于C#源程序在词法、语法上相对简单，更容易实施静态分析；(2)在C#语言版本的多次更新中，CIL指令相对稳定，因此对新版本的支持无需过多修改检测引擎；(3)可以经过简单的修改，使检测引擎适用于其他以CIL为中间语言的编程语言(如VB.NET)，实现对多语言的支持。

基于上述因素，本文提出一种在CIL层次上静态检测C#代码缺陷的方法，用以支持路径敏感、上下文敏感的过程间分析。我们采用启发式的深度优先方法遍历程序路径，与历史状态缓存机制相结合，可以提高静态检测的效率；同时为了便于实施别名分析，我们还提出一种基于内存区域的变量表示方法。为验证本文所述方法的有效性，我们实现了一个静态检测系统，并选择两个实际的C#开源项目作为测试用例；实验结果表明检测引擎能够快速、有效地检测C#程序中常见类型的代码缺陷。

---

到稿日期：2013-03-20 返修日期：2013-06-17 本文受国家自然科学基金项目(61170240，61070192)，核高基重大专项(2012ZX01039-004)资助。

**边 攀**(1987－)，男，硕士生，主要研究方向为信息安全；**梁 彬**(1973－)，男，博士，副教授，硕士生导师，主要研究方向为信息安全与系统软件，E-mail：liangb@ruc.edu.cn；**石文昌**(1964－)，男，博士，教授，博士生导师，CCF高级会员，主要研究方向为信息安全与软件系统。

## 2 问题分析

### 2.1 CIL 程序集的解析

C#源程序编译生成的托管程序集,除包含 CIL 指令流外,还包括其类的继承、接口的实现、方法摘要等信息,甚至还包括一些调试信息。缺陷检测引擎在加载程序集后需要识别这些信息,并分析 CIL 指令流,构建控制流程图(CFG,Control Flow Graph)[1],为后续的缺陷检测过程提供分析平台。Mono[8]是一个由 Novell 公司主持的开源项目,能够跨平台运行。Mono 可以解析 CIL 程序集,从中提取相关信息及获取 CIL 指令流。然而,Mono 提供的控制流程图结构并不适合直接用于静态分析。因此还需要在 Mono 的基础上分析 CIL 指令流,并为每个函数定义建立相应的控制流程图。幸运的是,由于 CIL 指令相对源码简单很多,因此容易实现控制流程图的构建。

### 2.2 路径爆炸

静态检测是一个遍历程序可执行路径、匹配检测规则的过程。如果目标程序中包含太多可执行路径,超出了计算机的分析能力,将会造成路径爆炸。而当程序中存在循环时,该问题会更加严重。此时,完全分析所有路径所需的时间开销将不可接受。路径爆炸问题严重影响着静态分析工具的检测效率和实用性。然而目前还没有能够有效解决路径爆炸问题的方法。在静态分析领域,普遍通过增加一些限制条件来加以缓解,以使分析工具能够正常工作。常见方法是设置路径上限,即当被分析的路径数达到一定数目时,将停止对相应程序的分析。显然,这种限制将导致某些路径未能被分析,从而造成漏报。为降低此类漏报,需要引入一些措施,优先分析那些最可能包含缺陷的代码。

程序中的某些路径可能包含一些相同的代码,如果每条路径都单独分析,有些代码可能会被重复分析,造成计算资源的浪费。为此,Hallem[5]等人引入状态缓存机制,为每个代码块(控制流程图上的基本块,Basic Block)设计一个历史状态集,用于记录进入该代码块时各变量的状态。在分析某个代码块之前,首先判断变量状态是否命中历史状态,如果所有变量都命中了历史状态,则无需分析后续代码。类似地,在过程间分析中,为每个函数设置函数摘要。不同的是,函数摘要除了记录进入时的状态,还需要记录对应的结果状态,以便于获得函数调用后的各变量的状态。

在实践中,我们发现采用传统的深度优先方法遍历程序路径时间开销较大。根本原因是深度优先遍历未能充分发挥状态缓存机制的作用。为此,本文提出一种启发式的遍历方法。本方法基于如下现象:(1)越靠近函数入口的代码对后续分析的影响越大;(2)相比而言,更容易从那些未被分析过或被分析次数较少的代码中检测出新的缺陷。基于上述观察,我们的启发式遍历方法优先分析那些靠近函数入口和被分析次数少的代码。这样做可以优先检测那些最有可能包含缺陷的代码。更重要的是,将本方法与状态缓存机制相结合,能够提高历史状态的命中率,降低时间开销。

### 2.3 别名问题

另一个亟需解决的是别名问题。在编译原理中,如果两个变量指向同一块内存空间,则称它们具有别名关系[4]。在静态分析领域,非指针变量之间也可能具有别名关系。例如,整形变量 $a$ 被赋值给 $b$,如果 $b$ 经过了上限检查,则变量 $a$ 也可以被认为经过了上限检查。因此,别名概念扩展如下:如果变量 $v_1$ 的状态发生了改变,变量 $v_2$ 的状态也将发生同样的改变,则变量 $v_1$ 和 $v_2$ 具有别名关系。如果静态检测工具不能识别变量间的别名关联,或将不具有别名关系的变量误认为别名变量,可能会使检测结果不准确。

在前人的工作中,通常将别名关系表示为变量对[4];一个变量对是一个二元组,记录着两个具有别名关系的变量。在实际应用中,这种方法在处理类或结构成员时,实施起来比较复杂。设想如下情形:当修改 $a$、$b$ 的状态时,此时不但要查找与 $a.b$ 互为别名的变量,还需要查找与 $a$ 互为别名的变量。为克服这种方法的不足,本文引入一种模拟内存区域(MR,Memory Region)的变量表示方法。这一方法借鉴了内存的分配策略,即互为别名的指针将指向同一块内存。在我们的方法中,变量的状态及其他信息被封装在一个内存区域中,而变量指向该区域。如果变量互为别名,则它们指向相同的内存区域。此时,当改变某个变量的状态时,不需要额外的操作,也会同时更改与其互为别名的变量的状态。可见,采用本方法能使别名分析更加便捷。

## 3 C#代码缺陷静态检测方法

在 CIL 层次对 C#程序实施静态检测,其流程如图 1 所示。首先解析 CIL 程序集,生成控制流程图;然后遍历控制流程图,并根据检测规则对潜在执行路径上的指令流进行分析,如果程序代码违反检测规则,则可能存在安全隐患,将被作为潜在缺陷报告给用户,供用户分析是否为真实缺陷。从是否涉及函数间的交互来看,静态分析分为过程内分析和过程间分析。

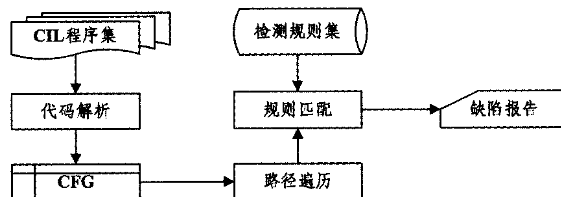


图 1 C#程序静态分析流程图

### 3.1 过程内分析

#### 3.1.1 代码解析

我们借助 Mono 实现对程序集的解析,获取每个函数定义的指令流,并进而生成控制流程图。控制流程图的节点是一些 BB 块(Basic Block,基本块),BB 块中的指令是顺序执行的,不存在执行流的跳转。关于控制流程图的构建,请参考编译原理的相关章节[1],本文对此不再赘述。

#### 3.1.2 检测规则的匹配

检测规则是静态分析的核心,通常用一个有限状态机来描述。静态分析引擎对程序的控制流程图进行遍历,并对各路径上的指令进行分析,识别各指令的语义。如果指令对应缺陷状态机中的操作,则根据变量的当前状态按照状态机的转换规则改变变量的状态。当变量状态属于异常状况时,说明发现了一个潜在缺陷。

如图 2 所示为资源未释放缺陷检测规则对应的缺陷状态机。它表示执行分配资源操作时,变量状态由“初始状态”转换为“资源未释放”;当执行释放资源操作时,变量状态将转换

为“资源已释放”；而执行其他操作时则保持原状态不变。如果在路径结束时，变量状态为“资源未释放”，则说明该变量所持有的资源未被释放，程序可能存在安全隐患。例如，经过分配资源操作 $v$=new *FileStream*()，变量 $v$ 的状态为“资源未释放”；而当变量 $v$ 的状态为“资源未释放”时，调用操作 $v$.*Close*()，则变量 $v$ 的状态转换为“资源已释放”。如果在某路径上，执行 $v$ =new *FileStream*()操作后未执行 $v$.*Close*()操作，则在该路径上打开的文件流未被关闭，存在安全隐患。

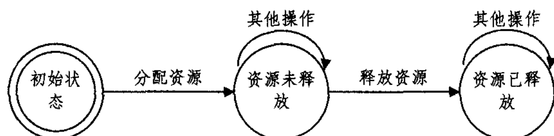


图 2　资源未释放检测规则对应的状态机

### 3.1.3　基于模拟内存区域的变量表示法

在缺陷分析的过程中，我们需要记录每个变量类型、变量状态等信息。为了方便进行别名分析，本文采用基于模拟内存区域的变量表示方法。在编译原理中，别名一般存在于指针之间。在程序执行时，一个指针保存着一个被分配的内存空间的地址；如果两个指针互为别名，那么它们指向的内存地址相同。基于上述现象，将变量的基本信息存储在一个内存区域中，而互为别名的变量指向同一个内存区域。

如图 3 所示为基于内存区域的变量表示方法示意图。一个变量包括一个唯一标识符，并指向一个内存区域。内存区域主要包含 4 个域，分别用于记录变量的状态、数据类型、结构成员及变量列表。(1)状态域记录着变量当前所处的状态，如“资源未释放”、“资源已释放”等。(2)数据类型域记录着变量的实际类型，例如在操作 $B$ $a$=new $A$()(类 $A$ 继承类 $B$)中，虽然变量 $a$ 的类型为 $B$，但实际分配的类型是 $A$，此时将 $A$ 记录在数据类型域中。(3)结构成员域是一个列表，记录着变量的所有成员的信息，而每个成员的表示方法与变量相同，同样包含标识符及相应的模拟内存区域。(4)变量列表域记录着哪些变量指向该模拟内存区域，在列表中的变量互相之间具有别名关系；如图 3 所示，变量 $a$ 和 $b$ 指向同一个 MR，那么该 MR 的变量列表包含 $a$ 和 $b$。

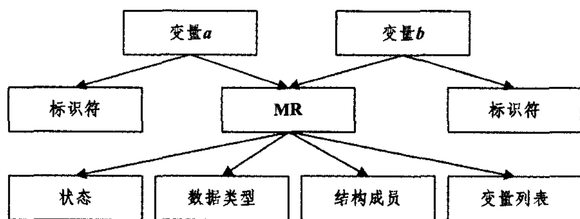


图 3　模拟内存区域的变量表示方法

这种表示方法便于实施别名分析。在匹配规则的过程中，当转换某变量的状态时(在静态检测中经常发生)，只需要改变相应 MR 的状态域，而不需要额外转换与该变量具有别名关系的变量状态。

一般而言，在静态分析中，当变量(不限于指针)之间存在赋值操作时，它们将具有别名关系。例如将变量 $a$ 赋值给变量 $b$，则 $b$ 将指向变量 $a$ 的 MR，并将 $b$ 添加到 MR 的变量列表中。解除别名关系通常是在对变量重新赋值时，例如赋予变量 $b$ 一个新的值，则 $b$ 将指向另外一个模拟内存区域 $MR_1$，同时将 $b$ 从 MR 的变量列表中删除。

### 3.1.4　启发式遍历方法

本文提出一种启发式遍历方法，其对深度优先方法进行了改进，能够使状态缓存机制充分发挥作用。遍历算法如图 4 所示。算法的输入包括待分析函数的控制流程图和初始变量表。变量表中记录着分析过程中所涉及的所有变量的信息。

```
输入: CFG, var_table
输出: var_table_list
声明: CFG表示控制流程图
        入口BB块entry_bb, 出口BB块exit_bb
        bb->ip表示bb的入口指令, bb->history表示着历史状态
        bb->times是bb的技术器, 记录着bb被遍历的次数
    var_table记录着分析过程中所涉及的所有变量的信息
    var_table_list由多个var_table组成
    work_list记录着即将被分析的BB块
算法:
1  s=<entry_bb, 0, var_table>
2  work_list := {s}
3  WHILE work_list != φ
4   从work_list头部取出<bb, ip, var_table>
5   IF bb == exit_bb
6    将var_table添加到var_table_list中
7    CONTINUE
8   ELSE IF bb->ip != ip
9    从指令ip开始分析bb块中的指令
10  ELSE IF var_table  bb->history
11   CONTINUE
12  ELSE
13   bb->history = bb->history ∪ var_table
14   bb->times ++
15   从指令ip开始分析bb块中的指令
16   FOREACH next_bb in bb->succs
17    var_table1 = COPY var_table
18    s = <next_bb, bb->ip, var_table1>
19    IF next_bb->times在bb的所有bb的可行后继中最小
20     将s插入到work_list头部
21    ELSE
22     将s插入到work_list尾部
23  其他操作
24  RETURN var_table_list
```

图 4　启发式遍历方法

遍历算法使用 *work_list* 记录待分析的 BB 块，BB 块(Basic Block，基本块)是组成控制流程图的节点，BB 块内的指令流是顺序执行的。在 *work_list* 中的每个元素是一个三元组〈*bb*，*ip*，*var_table*〉，其中 *bb* 表示将被分析的 BB 块，*ip* 表示待分析指令流的起始位置(在过程内分析中，*ip* 与 *bb* 的入口指令相同；而在过程间分析中，*ip* 与 *bb* 的入口指令可能并不相同)，而 *var_table* 记录着各变量的状态及其他信息。算法开始时将 *work_list* 初始化为控制流程图的入口 BB 块(1—2 行)。

算法从 *work_list* 中获取即将被分析的 BB 块及相关信息，直到 *work_list* 为空(3—4 行)。在循环体内部，根据 *bb* 的不同情况进行不同的处理：(1)如果 *bb* 是出口 BB 块，则将变量表添加到返回列表中，并继续取下一个 BB 块进行分析(5—7 行)；(2)如果 *bb* 的入口指令与 *ip* 不相同，则直接从 *ip* 开始，分析 BB 块中的指令流(8—9 行)，这一条件主要用于过程间分析；(3)如果变量表是 *bb* 历史状态的子集(即命中历史状态)，则说明已经以相同状态对后续代码进行了检测，不需要再重复检测(10—11 行)；(4)将未命中历史状态的变量添加到 *bb* 的历史状态中，并将 *bb* 的计数器加 1，然后分析 *bb* 中的指令流，与检测规则相匹配(13—15 行)。完成 *bb* 中指令流的分析之后，将 *bb* 的后继插入到 *work_list* 中。将计数器值最小的后继 BB 块添加到 *work_list* 头部，而其他 *bb* 块则添加到 *work_list* 尾部(19—22 行)。这样做可以保证那些尚未被分析的程序代码尽早得到检测。

整个控制流程图遍历结束之后，将执行一些后续操作(23行)，如清理数据结构、报告发现的缺陷等。在过程间分析中，最后还需要返回分析结果(24行)。

### 3.2 过程间分析

过程内分析的是局部代码，容易产生误报和漏报，因此需要引入过程间的分析。过程间分析当遇到函数调用时需要进入被调用函数进行分析。我们采用了一种与文献[5]相似的过程间方法。在开始分析新的函数之前，保存当前状态，并为新的遍历过程构建初始变量表(Refine 过程)；而被调用函数分析完成后，则恢复保存的变量表，并根据返回的结果(*var_table_list*)调整变量表的内容(Restore 过程)。Refine 和 Restore 操作如下：

• Refine。将实参及全局变量对应的变量信息复制到新的变量表中。

• Restore。Restore 过程相对复杂。由于控制流程图往往有多条路径，返回的 *var_table_list* 中可能包含多个变量表，对于其中的每个变量表 $var_table_1$，将当前变量表复制一份 $var_table_0$，并根据 $var_table_1$ 及参数的类型，对 $var_table_0$ 中变量的内容进行调整。假设 $var_1$ 为 $var_table_1$ 中的变量，$var_0$ 为其在 $var_table_0$ 对应的变量。(1)如果 $var_1$ 为全局变量或引用参数(在 C# 语言中由关键词 ref 或 out 修饰的参数)，则按照赋值操作处理，即将形参赋值给实参变量；(2) $var_1$ 为值类型参数(如整形、字符型等)，此时不对 $var_0$ 做任何操作；(3)而对于其他类型的参数在运行时实参本身不会改变，但内容可能会被修改，因此用 $var_1$ 指向的模拟内存区域的状态域、数据成员域覆盖 $var_0$ 的相关域。所有的 $var_table_0$ 组成链表 $var_table_list_0$，从该链表中选取一个作为当前状态表继续进行后续指令的分析，而其他变量则构建三元组 $\langle bb, ip', var_table_0 \rangle$，并插入到 *work_list* 的头部(其中 *bb* 为函数调用语句所在的 *bb* 块，$ip'$ 为函数调用指令的下一条指令，$var_table_0$ 为链表 $var_table_list_0$ 中的成员)。

## 4 实验

基于上述方法，我们实现了一个 C# 程序缺陷静态检测系统，其能够检测 C# 程序中存在的不适当的类型转换、资源未释放、资源释放后被使用、空对象解引用、函数返回值在使用前未被检测、拒绝服务等 6 种类型的缺陷。另外，我们还为检测系统手工配置大约 16690 个检测规则，涉及 793 个库函数。

为评估检测系统的功能和性能，我们选取了两个 C# 开源项目，作为静态检测系统的测试用例。这两个开源项目的基本情况如表 1 所列。其中 BlogEngine[2] 是一个开源的博客系统，而 SourceGrid[9] 是一款表格控制管理工具。

表 1 测试用例基本情况

| 项目名称 | 项目大小(MB) | 文件数(个) | 代码量(LOC) |
|---|---|---|---|
| BlogEngine. NET 1.6 | 3.08 | 88 | 26000 |
| SourceGrid 4.22 | 20.8 | 356 | 60000 |

检测系统运行在一台双核机器上，频率为 2.40GHz，内存为 2G，操作系统为 Red Hat Enterprise 6。对上述两个开源项目进行缺陷检测，结果如表 2 所列。从检测时间上来看，系统能够在较短的时间内完成检测任务。对检测结果进行人工审计，发现静态检测系统共报告 29 个缺陷，其中 16 个为真实缺陷，检测准确率达到 55.2%。

表 2 检测结果

| 项目名称 | 检测时间(秒) | 缺陷个数 | 误报数 | 正确率(%) |
|---|---|---|---|---|
| BlogEngine. NET 1.6 | 5.4 | 18 | 11 | 38.9 |
| SourceGrid 4.22 | 10.8 | 11 | 2 | 81.8 |
| 总计 | 16.2 | 29 | 16 | 55.2 |

图 5 为静态检测系统从 BlogEngine 中发现的一个空对象解引用的缺陷实例。文件 MetaWeblogHandler. cs 中的函数 *GetPage* 调用了文件 Page. cs 中的 *GetPage* 函数(607 行)，而当不存在以参数 *id* 为索引的页面时，后者将返回一个空对象。因此函数调用后 *page* 的值可能为 null。然而 *page* 在未被检查是否为空对象的情况下被解引用(609 行)，这将可能导致运行时错误。由于该缺陷的触发路径涉及多个函数，需要进行过程间分析才能被发现。

```
BlogEngine.Core/API/MetaWeblog/MetaWeblogHandler.cs:
602 internal MWAPage GetPage(string blogID, string pageID, ...)
603 {
    ......
607   Page page = Page.GetPage(new Guid(pageID));
608
609   sendPage.pageID = page.Id.ToString();
    ......
620 }
```

```
BlogEngine.Core/Page.cs:
266 public static Page GetPage(Guid id)
267 {
268   foreach (Page page in Pages)
269   {
270     if (page.Id == id)
271        return page;
272   }
273
274   return null;
275 }
```

图 5 从 BlogEngine 中检测到的空对象解引用的缺陷

实验结果表明 C# 源代码缺陷静态检测系统能够发现实际 C# 项目中的程序缺陷，具有较高的检测效率和准确率。

**结束语** 本文阐述了一种在 CIL 层次对 C# 代码缺陷进行静态检测的方法。在 CIL 层次进行静态分析，可以避免对源代码的依赖，且更容易实施缺陷检测。根据实际需求，我们采用启发式方法遍历目标程序的控制流程图，优先分析靠近函数入口的代码和最有可能检测出缺陷的代码，与状态缓存机制相结合，可以提高历史状态的命中率，从而提高静态检测的效率。另外，为便于实施别名分析，本文还引入了一种基于内存区域的变量表示方法。

为验证本方法的有效性，我们实现了一个基于 CIL 静态分析的 C# 程序缺陷静态检测系统，并选取两个 C# 开源项目作为测试用例。实验结果表明本方法能够高效、准确地检测 C# 程序中常见的代码缺陷。检测系统还存在一定的不足，目前仅支持 6 种常见的缺陷类型，而且规则库还需要进一步地充实。在未来的工作中，我们将针对不足之处进行改进。一方面，通过扩展检测规则，提升系统的检测能力；另一方面，可以通过增加新的检测引擎及相应的检测规则实现对其他类型的 C# 代码缺陷的检测，完善系统的检测功能。另外，未来还可以通过简单地修改检测系统，增加对其他基于. NET 框架的编程语言特性的处理，并配置相应编程规范，实现对多种编程语言的支持。

## 参考文献


[1] Aho A V, Lam M S, Ravi S, et al. Compilers: principles, techniques, and tools(2nd Edition)[M]. Addison-Wesley Professional, 2007

[2] BlogEngine[OL]. http://www.dotnetblogengine.net/

[3] Brian C, Jacob W. Secure programming with static analysis [M]. Addison-Wesley Professional, 2007

[4] Alain D. Interprocedural may-alias analysis for pointers: beyond k-limiting [C]// Proceedings on PLDI. 1994: 230-241

[5] Seth H, Benjamin C, Xie Yi-chen, et al. A system and language for building system-specific, static analyses [C]// Proceedings on PLDI. 2002: 69-82

[6] Heine D L, Lam M S. A practical flow-sensitive and context-sensitive C and C++ memory leak detector [C]// Proceedings on PLDI. 2003: 168-181

[7] 梁彬，候看看，石文昌，等. 一种基于安全状态跟踪检查的漏洞静态检测方法研究与实施 [J]. 计算机学报，2009，32(5)：899-909

[8] Mono[OL]. http://www.mono-project.com/Main_Page

[9] SourceGrid[OL]. http://sourcegrid.codeplex.com/

[10] 夏一民，罗军，张民选. 基于静态分析的安全漏洞检测技术研究 [J]. 计算机科学，2006，33(10)：279-282

（上接第 219 页）

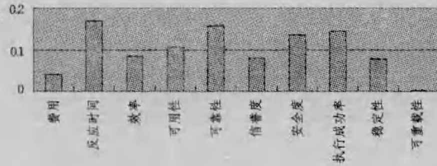


图 1　度量权重分布图

在上面各种度量参数分布和 Web 实验节点分布的基础上，系统采用 6 个监测点进行实际的预测实验。6 个监测点在发生变化的过程中采用逐渐变化的方法，采用过程变化量作为衡量系统 QoS 预测效果的标准，通过渐变过程，可以体现 QoS 预测效果的精细和准确度。采用传统方法进行预测的结果如图 2 所示。

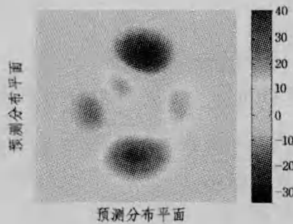


图 2　传统方法预测结果

从图 2 可以看出，采用传统的预测方法，可以实现各个变化量的预测，但是预测的结果在预测分布平面上很模糊，无法精细地辨别出渐变过程中各个阶段的变化量。

采用基于全域子空间分解挖掘的预测结果如图 3 所示。

图 3　全域子空间分解挖掘的预测结果

从图 3 可以看出，采用全域子空间分解挖掘的预测方法不但可以很好地将各个变化的监测点预测出来，而且可以实现渐变过程的精细化预测，预测结果在预测分布平面上很清晰，可以很好地辨别出渐变过程各个阶段的变化量。

所以，相对于传统的模糊预测结果，采用基于全域子空间分解挖掘的预测方法，预测结果更加精细，当然也更加准确。

**结束语**　研究了一种基于全域子空间分解挖掘的 QoS 准确预测方法。随着现代 Web 资源访问的迅猛增长，如何评价和预测是一个难点，所以 QoS 的准确预测是评判和选择最佳 Web 服务的一种重要标准。在传统的 QoS 预测方法中，经常采用普通时间平均值方法或者对各种评价参数进行一个简单的参数加权方法，此方法对于少量简单资源可以适用，但无法对大量 Web 服务下的资源进行准确预测，预测结果模糊。提出了一种基于全域子空间分解挖掘的 QoS 准确预测方法，采用全域分析的思想和子空间分解的方法，在全域对数据进行预处理，在子空间中对数据进行分解分析，提取数据的深层次特征，最后将两者进行有效的数据融合，实现对所分析数据的有效挖掘和准确预测。采用一组 Web 节点和拟定度量参数进行预测实验，结果显示，采用基于全域子空间分解挖掘的 QoS 预测方法，可以精确预测出渐变过程，结果准确，在 QoS 预测中具有广泛的应用价值。

## 参考文献

[1] 孔维梁. 基于二维 QoS 模型的 Web 服务组合[J]. 计算机科学，2008，35(11)：131-135

[2] 侯丽敏，张瑞坤. 基于 Agent 的 QoS 组播路由算法及仿真[J]. 计算机仿真，2011，28(1)：140-144

[3] 唐明董，姜叶春，刘建勋. 用户位置感知的 Web 服务 QoS 预测方法[J]. 小型微型计算机系统，2012，33(12)：2664-2667

[4] 华哲邦，李萌. 基于时间序列分析的 Web Service QoS 预测方法[J]. 计算机科学与探索，2013，7(3)：219-228

[5] 许利军，杨棉绒. QoS 组播路由的多种群遗传算法[J]. 科技通报，2012，28(5)：171-175

[6] 邹恩，刘泽华，方仕勇，等. 基于混沌遗传算法的组播路由优化研究[J]. 计算机工程，2011，37(3)：155-157

[7] Li Yan, Liu Yao, Zhang Liang-jie, et al. An exploratory study of Web services on the Internet [C] // Proceedings of the 2007 IEEE International Conference on Web Services (ICWS 07). Salt Lake City, UT, USA, 2007, Washington, DC, USA: IEEE Computer Society, 2007: 380-387

[8] Charif-Djebbar Y, Sabouret N. Dynamic service composition and selection through an agent interaction protocol[C]// Proceedings of the 2006 IEEE/WIC/ACM International Conference onWeb Intelligence and Intelligent Agent Technology Workshops (WI-IATW 06). Washington, DC, USA: IEEE Computer Society, 2006: 105-108

[9] Shao Ling-shuang. Research on quality ofWeb services management technologies [D]. Beijing: Peking University, 2009

[10] 曾欢，张灿，陈德元. 空间通信网中音视频传输的应用层 QoS 控制与测试方法[J]. 中国科学院研究生院学报，2011，28(1)：108-114

[11] 邵凌霜. 面向 Web Services 的服务质量管理技术研究[D]. 北京：北京大学，2009

[12] 张力娜，李小林. 一种基于 QoS 偏好的服务选择策略[J]. 重庆邮电大学学报：自然科学版，2013，25(6)：807-812