第 27卷第 3期 2010年 3月
计 算 机 应 用 研 究 Application Research of Computers
Vol 27 No. 3 Mar. 2010

基于超级块支配图插装的软件测试工具设计与实现 *

徐晓峰[a], 陈 艳[b], 李伊陶[a], 林晓鹏[a], 郭东辉[a,b]
(厦门大学 a.物理系; b.信息科学与技术学院, 福建 厦门 361005)

摘 要: 通过超级块支配图来分析软件测试探针的合理插装位置,可有效地减少插装探针数量,降低代码插装对程序的影响。基于超级块支配图的代码插装原理,设计一种针对 C语言的软件自动测试工具(SAT),介绍了该工具中词法语法分析器、静态分析器、代码插装器等主要功能模块的具体实现方案,同时对 SAT的插装性能进行了分析。

关键词: 代码插装; 覆盖测试; 超级块支配图
中图分类号: TP311 **文献标志码**: A **文章编号**: 1001-3695(2010)03-0923-05
doi: 10.3969/j.issn.1001-3695.2010.03.032

Design and implementation of software testing tool based on super block dominator graph

XU Xiao-feng[a], CHEN yan[b], LI Yi-Yang[a], LIN Xiao-peng[a], GUO Dong-hui[a,b]
(a. Dept of Physics, b. School of Information Science & Technology, Xiamen University, Xiamen Fujian 361005, China)

Abstract: This paper described the design and implementation of a coverage testing tool (SAT). It emphasized on the realization of main modules: lexer and parser, static analyzer, and code instrumenter. Compared to other tools that instruments each basic block, SAT used super block dominator graph to check which basic block should be instrumented so that both the number of instrumentation probes and runtime overhead of instrumentation are reduced effectively. Finally, used an example to show the functionalities of the tool as well as the discussed performance of SAT.

Key words: code instrumentation; coverage testing; super block dominator graph

软件测试是保证软件质量的重要手段,可分为手工测试和自动化测试。随着软件规模的不断扩大,手工测试已无法满足测试的需要,应用测试工具进行自动化测试能够完成许多手工测试难以实现的测试,达到有效缩短软件开发及测试时间、保证软件质量的目的[1],已成为软件质量和可靠性保证的关键技术手段。代码覆盖测试工具是目前软件测试的重要工具之一[2],它提供有效的覆盖信息用于量度测试完整性[3]和辅助测试员设计测试用例,从而提高测试效率。

代码覆盖测试工具通常是对被测试的元素[4](如语句、程序块、分支、谓词、分支 - 谓词等)进行探针插装,通过运行带有探针的被测程序获得程序运行的执行信息。但探针的插入会对被测程序产生负面影响,如使被测试程序的代码量增加、执行效率降低等。因此如何设计合理的插装策略、减少插装探针对被测程序的影响,是覆盖工具设计的关键问题之一[5,6]。近年来,多个文献中提到多种代码覆盖测试工具:文献[3,7~9]介绍了一款 C语言的覆盖测试工具 ATAC,该工具支持基本块、判定、C-uses、P-uses等多种覆盖准则;文献[10]介绍了基于程序插装的动态测试技术实现的 Safepro软件测试工具,具体讨论了动态测试的模型、数据流模型和动态跟踪数据的编码和解码技术、插装库设计与插装策略等内容,但这些工具都存在插装探针冗余问题;文献[6]介绍了一款利用程序控制流前支配树信息进行插装的测试工具 Dyninst,能有效地减少探针,但由于缺乏后支配树的信息,它的插装策略仍然存在冗余。而超级块支配图融合了前支配树和后支配树信息,可以有效地减少插装探针。

本文介绍了一个 C语言的块覆盖软件测试工具(SAT)的设计方案与实现,它采用超级块支配图对插装探针个数进行有效优化,减少了代码插装对被测程序性能的影响,同时提供图形化的界面,显示各个函数的块测试覆盖率,定位未覆盖的块,便于测试员直观地进行覆盖分析,以补充新的测试用例,提高测试覆盖率。

代码插装技术原理

SAT的插装策略是基于超级块支配图[11]性质实现的,因

收稿日期: 2009-06-17; **修回日期**: 2009-08-10 **基金项目**: 国家自然科学基金资助项目(60753001);国家教育部新世纪人才计划基金资助项目

作者简介:徐晓峰(1983-),男,福建晋江人,博士研究生,主要研究方向为软件测试自动化;陈艳(1981-),男,福建古田人,博士研究生,主要研究方向为实时嵌入式系统分析与测试;李伊陶(1984-),女,河南郑州人,硕士研究生,主要研究方向为通信与软件;林晓鹏,男,福建平潭人,博士研究生,主要研究方向为网络资源管理和任务调度;郭东辉(1967-),男,福建莆田人,教授,博导,主要研究方向为人工智能、计算机网络通信、集成电路设计自动化等(dhguo@xmu.edu.cn).

此本章首先对超级块支配图的相关定义和性质进行阐述。

. 超级块支配图的定义及性质

定义 1 基本块支配图是一个用二元组 $\langle N, E\rangle$表示的有向图，N是节点集合，E是边集合。支配图中的任一个节点 $n\in N$表示程序的一个基本块[12]；边 $e\in E$用有序的节点对表示，记为$\langle n_i, n_j\rangle$，表示 n_i直接前支配 n_j(或 n_i直接后支配 n_j)。

性质 1 基本块支配图上的祖先节点支配后裔节点。

证明 由基本块支配图定义可知。

定义 2 对于给定的基本块支配图 $G=\langle N, E\rangle$，图中的强连通分量的节点集合定义为超级块。

性质 2 一个超级块可能包含一个或多个基本块。

证明 由强连通分量的定义可知。

设基本块支配图中的超级块集合为 $\{S_m \mid m\leq k, k=$超级块总数$\}$，超级块 S_m 具有 C_m 个基本块，即 $S_m=\{n_{mi}\mid i\leq C_m\}$，可得：

性质 3 $S_m\subseteq N$ 且 $\bigcup_{l}^{k} S_m = N$, $S_m\cap S_n=\varnothing$ $(m\leq k, n\leq l, m\neq n)$。

证明 由超级块定义可知。

性质 4 在一次执行中，如果超级块 S_m 中的一个基本块被执行，那么 S_m 中其他基本块也被执行了。

证明 假设在一次执行中，超级块 $S_m\{n_{mi}\mid i\leq C_m\}$中的 n_{mi} 节点被执行。

a)若 $C_m=1$，结论成立。

b)若 $C_m>1$，任取 $n_{mi}, n_{mj}\in S_m$ $(n_{mi}\neq n_{mj})$，由定义 2得，n_{mi}与 n_{mj}强连通，即基本块 n_{mi}支配 n_{mj}，且基本块 n_{mj}支配 n_{mi}。根据基本块支配关系定义[12]：基本块 n_{mi}支配 n_{mj}，可知如果 n_{mj}被执行，则 n_{mi}也被执行；基本块 n_{mj}支配 n_{mi}，可知如果 n_{mi}被执行，则 n_{mj}也被执行。故可得，对于超级块中任意两个基本块，如果一个被执行，另一个也必然被执行。因此，超级块中的一个基本块被执行，那么超级块中其他基本块也被执行了。

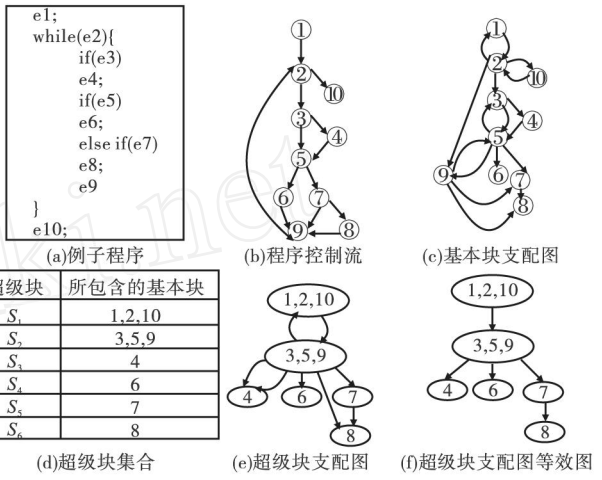
定义 3 如果超级块 S_m 中的一个基本块直接前支配(或者后支配)超级块 S_n 中另一个基本块，那么称超级块 S_m 直接支配超级块 S_n。

定义 4 超级块支配图(super block dominator graph, SBDG)可以用二元组$\langle S, D\rangle$表示，其中 S是节点集合，D是边集合。图中的节点 $S_m\in S$表示程序的一个超级块。边 $\langle S_m, S_n\rangle\in E$表示超级块 S_m 直接支配超级块 S_n。对于给定的超级块支配图，删除不影响其节点可达性的边[13]，即可得到它的等效图。以下本文中所说的超级块支配图均指其等效图。

性质 5 超级块支配图上，孩子节点执行了，意味着父节点也就执行了。

证明 假设存在超级块 $S_m=\{n_{mi}\mid i\leq C_m\}$，超级块 $S_n=\{n_{ni}\mid i\leq C_n\}$，且 S_m 是 S_n 的孩子节点。根据超级块支配关系定义，S_n 中必然存在一个基本块(可设为 n_{ni})直接前支配(或者后支配) S_m 中另一个基本块(可设为 n_{mi})。根据基本块支配关系定义，n_m如果执行，n_n就必然执行，且由性质 2可知，n_m如果执行，则 S_m 中所有的节点必然执行；n_n如果执行，则 S_n 中所有节点必然执行，故该性质得证。

以图 1(a)的例子程序为例，其程序控制流图如图 1(b)所示，根据支配节点算法[14]，可得其基本块支配图为图 1(c)。根据定义 2，图 1(c)中的强连通区域为超级块，如基本块 1、2、10组成超级块，详细的超级块集合如图 1(d)所示，且可得超级块支配图为图 1(e)，其等效图为图 1(f)。



(a)例子程序 (b)程序控制流 (c)基本块支配图

(d)超级块集合 (e)超级块支配图 (f)超级块支配图等效图

图1 超级块支配图的相关概念

. 插装方法

在普通的覆盖测试工具实现中，要获取各基本块的覆盖信息就必须对所有基本块都进行插装，如对图 1的例子，需要对基本块 1～10全部进行插装，共 10个探针。而利用以上超级块定义与性质，则只需要对一部分基本块进行插装即可获得所有基本块覆盖信息。依据超级块性质 4，要获得某超级块中基本块覆盖信息，只需在同一个超级块内插入一个探针。同时在超级块支配图上，依据性质 5可得，如果一个测试用例覆盖超级块 U的同时，还必须至少覆盖 U的一个孩子节点，那么 U就不需要插装，这些做法可减少冗余插装。例如图 1的例子程序，采用超级块支配图插装，则需插装的基本块为 1、4、7、8共四个探针。可见，采用超级块支配图的方式，可明显减少插装探针的数量。

工具的总体设计

SAT是一款利用超级块支配图插装实现的 C语言块覆盖测试工具，整体设计如图 2所示。主要功能组件有：

a)词法语法分析器。词法语法器读入预处理后的 C语言，构建程序语法分析树，并保存各个记号(token)的行列信息。其中 C语言经过预处理后已展开为无注释行，完成宏定义的替换，#include文件已经插到文件相应位置处，只保留条件编译为真的代码文件。

b)静态分析器。通过遍历语法分析树，根据程序结构将源程序按基本块为单元进行划分，并将基本块的静态信息按照一定的编码格式保存于静态文件(sat文件，数据编码如图 3(a))中，同时获取程序控制流信息，为自动生成测试结果作准备。

c)代码插装器。根据程序控制流信息获取各函数的超级

块支配图,并依据插装算法确定探针插装位置,在语法分析树的相应位置插入特殊的探针节点,最后从插装后的语法分析树中反解析出插装后的代码。

d)覆盖分析器。通过对比静态文件和跟踪文件获取覆盖信息,定位未覆盖代码,报告测试覆盖率。

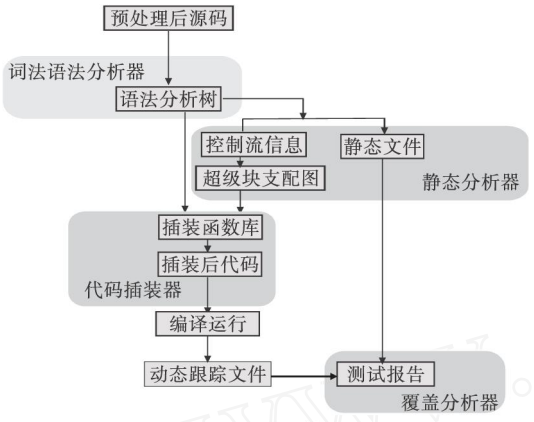
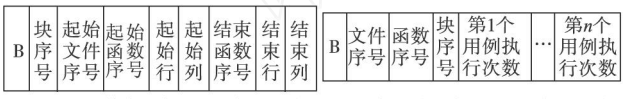


图2　STA的总体设计



(a)静态文件.sat的编码格式　(b)动态跟踪文件.trace的编码格式

图3　静态与动态文件编码格式

主要功能模块的实现

. 词法、语法分析器

词法语法分析器是代码覆盖测试工具的重要组件。首先它根据词法规则将源程序中的若干字符划分为若干记号[13],为了便于提取程序块的静态信息,词法语法分析器记录了各记号的静态信息(行列信息)然后根据记号识别相应的语法规则,并执行相应的语义动作,建立源代码的语法分析树。其工作及实现原理如图 4所示。

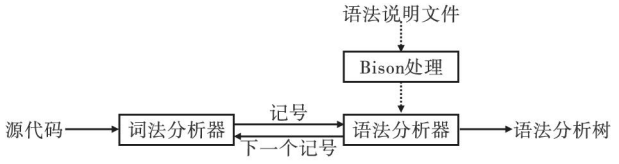


图4　SAT词法语法分析器实现和工作原理图

在 SAT中,词法分析器采用手工编程实现。语法分析器是通过编写 C语言语法说明文件,利用 Bison[15]工具处理后得到的。其中,语法说明文件中定义了语法结构规则以及语法分析树的语句,其核心是由一条或多条语法规则构成的规则段。该规则段定义了输入流应满足的语法规则以及相应的动作程序。

. 静态分析器

静态分析器通过扫描语法分析树,确定基本块开始和结束的边界,进行程序基本块的划分,划分算法如下所示:

a)确定入口语句,所用规则为:

(a)程序或者函数的入口点是入口语句;

(b)任何能由条件转移或无条件转移语句转移到的语句都是入口语句;

(c)紧跟在转移语句或条件转移后面的语句是入口语句。

b)函数调用语句单独作为一个基本块。

c)对于每个入口语句,其基本块由它和直到下一个入口语句(但不包含该入口语句)或程序结束的所有语句组成。

静态分析器将各个基本块的所在文件、行、列信息保存于静态文件中,同时保存基本块的控制流关系,以获取各函数的程序控制流图。

. 代码插装器

. . 探针位置的选择

从覆盖测试工具的使用目的上来看,插入的探针越多,可获得的有效覆盖信息就越详细。同时代码插装必然会对源代码程序的执行效率产生影响,所以要尽可能地避免插装冗余,通过选择合适的探针插装位置可以减少插装的冗余度。依据静态分析器获取的程序控制流信息,SAT可获得程序超级块支配图,并确定需要插装的基本块,即在超级块支配图中,对于给定的超级块 U,可以通过两条准则判定是否需要插装:a)如果 U的孩子节点少于两个,则 U需要插装;b)如果存在一条从程序入口到出口的路径,能够不经过超级块支配图中 U的任何孩子节点,则 U需要被插装。详细判定算法如下[12]:

```
probe(U)
{
if U has fewer than two children in the super block dominator graph
 then return true;
mark all nodes in the control flow graph as unvisited;
mark a respresentative basic block, r, of U as unvisited;
mark a representatives of the children of U in the super block dominator
graph as visited;
visit_predecessors (r);
visit_successors (r);
if both the entry and the exit nodes of the flow graph are marked as visi-
ted
then return true;
else return false;
}
visit_predecessors (n)
{
for each immediate predecessor, p, of node n in the flow graph do
    if p is not marked as visited then
    {
        mark p as visited;
        visit_predecessor(p);
    }
}
visit_successor ( n)
{
for each immediate successor, s, of node n in the flow graph do
    if s is not marked as visited then
    {
        mark s as visited;
        visit_predecessor(s);
    }
}
```

. . 插装探针的设计

针对需要插装的每一个基本块,SAT插装器在语法分析树

上选择合适的位置,进行探针函数的插装,最后将插装后的语法分析树解析成插装后的源代码。插装后的源码作为新的测试代码进行编译。编译时,插装探针函数以静态库的方式连入被测程序,生成带有插装探针的可执行文件;测试时,通过运行该可执行文件,插装的探针语句可将所监测块的执行情况按照一定编码格式保存于动态跟踪文件(.trace文件,数据编码如图 3(b))中。下面依据获取基本块动态跟踪信息的需要,讨论插装探针函数的设计与插装位置选择。

1)初始化探针,int level = int xSaT(int oldlevel, 0)

插入于每个函数的起始位置,返回变量 level来惟一标志所在函数层次,记录函数调用深度,用于解决函数调用时基本块覆盖信息的提取,识别待测元素所在函数层次关系。

2)块插装探针,int xSaT(int level, int blk)

其中 level为所在函数的层次;blk表示该块的序号,每执行一次则根据上文介绍的编码规则在动态跟踪文件中更新该块的被执行次数。对于顺序语句,探针插入于需插装的基本块之前;对于分支语句,为了使引入的探针语句不影响源代码语义,采用逗号运算符",",把探针语句和分支语句的条件表达式连接起来,构成逗号表达式,形式如下:

探针语句,条件表达式

该逗号表达式的计算顺序是从左到右,先计算探针语句,然后计算分支表达式,逗号表达式的值为最右边的条件表达式的值,从而插入后不会影响原代码的语义。

. 覆盖分析器

覆盖分析器负责读入静态文件和动态跟踪文件生成覆盖测试报告,包括以下两个功能:

a)定位未覆盖的代码段。依据超级块的性质 4和 5,并结合动态跟踪文件,可获取未覆盖的基本块,同时结合静态文件中的基本块行列信息,可定位为未覆盖的代码段。

b)测试覆盖率报告。对各个函数进行测试覆盖率计算,方法如下:

$$\text{测试覆盖率} = \frac{\text{至少被执行一次的基本块数量}}{\text{函数中基本块总数}} \times 100\%$$

应用结果分析

在完成 SAT系统设计与实现后,本文用一个实例展示 SAT的功能,并对 SAT的探针插装个数及探针插入对程序执行时间的影响进行了实验测试。

. 功能分析

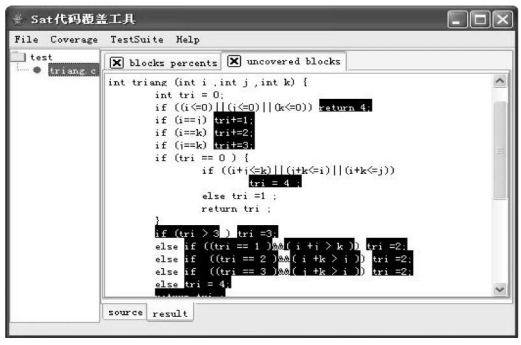
本文以三角形分类程序为例介绍 SAT的功能,源代码如下所示:

```
#include〈stdio.h〉
int triang (int i, int j, int k) {
    int tri = 0;
    if ((i<=0) ‖ (j<=0) ‖ (k<=0)) return 4;
    if (i==j)   tri++;
    if (i==k)   tri+=2;
    if (j==k)   tri+=3;
    if (tri == 0) {
        if ((i+j<=k) ‖ (j+k<=i) ‖ (i+k<=j))
            tri = 4;
        else tri =1;
        return tri;
    }
    if (tri > 3) tri =3;
    else if ((tri==1)&&(i+j>k)) tri=2;
    else if ((tri==2)&&(i+k>j)) tri=2;
    else if ((tri==3)&&(j+k>i)) tri=2;
    else tri = 4;
    return tri;
}
void main ()
{
    int a, b, c, t;
    printf ("enter 3 integers for sides of triangles \n");
    scanf ("%d%d%d", &a, &b, &c);
    t = triang(a, b, c);
    if (t==1) printf ("triange is scalene \n");
    else if (t==2) printf ("triange is isosceles\n");
    else if (t==3) printf ("triange is equilateral\n");
    else if (t==4) printf ("this is not triange \n");
}
```

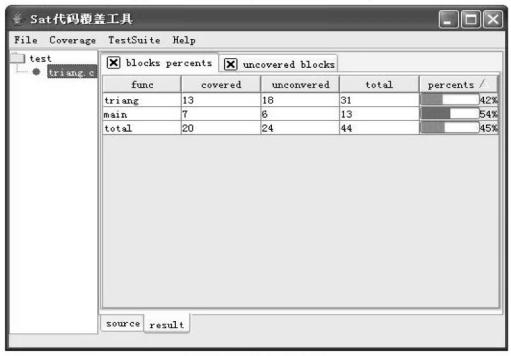
由于篇幅关系,这里列出插装后的部分代码,如下所示:

```
if ((i<=0) ‖ (xSaT(sat,2),(j<=0)) ‖ (xSaT(sat,3),(k<=0))) {
        xSaT(sat,4);
        return 4;
}
```

其中:xSaT是探针函数,sat为表示函数层次的变量。通过执行一个测试例子:输入整数 11、12、13,调用 SAT工具得到未覆盖代码如图 5(a)中深色部分代码,显示执行上述用例后测试覆盖率如图 5(b)所示。



(a)未覆盖代码(图中深色部分)



(b)测试覆盖率报告

图5 SAT的功能

. 性能分析

. . 探针数目对比

目前的代码覆盖工具(如 ATAC等)都是通过对每个基本块进行插装来获取覆盖信息。相比于这些工具,SAT采用了超级块支配图来选择插装探针的位置,在不丢失覆盖信息的情况

下减少了探针的个数。为此本文选择四组软件测试文献中常用的测试程序[7,16],将 SAT与 ATAC、Dyninst等工具的插装方式进行了实验比对,结果如表 1所示。从表中可以看出,SAT插入的探针个数最少,这将大大减少程序插装给被测程序造成的负面影响。

表 1　插装探针对比

被测程序	基本块个数	探针插装探针个数		
		ATAC	Dyninst	SAT
schedule	153	153	54	51
prinf_token	224	224	91	89
spiff	1 312	1 312	505	474
space	2 750	2 750	793	739

. .　执行时间对比

探针的引入,必然对被测程序的执行效率产生影响,本文选择 spiff程序,将 SAT与 ATAC工具进行执行时间实验,分别对比在无插装、使用 ATAC插装和使用 SAT插装三种情况下,执行相同的测试用例的执行时间,结果如表 2所示。从表中可以看出,与 ATAC相比,SAT在执行时间上减少了 31%。

表 2　执行时间比较

被测程序	无插装运行时间/s	ATAC插装执行时间/s	SAT插装后执行时间/s	时间减少/%
spiff	46.45	390.20	270.45	31

结束语

本文设计并实现了一种基于超级块插装的 C语言自动化测试工具 SAT,通过选定插装位置和优化设计插装策略,有效地减小了探针代码对程序运行性能的影响,并实现了未覆盖代码定位,测试覆盖率自动化分析。

参考文献:

[1] FEWSTER M, GRAHAM D. Software test automation[M]. [S.l.]: Addison-Wesley Professional, 1999.

[2] YANG Qian, LI J J, WEISS D. A survey of coverage based testing tools[C]//Proc of International Workshop on Automation of Software Test 2006: 99-103.

[3] HORGAN J R, LONDON S. A data flow coverage testing tool for C[C]//Proc of the 2nd Symposium on Assessment of Quality Software Development Tools. 1992: 2-10.

[4] FRANKL P G, WEYUKER E J. An applicable family of data flow testing criteria[J]. IEEE Trans on Software Engineering, 1998, 14(10): 1483-1498.

[5] LI J J, WEISS D M, YEE H. An automatically-generated run-time instrumenter to reduce coverage testing overhead[C]//Proc of the 3rd International Workshop on Automation of Software Test. 2008: 49-56.

[6] TIKIR M M, HOLLINGSWORTH J K. Efficient instrumentation for code coverage testing[C]//Proc of ACM SIGSOFT International Symposium on Software Testing and Analysis. 2002: 86-96.

[7] LYU M R, HORGAN J R, LONDON S. A coverage analysis tool for the effectiveness of software testing[J]. IEEE Trans on Reliability, 1994, 43(4): 527-535.

[8] HORGAN J R, LONDON S. Data flow coverage and the C language[C]//Proc of Testing, Analysis, and Verification Symposium. 1991: 87-97.

[9] HORGAN J R, LONDON S, LYU M R. Achieving software quality with testing coverage measures[J]. Computer, 1994, 27(9): 60-69.

[10] 孙昌爱,金茂忠. 基于程序插装的动态测试技术实现[J]. 小型微型计算机系统, 2001, 22(12): 1475-1479.

[11] AGRAWAL H. Dominators, super blocks, and program coverage[C]//Proc of the 21st Symposium on Principles of Programming Languages. 1994: 25-34.

[12] AHO A V, SETHI R, ULLMAN J D. Compilers: principles, techniques, and tools[M]. [S.l.]: Addison Wesley, 1986.

[13] MOYLES D M, THOMPSON G L. An algorithm for finding a minium equivalent graph of a digraph[J]. Journal of the ACM, 1969, 16(3): 455-460.

[14] LENGAUER T, TARJAN R E. A fast algorithm for finding dominators in a flow graph[J]. ACM Trans on Programming Languages and Systems, 1979, 1(1): 121-141.

[15] Bison[EB/OL]. http://www.gnu.org/software/bison/.

[16] WONG W E, QI Y, ZHAO L. Effective fault localization using code coverage[C]//Proc of the 31st IEEE Computer, Software, and Applications Conference. 2007: 449-456.

(上接第 922页)

[2] SIVASHANMUGAM K, VERMA K, SHETH A, *et al*. Adding semantics to Web services standards[C]//Proc of International Conference on Web Services. Washington DC: IEEE Computer Society, 2003.

[3] TSAI W T, HUANG Qian, XU Jing-jing, *et al*. Ontology-based dynamic process collaboration in service-oriented architecture[C]//Proc of IEEE International Conference on Service-Oriented Computing and Applications. 2007: 39-46.

[4] MARTIN D, BURSTEIN M, McDERMOTT D, *et al*. Bringing semantics to Web services with OWL-S[J]. World Wide Web, 2007, 10(3): 243-277.

[5] ASLAN M, AUER S, SHEN J. From BPEL4WS process model to full OWL-S ontology[C]//Proc of the 3rd European Semantic Web Conference. Berlin: Springer, 2006.

[6] BORDBAR B, HOWELLS G, EVANS M, *et al*. Model transformation from OWL-S to BPEL via SiTra[C]//Proc of the 3rd ECMDA-FA. Berlin: Springer, 2007: 43-58.

[7] ImportNET D1.3, D2.2[EB/OL]. [2009-05-15]. http://www.importnet-project.org/.

[8] GANGEMI A, MIKA P. Understanding the semantic Web through descriptions and situations[C]//Proc of CoopIS/DOA/ODBASE. Berlin: Springer, 2003: 689-706.

[9] FARQUHAR A, FIKES R, RICE J. The ontolingua server: a tool for collaborative ontology construction[EB/OL]. [2009-07-15]. http://ksl.stanford.edu/KSL_Abstracts/KSL-96-26.html

[10] 唐杰,梁邦勇,李涓子,等. 语义 Web中的本体自动映射[J]. 计算机学报, 2006, 29(11): 1956-1976.

[11] 刘博,范玉顺,倪悦. 协同业务过程中的语义一致性[J]. 清华大学学报:自然科学版, 2009, 49(4): 494-497.

[12] DAML[EB/OL]. [2009-07-15]. http://www.daml.org/services.

[13] MILANOVIC N, MALEK M. Current solutions for Web service composition[J]. IEEE Internet Computing, 2004, 8(6): 51-59.

[14] DOLCE[EB/OL]. [2009-07-15]. http://www.loa-cnr.it/DOLCE.html