

## 嵌入式软件语句覆盖率测试插桩技术

孙红利,王忠民,王文浪

(西安邮电学院 计算机学院,西安 710061)

(sunhongli.apple@163.com)

**摘要:**针对基于宿主机的嵌入式软件测试,提出一种单元测试中通用的语句覆盖率测试方法,通过插桩技术,采用向源代码插桩实现语句覆盖率测试。设计了测试代码的实现算法,通过测试代码可以自动完成向被测代码插桩。这些方法被成功地应用到笔者所在项目组开发的嵌入式软件仿真测试平台 ARMtest 上。利用这些方法,在嵌入式硬件系统未完成开发之前,可通过宿主机环境和仿真环境及时发现嵌入式软件开发初期的一些不足并加以完善。

**关键词:**嵌入式软件;宿主机环境;程序插桩;语句覆盖;单元测试;仿真测试

**中图分类号:** TP311.56 **文献标志码:** A

## Instrumentation technology for embedded software statement coverage testing

SUN Hong-li, WANG Zhong-min, WANG Wen-lang

(School of Computer Science and Technology, Xi'an University of Posts and Telecommunications, Xi'an Shaanxi 710061, China)

**Abstract:** For the host-based embedded software testing, a general method for statement coverage testing in the unit test was proposed. The method was implemented by piling technology before compiling the source code. The relevant algorithm of the test code, with which the piling technology would be automatically performed, was designed. And these approaches were eventually applied to the platform ARMtest for embedded software emulating and testing. By these approaches, the problems in the early embedded software development can be found and improved through the host-based environment and simulation environment.

**Key words:** embedded software; host-based environment; code instrumentation; statement coverage; unit test; emulating and testing

### 0 引言

随着嵌入式设备在越来越多的领域中得到使用,如何对日益复杂的嵌入式软件进行快速有效的测试已经成为目前的一大热点问题。从国内外现状来看,对嵌入式软件测试的研究,大多着重关注嵌入式软件的调试工作,即 Debugging 阶段,而鲜见系统的、全面的测试技术探讨。随着嵌入式软件质量保证要求的提高,对嵌入式软件的测试显然已经不能停留于仅仅通过调试就可以了,必然需要有更加全面的、系统化的测试技术支持<sup>[1]</sup>。传统的嵌入式软件测试一般采用手工的、插桩目标代码的方式。由于目标代码中语法、语义信息不完整,有时会造成对错误的分析定位不够准确;采用手工的方式进行,效率低、效果差,存在漏洞和隐患<sup>[2]</sup>。同时由于嵌入式软件需要采用交叉开发的方式,嵌入式软件测试也存在同样的问题,而语句覆盖率测试是软件测试最基本的方法,因此在开发初期,实现在没有目标机的环境下自动完成对嵌入式软件的语句覆盖测试是非常必要的。

本文在对现有的测试方法和插桩技术的总结分析基础上,改进了传统的测试方式,采用向源代码插桩的插桩方式提出了一种通用的嵌入式软件语句覆盖率测试方法,实现在没有目标机的环境下通过测试代码自动完成对嵌入式软件高效、准确的测试,克服了传统手工测试效率低、效果差的缺

点;由于源代码中包括全部的语法、语义信息,对源代码的分析能够达到最高的准确度。

### 1 嵌入式软件测试插桩原理规则

利用嵌入式软件仿真测试环境对嵌入式软件进行测试是目前国内外公认的一种有效的测试方法。本文针对嵌入式软件,提出了一种基于宿主机的测试模型,如图1所示。

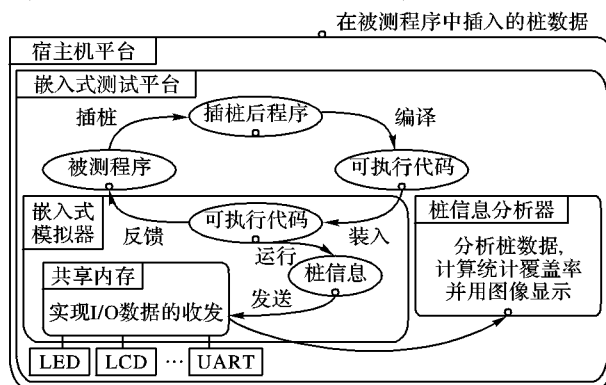


图1 基于宿主机的嵌入式软件测试模型

在实际应用中,嵌入式软件的运行离不开具体的外围设备。因此,在宿主机环境下,还必须对已有的外设以及即将出现的外设进行仿真。利用软件模拟中断机制,实现 ARM 内核与模拟外设之间的数据交互共享,将每个外设对应建立的共

收稿日期:2010-03-31;修回日期:2010-05-26。

基金项目:陕西省自然科学基金基础研究计划项目(SJ08-ZT14);西安市科技计划项目(CXY08017(1))。

作者简介:孙红利(1984-),女,陕西延安人,硕士研究生,主要研究方向:嵌入式系统、嵌入式软件测试;王忠民(1967-),男,陕西蒲城人,教授,博士,CCF会员,主要研究方向:嵌入式系统、智能机器人;王文浪(1964-),男,陕西渭南人,高级实验师,主要研究方向:嵌入式软件测试。

享内存作为数据的中转站,其交互机制如图 1 所示。至此,就建立了一个完整的仿真平台。对该平台进行完善,添加各种测试类型,最终实现一个适合于嵌入式软件的仿真测试环境。

通过仿真测试环境实现在只有宿主机环境的情况下完成对嵌入式软件全面、有效的测试,解决了在测试过程中由于宿主机和目标机的交联使用所出现的数据信息传输问题。在测试实现过程中,首先对被测代码进行插桩,运行插桩后文件获得的桩信息经过桩信息分析器可计算出覆盖率,并最终通过仿真平台完成流水灯等外设的模拟工作。

在整个测试实现过程中向源程序的插桩处于至关重要的地位,下面就本文所用的插桩原理与规则做详细说明。

### 1.1 程序插桩的思想

程序插桩是借助向被测程序中插入操作来实现测试目的的方法,是软件白盒测试中的一种基本的测试手段<sup>[3]</sup>。通过插桩,可以进一步了解程序执行过程中的一些动态特性<sup>[4]</sup>。本系统采用的插桩技术在动态测试之前根据测试人员所选的测试方式选择不同的插桩程序完成对源程序的自动插桩,插入已经设计好的桩函数,在动态测试阶段通过运行插桩后的程序可以获得所需要的程序运行信息。例如在单元测试的语句覆盖率测试中通过运行插桩后程序可以获得可执行语句中所执行到行的行号及该行的执行情况。本文所述的插桩技术支持如下的覆盖率测试信息<sup>[5]</sup>:

$$\text{语句覆盖率} = \frac{\text{运行的语句数}}{\text{总的可执行语句数}} \times 100\%$$

具体的插桩实现如图 2 所示。



图2 插桩实现流程

### 1.2 程序插桩的规则

#### 1.2.1 插桩点的选择

插桩是通过在所选的插桩点插入桩函数实现的。其中插桩点的选择是插桩技术必须考虑的关键问题。一般情况下,插桩点位置的选择应该满足在能够充分采集到所需信息的情况下使得代码的膨胀率最小,对程序的影响最小。本文所述测试方法,在被测程序所有的可执行语句中选择插桩点,对于一些声明语句及左右大括号等都不进行插桩。由于插桩是在保持源程序逻辑完整性和可执行性的情况下进行,所以实际测试过程通常在下面一些部位插桩:

- 1) 在第一个函数体之前插入一些初始化语句及桩函数;
- 2) for、do、do while、do until 等循环语句处;
- 3) if、else if、else 及 end if 等条件语句各分支处;
- 4) 输入输出语句之前;
- 5) 函数调用语句之前;
- 6) return 语句之前;
- 7) goto 语句之前。

#### 1.2.2 桩函数的设计

插桩操作插入的桩函数首先要是由与被测试程序相同的

语言编写而成的,满足源代码函数的语法规则,并且是能够编译通过的。另外,在设计桩函数时,要考虑的一个重要问题就是经过插桩操作,要获得程序执行过程中的哪些信息。其中测试类型不同所设计的桩函数也可能不同。本文所设计的桩函数 stub() 是专门针对单元测试的语句覆盖率测试的,通过所设计的桩函数可以采集到所执行行的行号和本行所执行的次数以及代码执行过程中的覆盖率情况,在必要时也可以统计出行执行的时间。在第 3 章算法描述中对桩函数做了具体的描述。

#### 1.2.3 插桩位置的确定

在程序插桩中除了要考虑插桩规则中所述的前两个问题外,插桩位置的确定也是一个至关重要的问题。通常针对不同的语句格式插桩位置会有所不同,所有的插桩位置选择都必须建立在确保插桩后的程序可以正确运行的基础上。

## 2 算法设计

针对基于宿主机的嵌入式软件测试,单元测试中的语句覆盖率测试算法描述如下。

步骤 1 构造初始化函数 Init() 及桩函数 Stub(), 为后面面向被测程序插桩做准备。

Stub 函数体中包含下列的语句:

```
LineNum chaLine;
ExeTime[ chaLine] ++;
Line[ chaLine] = LineNum << 16 | ExeTime[ chaLine];
```

相当于一个计数器的功能,当程序每执行到一处时,该点的执行次数将会加 1。

Init() 中包含对 LineNum、chaLine、ExeTime[] 等 Stub() 函数体中出现变量的初始化,以保证插桩后的文件可以正确运行。其中的数组 Line[] 是由一个整型数和一个整型数组合并而成的,LineNum 表示执行到行的行号,ExeTime[] 存放该行的执行次数。

此处的初始化语句和桩函数在插桩过程中都要被插入到被测测试程序中。

步骤 2 打开被测程序,对被测程序进行初次扫描,在扫描过程中按行读取源程序,统计被测试程序的总行数 ZLineNum 及被测测试程序中所有函数名的个数 fuc,并且找到 main 函数体及其下一个函数体中函数名出现的位置,将其分别记为 Lmain 和 LNmain,为后面面向 main 函数体插桩做准备。

在读取过程中用语句“ZLineNum++”记录总行数;用“Lmain = ZLineNum;”和“LNmain = ZLineNum;”分别记录 main 函数体及其下一个函数体中函数名具体所在行的行号,下面的语句则实现 main 函数体中函数名具体位置的记录。

```
fgets( buf, 1024, fp);
ZLineNum++;
if( ( index( "(", buf) != -1) && ( identf( buf) ) &&
(index( ";", buf) == -1) && ( index( "#", buf) == -1) )
fuc++;
if( strstr( buf, str_main) || strstr( buf, str_main) )
Lmain = ZLineNum;
```

步骤 3 在上次扫描的基础上再次扫描被测程序,记录所有的函数名所在位置,并将第一个及最后一个函数名所在行的行号分别赋值给 Fhshm 和 Lhshm。为单元测试的语句覆盖率测试做准备。其中用 Rfuc 来记录本次扫描所统计到的函数体个数。

```

if (Rfuc == 1)
    Fhsh = LineNum;
if (Rfuc == fuc)
    Lhsh = LineNum;

```

步骤4 对被测程序进行第三次扫描,分析其包含的所有语句,确定被测程序中所有的可执行语句。排除所有不插桩的情况并做记录。通常不插桩的情况如下:

- 1) 所有的声明语句所在行;
- 2) 左右大括号所在行;
- 3) 函数名所在行;
- 4) 空格行。

设置变量 Nstub,遇到上述所有的情况,将 Nstub ++;最后从总行数中将其排除即可得所有需要插桩语句的总行数 ZELine。

在这一步中要对所有不插桩的情况进行详细的分析并作处理,包括空格行及声明语句的处理等。

步骤5 最后一次扫描源程序,此次扫描过程中对被测程序进行边处理边将其写入插桩后文件的操作。在这一阶段需要做下列几个工作。

1) 结合上述插桩点选择一节中的内容,对所有需要插桩的语句做统计。对所有出现 for、do、do while、do until 等循环语句处及 if、else if、else 及 end if 等条件语句各分支处做标记,按照不同的语句类型做不同的标记。

2) 根据上面几步的标记,将所有不插桩的行直间写入插桩后的文件中;对于需要做信息采集的行,根据 1) 的标记选择插桩方式进行插桩;根据步骤2中标记的 main 函数体中函数名的位置插入桩语句,通过该语句可以统计到被测程序中所有的可执行语句的总行数。

通过运行插桩后的文件可以将需要采集的信息发送到共享内存,最后由分析程序从共享处获得信息,并按照式(1)对覆盖率进行实时性的计算。

上述算法实现的五个步骤中,前四步都可以当成是程序插桩的准备阶段,桩函数插入在步骤5中实现,并最终在分析程序中完成覆盖率的计算。

### 3 测试实现

下面以 LED 流水灯显示实验为例,来说明本文所提出的基于宿主机的嵌入式软件单元测试方法在嵌入式软件仿真测试平台 ARMtest 上的运用。针对流水灯程序,首先根据选择的覆盖率测试方式进行插桩,插桩后的程序运行结果,经过分析程序分析统计后即可得所需的语句覆盖率。如图3所示选择单元测试的语句覆盖率测试。

在插桩过程中要按照第2章所介绍的插桩思想对其进行插桩。首先根据插桩规则中所讲的插桩点选择方法,选出流水灯程序中所有的插桩点,例如其中的 for 语句和 while 语句:

```

for(i=0; i<51; i++);
while (1)

```

这些语句将被视为可执行语句中需要插桩的语句。还有其中的函数调用语句“DelayNS(50)”及“IOICLR=((LED\_TBL[i])<<18);”等也是需要采集信息的语句。接着按照插桩位置的选择原则选择插桩语句处桩函数的具体插入位置,例如上面所选的两句插桩位置分别选择在 for 语句和 while 语句下一行的大括号之后;而对于函数调用语句“DelayNS(50)”,按照插桩位置选择原则桩函数直间插在这

句之后就可以。

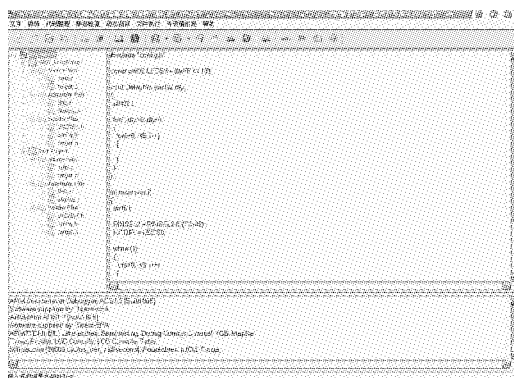


图3 ARMtest 测试平台

在进行完上述两步选择后,按照第2章算法描述中的步骤对流水灯程序进行插桩。首先由第2章中的步骤2统计出流水灯程序的总行数 ZLineNum = 47 及函数个数 fuc = 2,同时可以标记出各函数体出现的位置 fuc1 = 5, fuc2 = 29。按照上述标记对流水灯程序中所包含的两个函数体中的内容分别逐行判断,并对其中所有的不插桩行做统计,由上述所得的数据可以得到需要插桩的语句总数。例如我们要对其中的 main() 函数体进行单元测试的语句覆盖率测试,则最后统计出其中所有需要插桩的语句有第33、34、36、38、40、41、42、43、46行, ZLine = 9。由语句覆盖率计算公式及运行插桩后文件所得的结果可得语句覆盖率为 100%,如图4所示。

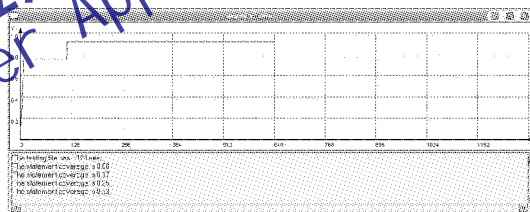


图4 语句覆盖率

图5为上述实验的流水灯程序在仿真测试环境下的显示结果。

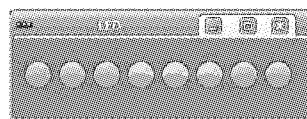


图5 流水灯实现

实验结果表明本文所述的单元测试语句覆盖率测试方法可以成功应用到嵌入式软件测试当中,完成在只有宿主机的情况下通过仿真环境对嵌入式软件进行快速、准确的测试。

在上述流水灯实验中插桩过程是完全自动完成的。传统的手工测试方式需要用户手工在源程序需要获取程序运行信息的地方写入信息采集代码段完成代码插桩工作,然后编译链接程序时需将包含覆盖信息采集功能函数的库链接进来,比较繁琐和低效;而且在不进行测试时,需要删除插入的代码或使用宏定义方式屏蔽插入的代码。而采用本文所述的自动插桩方式用户只需进行一些设置和选择,测试代码便可在编译链接时自动插入信息采集代码段,并自动链接相关库函数生成可执行文件,效率会明显提高;此外用户能通过测试软件对源程序进行词法语法分析来确定插桩代码的类型、位置和多少;插入过程、编译链接过程均无需用户介入,测试过程比较简单。

(下转第2744页)



实施变迁  $T_{reju}$  时,系统会主动保持缓冲区内的请求,因此待系统进入正常的状态后,可以继续处理请求。

根据图 3 和文献[6]的相关定理可知,任何具有有穷个变迁的连续时间的 SPN 同构于一个一维连续时间马尔可夫链。图 9 给出了与图 8 同构的 MC 模型,  $Q_i (i = 1, 2, 3, 4, 5)$  表示系统的状态。表 1 给出了 SPN 图与 MC 图的对应关系。

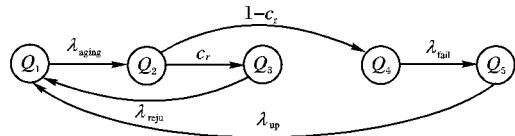


图 9 与图 8 同构的 MC 模型

表 1 图 8 与图 9 的对应关系

状态	$P_{healthy}$	$P_{aging}$	$P_{rejuvenation}$	$P_{undetected}$	$P_{failed}$
$Q_1$	1	0	0	0	0
$Q_2$	0	1	0	0	0
$Q_3$	0	0	1	0	0
$Q_4$	0	0	0	1	0
$Q_5$	0	0	0	0	1

根据图 8 还可以建立系统的可达图,如图 10 所示。椭圆表示标识集,弧线表示标识集之间的可能变迁,对应的变迁名称标注在旁边。标识集从(1)到(5)进行编号。(1)和(4)是工作状态;(2)由于有瞬时变迁,所以是一消失标识;(3)是 Rejuvenation 状态;(5)是失效状态。因此系统的稳态概率即为标识集(1)和(4)的稳态概率之和。

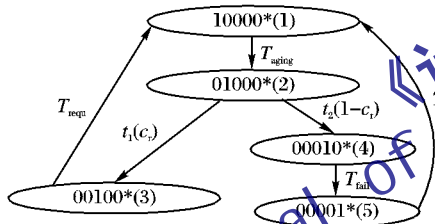
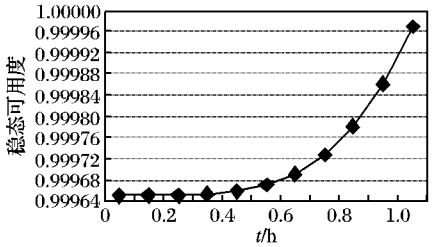


图 10 与图 8 对应的可达图

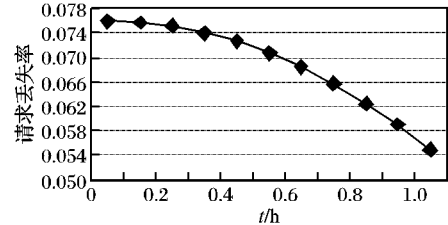
通过给出具体的变迁概率和缓冲区容量,可以对上述模型进行定量分析。假设变迁概率分别为:  $\lambda_{aging} = 1/500$ ,  $\lambda_{reju} = 40$ ,  $\lambda_{fail} = 1/500$ ,  $\lambda_{up} = 5$ ,  $\lambda_{requ} = 0.8$ ,  $\lambda_{service} = 1.0$ , 单位为  $h^{-1}$ 。仿真实验结果如图 11 所示。

根据仿真实验结果可知,系统的稳态可用性  $A_{ss}$  随预测概率  $c_r$  的增加而单调增加,服务请求丢失率  $R_{loss}$  随  $c_r$  的增加而单调减少。这种结果说明,当 Java 服务处于老化状态时,实施 Rejuvenation 策略将能够有效改善 Java 服务的稳态可用性。提高 Java 服务老化的预测率,可以进一步提升 Java 服务的稳

态可用性。



(a) 稳态可用性变化情况



(b) 请求丢失变化情况

图 11 仿真结果

4 结语

通过建模分析可知,实施软件再生策略可以有效改善软件系统可用性,通过提高软件老化预测率,可以进一步提升软件系统可用性。本课题将在如何提高 Java 服务老化的预测率上作进一步的研究。

参考文献:

[1] COTRONEO D, ORLANDO S, RUSSO S. Failure classification and analysis of the Java virtual machine[C]// The 26th International Conference on Distributed Computing Systems. Washington, DC: IEEE Computer Society, 2006: 17.

[2] GROTTKE M, MATIAS R, TRIVEDI K S. The fundamentals of software aging[EB/OL]. [2009-11-20]. <http://www.grottke.de/documents/FundamentalsOfSWAging.pdf>.

[3] KOLETTIS N, FULTON N D. Software rejuvenation: Analysis, module and applications[C]// Proceedings of 25th IEEE International Symposium on Fault-Tolerant Computing. Washington, DC: IEEE Computer Society, 1995: 381-390.

[4] 郑霄. 超级计算机的可用性评估研究[D]. 郑州: 信息工程大学, 2009.

[5] CHEN DEQING, MESSER A, BERNADAT P, et al. JVM susceptibility to memory errors[EB/OL]. [2009-11-22]. <http://www.cs.wm.edu/~riska/paper-jvm.pdf>.

[6] GARG S, HUANG Y, KINTALA K, et al. Minimizing completion time of a program by checkpointing and rejuvenation[C]// Proceedings of 1996 ACM SIGMETRICS Conference. New York: ACM, 1996: 252-261.

(上接第 2740 页)

4 结语

本文针对嵌入式软件测试这个热点问题,提出了一种基于宿主机的嵌入式软件测试方法;在传统的插桩技术基础上针对语句覆盖率测试提出了一种通用的程序插桩技术,实现在嵌入式软件仿真测试环境下对软件进行有效、快捷的测试,最终得到所需的语句覆盖率。该测试方法已经被成功地应用到嵌入式软件测试平台 ARMtest 中。大量例子表明在只有宿主机环境的情况下,针对单元测试的语句覆盖率测试,本文所提到的插桩技术是可行的。

参考文献:

[1] 奚雪峰. 嵌入式软件测试技术研究和典型案例实现[D]. 南京: 东南大学, 2004.

[2] 孙昌爱, 靳若明, 刘超, 等. 实时嵌入式软件的测试技术[J]. 小型微型计算机系统, 2000, 21(9): 920-924.

[3] 佟伟光. 软件测试[M]. 北京: 人民邮电出版社, 2008: 46-47.

[4] 刘玉东. 通信软件结构测试关键技术的研究及插桩器实现[D]. 北京: 北京交通大学, 2007.

[5] 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.

[6] 张丽. 基于嵌入式系统的软件结构覆盖测试技术[J]. 舰船电子, 2005(3): 63-66.