c/c++单元测试底层模拟技术①

刘跃勇[1], 王 彤[2], 罗 静[1]

[1](华南农业大学 信息学院, 广州 510642)
[2](广州凯乐软件技术有限公司, 广州 510630)

摘 要: 单元测试的目标是覆盖代码单元的功能逻辑, 内部输入是单元测试的关键问题. 目前大多采用打桩技术解决内部输入问题, 但打桩会造成函数失真, 而且难于解决底层函数不可控、难于初始化, 静态输入, 中断输入等内部输入问题. 针对这些问题, 提出了 c/c++单元测试底层模拟技术, 在测试用例中模拟、控制子函数的行为, 使得底层函数产生的数据像参数一样可以在用例中设置, 并且可以模拟局部数据和中断, 解决了打桩难于实现的多种内部输入问题.

关键词: 单元测试; 内部输入; 打桩; c/c++; 底层模拟

Underlying Simulation for c/c++ Unit Testing

LIU Yue-Yong[1], WANG Tong[2], LUO Jing[1]

[1](School of Information, South China Agricultural University, Guangzhou 510642, China)
[2](Guangzhou Kailesoft Co. Ltd, Guangzhou 510630, China)

Abstract: The goal of unit testing is to cover the functional logic of code unit, and internal inputs is the key of unit testing. Stub code has been used in most of the tools for c/c++ unit testing existed to solve the problem of internal inputs, while stub code may cause distortion of function, and it is difficult to solve these problems such as the uncontrollability and hard-initializing of underlying functions, the static inputs, the interrupt inputs, etc. To solve these problems, this paper proposed the underlying simulation technology for c/c++ unit testing, simulating and controlling the behavior of the sub-function in test cases, making the data generated by underlying function can be set in cases as the parameters, and it can simulate the local data and interrupt, avoid wasting time to write stub code, promote the testing efficiency.

Key words: unit testing; internal input; stub code; c/c++; underlying simulation

随着计算机技术的不断发展, 软件产业不断壮大, 软件质量成为产品质量不可或缺的组成部分, 而软件测试是软件质量保障的关键技术. 软件测试应贯穿于整个软件的开发过程, 主要包括单元测试、集成测试、系统测试、验收测试和配置审计等[1]. 成功的单元测试需要覆盖被测函数包括外部输入和内部输入的所有分类, 由于内部输入无法像全局变量、参数等外部输入一样在用例中设置, 而目前存在的针对于c/c++程序的测试工具 CppUnit、C++ Test、gtest 等通过打桩来解决内部输入问题的方法不仅会造成函数的失真, 而且难于解决多种内部输入问题, 同时需要花费大量的时间去编写桩代码, 因此, 本文提出了一种 c/c++单元测试底层模拟技术, 解决了前述各种用打桩难于实现的内部输入问题, 提高测试效率, 以保证软件质量, 大幅度降低后期测试和维护成本以提高开发商竞争力.

1 单元测试

单元测试[2-4]是完成对软件设计的最小单位正确性检验的测试工作. 单元测试主要依据是软件详细设计文档, 其目的是发现在程序单元内部所有重要的控制路径可能存在的各种错误. 单元测试需要进行功能、接口、执行路径、局部数据结构、语句和分

① 收稿时间:2012-03-31;收到修改稿时间:2012-05-15

支等各种覆盖、错误处理能力、边界等方面的测试. 一般采用白盒测试方法, 辅之以黑盒测试方法.

目前已有的 c/c++单元测试工具主要有: C++ Test、CppUnit 等. C++ Test[5]是 Parasoft 公司提出的一种 c/c++单元测试工具, 自动测试任何 c/c++类、函数或部件. CppUnit[5]是一种移植自 Java 单元测试框架 JUnit 的开源测试框架, 广泛应用于 c++单元测试.

上述的 c/c++单元测试工具都是采用打桩技术来解决内部输入问题. 桩, 或称桩代码, 是指用来代替关联代码或者未实现代码的代码. 打桩就是编写或生成桩代码. 打桩的目的主要有: 隔离、补齐、控制. 隔离是指将测试任务从产品项目中分离出来, 使之能够独立编译、链接、运行; 补齐是指用桩来代替未实现的代码; 控制是指在测试时, 人为设定相关代码的行为, 使之符合测试要求.

2 内部输入

单元测试是针对代码单元的独立测试, 一个函数, 在调用了底层函数的情况下(底层函数可能不存在、不可控、难于初始化、不得不隔离、甚至有错误), 如何能够独立测试? 而底层函数的输入, 可视为函数的内部输入, 另外局部静态输入、中断输入也属于内部输入. 因此, 解决内部输入问题, 是能顺利进行代码单元独立测试的关键.

C++ Test 和 CppUnit 都运用打桩来解决内部输入问题. C++ Test 中的桩函数分为三类[5]:

① 自定义桩函数: 用户自定义的桩函数, 自定义的桩函数以 “CppTest_Stub_”作为前缀.

② 安全桩函数: 当代码中使用到 rmdir(), remove(),rename()等 “危险”的函数时, C++ Test 将自动生成安全桩函数, 用以替换 “危险”函数.

③ 自动生成的桩函数: C++ Test 中提供了一个测试配置, 用户可通过该配置针对所选择源文件或者原工程自动生成桩函数.

CppUnit 中编写桩代码时, 可根据用例名来输入合适的值. 通过这种命名法, 可以解决简单的由于调用桩函数每次返回值相同而造成的失真问题, 但不能解决复杂的失真, 例如, 桩代码在同一用例中被多次调用, 而每次要求的输出不同; 或者桩与被测函数的关系的多对多的, 即一个桩函数被多个不同被测函数调用时, 桩输出与用例之间的关系就难以维护了. 如有函数 GetTemperature()用来获取环境温度, 调用的是实际代码, 而不是桩代码, 而真实的环境温度在短时间内很难大幅度变化, 这就遇到了测试中的不可控问题. 如果想另外编写桩代码来代替实际代码, 就会遇到很大的麻烦, 例如, 底层函数位于同一个文件或同一个类, 通常需要用编译条件来区分实际代码和桩代码, 不仅实现困难, 而且污染了产品代码. 对于以下的被测函数:

```
int AddPerson(PERSON* pData, CPersonMap* map)
{
    if(map->Search(&pData->name))
        return 0;
    map->Add(pData);
        return 1;
}
```

如果需要 map->Search(&pData->name)(搜索 pData->name 是否存在) 返回 true, 就必须保证 pData->name 存在于被搜索的空间中, 一般可以修改被搜索空间, 将pData->name的值先加入到其中, 同时还需要倒推到外部输入: 通过参数传递 pData->name 的值. 这样无疑大大增加了测试困难, 这就属于难于初始化问题. 难于初始化问题常见于测试比较高层的函数时, 很多输入都是间接输入, 本身很复杂, 但被测函数并不直接读写, 只是传递给底层函数以获得一个简单的内部输入, 而且调用的也是实际代码, 打桩难以解决.

此外, 当有局部静态变量时, 它与全局变量一样, 每个用例也可能需要设定不同的初始值(输入), 但外部无法访问, 这也是需要解决的内部输入问题——静态输入. 静态输入没有调用底层函数, 只涉及到局部静态变量, 当然也不能用桩代码来代替. 在嵌入式项目中, 还会遇到中断输入, 即被测函数运行过程中, 在某个位置, 系统调用了一个中断函数, 该函数修改了某个全局变量, 如果被修改的全局变量对被测函数的功能逻辑造成影响, 测试时也必须考虑. 中断输入是在不确定位置, 中断调用不确定的代码形成的, 也不能用桩来代替.

综上所述, 底层函数的不可控、难于初始化, 局部静态输入, 中断输入等函数内部输入问题的解决, 将是单元测试得以顺利进行的关键, 从而为后续测试工作乃至整个软件开发过程的有效进行提供了基础保障.

3 底层模拟技术

底层模拟就是在用例中模拟、控制子函数的行为, 使底层函数产生的数据像参数一样可以在用例中设置,

并且可以对局部数据和中断进行模拟.

底层模拟过程中会在测试工程代码中添加新的代码, 但不会修改被测代码, 实现流程如下:

① 设定并保存底层模拟值. 在测试工程中的底层函数调用之处调用自定义的宏, 将底层模拟的数据写入到 dll 文件中, 写入的内容包括: 底层函数名、参数序号、数据字节数、设定的数据等, 写入的内容在 dll 文件中采用链表存储方式, 方便存取, 同时, 设置该函数的底层模拟标记并保存到 dll 文件.

② 修改测试工程中应用了底层模拟的底层函数. 如果底层函数存在则直接修改, 不存在则修改其对应的桩函数.

③ 运行测试工程. 对底层函数做如下处理:

a. 查询该函数是否运用了底层模拟;

b. 如果未运用底层模拟, 则直接返回, 否则进入下一步 c ;

c. 根据函数名、参数序号从第①步中提及的 dll 文件中读取数据, 若参数序号为 0, 读取设置的返回值, 作为函数的返回值并返回, 若序号大于 0, 则读取 dll 中设置的数据赋值给序号对应的参数.

如有底层函数 GetTemperature()用来获取当前温度, 修改后如下:

```
int GetTemperature(int* pTemperature)
{
VUX_INTERNAL1("GetTemperature",int,_VUXOBJ,1,pTemperature,_VUXP,"int*")
        ……//实现代码未编写
        return 0;
VUX_STUB_END()
// 这个宏定义的是 "}", 以匹配 //VUX_INTERNAL1 宏中的定义
}
```

VUX_INTERNAL1 宏定义实现了前述 a,b,c 三步. 其中_VUXOBJ, _VUXP 用以区分对返回值或数的读取, 其它数依次为: 函数名, 返回值类型, 参数序号, 参数名, 参数类型.

4 底层模拟实验

为实现更好的交互性和易操作性, 实验中采用可视化图形界面设计.

① 解决底层函数不可控、难于初始化问题

如有以下被测函数:

函数是空调控制程序片断, 取得环境温度并计算制冷器需运行的时间; 参数 pWorkTime 是输出参数, 保存制冷器需运行的时间; 如果函数执行失败, 返回 0, 否则返回非 0 值:

```
int WorkTime(int* pWorkTime)
{
        int success = 0;//取环境温度是否成功
int temperature;   //环境温度
success = GetTemperature(&temperature);
//GetTemperature()取得环境温度
        if(!success)   return 0;
         //计算温度差, mExpectTemperature 是
//成员变量, 可以通过用例来设置, 假设为 25
int TempDiff = temperature - mExpectTemperature;
    if(TempDiff <= 0 || pWorkTime == 0)   return 0;
    *pWorkTime = TempDiff * 60;
     //为了简化问题, 这里假设温差一度,
     //需运行一分钟
     return 1;
}
```

其中 GetTemperature()函数有几种可能导致测试困难: 未编写、被隔离、调用实际代码但不符合测试需求. 测试时不需要考虑属于哪种情形, 都可以直接使用底层模拟来解决. 通过底层模拟可以模拟函数的返回值、各个参数值、函数调用次数、成员变量和全局变量. 例如: 可以通过底层模拟 GetTemperature()的返回值为 1(表示取环境温度成功), 如图 1 所示; 模拟取得的环境温度值 temperature(GetTemperature()的第 1 个参数)为 28, 如图 2 所示.

图 1　模拟返回值

图 2 模拟第一个参数值

底层模拟操作完成后，测试工程代码会自动添加如下代码:

模拟返回值时: SF_RETURN_ (1, "GetTemperature"); 表示设置 GetTemperature 函数的返回值为 1; 模拟参数值时: SF_PARAM_(28, "GetTemperature",1); 表示设置 GetTemperature 函数的第 1 个参数为 28.

对每个用例，测试者都可以模拟设置不同的 GetTemperature() 函数返回值及其取得的环境温度 temperature 值，也就避免了运用打桩带来的由于桩函数的返回值一般是固定值而导致函数失真的问题. 如此，对 GetTemperature()进行底层模拟处理后，被测函数 WorkTime()才能够顺利运行得以测试.

对于第 1 节中提到的 AddPerson(PERSON* pData, CPersonMap* map)函数测试中遇到的难于初始化问题，同样可以用上述的底层模拟技术来随意模拟 Search() 的返回值和参数值.

② 解决静态输入问题

如下被测函数:

函数是游戏程序中用于计算打击效果的代码片断，连续打击时效果随次数递减; 参数 reset 为输入参数，为 true 时重置打击次数; 返回打击效果.

```
int PowerEffect(bool reset)
{
    static int times = 0;
    if(reset) times = 0;
    times++;
    int effect[] ={9, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0};
if(times >= sizeof(effect) / sizeof(effect[0]))   return 0;
    return effect[times];
}
```

其中打击次数 times，由于是局部静态变量，用例中无法访问，却需要像全局变量一样，每个用例设定不同的初始值，导致被测函数 PowerEffect()难于测试. 可以通过底层模拟技术模拟局部静态变量 times 的值，从而控制其变化. 如下图 3 所示.

图 3 模拟局部静态变量值

底层模拟操作完成后，测试工程代码会自动添加如下代码:

SF_RETURN_(0, "_lcl_int_times"); 此处同样调用了 SF_RETURN_宏，因为对局部静态变量与返回值的设定原理一样，表示设定 int 局部静态变量 times(_lcl_int_times)的值为 0.

③ 解决中断输入问题

如下被测函数:

函数模拟中断对全局变量的修改; giVar 是 int 类型的全局变量.

```
int interrupt(int arg)
{
    if(giVar > 1)   return 1;
    else    return 0;
}
```

假如我们要测试的情形是: if(giVar > 1)语句前，产生了中断，并修改了 giVar 的值. 则可以在 if(giVar > 1)语句前通过底层模拟来模拟中断产生并修改 giVar 的值，如下图 4 所示.

其中 “34”就是模拟设定的值，在不同的用例中可以随意设定.

底层模拟操作完成后，测试工程代码会自动添加如下代码:

SF_RETURN_(34, "_lcl_int_giVar"); 设置全局变量与设置函数返回值原理一致，故此处仍调用 SF_RETURN_宏，表示设置 int 类型的全局变量 giVar(_lcl_int_giVar)为 34.

图 4 中断模拟

5 结论

通过实验验证，本文提出的底层模拟技术通过在用例中模拟、控制子函数的行为，使底层函数产生的数据可以像参数一样在用例中设定，解决了底层函数不可控、难于初始化等内部输入问题，同时使静态输入、中断输入等运用打桩技术难于实现的问题得以方便解决. 运用底层模拟技术，在用例中设定子函数的输出，使子函数的输出可以与参数等输入放在一起，实现了真正意义上的内部输入；用户不需要考虑调用的是桩代码还是实际代码，都可以使用底层模拟；多次调用同一子函数时，通过底层模拟可以设定不同值；用户不需要额外编写代码，也不需要维护桩代码，底层模拟的数据的维护与参数一致. 通过底层模拟技术，解决了打桩难于实现的多种内部输入问题，且避免了编写桩代码需要的大量时间消耗，大大提高了测试效率.

参考文献

1 林宁,孟庆余.软件测试实用指南.北京:清华大学出版社,2004:4–12,37–42.
2 William E. Perry. 软件测试的有效方法.第 2 版.兰雨晴,高静等译.北京:机械工业出版社,2003.41–88.
3 Kaner C.计算机软件测试.王峰等译.北京:机械工业出版社,2004.1–49.
4 白凯,崔冬华.基于 JUnit 自动化单元测试的研究.计算机与数字工程,2010,(2):52–54.
5 徐宏革,等.白盒测试之道——C++Test.北京:北京航空航天大学出版社,2011:63–69,194–204,234–244.

(上接第 10 页)

确定某系统综合评价为中等风险. 从而完成整个系统的安全评测工作.

表 6 系统综合评价表

系统名称	系统权重	威胁脆弱性关联值	业务系统资产风险值	风险等级
某系统	5	810.00	4000.00	中

通过这项工程，总结出了信息安全服务整合方法论和步骤，以便用到将来的多项信息安全监测工程中.

6 结论

综上所述，建立一个基于模糊综合评价的方法论最大的优点就是，整合等级保护测评、风险评估、信息安全检查的综合评测，可以轻松满足目前三项信息安全检查工作中所遇到的问题，同时通过本文的方法论，可以提供整合、定制化的信息安全技术服务，为用户真正做到贴心、符合标准、满足需求，解决实际问题的信息安全技术服务，真正帮用户解决头疼看头，脚痛医脚的问题，提供一个可定制化整合的信息安全评估服务. 此方法论的不足是，由于此项工作尚在研究阶段，无法得到衡量标准，缺乏案例库的累积. 同时，有些评价指标是否合理，需要此类工程的样本数达到一定数量后，才能总结出相应的衡量指标.

参考文献

1 GB/T 20984-2007.信息安全技术信息安全风险评估规范.
2 GB/T 18336-2001.信息技术安全技术信息技术安全性评估准则.
3 GB/T 22239—2008.信息安全技术信息系统安全等级保护基本要求.
4 李丹,董志国.基于模糊数学的工程项目投标机会选择模型.河南科学,2011,29(12):1499–1501.
5 盛勇,杜晓静,蒋黎明,徐建.服务计算环境中基于模糊修正的信任度量.计算机科学,2011,38(10A):83–86.