一种针对单元测试框架的测试脚本重用方法

祁琳莹，洪　玫，冯丽云，周　宁，文婷婷
（四川大学 计算机（软件）学院，成都 610065）

摘　要：单元测试框架下的软件测试将产生大量的测试脚本，在软件测试过程中如何有效利用现有的测试脚本，实现软件测试脚本（代码）的重用成为业界关心的一个重要问题。业界最常见的复用需求是当开发项目更换新的测试框架时，如何重用开发人员在原单元测试框架下积累的测试脚本。针对这一问题，提出了基于测试脚本移植的重用方案。通过对单元测试脚本的分析和自动翻译方法，将原测试脚本中包含的信息提取出来，解析为基于 XML 的中间脚本，然后再利用 XSLT 技术，依据 XML 记录的信息，自动生成目标框架的单元测试脚本，从而解决单元测试脚本的重用问题。最后实验验证了方案的可行性。

关键词：软件测试；单元测试框架；单元测试脚本重用；源代码翻译
中图分类号：TP311　　**文献标志码**：A　　**文章编号**：1001-3695(2013)06-1764-05
doi:10.3969/j.issn.1001-3695.2013.06.042

Method on reusability of test scripts under unit test framework

QI Lin-ying, HONG Mei, FENG Li-yun, ZHOU Ning, WEN Ting-ting
(*College of Computer(Software), Sichuan University, Chengdu* 610065, *China*)

Abstract: With the wide application of unit testing framework in software industry, some problems have cropped up. One of the principal problems is how can guarantee the test code which developers accumulate not to be wasted, that is how to reuse the test code. This article proposed a solution to solve this problem-reuse the existing unit test code by translating the unit test code automatically. The specific steps was as follows: first, extracted the test case information from the existing unit testing code; then, recorded the information in XML file; finally, generated test code under the rules of new unit testing framework automatically with the technology of XSLT. In the end, this paper verified the feasibility of the method.

Key words: software testing; unit testing framework; unit test script reuse; code translation

0　引言

随着软件行业的不断发展与完善，软件测试也随之兴起。单元测试是对代码的各个单元进行测试，能够尽早地发现软件中存在的 bug，降低修改软件所需的费用，是软件质量保证的基石。作为单元测试的辅助手段，各种用于单元测试的工具应运而生。如 Parasoft 公司针对 C + + 及 Java 分别提供的单元测试工具 C + + Test 和 JTest，能够根据业内编码标准验证代码的功能性，但是这类自动生成测试用例的工具发现软件缺陷能力有限，并且多为商用软件，使用成本较高。相比而言，单元测试框架通常都免费开源易获取，使得单元测试框架在软件行业广受欢迎，用户使用其提供的一系列基类和断言编写单元测试用例，如广为人知的 xUnit 家族。随着单元测试框架的广泛应用，一些问题也随之而来，主要有：

a）更换单元测试框架对原有单元测试代码资源的浪费

一个项目启动后，开发人员会选定一个单元测试框架来进行单元测试，主要包括测试用例的编写、测试的执行、测试结果的收集、输出测试报告等。开发人员依据单元测试的结果，对被测代码进行修改，提高代码质量。如果在项目的中期，要使用新的更优秀的单元测试框架，那么原来已经积累的大量的测试代码以及测试结果等资源将无法重用。

b）新的单元测试框架推广受阻

当企业开发出更加适合自身的新的单元测试框架，并向开发和测试人员推广时，这些框架的使用者会因为不愿意放弃自己原来积累的单元测试代码，而不愿意接受新的单元测试框架。这使得新的单元测试框架的推广工作往往很难进行。

以上两个问题都可以归结为对单元测试代码的重用问题。针对这一问题，目前大多数的解决办法是：a）重新编写原来的测试用例，这种做法低效且不现实；b）仍然使用原来的单元测试框架，这样必然影响单元测试的效率，不利于单元测试框架的良性发展。

如果无法很好地解决单元测试代码的重用问题，会影响单元测试效率的提高，也会影响单元测试框架的进步。本文在对现有的源码翻译方法以及工具研究的基础上，提出了一套针对不同单元测试框架下测试脚本的重用方案，成功地将在 Cpp Unit框架下的单元测试代码翻译成 Google Test 框架的单元测试代码，验证了本方案的可行性，为提高单元测试的效率以及推动单元测试框架的发展作出努力。

收稿日期：2012-10-16；**修回日期**：2012-12-14

作者简介：祁琳莹（1989-），女，四川成都人，硕士研究生，主要研究方向为软件自动化测试（2387797501@qq.com）；洪玫（1963-），女，教授，硕士，主要研究方向为软件工程、软件自动化测试；冯丽云（1988-），女，硕士研究生，主要研究方向为软件自动化测试；周宁（1989-），女，硕士研究生，主要研究方向为软件自动化测试；文婷婷（1988-），女，硕士研究生，主要研究方向为软件自动化测试.

1　相关工作

现有的代码重用方法主要有以下两种:

a)接口封装方法[1]。它是重用遗产系统代码的一个最简单的办法,该方法将遗产系统作为一个整体进行重用。其思想是:在遗产系统代码的外层用一系列新的接口对遗产系统进行封装,好比将遗产系统放在了一个黑盒子中,外部系统通过这些接口来与遗产系统进行通信和交互。最常用的就是采用面向对象的接口对遗产系统进行封装。目前,已经有一些编程语言提供了与其他编程语言进行通信的规范,例如 JNI 就是支持 Java 与其他语言编写的代码进行交互的一个规范。

b)源代码翻译方法[2]。其思想是将源语言编写的代码翻译为目标语言的代码。Martin 等人[3]研究了已有的 C 代码翻译为 Java 代码的策略,提出了一种新的翻译策略,重点介绍了两种将 C 语言中的指针转换到 Java 类型的方法。Martin[4]还介绍了一个将 C 代码翻译为 Java 代码的环境。Mossienko[5]介绍了一个 Cobol 代码到 Java 代码的自动翻译方法,该方法旨在保证生成代码的可维护性。石学林[6]对如何将 Cobol 语言的源代码自动地翻译为 Java 代码进行了研究,详细分析了 Cobol 与 Java 的差异以及翻译过程中存在的关键问题,最终设计并实现了一个将 Cobol 源代码翻译为 Java 源代码的系统,并在文献[7]中介绍了 Cobol 向 Java 翻译时的数据类型转换方法。苏灵燕等人[8]也对 Cobol 向 Java 翻译时的数据类型转换方法进行了研究。武成岗等人[9]研究了将 Cobol 语言代码翻译为 Java 代码时如何消除 PERFORM 和 GOTO 语句,提出了一种利用 switch、while 语句来替代上述两者进行程序结构控制,完成程序的等价变换。

2　单元测试脚本重用方案设计

2.1　单元测试脚本的可重用内容分析

单元测试脚本的组成包括测试用例代码、测试执行控制代码两个部分。单元测试代码的核心内容在测试用例部分,各个框架之间的测试用例代码的构成具有较大的相似性,能够实现对这部分代码进行自动或半自动翻译;测试方法包含了测试数据、预期结果,以及对测试结果的验证等信息,是测试用例代码最重要的部分;在测试用例代码部分,建立测试夹具、注册测试以及对测试用例的组织等内容与框架的相关性较大;由于不同单元测试框架间的封装程度不同,使得各框架间测试执行控制部分的代码差异性较大,因此测试控制代码的自动翻译困难较大;一个测试工程一般只需要一段测试执行控制代码,即使是手动重写,也只需要投入很少的人工和成本。

综上所述:单元测试代码中的主要信息包含在单元测试用例代码部分,对单元测试代码的复用,主要是为了复用单元测试用例代码部分包含的信息。因此,在对单元测试代码进行翻译时,主要是对测试用例部分代码作翻译。

2.2　基于代码翻译的单元测试脚本重用

单元测试代码翻译方案借鉴了源代码移植方法中的代码翻译思想,通过将原单元测试框架下的测试脚本翻译为目标框架下的测试脚本,达到对原框架下测试脚本的重用目标。

翻译评价标准不同,设计的翻译方法和策略也将不同。在对单元测试用例代码作翻译时,如果采用一对一的翻译,那么两框架之间就需要作两次相互翻译。当有 n 个单元测试框架时,就需要对 n 个单元测试框架作两两翻译,共需要 $n\times(n-1)$ 次翻译。当新添加一个单元测试框架时,需要为这个新框架与原有的 n 个框架相互建立翻译方案,共 $2n$ 次。当 $n=5$ 时,$n\times(n-1)=20$,$2n=10$。随着 n 的增加则增加到很大的工作量,比如当 $n=10$ 时,$n\times(n-1)=90$,$2n=20$。因此,这种一对一的翻译方案是不可取的。

本文参考了石学林[6]提出的五个源代码翻译的评价标准,首要考虑代码的功能等价性标准和代码翻译的自动化程度标准,提出了一种基于中间描述的抽象再实现的翻译方案,减少了翻译的工作量,如图 1 所示。

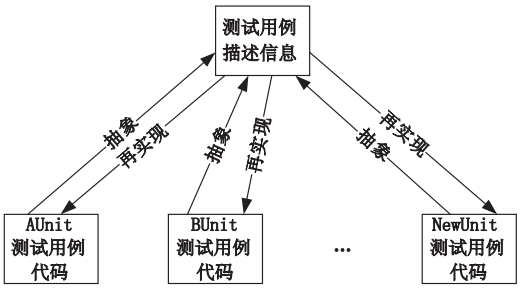

图1　基于抽象再实现的测试用例代码翻译方案

首先,将每个单元测试框架编写的单元测试用例代码中所包含的测试用例信息抽取出来,用一个通用的中间形式来描述。然后,为每个框架建立一个从测试用例描述信息到对应的单元测试用例代码的自动生成过程。当需要将 AUnit 单元测试框架的单元测试代码翻译成 BUnit 单元测试框架的测试代码时,只需要将 AUnit 的单元测试代码抽象为测试用例描述代码,再根据测试用例描述代码生成 BUnit 的单元测试代码即可。当有新的单元测试框架 NewUnit 添加进来时,只需要建立 NewUnit 的抽象过程,以及根据测试用例描述代码自动生成 NewUnit 的单元测试代码的过程即可。两个框架的单元测试代码的翻译过程如图 2 所示。

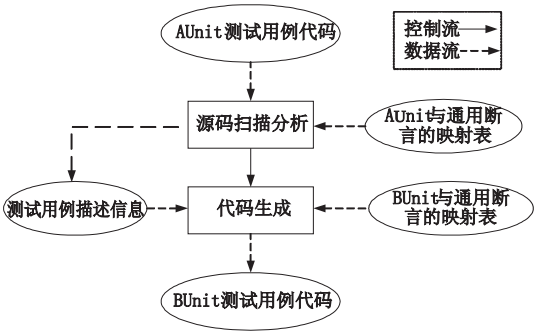

图2　单元测试用例代码翻译示意图

3　单元测试脚本重用方案实现

3.1　基于 XML 的单元测试用例信息描述

本文采用 XML 作为中间形式,描述单元测试用例信息。单元测试代码中一个较为完整的信息单元是测试套(test suite),因此以测试套为单元描述单元测试用例代码中的信息。定义了以下模式的 XML 来记录一个单元测试套的信息:

〈file〉标签:是根标签,记录该测试套所在的文件名,通常.h 和.c 或.cpp的名字一样,在此只记录文件名,不记录文件的后缀名。

〈include〉标签:记录文件中用到的头文件,只需记录测试用例代码中用于包含被测函数或被测类所在的头文件,以及其他与单元测试框架不相关的头文件。由于这些头文件与单元测试框架不相关,在翻译时不需要作改动。测试代码中用到的

单元测试框架的头文件,则不需要标记。

〈testsuite〉标签:记录测试套的名字。

〈declaration〉标签:记录测试用例中的一些声明语句。

〈setup〉标签:记录测试环境初始化方法的信息,其中包括〈fucname〉标签,记录初始化方法的函数名;〈code〉标签记录初始化方法的代码。

〈teardown〉标签:记录测试环境清理方法的信息,其中包括〈fucname〉标签,记录清理方法的函数名;〈code〉标签记录清理方法的代码。

〈testcase〉标签:记录测试用例的信息,其中包含〈tc_name〉标签,记录测试用例,也是测试方法或函数的名字;〈common_code〉标签,记录测试用例代码中的通用代码;〈assertion〉标签,记录测试用例代码中的断言,其中包括〈gen_name〉标签,记录该断言的通用名,〈para id = ""〉标签记录断言的参数,该标签包含属性 id,用于记录参数的顺序。

〈general_fuc〉标签用于记录测试代码中除测试方法以外的其他方法,这些方法通常与测试框架无关,其中包括〈name〉标签,记录该方法的函数名;〈code〉标签记录该方法的代码。

3.2 单元测试断言通用命名表设计

各个单元测试框架提供的断言的命名规则各不相同,即使是相同功能的断言,名字也不一样。因此,在单元测试代码翻译时,非常重要的工作就是对断言语句进行翻译——将使用一个框架的断言语句翻译成另一个框架的断言语句。为了完成不同的单元测试框架之间的断言的翻译,需要定义一个统一的命名规则,作为翻译过程的一个中间描述。表 1 为单元测试框架断言通用命名表的结构。

表 1 断言通用命名表

通用断言名(GEN_ASSERTION_NAME)	参数列表	备注
GEN_ASSERT_EQUAL	PARAMETER1 = Expected PARAMETER2 = Actual PARAMETER3 = …	用于判断两个值相等;若不相等,断言失败
GEN_ASSERT_FAIL	NONE	用于添加一个一定会失败的断言
…	…	…

最初建立断言通用命名表时,需要对现有的单元测试框架提供的所有断言进行分析对比,将功能相同或近似的断言分成一类,为每类功能相同的断言重新定义一个通用的名字,在断言通用命名表中记录该断言的公共参数列表,在备注中描述该断言的功能。对于某个框架独有的断言,将其单独列为一类并定义通用断言名,并记录其参数列表及功能描述。因此,断言通用命名表是各个框架的断言的并集。

断言通用命名表在使用的过程中,需要不断地维护与更新。当出现了新的单元测试框架,就需要检查当前的断言通用命名表是否能够覆盖新的单元测试框架提供的断言。如果新的单元测试框中有不能被断言通用命名表覆盖的断言,就需要将其作为一类新的断言,添加到断言通用命名表中,并为其定义通用断言名,记录其参数列表及功能描述。

3.3 框架断言与通用断言的映射

在建立好通用断言命名表后,还需要为每个单元测试框架建立一张自身断言与通用断言的映射表,为不同的框架断言建立联系,将某个框架的断言翻译为其他框架的断言,如表 2 所示。

表 2 单元测试框架断言与通用断言映射表

通用断言	参数列表	CppUnit 框架断言	参数列表
GEN_ASSERT_EQUAL	Expected,Actual	CPPUNIT _ ASSERT_EQUAL	Expected,Actual
GEN_ASSERT_EQUAL	Expected,Actual	CPPUNIT _ ASSERT_EQUAL_MESSAGE	Message, Expected, Actual
GEN_ASSERT_XXX	Para1,…	N/A	N/A
…	…	…	…

单元测试框架断言与通用断言的映射表的映射关系有以下三种:

a)完全映射。当具体单元测试框架的某个断言与某通用断言完成相同的功能,且参数列表也相同,那么该断言与该通用断言完全映射。表 2 通用断言 GEN_ASSERT _EQUAL 用于判定 Expected 与 Actual 相等,在 CppUnit 中,CPPUNIT_ASSERT_EQUAL 与之完成相同的功能,且两者的参数列表相同。

b)近似映射。当具体单元测试框架的某个断言无法找到与之完全映射的通用断言,但是存在与之功能相似的通用断言,称该断言与该通用断言为近似映射关系。如表 2 所示,CppUnit中的断言 CPPUNIT_ASSERT_EQUAL_MESSAGE 用于判定 Expected 与 Actual 相等,当两者不等时显示用户提供的信息。该断言与通用断言 GEN_ASSERT_EQUAL 功能类似,只是 GEN_ASSERT_EQUAL 没有让用户提供失败信息的功能。CPPUNIT_ASSERT_EQUAL_MESSAGE 与通用断言 GEN_ASSERT_EQUAL 近似映射。

c)无法映射。由于通用断言列表是所有单元测试框架断言的并集,那么某些通用断言在具体的单元测试框架断言中找不到与之有完全映射或近似映射的关系,则称这些通用断言与具体单元测试框架的断言为无法映射关系。如表 2 所示,通用断言 GEN_ASSERT_XXX 在 CppUnit 中没有与之有完全映射或近似映射关系的断言时,将 CppUnit 框架的对应项标记为 N/A,表示在 CppUnit 中没有与该通用断言对应的断言。

由于单元测试断言通用命名表会随着新的单元测试框架的出现而不断更新,同时,各个单元测试框架自身也在不断地升级,因此各个单元测试框架与通用断言的映射表也需要随着这些变动进行更新。

3.4 基于 XSLT 的测试用例代码生成

测试用例代码生成阶段完成以下工作:将测试用例描述语言描述的测试用例信息,依照某个具体的单元测试框架提供的测试代码编写规则或模板,自动地生成该框架的测试用例代码。由于采用了 XML 来标记测试用例代码中的信息,而对 XML 中的信息进行处理的利器即 XSLT[10]。采用 XSLT 技术的代码生成器读取对应具体业务模型的 XSLT 程序模板文件,读取描述具体业务模型信息 XML 文档,通过 XSLT 处理器,生成所需要的源代码[11]。

生成代码时,需要注意对断言的翻译。由于根据 XML 文件中记录的通用断言以及具体框架与通用断言的映射表来生成目标框架的断言语句,在生成具体框架的断言时,对于能够完全映射和近似映射的断言,能够生成相应具体框架的断言,对于无法映射的通用断言,则无法生成相应的断言。

4 实验验证

为了验证代码翻译方案的可行性,设计了以下实验:将

CppUnit 编写的单元测试代码,自动翻译为 GTest 的单元测试代码。

4.1　实验准备

实验采用的硬件环境为普通 PC 机,采用 Windows 7 操作系统,基础环境为 JDK 1.5,Web 应用服务器为 Tomcat 5.5,XSLT 处理器为 Apache Xalan 2.7。准备多段用 CppUnit 编写的单元测试代码。如图 3 所示为用 CppUnit 编写的单元测试脚本 ExampleTestCase.h 文件,图 4 为与之对应的源代码文件 ExampleTestCase.cpp。

```
#include〈cppunit/extensions/HelperMacros.h〉
class ExampleTestCase: public CPPUNIT_NS::TestFixture
{
  CPPUNIT_TEST_SUTTE(ExamleTestCase);
  CPPUNIT_TEST(testAdd);
  CPPUNIT_TEST_SUITE_END();
protected:
  double m_value1;
  double m_value2;
public:
  void setUp();
  ExampleTestCase();
  ~ExampleTestCase();
protected:
  void testAdd();
};
```

图 3　CppUnit 下的测试脚本 ExampleTestCase.h 文件

```
#include〈cppunit/config/SourcePrefix.h〉
#include "ExampleTestCase.h"
CPPUNIT_TEST_SUITE_REGISTRATION(ExampleTestCase);
ExampleTestCase()
{
}
ExampleTestCase()
{
}
void ExampleTestCase::setUp()
{
  m_value1 =2.0;
  m_value2 =3.0;
}
void ExampleTestCase::testAdd()
{
vdouble result = m_value1 + m_value2;
  CPPUNIT_ASSERT(result = =6.0);
}
```

图 4　CppUnit 下的测试脚本 ExampleTestCase.cpp 文件

4.2　实验步骤

a)根据 CppUnit、GTest 现有的断言,分别建立通用断言命名表、CppUnit 与通用断言的映射表、GTest 与通用断言的映射表。

b)将实验测试代码中包含的信息用例抽取出来,并将其中用到的断言语句转换成通用断言,用 4.1 节中定义的 XML 格式来表示。

c)为 GTest 建立根据 XML 描述的测试用例信息自动生成代码的 XSL 模板。

d)用 XSLT 处理器 Xalan,依据 b)中的 XML 文件和 c)中的 XSL 模板,自动生成 GTest 的测试脚本。

4.3　实验结果及分析

根据 4.2 节的实验步骤可以将多段 CppUnit 单元测试代码正确翻译为 GTest 单元测试代码,不会损坏或丢失原测试代码中的信息。

1)通用断言命名表,如表 3 所示;CppUnit 与通用断言的映射表,如表 4 所示;GTest 与通用断言的映射表,如表 5 所示。因为篇幅有限,只列出了上述脚本中用到的断言。

表 3　通用断言命名表

通用断言名(GEN_ASSERTION_NAME)	参数列表	备注
GEN_ASSERT_EQUAL	PARAMETER1 = Expected PARAMETER2 = Actual PARAMETER3 = …	用于判断两个值相等;若不相等,断言失败
…	…	…

表 4　CppUnit 与通用断言的映射表

通用断言	参数列表	CppUnit 框架断言	参数列表
GEN_ASSERT_EQUAL	Expected,Actual	CppUNIT _ ASSERT_EQUAL	Expected,Actual
…	…	…	…

表 5　GTest 与通用断言的映射表

通用断言	参数列表	CppUnit 框架断言	参数列表
GEN_ASSERT_EQUAL	Expected,Actual	ASSERT_EQUAL	Expected,Actual
…	…	…	…

2)测试用例信息文件 ExampleTestCase.xml,如图 5 所示。

```
〈? xml version = "1.0" encoding = "ISO - 8859 - 1" ?〉
〈file name = "ExampleTestCase" 〉
    〈include/〉
    〈testsuite〉ExampleTestCase〈/testsuite〉
    〈declaration〉protected:double m_value1;double m_value2;public:ExampleTestCase();~ExampleTestCase();protected:void testAdd();
    〈/declaration〉
    〈setup〉
      〈fucaname〉setUp〈/fucname〉
      〈code〉m_value1 =2.0;m_value2 =3.0;〈/code〉
    〈/setup〉
    〈teardown〉
      〈fucname/〉
      〈code〉〈/code〉
    〈/teardown〉
    〈testcase〉
      〈name〉testAdd〈/name〉
      〈common_code〉double result = m_value1 + m_value2;〈/common_code〉
        〈assertion〉
        〈name〉GEN_ASSERT_EQUAL〈/name〉
        〈para id = "1" 〉result〈/para〉
        〈para id = "2" 〉6.0〈/para〉
      〈/assertion〉
    〈/testcase〉
    〈general_fuc〉
      〈name〉ExampleTestCase〈/name〉
      〈code/〉
    〈/general_fuc〉
    〈general_fuc〉
        〈name〉~ExampleTestCase〈/name〉
        〈code/〉
    〈/general_fuc〉
〈/file〉
```

图 5　测试用例信息文件 ExampleTestCase.xml

3)自动生成.h 脚本文件的 XSLT 模板 h.xsl,如图 6 所示;自动生成.cpp 脚本文件的 XSLT 模板 cpp.xs,如图 7 所示。

```
〈? xml version = "1.0" encoding = "ISO - 8859 - 1" ?〉
〈xsl:stylesheet version = "1.0"
  xmlns:xsl = "http://www.w3.org/1999/XSL/Transform" 〉
  〈xsl:output method = "text" version = "1.0" indent = "yes"/〉
  〈xsl:template match = "/" 〉
    #include〈gtest.h〉
  〈xsl:for - each select = "file/include" 〉
    〈xsl:copy of select = "current()" /〉
  〈/xsl:for each〉
  class
  〈xsl:copy of select = "file/testsuite" /〉
  :public testing::Test{public:void SetUpTestCase();
  void TearDownTestCase();
  〈xsl:copy of select = "file/declaration" /〉
  }
  〈/xsl:template〉
〈/xsl:stylesheet〉
```

图 6　自动生成.h 脚本文件的 XSLT 模板

```
〈? xml version = "1.0" encoding = "ISO - 8859 - 1"?〉
〈xsl:stylesheet varsion = "1.0" xmlns xsf = "http://www.w3.org/1999/XSL/Transtorm"〉
  〈xsl:output method = "text" version = "1.0" indent = "yes"/〉
  〈xsl:template match = "/"〉
#include〈
〈xsl:copy of select = "/testsuite"/〉
.h〉
〈xsl; for - each select = "file/generat_fuc"〉
  〈xsl; copy of select = "/file/testsuite"/〉
  ::
  〈xsl; copy of select = "current()/name"〉
  (
  〈xsl; copy of select = "current()/code"/〉
  )
〈/xsl:for each〉
void SetUpTestCase(){
〈xsl:copy of select = "file/setup/code"/〉
}void TearDownTestCase(){
〈xsl:copy of select = "file/teardown/code"/〉
}
〈xsl:for each select = "file/testcase"〉
TEST_F(
〈xsl:copy of select = "/file/testsuite"/〉
〈xsl:copy of select = "current()/name"/〉
){
〈xsl:copy of select = "current()/common_code"/〉
〈xsl:for each select = "./assertion"〉
    〈xsl:if test = "/name = GEN_ASSERT_EQUAL"〉
      ASSERT_EQUAL(
      〈xsl:copy of select = "./para[@id = 1]"/〉
      〈xsl:copy of select = "./para[@id = 2]"/〉
      );
  〈/xsl:if〉
      〈/xsl:for each〉
}
      〈/xsl:for each〉
    〈/xsl:template〉
〈/xsl:stylesheet〉
```

图 7 测试用例信息文件 ExampleTestCase. xml

4)用 XSLT 处理器 Xalan,根据 h. xsl 和 cpp. xsl 中的转换规则,以及 ExampleTestCase. xml 中的测试用例信息,自动地生成 GTest 的测试脚本。

执行命令 java org. apache. xalan. xslt. Process -IN ExampleTestCase. xml -XSL h. xsl -OUT ExampleTestCase. h -INDENT -TEXT生成 ExampleTestCase. h 文件,如图 8 所示。执行命令:java org. apache. xalan. xslt. Process -IN ExampleTestCase. xml -XSL h. xsl -OUT ExampleTestCase. h -INDENT - TEXT 生成 ExampleTestCase. cpp 文件,如图 9 所示。

```
#include〈gtest.h〉
class ExampleTestCase:public testing::Test
{
public:
  void SetUpTestCase();
  void TearDownTestCase();
protected:
  double m_value1;
  double m_value2;
public:
  ExampleTestCase();
  ~ExampleTestCase();
protected:
  void testAdd();
}
```

图 8 自动生成的 GTest 下的 ExampleTestCase. h 文件

可以看出,通过将原有单元测试代码中的信息,提取到与具体单元测试框架无关的 XML 文件中,再利用 XSLT 转换技术,能够自动地生成某个具体单元测试框架的测试代码。实验结果说明本文所提出的对单元测试代码自动翻译的方法,能够有效地对已有的单元测试代码进行复用。

5 结束语

本文提出了一种可行的对单元测试代码进行自动化翻译的复用方案,该方案能自动按照原单元测试框架规则编写的单元测试代码,自动地翻译为按照目标单元测试框架的规则编写的单元测试代码。该方案把单元测试代码中的信息用自描述性的 XML 来记录,将不同框架编写的测试用例代码中的公用信息提取出来,使用与具体框架无关的格式来记录,再利用 XSLT 技术,最终将 XML 中记录的测试用例信息自动地生成具体单元测试框架的单元测试代码。实验表明,本文所提出的单元测试代码翻译方案能够比较有效地将原单元测试框架的单元测试代码自动地翻译为目标单元测试框架的单元测试代码。

```
#include〈ExampleTestCase.h〉
ExampleTestCase::ExampleTestCase
{
}
ExampleTestCase::~ExampleTestCase
{
}
void SetUpTestCase()
{
m_value1 = 2.0;
m_value2 = 3.0;
}
void TearDownTestCase()
{
}
TEST_F(ExampleTestCase,testAdd)
{
double result = m_value1 + m_value2;
ASSERT_EQUAL(result,6.0);
}
```

图 9 自动生成的 GTest 下的 ExampleTestCase. cpp 文件

参考文献:

[1] 陈明. 软件测试[M]. 北京:机械工业出版社,2011:13-56.

[2] 杨卫平,赵合计. 遗产软件的代码翻译[J]. 计算机工程,2004,30(6):83-84.

[3] MARTIN J, MULLER H A. Strategies for migration from C to Java [C]//Proc of the 5th European Conference on Software Maintenance and Reengineering. Washington DC: IEEE Computer Society,2001.

[4] MARTIN J. Ephedra: A C to Java migration environment: approaches, case studies and tools for migrating legacy systems from C and C + + to Java[M]. Germany: Lambert Academic Publishing,2009.

[5] MOSSIENKO M. Automated Cobol to Java recycling[C]//Proc of the 7th European Conference on Software Maintenance and Reengineering. Washington DC: IEEE Computer Society,2003:40.

[6] 石学林. Cobol2Java 源代码翻译关键技术研究[D]. 上海:中国科学院计算技术研究所,2005.

[7] 石学林,张兆庆,武成岗. Cobol 到 Java 翻译中的数据类型转换方法[J]. 计算机研究与发展,2006,43(2):336-342.

[8] 苏灵燕,武成岗,唐生林,等. Cobol 到 Java 源代码翻译中的数据类型转换[J]. 计算机应用研究,2008,25(3):771-774.

[9] 武成岗,张兆庆,乔如良,等. 代码翻译中 PERFORM 和 GOTO 语句复合结构的变换[J]. 软件学报,2004,15(4):475-486.

[10] 张朝阳. XML 开发典型应用:数据标记/处理、共享与分析[M]. 北京:电子工业出版社,2008:139-341.

[11] 陈翔,王学斌,吴泉源. 代码生成技术在 MDA 中的实现[J]. 计算机应用研究,2006,23(1):147-150.

[12] 刘铭,徐兰芳,骆婷. 编译原理[M]. 北京:电子工业出版社,2011:35-78.

[13] 王震江,马宏. XML 基础与实践教程[M]. 北京:清华大学出版社,2011:13-47.