TTCN-3 编译器测试用例自动扩展生成①

Automatic Extension and Generation of Test Cases for TTCN-3 Compiler

吴文娟[1] 蒋 凡[1] 任 峰[2] 金 鑫[1]

(1.中国科学技术大学 计算机科学技术系 安徽 合肥 230027; 2.鼎桥通信技术有限公司 上海 201206)

摘 要: 针对目前编译器采用手工测试方式存在出错率高、测试覆盖难以度量等问题，以 TTCN-3 语言编译器为例，在对 TTCN-3 核心语言进行等价类划分的基础上，手工编写黑盒测试用例作为输入，收集并求解被测 TTCN-3 编译器源码中的每个条件，据此修改由参考编译器生成的对应语法树相关结点，扩展生成新的测试用例。实验表明：根据对被测编译器源码中的条件收集而自动扩展生成的测试集，不仅可以提高测试集的正确性，还可以保证对被测编译器实现的条件覆盖。

关键词: TTCN-3 编译器测试 自动化测试 约束求解 条件覆盖 语法树

1 引言

TTCN-3(Testing and Test Control Notation version3)是欧洲电信标准协会 ETSI 于 2001 年推出的新一代协议与软件测试标准。作为一种全新的测试描述语言，其应用领域越来越广泛。使用该语言需要一个 TTCN-3 编译器。编译器作为一种系统级软件，对可靠性和稳定性要求比一般的应用软件高很多。传统的软件测试用于编译器这种可靠性要求较高的软件存在诸多不利因素，如：测试用例设计开发周期长、手工生成的测试用例错误较多、测试覆盖度量难以判定等。

本文提出一种针对TTCN-3编译器测试用例自动扩展生成的解决方案，该方法也适用于各种编译器测试用例的自动扩展生成。

2 问题描述

目前，有关编译器自动化测试的研究成果和技术日趋成熟，但基本上集中在对常用高级语言 C/C++、java 等编译器的研究领域[1,2]。而且大部分研究成果从特定语言的特性出发，提出的测试自动化解决方案都有一定的局限性。

其中，文献[1]针对 C 编译器的自动测试设计思想是：将 C 语言元素大致分类为 9 种数据类型、35 种运算符、8 种流程控制语句和函数，通过概率算法指定生成测试用例过程中各语言元素出现的概率来生成最终测试用例[2]。针对 java 编译器提出的自动化测试首先生成测试用例框架，继而生成控制流、最后补全其余字节码。

由于TTCN-3语言文法定义较之上述几种高级语言规模庞大，语言元素更为丰富，上述几种方法均不太适用。本文提出了一种新的测试用例自动生成方法：“TTCN-3 Compiler Automatic Testing”(以下简称 TCAT)。TCAT 的基本思路是：首先对 TTCN-3 核心语言及 BNF 范式进行等价类划分，由测试人员为每一个等价类手工编写一个黑盒测试用例(假定该测试用例语法语义均正确)，作为被测编译器输入，收集被测编译器源码中的条件约束并进行约束求解[3]；然后将每个黑盒测试用例作为参考编译器输入，进而生成一棵语法树，最后根据求解出的条件结果修改这棵语法树的相关结点，据此扩展生成新的满足被测编译器源码条件覆盖的测试用例。

3 TTCN-3编译器自动测试框架设计

TCAT 自动测试框架分为三部分：

(1) 手工编写的黑盒测试用例输入；

(2) 收集并求解被测编译器实现判定条件；

① 收稿时间:2008-10-29

(3) 参考编译器扫描测试用例生成语法树，修改语法树结点，扩展生成新的测试用例。

整个测试框架可以用图 1 表示：

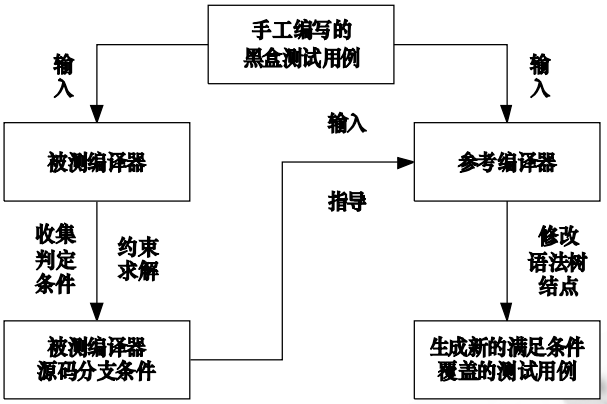

图 1　TTCN-3 编译器测试用例自动扩展框架

在该框架下，首先由手工编写的黑盒测试用例作为被测编译器输入，收集被测编译器实现的条件并进行约束求解；然后由参考编译器扫描上述黑盒测试用例，进而生成语法树；在此基础上以约束求解得到的被测编译器源码中的条件为依据修改相关语法树结点，扩展生成新的满足条件覆盖的测试用例。各部分设计描述如下：

3.1 手工编写的黑盒测试用例输入

在测试用例自动扩展生成时，作为输入的黑盒测试用例可由两种方式得到：文法测试生成算法自动生成；手工编写实现。

其中，文法测试以 Purdom 在 1972 年提出的“规则覆盖”算法为代表，该算法保证对上下文无关文法测试的 100%覆盖，但却没有考虑对文法内部结构的测试覆盖[4]。在后续研究人员的努力下，该算法得到了较大的改进和发展，比较成熟的有称为“上下文依赖规则覆盖算法”[5]，它通过对文法内部结构的收集，达到 100%的分支覆盖，但它仍无法保证生成的测试用例的语义正确性。

目前已有研究人员正针对文法测试的测试用例语义正确性进行研究，但还没有成熟的研究成果出现。因此，本文暂选取由测试人员通过等价类划分方式手工编写的黑盒测试用例作为测试用例自动扩展生成框架的输入，同时假设这些黑盒测试用例是语法语义正确的。

3.2 收集并求解被测编译器实现判定条件

为获取被测编译器源码实现信息，以指导自动扩展生成新的满足条件覆盖的测试用例集。本文采用手工编写的黑盒测试用例作为输入，收集与该测试用例相关的被测编译器实现的各个条件，并使用约束求解得到判定的其他条件，为自动生成新的测试用例提供依据。

收集条件并进行约束求解的基本思想可以用下面的例子说明：

```
1. bool PlusExprPostScan(COB* pLValue, COB* pRValue)
2. {
3.   ...;
4.   if(((TTC::INTEGER_TYPE = =eCode1) &&
5.     (TTC::INTEGER_TYPE = =eCode2)) ||
6.     ((TTC::FLOAT_TYPE= =eCode1) &&
7.     (TTC::FLOAT_TYPE = =eCode2)))
8.   {
9.       m_pType = pLValue->GetRealType();
10.  }
11.  else
12.  {
13.     return false;
14.  }
15.  return true;
16. }
```

图 2　被测 TTCN-3 编译器加法实现源码片断

图 2 是从被测 TTCN-3 编译器中截取的一段加法操作实现代码。若要覆盖其所有条件，需要 3 类参数：

- pLValue = Integer && pRValue = Integer；
- pLValue = Float && pRValue = Float；
- 其它。

若给定一个包含两个整型值相加(如：1+1)的测试用例，则满足上述第一类条件。被测编译器对该用例进行静态语义检查时将执行图 2 中的行 9 代码，此时需要收集 if 语句的每个条件(||表示或运算)：

$$\begin{cases} TTC::INTEGER\_TYPE = eCode1 \;\&\; \& \\ TTC::INTEGER\_TYPE = eCode2 \\ \| \\ TTC::FLOAT\_TYPE = eCode1 \;\&\; \& \\ TTC::FLOAT\_TYPE = eCode2 \end{cases} \quad (1)$$

且函数 PlusExprPostScan 将返回 true(行 15)。

为了得到函数 PlusExprPostScan 的所有条件，本文利用约束求解器对收集到的所有条件进行约束求解。

针对图 2 中的 PlusExprPostScan 函数，由于“TTC”是一个预先定义的名字空间，而 TTC::INTEGER_TYPE 和 TTC::FLOAT_TYPE 均是该命名空间中 TTCN_ELEMENT 的枚举元素。因此，对 else 分支的条件可以通过求解枚举集合 TTCN_ELEMEN 与

TTC::INTEGER_TYPE、TTC::FLOAT_TYPE 构成的集合的差集得到。随机从该差集中选一个元素作为函数 PlusExprPostScan 的输入，即可满足 else 分支条件。

3.3 扩展生成新的测试用例

为了得到待扩展黑盒测试用例的语法树并方便修改语法树结点，本文引入参考编译器的概念：即在测试用例的自动扩展生成中，采用一个功能和性能均已大量实践验证，实现基本是正确的参考编译器，生成黑盒测试用例的语法树，依据被测编译器的条件输入修改语法树结点，得到一棵新的语法树，最后将该语法树经过翻译，扩展生成新的满足被测编译器实现的条件覆盖的测试集。

以下是一棵对应于 TTCN-3 测试用例的语法树(图 3)：

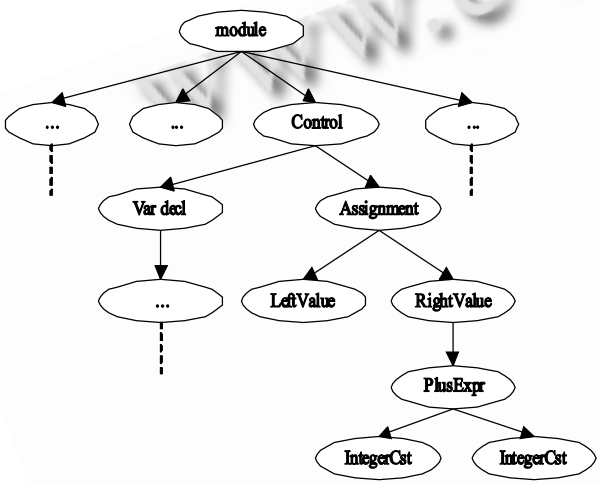

图 3　TTCN-3 测试用例语法树

假设根据收集到的被测编译器实现的条件，需要修改的结点为图中灰色标识的语法树结点(Integercst)，则为了保证修改后的语法树的上下文一致性，必须采用某种机制保证与被修改语法树结点相关的上下文信息也得到修改，图 3 使用浅灰色标识与被修改结点相关的语法树结点。 以下对此进行阐述。为了确定这些上下文相关语法树结点，以便在修改语法树结点的同时保证其上下文相关语法树结点也得到相应修改，其步骤如下：(1)保存语法树结点间的引用关系。即：对于每个语法树结点，在进行词法语法分析和静态语义检查后，将所有引用该语法树结点的结点均保存在一个引用表中，这样，当修改某个语法树结点的同时，通过查找该引用表即可将相关语法树结点也修改掉；(2)采用递归调用方式从 TTCN-3 测试用例的最顶层语法元素 module(见图 3)开始，逐层遍历各语法树结点，直至遇到由收集求解得到的条件决定需要修改的结点时，对其进行具体的修改。通过这两步保证了修改后的语法树的上下文一致性。

从语法树到 C++的翻译框架由本实验室已有成果提供支持，从语法树到 TTCN-3 语言的翻译主要借鉴已有成果的翻译思想和框架实现。

4　实例分析

以一个 TTCN-3 黑盒测试用例作为被测编译器输入，对本文提出的测试用例自动扩展生成方法进行详细说明。

```
1.  module Test_PlusExpr
2.  {
3.      control
4.      {
5.          var integer i;
6.          i= 1+2;
7.          log(i);
8.      }
9.  }
```

图 4　TTCN-3 加法操作测试用例

图 4 是一个测试 TTCN-3 编译器加法操作的测试用例。

通过约束求解得出被测编译器加法操作实现的每个条件：整数和整数相加、浮点数和浮点数相加、其他类型元素相加(参见 3.2 节)。因此，若要覆盖被测编译器有关加法操作实现的全部条件结果，需要对图 4 测试用例中的加法操作数进行修改。

对应的语法树节点即为上述测试用例中行 6 右边加法表达式的两个操作数 1 和 2。同时，为了保证修改后的语法树上下文一致性，行 5 的变量 i 的类型也要进行修改。

图 5 给出了该测试用例对应的标识符哈希表和主要语法树结点及其相互关系：

由上述分析知，图 5 中的整型常量 1 和 2 两个语法树结点需要修改。同时，由于 1+2 的结果赋值给了变量 i，因此，在语法树的生成过程中由赋值语句保存了其左值变量 i 引用右值“1+2”表达式的信息，故在修改完上述整型常量结点(如替换成浮点型常量结

点)，通过查找引用 1+2 表达式的结点信息即可确定变量 i 的类型也需要修改，(即 i 结点的 type 域替换成与表达式相同的类型)从而保证了修改后的语法树的上下文一致性。

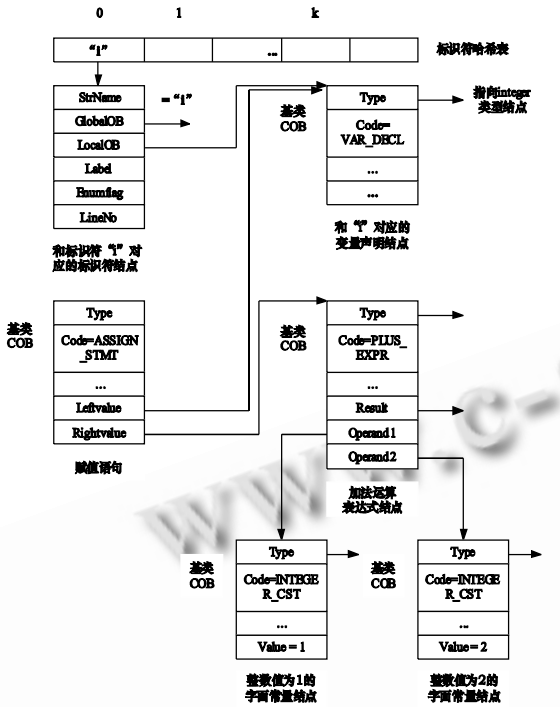

图 5　标识符哈希表与语法树结点

经过实验，扩展生成的新测试用例为 2 个，如图 6 所示：

```
module TestPlusExpr1
{
    浮点数和浮点数相加
    control
    {
        var float;i
        i= 2.0+2.0;
        log(i);
    }
}
```

```
module TestPlusExpr2
{
    布尔类型和布尔类型值相加
    control
    {
        var boolean;i
        i= true+true;
        log(i);
    }
}
```

图 6　扩展生成的新测试用例

实验表明：扩展生成的新测试用例(图 6)与原有测试用例(图 4)结合起来，即可保证对被测 TTCN-3 编译器加法操作实现的条件覆盖。

5　结束语

本文给出了一个采用黑盒和白盒测试相结合自动扩展生成 TTCN-3 编译器测试用例的方法 TCAT。目前，TCAT 已经通过实验用于 TTCN-3 测试工具 TTTDS2 的自动测试生成。实验表明 TCAT 能够生成满足收集到的被测编译器实现的条件覆盖的测试集，并可以发现判定的条件内部实现上的缺陷。通过收集被测编译器源码中的不同类型信息(路径、分支等)，也可以达到对被测编译器实现的其他测试覆盖。

由于 TCAT 需要借助参考编译器生成的语法树扩展生成新的测试用例，因此，参考编译器应是基于翻译架构实现的，对于以解释执行实现的编译器则不能作为参考编译器使用。此外，实验过程中对于被测编译器源码中条件的收集目前仍需要人工参与，今后可以采用自动化方法完成。

参考文献

1 耿言,陈英,史晋.C 编译器自动测试仿真研究.计算机仿真, 2003,20(10):139－142.

2 Yoshikawa T, Shimura K, Ozawa T. Random Program Generator for Java JIT Compiler Test System. Proceedings of the Third International Conference on Quality Software, 2003:20－23.

3 Cadar C, Ganesh V, Pawlowski PM., et al. EXE: Automatically Generating Inputs of Death. Proceedings of the 13th ACM Conference on Computer and communications security, 2006:322－335.

4 Purdom P. A sentence generator for testing parsers. BIT Numerical Mathematics, 1972,12(3):366－375.

5 沈扬,陈海明.基于上下文依赖规则覆盖的句子生成.计算机工程与应用, 2005,41(17):96－100.