TTCN-3 类型系统测试用例集自动生成①

Test Suite Generation for TTCN-3 Type System

蒋　凡　金　鑫　吴文娟　(中国科学技术大学 计算机科学与技术系 安徽 合肥 230027)

摘　要： 针对编译器测试中最为重要的测试用例集构造问题，提出了针对 TTCN-3 语言类型系统的编译器测试用例集层次化、结构化的自动生成方案。语法方面，严格遵从语言规格说明中的扩展巴科斯-瑙尔范式(EBNF)；语义正确性上，采用定义“元素定义偏序文件”、建立抽象语法树等多种方法加以保证。实验表明新方案极大提高了测试用例集的生成效率，对 TTCN-3 类型系统语法、语义两方面都达到很好的测试覆盖，增强了发现编译器缺陷的能力。该方案对于其他语言的编译器测试也具有参考价值。

关键词： TTCN-3 类型系统 编译器测试 测试用例集自动生成 层次化、结构化方法

1　引言

TTCN-3(Testing and Test Control Notation Version 3)是由欧洲电信标准协会 ETSI 发布的标准化测试语言，被大量应用于协议及软件测试。如何构造出有效全面的测试用例集，使 TTCN-3 编译器通过全面可靠的测试，已经成为该语言能否更为广泛应用的关键问题。

手工构造方法虽然能生成目的明确的测试用例集，但有两个缺陷是其无法避免的: 1) 效率低下。2) 无法保证全面的测试覆盖。在自动测试用例集生成方面，Purdom 1979 年提出的算法以一个上下文无关文法为输入，生成了最短句子，而且文法中每条产生式被至少使用一次[1]。但该算法生成的只是最短句子的集合，并未考虑嵌套、递归等复杂情况，更重要的是，该算法并不保证生成的句子语义上的正确性。沈扬等人在其基础上提出了改进算法[2]，改进后的算法可以生成能对语法进行更细致的上下文相关覆盖的句子集合，但仍然没有考虑语义方面。Kalinov 等用复杂的 Montages 公式对 mpC 语言的表达式的语义进行描述，通过带反馈的递增式生成得到测试用例集[3]，但是公式本身的构造相当困难。Yoshikawa 等人在解决 Java JIT compiler 测试问题时提出随机生成测试用例集的方案[4]。该方案采用先生成测试用例框架，再生成控制流，最后填入比特码的方式。当生成过程遇到多种可选情况时，采用随机选取的策略。这种层次化的生成方法提高了测试用例集的生成效率，但大量的随机选择却导致了生成过程缺乏一定的导向性，无法做到全面的测试。

类型系统是程序语言的核心，特别针对 TTCN-3 而言，需要具有完备的类型系统，才能对各种复杂的消息结构进行描述。本文针对 TTCN-3 类型系统的特点，在总结手工测试经验的基础上，借鉴了文法句子自动生成、编译器构造等方面的技术，提出了层次化、结构化，并能保证语义正确性的高效的测试用例集自动生成方案。实验结果表明，该方案在生成效率、代码覆盖率、缺陷发现方面有着更好的表现。

2　TTCN-3类型系统介绍

TTCN-3 作为一种通用的协议一致性测试语言，能对协议中使用的复杂信息进行描述，其类型系统亦非常丰富。

ETSI 对 TTCN-3 类型系统作了详细说明[5]，将其类型分为简单基本类型、基本串类型、结构类型、特殊数据类型、子类型(包括列表约束 list、范围约束 range、长度约束 length)、配置类型及默认类型。其中，结构类型的嵌套层次及域的个数可以为任意多。而子类型约束按照特定规则，还可进行组合。图 1 为两个典型的类型定义。

① 收稿时间:2008-12-24

© 中国科学院软件研究所 http://www.c-s-a.org.cn

```
Type integer IntMix (1，3，5，10..30，99，100);
type record my_record
{
    IntMix outerF1,
    record
    {
        integer nestedF1,
        enumerated
        {
            enum1,
            enum2
        } nestedF2
    }outerF2,
    Record of Boolean outerF3
}
```

图 1 TTCN-3 类型举例

类型定义只能出现在模块定义部分，而常、变量定义等却能出现在多个位置，如函数(function)、可选步(altstep)、测试例(testcase)等结构内，这体现了使用常、变量时不同的上下文环境。

3 测试用例集自动生成思想

要对TTCN-3 语言编译器的类型系统进行全面的测试，生成的每个测试用例就应该包含以下语法单元：类型、类型相应的值集合(包括常量、变量)以及使用这些量的上下文框架环境。这些语法单元之间有一定的相关性，单元本身，如类型定义、值等，也具有结构性和层次性。在生成测试用例的过程中，要充分利用了这些特性。

本文就类型系统本身的特点，提出的测试用例集生成方案，以语法为驱动，在保证语法结构正确的基础上，进一步保证语义的正确性。具体思想是：以TTCN-3 语言规格说明给出的 EBNF(扩展巴科斯-瑙尔范式)为 Purdom 算法的输入，然后层次化的产生测试用例生成过程所需的语法单元(类型及测试用例的框架)。值的构造则在类型定义所对应的语法树节点基础上，使用递归下降的办法生成。最后将生成的各部分组合起来完成测试用例的构造。

文献[1,2,6,7]中的算法大都是将整个文法做为 Purdom 算法的输入进行一次生成。这种方法存在很大缺陷，因为在处理数量庞大的产生式时，如果单纯使用 Purdom 算法，生成的测试用例集中就会存在大量的重复部分，生成过程也会变得缓慢，效率十分低下。因此本文采用层次化的生成方法，将整个测试用例集的生成分为框架构造、类型构造和值构造三个层次，其中的前两个层次也用到了 Purdom 算法，但是，通过采用分层的思想，算法中需要处理的文法就得到了很大简化，极大提高了生成效率。

另一方面，Purdom 算法只能保证生成结果语法上的正确，语义方面则被完全忽略，其他文献[6,7]对 Purdom 算法扩展，可以处理部分语义，但是，给出的方法要么处理过程很复杂，要么不能很好的描述类型系统特定的语义要求。本文针对类型系统的语义特点，通过如下方法对语义正确性加以保证:

1) 在框架生成过程中，会出现在生成语言元素 A 时，发现语言规范要求在生成 A 之前，必须先定义元素 B 的情况。对这种元素出现先后次序的要求，可用事先给出的“元素定义偏序文件”来满足。

2) 类型约束的语义信息，由该类型对应的语法树中相应的属性域体现，并在进行值构造时使用。

3) 对于已经在测试用例中出现的类型、常量、变量等语言元素的定义，将其名字信息放入层次链表中，逐步构建出作用域空间，最终以树的形式，存放各作用域空间内可见的标识符。在使用已定义的语言元素时，必须满足层次链表要求，这就反映了 TTCN-3 中标识符作用域的概念。

通过以上方法，能够高效生成覆盖率高，且语法、语义都正确的测试用例集。

4 测试用例集生成系统

测试用例集生成系统的结构如图 2 所示。扩展模块将 TTCN-3 规范中的 EBNF 转换为 BNF，然后分别调用类型构造、框架构造、值构造模块生成相应语法元素，最终组合成测试用例。一系列组合之后，就得到了测试用例集。

整个生成过程之前，我们向类型列表中放入所有基本类型和基本串类型，值列表中放入多个与类型列表中的成员相对应的初始值。这些初始值的选取必须有一定的代表性，如 integer 类型的值应当包括正、负、零值，无穷小与无穷大等。类型生成过程会不断用到已有类型和值，如作为子类型定义时所需的父类型、作为范围约束(range)上下界的值、作为列表约束(list)的值等。而生成的结果则继续放入类型列表和值列表中，于是，类型列表和值列表逐渐丰富起来。通过这种方式，可以生成任意复杂度的类型和值。

© 中国科学院软件研究所 http://www.c-s-a.org.cn

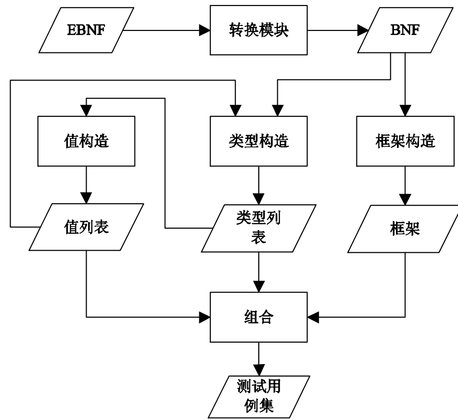

图 2　生成系统结构

本方案中使用到的 Purdom 算法相当复杂，为便于理解，这里给出一个对算法的简单介绍。该算法的执行是测试用例生成的基础，它以定义语言的文法产生式为依据，生成最短句子的最小集合，从而达到对所有产生式的全面覆盖。算法实现分为三步，首先需要收集文法中符号的信息，以决定使用哪个产生式来重写非终结符；然后获取产生式的信息，记录最短推导路径，以便能最快地生成句子；最后则是生成句子集合，生成过程会根据规则覆盖的情况(通过数组 MARK 记录)来判断当前使用哪个产生式来进行推导。

以下是整个生成过程的详细描述。

4.1 EBNF 转换

Purdom 算法的输入是 BNF，而定义 TTCN-3 的却是 EBNF，这就需要在调用算法之前，对其进行转换。转换包括两部分，一方面去除 EBNF 中特有的结构信息，另一方面对 EBNF 中某些非终结符进行替换，以满足层次化生成的需要。具体工作如下：

EBNF 中包含特有的形如“[abc]”，“{abc}”和“{abc}+”等结构信息的的产生式(其中，[abc]表示串 abc 可出现 0 次或 1 次，{abc}表示可出现 0 次或多次，{abc}+表示可出现 1 次或多次)，通过改写，可将其转换为 BNF。为使最终生成的程序结构更全面，可在参数的控制下，将可多次出现的结构进行转换。如：record 定义中，指定域的个数上限为 3 个，则 record 对应的 EBNF 转换后的情况如图 3。

在生成过程的不同层次中，某些非终结符的作用是不一样的。如在框架生成层次中，类型义、变常量定义等非终结符是不应该继续扩定展生成的，为达到这个目的，可将不必扩展的非终结符替换为临时终结符。而在类型生成层次中，这些非终结符表示其原本的含义，不用替换。

扩展前的EBNF:
RecordDef: : =RecordKeyword StructDefBody
RecordKeyword: : =“record”
StructDefBody: : =StructTypeId “{” [StructFieldDef { “,” StructFieldDef }] “}”
指定重复次数为3次后得到的BNF:
RecordDef: : =RecordKeyword StructDefBody
RecordKeyword: : =“record”
StructDefBody: : = StructTypeId “{” “}”
StructDefBody: : = StructTypeId “{” StructFieldDef “}”
StructDefBody: : = StructTypeId “{” StructFieldDef “,” StructFieldDef “}”
StructDefBody: : = StructTypeId “{” StructFieldDef “,” StructFieldDef “,” StructFieldDef “}”

图 3　record 类型 EBNF 转换

4.2 类型构造

类型定义的生成，以 EBNF 转换后得到的 BNF 为输入，将类型定义产生式左端的非终结符“TypeDef”作为文法开始符调用生成算法得到。

生成过程中，在处理复杂的结构类型时(如图 1 中的 my_recordType 定义中出现了三层嵌套)，会标识可扩展点以支持任意复杂的类型定义的生成。可扩展点表示类型定义中可以嵌套为内部类型的非终结符，如 EBNF 文法产生式 23[右端的非终结符 NestType-Def。可扩展点的集合已经事先给出。通过用已生成的类型定义对可扩展点进行替换，便可生成多层嵌套结构类型定义，而嵌套的层次则由用户参数控制。为保证替换的正确性，类型列表中已生成的每个类型会有一个信息结构与之对应。信息结构中包含的信息有：是否包含可扩展点、是否可作为嵌套类型、已嵌套层次等。当类型构造模块需要某个类型时，会从类型列表中根据信息结构，选取合适的已定义类型。

在进行子类型构造时，会用到确定类型对应的值，这些值从初始化值列表中得到。如构造带 Range 约束的整型子类型时，其上下界都是整型数值，这对值是从初始值列表中随机选取。通过对值按大小进行排序，以符合上界值必须大于等于下界值的语义要求。

整个类型列表的生成是一个由简单到复杂、由单层次到多层次的过程。

4.3 值构造

值的构造是通过遍历类型定义节点对应的语法树得到。

© 中国科学院软件研究所 http://www.c-s-a.org.cn

有两种方法可得到关于类型定义的语法树节点：1）通过在每条产生式后加入语法树创建动作，使得 Purdom 算法在生成类型定义同时，完成对应语法树的建立。2）使用已有编译器对生成的类型定义进行扫描后，取出相应语法树。考虑到语法树的建立有成熟的技术，不同的编译器实现在语法树建立的过程上基本上是一致的，因此可相信经过编译器扫描后生成的语法树是正确性的。本文使用第 2 种方法得到类型定义的语法树。如图 4 为类型 my_recordType 对应的语法树节点。

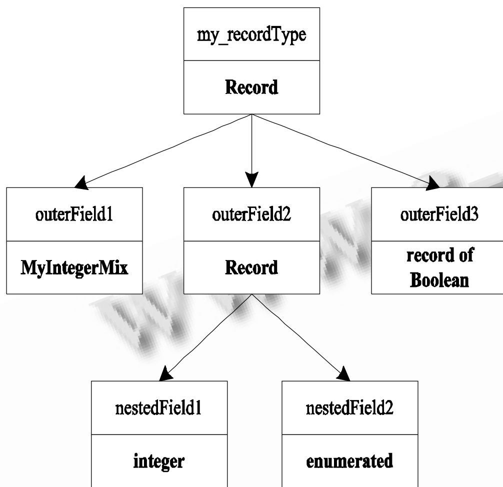

图 4　my_recordType 对应抽象语法树

有了语法树之后，值的生成变得很简单。采用递归下降的算法，从根节点逐步向下扩展，当遇到某种类型的子节点后，先查找值列表中是否已存在该类型对应的值，如有，则使用这个值，该子节点的处理完毕。如无，则继续扩展这个子节点。最坏情况下扩展到最下层子节点时，所需要的值就是一些基本类型所对应的值，这些值是可以从初始值列表中选取的。为使得生成的值具有代表性，要求在构造初始值列表时，基本类型的值本身具有代表性。由于类型的语法树已经包含了各种约束信息，进行值构造的时候将这些约束加以考虑，就能保证由其生成的值在语义上的正确性。图 5 为根据 my_recordType 类型的语法树生成的值。

将生成的值(加上其类型信息)放入值列表中，为构造更加复杂的值及类型提供支持，有效地提高了重用率。

4.4 框架构造

类型系统测试用例的构造，需要有常量、变量定义及使用的上下文环境。这部分工作由框架构造子模块完成。

```
Var my_recordType recordVar: =
{
    outerField1: =99,
    outerField2: =
    {
        nestedField1: =3,
        nestedField2: = nestedEnum1,
    },
    outerField3: ={true,  false}
}
```

图 5　my_recordType 类型变量

首先扫描 BNF，找到可以推导出常量、变量定义的非终结符 Nt(已由第一阶段转换为临时终结符)，然后调用“修改 Purdom 算法”，生成包含 Nt 的正确的测试用例框架。

“修改 Purdom 算法”是通过修改原 Purdom 算法第三阶段的数据结构 MARK 的值来实现的。MARK 为布尔型的一维数组， MARK[Nt]表示非终结符 Nt 的使用情况，为 true 表示该非终结符已被使用，false 表示未被使用过。将非终结符 Nt 的 MARK 值修改为 false，其他非终结符的 MARK 值设为 true。最后以语言的开始符作为文法开始符，执行 Purdom 算法，就会得到可包含非终结符 Nt 的上下文环境。

由于很多定义存在着先后顺序，如要在 testcase 中定义常量，必须构造 testcase 定义，而 testcase 的构造要求先有 component 的定义，这种先后关系在生成过程中必须考虑。本文引入“元素定义偏序文件”来解决这个问题。上述先后关系在“元素定义偏序文件”中表现为一条不等式：testcase<component。在生成框架过程中，通过查找“偏序关系文件”，如存在需事先定义的语言元素，则通过调用 Purdom 算法，生成相应的语言元素并插入框架中。

图 6 是以 testcase 为上下文的测试用例框架，其中 component 的定义是根据偏序文件的描述添加的。

```
Module module_1
{
    TYPEDEFINITION;
    type component MTC_Type
    {
    }
    testcase Tc01() runs on MTC_Type
    {
        CONSTDEFINITION;
        VARDEFINITION;
        setverdict(pass);
    }
}
```

图 6　testcase 为上下文的测试用例框架

© 中国科学院软件研究所 http://www.c-s-a.org.cn

4.5 组合成测试用例

在生成类型和值时，已经建立起两者的对应关系，从类型列表和值列表中选取相对应的类型及值，填入框架中，最后，在每个测试用例的后面加上“serverdict(pass);”语句便可完成一个测试用例的生成。通过类型、值、框架的不同组合，得到不同的测试用例，再将不同类型的测试用例分别存放在不同的目录中，从而完成整个测试用例集的构造。

5 实验数据及分析

通过指定待生成类型不同的嵌套层数及转换 EBNF 时的参数，我们得到了表 1 中的数据。

表 1　测试用例集生成结果统计

嵌套层次	测试用例数	缺陷发现数	代码覆盖率	生成时间（秒）
1	31	2	35%	0.2
3	122	5	62%	1.6
4	256	6	83%	4.5
5	698	6	85%	12.5
6	2305	7	86%	61.7

从表 1 中可知，随着嵌套层数的增加，得到的测试用例集迅速增大，发现的缺陷也随之增多。当嵌套层次过大时，不仅生成代价会很大，而且也不会显著的提高类型系统对代码的覆盖率，所以嵌套层次的指定应该趋向于一个合理值。结合在 TTCN-3 工业测试用例编写方面的经验，嵌套层数指定为 4 次最好。

手工开发的测试用例集生成过程中，测试小组 4 个人花了 1 周完成了 103 个测试用例的开发，而采用本方案，则可以自动完成测试用例集的生成。将两种方法得到的测试用例集应用在编译器开发的测试中，结果表明本方案生成的测试用例集完全包含手工编写的测试用例集。在指定嵌套层次为 4 的情况下，用例个数大约为手工编写的测试用例集的 2-3 倍，代码覆盖率及发现的缺陷数比手工测试用例都有了明显提高，具体数据如下:

表 2　测试结果对比

待测版本	手工测试例发现缺陷数及代码覆盖率	自动生成测试例发现缺陷数及代码覆盖率
V1.0	24，77%	32，82%
V1.3	12，75%	15，83%
V2.0	5，79%	7，84%

6 总结与展望

本文就 TTCN-3 类型系统测试用例集的自动生成提出了一种解决方案，该方案严格按照语言规范说明，采用层次化的生成思想，使用多种办法确保测试用例在语法、语义上的正确性，最终得到的测试用例集对类型系统进行了全面的覆盖，实验结果表明了该方案的可行性和有效性。

除了类型系统，TTCN-3 还有其他丰富的特性，如操作语义、程序流程控制等,针对这些特性的测试用例构造，是今后必须考虑的。

另外，对程序中错误的正确处理是编译器非常重要的一方面，如何生成包含错误的测试用例(异常测试用例)，用以对编译器报错、异常处理机制、健壮性等进行测试也是下一步工作的重点。

参考文献

1 Malloy BA, Power JF. An interpretation of purdom's algorithm for automatic generation of test cases. 1st Annual International Conference on Computerand Information Science, Orlando, FL, 2001.
2 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, 2005,17:96－110.
3 Kalinov A, Kossatchev A. Petrenko A. Coverage-Driven Automated Compiler Test Suite Generation Proc. of LDTA'2003(Language descriptions, Tools and Applications). Elsevier, 2003.
4 Yoshikawa T, Shirnura K. Random program generator for Java JIT compiler test system. QSIC2003 (the 3rd International Conference on Quality Software), 2003.
5 ETSI. ES 201 873-1. The Testing and Test Control Notation version 3. Part 1: TTCN-3 Core Language. France: ETSI 2007－02.
6 Bazzichi F, Spadafora I. An automatic generator for compiler testing. IEEE Transactions on Software Engineering, 1982, SE-8(4):343－353.
7 Celentano A, Reghizzi S, Vigna DP, Ghezzi C. Compiler testing using a sentence generator. Software–Practice and Experience, 1980,10:897－913.

© 中国科学院软件研究所 http://www.c-s-a.org.cn