基于模板的代码转换工具①

刘 毅，陈香兰，李 曦

(中国科学技术大学 计算机科学与技术系，合肥 230027)

摘 要：代码转换是代码自动生成过程中的重要环节。提出一种基于模板、可适用于任意文法描述代码之间转换的“属性匹配-替换”算法。利用该算法，成功实现了 OSEK 规范中 OIL 语法描述代码到 C 语言代码的转换。

关键词：代码转换；代码自动生成；模板；OSEK；OIL

Template Tool for Code Conversion

LIU Yi, CHEN Xiang-Lan, LI Xi

(Department of Computer Science, University of Science and Technology of China, Hefei 230027, China)

Abstract: Code conversion technology plays an important role in fields of code generation. This paper presents a template-based “attributes match–n, replace” algorithm for code conversion between programming languages. Using this method, we successfully convert OIL, which is one of important parts of OSEK specifications, to C.

Keywords: code conversion; code generation; template; OSEK; OIL

1 简介

代码自动生成是提高软件生产率的重要手段，它根据工程设计中的设计结果(数据模型)依照一定的领域规则(模板)生成程序代码。代码转换是代码自动生成过程中的重要环节。代码转换技术还被广泛运用在软件维护、遗留系统的升级改造以及软件逆向工程等领域中。

代码转换根据其目标语言的生成方式可分为固定转换与可配置转换两种。区别在于前者在转换时源代码与目标代码之间是一一映射的关系；后者可以通过配置生成不同的目标代码。可配置转换工具一般被称作模板工具或模板引擎，其配置文件被称为模板。

现有模板工具大多采用"变量替换"算法。采用此算法的功能代码由变量、数值操作以及控制语句构成。这种算法运作过程是：首先，制定变量定义规则并预定义系统环境变量；然后，模板工具在读入代码时根据代码内容定义变量并将这些变量赋值；最后，在生成代码阶段将模板文件中的变量名用变量值替换。这种替换算法破坏了原数据模型的结构化，为复现原来的结构，模版中不得不编写大量的代码，造成模版编写复杂，运行效率低下。

本文提出“属性匹配-替换”算法(简称“属性替换”算法)将依照语法树的节点属性构造模板，且不破坏原语法树结构。算法通过输入代码构造代码语法树，并依据模板构造模板语法树(匹配节点树)。然后自顶向下对两棵树进行匹配。对于匹配节点，用模板语法树节点属性的替换代码块替换代码文件中的相应部分。

OSEK 标准是汽车电子操作系统接口规范。它通过 OIL[1]定义应用程序配置文件描述方式，实现代码描述的开放性。OIL 描述文件独立于 OSEK OS[2]，经 OIL 转换工具转换后才能与 OS 及其他文件一起编译连接。目前的 OIL 转换工具均由 OS 生产厂商研发，并与其 OS 完全绑定(如 Trampoline 的 goil[3]和 freeOSEK 的 Generator.php[4])，不同厂商的 OIL 工具与 OS 完全不兼容。本文设计的 OIL 模板转换工具 OILConverter 解决了此问题。

下文将描述“属性替换”算法的原理，给出 OILConverter 的设计与实现过程，最后使用该工具分别对 Trampoline 和 FreeOSEK 的 OIL 描述进行 C 语言转换，结果表明该工具能够完全兼容 Trampoline 和

① 基金项目:电子信息产业发展基金(财建[2008]329,工信部运[2008]97)
收稿时间:2010-05-19;收到修改稿时间:2010-07-15

FreeOSEK。

2 “属性替换”算法原理

本节首先介绍算法涉及到的几个术语，然后分小节介绍算法原理。术语如下：

代码语法树：基于待转换的代码构建的抽象语法树。与此类似，基于模板构造的抽象语法树叫做模板语法树。

匹配节点：为构造替换输出，需要在模板中捕获代码语法树中每个节点的属性，这种用以捕获属性及替换输出的节点称为匹配节点。

匹配关系：若某一匹配节点与某一代码语法树节点之间存在映射关系，则称匹配成功。它们之间是多对多的映射关系。

层次匹配：若匹配节点树中某节点与代码语法树中某节点存在匹配关系，则它们的子节点间存在匹配关系。

匹配节点树：模板语法树节点由两个部分构成匹配部分与替换部分，前者用来获取代码语法树节点的属性即匹配节点，后者用以构造替换称为替换节点。由匹配节点构成的自顶向下的层次关系，称之为匹配节点树。

条件匹配：对于某两个数值，若在指定条件下满足指定关系，则称它们满足条件匹配关系。

属性匹配：代码语法树中每个节点都对应一个产生式。为匹配此产生式定义一个由属性名与属性值构成的二元组。此二元组与代码语法树中某节点满足如下关系则称它们满足属性匹配关系：

- 代码语法树节点与此产生式之间存在映射关系；
- 该节点的由属性名与属性值构成的键值对集合中，存在键与二元组中属性名相同而且值与属性值满足条件匹配关系的元素。

匹配节点类：匹配节点的结构与上述二元组类似。只是匹配节点用以匹配整个代码语法树，因此多了层次结构。每个代码语法树节点均可有多个类似的二元组与之存在属性匹配关系，在这些二元组中相同属性名的归为一类，称之为“匹配节点类”。

匹配节点集合：代码语法树节点与匹配节点为多对多的映射关系，与同一代码语法树节点存在映射关系的匹配节点组成的集合称为“匹配节点集合”。

匹配节点查找算法：代码语法树中每个节点都对应一个产生式。匹配节点与代码语法树节点成功匹配需要满足两个条件：层次匹配与属性匹配。根据给定的代码语法树节点，查找出与之匹配的匹配节点为匹配算法的核心。算法思想为：

若代码语法树节点为代码语法树根节点则将该节点的匹配节点集合与匹配节点树根节点的子节点集合中的元素逐一匹配，匹配成功的匹配节点即为所求；

否则，根据层次匹配关系将该节点的匹配节点集合与其父节点对应的匹配节点的子节点集合中的元素逐一匹配，匹配成功的匹配节点即为所求；

2.1 约定

终结符用小写字母表示，非终结符用大写字母表示。候选式中全为终结符的非终结符称之为简单非终结符，反之候选式中含有非终结符的非终结符称之为复杂非终结符。本章约定 A、B、C 为简单非终结符，其余大写字母为复杂非终结符。其中 A ::= a | b | c ; B ::= x | y | z。

抽象 BNF。对 BNF 按照如下方式进行语义抽象：

排序：匹配规则与属性在产生式中的次序无关，为描述清晰简单属性前置非终结符前置并分隔非终结符。

消去空候选式，如产生式 S ::= r A | ε 记为 S ::= r A；

简化节点层次，如 S ::= A R; R ::= r 记为 S ::= r A。

2.2 匹配节点树的构造

对形如 S ::= r A B 的产生式，用(Name(A), Value(A))表示匹配节点，其中 Name(A)与 Value(A)表示 A 的属性名与属性值，用 A(Value(A))进行简记。其中 Value(A)= ε 表示匹配任意值。即 A(a) 匹配 S ::= r a，A()匹配 S ::= r (a | b | c)。同时 A()、B()分别表示匹配节点类 A、B。[A,B]为此产生式的匹配节点类集合。

'>>'表示匹配， A(a)匹配 S ::= r a，记作 A(a) >> S ::= r a。

层次('.')：对形如 S ::= A T ; T ::= r B 的产生式，A().B() >> S ::= r A B，称 B()为 A()的子节点，A()为 B()的父节点。

匹配节点树的构造：空节点为根；无父节点的匹配节点为根节点的子节点；对形如 A().B()的匹配节点，将 B()作为 A()的子节点。

2.3 匹配节点的冲突与选择

匹配节点与产生式匹配过程中存在三类匹配冲突：多个匹配节点匹配同一个产生式、同一个匹配节点匹配多个产生式以及一个匹配节点匹配一个产生式的多个属性。以下分别简记为第一类冲突、第二类冲突与第三类冲突。

“一类冲突”：对形如 S ::= A B 产生式存在 A()，B()两种类型的匹配节点。即 A(a) >> S ::= a B ; A() >> S ::= (a | b | c) B ; B(x) >> S ::= A x ; B() >> S ::= A (x | y | z)。A(a)与 B(x)均匹配 S ::= a x。当 S::= a x 这种推导出现时匹配节点 A(a)、B(x)、A()、B()的选取存在冲突。

处理一类冲突的两种方案：1) 对于同一个产生式只定义一类匹配节点，对形如 S ::= A B 或者选择 A 类匹配，或者选择 B 类匹配；2) 定义优先级。后者能够支持多类节点并存以提高匹配节点设计的灵活性。

优先级的指定：1) 同一类匹配节点，空值优先级最低，其余依照在匹配规则中出现的先顺后序定义，先出现的优先级高，如 A(a)、A(b)平级而 A()最低；2) 不同类匹配优先级相同，先在匹配规则中出现的优先级高，空值优先级最低，不同类的空值匹配，先出现的优先级高，如 A(a)、A(b)、B(x)、B(y)优先级相同，A()与 B()优先级最低。匹配规则一般在模版中编写，即匹配节点类集合中元素的优先级由模版决定。

匹配节点的选择：对于含有多个简单非终结符的产生式，将每一个非终结符均定义一类匹配节点毫无意义，而且匹配类别越多匹配规则的编写越复杂。匹配节点的选择算法，1) 推导出 Id 的属性优先，但若多个属性都能推导出 Id，只选择其中之一； 2) 对于不能推导出 Id 的属性，优先级与候选式个数成正比。

“二类冲突”，对形如 S ::= r A B | q A C 的产生式，A 类匹配节点同时匹配两个或多个候选式，如，A(a) >> S ::= r a B 与 A(a) >> S ::= q a C 并存，匹配节点 A(a)难以确定匹配哪个产生式。

“二类冲突”的三种解决方案：1) 取消产生冲突的该类匹配节点；2) 无冲突的匹配节点“下移”，是指定义新的非终结符并把该类以外属性作为新定义的非终结符的候选式，最后用新定义的非终结符与产生冲突的属性一起替代原来的产生式，如，定义 X ::= r B | q C 原来的产生式替换为 S ::= A X；3)产生冲突的该类匹配节点“下移”，是指定义新的非终结符并把该类属性作为新定义的非终结符的候选式，最后用新定义的非终结符与其他属性一起替代原来的产生式，如，定义 X ::= A 原来的产生式替换为 S ::= r B X | q C X ，这样 B().A() >> S ::= r B A ; C().A() >> S ::= r C A 而不再产生冲突。第一种方案处理简单但降低了匹配规则编写的灵活性，第三种方案不利于节点层次优化，本文采用第二种方案。

“三类冲突”，对形如 S ::= r A q A C 的产生式，A 类匹配节点同时匹配两个或多个属性，如，A(a) >> S ::= r a q A C 与 A(a) >> S ::= r A q a C 并存，匹配节点 A(a)难以确定匹配哪个属性。

“三类冲突”的解决方案：产生冲突的属性“下移”，是指定义新的非终结符并将该属性作为新定义的非终结符的候选式，最后用新定义的非终结符与其他属性一起替代原来的产生式，如，定义 E ::= r A ; F ::= q A 原来的产生式替换为 S ::= E F C，这样 E().A() >> S ::= r A F C; F().A() >> S ::= r E q A C 而不再产生冲突。

2.4 节点匹配的层次优化与匹配节点的提升

节点匹配算法中增加层次的优势在于能够提供精确匹配，但层次过多会导致匹配规则编写复杂容易引发错误。减少层次的有效方法是匹配节点的合并与提升，但对匹配节点的盲目合并或提升会导致“二类冲突”。“二类冲突”只会在兄弟节点间产生。

匹配节点的“合并”：对形如 N ::= r E ; T ::= p F ，且不存在 S::= N | T(即 N、T 产生式不同时为某一产生式的候选式)，定义新产生式 X ::= r | p，并定义 X(r) >>

S ::= r E ; X(p)>> T ::= p F ; X() >> N ::= r E 且 X() >> T ::= p F。

匹配节点提升算法：1) 对整个匹配节点树进行“二类冲突”检测，发现“二类冲突”用无冲突节点“下移”方案解决；2) 深度优先遍历，若某节点与兄弟节点的所有子节点不产生“二类冲突”且与父节点的兄弟节点的所有子节点均不产生“二类冲突”，则该节点“上移”，作为父节点的兄弟节点；3) 对每个节点上移至产生“二类冲突”为止。

2.5 替换算法描述

模板语法树节点有匹配节点与替换节点构成，相应的模板有匹配部分与替换部分构成。替换算法即为对匹配部分获取的属性值与替换部分的功能代码进行处理的算法。

依照替换字符串的作用替换可分为纯文本替换与功能替换。纯文本替换是指用来替换的字符串作为纯文本输出，功能替换是指用来替换的字符串具有特定的含义，需要转义后构造输出。

产生式属性值的存储与传递。以产生式的属性名为键，属性值为值，组成键值对，并与其他辅助键值对一起放入数组或链表。替换代码块以属性名或其他辅助键为参数，获取键值。

普通替换与参照替换。依照产生式的语义替换可分为普通替换与参照替换。普通替换是指某一产生式在语义上不依赖于其他产生式，如变量定义；参照替换是指某一产生式在语义上依赖于其他产生式，如变量赋值。对于普通替换只需提供本产生式的属性名与属性值，参照替换还需提供被参照产生式的属性名与属性值。

3 OILConverter设计与实现

本节描述 OILConverter 的设计过程，包括匹配节点树设计，模版文件结构描述，属性值的传递及处理方式。最后以 Trampoline 中的 OIL 转换为例给出实验分析。

3.1 匹配节点树的设计

使用“属性替换”算法对 OIL 进行分析后，构造的匹配节点树如表 1 所示。

表 1　匹配节点树结构

匹配节点	可能的子节点
root	implementation,application
implementation	objectType,objectName,attributeType, attributeName,boolValue,enumValue
objectType	attributeType,attributeName,boolValue, enumValue
objectName	attributeType,attributeName,boolValue, enumValue
enumValue	attributeType,attributeName,boolValue, enumValue
boolValue	attributeType,attributeName,boolValue, enumValue
attributeType	attributeType,attributeName,boolValue, enumValue
attributeName	attributeType,attributeName,boolValue, enumValue
application	attributeValue,attributeNameType, attributeType,attributeName

3.2 模板文件结构

模板文件由节点替换语句组成，每条语句由替换节点与替换代码块两个部分构成。前者负责构造替换节点树，后者负责替换代码。模板文件的 BNF 表示如图 1 所示。

```
<File> ::= <rules>;
<rules> ::= ε| <rule> <rules> ;
<rule> ::= "${" <repNode> "}" <replaceRule> ;
<repNode> ::= <Ident> "(" <repAttriValue> ")" ;
<repAttriValue> ::= ε | <Ident> | <Number> ;
<replaceRule> ::= ":" "{" <rule> "}"
  | <replaced> "=>" "{" <replaceSection> "}" ;
<replaced> ::= ":" | "${" <repNode> "}" | <String> ;
<replaceSection> ::= ε
  | <replaceSectionItem> <replaceSection> ;
<replaceSectionItem> ::= "@{" <auxFuncCode> "}"
  | "${" <headOrTail> "}" ":" "=>" "{" <replaceSection> "}"
  | <output_as_what_they_are> ;
<headOrTail> ::= "head" | "tail" ;
<auxFuncCode> ::= <Ident> "(" <parameters> ")" ;
<parameters> ::= ε | <parameter> "," <parameters> ;
<parameter> ::= <Ident> | <Number> | <String> | <auxFuncCode> ;
```

图 1　模板文件的 BNF 描述

3.3 属性值的传递方式与辅助函数

模板中采用“属性名+"()"”的形式获取该属性名对应的属性值。模板工具提供了大量的辅助函数，有变量定义函数、算术运算函数、字符串操作函数、关系运算函数、逻辑运算函数、分支函数、子节点指定

函数 children 及其他辅助函数"aux"、"debug"等。其中 children 函数用以指定子节点的匹配节点，其作用类似于 C 语言中的 goto 语句，它的定义对于处理递归文法与简化模板编写都具有十分重要的意义。

3.4 转换工具的应用实例

```
tpl_exec_static static_descriptor_of_task_periodicTask = {
 CONTEXT_OF_TASK_periodicTask,
 STACK_OF_TASK_periodicTask,
 function_of_task_periodicTask,
 NULL,
 task_id_of_periodicTask,
 (tpl_priority)2,
 1,
 TASK_EXTENDED,
 #ifdef WITH_AUTOSAR_TIMING_PROTECTION
  NULL
 #endif
};
```

图 2 goil 转换 TASK 后静态部分输出

```
${objectType(TASK)} :{
 ${self()}=>{
  tpl_exec_static @{strcat("static_descriptor_of_task_",objectName())} = {
   @{strcat("CONTEXT_OF_TASK_",objectName())},
   @{strcat("STACK_OF_TASK_",objectName())},
   @{strcat("function_of_task_",objectName())},
   NULL,
   @{strcat("task_id_of_",objectName())},
   (tpl_priority)2,
   1,
   TASK_EXTENDED,
   #ifdef WITH_AUTOSAR_TIMING_PROTECTION@{echo("\n")}
    NULL@{echo("\n")}
   #endif @{echo("\n")}
  };
 }
}
```

图 3 相同功能的 OILConverter 模板

TASK 对象表示应用程序中的一个 OSEK 操作系统任务，其作用类似于操作系统中的进程。以 TASK 对象为例，在 trampoline 项目的 defaultApp Workstation.oil 文件中有这样一个 TASK 对象：TASK periodicTask{PRIORITY =5; AUTOSTART= FALSE; //…}。goil 对该对象的转换分为动态静态两个部分，其中静态部分输出代码如图 2 所示。通过对 goil 源码的分析，TASK 对象的静态部分在 OILConverter 的模板表示如图 3 所示。实验结果显示二者输出完全一致。

4 总结

本文提出了基于“属性替换”思想的代码转换算法，并基于该算法设计出了 OILConverter 工具。实验结果表明该工具能够完全兼容 Trampoline 与 FreeOSEK。

参考文献

1 OSEK Group. 2004. OSEK/VDX system generation OIL: OSEK Implementation Language version 2.5. http:// www.osek-vdx.org.

2 OSEK Group. 2005. OSEK/VDX operating system specification version 2.2.3. http://www.osek-vdx.org

3 Trampoline Group. http://trampoline.rts-software.org/

4 FreeOSEK Group. http://opensek.sourceforge.net/