第 34 卷　第 13 期　　计　算　机　工　程　　2008 年 7 月
Vol.34　No.13　　Computer Engineering　　July 2008
·软件技术与数据库·　文章编号：1000—3428(2008)13—0043—03　文献标识码：A　中图分类号：TP311

C/C++软件测试工具的元数据结构设计与实现

沈　雷，李　翔，邵培南
(华东计算技术研究所，上海 200233)

摘　要：针对用 C/C++语言进行的语义分析，设计一种中间结构，即元数据结构。元数据结构实现了源代码的语义层次上的抽象，通过元数据结构和相关应用语义配置，过滤出源程序中符合应用的语义内容，实现软件测试工具的程序插装等功能。在中间结构实现过程中，构造一个二次解析引擎，以解决传统解析方法的复杂性及不确定性，实现对各种编程语言的支持。

关键词：软件测试；语义层次；元数据结构

Design and Implementation of Meta Structure in C/C++ Test Tool

SHEN Lei, LI Xiang, SHAO Pei-nan
(East-China Institute of Computer Technology, Shanghai 200233)

【Abstract】 This paper proposes a new meta structure which makes C/C++ source codes with semantic tag. Using the meta structure with semantic configuration related to application, test tool can filter some semantic tags for instrument and so on. A dual parser engine is developed to create the structure, which avoids the complexity and indeterminacy of classic methods, and can support kinds of programming languages very well.

【Key words】 software test; semantic levels; meta structure

1　C 软件测试工具

嵌入式软件测试工具主要针对 C/C++语言进行语义分析，实现软件的静态和动态 2 种视图。通过软件的静态视图，测试人员可以观测软件的组织结构、控制流程、数据流图等，并进行质量度量。动态视图测试人员可以观察软件的执行状态，分析执行性能，并记录测试实例对源程序的覆盖信息。上述所有功能要求对 C/C++程序的语义信息进行记录，然后在相应的应用中抽取语义信息，实现静态功能和动态功能。为了满足以上需求，嵌入式软件结构测试工具应具有图 1 所示的结构。

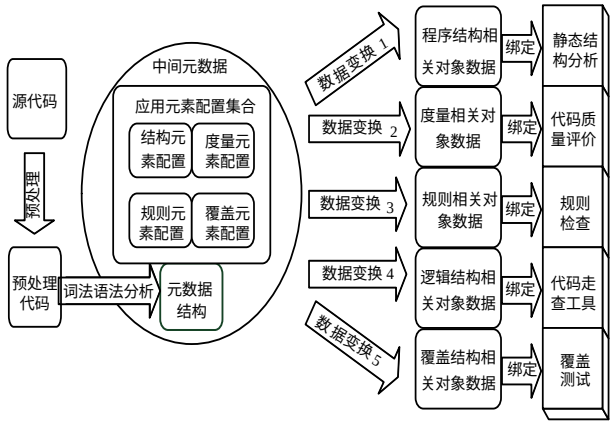

图 1　嵌入式软件测试工具结构

静态分析过程主要利用语法分析器对原程序进行语法分析，在此基础上构建中间结构，并进行静态分析，然后将静态分析结果保存成为一个中间结果结构，使其可以方便地以不同形式的图表显示出来。

动态分析分为动态插装与测试分析 2 个部分。插装是指在原程序中插入测试监测代码，用以跟踪程序覆盖的情况。动态插装是在中间结构的基础上进行插装，产生插装后的源代码文件。插装文件在开发环境下编译连接成可测试程序。测试分析的过程包括管理测试实例、执行测试、记录覆盖信息并分析、通过各种可视化界面输出给用户。

可以看出，不管是软件静态质量分析、还是覆盖测试分析，中间结构都占有十分重要的地位，是各个部分得以实现的基础，在整个工程中处于核心的位置。因此，设计一个合理有效的中间结构表达方式，可以简化后续工作的复杂度，提高程序运行的效率，并且为程序将来在功能上的扩充做好准备。

2　中间结构

中间结构[1]通过分析获得源程序的语法结构和一定的语义信息构建而成。中间结构的建立保持方案的独立性。软件结构分析包括类表、类继承关系、类-函数耦合关系、函数调用关系、控制流程。

静态文档分析源程序的复杂度、全局变量文件信息等。所以中间结构必须包含相关的全部信息：类，函数，变量本身的信息，例如全局变量和静态变量定义、类的结构、友员、静态成员，函数的内部结构、返回值、参数，以及它们之间的交互信息，如函数调用关系、类-函数耦合关系、类-类耦合关系、类继承关系。

通过语法分析，本文构建了软件测试系统的核心数据结构——层次语义模型(Hierarchical Semantics Model, HSM)，图 2 即中间结构的层次模型。一个模型建立完成后需要一种对应的数据存储结构，这里选择元数据[2]结构作为中间结构的数据表现形式。

作者简介：沈　雷(1983－)，男，硕士研究生，主研方向：软件测试，软件工程；李　翔，工程师；邵培南，研究员

收稿日期：2007-07-10　**E-mail：**shenyutian@126.com

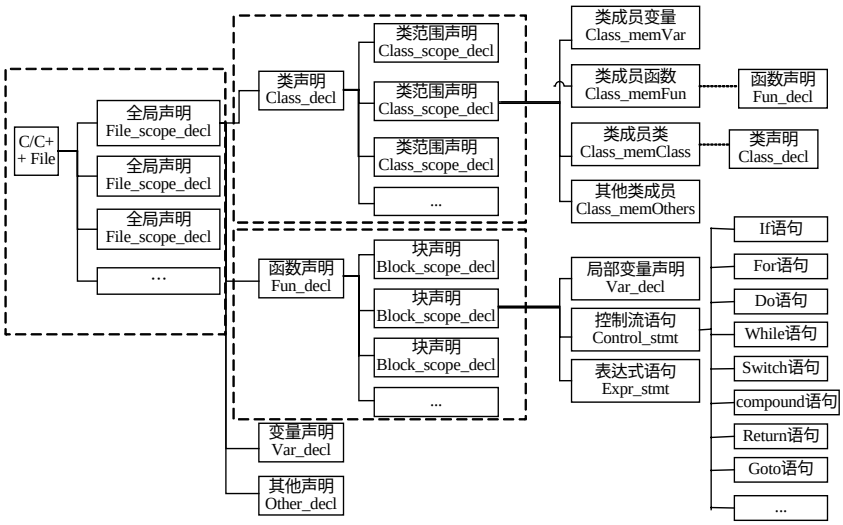

图 2　中间结构语义层次模型

3　语法解析

3.1　语法解析的目的

语法分析的功能有：(1)传递程序语法成分的信息(类定义、函数定义、引用点、各种语句、表达式、文件的起始和结束点、统计信息等)；(2)分析全局、静态变量的定义、引用信息；(3)函数引用的分析要进行形参和实参的比较，要处理缺省参数的问题、缺省的类型转换问题、参数匹配的规则应用问题，分析重载函数、隐式调用的构造函数和析构函数问题；(4)条件表达式的后缀式表示。

因此，一个良好的解析结果结构，要求能够很清晰地表达出上述各项信息。

3.2　待解决的问题

完成语法解析的工作需要选择一个解析器，经过比较后最终选择了programmar[3]。它提供ActiveX接口，使得在程序中调用其功能十分方便。通过编写语法定义语言(Grammar Definition Language, GDL)，经过编译生成一个二进制的语法文件(GMR)，程序运行时，解析引擎可以直接使用语法文件中包含的信息对输入的数据流进行处理，最后得到一棵解析树，将输入流中的对象映射为语法符号，可以通过调用相应的API，在树中定位、查找获取数据元素。利用programmar的解析引擎编写c/cpp的语法定义语言cpp.gdl并编译生成cpp.gmr之后，便可以对c及cpp文件进行词法和语法分析，生成一棵语法树。

cpp.gdl 中将所有声明都定义成统一的形式：

```
 declarator<BACKTRACK> ::=
        [template_spec]
        [typedef_tag]
        [decl_attr_list]
( (class_spec | enum_spec) [type_attr_list] [decl_item_list]
        | type_name [type_attr_list] decl_item_list
| decl_item (? ^.*method_stuff | ^.*typedef_tag))
        (";" |(? ^.*method_stuff.method_body));
```

这导致了解析树中表示类、函数及变量等声明的节点在结构上是相同的。

除此之外，解析树中表示函数声明和函数调用的节点在无参数的情况下无法区分，当输入文件如下时：

```
void hello();
void main()
{
hello();}
```

经过解析后得到的解析树中，2 个 hello 节点的结构在 decl_item 子树中是完全相同的(图 3)。

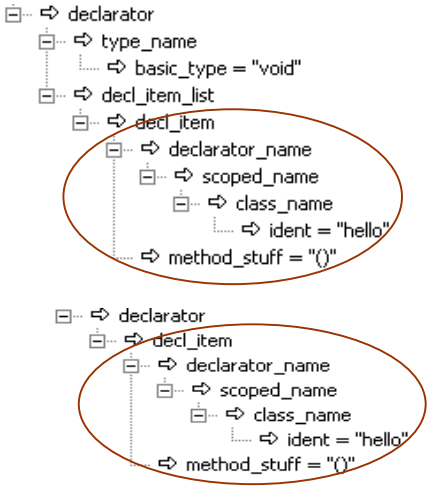

图 3　函数声明和函数调用的解析树

这样，无法根据该节点的解析结果确定节点的类型，必须结合解析树中和该节点相关的其他节点加以分析，才能获得准确的结果。

语法解析中还有其他的一些问题，例如对声明节点确定声明的作用范围，在解析树中都没有直接的表示，要从这样一个数据结构中直接提取语法信息是十分困难的。

3.3　解决方案

为了解决上述问题，消除语义上的不确定性，需要对得到的语法树进行额外的分析。通常有 2 种方法：自顶而下和由底向上[4]。

自顶而下的方法是从解析树的根节点开始按层次一个个遍历其子孙节点，直到所获得的信息能够唯一确定目标节点类型时才停止追溯。为了方便分析过程，输入的源文件已经过预处理，不再包含以“ #... ”开头的宏定义。生成的语法树根节点的子节点仅由一个个 file_scope_decl 组成：

```
file_scope_decl ::=
 pragma_stmt
        | linkage_spec
        | using_directive
        | using_decl
        | namespace_def
        | namespace_alias_def
        | declarator
        | ";"
        ;
```

得到 file_scope_decl 子节点类型后，继续往下分析，此处以 declarator 节点为例，要确定一个声明的具体类型，必须遍历 declarator 的子节点：

```
declarator<BACKTRACK> ::=
  [template_spec]
  [typedef_tag]
  [decl_attr_list]
((class_spec|enum_spec)    [type_attr_list][decl_item_list]
  | type_name [type_attr_list] decl_item_list
|decl_item(?^.*method_stuff| ^.*typedef_tag))
  (";" |(? ^.*method_stuff.method_body));
```

可以根据子节点内容得到该声明的类型，继而进行下一

步的分析。如果 declarator 的子节点中存在 class_spec，则说明这个 declarator 是一个类定义，接着依据类的内部结构往下分析：

```
class_spec ::=
  class_key
  [class_specifier_list]
  [class_name[":"inheritance_spec_list]
  | class_name_error]
  [class_definition];
```

到这一步，可以获得该类的定义信息了。class_key 表示类定义的类型是 class, union，还是 struct；class_name 表示类的名称；如果存在 inheritance_spec_list，包含了该类的继承类；class_definition 是类定义的过程语句：

```
class_definition ::=
        "{"                    [class_scope_decl_list]
        ("}" | RECOVER_TO_RBRACE ) ;
```

class_scope_decl_list 是类内部的声明语句，包括友元声明、类成员函数声明、类成员变量声明以及如下关于类变量的权限声明：

```
class_scope_decl_list ::=
        {class_scope_decl} ;
class_scope_decl ::=
           access_decl ":"
        | friend_decl
        | declarator
        | pragma_stmt
        | ";"
        ;
```

获取各个声明的信息后，对类层次的分析基本完成。

其他的分析过程，例如对函数的分析过程，以及对控制流语句的分析过程与此类似。

自顶而下的方法从树根节点开始分析，按层次逐步推进，条理清晰。但它仅适用于对整个文件做解析，无法对文件中某一选定的内容进行解析。

由底向上则是从生成解析树的叶子节点进行分析，往上回溯，直到找到可以确定该节点类型的标志性节点。

在解析树中，叶子节点基本上由下列几种类型组成：

```
basic_type|ident|numeric|typedef_tag|class_key|
access_spec|string_literal|operator
…
```

除了 ident 之外，其他节点都有固定的含义，可以直接确定节点类型，因此，只需要对 ident 节点进行分析。往上递归查找其祖先节点，直到找到可以确定其类型的节点。例如，在树中找到 ident 节点之后向上找祖先，如果最先遇到的是 qualified_name 节点，说明这是一次函数调用，ident 节点是被调用的函数名；如果最先遇到的是 declarator_name 节点，则说明这是一次声明，判断声明的类型还需要继续往上回溯，找到 declarator 节点后才能确定。

由底向上方法的优点是从叶子节点开始，适合分析源代码中某一选定项。但分析过程需要回溯很多层次，对不同叶子节点的分析过程中可能存在大量重复步骤，效率较低。

不管是自顶而下还是由底向上，它们的实现都需要在程序中编写大量的代码处理各种情况，结果导致程序过高的复杂度，并且最终得到的程序不具有重用性，一旦需要改进系统使其能实现对 C 以外的语言(比如 Java)的支持，那么几乎需要重写全部的代码，这样便导致了程序开发的低效，开发的周期也将变得很长。

3.4 改进的解决方案

为了让程序开发变得高效，并且开发好的系统将来可以扩充，实现对不同语言的支持，需要一种更好的方法。由于 programmar 可以根据语法定义语言文件描述的不同语法对不同输入流进行解析，因此可以设计一种语法定义语言，它以 programmar 解析后的语法结果为输入，经过第 2 次解析消除语义模糊性，得到结构更清晰、更准确的语法树，即所需要的元数据结构。

为了实现这个目标需要做的准备工作包括：对原始 cpp.gdl 文件进行修正，避免其中的缺陷导致对某些源文件无法解析或者解析出错误的结果；对其进行优化，调整生成树的内部结构，使第 2 次解析可以更方便地进行。

因 programmar 解析的输入流是一个字符串流，在第 1 次解析之后，必须对得到的解析树进行转换，使之成为由一串字符表示的树。具体的实现方法有很多，这里采用层次遍历法。转换后树的结构如下：

```
{root{A}{B}{C}{…}}
```

大括号对的内部表示一棵树。“ { ”后面紧跟的 root 是该树的根节点，之后是一连串的大括号对，每个括号对表示以 root 节点子节点为根的一棵子树，即 A, B, C 各表示 root 的不同子节点树。对于叶子节点转换后的树，则只有一个 root，形如“ {root} ”。这个结构清晰地体现了树的内部结构，使得第 2 次解析对输入流的分析更加容易。

二次解析消除了数据元素节点含义的模糊性，删除了部分多余的语法符号节点，得到的元数据结构精练易懂。二次解析均利用 programmar 引擎提高了程序功能的可扩充性，通过编写相应的语法定义语言，无须对程序代码进行修改就可以实现对各种不同语言的支持。

3.5 示例

输入源码如下：

```
class a
{
public:
  int i;
  class d
  {
  };
private:
  void   hello();
}
```

将 programmar 一次解析后的结果转换成输入流：

```
{program{file_scope_decl{declarator{class_spec{class_key}{class_name{ident}}}{class_definition{class_scope_decl_list{class_scope_decl{access_decl{access_spec}}}{class_scope_decl{declarator{type_name{basic_type}}{decl_item_list{decl_item{declarator_name{scoped_name{class_name{ident}}}}}}}}}{class_scope_decl{declarator{class_spec{class_key}{class_name{ident}}{class_definition}}}}{class_scope_decl{declarator{type_name{basic_type}}{decl_item_list{decl_item{declarator_name{scoped_name{class_name{ident}}}}}{method_stuff}}}}}}}}}
```

进行第 2 次解析后，得到中间结构表示树如图 4，所展示的程序结构信息要比原语法树清晰很多。

（下转第 48 页）