中国科学 E辑：信息科学 2008 年 第 38 卷 第 12 期：2084 ~ 2098
www.scichina.com info.scichina.com
《中国科学》杂志社
SCIENCE IN CHINA PRESS

Garden:一种面向领域语言的集成开发环境

张乃孝[①②]，琚小明[①*]，孙猛[③]

① 华东师范大学软件学院，上海 200062;
② 北京大学数学科学学院，北京 100871;
③ 数学与计算机科学国家研究所，阿姆斯特丹 1090GB，荷兰
*E-mail: xmju@sei.ecnu.edu.cn

收稿日期: 2007-11-14; 接受日期: 2008-06-20
国家自然科学基金(批准号: 60473056)、上海市基础研究重点项目(登山行动计划)(批准号: 06JC14022)和中国高技术研究发展计划(批准号: 2006AA01Z165, 2007AA010302)资助项目

摘要 随着计算机技术的发展，通用语言已经不能满足各种领域应用程序开发的需要. 由于领域语言提供领域专用术语和符号的概念，支持该领域中的各种处理，能够简洁、有效地构造该领域应用程序，因此，领域语言成为当前计算机语言研究中的热点. 文中在面向模型的变换型软件开发方法和语言的抽象与封装机制研究的基础上，设计与实现了一种面向语言的领域语言的集成开发环境 Garden，它包括软件开发环境和程序开发环境. 软件开发环境以 GarAda 解释器为核心，用于支持领域语言开发的各过程；程序开发环境以领域语言编译器为核心，用于支持领域用户程序开发的各过程. Garden 的研制成功，为领域语言的自动生成探索了一条切实可行的途径.

关键词
Garment
Garden
领域语言
面向语言
自动生成

通用语言(general purpose languages，例如 ADA 语言、C++语言等)应用面广，适于开发各种应用程序，但是，该类语言文本繁复，用户难于精通，因而大大影响了程序开发的效率和被开发程序的效率. 针对这一现象，提出领域语言(domain-specific languages，例如文本编辑语言、数据库查询语言等)概念，成为新的研究热点. 领域语言通常支持该领域专用术语和概念，提供足够的语言成分支持该领域中问题的求解，使该领域的工作者可以简明、有效地构造领域应用程序. 领域语言还具有高效和可靠等特点，也被称为小语言[1, 2].

随着计算机技术的广泛应用，需要越来越多的领域语言. 但是传统的定义和实现语言的方法复杂繁琐、开发周期长、效率低，极大地阻碍了领域语言开发和应用. 要突破上述障碍，必须解决 2 个问题：一是从更加抽象的观点理解软件和软件的开发，创建一种新的面向语言的软件开发方法；另一方面是，要设计一种比传统模块更加抽象的机制，用于封装一个语言的定义，

实现一个面向语言的开发环境来支持领域语言的自动生成.

在上述思想的驱动下, 在 20 世纪 90 年代, 我们提出了一种系统的软件开发方法, 称为面向模型的变换型软件开发方法(model-oriented specification and transformation, 简称为 MOSAT)[3~7], 把软件的开发抽象成从规范说明出发, 经过一系列保持正确性的变换, 最终得到一个正确而有效的软件过程. 用公式来表示, 即为

规范 + 变换 = 软件.

MOSAT 方法把软件理论与语言理论的研究结合起来, 把软件组件技术和面向对象技术提升到面向语言和语言组件的层次上来, 把数据抽象概念升华到语言抽象的层次, 把抽象数据类型作为语言的一种基本成分, 把程序设计语言作为软件理论研究的一类主要对象, 把一类专用软件的规范抽象成(软件界面)语言的规范, 把这类软件的实现抽象为语言的归约变换. 在此基础上探讨语言间的继承、屏蔽和扩充关系. 从而可以根据语言抽象的原则将语言分层, 不同的用户处理不同的问题时, 可以定义不同的领域语言, 这些语言构成一个语言族.

该方法建立了一种统一的分层开发模型, 把程序开发、软件开发和软件开发环境的开发视为 3 种不同层次上的活动, 把软件开发环境的研究从程序开发环境的研究中分离出来. 它们之间的关系可以用图 1 来表示.

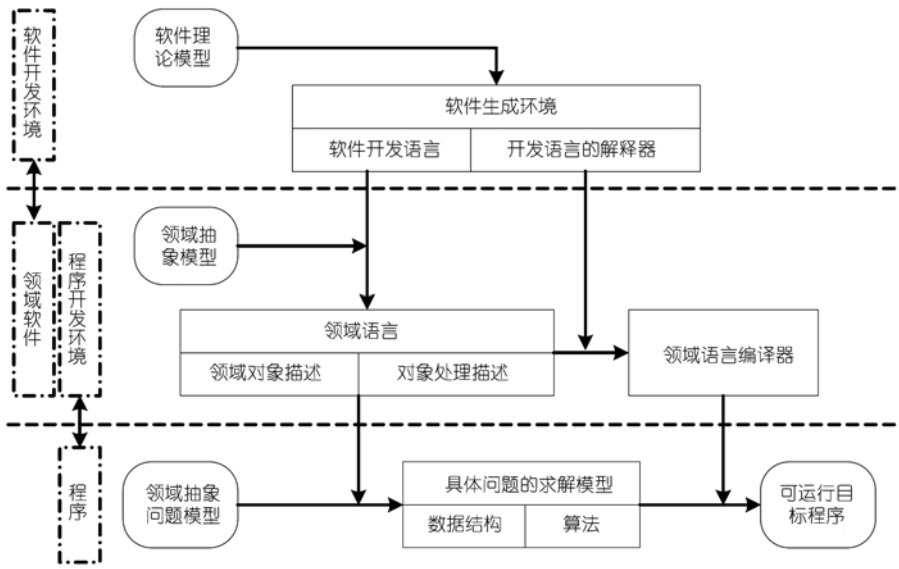

图 1 MOSAT 方法中的分层开发模型

软件开发环境[1)]的设计者从软件的理论出发, 为软件开发建立一个统一的软件生成环境, 包括为软件开发者提供一个有效的软件开发语言和这个语言的解释器. 软件开发者以领域的抽象模型为背景, 使用软件开发语言描述所要开发的领域语言, 用于描述领域对象和对象的

1) 根据文献[8]的观点, 本文区分软件开发环境和程序开发环境. 软件开发环境泛指开发包括领域软件和领域语言在内的各种软件的环境, 在本文范围内, 程序开发环境、领域语言与领域软件是同义词

处理, 通过软件开发语言的解释器生成领域语言的编译器; 程序开发者则从领域抽象问题求解模型出发, 使用该领域专用的领域语言描述具体问题的求解模型, 再通过领域语言的编译器将其转换成计算机可运行的、能够解决实际问题的目标程序.

为了实现 MOSAT 方法, 首先要研究软件生成环境中的软件开发语言, 为此, 我们提出了一种元语言, 用于描述领域软件所需要的领域语言, 这种语言的核心机制称为 Garment[8~11]. Garment 不仅能够定义领域语言中各成分的语法, 还给出了相应的语义, 可以描述语言间的继承、屏蔽和扩充等关系. 一个 Garment 规范就是一个语言的抽象与封装, 它是 Garment 语言描述的一个领域语言的完整定义.

使用 Garment 机制能够使得领域语言的开发具有层次性和系统性, 可开发出一个以目标语言(即归约到最终的可执行的语言, 如机器语言、汇编语言或者 C 语言等)为根的树形语言族, 并把它保存于语言知识库中. 程序设计者可以根据需要, 在语言族中选择适合描述其领域问题的语言进行程序设计, 或在此基础上利用 Garment 机制组装定义出一种新的更适用的语言[12~14].

本文主要介绍以 GarAda (Garment for Ada)为核心的软件开发环境(Garment development environment, 简称为 Garden)的设计与实现. 本文的结构组织如下: 第 1 节中简单介绍软件开发语言 GarAda; 第 2 节详细说明软件开发环境 Garden 的设计与实现; 第 3 节给出在 Garden 系统中已经实现的 4 种语言实例; 第 4 节是相关工作的比较; 最后, 第 5 节是结论和对下一步研究工作的打算.

1 GarAda

关于 Garment 的设计思想, 在文献[8]中有详细讨论, GarAda 是在 Garment 设计思想指导下, 具体设计的以 Ada 语言为目标的定义和开发领域语言的元语言. 限于篇幅, 本节先简单给出 GarAda 的主要框架, 然后介绍表达式和类型等组件的定义, 最后使用 GarAda 描述一个小语言 calculator 的实例. 希望读者可以从本节的介绍了解到 GarAda 的主要特征和使用方法, 详细定义可参考文献[15].

1.1 主要框架

根据面向语言的开发方法[16], 程序设计语言作为软件理论研究的一个基本对象, 一个 Garment 封装了一个语言的完整定义, 包括语法和语义信息. 另外, Garment 机制中还融入了组件化的设计思想, 针对过程型语言的需要, 在 GarAda 中预设了词法、表达式、语句等常用组件, 提供灵活的机制, 实现了对整个语言、整个组件或者各个语言部件定义的继承和扩充功能.

定义一个语言的 GarAda 规范, 首先要做的就是定义一套相应的组件. 从基本符号出发定义组件的工作是非常细致、繁琐的, 为了方便各种组件的定义和重用, 减少定义语言的工作量, GarAda 元语言提供了一些新机制. 这些机制允许定义一个语言时可以从某个已有语言继承所有组件, 然后在继承的基础上加以扩充.

下面给出的是GarAda的框架, 其中的语法描述中使用了扩充的BNF范式 [1]:

garment ::= “garment” language_name_1 [“extend” language_name_2]
[token_component]
[decl_component]
[expr_component]
[stmt_component]
[type_component]
[prog_component]
“end” [“garment”];

由以上的语法定义可以看出, 语言的 GarAda 规范由关键字 garment[2]开始, 并以 end garment(或 end)以及分号‘;’结尾, 其中包含一系列组件的定义. 而 language_name_1 是需要定义语言的名字, language_name_2 是为新语言定义提供重用组件的已有语言的名字, 也称为 language_name_1 的父语言. 父语言可以省略, 这时新定义的语言没有父语言.

各组件的具体定义既有共性, 也有个性, 细节可参见文献[14, 15]. 下面举表达式组件和类型组件为例加以说明.

1.2 表达式组件的定义

表达式组件用于定义表达式的语法和语义. 具体定义如下:

expr_component ::= “expr_component” [“include” expr_component_name] “:”
{ expr_rule }-@
“end” [“expr_component”] “;”

其中关键字 include 后面 expr_component_name 表示一个已经存在的表达式组件. 新定义的表达式组件将继承该组件的所有规则, 并在此基础上通过增加新规则来做进一步的扩充.

表达式规则 expr_rule 是用于描述领域语言程序中表达式的构造方式. expr_rule 定义如下:

expr_rule ::= [“aux”] [rule_name] local “in” syntax “return” type [“==>” trans],

其中关键字 aux 出现时表示本规则是一条辅助规则, 否则称为接口规则, 辅助规则仅仅局部于一个组件, 是定义接口规则的辅助成分, 不能用作领域语言的表达式规则独立使用; local 指示在表达式规则中的局部说明, 主要用来定义表达式的抽象语法; 关键字 in 之后的 syntax 指明所定义的表达式的具体语法; 符号==>右面的 trans 称为转换方式, 它给出==>左面定义的表达式的语义(用目标语言描述的表达式的操作语义). type 是语言定义中所有数据类型的集合, 在这里它用于说明表达式的值的数据类型, 放在关键字 return 的后面; 有关 local, syntax, type 和 trans 等的定义细节在这里省略.

下面是一个具体表达式的定义规则:

---

1) 扩展的和一般的 BNF 的主要区别在于扩展的 BNF 多了{}和[]. 意义如下:
① {α}表示由一个或多个α构成的一个表, 各部分元素之间没有间隔;
② {α}-s 的意义与{α}类似, 只是相邻的α元素之间都被字符 s 分隔;
③ [α]表示这部分内容可以缺省

2) 需要注意的是, 这里的关键字 garment 是元语言 Garment 中预定义的. 在使用 Garment 定义领域语言时, 被定义语言中的关键字将作为字符串常量, 在规则的具体语法中定义. 它们属于不同的层次

int_array_expr ident name, expr i:Int in name {“[“i”]”} return Int ==>name “(“{i}-”,“”)”.

这个名为 int_array_expr 的接口规则定义了返回整数类型的下标表达式. 抽象语法是: ident name, expr i:Int, 其中标识符 name 是该数组的名字; i 是整数类型的表达式. 具体语法是: name {“[“i”]”}, 其中花括号{}的使用表明该数组表达式可以是多维的. 这里变换前后{}中的内容有所不同: 领域语言程序中每一维下标都以方括号[ ]括起, 而在 Ada95 语言里, 所有下标共处于一个圆括号( )中, 下标之间用逗号隔开; 更加重要的是: 由于“==>”左边的 i 可以是任意的整数表达式, 所以“==>”右边的 i 代表的应该是使用有关规则对 i 进行变换的结果.

规则是组件定义中最重要的部分, 它的作用是为组件定义新的语言成分. 上面给出的例子是最简单的情形. 对于结构比较复杂的语言成分, 其变换规则也比较复杂, 为了提高规则的描述能力, 在 GarAda 元语言中引入了结构指代符和分块指代符等. 具体介绍将在后面给出.

1.3 类型组件的定义

类型组件 type_component 是一种复合组件, 它定义了领域语言中类型系统, 包括各个类型的表示方法以及相关的操作. 具体定义如下:

```
type_component ::= “type_component” [“include” type_component_name]“:”
                        {type_def }-%
                   “end” [“type_component”]“;”
```

类型组件是以关键字 type_component 和 end(以及可选的 type_component)作为首尾, 其间是由一系列的类型定义 type_def 组成的, 类型定义之间以%为分隔符. 每个 type_def 描述对一个类型的定义和实现, 其中包括对类型定义的字面量的定义, 该类型的各种操作定义以及相应的实现细节. 通过“include” type_component_name, 可以重用已有的类型组件.

类型组件与其他组件的关系很紧密, 每个类型定义实际上也定义了与类型有关的一些词法元素、表达式、语句以及必要的函数和过程. 把它们封装在一个类型定义中, 目的是便于类型的重用.

type_def 的定义如下:

```
type_def ::= “type”   abs_type_name
                “repr”   rep_type
          [ “literals:”          /* 该类型的字面量*/
                token_rules   ]
          [ “operators:”       /* 该类型定义的算子*/
                op_rules      ]
          [ “expressions:”     /* 该类型定义的表达式*/
                expr_rules    ]
          [ “functions:”       /* 该类型定义的函数*/
                func_rules    ]
          [ “procedures:”     /* 该类型定义的过程*/
                proc_rules    ]
     “end” [ abs_type_name ]
```

从上面的介绍可以看出, 在 GarAda 中支持 3 种层次的重用方法[17, 18]:

- 语言的完整定义作为最高层的可重用部件, 即重用一个 GarAda 规范.
- 语言的一类语法成分的定义作为中层的可重用部件, 例如重用语言中所有表达式.
- 语言的具体语法成分的定义作为一个变换程序模块低层的可重用部件, 例如语言中一个具体语句.

在下一节中, 读者将看到重用一个 GarAda 规范的例子.

1.4 GarAda 规范举例

下面这个例子中, 计算器语言支持整数的加法、减法和乘法计算, 并且支持正负数及括号表达式, 语句以“go”开始引导. 以下是用 GarAda 描述的该计算器语言的规范的例子:

```
garment CALCULATOR
  token_component:
      aux digit    {0..9} x in x
    @ num          digit x in {x}
    @ sign         {+,-} s in s
  end token_component;
  type_component:
    type cal repr Integer
      literals:
        cal_literal    num x, sign s in [s]x
      operators:
          “+”    i:cal, j:cal return cal    ==>    i “+” j
        @   “–”    i:cal, j:cal return cal    ==>    i “–” j
        @   “*”    i:cal, j:cal return cal    ==>    i “*” j
      expressions:
          signed_expr    expr r:cal, sign s in s r return cal ==> s r
        @ bracket_expr    expr r:cal in “(” r “)” return cal ==> “(“ r ”)”
    end
  end type_component;
  prog_component:
    program          expr r : cal in “go” r
                     ==>
                     “with Ada. Text_IO;”
                        “use Ada. Text_IO;”
                        “procedure Calcu is”
                        “package Int_IO is new Integer_IO(Integer);”
                        “use Int_IO;”
                        “begin”
                        “put(” r “);”
                        “end Calcu;”
```

```
    end prog_component;
end garment;
```

在上面的这个例子中, 语言 CALCULATOR 的 GarAda 规范中包含了 3 个组件, 它们分别是词法组件、类型组件和程序组件. 词法组件中定义了该语言的基本词法元素“数”(num, 由数字串组成)和“符号”(sign, 即正负号). 而类型组件中则定义了语言 CALCULATOR 的唯一类型“cal”, 并具体定义了其表示方法和操作等内容. 最后程序组件给出了该语言程序之结构及其转换至目标语言的规则.

下面我们运用 Garment 提供的最高层次重用机制, 对该语言进行继承和扩充.

```
garment CALCULATOR_1 extend CALCULATOR
    type_component:
        type cal_1 repr Float
            literals:
                cal_1_literal    num whole fraction, sign s in [s]whole“. ”fraction
            operators:
                “/”    i:cal, j:cal return cal_1 ==> “Real(” i “)” “/” “Real(” j “)”
             @ …
            …
        end
    end type_component;
    prog_component:
        program_1    expr r : cal_1 in “go” r    ==>  …
    end prog_component;
end garment;
```

新定义的语言 CALCULATOR_1 完全继承了父语言的所有组件, 并在类型组件中增加了一个类型 cal_1, 它表示实数类型. 相应地, 又为这个新类型在程序组件中添加了规则 program_1, 它处理的是 cal_1 类型的表达式. 需要注意的是: 我们在定义该类型的字面量 cal_1_literal 时, 使用了规则 num, 却没有给出 num 的定义. 这是因为 num 已经在父语言的词法组件中做了定义. 由于继承的关系, 我们可以直接使用 num 规则, 不需要在新语言的定义中再做说明.

2 Garden

Garden 是以 GarAda 为软件开发语言, 设计与实现的一种面向语言的领域语言的集成开发环境[19]. 下面首先介绍它的总体结构, 然后分别介绍 Garden 中的几个主要功能模块实现思想.

2.1 系统概述

Garden 系统提供了比较简单, 但功能较完全的一个集成开发环境. 它支持 2 种运行模式: 领域语言开发模式和领域程序开发模式. 语言知识库在系统中起到支持和连接这 2 种开发模式的作用.

2.1.1 领域语言开发模式

领域语言开发模式支持用户在开发领域语言工作中的各种活动, 包括领域语言定义的输入、编辑、修改、处理和存储等, 以及在开发新的领域语言时对语言库里已有语言、部件的重用.

领域语言定义的输入、编辑、修改用一个常规的正文编辑器进行, 领域语言定义的处理通过 GarAda 解释器完成. 在对一个领域语言定义的处理中, 可能发现其中存在错误与不完全之处. 这时需要用户对定义进行修改, 然后重新处理. GarAda 解释器在处理中检查领域语言的词法、语法和语义的一致性问题.

领域语言开发模式完成领域语言的开发工作, 把正确定义的领域语言的完整的 GarAda 规范和构造领域语言编译器的基本(语法和语义)信息保存在语言知识库中. 用 L1 表示要开发的领域语言, 其开发流程见图 2 所示.

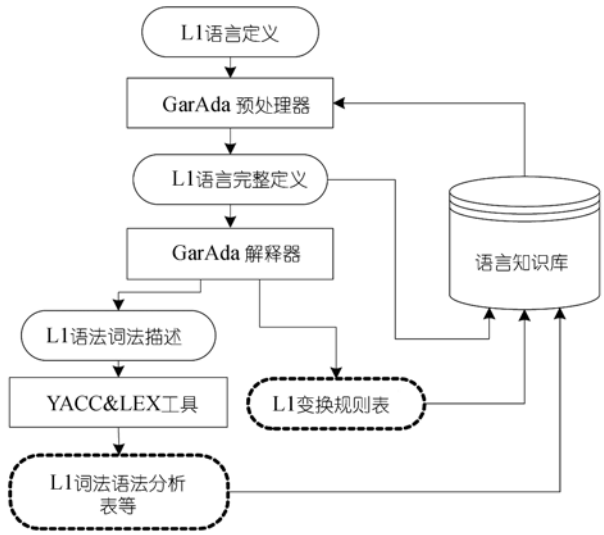

图 2 语言开发流程图

为了支持领域语言开发模式, 在系统中实现了 Garment 预处理器、Garment 解释器和改进的 YACC 和 LEX 工具.

2.1.2 领域程序开发模式

领域程序开发模式支持用户利用系统中已经定义好的领域语言, 开发领域具体问题的应用程序. 用户在开发一个领域程序时, 需要从语言库中调出相应语言的语法和语义信息, 建立特定语言的程序设计环境. 用户进行程序开发的工作包括: 领域语言选择; 应用程序的编辑修改; 应用程序的加工处理, 包括词法、语法检查和到 Ada 程序的变换, 以及需要时的格式化打印, 最后得到需要的格式化的 Ada 程序.

领域程序开发流程见图 3 所示. 为了支持领域程序开发模式, 系统中提供了通用语言处理器、通用程序变换器和格式化输出程序.

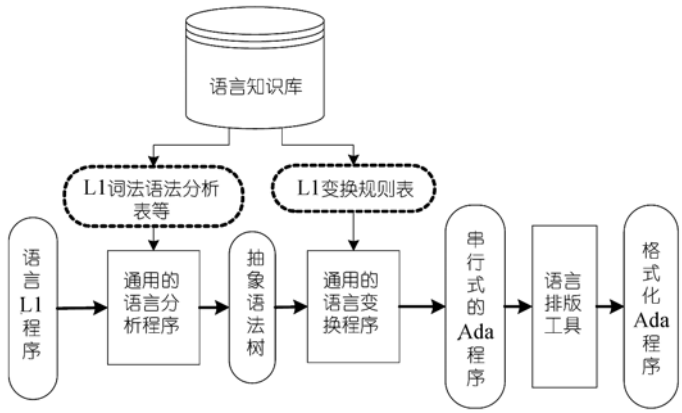

图 3　程序开发流程图

2.1.3　语言知识库

领域语言的定义信息有 2 种不同的存储方式: 第 1 种是在开发该语言时, 以 GarAda 规范的形式存储为普通的正文文件; 第 2 种是在完成领域语言的开发后, 把定义好的领域语言的完整 GarAda 规范及其编译器需要的全部语法和语义信息存储到语言知识库里, 统一进行管理, 供用户在此后开发领域程序时, 或者在定义新的领域语言时使用.

Garden 系统维护一个语言知识库, 包括一个语言表、一个语言引用表和入库语言信息, 并且提供了基本的管理和保护功能.

2.2　GarAda 预处理器

由于 GarAda 支持语言及其组件的继承和扩充, 所以在 Garden 中设计了一个 GarAda 预处理器. 其主要功能是将领域语言的 GarAda 规范与该语言的父语言的 GarAda 规范归并, 形成一个完整的领域语言 GarAda 规范, 然后提供给 GarAda 解释器处理.

处理语言或组件继承就是当扫描到“extend”或者“include”关键字时, 再往后读取要继承的语言或组件名, 组件名应该包含有语言名为其前缀, 根据这个语言名到语言库中可以得到该语言的 GarAda 规范的原始文件(.gmt 文件), 再从该文件中提取对应的组件的定义, 插入结果文件中.

在定义领域语言的 GarAda 规范中, 支持类似 C 语言的注释, GarAda 预处理器也负责注释的过滤.

2.3　GarAda 解释器

GarAda 解释器是根据前面介绍的软件开发语言 GarAda 的定义设计的, 它能够对经过预处理器加工的领域语言的 GarAda 规范进行词法和语法分析, 并从中提取领域语言的词法、语法和语义(变换)的完整信息, 把这些信息保存在语言知识库中, 这些信息与通用语言处理器、通用程序变换器结合, 自动构造出专用领域语言的编译器.

假设被开发的语言是 L1, GarAda 解释器首先对 L1 的 GarAda 规范进行词法和语法分析,

如果存在词法或语法错误, GarAda 解释器报告存在的基本错误类型和位置, 以便用户进行修改. 对语法正确的 GarAda 规范, 根据变换规则, 解释器提取语言的语义信息. 换句话说, GarAda 解释器包括 2 个小模块: 一个负责对 GarAda 规范进行词法与语法的分析; 另一个负责对 GarAda 规范进行语义的提取. 前一个模块构造出领域语言 L1 合乎 YACC&LEX 规定的词法和语法分析表. 后一个模块构造出能够为通用的语言变换程序所需要的变换规则表.

下面就 GarAda 解释器实现中几个关键的问题进行说明.

2.3.1 产生式改写

由于 GarAda 规范也是采用扩展的 BNF 来描述领域语言的语法, 而 YACC 的输入文件采用一般的 BNF 来描述, 因此 GarAda 的一个工作就是把用扩展 BNF 描述的产生式改写成用一般 BNF 描述的产生式.

2.3.2 类型处理

GarAda 元语言允许领域语言定义自己的类型, 因此在处理领域语言的 GarAda 规范的时候, 就需要进行类型检查. 这里需要说明的是这里只是对领域语言的 Garment 规范进行类型检查, 对领域语言编写的程序的类型检查由通用领域语言处理器去处理.

2.3.3 复杂规则的处理

在 1.2 小节中提到, 为了提高规则的描述能力, 在规则中引入了结构指代符和分块指代符等特殊控制符.

下面先看一个简单的例子, 在使用 GarAda 描述并行语言 UNITY 时, 对标识符使用了下面的变换规则:

```
id      letter x, letter_digit y in x[{y}]
                    ==> #1: x[{y}]
                        #2: “L_”x[{y}]
                        #3: “FP_”x[{y}]
```

==>左边的 x[{y}]说明标识符是由字母(用 x 表示)开始, 其后是 0 个或多个字母或数字(用 y 表示)的序列; 而==>右边则将标识符转换为 3 个部分(分别用分块指代符#1, #2 和#3 表示): 标识符保持不变、给标识符加前缀 L_以及给标识符加前缀 FP_. 这样做是为了处理多重赋值和不动点问题. 然后在这个规则的基础上, 可以对整型数组的声明定义如下规则:

```
int_decl    id a, positive_num i
                in $1{a}-“,” “:” “array” “[” $2{i}-“, ” “]” “of ” “integer”
                ==>$1{a. #1“,”a. #2“,”a. #3“: array(”$2{“1.. ”i}-“,” “)” “of Integer;”}
```

其中$1 和$2 是结构指代符. 通过规则的转换, 源程序中的每一个数组变量声明会变成 Ada95 中的 3 个数组变量声明. 例如: s1,s2,s3:array[2,3]of integer 转换为

```
s1,L_s1,FP_s1:array(1. . 2,1. . 3)of Integer;
s2,L_s2,FP_s3:array(1. . 2,1. . 3)of Integer;
```

s3,L_s3,FP_s3:array(1. . 2,1. . 3)of Integer.

由于规则的匹配和替换工作都是在抽象语法树上进行, 所以在 GarAda 解释器中, 对于结构指代符和分块指代符的处理, 主要是在规则的内部表示中记录相关的依赖关系, 给目标语言程序的生成提供正确的信息. 例如, 对于分块指代符, 在规则的内部表示中, 主要增加了一组指针指向不同块目标子树结点的位置, 以便于高层规则需要使用时, 可以很快找到指定的目标块; 对于结构指代符, 在对应语法树的结点中增加一个整数变量, 记录该结构分量重复的次数.

2.4 LEX 和 YACC 工具

有**两**个方面的原因需要利用语法分析程序自动生成工具 YACC 和词法分析程序自动生成工具 LEX. 一方面是利用功能强大的辅助工具, 可简化领域语言分析程序的构造过程; 更重要的原因是, 在 Garden 中要求领域语言的编译程序是自动生成的. 系统中设计的通用的语言分析和通用的语言变换程序需要统一格式的数据来驱动, 才能构成通用的编译系统.

由于本系统的具体要求, 使用前先对标准的 YACC 和 LEX 工具进行一定的改造, 当用它们分析完领域语言的词法和语法文件后, 生成的词法、语法分析表不再输出到文件中, 而是直接以通用语言分析程序能够接收的格式保存起来, 使得通用语言分析程序可以直接使用.

2.5 通用的语言分析程序

通用的语言分析程序是一个通用的表格驱动的语言处理器. 它使用一个词法描述表和一个语法描述表作为工作的基础, 实施对领域语言程序的处理. 这个程序的功能与一般编译程序的前端类似, 包括对领域语言程序的词法和语法分析检查, 作为处理结果产生的是用户程序的抽象语法树, 树中还要包含一些为方便后续工作而用的信息.

通用的语言分析程序除了需要词法和语法分析表以外, 还需要符号表. 符号表是系统在载入领域语言的时候, 从该领域语言的相关文件中读取的.

2.6 通用的语言变换程序

通用的语言变换程序需要依靠语言开发模块生成的具体领域语言的变换规则集, 将领域语言程序的抽象语法树变换成对应的 Ada 程序的语法树.

程序变换器采用后序周游的方式扫描语法树, 进行匹配和置换. 对周游到的每一个结点用变换规则表中的规则调用“树匹配函数”逐一进行匹配. 如果匹配成功, 则根据该规则调用“替换函数”构造新的语法子树, 并将构造出来的子树替代所匹配的子树. 这样当把所有结点都扫描一遍后, 新的语法树就完全替代了原来的语法树[4]. 在变换过程中还要考虑包括结构指代符和分块指代符等特殊问题的处理, 其细节这里从略.

2.7 格式化输出

Ada 语言排版工具负责将变换结果的 Ada 程序格式化输出. 由于变换程序变换出来的 Ada 源程序并没有考虑到可读性因素, 只是按照内部语法树表示打印成一行不间断的文本, 因此可读性很差. 所以需要将这个文件经过格式化处理, 生成具有一定格式的、可读性强的 Ada 程序.

具体实现时采用关键字驱动的方式. 注意到每一特殊结构至少对应一个关键字, 这使通

过处理关键字而得到符合要求的 Ada 文件成为可能.

3 应用实例

在 Garden 系统中, 使用 GarAda 元语言成功地开发了几个典型的小语言.

3.1 基于 Ada95 的软件规格说明元语言

Ada 语言是美国国防部(DoD)支持开发的一个大型通用程序设计语言. Ada95 是在十多年来广泛应用的基础之上, 经过认真讨论和修改, 特别是扩充了面向对象等软件工程的最新技术后, 推出的 Ada 最新版本. 被许多国家(包括中国)指定为军方软件的标准.

但是, Ada 语言的功能十分全面和强大, 语言本身的复杂程度也很高. 要许多程序员全面掌握 Ada 语言不但非常困难, 也没有这个必要. 为了能够直接支持不同领域特点的问题进行编程, 并且最后得到指定的 Ada 语言程序, 我们立项研究“基于 Ada95 的软件规格说明元语言”, 称之为 GarAda.

使用 GarAda 定义的(基于 Ada95 的面向领域软件的)规格说明语言可以使领域程序员突破一般通用语言所迫使他们进行的面向计算机的思维方式, 代之以用接近问题空间的, 更简明的领域语言进行思维和设计. 系统可以自动把使用面向领域的软件规格说明语言描述的程序, 变换成使用 Ada95 语言描述的程序. 使用这种方式进行应用系统开发, 不但可以提高开发的效率, 同时也能提高开发的质量[15,19].

本系统 2000 年经使用单位测试合格, 通过专家正式验收 1).

3.2 控制决策语言 VERT 及其程序到 Ada 语言程序的转换系统

VERT 是一种网络分析方法, 可以用于风险决策. 我们与海军某部柳克俊教授共同合作, 抽取出一个 VERT 语言, 用 GarAda95 实现了从 VERT 程序到 Ada95 程序的转换[20,21]. 使用户可以方便地描述决策中所需要模拟的过程. 从而大大减轻程序员的工作量和工作难度, 极大地提高工作效率. 该系统使用后, 反映良好[22].

3.3 并行语言 UNITY-BD 及其程序到Ada 语言程序的转换系统

UNITY[15]是一种并行程序设计的理论模型, 其中包含一个并行计算语言以及相应的证明系统. BD-UNITY 是北京大学计算机系袁崇义和屈婉玲教授在标准 UNITY 语言的基础上发展起来的. 它不仅保留了标准 UNITY 的表达能力和逻辑系统, 而且还对标准 UNITY 的一些不足之处做出了改进. 我们用 Garden 实现了 BD-UNITY 语言[15].

3.4 C 语言子集

C 语言作为应用广泛的通用语言之一, 在嵌入式系统程序的开发中也起着非常重要的作用. 在 Garden 系统实现中, 用 GarAda 规范描述了 C 语言的复合语句, 以复合语句块为基础, 定义并且实现了 C 语言的一个较大的子集.

1) 2000 年 12 月 19 日, 北京大学承担的“ADA 软件质量评测系统”在有关单位主持下进行验收. 验收组认为该课题完成情况良好, 一致同意通过该课题的验收

4 相关工作比较

从 20 世纪 90 年代后期开始, 面向语言模型的研究一直是软件工程领域中的研究重点之一. 荷兰 CWI 的研究者提出的 ASF+SDF 模型[23,24], 使用代数方法刻画了 SDF2 用户描述语言, 构建了基于 SDF2 的元语言环境, 提出了对“语言块”和“语言成分”进行封装的想法. 瑞士的 TIK 和意大利的 L'Aquila 大学合作的 Montages 项目[25,26], 为领域语言的设计和快速原型化提供一个环境. 用户可以通过一个框架对领域语言的语法和语义进行形式化的描述. 德国 Rostock 大学的 LDL 项目[27]研究目标是以语言为中心的软件领域提供一个形式化的开发环境, 他们采用属性文法作为语义的描述方法. 德国 Paderborn 大学主持开发的 Jacob 项目[28,29]的目标是设计一个新的领域语言开发环境, 用户可以通过组装和配置语言部件的方法开发语言. 语言部件由领域专家提供, 而语言设计者将领域专家对部件的描述转换成为系统的一个可接受的形式, 并且将各种部件按领域专家的制定, 做正确的组合, 生成目标语言.

意图编程(intentional programming)[30,31]是微软剑桥研究院启动的项目, 它是一种用意图(intention)的观点来描述程序的方法. 通过选定一个基语言, 新语言的意图可以通过对基语言的扩充定义出来, 意图的语义用转换规则来描述, 转换规则描述的是如何将新的意图转换成为基语言可接受的形式. 最近备受关注的LOP (language oriented programming)[1)]可以认为是围绕DSL建立软件模型和开发的一种新思想和新风格. LOP认为: “编程”不是写一组计算机指令, 而应该是对某个问题的清晰无误的表达; “编程”不能仅仅限制在文本中, 程序不应该天生就是文本; Sergey Dmitriev将LOP编程语言分为 3 个部分: 结构、编辑器和语义; Language Workbench是进行LOP的工具, 使用Workbench的目的是使DSL实现变得容易. 根据这一理解, Intentional Software的IP、JetBrains的Meta-Programming System和微软的Software Factories都属于是一个具体的Language Workbench[2)].

面向语言的可信模型是最近国际关注的焦点. 图灵奖获得者 Tony Hoare 提出的挑战性的课题: 可验证编译器 [3,4)], 刚刚处于起步阶段, 就吸引了如微软、惠普等国际著名公司的注意.

国内也有不少科研单位对面向语言的模型进行了初步的研究. 如北京航空航天大学李未院士领导的研究小组, 以及国防科学技术大学陈火旺院士领导的研究小组, 近些年来一直对软件的可靠性作了不少有益的探索, 取得了不少重要的成果[32,33]. 中国科学院计算所张兆庆教授领导的课题组定义了一个可扩展的编译体系结构——X体系结构, 目标是支持用户扩展语言机制、编码领域抽象和优化技巧[34].

北京大学张乃孝负责的课题组从20世纪90年代开展了领域语言开发方法的研究, 并取得了许多成果. 本文的工作是该课题组长期共同研究的一部分成果(与本文相关的另外一部分成

1) Dmitriev S. Language Oriented Programming: The Next Programming Paradigm. http://www.onboard.jetbrains.com/is1/articles/04/10/lop/. 2004, 11

2) Fowler M. Language Workbenches: The Killer-App for Domain Specific Languages? http://martinfowler.com/ articles/languageWorkbench.html. 2005, 6

3) Hoare T. The ideal of program correctness. http://www.jaist.ac.jp/~bjorner/ae-is-budapest/talks/Sept19am1_Hoare.ppt. 2006, 9

4) Bicarregui J C, Hoare C A R, Woodcock J C P. The verified software repository: a step towards the verifying compiler. http://epubs.cclrc.ac.uk/bitstream/671/repository-29-mar-05.pdf

果——面向语言开发方法与语言演算的理论模型[3~17]目前正在整理). 与国内外同行的工作相比较, 本工作有以下特点: 起步比较早, 在研究初期就强调从科学的方法论出发, 理解软件和软件开发的过程, 严格区分程序和软件, 指出软件开发环境与程序设计环境的区别; 从(广义的)计算科学高度建立了一种统一的分层开发模式; 把语言理论与软件理论结合起来, 以语言作为软件理论研究的一种基本对象(第一类值), 定义了语言的抽象与封装机制; 从理论上研究了语言的演算系统. 本项研究的最终目标, 希望不但可以降低程序设计和软件开发的复杂度, 从根本上解决软件危机问题; 同时还希望建立科学的面向语言的方法论, 对于软件的正确性和编译系统的正确性研究提供有效的支持.

5 结论与下一步工作

本文在面向模型的变换型软件开发方法和语言的抽象与封装机制 Garment 研究的基础上, 设计与实现了一种面向语言的领域语言的集成开发环境 Garden, 包括软件开发环境和程序开发环境. 软件开发环境以 GarAda 解释器为核心, 用于支持领域语言开发的各过程; 程序开发环境以领域语言编译器为核心, 用于支持领域用户程序开发的各过程. Garden 的研制成功, 为领域语言的自动生成探索了一条切实可行的途径.

本文的主要贡献在于:

1) 发展了 MOSAT 的思想, 明确提出对于软件的理解: 规范 + 变换 = 软件;
2) 设计与实现了面向过程的元语言 GarAda 解释器;
3) 实现了通用的表格驱动的语言编译器;
4) 实现了语言知识库的基本管理功能;
5) 成功利用 Garden 开发了多个小语言.

今后的主要工作包括:

1) 设计与实现面向其他语言范型(例如说明型)的元语言;
2) 设计与实现基于其他目标语言(例如汇编)的开发环境;
3) 提高语义变换规则描述的抽象度(例如从操作语义提高到指称或者/和代数语义);
4) 设计与实现语言组件的多态性;
5) 深入研究面向语言方法学的理论基础.

致谢 感谢陆汝钤、杨芙清、李未、何积丰、何新贵等院士和 David Gries、Dines Bjorner、孙怀民、张兆庆等教授的慷慨支持和帮助, 无论是研究的思想, 还是具体研究方法, 以至许多论文的表达, 他们都给予作者许多有益的建议.

参考文献

1 Bentley J L. Programming pearls: little languages. Commun ACM, 1986, 29(8): 711—721 [DOI]
2 Goldrei S E C. The design, implementation and use of domain specific languages. Sydney: School of Information Technologies, University of Sydney, 2004

3 张乃孝, 许卓群, 屈婉玲. 面向模型的变换型软件开发方法研究. 理论计算机科学, 1994, 2: 54—64
4 张乃孝. 程序变换过程的分析与设计. 计算机学报, 1994, 17(6): 473—476
5 张乃孝. 程序变换在程序语言中的一种表示——兼论变换型语言. 软件学报, 1993, 4(5): 17—23
6 屈婉玲, 张乃孝. 用变换型方法模拟开发电话交换系统. 计算机研究与发展, 1995, 32(7): 11—16
7 张乃孝, 郑红军, 裘宗燕. 语言的抽象、封装与变换型开发方法. 软件学报, 1998, 9(7): 496—500
8 Zhang N X, Zheng H J, Qiu Z Y. Garment—a mechanism for abstraction and encapsulation of languages. ACM Sigplan Notices, 1997, 32(6): 53—60 [DOI]
9 张乃孝, 郑红军. 程序设计语言的抽象与语言族模型. 北京大学学报(自然科学版), 1997, 33(5): 650—657
10 郑红军, 张乃孝. 一种带约束的多态类型系统. 计算机学报, 1999, 22(4): 343—350
11 郑红军, 张乃孝. Garment 中的归约语义. 计算机研究与发展, 1998, 35(6): 486—490
12 Liu Y, Zhang N X. Defining domain-specific languages using polymorphic and orthogonal machanisms. In: Proceedings of SCI2004 (the 8th World Multi-conference on Systematics, Cybernetics and Informatics). Orlando, 2004
13 Liu Y, Zhang N X. On concept-based definition of domain-specific languages. In: the 4th International Conference on Formal Engineering Methods (ICFEM2002). LNCS. Shanghai: Springer-verlag, 2002. 237—248
14 Liu Y, Zhang N X. Developing domain-specifc languages in concept-based development method. In: the 6th IASTED International Conference on Software Engineering and Applications(SEA2002). Cambridge, 2002
15 和华. Garment 定义语言方法及实例研究. 硕士学位论文. 北京: 北京大学, 2000
16 范少锋. 面向语言的领域语言开发方法研究. 博士学位论文. 北京: 北京大学, 2006
17 Qiu Z Y, Zhang N X, Wang M H. Software reuse in the Garment approach. In: Proceedings of the Interational Symposium on Future Software Technology. ISFST-98 P. Japan: Software Engineers Association, 1998. 323—326
18 Zhang N X, Liu Y. A component-based framework and reusability in Garment. In: Proceedings of the Asia-Pacific Software Engineering Conference. Macau: IEEE Computer Society, 2001. 411—418
19 陈斌. GarAda 系统的设计与实现. 硕士学位论文. 北京: 北京大学, 2001
20 张培刚. 领域语言 VERT 的设计及变换型实现. 硕士学位论文. 北京: 北京大学, 1999
21 陈旭盛. VERT 领域语言及其到 Ada95 的转换. 硕士学位论文. 北京: 北京大学, 1999
22 柳伟明, 张培刚, 陈旭盛. 领域语言开发环境 GarAda95 在风险决策管理中的应用. 计算机科学, 1999, 26(10): 134—135
23 Brand M, Heering J, Klint P, et al. Compiling rewrite system: the ASF+SDF compiler. ACM T Progr Lang Sys, 2002, 24: 334—368 [DOI]
24 Brand M, Deursen A, Heering J, et al. The ASF+SDF meta-environment: a component-based language development environment. Compiler Construction. Berlin/Heidelberg: Springer, 2001. 365—370
25 Anlauff M, Kutter P, Pierantonio A. Formal aspects of and development environments for montages. In: Proceedings of the 2nd International Workshop on the Theory and Practice of Algebraic Specifications, Electronic Workshop in Compwting. Amsterdam, 1997. 25—26
26 Anlauff M, Kutter P, Pierantonio A. Montages/GemMex: A Meta Visual Programming Generator. TIK-Report 35, ETH. 1998
27 Lammel R. An interactive language library. In: Procceedings PEPM'99, ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation. BRICS Notes Series NS-99-1, 1999
28 Kastens U, Pfahler P. Compositional design and implementation of domain-specific languages. In: Proceedings of the IFIP Working Conference on Systems Implementation'00: Languages, Methods and Tools. London: Chapman & Hall, 1998. 152—165
29 Pfahler P, Kastens U. Language design and implementation by selection. In: Proceedings of the 1st ACM-SIGPLAN Workshop on Domain-Specific-Languages'97. Paris, 1997. 97—108
30 Simonyi, Charles. The Death of Computer Languages-The Birth of Intentional Programming. Microsoft Research Technical Report MSR-TR-95-52. 1995. 25
31 Simonyi C. Intentional Programming-Innovation in the Legacy Age. Presentation at IFIP WG 2. 1 on Algorithmic Languages and Calculi. 1996. 26
32 陈火旺, 王戟, 董威. 高可信软件工程技术. 电子学报, 2003, 31(12A): 1933—1938
33 孙昌爱, 金茂忠. 软件体系结构描述研究与进展. 计算机科学, 2003, 30(2): 136—139,167
34 胡伟平, 张兆庆, 乔如良. 支持语言扩展的编译基础结构——X 体系结构的研究. 计算机学报, 1999, 22(4): 403—408