第39卷 第2期 计 算 机 科 学 Vol. 39 No. 2
2012年2月 Computer Science Feb 2012

基于程序分析的代码查询技术

曾 锃 赵建华
(南京大学计算机科学与技术系 南京 210093)

摘 要 提出了一种基于程序分析的代码查询技术,它能有效地应用于代码审查、程序自动插桩等常用的软件工程的研究场景。它通过代码静态分析获得程序元素信息,并将其保存为中间结构,作为代码查询过程的目标集合;查询过程以程序元素为目标,查询语言以谓词逻辑表达式的形式描述查询条件。基于此技术,实现了一个面向C/C++语言的代码查询工具。

关键词 代码查询技术,查询语言,静态分析
中图法分类号 TP311.5 **文献标识码** A

Code Query Technology Based on Program Analysis

ZENG Zeng ZHAO Jian-hua
(Department of Computer Science and Technology, Nanjing University, Nanjing 210093, China)

Abstract A new code query technology based on program analysis was presented. It gives aid in several scenarios of software engineering and research, such as code review, program automated instrumentation and so on. By means of static analysis, program elements is gathered by an extractor, which is responsible for mapping sources to an intermediate structure used in process of query. Query language proposed in this paper depicts query condition in the formalism of predicate logic expression. On the basis of this technology, a tool for querying code in C/C++ language was implemented as an Eclipse plug-in.

Keywords Code query technology, Query language, Static analysis

1 引言

代码查询技术在软件分析和测试工作中发挥着重要的作用,在软件架构分析[1]、逆向工程[1,2]、一致性验证[3]、代码审查[4,5]、程序插桩等方面[6]都有广泛的应用。

代码审查是软件工程化实践中不可忽视的环节,是提高软件安全性、可靠性的重要技术。在这个过程中,审查人员通过人工审查或者自动化的工具,发现代码中存在的潜在缺陷、检查程序是否遵守一些编码标准等。在自动化的代码审查过程中,找出不遵守编码标准的代码是工作的关键。代码查询技术可以用来解决这一问题,因为它可以在源代码中找出满足特定查询条件的代码片段。

在动态测试中,插桩[7,8]是一个很重要的测试手段,有广泛的应用。它在被测试程序的特定位置(称作探测点)中插入一些"桩"代码,这些"桩"代码(称作探测器)通常用来对探测点附近程序语句的执行、变量的变化等情况进行检查。然后测试者运行插桩后的程序,当运行到探测点时测试所要搜集的信息会被自动记录下来。在自动插桩过程中,探测点位置的查找和定位是一个关键步骤。代码查询技术应用到插桩工作中,就可以根据测试人员不同的需求,在程序中找到满足特定要求的代码位置并设置插桩点。

本文提出了一种基于程序分析的、以程序元素为目标的代码查询技术,并设计了一个基于谓词逻辑表达式的查询语言,用来描述用户的查询需求。基于此技术,本文实现了一个面向C/C++语言的原型工具C2Parser,并以Eclipse插件的形式发布。

本文第2节介绍研究工作的整体框架;第3节描述文中工具支持的查询语言;第4、5节介绍工具的两个重要模块,其中第4节是信息抽取模块:工具从源代码中抽取程序信息并保存到中间结构的过程;第5节是目标检索模块:对用户输入的查询命令进行解析并从上一模块保存的中间结构中检索出满足查询条件的目标元素的过程;第6节给出工具的实际应用场景;第7节是对相关工作的介绍和比较;最后是结束语和未来工作的展望。

2 总体框架

C2Parser是基于本文中提出的代码查询技术实现的原型工具。它以C/C++工程源代码和描述用户查询条件的查询命令S为输入,以满足查询条件的目标元素集合为输出。其大致工作流程如图1所示。首先,工具借助于CDT对工程源代码进行静态分析,从中收集出程序结构元素的信息集EC。然后,工具接收用户输入的查询命令S并进行解析,从中提

到稿日期:2011-03-15 返修日期:2011-05-25 本文受核高基项目(2009ZX01036-001-001)资助。

曾 锃(1987-),男,硕士生,主要研究方向为代码分析与验证,E-mail:zz913@seg.nju.edu.cn;赵建华(1971-),男,博士,教授,博士生导师,主要研究方向为软件工程、程序分析、形式化方法。

取出针对用户需求的查询条件。接着，工具在程序元素集 *EC* 中检索出满足查询条件的程序元素，并将结果元素集合作为输出返回给用户。

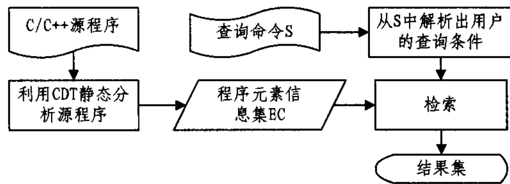

图 1 C2Parser 的工作过程

3 查询语言

为了能准确地描述用户的查询需求，本文设计了一种查询语言。它要求用户在查询命令中指明查询目标的程序元素类型(目前支持的类型有类、方法、语句和变量)以及目标元素应该满足的查询条件。用户在对查询条件的表述中还可能需要引入其他程序元素来帮助描述目标元素的查询条件，目前工具支持的查询条件包括两类：属性条件和关系条件，并可以通过逻辑连词(&&、||、!)和量词(exist、all)对多个条件进行组合。接下来将详细介绍查询语言的语法以及元素的属性和元素间关系的定义。

3.1 二级标题语言的语法

用户输入的查询命令 *S* 应该遵循的语法规则如图 2 所示。

```
S : = find Id:T satisfying CS
T : = variable
      | statement
      | function
      | class
CS : = {exist Id:T}{all Id:T} CE
CE : = CE && CE
      | CE || CE
      | ! CE
      |(CE)
      | Id. Att='value'
      | Id Rel Id
Att : = name
      | dataType
      | specificType
      | returnType
      | paramsType
Rel : = extend
      | use
      | change
      | isIn
      | call
```

图 2 查询语言的语法

查询命令 *S* 以关键字 find 开始，后跟一个 *Id*:*T* 的声明格式，再接 satisfying 关键字，最后以条件语句 *CS* 结尾。它的含义是要查询满足查询条件 *CS* 的 *T* 类型的程序元素。

语法规则中的 *T* 是指程序元素的类型，它可以是 variable、statement、function 和 class 的其中之一，其分别对应于变量、语句、方法、类。*Id*:*T* 声明了一个类型为 *T*、别名为 *Id* 的元素。*Id* 作为它声明元素的别名在之后的查询条件表达式 *CE* 中使用。在条件表达式中出现的 *Id* 必须先被声明，并且同名的 *Id* 不能被重复声明，以免混淆。紧跟在关键字 find 之后声明的 *Id* 就对应于查询目标元素。

条件语句 *CS* 描述的是目标元素需要满足的查询条件。它的语法规则在形式上符合带量词的一阶谓词公式，具体含义是：存在 T_{11} 类型的元素 Id_{11}，T_{12} 类型的元素 Id_{12}…，且对任一 T_{21} 类型的元素 Id_{21}，T_{22} 类型的元素 Id_{22}…，使得条件表达式 *CE* 成立。其中，$Id_{1i}(i=1,2\cdots)$ 是紧跟关键字 *exist* 之后的元素的别名，并且它声明的类型为 T_{1i}；$Id_{2j}(j=1,2\cdots)$ 是紧跟关键字 *all* 之后的元素别名，并且它声明的类型为 T_{2j}。

条件表达式 *CE* 是一个布尔逻辑表达式，支持与('&&')、或('||')、非('!')的逻辑运算符。用户还可以通过在条件表达式中添加括号'('和')'，来改变运算符的运算先后次序。

可以看到，在有关 *CE* 的语法规则中有两个条件表达式 *Id*. *Att*='value'和 *Id Rel Id*，它们是结构最简单的条件表达式。本文把这两个表达式称作原子表达式。任何一个条件表达式都是在一个或多个原子表达式的基础上添加与'&&'、或'||'、非'!'以及括号'('和')'等运算符拼接而成的。其中，*Id*. *Att*='value'称为属性条件表达式。本文规定：如果元素 *Id* 的 *Att* 属性的值等于 value，则表达式的值为 True；否则表达式的值为 False。Id_1 *Rel* Id_2 称为关系条件表达式，规定：如果元素 Id_1 与 Id_2 间存在关系 *Rel*，则表达式的值为 True，否则表达式的值为 False。

下一节将详细介绍当前工作中已经引入的元素的属性和元素间关系。不同类型的元素拥有的属性可能不同，不同类型元素间存在的关系也可能不同。

3.2 元素的属性和元素间关系的定义

3.2.1 元素的属性

针对各个不同类型的元素，本文分别定义了它们各自具有的属性，这些属性值都是在程序分析阶段收集得到的。

variable 类型元素的属性 *Att* 包括：(1) name，表示程序中声明的变量名；(2) dataType，表示变量的数据类型；(3) specificType，表示变量的具体类别，取值可以是局部变量'internal'、成员变量'field'或者全局变量'global'。

statement 类型元素的属性 *Att* 包括：specificType，表示语句的具体类别，取值包括复合语句'compound'、表达式语句'expression'等。

function 类型元素的属性 *Att* 包括：(1) name，表示程序中声明的方法名；(2) returnType，表示方法的返回类型；(3) paramsType，表示方法的参数类型；(4) specificType，表示方法的具体类别，指明是成员方法还是全局方法。

class 类型元素的属性 *Att* 包括：name，表示类名。

3.2.2 元素间关系

在进行代码查询时，用户还可以根据元素之间的关系来指明被查找的程序元素需要满足的条件。这些元素间的关系信息也是在程序分析阶段收集得到的。目前定义的元素间的关系 Rel 包括：类之间的继承关系 extend，语句与语句、方法、类之间以及成员方法与类之间的从属关系 isIn，由语句中方法调用造成的语句与方法、方法与方法等之间的调用关系 call，与变量有关的使用关系 use 和改变关系 change。

目前，总体上本文引入了 9 种元素属性信息和 18 种元素间关系。该查询语言具有很好的可扩展性，在后续工作中，可以根据需要添加更加丰富的元素属性和元素间关系。

这里，给出一个例子来具体说明查询语言的应用。假设要找出名为 funcF 的全局方法中改变了 int 型变量 i 的值的语句，则查询命令 S 可以是

```
S:= find s:statement satisfying
    exist f:function exist v:variable
    f.name='funcF' && f.specificType='global' &&
    v.name='i' && v isIn f && s isIn f && s change v
```

4 程序元素信息的收集

本节将介绍 C2Parser 的信息抽取模块的工作过程，即对程序结构元素信息的收集。模块具体实现流程：首先，工具以 C/C++工程源代码为输入，利用 Eclipse 的 CDT 插件解析 C/C++源代码，生成代码对应的抽象语法树(Abstract Syntax Tree, AST)。然后，工具通过遍历访问 AST 来收集所关注的 4 类程序结构元素信息，并保存在相应的中间结构——程序元素信息集合中。这样，在目标检索模块中工具就能够根据用户输入的查询命令在程序元素信息集合中检索出符合查询条件的目标元素。

4.1 CDT[1)]

CDT(C/C++ Development Tooling)是 Eclipse 平台下的一个插件。它提供了 Eclipse 平台下支持 C/C++编辑、编译、调试、执行的 IDE。CDT 是开源的，并且为开发者提供了访问代码编译过程中的中间结构——抽象语法树 AST 的相关 API。因此，开发者可以基于 AST 完成程序静态分析工作。同时，CDT 的 AST 数据结构严整、访问便捷、内容详细，除了传统的 AST 语法描述外，还提供了绑定等语义信息，为 C/C++语言的程序分析等工作提供了更大的便利。

对 AST 的搜索或者访问是基于 Visitor(访问者)模式实现的。CDT 中已经实现了抽象类 ASTVisitor 来支持对 AST 的遍历。开发者若要根据自己的要求实现对 AST 的访问和操作，只需要创建一个新的 ASTVisitor，并通过重载其中的 visit 方法来实现对各类型节点的访问和节点上的信息收集。利用 CDT 这样的机制以及各种接口，可以很方便地展开本文的工作。

4.2 元素信息的收集

程序元素信息的收集工作是基于 CDT 的 AST 来实现的。在这部分工作中，需要从工程源代码的 AST 中提取出所有的变量、语句、方法及类元素的信息，并设计一定的数据结构将其保存在内存中。这里的信息包括：(1) 各类型元素的属性信息和元素间关系信息，这部分信息作为后面检索目标元素所对象集；(2) 其他辅助信息，比如这里为各个程序元素保存了它在 AST 上对应的节点，通过对这些 AST 节点的二次访问来确定元素间存在的关系。另外，工具在这里还记录了各元素在源文件中的位置信息，以便对检索结果集合中的元素定位。

为此，本文设计了一个元素收集器 ElementCollection (EC)来保存所有元素的信息。EC 中包括 4 个 Set 结构，即 vSet、sSet、fSet、cSet，分别用来保存 variable、statement、function 和 class 类型的元素。

程序元素信息收集的具体过程如图 3 所示。首先，工具对工程中全部源文件的 AST 进行一次遍历。在 AST 中，一个节点对应于程序中的一个代码结构，如果当前节点对应于需要关注的 4 种类型的程序元素所对应的代码结构，工具就在 EC 中相应类型的 Set 中新建一个元素实例，同时解析该 AST 节点，并在实例中记录下元素的属性信息；另外，实例中还要保存当前的 AST 节点。待全部的 AST 遍历结束之后，所有元素的实例都已经被创建，工具再对各元素实例中保存的 AST 节点二次遍历，收集元素间的关系信息。当访问到一个元素 x 的 AST 节点时，工具会从中解析出与 x 存在某种关系 r 的元素 y 的标识信息。这里的标识信息是指可以区分该元素与同类型其他元素的特征信息。然后，工具会根据此标识信息到 EC 中找出与之匹配的元素 y 的实例，建立 x 与 y 对应实例的关联。这样就完成了元素 x 与 y 间关系 r 的确定。

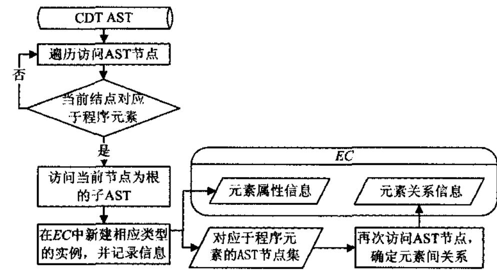

图 3 元素信息收集流程图

至此，EC 中完成了对工程中源程序的全部元素信息的收集。工具的下一个模块的工作是根据用户输入的查询命令在 EC 中检索满足查询条件的程序元素。

5 目标元素的检索

本节介绍 C2Parser 中目标检索模块的工作过程：首先，工具读取用户输入的查询命令 S，并对它进行字符串的扫描，判断它是否满足第 3 节提出的语法规则，并在扫描的过程中为接下来的检索过程做一些预处理；然后，若 S 满足语法规则，工具将根据 S 中规定的查询条件在信息收集模块得到的元素信息集合 EC 中检索出满足查询条件的结果元素集合，并返回给用户。

5.1 语法检查及检索准备工作

```
算法: CE 转换为二叉树
输入: 条件表达式 CE
输出: 树的根节点 root
对 CE 中的每一个原子条件表达式 atomCE {
    //建立与 atomCE 对应的树节点 node
    node.data ←atomCE;
    node.lchild ←null;
    node.rchild ←null;
}

对 CE 中的每一个逻辑运算符 op, (op ∈ {&&,||,!})按运算优先级由高到低先

后做{
    //建立与 op 对应的节点 node
    node.data ←op
    if(op=="!") {
        //op 的唯一运算子是 CE0, CE0 对应的树节点是 node0
        node.lchild ←null;
        node.rchild←node0;
    }
    else {
        /** op 的左右运算子是 CE1、CE2, 它们对应的树节点分别是
           node1、node2 */
        node.lchild ← node1;
        node.rchild← node2;
```

图 4 CE 转换为二叉树的算法

---

1) http://www.eclipse.org/cdt/

工具对查询命令 S 的字符串从头开始扫描，检查 S 是否满足语法规则的描述。在这个过程中，工具还记录下 S 中的元素声明信息(形如“ $Id:T$”)。对于每一个这样的声明，需要保存它的别名 Id、类型 T 等信息。在扫描条件表达式 CE 时，工具除了做语法检查外，为了准备接下来计算条件表达式，还需要将 CE 转换为二叉树结构，以实现条件表达式的计算。具体算法如图 4 所示。

5.2 检索查询目标

工具完成上一节中的处理工作，并且在扫描过程中没有发现语法错误，则进入查询目标的检索过程。它按照查询命令 S 中的查询条件在程序元素信息集 EC 中检索出满足条件的目标元素集合。在上一节的工作中，EC 收集了 4 种类型的元素信息。在 EC 中，每一种类型的元素都对应一个 Set 结构，保存了这一类型的全部元素。检索的过程就是对 EC 中的元素进行遍历，并查找满足查询条件的元素的过程。

对于一个用户输入的查询命令：

S：= find target：T_0 satisfying

exist Id_{11}：T_{11} exist Id_{12}：T_{12}… exist Id_{1m}：T_{1m}

all Id_{21}：T_{21} all Id_{22}：T_{22}… all Id_{2n}：T_{2n}CE

工具在完成语法解析之后，根据目标元素 *target* 声明的类型 T_0，在 EC 中获得 T_0 类型对应的元素集 Set_0 ($Set_0 \in \{vSet、sSet、fSet、cSet\}$)，即 *target* 元素可能指向的值集。接下来，将 Set_0 中的元素循环地“赋值”给 *target*，记录下其中可以让 S 中提出的查询条件得到满足的赋值，并将其存入查询结果集合 Rs 中。对于在关键字 exist 之后声明的元素 Id_{1i} ($1 \leqslant i \leqslant m$)，根据它的声明类型 T_{1i} ($1 \leqslant i \leqslant m$)，可以在 EC 中取得与 T_{1i}类型对应的元素集 Set_{1i}。设 Set_{1i}中元素个数为 N_{1i}，即对于 Id_{1i}，它的可选赋值有 N_{1i}个。那么，对于 $Id_{11}, Id_{12}, \cdots, Id_{1m}$这 m 个元素，有 $N_1 = N_{11} * N_{12} * \cdots * N_{1m}$种可能的赋值组合。同样，对于 $Id_{21}, Id_{22}, \cdots, Id_{2n}$这 n 个元素，有 $N_2 = N_{21} * N_{22} * \cdots * N_{2n}$种可能的赋值组合，其中 N_{2j} ($1 \leqslant j \leqslant n$)是元素 Id_{2j}的可选赋值个数。前面提到，S 中的查询条件 CS 的含义是：存在 T_{11}类型的元素 Id_{11}、T_{12}类型的元素 Id_{12}…T_{1m}类型的元素 Id_{1m}，对任一个 T_{21}类型的元素 Id_{21}、T_{22}类型的元素 Id_{22}……T_{2n}类型的元素 Id_{2n}使得条件表达式 CE 成立。要使 *target* 的当前赋值 *tempTarget* (*tempTarget* $\in Set_0$)满足查询条件 CS，必须使 *target* 赋值为 *tempTarget* 时 $Id_{11}, Id_{12}, \cdots, Id_{1m}$这 m 个元素的 N_1 种赋值组合中存在一种组合，使得 $Id_{21}, Id_{22}, \cdots, Id_{2n}$这 n 个元素的全部 N_2 种赋值组合中任一种组合都能使条件表达式 CE 的值为 True。如果能够满足，就将 *tempTarget* 添加到结果元素集合 Rs 中。具体的处理过程如图 5 中所示。

```
算法：检索满足查询条件的目标元素
输入：程序元素集 EC，条件表达式 CS
输出：Rs
从 EC 中获得与 target 类型一致的元素集合 Set0;
对 Set0 中的任一元素 tempTarget {
        target ←tempTarget;
        对 CS 中在 exist 后声明的 Id1i (1≤i≤m)和 all 后声明的 Id2j (1≤i≤n)分别
从 EC 获得与其类型一致的元素集合 Set1i 和 Set2j;
        用穷举的方式从 Set1i 和 Set2j 中取元素对 Id1i 和 Id2j 这 m+n 个对象赋值;
        计算这 N1*N2 个情况下 CE 的值。


        if(∃temp1i∈Set1i   ,∀ temp2j∈Set2j   compute(CE)==True)
```

图 5 检索目标元素的算法

在图 5 的算法中需要计算条件表达式对于特定的变量取值是否成立。在上一小节中，已经将条件表达式 CE 转换成了二叉树。容易发现，在这个二叉树的结构中，非叶子节点对应于 CE 中的逻辑操作符(&&、||或者!)，叶子节点都对应于原子表达式。这里可以把对条件表达式的计算转变为相应二叉树上的递归处理过程。实现一个方法 compute(BTNode)，它的实现算法如图 6 所示。

```
算法：计算条件表达式 CE
输入：CE 转换成的二叉树的根节点 root
输出：计算结果 r   (r∈{True, False})
if(root 对应于一个逻辑操作符) {
        if(root 对应于&&) {
                r=compute(root.lChild)&& compute(root.rChild);
        }
        else if(root 对应于||) {
                r=compute(root.lChild)|| compute(root.rChild);
        }
        else if(root 对应于!) {
                r=! compute(root.rChild);
        }
else if(root 对应于一个原子条件表达式 atomCE) {
        直接计算 atomCE，结果作为 r 返回;
}
```

图 6 计算条件表达式 CE 的算法

通过算法 5 中的处理过程，工具就可以完成对查询目标元素的检索，并将结果元素集合 Rs 输出给输入查询命令的用户。因为 CDT 的 AST 结构中记录着每个 ASTNode 对应的代码片段在源程序中的位置信息，所以 Rs 中的结果元素可以与实际代码位置关联起来。

6 工具的应用

本节将工具 C2Parser 应用于几个实际的场景，以验证它在实际工作中的应用价值。

1)在工业生产中，一些微机的芯片存储空间很小，供函数调用使用的栈空间也很小，这就要求在芯片上运行的应用程序应避免声明数组类型的局部变量，以免造成栈溢出。这个问题在针对这类程序的代码审查工作中很受关注。本文的工具可以用来辅助这个审查过程，提高效率。借助于它，审查人员可以找出程序中出现的全部数组类型的局部变量，极大地缩小需要审查的代码范围。相应的查询命令 S 是：

find x：variable satisfying x. specifictype='internal'

&& x. datatype='array'

为了证明工具的有效性，本文以一个 C++开源项目 NppTags 为测试对象。NppTags 中有 26 个 Class、1904 个 Function、2 万行左右的代码。对这个项目运行工具，并执行上面的查询命令 S，结果返回 70 个数组类型的局部变量。

2)利用本文的工具验证 MISRA[2] C 编码标准中的部分规则。MISRA C 是汽车工业软件可靠性联合会在 2004 年提出的 C 语言编码标准。通常认为，如果一个 C 语言程序能够完全遵守这些标准，则它是易读、可靠、可移植和易于维护的。它包括 127 条规则，涉及标识符、类型、表达式、控制流、函数等语言的多个方面。本文的工具可以验证其中有关变量、语句、方法的部分规则，找出不符合这些规则的源码。比如，规则 14.4 不应使用 goto 语句，以及规则 14.5 不应使用 continue 语句。这两条规则相应的查询命令是 S_1：find x：statement satisfying x. specifictype='goto'和 S_2：find x：statement satisfying x. specifictype='continue'。

我们依然以 NppTags 项目作为测试对象运行工具，分别

[2] http://www.misra.org.uk/

输入查询命令 S_1 和 S_2，结果分别得到 0 个 goto 语句和 3 个 continue 语句。还比如规则 15.3：switch 语句的最后子句应该是 default 子句。因为提取的信息粒度不够，本文的工具不能判断一个语句是不是 switch 语句的最后一个子句，但它可以找出不包含 default 子句的 switch 语句，这样的 switch 语句同样是不满足规则 15.3 的。相应的查询命令 S 是：

```
find x:statement satisfying
  all y:statement x.specifictype= 'switch'
  &&(! y isIn x || ! y.specifictype= 'default')
```

依然以 NppTags 项目作为测试对象运行工具，输入 S，结果返回 64 个不满足规则 15.3 的 switch 语句。

本节给出了本文工具适用的两个实际的应用场景，在一定程度上验证了本文代码查询技术的研究意义以及工具的有效性。

7 相关工作

代码查询技术普遍采用了抽取-查询-展示(extract-abstract-present)[1,11]的模式来实现。它包括 3 个步骤，即抽取：扫描源代码，将其映射到某种中间结构，比如图；查询：对中间结构进行一些处理和查询操作，以获得查询结果；展示：展示查询结果。在一种代码查询技术提出的同时，通常会引入一个与领域相关的查询语言来描述查询的需求。

Holt[2,9]在 20 世纪 90 年代实现了 Grok 工具，并首先提出了基于 Tarski 关系代数[10]的代码查询语言形式。它主要用来做基于 C++的软件架构分析，所关注的程序中的信息粒度较大、信息量有限，没有处理有关变量、语句层次的程序信息。

同样基于 Tarski 关系代数的还有 2007 年 Storm 发布的 JRelCal[12]，它专门用作 Java 程序的外接 lib。JRelCal 本身不提供抽取过程的支持，只是以 RSF 格式文件作为中间结构进行查询操作。

2007 年，Moor 等人提出了一种通用的面向对象的查询语言. QL[13]，并基于此开发了 SemmleCode[3]——一个 Eclipse 平台的插件，用于 Java 源码的分析。. QL 的语言形式是基于 Codd 关系代数[14]的。

本文提出的以程序元素为目标、面向 C/C++的查询技术是基于一阶谓词逻辑的查询语言形式的。相比于其他查询语言，其语法结构相对简明易懂。同时，它支持处理的程序信息覆盖面广、粒度较小，包括变量、语句、方法、类的属性信息以及它们之间的各种关系信息。虽然目前处理的用户查询命令是一条语句，但该语言对用户需要查询的目标仍具有相当可观的表述能力，且语言本身还有很好的扩展性。

结束语 代码查询技术在程序分析和测试研究中有广泛的应用。本文提出并实现了一种基于程序分析的面向 C/C++的代码查询技术，其能应用于代码审查、程序自动插桩等常用的软件工程工作场景。它通过代码静态分析获得程序元素信息并保存为中间结构，作为代码查询过程的目标集合；查询过程以程序元素为目标，以谓词逻辑表达式的形式来描述查询条件。基于此技术，本文实现了一个基于 C/C++语言的原型工具 C2Parser。通过将其应用于几个实际场景，验证了工具的有效性和研究意义。

下一步的研究工作将在以下几个方面展开：第一，扩充目标语言。只需要利用 JDT 实现对 Java 程序元素信息的收集，就可以将本文的工作迁移到 Java 语言上来。第二，扩充查询语言。可以引入更多种类的元素，如表达式类型；引入更丰富的元素属性和元素间关系；支持输入语句更丰富的表达，比如支持对查询结果的二次查询，加入形如 find…in(find…)的语法支持。

参 考 文 献

[1] Feijs L, Krikhaar R, van Ommering R. A relational approach to support software architecture analysis[J]. Software-Practice and Experience, 1998, 28(4): 371-400

[2] Holt R C. Structural manipulations of software architecture using Tarski relational algebra[C]//WCRE'98. IEEE Computer Society, 1998: 210

[3] Verbaere M, Hajiyev E, de Moor O. Improve software quality with SemmleCode: an Eclipse plugin for semantic code search[C]//OOPSLA'07. New York, NY, USA: ACM, 2007: 880-881

[4] Fagan M. Design and code inspection to reduce errors in program development[J]. IBM Systems Journal, 1976, 15(3): 182-211

[5] Gilb T, Graham D. Software Inspection[M]. AddisonWesley, 1993

[6] Marin M, Moonen L, van Deursen A. Soquet: Query-based documentation of crosscutting concerns[C]//ICSE'07. IEEE, 2007: 758-761

[7] Huang J C. Program Instrumentation and Software Testing[J]. Computer, 1978, 11: 25-32

[8] Chen S K, Fuchs W K, Chung J Y. Reversible debugging using program instrumentation[J]. IEEE Transactions on Software Engineering, 2001, 27(8): 715-727

[9] Holt R C. Binary relational algebra applied to software architecture[R]. CSRI Technical Report 345. Computer Systems Research Institute, University of Toronto, March 1996

[10] Tarski A. On the calculus of relations[J]. The Journal of Symbolic Logic, 1941, 6(3): 73-89

[11] Tilley S R, Paul S, Smith D B. Towards a framework for program understanding[C]//IWPC'96. IEEE Computer Society, 1996: 19-28

[12] Rademaker P. Binary relational querying for structural source code analysis[D]. Utrecht, Netherlands: Software Technology Group, Universiteit van Utrecht, 2008

[13] de Moor O, Verbaere M, Hajiyev E. Keynote address:. QL for source code analysis[C]//SCAM. 2007: 3-16

[14] Codd E F. A relational model of data for large shared data banks[J]. Commun. ACM, 1970, 13(6): 377-387

[15] Holt R C, Winter A, Wu J. Towards a common query language for reverse engineering[R]. 8/2002. Fachbereich Informatik, Universitat Koblenz Landau, June 2002

(上接第 135 页)

[14] Choi O, Han Sang-yong, Abraham A. Semantic matchmaking services model for the intelligent Web services [C]//International Conference on Computational Science and Applications. UK: IEE Press, 2006

[15] Zhu Zheng-zhou, Wu Zhong-fu, Zhou Shang-bo. An e-Learning Services Discovery Algorithm Based on User Satisfaction[C]//The 2008 IEEE International Conference on Networking, Architecture, and Storage. 2008: 342-348