RETE 网络中的优化编译模式及其 PVS 形式验证*)

刘晓建　陈　平
(西安电子科技大学软件工程研究所　西安710071)

Optimizing Compiler Schemas in RETE Network and their Formal Verification with PVS

LIU Xiao-Jian　CHEN Ping
(Institute of Software Engineering, Xidian University, Xi'an 710071)

Abstract In the compilation of rule program to the intermediate code—RETE network, optimizing compilation is an important compiler schema, and is a necessary step in the compiler verification. In this paper, we discuss optimization schemas in rule program compilation, and prove the semantic equivalence theorems of these schemas. Firstly, the structure of RETE network and its PVS specification are represented. Secondly, three kinds of optimization schemas are listed. Then algorithms evaluating semantics of target RETE network are given. Finally, we prove the semantic equivalence theorems with theorem prover PVS (Prototype Verification System).

Keywords RETE network, Rule-based system, Compiler verification, Mechanized theorem proving, PVS

1. 引言

基于 RETE 算法[1]的规则系统是实现指挥控制系统中的高性能战场态势监视与文电路由转发的核心计算模型。传统的规则系统，如 OPS5，CLIPS[2]等都是自封闭的规则系统，很难和常用的程序设计语言，如 C/C++互操作。iLog Rules[3]虽然可以满足规则语言与 C++语言互操作的要求，但是由于不提供源码，不能满足"可信软件工程技术"项目所需要的"开放源码与文档以备安全审查"的基本要求。因此，我们在 CLIPS 的基础上[4]，参考 iLog Rules 设计和开发了满足指控要求的规则语言编译器。由于指控系统的健壮性直接关系到战争的胜败，因此必须验证在指控系统中使用的规则语言编译器[5~7]，其中对规则语言编译器中优化编译模式的验证是编译器验证中的重要环节。

2. RETE 网络的结构及其 PVS 形式描述

在规则程序中，每一条规则的形式为：LHS→RHS，其中 LHS(Left Hand Side)为规则的条件部分，RHS(Right Hand Side)为动作部分。规则 LHS 中 pattern 的 BNF 范式见文[3]。规则源程序经过规则编译器编译成 RETE 网络中间代码形式，以进行快速模式匹配。图1为 RETE 网络的拓扑结构。

其中root为根节点(ROOT)，Ci为class节点(CLASSNODE)，fij 为 pattern 节点(PATTERNNODE)，Ji 和 Jij 为 join 节点(JOINNODE)。Pattern 网络为二叉树结构，树的叶节点为 ALPHANODE；JOINNODE 可以分为三类，即 BETANODE，NEGNODE 和 NANDNODE。alpha 内存和 beta 内存分别与 ALPHANODE 和 JOINNODE 相关联。

下面是 RETE 网络的 PVS 形式规约。

定义1　RETE 网络的 PVS 形式规约。

```
rete_network: THEORY
  BEGIN
    CLASS, REL: TYPE
    %网络节点类型
    NODE: TYPE+
    ROOT, CLASSNODE, RATTERNNODE, JOINNODE: TYPE
    +FROM NODE
    %ALPHA 节点
    ALPHANODE: TYPE+FROM PATTERNNODE
    %JOINNODE 的分类
    BETANODE, NEGNODE, NANDNODE: TYPE + FROM
    JOINNODE
    %常量测试和约束测试
    CONSTTEST: TYPE=pred[CLASS]
    JOINTEST: TYPE=pred[[REL, CLASS]]
    %PATTERNNODE 的结构
    child: VAR[PATTERNNODE→PATTERNNODE]
    sibling: VAR[PATTERNNODE→PATTERNNODE]
    constTest: VAR[PATTERNNODE→CONSTTEST]
    joinTest: VAR[PATTERNNODE→JOINTEST]
    class: VAR[PATTERNNODE→CLASS]
    entry: VAR[ALPHANODE→JOINNODE]
    %JOINNODEA 节点的结构
    rhs: VAR[JOINNODE→NODE]
    lhs: VAR[JOINNODE→JOINNODE]
    nextLevel: VAR[JOINNODE→JOINNODE]
    rightDrive: VAR[JOINNODE→JOINNODE]
    rightMatch: VAR[JOINNODE→JOINNODE]
    %alpha 与 beta 内存
    alphaMem: VAR[ALPHANODE→set[CLASS]]
    betaMem: VAR[JOINNODE→set[REL]]
    %RETE 网络的定义
    RETE: TYPE+=set[NODE]
    %区分树网络
    DISCTREE: TYPE+=set[CLASSNODE]
    PATTERNNETWORK: TYPE+=set[PATTERNNODE]
    JOINNETWORK: TYPE+=set[JOINNODE]
  ENDrete_network
```

3. RETE 网络中的三种优化编译模式

在把规则程序编译到 RETE 网络的过程中，要对其进行优化编译。优化的关键是考虑到节点之间的共享，根据共享节点的类型，我们把优化编译分为下面三种模式：

·PATTERN 网络中相同常量测试节点的共享

·JOIN 网络中 ALPHANODE 的共享

·JOIN 网络中 JOINNODE 的共享

* 基金项目："十五"国防预先研究项目基金"可信软件工程技术"(413150501)。刘晓建　博士生，主要研究领域为软件工程中的形式化方法。陈　平　博士，教授，博士生导师，主要研究领域为软件工程。

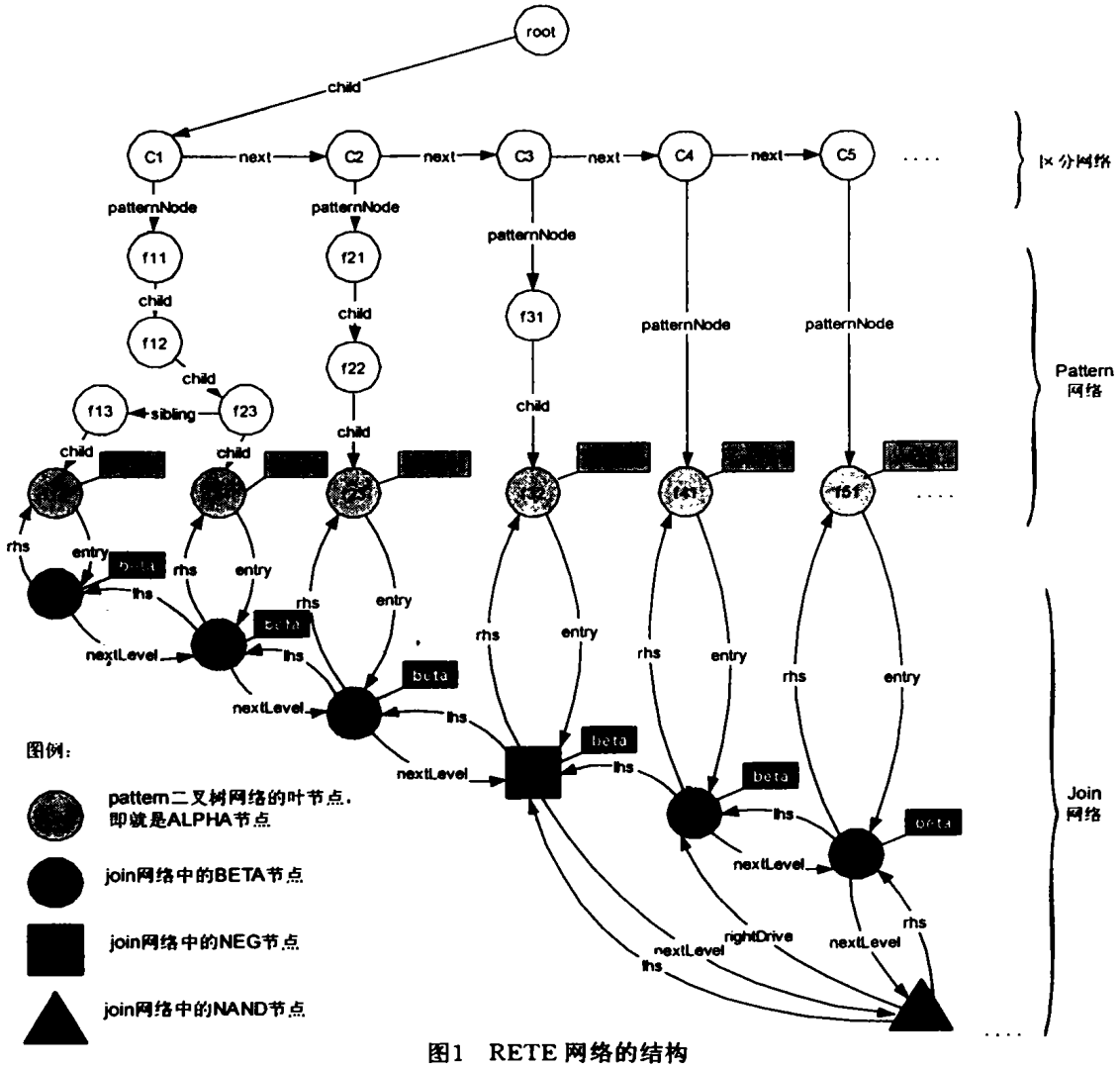

图1 RETE 网络的结构

模式1 PATTERN 网络中相同常量测试节点的共享

对于形如

```
⟨lhs⟩::=⟨simple-pattern⟩⟨any-pattern⟩*
        (Ci(fi1)(fi2)(fi3)(fi4)⟨JOINTEST*⟩)
        ⟨any-pattern⟩*
        (Ci(fi1)(fi2)(gi3)(gi4)⟨JOINTEST⟩*)
        ⟨any-pattern⟩*
        (Ci(fi1)(hi2)(hi3)⟨JOINTEST⟩*)
        ⟨any-pattern⟩*
```

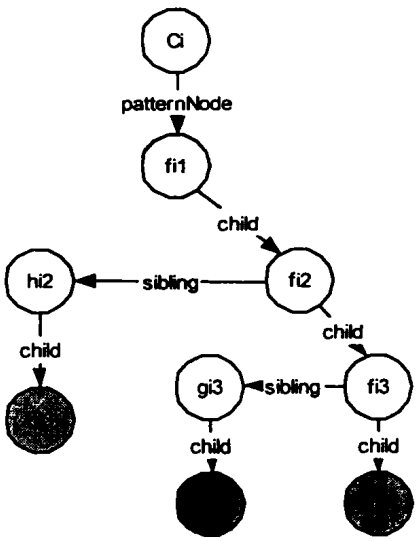

图2 优化模式1的 PATTERN 网络

的规则 LHS,其中 Ci 为 pattern 所引用的类,fi1,fi2,fi3,fi4,gi3,gi4,hi2和 hi3为常量测试。显然这三个 pattern 共享常量测试 fi1和 fi2。

对于这种几个 pattern 共享相同的类和常量测试的规则程序,在形成 PATTERN 网络时,优化为二叉树结构,如图2所示。

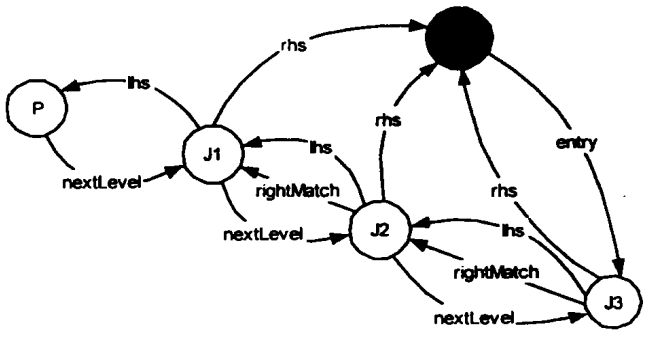

图3 优化模式2的 JOIN 网络

其中带阴影的节点是 ALPHANODE。

模式2 JOIN 网络中 ALPHANODE 的共享

对于形如

```
⟨lhs⟩::=P
        (Ci(costTesti)(joinTest11))
        (Ci(constTesti)(joinTest12))
```

(Ci(constTesti)(joinTest13))
〈any-pattern〉*

的规则 LHS，其中 P≙〈simple-pattern〉〈any-pattern〉*，constTesti 为三个 pattern 都共享的常量测试，joinTest11，joinTest12和 joinTest13为 pattern 间的约束测试。对于这种情形，在编译成 JOIN 网络时共享相同的 ALPHANODE，如图3所示，其中 A 为共享的 ALPHANODE，P、J1、J2和 J3为 JOINNODE。

模式3 JOIN 网络中 JOINNODE 的共享

对于形如

$\langle lhs_1\rangle ::= P$
((C1(constTest1)(joinTest21))
〈any-pattern〉*
$\langle lhs_2\rangle ::= P$
(C2(costTest2)(joinTest22))
〈any-pattern〉*

的两条或多条规则的 LHS，其中 P 为完全相同的 patterns。对于这种情形，在编译成 JOIN 网络时，共享相同的 JOINNODE。如图4所示。

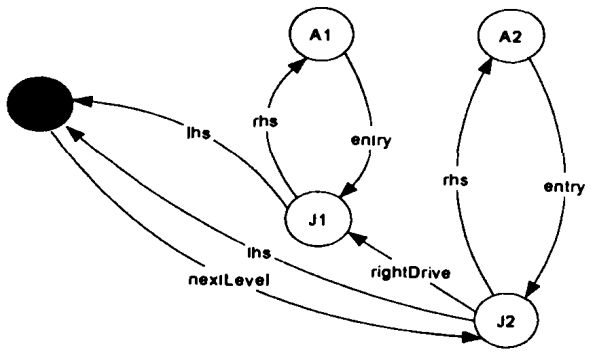

图4 优化模式3的 JOIN 网络

其中，标记为 P 的节点表示共享的 JOINNODE，J1为$\langle lhs_1\rangle$中的 JOINNODE，J2为$\langle lhs_2\rangle$中的 JOINNODE，A1和 A2表示 ALPHANODE。

4. 优化编译模式的形式验证

验证这些优化模式的正确性，也就是证明优化前规则 LHS 的语义和优化编译后 RETE 网络的语义等价。其中规则 LHS 的语义可以用关系集合来描述。为了求出优化编译后的 RETE 网络的语义，我们给出如下算法。

4.1 RETE 网络的语义标记算法

对于 PATTERN 网络，我们给出算法 labelAlphaMem 遍历二叉树，求出 ALPHANODE 上关联的 alphaMem 的关系集合。算法中的数据结构见定义1。

算法1 标记任一 CLASSNODE 下的 pattern 网络叶节点上的 alpha 内存

```
输入:patternNode—当前要标记的 pattern 节点
     constTest—从父节点传下来的常量测试公式
     class—pattern 节点的类
Algorithm labelAlphaMem(patternNode,constTest,class){
  while(patternNode≠nil){
      if(patternNode.sibling≠nil)
          //标记 sibling 节点
            labelAlphaMem (patternNode.sibling, constTest,
      class);
      //按照中序遍历 pattern 网络
      patternNode.class:=class;
      patternNode.constTest := patternNode.constTest ∧ con-
                               stTest;
      constTest:=patternNode.constTest;
      if(patternNode.child=nil){
          //标记叶节点上 alpha 内存
          patternNode.alphaMem:={fact:patternNode.class|
                                 patternNode.constTest
                                 (fact)}
          returen;
      }
      patternNode:=patternNode.child;
  }
}
```

对于 JOIN 网络，我们用算法 labelJoinNodes 依次标记 JOIN 网络上与 JOINNODE 关联的 beta 内存。其中"—"为集合的差集算子，$\triangleright\!\triangleleft_p$ 为关系演算中的连接算子，$\pi_{(schema(R))}$为投影算子，p 为连接谓词，R 为要投影的关系。算法中的数据结构见定义1。

算法2 递归标记 JOIN 网络切片上的所有 JOINNODE 节点的 beta 内存

```
输入:JOIN 网络的首节点
输出:标记输入节点及其以下的所有节点
Algorithm labelJoinNodes(firstNode)
{
   JOINNODE joinNode,listOfJoins;
     //标记首节点
   firstNode.betaMem:=firstNode.rhs.alphaMem;
   joinNode:=firstNode.nextLevel;
   //标记 BETANODE
   if(joinNode∈BETANODE){
       joinNode.betaMem:=joinNode.lhs.betaMem
            ▷◁joinNode.joinTest joinNode.rhs.alphaMem;
   }
   //标记 NEGNODE
   elseif(joinNode∈NEGNODE){
        joinNode.beta := joinNode.lhs.betaMem-πschema(joinNod-
                                    e.lhs.betaMem)
       (joinNode.lhs.betaMem ▷◁joinNode.joinTest joinNode.rhs.al-
   phaM-em);
   }
   //标记下一个节点
   listOfJoins:=joinNode.nextLevel;
   if(listOfJoins≠nil)
     {
         //判断 NANDNODE 节点，并标记
       if(listOfJoins.rhs=joinNode){
           listOfJoins.betaMem:=listOfJoins.lhs.betaMem-
                 πschema(listOfJoins.lhs.betaMem) listOfJoins.rhs.beta
           Mem;
           labelJoinNodes(listOfJoins.nextLevel);
         }
         //按照深度优先的方式遍历 JOIN 网络
       else while(listOfJoins≠nil){
         labelJoinNodes(listOfJoins);
         listOfJoins:=listOfJoins.rightDrive;
       }
   }
   return;
}
```

4.2 优化编译模式的形式验证

定理1 记优化模式1中的三个 pattern 分别为：

p1=(Ci (fi1) (fi2) (fi3) (fi4) 〈JOINTEST〉*)，

p2=(Ci (fi1) (fi2) (gi3) (gi4) 〈JOINTEST〉*)

p3=(Ci (fi1) (hi2) (hi3) 〈JOINTEST〉*)

p1，p2和 p3的语义分别为 M(p1)，M(p2)和 M(p3)。按照优化模式1编译后的 PATTERN 网络(见图2)中对应 p1，p2和 p3的 alpha 内存分别为 alphaMem1，alphaMem2和 alphaMem3，则 M(p1) = alphaMem1 ∧ M(p2) = alphaMem2 ∧ M(p2) = alphaMem3

证明：显然

$M(p1) = \{t: Ci \mid fi1(t) \wedge fi2(t) \wedge fi3(t) \wedge fi4(t)\}$

$M(p2) = \{t: Ci \mid fi1(t) \wedge fi2(t) \wedge gi3(t) \wedge gi4(t)\}$

$M(p3) = \{t: Ci \mid fi1(t) \wedge hi2(t) \wedge hi3(t)\}$

由算法1我们可以求出优化后的 PATTERN 网络上的 alpha 内存，显然有

M(p1) = alphaMem1，M(p2) = alphaMem2 和 M(p2) = alphaMem3，定理1得证。

定理2 记优化模式2中三个 pattern 分别为：

p1=(Ci (constTesti) (joinTest11))

p2=(Ci (constTesti) (joinTest12))

p3=(Ci (constTesti) (joinTest13))

则 M(P p1 p2 p3) = M($RETE_{P\,p1\,p2\,p3}$)，其中

$RETE_{P\ p1\ p2\ p3}$为规则 LHS :P p1 p2 p3优化后的 RETE 网络见图3,M 为语义函数。

证明:M(P p1 p2 p3)

$= \{t: R\times Ci\times Ci\times Ci \mid \exists\ t1:R\cdot(t[1]=t1 \wedge \exists\ t2:Ci\cdot(constTesti(t2) \wedge joinTest11(t1,t2) \wedge t[2]=t2 \wedge \exists\ t3:Ci\cdot(constTesti(t3) \wedge joinTest12(t1,t2,t3) \wedge t[3]=t3 \wedge \exists\ t4:Ci\cdot(constTesti(t4) \wedge joinTest13(t1,t2,t3,t4) \wedge t[4]=t4))))\}$ (1)

由算法2得到图3的 RETE 网络的语义为

$M(RETE_{P\ p1\ p2\ p3}) = ((R \bowtie_{joinTest11} A) \bowtie_{joinTest12} A) \bowtie_{joinTest13} A$ (2)

其中 R=M(P),A 为共享的 ALPHANODE,即 A={t:Ci|constTesti(t)}

为了证明 $M(P\ p1\ p2\ p3)=M(RETE_{P\ p1\ p2\ p3})$,我们将公式(1)和(2)表示成定义2中 PVS 形式规约中的公式 lhs1和 rete1,并把定理 $M(P\ p1\ p2\ p3)=M(RETE_{P\ p1\ p2\ p3})$表示成 PVS 定理 th1。

定理3 记优化模式3中的 pattern 分别为:

p1=(C1 (constTest1) (joinTest21))

p2=(C2 (constTest2) (joinTest22))

则 $M(P\ p1)=M(RETE_{P\ p1}) \wedge M(P\ p2) = M(RETE_{P\ p2})$

证明:

$M(P\ p1)=\{t: R\times C1 \mid \exists\ t1:R\cdot(t[1]=t1 \wedge \exists\ t2:C1\cdot(constTest1(t2) \wedge joinTest21(t1,t2) \wedge t[2]=t2))\}$ (3)

$M(P\ p2)=\{t: R\times C2 \mid \exists\ t1:R\cdot(t[1]=t1 \wedge \exists\ t2:C2\cdot(constTest2(t2) \wedge joinTest22(t1,t2) \wedge t[2]=t2))\}$ (4)

优化后的 RETE 网络(见图4)的语义由算法2求得:

$M(RETE_{P\ p1}) = R \bowtie_{joinTest21} A1$ (5)

$M(RETE_{P\ p2}) = R \bowtie_{joinTest22} A2$ (6)

其中 R=M(P),A1={t:C1 | constTest1(t)},A2={t:C2 | constTest2(t)}

我们把公式(3)(4)(5)(6)分别表示为定义2的 PVS 公式 lhs2,lhs3,rete2和 rete3,并把定理3表示为 PVS 定理 th2。

为了机械化证明上面的定理2和定理3,我们给出如下的 PVS 形式规约和证明策略。

定义2 定理2和定理3证明的 PVS 形式规约

```
prf:THEORY
  BEGIN
   %定义类型
   Ci,C1,C2:TYPE
   R:TYPE
   %定义常量测试和约束测试
   constTesti:pred[Ci]
   constTest1:pred[C1]
   constTest2:pred[C2]
   joinTest11:[R,Ci→bool]
   join Test12:[R,Ci,Ci→bool]
   joinTest13:[R,Ci,Ci,Ci→bool]
   joinTest21:[R,C1→bool]
   joinTest22:[R,C2→bool]
   %定义 alpha 内存
   A:set[Ci]={t:Ci|constTesti(t)}
   A1:set[C1]={t:C1|constTest(t)}
   A2:set[C2]={t:C2|constTest2(t)}
   lhs1:set[[R,Ci,Ci,Ci]]={t:[R,Ci,Ci,Ci]|EXISTS(t1:R):(t'1
              =t1 AND
              EXISTS(t2:Ci):(constTesti(t2)AND join Test11
              (t1,t2)AND t'2=t2 AND
                 EXISTS (t3: Ci): (constTesti (t3) AND join
              Test12(t1,t2,t3)AND t'3=t3 AND
                    EXISTS (t4: Ci): (constTesti (t4) AND join
              Test13(t1,t2,t3,t4)AND t'4=t4))))}
   rete1:set[[R,Ci,Ci,Ci]]={t:[R,Ci,Ci,Ci]|EXISTS(t1:R):(t'
              1=t1 AND
              EXISTS(t2:Ci):(member(t2,A)A ND join Test11
              (t1,t2)AND t'2=t2 AND
                 EXISTS (t3: Ci): (member (t3, A) AND join
              Test12(t1,t2,t3)AND t'3=t3 AND
                    EXISTS (t4: Ci): (member (t4, A) AND join
              Test13(t1,t2,t3,t4)AND t'4=t4))))}
   th1:THEORY
      subset?(lhs1,rete1)AND subset?(rete1,lhs1)
   lhs2:set[[R,C1]]={t:[R,C1]|EXISTS(t1:R):(t'1=t1AND
              EXISTS(t2:C1):(constTest1 (t2) AND join Test21
              (t1,t2)AND t'2=t2))}
   lhs3:set[[R,C2]]={t:[R,C2]|EXISTS(t1:R):(t'1=t1AND
              EXISTS(t2:C2):(constTest2 (t2) AND join Test22
              (t1,t2)AND t'2=t2))}
   rete2:set[[R,C1]]={t:[R,C1]|EXISTS(t1:R):(t'1=t1AND
              EXISTS(t2:C1):(member(t2,A1)AND join Test21
              (t1,t2)AND t'2=t2))}
   rete3:set[[R,C2]]={t:[R,C2]|EXISTS(t1:R):(t'1)=t1AND
              EXISTS(t2:C2):(member(t2,A2)AND join Test22
              (t1,t2)AND t'2=t2))}
   th2:THOERY
      (subset?(lhs2,rete2)AND subset?(rete2,lhs2))AND
      (subset?(lhs3,rete3)AND subset?(rete3,lhs3))
END prf
```

我们在 Redhat Linux7.2平台上运行的 PVS V2.4中,用如下的证明策略:

(grind)(detuple-boundvars)(grind)证明了两个定理 th1和 th2,因此定理2和定理3得证。

结束语 优化编译模式的验证保证了规则程序在优化编译前后的语义等价性,但它只是整个规则编译器验证中的一个必要条件。进一步的研究工作还有对规则编译器中的各种 pattern 变换,如 exists/forall pattern 变换,and-or 变换,not-or 变换等进行验证,以及对整个规则编译器代码进行翻译确认(Translation Validation)[9~11]。

参 考 文 献

1 Forgy C. Rete: A fast algorithm for the many-pattern/many-object pattern-match problem. Artificial Intelligence, 1982, 19(1): 17~37

2 Brownston L, Farrel R, Kant E, Martin N. Programming Expert Systems in OPS5: An Introduction to Rule-Based Programming. Addison-Wesley, 1985

3 ILOG Rules User's Manual, ILOG Corp, 1996

4 刘晓建,陈平. 一种基于规则的系统的软件体系结构的实现分析. 计算机科学,2002,29(6):134~136

5 Stringer-Calvert D. Mechanical verification of compiler correctness: [Ph. D. Thesis]. Department of Computer Science, University of York, 1998. URL: http://www.csl.sri.com/papers/thesis/

6 Goos G, Zimmermann W. Verification of Compilers. In: Proc. of Correct System Design LNCS 1710, Springer-Verlag 1999. 201~230

7 Strecker M. Formal Verification of a Java Compiler. in Isabelle. In: Proc. Conf. on Automated Deduction (CADE), LNCS 2392, Springer Verlag, 2002

8 PVS Manuals. URL: http://pvs.csl.sri.com/manuals.html

9 Pnueli A, Siegel M, Singerman E. Translation Validation. In: Proc. of TACAS'98, LNCS 1384, 1998. 151~166

10 Zuck L, Pnueli A, Leviathan R. Validation of Optimizing Compilers: [Technical report]. CIMS/CS/NYU. URL: http://www.cs.nyu.edu/validator/pubs.htm l

11 Zuck L, Pnueli A, Goldberg B. Translation Validation of Loop Optimizations in Optimizing Compilers: [Technical report]. CIMS/CS/NYU. URL: http://www.cs.nyu.edu/validator/pubs.html