# 基于计算解语义的逻辑程序测试和调试框架

赵岭忠[1] 廖伟志[2] 钱俊彦[1] 古天龙[1]

(桂林电子科技大学计算机与控制学院 桂林 541004)[1] (广西师范学院信息技术系 南宁 530023)[2]

**摘 要** 逻辑程序开发过程中需要花费大量的时间用以程序调试,原因之一是调试通常包含大量的用户交互。减少对调试过程不必要的调用能够提高软件开发的效率。程序测试中得到的由同一个错误引发的多个症状是引发对调试过程不必要调用的因素之一。给出了一种逻辑程序测试和调试框架,其中测试用例的生成、症状的发现和调试(包括诊断和改错)交叉进行,由同一个错误引发的症状只有一个可引发调试过程执行,并以此方式避免了对调试过程不必要的调用。然后,利用一种基于约束的Prolog计算解语义,该框架被实例化为一种Prolog程序的测试和调试算法,本实例表明了该算法的应用。

**关键词** 测试,调试,逻辑程序,计算解语义

**中图法分类号** TP31 **文献标识码** A

## Framework for Integrated Testing and Debugging of Logic Programs Based on Computed Answers Semantics

ZHAO Ling-zhong[1] LIAO Wei-zhi[2] QIAN Jun-yan[1] GU Tian-long[1]

(School of Computer and Control, Guilin University of Electronic Technology, Guilin 541004, China)[1]

(Department of Information Technology, Guangxi Teachers Education University, Nanning 530023, China)[2]

**Abstract** Debugging logic program is a time-consuming process that usually contains considerable manual interaction. Reducing unnecessary calls to a debugging procedure can improve the efficiency of software development. Same-error-source symptoms obtained in program testing is a source of unnecessary calls to a debugging procedure. This paper proposed an integrated testing and debugging framework, in which the generation of test cases, discovering symptoms and the debugging (including the diagnosis and correction) of the program under consideration (PUC) are interleaved in such a way that only one of the symptoms with same-error-source relation between each other will lead to the execution of a debugging procedure. In this way unnecessary calls to the procedure are effectively avoided. With a constraint-based fixpoint semantics for Prolog, the framework is instantiated to a novel testing and debugging algorithm, whose applicability and effectiveness are shown by an example in this paper.

**Keywords** Testing, Debugging, Logic programs, Computed answers semantics

## 1 引言

软件测试是一种常用的保障软件质量的方法。通常的方法是对待测试的程序(PUC)进行测试,在测试结束后调用调试过程对测试中发现的错误进行定位和纠正。软件测试可分为基于规格的测试和基于实现的测试。在基于规格的测试中,测试的输入数据和预期结果均由程序的规格产生;而在基于实现的测试中,测试的输入数据从程序的实现中获得,而预期结果则由oracle(由用户或测试人员充当)给出。本文研究后者。

本文中,使PUC运行异常的测试用例被称为症状[1,2]。一般测试方法存在的问题是:在测试过程中发现的症状之间常常存在"同源"关系。如果同一错误导致了 $\sigma_1$ 和 $\sigma_2$ 的产生,则称症状 $\sigma_1$ 和症状 $\sigma_2$ 具有同源关系。典型的情况是,过程 $p$ 调用了一个包含错误的过程 $q$;则测试过程将产生两个症状,一个对应于过程 $p$,另一个对应于过程 $q$。这两个症状具有同源关系。对其中一个症状进行调试即可发现引发两个症状的错误,但按照通常的方法,调试过程将分别以两个症状为输入,执行两次。这就增加了调试的工作量。由于目前为止程序调试尚不能完全自动化,通常需要测试人员的干预,因而以上问题对程序开发的效率有较大的影响。在逻辑程序中,子句体中的每一个文字均被视为过程调用。因而以上问题在逻辑程序的测试和调试中更加严重。例如,已知程序 $P$:

{insert (X,[Y|U],[Y|V]) :-Y<X,insert(X,U,V).

insert(X,[Y|U],[X,Y|U]) :-X≤Y. insert(X,[],[X]).

到稿日期:2008-10-21 返修日期:2008-12-26 本文受国家自然科学基金(60803033,60663005)和广西青年科学基金(桂科青0728093,桂科青0542036)资助。

**赵岭忠** 博士,副教授,主要研究方向为软件验证测试、形式化技术等,E-mail:zhaolingzhong@guet.edu.cn;**廖伟志** 博士,副教授,主要研究方向为模型检验、嵌入式实时系统;**钱俊彦** 博士,副教授,主要研究方向为软件验证、模型检验;**古天龙** 教授,博士生导师,主要研究方向为实时混杂系统理论、形式化方法等。

sort([],[]).

sort([X|Y],Z) :-sort(Y,U),insert(X,U,Z)},

如果把过程 insert 的第一个子句体中的文字 insert($X$,$U$,$V$)改变为 insert($Y$,$U$,$V$),则对该程序的测试可发现两个症状,如 sort([6,−2,4],[−2,−2,4])和 insert(10,[6],[6,6])。两症状的产生是由同一个错误导致的。

为了解决该问题,本文提出一种基于逻辑程序不动点语义的综合逻辑程序测试和调试框架。该方法主要对逻辑程序相对于计算解[1] 规格的部分正确性进行测试和调试。

程序 $P$ 相对于计算解规格的部分正确性定义如下:

令 $O_P$ 为程序 $P$ 中最一般目标计算解的集合,$S$ 是描述 $P$ 中最一般目标预期计算解的计算解规格,$P$ 相对于 $S$ 是部分正确的,如果 $O_P$ 是 $S$ 的子集。

其中,最一般目标是指形如 $p(X_1,\cdots,X_n)$ 且 $X_1,\cdots,X_n$ 为不同变量的目标。显然,任意原子均为某个最一般目标的实例。本文没有给出程序计算解规格的形式定义,但假定对每一个程序 $P$ 存在一个 oracle。已知测试输入和相应的程序输出,oracle 可判定该输出是否是预期的结果,即判定给定的测试用例是否为症状。所以 oracle 在本文中充当了程序计算解规格的角色。

概括地讲,本文的测试和调试方法是对 PUC 不动点语义计算的模拟。基本思想是从计算解语义不动点计算过程的中间结果中获取测试用例。如果测试中发现了一个症状 $\sigma$,则调用调试工具对导致该症状的错误进行定位并纠正。然后重新计算程序的计算解语义,并产生新的测试用例。此时,与 $\sigma$ 具有同源关系的症状不会再出现。PUC 语义的计算和针对某个症状对 PUC 进行调试的过程相互交叉,直至语义计算结束,或者测试人员有足够的信心认为 PUC 相对于计算解而言是部分正确的。对于后一种情况,一个可能的标准是连续 $m$ 步中没有发现任何新的程序错误,或者已经产生的测试用例已经获得了预期的子句覆盖率或路径覆盖率。因此,即使 PUC 的语义并非有限步骤内可计算,该方法仍可应用。

## 2 相关工作

在软件开发过程中,通常的做法是把测试和调试视为不同的开发阶段,测试的输出结果作为调试的输入。该方法的主要问题是,具有同源关系的症状的存在可能导致对调试过程的不必要的调用。在逻辑程序和 Prolog 程序测试中,多数方法遵循以上通常的做法,其中包括 CPM 测试[3,4]、基于控制流的测试[5] 和 PROTest II 测试系统等[6,7]。

最常用的调试方法是声明式调试[8-14]。由于本文方法仅仅是使用现有的程序调试技术,因此不再深入讨论各种调试方法的特点。

本文提出的方法把逻辑程序的测试和调试综合起来。从这个角度来讲,与我们的工作最接近的是 IDTS(集成的调试,测试和切片)系统[15,16]。IDTS 是一个算法调试和基于实现的测试系统。该系统以不同的方式把测试和调试集成起来。该系统的主要特征是利用 Prolog 程序 CPM 测试的结果减少算法调试中的用户交互,同时利用程序调试过程中得到的信息消除 CPM 测试配置(configuration)中的矛盾。IDTS 系统没有考虑同源症状的调试问题。

抽象调试、基于抽象解释的调试和演绎调试[17-21] 是用于发现和定位逻辑程序中的错误的另外一类技术。抽象调试[17,18] 中通常包含一个描述程序某一方面可见行为的参数 $\alpha$[22]。其主要思想是比较程序 $P$ 的形式规格 $I_\alpha$ 和描述 $P$ 的可见行为 $\alpha$ 的不动点语义 $lfp(T_{P,\alpha})=O_\alpha(P)$。程序 $P$ 相对于 $I_\alpha$ 是部分正确的,如果 $O_\alpha(P)\subseteq I_\alpha$;$P$ 相对于 $I_\alpha$ 是完备的,如果 $I_\alpha\subseteq O_\alpha(P)$;$P$ 相对于 $I_\alpha$ 完全正确,如果 $I_\alpha=O_\alpha(P)$。如果 $I_\alpha\neq O_\alpha(P)$,那么程序中的某个子句中必然存在错误。为此对程序 $P$ 中的每一个子句,设计一个特定的语义函数算子,用于确定相应的子句中是否是正确的。在抽象调试中,传统调试技术中 oracle 的功能由程序的规格 $I_\alpha$ 来完成。该方法对于一类其直承算子具有唯一不动点的逻辑程序是有效的。该方法存在的问题是:必须要求程序规格的有限的。由于一般情况下程序的行为是无限的,因此需要利用一些抽象技术获得此有限性。例如文献[18]中,一个无限的程序规格 $I$ 被近似为一个序偶($I^+$,$I^-$),其中 $I^+$ 是 $I$ 的子集,$I^-$ 是 $I$ 的补集的子集。相应地,部分正确性和完备性的概念被分别替换为两个相对较弱的概念:$p$-正确性和 $p$-完备性。遗憾的是,获得有限规格的过程可能导致程序中的某些错误不能被发现。以类型抽象为例,假定感兴趣的类型有两种:$gl$ (ground list) 和 $ng$ (非 ground list)。如果每一个 Prolog 项均被抽象为其类型,那么第一部分中提到的症状 $sort$([6,−2,4],[−2,−2,4])将不能被发现,因为该症状将被抽象为 $sort(gl,gl)$。基于抽象解释的调试和演绎调试中存在的问题可类似分析。

本文方法与抽象调试有较为密切的联系。如果把抽象调试视为一种集成的测试和调试技术,那么该方法表明在逻辑程序测试中测试用例可直接从程序的不动点语义获得。具体地,可以通过检查 $O_\alpha$(P)中的每一个元素均属于 $I_\alpha$ 来说明逻辑程序相对于可见行为 $\alpha$ 的部分正确性。当程序的预期行为 $I_\alpha$ 没有有限的表示或 $O_\alpha$(P)并非有限可计算时,可以利用如下条件近似以上检查过程,即在计算程序 P 的不动点语义的每一步获得的 $O_\alpha$(P)的元素均属于 $I_\alpha$。该思想与本文方法的思想一致,其中可见行为 $\alpha$ 是程序的计算解。对于 $O_\alpha$(P)的元素 e,通过检查 e 的所有代表性实例均和程序的预期语义相一致可近似地确定元素 e 在 $I_\alpha$ 中的成员资格。通过选择适当的停机机制,本文方法的测试质量可以得到提高。

## 3 逻辑程序测试和调试框架

令 $T_P$ 为逻辑程序 $P$ 不动点语义的直承算子,$I_0$ 是最小计算解解释,$T_P(I_0)$ 描述了语义计算的第 $i$ 步获得的最一般目标的计算解。描述过程 $p$ 在程序 $P$ 中执行所获得计算解的元素 $\sigma\in T_P(I_0)$,被称为过程 $p$ 的一个测试用例规格;对该规格进行实例化可生成一个或多个 $p$ 的测试用例,从而测试用例可视为计算解的实例。显然,根据计算解的定义,所获得的测试用例在 PUC 上的执行必然成功,即必然存在该测试用例的一个 SLD-拒绝路径。oracle 可利用自身知识判定该测

[1] 在逻辑程序设计中,目标 G 在程序 P 中执行的过程就是按照某种规则(如 Prolog 采用的深度优先规则)搜索以 G 为根的 SLD-推导树的过程。任何以空目标结束的 SLD-推导路径均对应 G 在 P 中的一个计算解,此时称该路径为 G 在 P 中的一个 SLD-拒绝路径(SLD-refutation)。

试用例是否是一个症状。

下面给出逻辑程序的综合测试和调试框架。

步骤 1　$i:=0$;

步骤 2　计算 $T_P(I_0)$并从中获得未出现在 $T_P^0(I_0),\cdots,T_P^{-1}(I_0)$中的测试用例规格的集合 $D$;

步骤 3　令 $D[p]$表示过程 $p$ 的测试用例规格的集合,对程序 $P$ 中的每一个过程 $p$ 和每一个 $D_i\in D[p]$,重复步骤 3.1 和步骤 3.2;

步骤 3.1　对 $D_i$ 进行实例化,使之满足用户定义的过程 $p$ 的性质规格,从而获得一个有限的测试用例集 $S$;

步骤 3.2　对 $S$ 中的每一个元素 $s$,如果 oracle 判定 $s$ 是一个症状,则调用逻辑程序诊断工具对导致该症状的错误进行定位和纠正,然后转步骤 1;

步骤 4　如果(满足测试停止条件),返回;

步骤 5　i++;转步骤 2。

实际上,步骤 2 是计算 $T_P$ 最小不动点过程中的一次迭代。对第二步中获得的每一个计算解,必须检查其是否为预期的计算解。一般情况下,一个计算解具有无限多个实例。因此检查一个计算解是否为预期结果是通过检查其代表元素(测试用例)在预期语义中全部成功来完成的,这是一种近似的方法。本框架在生成代表元素的过程中考虑了用户提供的性质规格。这些测试用例在步骤 3.1 产生,检查其是否为预期结果则在步骤 3.2 中完成。一旦发现了一个症状,就调用调试过程对导致出现该症状的错误进行处理。通过这种方式,便避免了对调试过程不必要的调用。

关于该框架,有以下说明。

1) 理论上,测试用例规格的任意不含变量的实例均可作为测试用例。为了从某种程度上消除测试用例生成过程的随机性,我们按照用户提供的性质规格生成测试用例。过程 $p$ 的性质规格规定了过程 $p$ 输入参数的关键性质。以此方式生成的测试用例可同时满足测试用例规格和性质规格。第 5 节中将采用一种形如 CPM 测试帧的性质规格描述格式。

2) 测试用例的生成可以手工也可自动完成。这取决于所采用的逻辑程序计算解语义的性质。第 4 节将介绍一种基于约束的计算解语义,第 5 节将利用该语义对以上框架进行实例化。该语义允许通过使用适当的约束求解器,自动生成测试用例。

3) 该框架假定存在一个逻辑程序诊断过程;给定一个症状,该过程能够对导致该症状的错误进行定位。这里将采用现有的诊断方法,比如文献[8-13]中给出的声明式调试算法,而不是设计新的诊断算法。

4) 通常,程序的计算解语义并非有限可计算,从而也不能够为过程的每一个计算解生成测试用例。所以该框架需要一个停机保证机制,以保证测试和调试过程的中止和程序测试的质量。下面讨论两种机制。首先可以考虑最后一次发现症状到目前所经历的语义计算步骤,比如用变量 *continousSuccNum* 表示。一旦 *continousSuccNum* 达到某个最大值 MAX,测试和调试过程中止。一般地,MAX 越大测试人员对当前测试程序部分正确性的信心就越大。第二种机制则利用测试数据覆盖分析决定到目前为止是否考虑了足够数量的计算解和足够数量的测试用例。对于做何种覆盖分析,可以有多种选择,其中包括基于控制流的路径覆盖分析和子句覆盖分析[6,7]。比如对于后者可以定义:一个测试用例集合覆盖了一个程序,如果这些测试用例的执行能够激活所有的程序子句。依照该定义,类似于 PROTest II 系统中子句覆盖分析器的分析工具可用于该目的。任给测试用例集 $S$,该分析器可计算 $S$ 的覆盖率[5,6]。例如,对于引言中的程序 $P$,测试用例 insert(2,[3,5])覆盖了过程 insert 的第二个子句 $c$,而 sort([5,3],[3,5])则覆盖了 $c$ 之外的所有子句。以该信息为依据,测试人员能够决定何时停止测试和调试过程。

只要给出了逻辑程序的不动点计算解语义,以上框架可通用于逻辑程序相对于计算解规格部分正确性的测试和调试。但与不同语义对应方法的自动化程度却有很大的不同。通过选择合适的计算解语义,框架中的步骤 2 和步骤 3.1 可以自动化。该方法的完全自动化则取决于包含诊断和纠错在内的程序调试过程的自动化。但既然该框架可利用现有的调试工具,则该方法的自动化水平可随着逻辑程序自动调试技术的进展而得到提高。

## 4　计算解语义

本节介绍一种由 Spoto 提出的基于约束的计算解语义[23],并假定这里考虑的 Prolog 程序不包含 cut 操作。该语义将用于说明本文方法的应用。选择该语义的理由有二:其一,该语义是目标独立的指称语义,即程序任意目标的计算解的集合均可由程序语义获得;换句话说,该语义包含了描述在一个程序中执行任意目标可以得到的所有可能计算解的完整信息。其二,该语义使用约束序列集合作为语义域,从而可使用约束求解器自动获得程序的测试用例。本文假定读者熟悉基本的代数结构和 Prolog 语言。序列是元素的有序组合,其中允许存在重复元素。由集合 $E$ 的元素构成的所有序列的集合记作 $Seq(E)$,其中非空序列的集合记作 $Seq^+(E)$。符号"::"表示两个序列的串接操作。空序列记作 $\epsilon$。

### 4.1　Prolog 文法

为了简化语义算子的设计,该计算解语义使用了一种抽象 Prolog 程序文法。其基本思想是把 Prolog 程序视为 CLP 的一个实例[24]。由于二元谓词表达式 $p(X,Y)$可表示为 $\exists Z(Z=(X,Y)\wedge p(Z))$,不失一般性,Prolog 程序中的所有谓词均假定为一元谓词。子句具有以下形式:$p(X)$ :- $G_1$ or…or $G_n$,其中 $n\geqslant 1$,$G_i(i=1,\cdots,n)$被称为过程 $p(X)$的第 $i$ 个定义并由下面文法中的非终结符 $G$ 产生:

$$G::=c\mid p(X)\mid \text{exists } X.G\mid G \text{ and } G$$

其中,$c$ 是一个基本约束(定义见下文),$X$ 是一个程序变量。对任意谓词 $p$,表达式 $p(X)$被称作过程调用。如果目标 $G$ 包含过程调用,则称 $G$ 是发散的。不发散的目标被称为是收敛的。在由 $G$ 开始的 SLD-推导的某个步骤,如果得到的子目标 $G'$是发散的,则称 $G$ 到 $G'$的 SLD-推导路径是发散的。

**定义 1**　基本约束域是一个格$\langle\mathcal{B},\leqslant,\vee,\wedge,true,false\rangle$,且满足以下条件:

1) 对任意变量 $X$ 和 $Z$,$\mathcal{B}$ 中包含元素 $\delta_{X,Z}$,表示变量 $X$ 和 $Z$ 相等;

2) 存在 $\mathcal{B}$ 上的一元运算符 $\exists X$,表示隐藏一个约束中所有与变量 $X$ 相关的信息后所得的约束。

使用运算符 $\exists X$ 是避免 Prolog 目标 SLD-推导过程中重命名问题的一个简单途径。如:把约束 $b$ 中的变量 $X$ 重命名

为 $Z$ 可通过表达式 $\exists X(\delta_{X,Z} \wedge b)$ 来实现。对基本约束域进行扩展可得可见性约束集的概念.

标准 Prolog 可直观地转化为本文的抽象文法。例如：Prolog 程序 P：

$\{p(X)$ :-$q(X), r(X). q(X)$ :-$X=4. q(X)$ :-$X=5. r(X)$ :-$X=5, q(X).\}$

可转化为 $P'$：

$\{p(X)$ :-$q(X)$ and $r(X). q(X)$ :-$X=4$ or $X=5. r(X)$ :-$X=5$ and $q(X).\}$。

### 4.2 语义域

文献[23]计算解语义的基本思想是利用约束描述 Prolog 的控制规则和目标 SLD-推导过程中得到的计算解。基本约束足以描述计算解，例如替换 $(X/2, Y/3)$ 可以简单地用基本约束 $(X=2 \wedge Y=3)$ 来表示。为了描述 Prolog 的控制规则则需要引入可见性约束。

**定义 2** 可见性约束集 $\mathscr{O}$ 是包含集合 $\mathscr{B}$ 且满足以下条件的最小集合：

1) 如果 $S \subseteq \mathscr{O}$，则 $\sqcap S$ 和 $\sqcup S$ 均属于 $\mathscr{O}$；

2) 如果 $o \in \mathscr{O}$ 则 $o$ 的非 $-o \in \mathscr{O}$。

下文中用 $o_1 \sqcap \cdots \sqcap o_n$ 表示 $\sqcap\{o_1, \cdots, o_n\}$，用 $o_1 \sqcup \cdots \sqcup o_n$ 表示 $\sqcup\{o_1, \cdots, o_n\}$，用 $true_\mathscr{O}$ 和 $false_\mathscr{O}$ 分别表示 $\sqcap\{\}$ 和 $\sqcup\{\}$。

可见性约束可视为基本约束的集合，基本约束和可见性约束的可满足性定义如下。

**定义 3** 基本约束 $b$ 在约束集 $S$ 和解释 $I$ 中可满足，当且仅当存在自由变元指派 $\rho$ 使 $\vdash \rho(S \wedge b)$。

**定义 4** 可见性约束的可满足性定义如下：

1) $o \in \mathscr{B}$ 在 $S$ 和 $I$ 中可满足，当且仅当作为基本约束 $o$ 在 $S$ 和 $I$ 中可满足；

2) $o_1 \sqcap o_2$ 在 $S$ 和 $I$ 中可满足，当且仅当 $o_1$ 和 $o_2$ 在 $S$ 和 $I$ 中分别可满足；

3) $o_1 \sqcup o_2$ 在 $S$ 和 $I$ 中可满足，当且仅当 $o_1$ 或 $o_2$ 在 $S$ 和 $I$ 中可满足；

4) $-o$ 在 $S$ 和 $I$ 中可满足，当且仅当 $o$ 在 $S$ 和 $I$ 中不可满足。

可见性约束可视为基本约束的集合。由定义可知，基本约束的可满足性与一阶逻辑中命题的可满足性概念一致，而可见性约束的可满足性则反映了构成该约束的各个基本约束的可满足性。令 $o = X=4 \sqcap X=2, S=\emptyset$，则 $o$ 在 $S$ 中可满足，因为 $X=4$ 和 $X=2$ 在 $S$ 中分别可满足。显然，在以上可满足性定义下，“$\sqcap$”不同于基本约束上的“$\wedge$”操作。可见性约束 $true_\mathscr{O}$ 在约束集 $S$ 和解释 $I$ 中可满足当且仅当 $S$ 在空约束集和 $I$ 中可满足，$false_\mathscr{O}$ 在任意约束集和解释中均不可满足。本文假定基本约束或可见性约束的可满足性可判定。

在约束集 $\mathscr{O}$ 上定义了以下运算：

· 一元运算 $\infty obs$：用于把基本约束转化为可见性约束。为了方便，常把 $b \infty obs$ 写成 $b$，从上下文可判断其为基本约束还是可见性约束。

· 二元运算 · ：用于利用基本约束对可见性约束进一步实例化，其定义如下：$b' \cdot (b \infty obs) = (b' \wedge b) \infty obs$；$b' \cdot \sqcap S = \sqcap\{b' \cdot o \mid o \in S\}$；$b' \cdot \sqcup S = \sqcup\{b' \cdot o \mid o \in S\}$；$b' \cdot (-o) = -(b' \cdot o)$。

· 一元运算 $\exists X(o)$：把约束 $o$ 中的变量 $X$ 转化为约束变量，其定义如下：$\exists X(b \infty obs) = (\exists X(b)) \infty obs$；$\exists X(\sqcap S) = \sqcap\{\exists X(o) \mid o \in S\}$；$\exists X(\sqcup S) = \sqcup\{\exists X(o) \mid o \in S\}$；$\exists X(-o) = -(\exists X(o))$。

为了描述方便，下文在陈述约束的可满足性时不再明确说明对应的解释。直观地，可见性约束被用于描述计算解是如何受到 SLD-推导中发散目标的影响的，因而也可用于确定一个计算解是否是实际可得到的。

在以上定义的基础上可以定义计算解语义的语义域 $\mathrm{Seq}^+(\mathbb{CA})$：

$$\mathbb{CA} = C \wedge C^d$$

其中 $C^c = \{o +^c b \mid o \in \mathscr{O}, b \in \mathscr{B}\}$ 是收敛约束的集合，$C^d = \{o +^d b \mid o \in \mathscr{O}, b \in \mathscr{B}\}$ 是发散约束的集合。收敛约束 $(o +^c b)$ 表示一个由 $b$ 描述的计算解，且该计算解可实际产生仅当 $o$ 是可满足的。发散约束 $(o +^d b)$ 表示 *SLD*-推导过程中得到一个发散的目标，其中 $b$ 描述与该目标对应的替换，$o$ 描述产生发散目标的条件。约束序列 $s \in \mathrm{Seq}^+(\mathbb{CA})$ 中约束的顺序反映了在目标消解过程中不同约束产生的先后顺序。

例 1 已知 Prolog 程序 $P$：$\{p(X)$ :-$(X=4$ and $p(X))$ or $q(X). q(X)$ :- $X=3$ or $X=5.\}$，由目标 $p(X)$ 开始的 SLD-推导将首先产生一个发散约束 $(true_\mathscr{O} +^d X=4)$，然后产生两个收敛约束 $(-(X=4) +^c X=3)$ 和 $(-(X=4) +^c X=5)$。约束 $(true_\mathscr{O} +^d X=4)$ 表示当 $(X=4)$ 可满足时得到发散的目标。可见性约束 $-(X=4)$ 表示仅当 $(X=4)$ 不可满足时才能得到计算解 $X/3$ 和 $X/5$，否则从目标 $p(4)$ 开始的计算就会遇到一个无限 SLD-推导路径，使得以上两个计算解不能够得到。

**定义 5** 下列映射描述了约束序列的性质：

$$\delta(s) = \sqcap\{o \sqcup b \infty obs \mid o +^d b \in s\},$$

$$\xi(s_1, s_2) = \sqcup\{o \sqcap (b \cdot \delta(s_2)) \mid o +^c b \in s_1\}。$$

直观地，如果发散条件 $\delta(s)$ 可满足则 $s$ 描述的 SLD-推导路径是发散的；而条件 $\delta(s_1, s_2)$ 则表示 $s_1$ 中包含一个计算解使 $s_2$ 描述的 SLD-推导路径发散。

### 4.3 计算解语义

首先给出计算解解释的定义。

**定义 6** 计算解解释 $I$ 是一个映射，它为程序中的任一谓词符号 $p$ 关联一个 $\mathrm{Seq}^+(\mathbb{CA})$ 中的元素 $I(p)$，描述从过程调用 $p(\alpha)$ 开始的所有可能 SLD-推导中得到的计算解，其中 $\alpha$ 是一个不允许在任何 Prolog 程序子句中出现的变量。

如果程序 $P$ 中过程 $p$ 的定义为：$p(Y)$ :-$G_1$ or$\cdots$or $G_n$，$I$ 是 $p$ 的当前指称，直承算子 $T_P(I)(p)$ 从 $I$ 中计算 $p$ 的新指称。

$$T_P(I)(p) = \exists Y(\delta_{Y,\alpha} \circledcirc (\mathscr{J}_P[\![G_1]\!](I \oplus \cdots \oplus \mathscr{J}_P[\![G_1]\!]I)),$$

其中 $\mathscr{J}_P[\![G]\!]I$ 是由 $I$ 计算得到的目标 $G$ 的指称，其定义如下：

$$\mathscr{J}_P[\![c]\!]I = (true_\mathscr{O} +^c c),$$

$$\mathscr{J}_P[\![p(X)]\!]I = I(p)[X/\alpha]$$

$$\mathscr{J}_P[\![exists\ X.\ G]\!]I = \exists X \mathscr{J}_P[\![G]\!]I,$$

$$\mathscr{J}_P[\![G_1\ and\ G_2]\!]I = \mathscr{J}_P[\![G_1]\!]I \oplus \mathscr{J}_P[\![G_2]\!]I。$$

**定义 7** $T_P$ 定义中出现的操作符定义如下：

(1) 已知 $b' \in \mathscr{B}$，$b' \circledcirc (s_1 :: s_2) = b' \circledcirc s_1 :: b' \circledcirc s_2$，$b' \circledcirc (o +^c b) = b' \cdot o +^c b' \wedge b$；$b' \circledcirc (o +^d b) = b' \cdot o +^d b' \wedge b$。

(2) $\exists X(s_1 :: s_2) = \exists X(s_1) :: \exists X(s_2)$；$\exists X(o +^c b) = \exists X(o) +^c \exists X(b)$；$\exists X(o +^d b) = \exists X(o) +^d \exists X(b)$。

(3) 已知 $o' \in \mathscr{O}$，$o' \odot (s_1 :: s_2) = o' \odot s_1 :: o' \in s_2$；$o' \odot (o +$

$^c b)=o' \sqcap o+^c b; o' \odot (o+^d b)=o' \sqcap o+^d b$。

(4) $(s_1 :: s_2) \otimes s = s_1 \otimes s :: -\xi(s_1, s) \odot (s_2 \otimes s)$; $(o+^c b) \otimes s = o \odot (b \textcircled{\otimes} s)$; $(o+^d b) \otimes s = o+^d b$。

(5) $s_1 \oplus s_2 = s_1 :: -\delta(s_1) \odot s_2$。

(6) $s[X/\alpha] = \exists_\alpha (\delta_{X,\alpha} \textcircled{\otimes} s)$。

给定一个计算解解释集合上适当定义的偏序关系"$\leqslant$",程序 $P$ 的计算解语义可以定义为 $T_P$ 的最小不动点。最小解释 $I_0$ 定义为:对任意谓词 $p$, $I_0(p)=true_n+^d true$。$T_P$ 的最小不动点定义为:$lfp(T_P)=lub_{i\geqslant 0}((T_P)^i(I_0))$。

**定义 8** 程序 $P$ 的计算解语义 $\mathcal{F}_P$ 定义为 $T_P$ 的最小不动点:$\mathcal{F}_P=lfp(T_P)$。

## 5 测试和调试框架的实例化

### 5.1 测试用例的生成

本节以 Prolog 为例,给出以上逻辑程序测试和调试框架的一个实例。第 4 节中计算解语义的主要特点是 Prolog 被视为约束逻辑程序设计(CLP)语言的一个实例,其中 $\mathcal{B}$ 是约束域。值得说明的是:CLP 语言的每一种实现均以某个约束域 $D$ 为参数(如线性算术约束域、布尔约束域等)并包含一个相应的约束求解器。已知 $D$ 上的约束集合 $C$,该约束求解器可判定 $C$ 的可满足性,并在 $C$ 可满足时给出 $C$ 中变量的一个成真赋值。给定一个由约束和原子目标构成目标 $G$,CLP 引擎能够把该目标归结为一个约束集,并利用内嵌于 CLP 引擎的约束求解器求解该约束集。

比如,已知 CLP 程序:

{sumto(0,0). sumto($N$, $S$) :- $N>1$, $N<=S$, sumto($N-1$,$S-N$).}

目标"$S<=3$,sumto($N$,$S$)"的执行将产生如下计算解:($N=0$,$S=0$),($N=1$,$S=1$),($N=2$,$S=3$)。

在第 4 节的计算解语义中,过程 $p$ 的指称描述了最一般目标 $p(\alpha)$ 的 SLD-推导过程中得到的计算解。收敛约束 $(o+^c b)$ 表示测试用例规格,包含可见性约束 $o$ 和基本约束 $b$。

**定义 9** $p$ 是程序 $P$ 的一个过程,$n$ 是 $p$ 的参数的个数,Prolog 项 $(d_1,\cdots,d_n)$ 被称为过程 $p$ 的一个测试用例,如果 $(d_1,\cdots,d_n)$ 中不包含任何变量。为了方便,也称 $p(d_1,\cdots,d_n)$ 是过程 $p$ 的测试用例。测试用例 $(d_1,\cdots,d_n)$ 满足测试用例规格 $(o+^c b)$,如果 $b$ 和 $o$ 在约束集 $\{\alpha=(d_1,\cdots,d_n)\}$ 中均可满足。

注:这里 $\alpha$ 是一个变量,与第 4 节计算解语义中 $\alpha$ 的使用相对应。

为了从测试用例规格获得测试用例,可以使用约束求解器。假定基本约束 $\mathcal{B}$ 上的约束求解器为 *sol*,按照可见性约束可满足性的定义,*sol* 可用于判定可见性约束的可满足性。利用该约束求解器,测试用例可自动从测试用例规格产生。如果 $\mathcal{B}$ 是项等式、线性算术约束和布尔约束的集合,则 *sol* 类似于 CHIP[25] 和 Prolog III[26] 系统中使用的约束求解器。在此不再讨论该约束求解器的细节。

### 5.2 性质规格及其转化

这里采用与 CPM 测试中的测试帧类似的结构,来定义测试人员关心的过程属性。过程的重要性质用类(category)表示,每一个类又被划分为多个选择(choice)。如果 $P=\{p_1,\cdots,p_m\}$ 表示一个 Prolog 程序,其中 $p_i(i=1,\cdots,m)$ 是过程,过程参数的值域为 $D$,则过程的性质规格可定义如下:

**定义 10** $p$ 是程序 $P$ 的一个过程,$n$ 是 $p$ 的输入参数的个数,$D^n$ 上的等价关系被称作过程 $p$ 的一个类。

**定义 11** $p$ 是程序 $P$ 的一个过程,$c$ 是 $p$ 的一个类,$c$ 的一个等价类被称作一个选择。

**定义 12** $p$ 是程序 $P$ 的一个过程,$\sum_p=\{\delta_1,\cdots,\delta_k\}$ 是过程 $p$ 的类的集合,其中 $\delta_i=\{c_{i1},\cdots,c_{iw}\}$ $(i=1,\cdots,k)$ ($c_{ij}$ 是 $\delta_i$ 的一个选择),笛卡儿集 $(\delta_1\times\cdots\times\delta_k)$ 的一个元素被称作 $p$ 的一个属性帧。$p$ 的属性帧的集合被称作 $p$ 的性质规格。

**定义 13** $f=(c_1,\cdots,c_k)$ 是过程 $p$ 的一个属性帧,Prolog 项 $t=(d_1,\cdots,d_n)$ 满足 $f$,如果 $t\in c_1\cap\cdots\cap c_k$。

下文中,如不特别说明,则认为过程 $p$ 的性质规格覆盖整个输入域 $D^n$。

在本文程序测试中,过程 $p$ 的性质规格 $\mathcal{I}_p$ 提前由用户给出,且每一个属性帧 $f\in\mathcal{I}_p$ 均被转化为一个等价的基本约束 $const(f)\in\mathcal{B}$,其目的是为了能够利用约束求解器产生满足 $f$ 的测试用例。

在 CPM 测试中没有规定描述类及其选择的方式,与此类似,本节方法也没有规定定义过程输入参数性质的方法,因此不能给出把一个属性帧转化为基本约束的通用规则。不过,结合本文采用的抽象文法,则可进一步规定属性帧 $f$ 和基本约束 $const(f)$ 的关系。

令 $f=(c_1,\cdots,c_k)$ 是过程 $p$ 的一个属性帧,$t$ 是一个 Prolog 项,则属性帧 $f$ 和约束 $const(f)$ 满足如下条件:$const(f)$ 在约束集 $\{\alpha=t\}$ 中可满足当且仅当 $t\in c_1\cap\cdots\cap c_k$。

在上述关系下,定义 13 可改写为:

**定义 14** 已知过程 $p$ 的属性帧 $f=(c_1,\cdots,c_k)$,一个 Prolog 项 $t=(d_1,\cdots,d_n)$ 满足 $f$,如果 $const(f)$ 在约束集 $\{\alpha=t\}$ 中可满足。

本文采用的属性帧转化方法与文献[27]中采用的把测试帧转化为基本约束的方法完全相同。下面举例说明。

例 2 已知程序 $P'$,其中第一个子句中包含一个错误:

{insert (X,[Y|U],[Y|V]) :- Y<X,insert(Y,U,V).

insert(X,[Y|U],[X,Y|U]) :- X≤Y. insert(X,[],[X]).},

过程 insert 的性质规格 $\mathcal{I}_{insert}$ 表述如下:

令 $\sum_{insert}=\{\delta_1,\delta_2\}$,其中 $\delta_1=\{d_0,d_1,d_2\}$, $\delta_2=\{e_0,e_1,e_2\}$,其中 $d_0=\{(X,[])\mid X\in D\}$, $d_1=\{(X,[Y])\mid X,Y\in D\}$, $d_2=\{(X,[Y,Z|W])\mid X,Y,Z\in D,W\in D^*\}$; $e_0=\{(X,[])\mid X\in D\}$, $e_1=\{(X,[Y|Z])\mid X,Y\in D,Z\in D^*,X\leqslant Y\}$, $e_2=\{(X,[Y|Z])\mid X,Y\in D,Z\in D^*,X>Y\}$。

则性质规格可定义为:$\mathcal{I}_{insert}=\{(d_0,e_0),(d_1,e_1),(d_1,e_2),(d_2,e_1),(d_2,e_2)\}$,该规格可转化为以下约束:

$const(d_0,e_0)=\exists X,V(\alpha=(X,[],V))$,

$const(d_1,e_1)=\exists X,Y,V(\alpha=(X,[Y],V))$

$\wedge\exists X',Y',Z',V'(\alpha=(X',[Y'|Z'],V')\wedge X'\leqslant Y')$,

$const(d_1,e_2)=\exists X,Y,V(\alpha=(X,[Y],V))$

$\wedge\exists X',Y',Z',V'(\alpha=(X',[Y'|Z'],V')\wedge X'>Y')$,

$const(d_2,e_1)=\exists X,Y,Z,W,V(\alpha=(X,[Y,Z|W],V))$

$\wedge\exists X',Y',Z',V'(\alpha=(X',[Y'|Z'],V')\wedge(X'\leqslant Y')$,

$const(d_2,e_2)=\exists X,Y,Z,W,V(\alpha=(X,[Y,Z|W],V))$

$\wedge\exists X',Y',Z',V'(\alpha=(X',[Y'|Z'],V')\ (X'>Y')$。

注意：过程的性质规格仅规定了过程输入参数的性质，因而不需要对 insert 参数中表示过程输出的变量 $V$ 增加任何限制。已知属性帧 $f$，约束 $const(f)$ 的可满足性可利用 5.1 节中讨论的约束求解器 *sol* 判定。

例如，假定 $D$ 是整数集，输入数据 insert(2,[],$V$)，insert(2,[3],$V$)，insert(3,[2],$V$)，insert(2,[3,5],$V$)和 insert(3,[2,5],$V$)分别满足属性帧$(d_0,e_0)$，$(d_1,e_1)$，$(d_1,e_2)$，$(d_2,e_1)$和$(d_2,e_2)$。根据定义 9 和定义 14，可利用约束求解器获得满足测试用例规格和属性帧的测试用例。

### 5.3 一种综合的测试和调试算法

本节采用 E. Y. Shapiro 的算法诊断技术对 Prolog 程序中的错误进行定位，而错误的纠正则手工完成。利用以上方法，上一节中讨论的 Prolog 程序测试和调试框架可实例化为如下算法，其中采用了一种简单的停机机制。从最后一次发现症状到目前所经历的语义计算的步骤数 *continousSuccNum* 被用于确定中止测试和调试过程的时间。

```
i:=0; continousSuccNum:=0;
DO
{
START: 计算 T_P^(i)(I_0) 并从中获得未在 T_P^(0)(I_0),…,T_P^(i-1)(I_0) 中出
       现的新的测试用例规格(即收敛约束)的集合 D;
   令 D[p]⊆D 表示过程 p 的测试用例规格的集合;
   对任意过程 p,DO
   { 对任意 D_j∈D[p] DO
     { 对任意 I∈J_p,使用约束求解器 sol 生成最多一个满足 I
          和 D_j 的 p 测试用例;
       令 S 是所生成的测试用例集,对任意 s∈S,
       Do
       { 如果 s 是一个症状
           { 调用 Prolog 程序调试工具,纠正程序错误;
             i:=0;continousSuccNum:=0;
             Goto START;
           }
         否则 continousSuccNum ++;
       }
     }
   }
   i++;
} WHILE(continousSuccNum <= MAX)
```

## 6 实例分析

下面的例子说明了 5.3 节中算法的应用。

例 3　以例 2 中程序 $P'$ 为例，并考虑过程的性质规格。程序 $P'$ 被转化为如下程序 $P$：

```
{insert (T)
:- exists X,Y,U,V,S. (T=(X,[Y|U],[Y|V]) and
 (Y<X) and S=(Y,U,V) and insert(S)) or
 exists X,Y,U. (T=(X,[Y|U],[X,Y|U]) and X≤Y)
 or
 exists X (T=(X,[],[X])). }.
```

为了描述方便，采用一个简单的性质规格：

$J_{insert}=\{d_0,d_1,d_2\}$，相应的约束为：

$const(d_0)=\exists X,V(\alpha=(X,[],V))$，

$const(d_1)=\exists X,Y,V(\alpha=(X,[Y],V))$，

$const(d_2)=\exists X,Y,Z,W,V(\alpha=(X,[Y,Z|W],V))$。

测试和调试过程交叉进行如下：

令 $cond=\exists X,Y,U,V,S\ (\alpha=(X,[Y|U],[Y|V])\wedge(Y<X)\wedge S=(Y,U,V))$，则有：

$I_0(\text{insert})=true_\theta+^d true$；

$T_P^{(1)}(I_0)(\text{insert})=true_\theta+^d cond$

$::-cond+^c\exists X,Y,U(\alpha=(X,[Y|U],[X,Y|U])\wedge X\leqslant Y)$

$::-cond+^c\exists X\ (\alpha=(X,[],[X]))$；

从 $T_P^{(1)}(I_0)(\text{insert})$的第二个约束可得如下测试用例：

- (2,[3],[2,3])：满足 $d_1$，
- (2,[3,9,4],[2,3,9,4])：满足 $d_2$，
- 不存在满足 $d_0$ 的测试用例。

从 $T_P^{(1)}(I_0)(\text{insert})$的第三个约束，可得一个测试用例：

- (2,[],[2])：满足 $d_0$，
- 不存在满足 $d_1$ 和 $d_2$ 的测试用例。

然后，oracle 可判定所有这些测试用例均非症状，语义计算继续：

令 $cond=\exists X,Y,U,V,S(\alpha=(X,[Y|U],[Y|V])\wedge(Y<X)\wedge S=(Y,U,V))$，

$cond'=\exists X',Y',U',V',S'(S=(X',[Y'|U'],[Y'|V'])\wedge(Y'<X')\wedge S'=(Y',U',V'))$，则：

$T_P^{(2)}(I_0)(\text{insert})=true_\theta+^d cond\wedge cond'$

$::-(cond\wedge cond')+^c cond\wedge\exists X',Y',U'(S=(X',[Y'|U'],[X',Y'|U'])\wedge X'\leqslant Y')$

$::-(cond\wedge cond')+^c cond\wedge\exists X'(S=(X',[],[X']))$

$::-cond+^c\exists X,Y,U(\alpha=(X,[Y|U],[X,Y|U])\wedge X\leqslant Y)$

$::-cond+^c\exists X(\alpha=(X,[],[X]))$。

$T_P^{(2)}(I_0)(\text{insert})$中的第二、第三个约束是新产生的收敛约束。从第二个测试用例规格可得如下测试用例：

- (4,[3,5,100],[3,3,5,100])：满足 $d_2$，不存在满足 $d_0$ 和 $d_1$ 的测试用例。

Oracle 判定该测试用例是一个症状。以该症状为输入，运行 E. Y. Shapiro 诊断算法可发现第一个子句中错误。通过把 insert($Y,U,V$)改正为 insert($X,U,V$)，可纠正该错误。新程序如下：

```
{insert (T)
 :- exists X,Y,U,V,S. (T=(X,[Y|U],[Y|V]) and (Y<X)
    and S=(X,U,V) and insert(S)) or
    exists X,Y,U. (T=(X,[Y|U],[X,Y|U]) and X≤Y) or
    exists X. (T=(X,[],[X])). }.
```

接下来，重新计算程序的计算解语义，而无需从 $T_P^{(2)}(I_0)(\text{insert})$的第 3 个约束中产生任何测试用例。

令 $cond=\exists X,Y,U,V,S(\alpha=(X,[Y|U],[Y|V])\wedge(Y<X)\wedge S=(X,U,V))$，

$cond'=\exists X',Y',U',V',S'(S=(X',[Y'|U'],[Y'|V'])\wedge(Y'<X')\wedge S'=(X',U',V'))$，则有：

$T_P^{(1)}(I_0)(\text{insert})=true_\theta+^d cond$

$::-cond+^c\exists X,Y,U(\alpha=(X,[Y|U],[X,Y|U])\wedge(X\leqslant Y)$

$::-cond+^c\exists X\ (\alpha=(X,[],[X]))$；

$T_P^{(2)}(I_0)(\text{insert})=true_\theta+^d cond\wedge cond'$

$::-(cond \wedge cond') +^c cond \wedge \exists (X', Y', U' (S=(X', [Y'|U'], [X', Y'|U']) \wedge X' \leqslant Y')$

$::-(cond \wedge cond') +^c cond \wedge \exists X' (S=(X', [], [X']))$

$::-cond +^c \exists X, Y, U (\alpha=(X, [Y|U], [X, Y|U]) \wedge X \leqslant Y)$

$::-cond +^c \exists X (\alpha=(X, [], [X]))$;

从 $T_P^{(2)}(I_0)$(insert)的第二个约束可得下面的测试用例：

• (4,[3,5,100],[3,4,5,100])：满足 $d_2$，不存在满足 $d_0$ 和 $d_1$ 的测试用例。

从 $T_P^{(2)}(I_0)$(insert)的第 3 个约束可产生以下测试用例：

• (4,[3],[3,4])：满足 $d_1$，不存在满足 $d_0$ 和 $d_2$ 的测试用例。

以上两个测试用例均非症状，接下来的测试和调试过程未能发现其他程序错误，算法结束。

**结束语** 本文给出了一种集成的逻辑程序测试和调试框架，用于测试逻辑程序相对于计算解规格的部分正确性。其主要目的是通过减少对调试过程不必要的调用，提高利用逻辑程序开发软件系统的效率。由于逻辑程序设计语言的计算解不动点语义已经得到深入而广泛的研究，因而该框架具有广泛的可用性。利用一种基于约束的 Prolog 计算解语义，该框架被实例化为一个集成的 Prolog 程序测试和调试算法。通过使用适当的约束求解器，测试用例能够自动产生。本文方法的自动化水平可通过使用自动化程度更高的 Prolog 程序调试工具而得到提高。将来的工作包括开发基于本文框架的测试和调试工具，及其在较大规模程序中的应用。

## 参 考 文 献


[1] Comini M, Levi G, Vitiello G. Efficient Detection of Incompleteness Errors in the Abstract Debugging of Logic Programs[C]// Proceedings of AADEBUG 1995. 1995:159-174

[2] Ferrand G. The Notions of Symptom and Error in Declarative Diagnosis of Logic Programs[C]// Proceedings of the First International Workshop on Automated and Algorithmic Debugging, LNCS. 1993, 749:40-57

[3] Ostrand T J, Balcer M J. The Category - Partition Method for Specifying and Generating Functional Tests[J]. Communications of ACM, 1988, 31(6):676-686

[4] Chen T Y, Poon Pak-Lok, Tse T H. A Choice Relation Framework for Supporting Category-Partition Test Case Generation [J]. IEEE Transactions on Software Engineering, 2003, 29(7): 577-593

[5] Luo G, Bochmann G, Sarikaya B, et al. Control Flow Based Testing of Prolog Programs[C]// Proceedings of the 3rd International Symposium on Software Reliability Engineering. 1992: 104-113

[6] Belli F, Jack O. Implementation - based Analysis and Testing of Prolog Programs[C]// Proceedings of the 1993 ACM SIGSOFT International Symposium on Software Testing and Analysis. 1993:70-80

[7] Belli F, Jack O. PROTest II, Testing Logic Programs[R]. 1992/13, ADT, October

[8] Shapiro E Y. Algorithmic Program Diagnosis[C]// Proceedings of the Ninth Annual ACM Symposium on Principles of Programming Languages (POPL'82). 1982:299-308

[9] Lloyd J W. Declarative error diagnosis [J]. New Generation Computing, 1987, 5(2):133-154

[10] Lu L. A Generic Declarative Diagnoser for Normal Logic Programs[J]. LNAI, 1994, 822:290-304

[11] Lu L. Use of Correctness Assertions in Declarative Diagnosis [C]// Proceedings of the 2005 ACM symposium on Applied computing. 2005:1404-1408

[12] Silva J. Algorithmic Debugging Strategies[C]// Proceedings of International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR 2006). 2006:134-140

[13] Silva J, Chitil O. Combining Algorithmic Debugging and Program Slicing[C]// Proceedings of the 8th ACM SIGPLAN symposium on Principles and practice of declarative programming. 2006:157-166

[14] Ducassé M, Noyé J. Logic Programming Environments: Dynamic Program Analysis and Debugging[J]. Journal of Logic programming, 1994, 19-20:351-384

[15] Horváth T, Gyimóthy T, Alexin Z, et al. Interactive Diagnosis and Testing of Logic Programs[C]// Proceedings of the Third Finnish-Estonian-Hungarian Symposium on Programming Languages and Software Tools Kääriku, Estonia. 1993:34-46

[16] Kókai G, Harmath L, Gyimóthy T. IDTS: a Tool for Debugging and Testing of Prolog Programs[C]// Proceedings of LIRA'97, The 8th Conference on Logic and Computer Science. 1997:103-110

[17] Comini M, Levi G, Vitiello G. Abstract Debugging of Logic Program[C]// Proceedings of the International Workshop on Meta-Programming in Logic (META). 1994:440-450

[18] Comini M, Levi G, Vitiello G. Declarative diagnosis revisited[C]// Proceedings of International Logic Programming Symposium. 1995:275-287

[19] Hermenegildo M, Puebla G, Bueno F, et al. Integrated Program Debugging, Verification, and Optimization Using Abstract Interpretation (and the Ciao system preprocessor) [J]. Science of Computer Programming, 2005, 58(1/2):115-140

[20] Dershowitz N, Lee Y. Deductive Debugging[C]// Proceedings of the Fourth IEEE Symposium on Logic Programming. 1987:298-306

[21] Lu L, Greenfield P. Logic Program Testing Based on Abstract Interpretation[C]// Proceedings of the International Conference on Formal Methods in Programming and Their Applications, LNCS. 1993, 735:170-180

[22] Gabbrielli M, Levi G, Meo M C. Observable Behaviors and Equivalences of Logic Programs[J]. Information and Computation, 1995, 122(1):1-29

[23] Spoto F. Operational and Goal-independent Denotational Semantics for Prolog with Cut[J]. The Journal of Logic Programming, 2000, 42:1-46

[24] Jaffar J, Maher M J. Constraint Logic Programming: A survey [J]. Journal of Logic Programming, 1994, 19-20:503-581

[25] Dincbas M, Van Hentenryck P, Simonis H, et al. The Constraint Logic Programming Language CHIP[C]// Proceedings of the International Conference on Fifth Generation Computer Systems. 1988:693-702

[26] Colmerauer A. An Introduction to PROLOG-III[J]. Communications of the ACM, 1990, 33(7):69-90

[27] Zhao Lingzhong, Gu Tianlong, Qian Junyan. Test Frames Updating in CPM Testing of Prolog Programs[J]. Software Quality Journal, 2008, 16(2):277-298