Acta Cybernetica 15 (2001) 257–278.

The Debug Slicing of Logic Programs*

Gyöngyi Szilágyi†, László Harmath† and Tibor Gyimóthy†

Abstract

This paper extends the scope and optimality of previous algorithmic debugging techniques of Prolog programs using slicing techniques. We provide a dynamic slicing algorithm (called Debug slice) which augments the data flow analysis with control-flow dependences in order to identify the source of a bug included in a program.
We developed a tool for debugging Prolog programs which also handles the specific programming techniques (cut, if-then, OR). This approach combines the Debug slice with Shapiro's algorithmic debugging technique.

1 Introduction

Slicing methods are widely used for the debugging [25], testing [2] and maintenance of imperative programs [1, 12]. Intuitively, a slice should contain all those parts of a program that may affect the variables in a set V at a program point p [26]. Slicing algorithms can be classified according to whether they only use statically available information (*static slicing*), or compute those program points which influence the value of a variable occurence for a specific program input (*dynamic slice*). Dynamic slicing methods are more appropriate for debugging than static ones as during debugging we generally investigate the program behaviour under a specific test case. The main advantage of using a dynamic slice during debugging is that many statements can be ignored in the process of bug localization.

Different dynamic slicing methods have been introduced for debugging imperative programs [23]. Most of these methods are based on a dependence graph which contains the explicit control dependences and data dependences of the program. In [9, 14] a slicing method was introduced for logic programs, and this method being used to improve the efficiency of the Shapiro's algorithmic debugging algorithm [19]. The slice presented in [9] contains those parts of a program that actually have an influence on the value of an argument of a predicate. This type of slice (called *data flow slice*) is safe if the structure of the proof tree for a goal is not changed [22].

*This work was supported by the grants OTKA T52721, and IKTA8/99.

†Research Group on Artificial Intelligence Hungarian Academy of Sciences Address: 6720 Hungary, Szeged, Aradi vértanuk tere 1. E-mail:{szilagyi, harmat, gyimi}@inf.u-szeged.hu

However, during debugging to find a source of a bug (i.e. a bug instance) we also need to identify those predicates that actually did not affect an argument in a predicate but could have affected it had they been evaluated differently (had their boolean outcome been different). We can say that these predicates are in the Potentially Dependent Predicate Set. Note that a different evaluation of the predicates in this set could change the success branch of the SLD-tree (where the bug was manifested).

Consider the following example.

Example 1 *The buggy program is:*

1. p(A,X) :- q(A,X).
2. q(A,X) :- **A > 0**, *X is 2.*
3. q(A,X) :- X is 3.

The correct program should be:

1. p(A,X) :- q(A,X).
2. q(A,X) :- **A = 0**, *X is 2.*
3. q(A,X) :- X is 3.

Executing this program for the goal $p(0, X)$ the given solution is $X = 3$, while we expect $X = 2$. So a bug must be included in the program somewhere.
Creating the dynamic data flow slice for an instance of X, it does not contain the buggy predicate $A > 0$ because X does not exactly depend on the predicates of clause 2, there being only control dependences between them. This means that if $A > 0$ had been evaluated differently it could have affected the solution of X. Our new slicing approach contains the buggy predicate $A > 0$ (see Section 4.2).

In this paper we introduce a new type of slicing called *Debug slicing* for Prolog programs without side effects. A Debug slice of an Augmented SLD-tree includes those predicates that may affect the value of an argument in any success branch's predicate. So this slice is very suitable for debugging. The Debug slice is the set of predicates which contains the Potentially Dependent Predicates and their data dependences.

This slicing method has been integrated into an interactive algorithmic debugging tool to reduce the number of questions to the user during a debugging session [14]. The size of the debug slice is larger than the size of the data flow slice, but the data flow slice is not safe for debugging. On the other hand the Debug slice contains all parts of the program that may be responsible for the incorrect behaviour at some selected argument position.

In the next section the basic concepts of logic programming, algorithmic debugging and slicing are presented. Section 3 then provides a detailed description of the construction of those structures needed in an outline of the Debug slice algorithm (Augmented SLD tree, Skeleton(n), (Directed) Proof Tree Dependence Graph, General Data Flow Slice). The computation of the Debug slice on the basis

of these structures is described in Section 4. The first results of a prototype implementation of Debug slice algorithm are discussed in Section 5. Finally, in Section 6 we summarize related work and outline further studies.

2 Preliminaries

In this section we present some basic concepts (logic programming, algorithmic debugging, slicing) needed to outline the Debug slicing algorithm.

2.1 Logic Programming

A **first order alphabet** consist of variables, predicate symbols and function symbols (which include constants) [24].

A **variable** is represented by an upper case letter followed by a string of lower case letters and/or digits.

A **function symbol** is a lower case letter followed by a string of lower case letters and/or digits.

The **constants** include integers and atoms, a constant is a function symbol of arity 0. The symbol for an **atom** can be any sequence of characters.

A variable is a **term**, and a function symbol followed by bracketed n-tuple of terms is a (compound) term. Thus $f(g(X), head)$ is a term when f, g and *head* are function symbols and X is a variable.

A predicate symbol immediately followed by a bracketed n-tuple of terms is called an **atomic formula**, or atom.

Let $h, a_1, \cdots, a_m$ be atomic formulae for some $m \geq 0$ and let $X_1, \cdots, X_l$ be all variables occurring in these formulae.

Then the formula $\forall X_1 \cdots \forall X_l(h \leftarrow a_1, \cdots, a_m)$ is called a **definite clause**. If $m = 0$ the formula is called a **fact**. The atomic formula h is called the *head* of the clause, while $a_1, \cdots, a_m$ is called its *body*. A **goal** is a definite clause with empty head. Since all variables of a definite clause are universally quantified we can omit the quantifiers.

A clause or an atom is **ground** if it has no variable.

A **normal program** is a set of program clauses.

A **substitution** is a finite set (possible empty) of pairs of the form $X \rightarrow t$, where X is a variable and t is a term and all the variables X are distinct. For any substitution $\sigma = \{X_1 \rightarrow t_1, \cdots, X_n \rightarrow t_n\}$ and term s, the term $s\sigma$ denotes the result of replacing each occurrence of the variable X_i by t_i $(i = 1, \cdots, n)$. The term $s\sigma$ is called an **instance** of s.

A substitution σ is called a **unifier** for two terms s_1 and s_2 if $s_1\sigma = s_2\sigma$. Such a substitution is called **the most general unifier** of s_1 and s_2 if for any other unifier σ_1 of s_1 and s_2, $s_1\sigma_1$ is an instance of $s_1\sigma$. If two terms are unifiable then they have uniquie most general unifier.

A **computation of a logic program** P [19] can be described informally as follows. The computation starts from some initial goal g and can have two results:

success or failure. If a computation succeeds, then the final values of the variables in g are considered of as the output of the computation. A given goal can have several successful computations, each resulting in a different output.

The computation progresses via nondeterministic goal reduction, at each step we have some current goal $G = g_1, \cdots, g_n$. A clause $C = a \leftarrow b_1, \cdots, b_k$ in P is then chosen nondeterministically; the head of the clause a is then unified with g_1, say, with substitution σ, and the reduced goal is $G' = (b_1, \cdots, b_k, g_2, \cdots, g_n)\sigma$. The computation terminates when the current goal is empty. Then G' is said to be *derived* from G and C.

Let P be a logic program and G a goal. A *derivation* of G from P is a possible infinite sequence of triples $< G_i, C_i, \sigma_i >$, $i = 0, 1, \cdots$ such that G_i is a goal, C_i is a clause in P with new variable symbols not occuring previously in the derivation, σ_i is a substitution, $G_0 = G$, and G_{i+1} is derived from G_i and C_i with substitution σ_i, for $i \geq 0$. If there is a derivation of G from P such that $G_l = \Diamond$ (the empty goal) for some $l \geq 0$ we say that P *succeeds* on G. We assume by convention that in such a case $C_l = \Diamond$ and $\sigma_l = \{\}$.

2.2 Algorithmic Debugging of Logic Programs

Algorithmic debugging is a process where the user and a debugging system interactively try to locate the source of an externally visible bug in the program [19]. There are a number of features of Prolog program which distinguish it from other programming languages. A Prolog program can have both a declarative and a procedural reading, and may or may not be multi-directional and even it can be nondeterminate. The computation model of Prolog is based on goal invocation as well as goal success and failure. Thus errors in Prolog programs occur when, for example, they finitely fail on goals that should succeed, or succeed on goals that should fail.

A *bug manifestation* is undesired program behaviour, i.e. an undesired sequence of solutions computed by the program for a goal. A *bug instance*, which is a predicate instance, is a cause for a top goal bug manifestation. Our algorithm identifies a set of predicates of a program which can cause the bug mainfestation (i.e. an undesired solution with respect to a variable of the top goal).

Shapiro's algorithm [19] traverses the proof tree of a program in different ways and asks the user about the expected behaviour of each resolved goal. The *bottom-up method* traverses the proof tree in postorder manner and asks the oracle about the correctness of the computed values of the nodes. If the result at a node is incorrect and all sons of this node are evaluated correctly the algorithm identifies the clause applied to this node as a buggy one. The query complexity of this method is linear in the size of the tree.

The second method investigates the nodes in a *top-down* manner. If the result computed at a node is evaluated correctly by the oracle then the algorithm does not visit the nodes inside the sub-tree. Using this approach the query complexity can be reduced to a linear dependence in the depth of the proof tree.

The most efficient technique is the *divide-and-query* startegy which requires a num-

ber of queries logarithmic in the size of the proof tree. The divide-and-query algorithm splits the proof tree into two approximately equal parts, and makes a query for the node at the splitting point. If this node gives an incorrect evaluation the algorithm goes on recursively to the sub-tree associated with this node. If the node's answer is correct its sub-tree is removed from the tree and a new mid-point is computed.

A Prolog program may use a number of programming techniques specific to Prolog (cut, if-then, OR). We developed a tool for debugging Prolog programs incorporating these specific programming techniques as well for finding the source of a bug, i.e. for identifing a bug instance.

2.3 Slicing

Slicing is a program analysis technique originally developed for imperative languages [26]. Later improvements are presented in [23, 21, 15, 13].
Intuitively a program slice with respect to a specific variable V at some program point p (which can be a variable or an argument position of a predicate) contains all those program points that may affect the value of the variable or may be affected by the value of the variable. The tuple $< V, p >$ is called a *slicing criterion* and a slice is computed with respect to one.
Slicing techniques can also be classified into *static* and *dynamic* ones.
Static slicing is based on an analysis of the program without executing it so it may be imprecise if it contains data flow which is actually not manifested during a particular execution.
Dynamic slicing is based on the program's execution and hence extracts the precise data flow. A dynamic slice may be different for each execution and so shall always be produced separately whenever the program run.
In addition, slicing can be classified into *forward* and *backward* types. Suppose our slicing criterion is $< V, p >$. Forward slicing with respect to $< V, p >$ contains all those program points that may have its value modified if $< V, p >$ is modified. Backward slicing contains all those program points which, if modified, might change the value of V.
In the case of imperative languages a possible program representation for the program's dependences is the Program Dependence Graph [7, 16, 11].

The problem of slicing logic programs is more complicated than for the imperative case. Before a slice over a logic program can be produced, its implicit data flow has to be approximated. To approximate the data flow the implicit input/output data dependences have to be extracted from the program.
Program slicing has been widely studied for imperative programs [23], but research on slicing logic programs is just beginning. To our knowledge, only a few papers deal with the problem of slicing logic programs [9, 20, 27, 22].
The main contribution of this article was to furnish a dynamic slicing algorithm which augments the data flow analysis with control-flow dependences so as to make the slicing algorithm better suited for debugging.

3 Basic structures and theorems for constructing the Debug Slice

Our slicing is based on a dependence based approach. In [9] a Dependence Graph was constructed for a proof tree i.e. for a success branch of the SLD-tree. We would also like to extend this definition to the failure branches of the SLD tree. This is why this section provides a detailed description of the necessary structures needed to outline this extension: the Augmented SLD-tree, Skeleton(n) (a modified derivation tree), (Directed) Proof Tree Dependence Graph (PTDG) and General data flow slice. The computation of the Debug slice on the basis of these structures is described in the next section.

The *Augmenetd SLD-tree* shows the execution order of the statements for a given input.

Skeleton(n) is always built for one branch of the Augmented SLD-tree, and its nodes represent the data flow information needed for preparing the *Proof Tree Dependence Graph.*

A *General slice* of a logic program with respect to a variable V is constructed using the Proof Tree Dependence Graph, which contains those predicates of a derivation that may affect the value of V.

3.1 The Augmented SLD-tree of Logic Programs

The derivation of a goal from a program P can be represented by a tree called SLD-tree. Each branch of the SLD-tree [17] is a derivation of a program for a goal. Branches corresponding to successful derivations are called *success branches*, while branches of the infinite derivations are called *infinite branches*; those corresponding to failed derivations are called *failure branches.* The Prolog interpreter searches the SLD-tree to find success branches. The Prolog system always selects the leftmost atom in a goal along with a depth-first search rule. The program clauses are then tested in their original order in the program.

An SLD-tree may have many failed branches and very few or just one success branch. Control information supplied by the user may prevent the interpreter from construction of failed branches. To control the search the concept of *cut*(!) is introduced in Prolog. The atom "!" is handled as an ordinary atom in the body of a clause. When a *cut* is selected for resolution it succeeds immediately (with the empty substitution) [18]. The node where *cut* is selected will be called the *cut node.* A cut node may be reached again during backtracking. In this case the normal order of tree traversal is altered - by definition of *cut* the backtracking continues above the node origin (!). (If *cut* occurs in the initial goal, the execution simply terminates). So *cut* has the following effect: after success of "!" no backtracking to the literals in the left-hand part is possible. However, in the right-hand part execution goes on as usual.

We add these pieces of information to the SLD-tree, identify each node with an unique mark, and use a list ($pred\_def\_ref()$) in order to know which program

clauses (corresponding to the selected predicate) are used at a node to execute the next step. We also deal with the pruning effect of cuts. The following definition provides a formal description of the modified SLD-tree. The node label contains the whole list of goals $((G', R) = (a_1, a_2, \cdots, a_n))$. The actual goal (G') is the first in this list (a_1 in our case). A node has a child for every program clause whose head (h_m) could be unified with the actual goal. The list of these clauses for every node is given in *pred_def_ref()* (Definition 1.1). If the actual goal were *cut(!)*, the corresponding branches of the tree would be pruned (Definition 1.2 and 1.3).

Definition 1 *Let P be a Prolog program and G a goal. An* **augmented SLD-tree** *for $P \cup \{G\}$ is a tree which satisfies the following:*

- Each node label is triple $< Mark, (G', R), pred\_def\_ref(G') >$, where $Mark$ is a *unique identification* of the node, (G', R) is a (possibly empty) conjunction of goals (*resolvent*). G' is the leftmost goal in the resolvent (called *selected goal*) and $pred\_def\_ref(G')$ is a *reference list* for the predicate definitions corresponding to the leftmost goal G'. We assign the empty resolvents with a true value.

- The root node is $< Mark, (G, true), pred\_def\_ref(G) >$.

- Let $< M, (a_1, a_2, \cdots, a_k), pred\_def\_ref(i_1, \cdots, i_l) >$ be a node in the tree (so a_1 is the selected atom), where $i_m (m = 1, \cdots, l)$ is the identity number of input clauses $h_m \leftarrow b_{m_1}, \cdots, b_{m_q}$ such that a_1 and h_m are unifiable with most general unifier σ.

  1. Then this node has a child
     $< M_{i_m}, (b_{m_1}, b_{m_2}, \cdots, b_{m_q}, a_2, \cdots, a_k)\sigma, pred\_def\_ref(b_{m_1}) >$ for each $i_m (m = 1, \cdots, l)$. The edges immediately below a node and also the $pred\_def\_ref()$ list are ordered from left to right, according to the program clause order.

  2. If $h_m \leftarrow b_{m_1}, \cdots, b_{m_q}$ has cuts the child is
     $< M_{i_m}, (b'_{m_1}, b'_{m_2}, \cdots, b'_{m_q}, a_2, \cdots, a_k)\sigma, pred\_def\_ref(b'_{m_1}) >$,
     where $b'_{m_1}, \cdots, b'_{m_q}$ are obtained from $b_{m_1}, \cdots, b_{m_q}$, replacing all cuts with one same unique annotation such as "cut(M)", where M is the node's identification mark.

  3. If the following selected atom (b'_{m_1} or a_2) were cut(M), we use the pruning rule below, and the next element of the list that follows cut(M) will be the selected goal.
     The pruning rule: If "Mark" is the argument of cut(), consider the path W from the actual node up to the node marked Mark. All descendants of this node to the right of W are removed.

- Nodes with an empty R list in resolvent have no children.

We will refer to the Augmented SLD-tree simple as SLD-tree. During the execution of a program for a goal to find the first success branch of the SLD-tree, only a part of it is walked by the Prolog interpreter. We will call this part of the SLD-tree the **Trace-tree** because we can build the Trace-tree from the trace of a program for a goal given by the interpreter. Figure 1 shows the Trace-tree of the program in Example 2 for the goal $a(Y)$.

Example 2 *We now illustrate our definition of the Trace-tree in a simple example:*
1. a(Y):-b(X), c(Y), d(X).
2. c(Y):-b(Y), e(Y).
3. b(1).
4. b(2).
5. b(3).
6. d(2).
7. e(3).

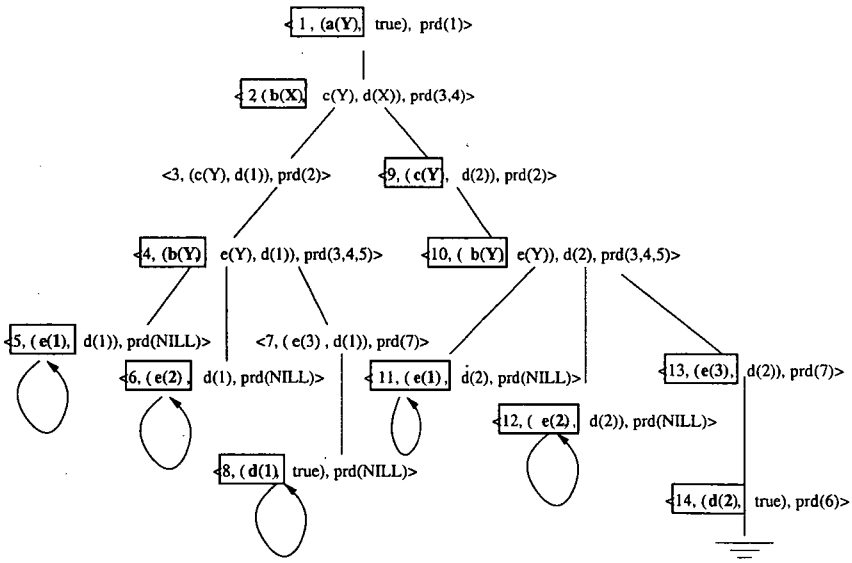

Figure 1: The Trace-tree for the goal a(Y) and the Debug slice in frames (see Section 4.2).

Example 3 *Figure 2 shows the pruning effect of the cuts in the following example for the goal $a(X)$.*
1.a(X) :- b(X), c(X).
2. a(X) :- g(X).
3. b(X) :- c(X), !, d(X).
4. b(X) :- e(X), h(X).
5. c(1).

6. c(2).
7. d(2).
8. e(1).
9. h(1).
10.g(3).

The removed part of the Trace-tree is depicted by a broken line (the pruning effect of the cut).

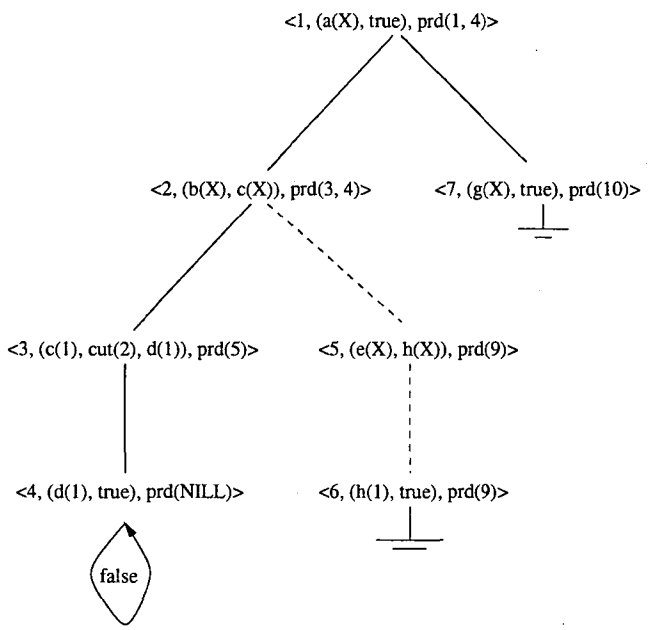

Figure 2: The pruning effect of the cut in Example 3 for the goal a(X).

3.2 Skeleton(n)

The SLD-tree representation is unsuitable for representing the data flow information of a logic program (for a given goal). The structure $Skeleton(n)$ [24] is used to represent this information, where n identifies a leaf node of the SLD-tree. $Skeleton(n)$ is basically a derivation tree defined for one branch of the SLD-tree (from the root to the node marked by n). To improve the approximation of the implicit data flow $Skeleton(n)$ contains directionality information as well.

We will use the notion of *clause instance* ($c\sigma$) which means that a substitution σ is applied for every predicate of c.

We extended the definition of the derivation tree used in [24] to our case.

Definition 2 Proof Tree
For a program P a **proof tree** *is any labeled, ordered tree T such that*

1. *Every node is labeled by an instance name of a clause of P.*
2. *Let n be a node labeled by an instance name $< c, \sigma >$, where c is the clause $h \leftarrow a_1, \cdots, a_m$ $(m \geq 0)$ and σ is a substitution. Then n has m children, for $i = 1, \cdots, m$, the i-th child of n being labeled $< c_i', \sigma_i' >$, where c_i' is a clause with head h_i' such that $a_i\sigma = h_i'\sigma_i'$.*

The reasoning behind this definition is that every tree is obtained by combining appropriate instances of program clauses. The precise meaning of "appropiate instances" is expressed in condition 2. A logic program defines a set of derivation trees. This may be viewed as a semantic of definite programs, and can be related to the concepts of proof defined in symbolic logic.
A derivation tree represents one branch of the SLD-tree (one derivation), but in a more suitable format for representing the data flow.
Let us modify this definition to suit our present needs.
We need not know exact the substitution itself, it is enough to know which variables are ground at call of a predicate and which are ground at success. Basically having to investigate directionality information of the tree using some groundness annotation.
Let us suppose that we can identify each argument position of the clauses of a derivation tree with a tuple (the formal definition of the argument position is given at the end of this subsection).

Groundness information associated with a derivation tree will be expressed as an annotation of its argument positions. The annotation classifies the argument positions of a derivation tree. The positions are classified as *inherited* (marked with $\downarrow$), *synthesized* ($\uparrow$) and *dual* ($\updownarrow$). An annotation is *partial* if some positions are dual. Formally speaking, an annotation is a mapping ν from the positions in the set $\{\downarrow, \uparrow, \updownarrow\}$ [6].

The intended meaning of the annotation is the following. An **inherited argument position** is a position in which every variable is ground at time of calling, that is when the equation involving this position is first created during the construction of the derivation tree. A **synthesized argument position** is a position in which none of the variables are ground at time of calling, and every variable is ground at success, that is when the subtree having the position in its root label is completed in the computation process. The **dual argument positions** of a proof are those which are neither inherited nor synthesized. The annotations are collected during the execution of a program for a given goal. We notice that the argument positions are annotated at the present version of our tool. But the annotation of the variable positions would provide more precise dependences so we are planning to extend the annotation to variable positions.
We now introduce the following auxiliary terminology relevant to the annotated positions of a LP program. The inherited positions of the head atoms and the synthesized positions of the body atoms are called **input positions**. Similarly, the

synthesized positions of the head atoms and inherited positions of the body atoms are called **output positions**. The others are **dual**. Note that dual positions are not strictly classified as input or output ones. Alternatively, if we say that a position is annotated as an output we mean that it is annotated as inherited provided it is a position in a body atom, or annotated as synthesized if it is a position of the head of a clause.

Now we are ready to define $Skeleton(n)$.

Definition 3 Skeleton(n)
Let T be a SLD-tree, $< n, (G', R), pred\_def\_ref(G') > \in nodes(T)$ a leaf node identified by n. Consider the path W in T from the root to n, which identifies a derivation for the root goal.
Then, $Skeleton(n)$ is a labeled ordered tree such that

1. *Every node is labeled by a double $< Mark, \nu(c) >$, where Mark is an unique identification, and $\nu(c)$ is the annotated clause instance.*

2. *The root node is labeled by $< 1, \nu(c) >$, where the root goal was unified with c during the given derivation.*

3. *Let k be a node labeled by*
   $< Mark, p([X_1, \nu(X_1)], \cdots, [X_{k_0}, \nu(X_{k_0}])) :-$
   $a_1([Y_1, \nu(Y_1)], \cdots, [Y_{k_1}, \nu(Y_{k_1})]), \cdots, a_m([V_1, \nu(V_1)], \cdots, [V_{k_m}, \nu(V_{k_m}])) >$.
   Then k has m children, for $i = 1, \cdots, m$, the i-th child of k being labeled $< Mark, \nu(c'_i) >$, where c'_i is a clause whose head was unified with a_i during the given derivation.

Figure 3 shows Skeleton(5) of Example 2. The variable Y of $a(Y)$ in node 1 is annotated as output, since it would be ground at success of $a(Y)$, and $a(Y)$ is a head atom. For the same reason X in node 2, Y in node 3 and in node 4 are annotated as output. The variable Y in node 5 is ground at call, so it is annotated as input.

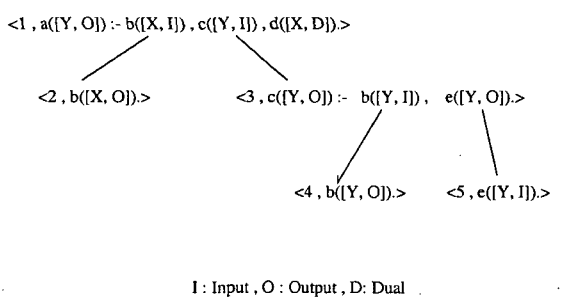

Figure 3: Skeleton(5) for Example 2.

We can refer to the k-th *argument position* in each node of the skeleton by the tuple $(Mark, i, k, V)$, where $Mark$ is the identity number of the node, i is the number of the predicate in the clause (from 0 to m), k is the argument position of a_i and V is the set of variables at this position. If V also contains Input and Output variables it is annotated as Dual.
Denote $Pos(S)$ the set of argument positions of the Skeleton(n) S.

As can be seen from the definition of the Augmented SLD-tree and $Skeleton(n)$, there is an one to one correspondence between the nodes belonging to one branch of the SLD-tree (identified by n) and the nodes of the corresponding $Skeleton(n)$. This correspondence is based on the fact that both structures describe the same derivation for a goal step by step. In our formalism the Mark of a node highlights this correspondence.

Let T be an SLD-tree for the goal g, $n \in nodes(T)$ a leaf of T, and S the $Skeleton(n)$. Then, there is a map from the nodes of S to the nodes of T such that:
$\phi : nodes(S) \rightarrow nodes(T)$
$< Mark, \nu(p : -a_1, \cdots, a_m) > \rightarrow < Mark, (p, R), pred\_def\_ref(p) >$.
If $S' \subseteq nodes(S)$ then denote $\phi(S')$ the corresponding subset of $nodes(T)$ such that $\phi(S') = \{\phi(n) | n \in S'\}$.
For $n \in nodes(T)$ let $\phi^{-1}(n) = m \in Pos(S)$, such that $\phi(m) = n$.

Now we are ready to define the Proof Tree Dependence Graph.

3.3 Proof Tree Dependence Graph

We would like to represent the data flow of a derivation tree. In a logic program data can be transferred in two ways: firstly from one clause to another via unification, and secondly within a clause multiple occurrence of variables result in data dependences [3, 4]. The following definition reflects these conditions.

Definition 4 Proof Tree Dependence Graph (PTDG: $T_{g,n} = (Pos(S), \sim_T)$)
Let T be an SLD-tree for the goal g, $n \in nodes(T)$ a leaf of T and S the Skeleton(n), $\beta, \delta \in Pos(S)$.

- *The nodes of PTDG are the elements of $Pos(S)$.*
- *$\beta \sim_T \delta$ iff one of the following conditions holds:*
  1. *β and δ have common variable in their variable set V (**local edge**)*
  2. *the predicate of δ was unified with the predicate of β, and β and δ are both the k-th argument position of their predicate (**transition edge**).*

It follows directly from the definition that the dependence graph is constructed only for one branch of the SLD-Tree (identified by n), for Skeleton(n). But of course we can construct a PTDG for every Skeleton(n) (n is a leaf node of T), that is for every branch of the SLD-tree T. Figure 4 shows the PTDG for $Skeleton(5)$.

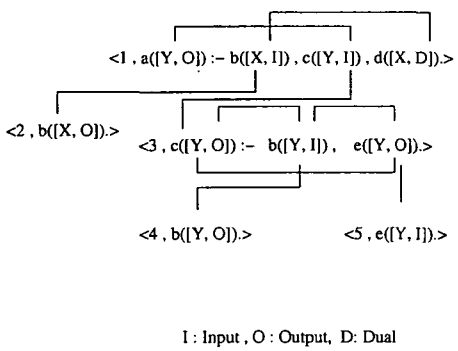

Figure 4: The Proof Tree Dependence Graph for $Skeleton(5)$

3.4 Directed Proof Tree Dependence Graph

As mentioned earlier we would like to better approximate the implicit data flow by introducing directionality using an annotation technique. The annotations can be collected during the execution of the program. Based on this annotation the Proof Tree Dependence Graph can be directed because the data flows from an Input position to an Output one via a local edge, and from an Output position to an Input one via an transition edge. This can be expressed more precisely in the following definition.

Definition 5 Directed Proof Tree Dependence Graph
Let $T_g = (Pos(S), \sim_T)$ be a proof tree dependnece graph, $\alpha, \beta \in Pos(S)$. Then the **directed proof tree dependence graph** *is $\vec{T}_{g,n} = (Pos(S), \rightarrow_T)$, where*

1. *$\alpha \rightarrow_T \beta$ if $\alpha \sim_T \beta$, $\sim_T$ is a local edge and one of the following conditions holds:*
   - *α is an Input position and β is an Output position*
   - *α is a Dual position and β is an Output position*
   - *α is an Input and β is a Dual position*
   - *α is a Dual and β is a Dual position (in this case $\alpha \rightarrow_T \beta$ and $\beta \rightarrow_T \alpha$)*
2. *$\alpha \rightarrow_T \beta$ if $\alpha \sim_T \beta$, the positions being connected by a transition edge and satisfying one of the following conditions:*
   - *α is an Output position and β is an Input position*
   - *α is a Dual and β is a Dual position (in this case $\alpha \rightarrow_T \beta$ and $\beta \rightarrow_T \alpha$)*

It is quite easy to check the validity of these rules. It is possible to define more precise conditions to direct the edges (referring to the textual occurrences of the positions), but it would be too complicated to present them here. Our experience shows that the use of these rules (which permit in some cases non realisable data flow) gives good results. Our slicing algorithm applies to this Directed Proof Tree Dependence Graph. The Directed Proof Tree Dependence Graph for Skeleton(5) is depicted in Figure 5.

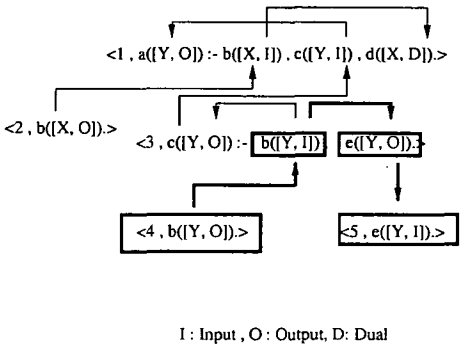

Figure 5: The Directed Proof Tree Dependence Graph and the data flow slice (see Section 3.5) with respect to $(5, 0, 1, \{Y\})$

3.5 General Data Flow Slice

In this section a *general slice definition* is given, which shows that a given argument position of Skeleton(n) which other argument positions depends on (from the aspect of data flow).

Definition 6 Slice$(T_{g,n}, \alpha)$
Let P be a logic program, T a SLD-tree for the goal g, $n \in nodes(T)$ a leaf of T, S Skeleton(n) and $\vec{T}_{g,n} = (Pos(S), \rightarrow_T)$ the corresponding Directed Proof Tree Dependence Graph. Let $\alpha \in Pos(S)$.
A **slice** *$(T_{g,n}, \alpha)$ over $\vec{T}_{g,n}$ with respect to α*

- *is a subgraph of $\vec{T}_{g,n}$*
- *a node $\beta \in Pos(S)$ is in the slice iff $\beta \rightarrow_T^* \alpha$*

This is a general data flow slice definition that is valid for one derivation path of the SLD-tree which may be a failed branch.

Figure 5 shows $slice(T_{a(y),5}, (5, 0, 1, \{Y\}))$. The argument of $e(Y)$ in node 5 $((5, 0, 1, \{Y\}))$ is an Input position, the slice being constructed with respect to this position. The set of all positions from which there is a directed path to this argument position (this is the slice with respect to $(5, 0, 1, \{Y\})$) contains the argument position of $e(Y)$ and $b(Y)$ in node 3, and $b(Y)$ in node 4.

The data flow slice given in [9] is a special case of this slice definition, which is defined only for the success branch of the SLD-tree.

4 Debug Slice of Logic Programs

As mentioned before our aim is to combine our slicing technique with an algorithmic debugging tool [14] in order to locate the source of a bug with fewer user interactions by avoiding posing unnecessary questions. In [9] a data flow slice was defined for the success branch of the SLD-tree (Trace-tree). We extended the data flow slice definition to the full SLD-tree (Trace-tree), based on the Directed Proof Tree Dependence Graph.

To take care of the control flow mentioned in Subsection 4.1 we specify the *Potentially Dependent Predicates Set (PDPS)* which contains the predicates that actually did not affect the selected argument, but could have done so had they been evaluated differently (i.e. had they succeeded or failed).

Lastly in Subsection 4.2 the *Debug slice* is defined on the Augmented SLD-tree (Trace tree) which includes the Potentially Dependent Predicates, their associated data dependences and the predicates affected by some cut.

4.1 Potentially Dependent Predicates

Sometimes we cannot find the source of a bug just by analyzing the data flow for the success branch of the Trace-tree. So we have to examine which predicates might cause a branch of the Trace-tree to fail, or what would have happened if a predicate had succeeded but actually failed, or if it should have failed but actually succeeded. We concentrated on the leftmost (goal) predicate of an SLD node so the slice is defined for these predicates. The following definition covers these cases.

Definition 7 Potentially Dependent Predicate (PDPS)
Let P be a logic program, T the Trace-tree for the goal g. A leftmost (selected) predicate in a node of T is in the **Potentially Dependent Predicate Set (PDPS)** *if it actually did not affect the value of an argument of a predicate in the success branch of T, but could have affected it had its boolean outcome been different.*

In the following we try to identify those predicates which satisfy this condition.

Lemma 1 *Let P be a logic program, T the Trace-tree for the goal g. Then PDPS= {The predicates of the success branch of T} ∪ {The predicates of the failed leaves of T}.*

Proof To prove the validity of this Lemma we have to demonstrate that these predicates really satisfy the condition of Definition 7 while the other predicates of T do not. To achieve this, we classify the predicates of T in such a way that the categories cover all predicates belonging to T. Notice that we use "the selected variable" expression but it could have been any variables of the program whose values do not satisify our expectations (i.e. where a bug was manifested). The

PDPS is the same for every selected variable, so it is created for a Trace tree built up for a given goal.

- **If a predicate should have failed but actually succeeded**

  1. If this predicate (selected goal) belongs to the success branch of the Trace-tree, then its boolean outcome could have affected the value of the selected variable (argument), so it could have been the source of the bug. We notice that this situation caused the modification of the structure of the Trace-tree.
  2. If this predicate belongs to a failure branch of the Trace-tree, then its boolean outcome could not have affected the value of the selected variable because if had it failed it would then have caused the pruning of the subtree below this predicate. But this would have not modified the structure of the other parts of the Trace-tree.

- **If a predicate should have succeeded but it failed**

  Then this predicate is a leaf of a failed branch of the Trace-tree (because these are the only failed predicates). Its boolean outcome could have affected the value of the selected variable because it might have modified the structure of the Trace-tree.

To extend this lemma we notice that if the user had found a bug in a success branch of the SLD-tree which was not the first one, then the predicates of the previous success branches did not belong to the PDPS because if they had failed it would not have affected the structure of the later branches of the SLD-tree.

Example 4 *The PDPS of the Trace-tree in Figure 1 is the following (The nodes are identified with their marks):*
$PDPS$=$\{1, 2, 5, 6, 8, 9, 10, 11, 12, 13, 14\}$.

4.2 Debug Slice

In this section our main result *the Debug slice* is specified based on the definitions and Lemma of the previous sections. The Potentially Dependent Predicate Set of a Trace-tree (for a logic program P and goal g) includes all those predicates whose boolean outcome may affect the value of an argument in a success branch's predicate. The Debug slice deals with control dependences as well. The Debug slice is a set of predicates which contains the Potentially Dependent Predicates [Section 4.1], their data dependences [Section 3.5] and the predicates affected by some *cut*. Since the Debug slice contains all predicates of the success branch of T, the Debug slice is the same with respect to every selected argument. Hence the Debug slice is defined for a logic program P and goal g.
An interesting question is the effect of cuts. If there is a node in the Trace-tree whose leftmost goal is cut($Mark$) we remove all descendants of this node to the

right of the path W up to the node marked $Mark$. We denote this kind of path by cut(W).

An informal definition for the Debug slice is the following.
Let P be a logic program, T be the Trace-tree for the goal g.
The *Debug slice of* T consists of the following predicates:

1. The predicates of the Potentially Dependent Predicate Set (PDPS)
2. The predicates specified by the data flow of the predicates of PDPS
3. The predicates that belong to some cut(W) of T

1. The predicates of the Potentially Dependent Predicate Set (PDPS)

The Potentially Dependent Predicate Set of an Trace-tree includes all predicates whose boolean outcome may affect the value of an argument in a success branch's predicate. So these predicates affect the control dependences. Lemma 1 describes those predicates which belong to the PDPS.
We would like to extend this set. But then we must first see how is it possible to describe the new predicates that are introduced by the data flow and then see why $cut(W)$ belongs to the Debug slice too.

2. The predicates specified by the data flow of the predicates of PDPS

Since PDPS consists of two subsets, the first point is dealt with by examining two cases in turn.

1. **The predicates that belong to the success branch of T**

   Here the data flow does not introduce new predicates into the Debug slice as the data flow slice is valid for one given branch of the Trace-tree (T), and all predicates of the success branch of T are in the Debug slice.

2. **The predicates of the failed leaves of T**

   Let $n \in PDPS$ such that n is a leaf node of a failed branch of T, S the Skeleton(n), $\vec{T}_{g,n}$ the corresponding directed Proof Tree Dependence Graph (see Section 3.4), and $\phi^{-1}(n)$ the corresponding node of S (see Section 3.2). Suppose that $\phi^{-1}(n)$ is labeled by $< n, p : -a_1, \cdots, a_m >$.
   Next, construct $slice(\vec{T}_{g,n}, \alpha)$ for every $\alpha \in Pos(S)$ such that α is an argument position belonging to the head predicate p.
   Let $H = \cup_{n,\alpha}\{k \in nodes(S) \mid k$ has at least one head argument position in $slice(T_{g,n}, \alpha)$, α is an argument position of p, n is a failed leaf of $T\}$.
   Afterwards, map H back to the Trace tree ($\phi(H)$). So $\phi(H)$ contains those predicates which are specified by the data flow of the failed leaves of T.
   Let the set of these predicates be denoted by S_1.
   For example, one can see in Figure 5 that $n = 5$ is a failed leaf of the Trace tree (Figure 1), then $\phi^{-1}(5)$ is $< 5, (e(Y, I) >$. As the clause contained in this node is a fact, the head predicate is $e(Y)$, which has one argument position

Y. Constructing the slice with respect to this argument position it contains head argument position from node 4 and 5. So $\phi(S1)$ in this case contains nodes 4 and 5 of the Trace tree in Figure 1.

Now we will address the second point.

3. The predicates that belong to some cut(W) of T

If there is a node in the Trace-tree whose **leftmost goal is cut(Mark)** we remove all descendants of this node to the right of the path W up to the node marked $Mark$. So a cut may effect the control dependences and those nodes that belong to W have to be added to the Debug slice. The removed part of the Trace-tree might have the right solution.

Let the set of these predicates be denoted by S_2.

Example 5 *In Example 3 if we had not had cut in clause 3, we would have got* $X = 1$, *so* $< 5, e(X) >$ *and* $< 6, h(X) >$ *would have affected the value of the variable* X *in* $a(X)$, *but this would not have shown up in data flow analysis.*

Definition 8 *The* **Debug slice** *of an Augmented Proof Tree for a goal* g *is the following set:*
Debug slice= $PDPS \cup S_1 \cup S_2$.

Example 6 *In Example 1 the PDPS contains the buggy predicate* $A > 0$, *so* $A > 0$) *is in the Debug slice, but the data flow slice does not have it because this predicate belongs to a failed branch of the SLD-tree for the goal* $p(0, X)$.

Example 7 *We will now go on with Example 2.*

In order to construct the Debug slice we furnish the sets PDPS, S_1, S_2.

1. We know (see Section 4.1) that **PDPS** $= \{1, 2, 5, 6, 8, 9, 10, 11, 12, 13, 14\}$ for the Augmented SLD-tree in Figure 1.
2. To get S_1 we have to construct a Skeleton(n) for every $n \in \{1, 2, 5, 6, 8, 9, 10, 11, 12, 13, 14\}$ and the corresponding Proof Tree Dependence Graphs. We also have to specify a $slice(\vec{T}_{g,n}, \alpha)$ for every α argument position of n in the Proof Tree Dependence Graph and to state every node of T that belongs to these slices. Figure 5 shows the Proof Tree Dependence Graph for Skeleton(5) and $slice(\vec{T}_{g,n}, (5, 0, 1, \{Y\}))$. We urge the reader to construct all the slices for each argument position of $\{1, 2, 5, 6, 8, 9, 10, 11, 12, 13, 14\}$. In our case the only node in T specified by these slices is node 4. So $\vec{S}_1 = \{4\}$.
3. We had no cut in this example, so $\vec{S}_2$ is empty.

Then, **Debug slice**= $\{\{1, 2, 5, 6, 8, 9, 10, 11, 12, 13, 14\} \cup \{4\} \cup \{\}\} =$
$\{1, 2, 4, 5, 6, 8, 9, 10, 11, 12, 13, 14\}$

In this example the only nodes that are not in the Debug slice are 3 and 7. The Debug slice of the Trace-tree built for Example 2 is emphasised and framed on Figure 1.

5 Prototype Implementation

We have developed a prototype in Sicstus Prolog using the complete framework described for slicing Prolog programs. The implementation handles specific programming techniques (cut, if-then, OR). The Trace-tree is constructed from the call ports of the trace given by the Interpreter. The slice with respect to a node of the Trace-tree is created. The slicing technique, if desired, can be combined with Shapiro's debugging method [14, 19]. To first approximation the slice is built up for the success branch of the Trace-tree. The slice is then constructed with respect to the given argument position, and during a debugging session the system asks for the validity of just those nodes that are in the slice. If it is unsuccessful in locating the source of the bug, the Debug slice is constructed and the user can then request data flow information as well for any leaf node predicate of the Debug slice. A graphical interface draws the Trace-tree and highlights those nodes that are included in the data flow slice and in the Debug slice [10].

We tested our Debug Slice algorithm on several small Prolog programs. These programs can produce big fail branches for some inputs. The test results are shown in the Table below. We examined the number of nodes and arguments in the whole Trace tree, in the success and failed branch of the Trace tree, and lastly in the Debug slice.

	Complete Tree		Success Branch		Failed Branch		Debug Slice	
	Nodes	Arg.	Nodes	Arg.	Nodes	Arg.	Nodes	Arg.
1.	14	14	6	6	8	8	13	13
2.	15	26	6	9	9	17	7	10
3.	19	39	7	11	12	28	11	20
4.	34	64	15	15	19	49	23	39
5.	267	618	11	13	256	605	16	23
6.	316	769	15	15	301	754	23	39
7.	520	1393	24	49	496	1344	181	423
8.	622	1240	6	9	616	1231	7	10
9.	1142	2687	5	9	1137	2678	7	13

The test results demonstrate that if the number of the failed branch's nodes is high and the data flow slices for the failed branch's leaf predicates do not contain too many predicates, then the Debug slice is significantly smaller than the whole Trace tree. The test results of course depend on the size and type of input, as well. The Debug slice method handles types of bugs which the conventional data-flow slice technique misses. Certain types of bugs were found during testing which were missed by the data-flow slice but were identified using the Debug slice method, as they appeared in the failure branches of the SLD-tree. These types include cases, when:

- a cut is mis-placed.
- a failed predicate is mis-printed (its name or arity) or a condition $(<, >, =)$ is failed.

- a wrong data value has reached the failed node; so in the data-flow from the root to the failed node, a wrong constant value, a mis-printed predicate or a failed condition has appeared.

We notice that the system finds only those mis-printed predicates of the failure branches of the SLD-tree which occur in the data-flow of a failed predicate, or are affected by a cut.

6 Related Work and Discussion

While program slicing has been widely studied for imperative programs [23], relatively few papers have dealt with the problem of slicing logic programs [9, 20, 22, 27].

Gyimóthy and Paaki present in [9] a specific slicing algorithm for sequential logic programs in order to reduce the number of user queries of an algorithmic debugger. But they only analyzed the data dependences for the success branch of the SLD-Tree (Trace-tree). Sometimes it is insufficient to locate the source of a wrong solution because the cause of the erroneous result may also be an invalid the proof tree structure. We solved this problem by dealing with control dependences, as well. So the data flow slice given in [9] is a special case of our approach.

Schoening and Ducasé have proposed a backward slicing algorithm for Prolog which produces executable slices [20]. An executable slice is usually less precise than a general slice [23], and their algorithm is only applicable to a limited subset of Prolog programs. Our aim was to develop a tool for debugging Prolog programs that also handles specific programming techniques.

In [27] Zhao at al. presented a new program representation called the argument dependence net for concurrent logic programs in order to produce static slices at the argument level. Dynamic slicing usually produces more precise slices than static ones because it only considers a particular execution of a program. We chose the dynamic version because our application focuses on debugging .

In [17] Pereira and Calejo examined the wrong solution suspect set (WSS) and the missing solution suspect set (MSS). It is possible to refine WSS using our general slice definition.

In [22] a dynamic slicing method was presented for constraint logic programs based on variable sharing and groundness analysis. In the paper the declarative formulation of the slicing problem for constraint logic programs was also described.

The Debug slicing method for Prolog programs was introduced in this paper. This slicing technique is very appropiate for debugging because it deals with control dependences as well. This slicing method will be integrated into the IDTS interactive algorithmic debugging tool [14]. This tool employs an improved version of Shapiro's debugging method [19] for identifying a buggy clause. By using slicer modules the number of user interactions can be reduced during the debugging process. The data flow slice is usually much smaller than the Debug slice, so in the first step we can try to locate the bug in the data flow slice. However it may happen that the data flow slice does not contain the buggy clause. In this case the debugging process has to be extended to the nodes of Debug slice. We tested our slicing tool

on small Prolog programs [10]. Now we plan to improve the implementation of the tool so that it will be able to analyze real-sized Prolog programs. We also would like to compare the size of different slices of big SLD-trees.

References

[1] D.C.Atkinson and W.G.Griswold: The Design of Whole-program Analysis Tools. *In Proceedings of the 18th International Conference on Software Engineering*, pages 16-27, Berlin, March 1996.

[2] S. Bates, S. Horwitz: Incremental program testing using program dependence graphs. *In Proceedings of Conference Record of the Twentieth ACM Symposium on Principles of Programming Languages* , pages 384-396, Charleston, 1993.

[3] J. Boye, J. Paakki, J. Maluszyński: Dependency-Based Groundness Analysis of Functional Logic Programs. *Research Report LiTH-IDA-R93-20*, Department of Computer and Information Science, Linköping University, 1993.

[4] J. Boye, J. Paakki, J. Maluszyński: Synthesis of Directionality Information for Functional Logic Programs. *In Proceedings of 3rd International Workshop on Static Analysis*, LNCS 724, Springer-Verlag, pages 165-177, Padova, 1993.

[5] J. Cheng: Dependence Analysis of Parallel and Distributed Programs and its Applications. *In Proceedings of International Conference on Advances in Parallel and Distributed Computing (IEEE-CS)*, 1997.

[6] P. Deransart and J. Małuszyński: *A grammatical view of logic programming.* The MIT Press 1993.

[7] J. Ferrante, K.J. Ottenstein and J. D. Warren: The Program Dependence Graph and its Uses in Optimization. *ACM Transactions on Programming Languages and Systems*, 9(3): pages 319-349, July 1987.

[8] T. Gyimóthy, Á. Beszédes and I. Forgács: An Efficient Relevant Slicing Method for Debugging. *In Proceedings of 7th European Software Engineering Conference (ESEC'99)*, pages 303-322, Toulouse, France, September 1999.

[9] T. Gyimóthy and J. Paakki: Static Slicing of Logic Programs.*In Proceedings of Second International Workshop on Automated and Algorithmic Debugging (AADEBUG'95)*, pages 85-105, Saint Malo, France, May 1995.

[10] L. Harmath, Gy. Szilágyi, T. Gyimóthy and J.Csirik: Debug Slicing of Logic Programs. *Technical Reports 99/04*, RGAI, Szeged, 1999, *www.inf.u-szeged.hu/rgai/.*

[11] S. Horwitz, J. T. Reps and D. Binkeley: Interprocedural Slicing Using Dependence Graphs. *Proceedings of the ACM SIGPLAN'88 Conference on Programming Language Design and Implementation*, SIGPLAN Notices, 23(7), pages 35-46, July 1988.

[12] S. Horwitz and T. Reps: The Use of Program Dependence Graphs in Software Engineering. *In Proceedings of 14th International Conference on Software Engineering*, pages 392-411, Melbourne Australia, May 1992.

[13] Kamkar M.: Interprocedural Dynamic Slicing with Applications to Debugging and Testing. Linköping Studies in Science and Technology - Dissertation No. 297, Department of Computer and Information Science, Linköping University, 1993.

[14] G. Kókai, L. Harmath, T. Gyimóthy: Algorithmic Debugging and Testing of Prolog Programs. *In Proceedings of the 14th International Conference on Logic Programming (ICLP'97). Eighth Workshop on Logic Programming Environments*, pages 14-21, Leuven Belgium, July 1997.

[15] B. Korel and J. Rilling: Application of Dynamic Slicing in Program Debugging. *In Proceedings of the Third International Workshop on Automatic Debugging (AADEBUG '97)*, Linköping, Sweden, May 1997.

[16] K. J. Ottenstein and L. M. Ottenstein: The Program Dependence Graph in a Software Development Environment. *Proceedings of the ACM SIGSOFT/SIGPLAN Symposium on Practical Software Development Environments, 1984*, SIGPLAN Notices, 19(5), pages 177-184, May 1984.

[17] L.M. Pereira and M. Calejo : A Framework for Prolog Debugging, *ICLP/SLP*, pages 481-495, 1988.

[18] U. Nilsson and J. Maluszyński: Logic, Programming and Prolog. *John Wiley and Sons*, 1990.

[19] E.Shapiro: Algorithmic Debugging. *The MIT Press*, 1993.

[20] S. Schoenig and M. Ducassé: A Backward Slicing Algorithm for Prolog. *In Proceedings of 3rd International Static Analysis Symposium (SAS'96)*, LNCS 1145, Springer-Verlag, pages 317-331, 1996.

[21] C. Steindl: Intermodular Slicing of Object-oriented Programs. *In International Conference on Compiler Construction (CC'98)*, 1998.

[22] Gy. Szilágyi, T. Gyimóthy, J. Maluszyński: Slicing of Constraint Logic Programs. *Linköping University Electronic Press* 1998/020, *www.ep.liu.se/ea/cis/1998/002.*

[23] F. Tip: A survey of Program Slicing Techniques. *Journal of Programming Languages*, Vol.3., No.3, pages 121-189, September, 1995.

[24] U. Nilsson and J. Maluszyński: Logic, Programming and Prolog. *John Wiley and Sons*, 1990.

[25] M. Weiser: Programmers use slices when debugging. *Communications of the ACM*, pages 446-452, July 1982.

[26] M. Weiser: Program Sicing. *In Proceedings of Transactions on Software Engineering (IEEE)*, pages 352-357, July 1984.

[27] J. Zhao, J. Cheng and K. Ushijima: Slicing Concurrent Logic Programs . *In Proceedings of Second Fuji International Workshop on Functional and Logic Programming*, pages 143-162, 1997.