Literal Dependence Net and Its Use in Concurrent Logic Programming Environment

Jianjun Zhao, Jingde Cheng, and Kazuo Ushijima
Department of Computer Sci.& Communication Eng.
Kyushu University
6-10-1 Hakozaki, Fukuoka 812, Japan
{zhao, cheng, ushijima}@csce.kyushu-u.ac.jp

Abstract

Program dependences are dependence relationships holding between statements in a program which can be used to infer about the behavior of the program. In this paper we propose a general framework for dependence analysis for concurrent logic programs, in particular for Flat Concurrent Prolog programs. The first contribution of this paper is to present two language-independent program representations for explicitly representing control flows and/or data flows in a concurrent logic program. Based on these representations, program dependence analysis for concurrent logic programs becomes possible. The second contribution is to present a dependence-based representation named the Literal Dependence Net *(LDN) for explicitly representing primary program dependences in a concurrent logic program. Moreover, we discuss some possible applications based on the LDN, which include program debugging, testing, complexity measurement, and maintenance in a concurrent logic programming environment.*

1 Introduction

Program dependences are dependence relationships holding between statements in a program that can be regarded as an intrinsic attribute of a program and used to infer about behavior of the program. Intuitively, if the computation performed by a statement affects the behavior of another statement, then there may exist some dependence relationship between the statements. Program dependences in a program can be determined by analyzing control flows and/or data flows in the program. Dependence-based program representations such as *Program Dependence Graph* or *Process Dependence Net* for imperative sequential or concurrent programs, have been developed as an important program representation tools used in program optimization, understanding, testing, debugging, information-flow control, complexity measurement, and maintenance [3-5,8-10,14-19,25,26]. The seminal works on program dependence-based representations for imperative sequential programs are due to the Kuck et al.[16] and Ferrante and Ottenstein et al.[8,17], and its adaptation to imperative concurrent programs has been proved successful [3,4]. This paper is based on the framework initiated and developed by Cheng [3], which studied the program dependences for imperative concurrent programs. In addition to the usual control and data dependences proposed and studied for imperative sequential programs, Cheng [3] introduced three new kinds of primary program dependences for imperative concurrent programs called *selection dependence*, *synchronization dependence*, and *communication dependence*. He also presented a program representation for imperative concurrent programs called the *Process Dependence Net*, which is an arc-classified digraph to explicitly represent five types of primary program dependences in the programs, and also discussed some applications of the PDN in development of distributed systems.

However, although some dependence-based program representations have been proposed and studied for imperative sequential and concurrent programs, until recently, there is no dependence-based representation proposed for concurrent logic programs. In this paper we extend the framework of Cheng [3] to concurrent logic programs, taking the Flat Concurrent Prolog (FCP) [22,23] as the target language of this study. This paper makes the following contributions:

First, we propose two graph-theoretical program representations for explicitly representing control flows and/or data flows in a concurrent logic program. Based on these representations, program dependence analysis for imperative concurrent programs can be extended to concurrent logic programs naturely.

Second, in addition to the data dependences in concurrent logic programs, we propose three new types of primary program dependences named selective control, synchronization, and communication dependences to represent the control flows and interprocess interactions in the programs. Moreover, we propose a dependence-based representation named the *Literal Dependence Net* (LDN) for explicitly representing these primary program dependences in a concurrent logic program.

Third, we discuss some possible applications of LDN to programming activities including program slicing, debugging, testing, complexity measurement, and maintenance in a concurrent logic programming environment.

Dependence analysis of sequential logic programs has been discussed by Chang et al.[2], and Warren et al.[24]. They described how the data dependence information can be used both to parallelize Prolog programs and to improve its backtracking behavior without incurring runtime overhead significantly. Data dependence analysis has also been investigated by Debray [6], who considers the static inference of modes and data dependences information in sequential logic program. King et al. [13] proposed a framework for schedule analysis for concurrent logic programs, which was built from the notion of a data-dependence to define a procedure for creating threads.

In this paper our aim is quite different. In addition to analyze the data dependences in a concurrent logic program as mentioned above, we also analyze selective control, synchronization, and communication dependences in the program, and our motivation is to develop a dependence-based, unified representation tool useful in various concurrent logic programming activities including program debugging, testing, complexity measurement, and maintenance. To our knowledge, this work has not been done elsewhere in the literature.

The rest of the paper is organized as follows. Section 2 introduces the syntax of FCP and defines two new representations named the And/Or parallel control-flow net and And/Or parallel definition-use net for explicitly representing control flows and/or data flows in a concurrent logic program. Section 3 shows how to construct the And/Or parallel control-flow net and And/Or parallel definition-use net for an FCP program. Section 4 defines the selective control, data, synchronization, and communication dependences in a concurrent logic program based on its And/Or parallel control-flow net and And/Or parallel definition-use net, and presents the Literal Dependence Net. Section 5 shows some possible applications of the LDN in a concurrent logic program development environment. Concluding Remarks are given in Section 6.

2 Preliminaries

2.1 Syntax of FCP

We assume that readers are familiar with the basic concepts of logic programs, and throughout this paper, we will restrict ourselves to FCP. This language illustrates the basic mechanisms of concurrent logic programming.

A *program* is a finite set of guarded clauses. A *guarded clause* is a formula of the form: $H : -G_1, G_2, ..., G_n | B_1, B_2, ..., B_m.(m, n \geq 0)$. where $H. B_1, B_2, ..., B_m$ are literals. $G_1, G_2, ..., G_n$ are guard test predicates. H is called the head of clause. $G_1, G_2, ..., G_n$ the *guard*, and $B_1, B_2, ..., B_m$ the *body*. ": $-$", read if, denotes implication, and "|" is called the *commit* operator. If the guard is empty the clause is written as: $H : -B_1, B_2, ..., B_m$. and the commit operator is omitted. If the body is empty and the guard is not empty, the clause is written as: $H : -G_1, G_2, ..., G_n \mid true$. And if both are empty the clause is called a *unit clause*. and is written simply as: H. A clause whose body includes exactly one goal is called an *iterative clause*. A *procedure* is a set of clauses each of which has the same head literal.

In this paper we assume a fixed finite set of *guard test predicates*, including $integer(X)$, $X < Y, X = Y$, and $X \neq Y$. These predicates require their arguments to be ground and therefore all variables appearing in the guard of a clause have to appear in the head. Moreover, we assume that the guard of a clause only contains one guard test predicate for expository purposes. In what follows, we let P be a program, and let V_{0i} and $V_{1i}, V_{2i}, ..., (i = 0, 1, ..., m)$ be the goal and clauses of the P respectively. The literals of the goal and every clause are numbered by $i = 0, 1, ..., m$.

2.2 The Process Reading of Logic Programs

Concurrent logic programming languages apply a new reading of logic programs, i.e., the *process reading*. According to this reading, each goal atom $p(T_1, ..., T_n)$ is viewed as a network of concurrent processes, whose process interconnection pattern is specified by the logical variables shared between goal atoms. Processes communicate by instantiating shared variables and synchronize by waiting for some logical variable to be instantiated. This view is summarized in Table 1. The possible behaviors of a process are specified by guarded clauses. A process can terminate (empty body), change state (unit body), or become several concurrent processes (a conjunctive body). This is summarized in Table 2.

Table 1. The Process Reading of Logic Programs

Process model	**Concurrent logic programming model**
Process	Goal atom
Process Network	Goal(collection of atoms)
Instruction for process action	Clause (see Table 2)
Communication channel	Share logical variable
Communication	Instantiation of a shared variable
Synchronization	Wait until a shared variable is sufficiently instantiated

Table 2. Clauses as Instructions for Process Behavior

Process behavior	**Relative clause**
Terminate	$A : -G \vert true$
Change state (i.e.,become a different process)	$A : -G \vert B$
Become k concurrent processes	$A : -G \vert B_1, ..., B_k$

2.3 Terminology

Definition 2.1 A *digraph* is an ordered pair(V, A), where V is a finite set of elements called *vertices*, and A is a finite set of elements of the Cartesian product $V \times V$, called *arcs*, i.e., $A \subseteq V \times V$ is a binary relation on V. For any arc $(v1, v2) \in A$, v_1 is called the *initial vertex* of the arc and said to be *adjacent to* v_2, and v_2 is called *terminal vertex* of the arc and said to be *adjacent from* v_1. A *predecessor* of a vertex v is a vertex adjacent to v, and a *successor* of v is a vertex adjacent from v. The *in-degree* of vertex v, denoted by in-degree(v), is the number of predecessors of v, and the *out-degree* of a vertex v, denoted by out-degree(v), is the number of successors of v. A *simple digraph* is a digraph(V, A) such that no $(v, v) \in A$ for any $v \in V$.

Definition 2.2 An *arc-classified digraph* is an n-tuple$(V, A_1, A_2, \ldots, A_{n-1})$ such that every (V, A_i) $(i = 1, \ldots, n-1)$ is a digraph and $A_i \cap A_j = \phi$ for $i = 1, 2, \ldots, n-1$ and $j = 1, 2, \ldots, n-1$. A *simple arc-classified digraph* is an arc-classified digraph $(V, A_1, A_2, \ldots, A_{n-1})$ such that no $(v, v) \in A_i$ $(i = 1, \ldots, n-1)$ for any $v \in V$.

Definition 2.3 A *path* in a digraph (V, A) or an arc-classified digraph $(V, A_1, A_2, \ldots, A_{n-1})$ is a sequence of arcs $(a_1, a_2, \ldots, a_l)$ such that the terminal vertex of a_i is the initial vertex of a_{i+1} for $1 \leq i \leq l-1$, where $a_i \in A (1 \leq i \leq l)$ or $a_i \in A_1 \cup A_2 \cup \ldots \cup A_{n-1} (1 \leq i \leq l)$, and $l (l \geq 1)$ is called the *length* of the path. If the initial vertex of a_1 is v_I and the terminal vertex of a_l is v_T, then the path is called a path from v_I to v_T, or path $v_I - v_T$ for short.

Definition 2.4 An *And/Or parallel control-flow net* (CFN) is a 8-tuple $(V, V_{and}, V_{or}, A_c, A_{p_{and}}, A_{p_{or}}, s, t)$, where $(V, A_c, A_{p_{and}}, A_{p_{or}})$ is a simple arc-classified digraph such that $A_c \subseteq V \times V, A_{p_{and}} \subseteq V_{and} \times V, A_{p_{or}} \subseteq V_{or} \times V, V_{and} \subset V$ is a set of elements, called *AND-parallel execution vertices*, $V_{or} \subset V (V_{or} \cap V_{and} = \phi)$ is a set of elements, called *OR-parallel execution vertices*, $s \in V$is a unique vertex, called *start vertex*, such that in-degree$(s) = 0$, $t \in V$ is a unique vertex, called *termination vertex*, such that out-degree$(t) = 0$ and $t \neq s$, and for any $v \in V (v \neq s, v \neq t)$, there exists at least one path from s to v and at least one path from v to t. Any arc $(v1, v2) \in A_{p_{and}}$ is called an *AND-parallel execution arc*, any arc $(v1, v2) \in A_{p_{or}}$ is called an *OR-parallel execution arc*.

Definition 2.5 An *And/Or parallel definition-use net* (DUN) is a 7-tuple $(N_c, \Sigma_v, D, U, \Sigma_c, S, R)$, where $N_c = (V, V_{and}, V_{or}, A_c, A_{p_{and}}, A_{p_{or}}, s, t)$ is a CFN, Σ_v is a finite set of symbols, called *variables*, and $D : V \rightarrow \Sigma_v$ and $U : V \rightarrow \Sigma_v$ are two partial functions from V to the power set of Σ_v, Σ_c is a finite set of symbols, called *channels*, and $S : V \rightarrow \Sigma_c$ and $R : V \rightarrow \Sigma_c$ are two partial functions from V to the power set of Σ_c.

Note that the above definitions of CFN and DUN are graph-theoretical, and therefore, they are independent of any concurrent logic programming languages. Moreover, CFNs and DUNs of sequential logic programs can also be constructed in similar way as described above for concurrent logic programs.

3 CFNs and DUNs of concurrent logic programs

Using the process reading of concurrent logic programs, the action of process can be divided into two aspects, i.e., control actions and data processing actions. Control actions include termination, iteration, forking, and creation of new processes. These are specified explicitly by the clauses of a concurrent logic program. Control actions correspond to the control flows in a program. Data processing actions include communication and various operations on data structures, e.g., single-assignment, inspection, testing, and construction. Data processing actions are specified implicitly by the arguments of the head and body literals of a clause, and are realized via unification. Data actions correspond to the data flows in the program. we can construct the And/Or parallel control-flow net in a concurrent logic program to represent the control flows in the program, and construct the And/Or parallel definition-use net to represent data flows in the program. We use FCP as the target programming language to show how to construct CFNs and DUNs of concurrent logic programs.

3.1 Constructing CFNs of FCP programs

We now describe the informal translation rules for constructing the And/Or parallel control-flow net of an FCP program. In the CFN, we use vertices to represent the head and body literals and guards in the program, use AND-parallel execution vertices to represent the guard or head literals (if guards are empty) of clauses, use OR-parallel execution vertices to represent the procedure literal which corresponds to a group of clauses with the same head literal, use arcs to represent possible control flows between literals. Moreover, we use a unique start vertex s and a unique termination vertex t to represent the beginning and the end of the program, represectively. Note that in the CFN, the vertices s and t do not correspond to any clause in the program, only for easy representation. As what follows, we show the basic informal translation rules for constructing the CFN of a FCP program. This is based on the process reading of logic programs.

A *unit clause* is a definite clause with an empty body: $p(T_1, T_2, ..., T_n)$, specifying that a process in a state unifiable with $p(T_1, T_2, ..., T_n)$ can reduce itself to the empty set of processes, and therefore terminate. We create two vertices to represent the unit clause, as shown in Fig. 1(a), one is for literal p, the other is the termination vertex t, and there is an arc from p to t.

An *iterative clause* is a clause with one body literal: $p(T_1, T_2, ..., T_n)\ :-\ G \mid q(S_1, S_2, ..., S_m)$, specifying that a process in a state unifiable with $p(T_1, T_2, ..., T_n)$ can change its state to $q(S_1, S_2, ..., S_m)$. The program state is changed to q/m (i.e., forking), and the data state to $(S_1, S_2, ..., S_m)$. There is a special case when the body literal of an iterative clause is the same as its head literal, i.e., $p(T_1, T_2, ..., T_n)\ :-\ G \mid p(S_1, S_2, ..., S_n)$. We create two vertices to represent the head and the body literals respectively. As shown in Fig.1 (b), p is for the head literal, g is for the guard, and q is for the body literal, and there are two arcs from p to g and g to q.

A *general clause* has the following form: $p(T_1, T_2, ..., T_n)\ :-\ G \mid Q_1, Q_2, ..., Q_m$, specifying that a process in a state unifiable with $p(T_1, T_2, ..., T_n)$ can replace itself with m new processes as specified by $Q_1, Q_2, ..., Q_m$, i.e., each atomic goal in the body of the clause is reduced in AND-parallelism. As shown in Fig.1(c), we create a vertex p for the head literal, an AND-parallel execution vertex q for the guard of the clause, m vertices for m subgoal processes (body literals) created by the clause, and there are m arcs from g to $q_1, q_2, ..., q_m$.

A *goal clause* with the form: $:-\ Q_1, Q_2, ..., Q_m$ specifies that m new processes can be invoked to execute in parallel. We create a start vertex s to represent the start of the program, use m vertices to represent m subgoals in the goal. There are m arcs from the s to $v_1, v_2, ..., v_m$ as shown in Fig.1 (d).

A *procedure* corresponding to *OR-parallel processes* with the form:

$$\begin{gathered} p(T_{11}, T_{12}, ..., T_{1n})\ :-\ G_1 \mid Q_1. \\ p(T_{21}, T_{22}, ..., T_{2n})\ :-\ G_2 \mid Q_2. \\ ...., \\ p(T_{m1}, T_{m2}, ..., T_{mn})\ :-\ G_m \mid Q_m. \end{gathered}$$

specifies that the clauses in the procedure is reduced in OR-parallelism. To construct the sub-CFN of a procedure, we can construct the sub-CFN of each clause firstly, then use an OR-parallel execution vertex to represent the procedure literal and connect procedure literal to every head literal of clauses respectively, as shown in Fig.1 (e).

Having these basic translation rules, it is not difficult to translate an FCP program to its CFN. First, we can generate the sub-CFNs for the goal and each clause of the program respectively. Then we can generate the sub-CFGs for each procedure. Finally, we can generate the total net by connecting the procedures of the program by OR-parallel execution arcs. The

CFN of a concurrent logic program can be used to define the selective control dependence in the program. Fig.2 shows a FCP program and its CFN.

3.2 Constructing DUNs of FCP programs

In order to formally define the data, synchronization, and communication dependences in a concurrent logic program, we construct the DUN of the program. The DUN of a concurrent logic program can be regarded as an annotated CFN of the program, whose vertices are those for its CFN,

and annotated in four functions. First, there is a function $D(v)$ for the set of all variables defined at vertex v. Second, there is a function $U(v)$ for the set of all variables used at vertex v. Third, there is a function $S(v)$ for the set of all channel variables sent at vertex v. Lastly, there is a function $R(v)$ for the set of all channels received at vertex v. To construct the DUN of an FCP program, we should define these four functions explicitly. In a logic program, the variables with the same name in different clauses are $different$ variables. To avoid misusing variables, we make a transformation for a concurrent logic program before constructing the DUN of the program. Through this transformation, there is no variable name shared by any two clauses. In what follows, the construction of the DUN of a FCP program is based on its transformed program.

To determine the set of $D(v)$ and $U(v)$ of a literal corresponding to a vertex v in the CFN, we use two abstract interpretations. One abstract interpretation which is a variation on that proposed by Sehr [20] is done to get the set $U(v)$ of variables used at each literal, The set $D(v)$ of variables defined at each literal are found by another abstract interpretation called *mode inference* that infers approximate bindings for variables taking aliasing into account. This work are based on that proposed by Dabray [6] and Sehr [20]. To obtain the sets of $S(v)$ and $R(v)$, we can do as follows. If there is a literal represented by a vertex v in the CFN containing a read-only variable X, we add X into $R(v)$. If there is a literal represented by v containing a writable variable X such that $?X$ is its read-only variable occurred in another literal, we add X into $S(v)$. The DUN of a concurrent logic program can be regarded as the a CFN with the information concerning definitions and uses of variables and communication channels at each program point. As an example, Fig.3 shows a simple FCP program, its transformed program, and its DUG.

4 Program Dependences and Literal Dependence Net

Generally, a concurrent logic program consists of a number of processes. These processes are not independent because there exist interprocess synchronization and communication among them. As a result, there are not only intraprocess control flows and data flows but also interprocess interactions among these control flows and data flows. Based on the CFN and DUN of a concurrent logic program, we define four types of primary program dependence relationships, i.e., selective control dependence, data dependence, synchronization dependence, and communication dependence, to capture such attributes of a concurrent logic program.

4.1 Selective Control Dependences

In concurrent logic programs, there are no specific control mechanism (conditional branch structure, for instance) such as in traditional imperative programs. The structure that can be regarded as presenting the control primitive is the guard of a clause. Informally, a literal u is directly selective control-dependent on a nondeterministic selection guard v in a clause if whether u is executed or not is directly determined by the guard result of v. Definition 4.1

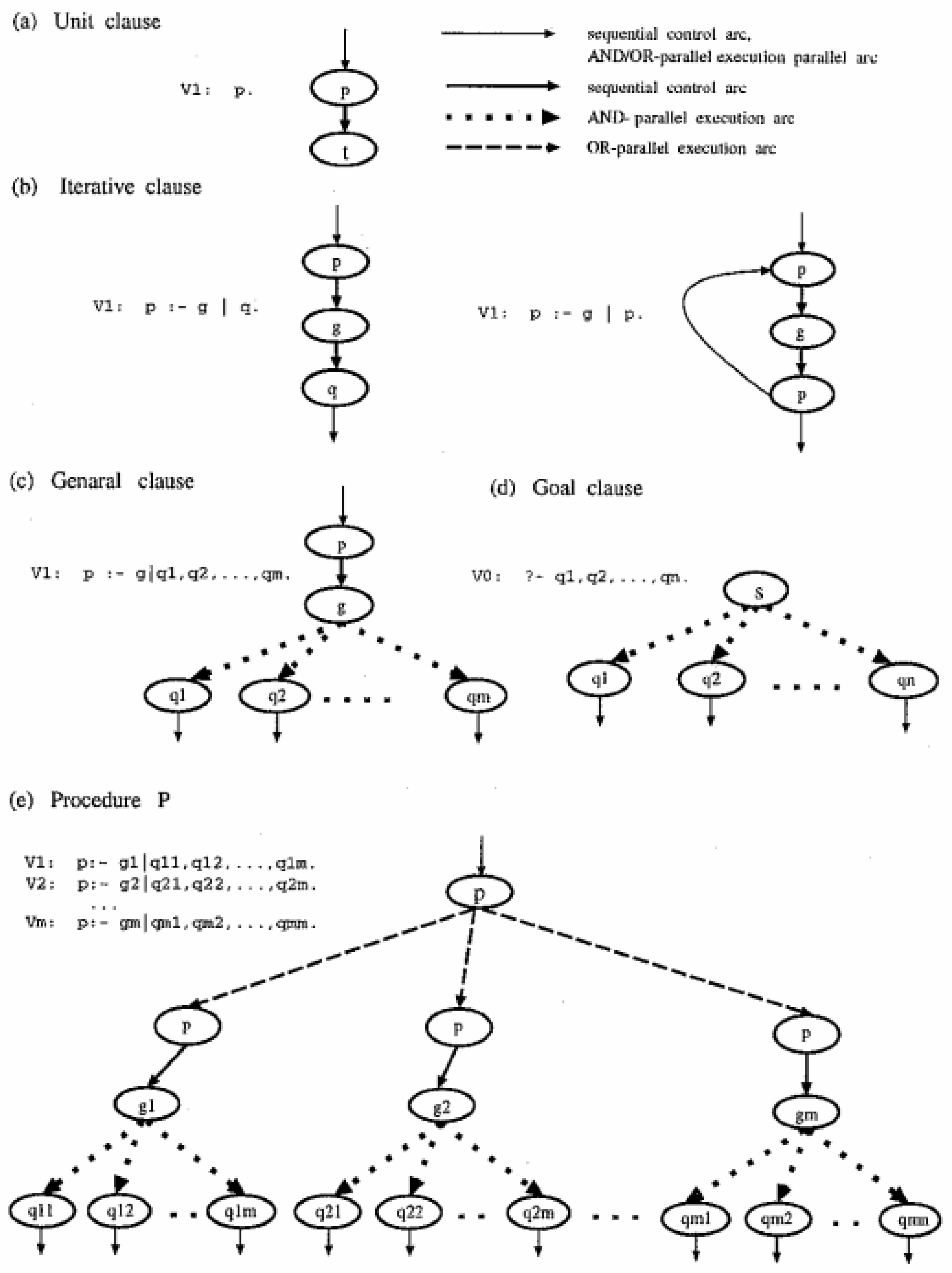

(a) Unit clause
V1: p.
p
t
sequential control arc, AND/OR-parallel execution parallel arc
sequential control arc
AND- parallel execution arc
OR-parallel execution arc
(b) Iterative clause
V1: p :- g | q.
V1: p :- g | p.
(c) Genaral clause
V1: p :- g|q1,q2,...,qm.
(d) Goal clause
V0: ?- q1,q2,...,qn.
(e) Procedure P
V1: p:- g1|q11,q12,...,q1m.
V2: p:- g2|q21,q22,...,q2m.
...
Vm: p:- gm|qm1,qm2,...,qmm.

Fig.1 Informal translation rules for constructing the CFNs of FCP programs.

```
V0 :   ?- insert(s,[1,3,4,List])

V1:    insert(X,[Y|Ys],[Y|Zs]):-
           X>Y |
           insert(X,Ys?,Zs).
V2:    insert(X,[Y|Ys],[X,Y|Ys]):-
           X<=Y|
           true.
V3:    insert(X,[],[X]).
```

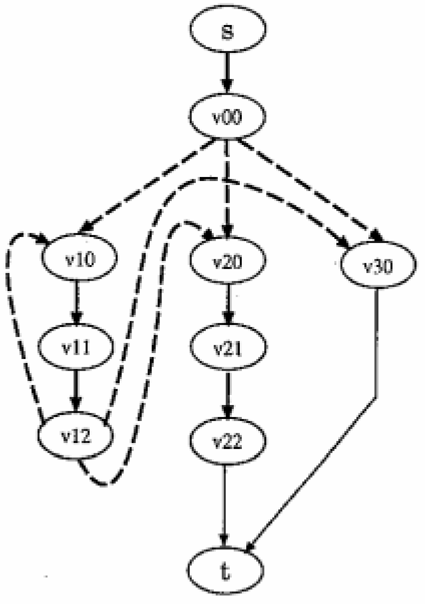


Fig.2   An FCP program and its CFN.

give the formal definition of selective control dependence in a program based on the CFG of the program.

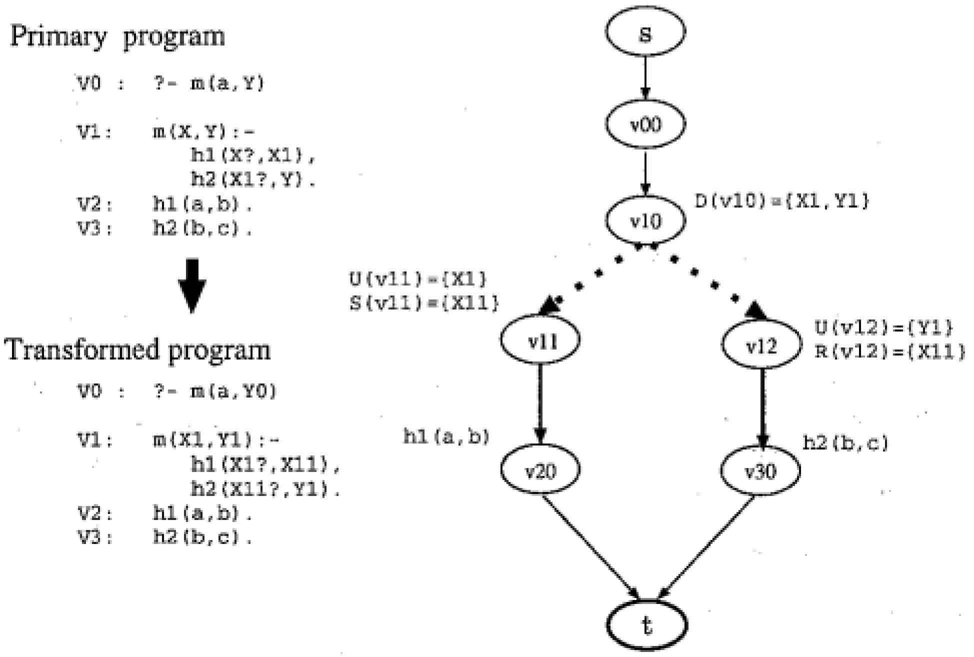
Definition 4.1 Let $(V, V_{and}, V_{or}, A_c, A_{p_{and}}, A_{p_{or}}, s, t)$ be the CFN of a concurrent logic program, and u and v be any two vertices of the net. u is *directly selective control-dependent* on v iff $(v, u) \in A_{p_{or}}$, i.e., (v, u) is an OR-parallel execution arc.

Intuitively, selective control dependence can occur if in a concurrent logic program there are some clauses which have the same head literals. Given a goal and clauses, the OR process tries to unify the clauses'heads with the goal, which satisfies the constraints posed by the guard. Only one of these clauses will be selected in a nondeterministic manner, the commit is executed, the rest of the clauses are aborted, and the OR parallelism terminates. For instance, in Fig.4 (b), vertices v_{10}, v_{20}, v_{30} are directly selective control-dependent on vertex v_{00}, because given the goal $insert(s, [1, 3, 4], List)$, the OR process will unify it with these clauses in OR-parallelism.

4.2 Data Dependences

Data dependences of logic programs have been widely studied in the literatures [2,6,20]. In this paper, in contrast to the traditional approach, mode inference is not used alone to compute data dependences. Rather, the definition sets $D(v)$ and the use sets $U(v)$ are intersected to compute data dependence in the way more similar to imperative language techniques. Suppose a literal v executes before a literal u in a clause, informally, u is directly data-dependent on v if v defines a variable x and u uses x. Definition 4.2 gives the formal definition of data dependence in a program based on the DUG of the program.

Definition 4.2 Let $(N_c, \Sigma_v, D, U, \Sigma_c, S, R)$ be the DUG of a concurrent logic program, where $N_c = (V, V_{and}, V_{or}, A_c, A_{p_{and}}, A_{p_{or}}, s, t)$ is the CFN of the program, and u and v be any two vertices of the net. u is *directly data-dependent* on v iff there is a path $P = (v_1 = v, v_2), (v_2, v_3), ..., (v_{n-1}, v_n = u)$ from v to u such that $(D(v) \cap U(u)) - D(P') \neq \phi$ where $D(P') = D(v_2) \cup ... \cup D(v_{n-1})$.



Primary program

```
V0 :   ?- m(a,Y)

V1:    m(X,Y):-
           h1(X?,X1),
           h2(X1?,Y).
V2:    h1(a,b).
V3:    h2(b,c).
```

Transformed program

```
V0 :   ?- m(a,Y0)

V1:    m(X1,Y1):-
           h1(X1?,X11),
           h2(X11?,Y1).
V2:    h1(a,b).
V3:    h2(b,c).
```

Fig.3 A simple FCP program and its DUN.

Note that the data dependences in a program can be determined by analyzing the data flows of the program, There are some efficient algorithms to compute the control and data dependences in a imperative program based on the control flow graph of the program, These algorithms can be modified to compute the selective control, data dependences in a concurrent logic program based on the CFN and DUN of the program. As an example, Fig.4 shows two simple FCP programs and their LDNs. For instance, in Fig.4 (a), vertex v_{11} is directly data-dependent on vertex v_{10} due to the head literal of clause $V1$ is a definition point for the variable $X1$, and the first body literal is a use point for it.

4.3 Synchronization Dependences

Synchronization mechanism in concurrent logic programming languages includes two aspects. One is when a clause is invoked, the subgoals in the body of clause are executed in parallel; on the other hand, when two processes want to communicate each other, they should be synchronized by waiting for logical variables to be instantiated. Informally, a literal u is directly synchronization-dependent on another literal v in a clause if the start and/or termination of execution of v directly determines whether or not the execution of u starts and/or terminates. Definition 4.3 gives the formal definition of synchronization dependence in a program based on the DUG of the program.

Definition 4.3 Let $(N_c, \Sigma_v, D, U, \Sigma_c, S, R)$ be the DUN of a concurrent logic program, where $N_c = (V, V_{and}, V_{or}, A_c, A_{p_{and}}, A_{p_{or}}, s, t)$ is the CFN of the program, and u and v be any two vertices of the net. u is *directly synchronization-dependent* on v iff any of the following conditions hold:

(1) $(v, u) \in A_{p_{and}}$, i.e., (v, u) is a AND-parallel execution arc.

(2) $S(v) = R(u)$.

Intuitively, synchronization dependence can occur between the guard or head literal (if guard is empty) of a clause and each of its body literals (subgoals) executed in parallel, and also occur between two body literals which have a shared logical variable as a communication channel. For instance, in Fig.4 (a), vertices v_{11}, v_{12} are directly synchronization-dependent on vertex v_{10}, because the unification of $m(a, Y0)$ with $m(X1, Y1)$ successful makes the body literals $h1(X1?, X11)$ and $h2(X11?, Y1)$ executed in parallel. The vertex v_{12} is directly synchronization-dependent on vertex v_{11} due to shared logical variable $X11$.

4.4 Communication Dependences

In a concurrent logic program, the communication between processes is realized by communication channels, for instance, the read-only logical variables in FCP. When two processes want to communicate each other, they should synchronize firstly, then some data can transfer from one process to another one by instantiating shared logical variables. Informally, a literal u is directly communication-dependent on another literal v if the value of a variable computed at v has a direct influence on the value of a variable computed at u by an interprocess communication. Definition 4.4 gives the formal definition of communication dependence in a program based on the DUN of the program.

Definition 4.4 Let $(N_c, \Sigma_v, D, U, \Sigma_c, S, R)$ be the DUG of a concurrent logic program, where $N_c = (V, V_{and}, V_{or}, A_c, A_{p_{and}}, A_{p_{or}}, s, t)$ is the CFN of the program, and u and v be any two vertices of the net. u is *directly communication-dependent* on v iff there exists a vertex v' such that $R(u) = S(v')$, and v' is directly data-dependent on v.

Note that the definition of communication dependence is based on the definition of synchronization dependence. Intuitively, communication dependence can occur between two literals which are in different clauses and involve some communication operation by synchronization. In a clause, there exists no communication dependence because of the data transferred by intraprocess data flows. For instance, in Fig.4 (a), vertex v_{30} is directly communication-dependent on vertex v_{11} since the data may transfer from literal $h2(b, a)$ to literal $h1(X1?, X11)$ by instantiating the shared logical variable $X11$.

4.5 Literal Dependence Net

We use an arc-classified digraph named the Literal Dependence Net to represent all four types of primary program dependences in a concurrent logic program. The net has one vertex for each vertex in the CFN for the program except the start and terminate vertices. There is an arc in the literal dependence net for one of each type of dependences.

Definition 4.5 The *Literal Dependence Net* (LDN) of a program is an arc-classified digraph (V, Con, Dat, Syn, Com), where V is the vertex set of the And/Or parallel control-flow Net of the program, but except the start and terminate vertices; Con is the set of selective control dependence arcs such that any $(u, v) \in Con$ iff u is directly selective control-dependent on v; Dat is the set of data dependence arcs such that any $(u, v) \in Dat$ iff u is directly data-dependent on v; Syn is the set of synchronization dependence arcs such that any $(u, v) \in Syn$ iff u is directly synchronization-dependent on v; and Com is the set of communication dependence arcs such that any $(u, v) \in Com$ iff u is directly communication-dependent on v.

As an example, Fig.4 shows a digraph representation of the LDNs of the transformed FCP programs in Fig.2 and Fig.3.

5 Applications

Having LDN as a unified representation of a concurrent logic program, we describe some possible applications based on LDN in a concurrent logic programming environment. Some applications are directly based on the LDN of a program, and the others are based on slices of the program.

5.1 Slicing and Debugging

A *program slice* consists of the parts of a program that (potentially) affect the values computed at some point of interesting, referred to as a *slicing criterion*. The parts of a program which have a direct or indirect effect on the values computed at a slicing criterion C are called the *program slice with respect to criterion C*. The task of computing program slices is called *program slicing*.

The original concept of a program slice was introduced by Weiser [25,26]. Weiser claims that a slice corresponds to the mental abstractions that people make when they debug a program, and suggests the integration of program slicers in debugging environments. After that, various slightly different notions of program slices and a number of methods to compute slices have been proposed for imperative programs [4,8,10]. However, until recently, there is no program slicing method proposed and studied for logic programs [7]. In this paper, we propose some static or dynamic slices of concurrent logic programs.

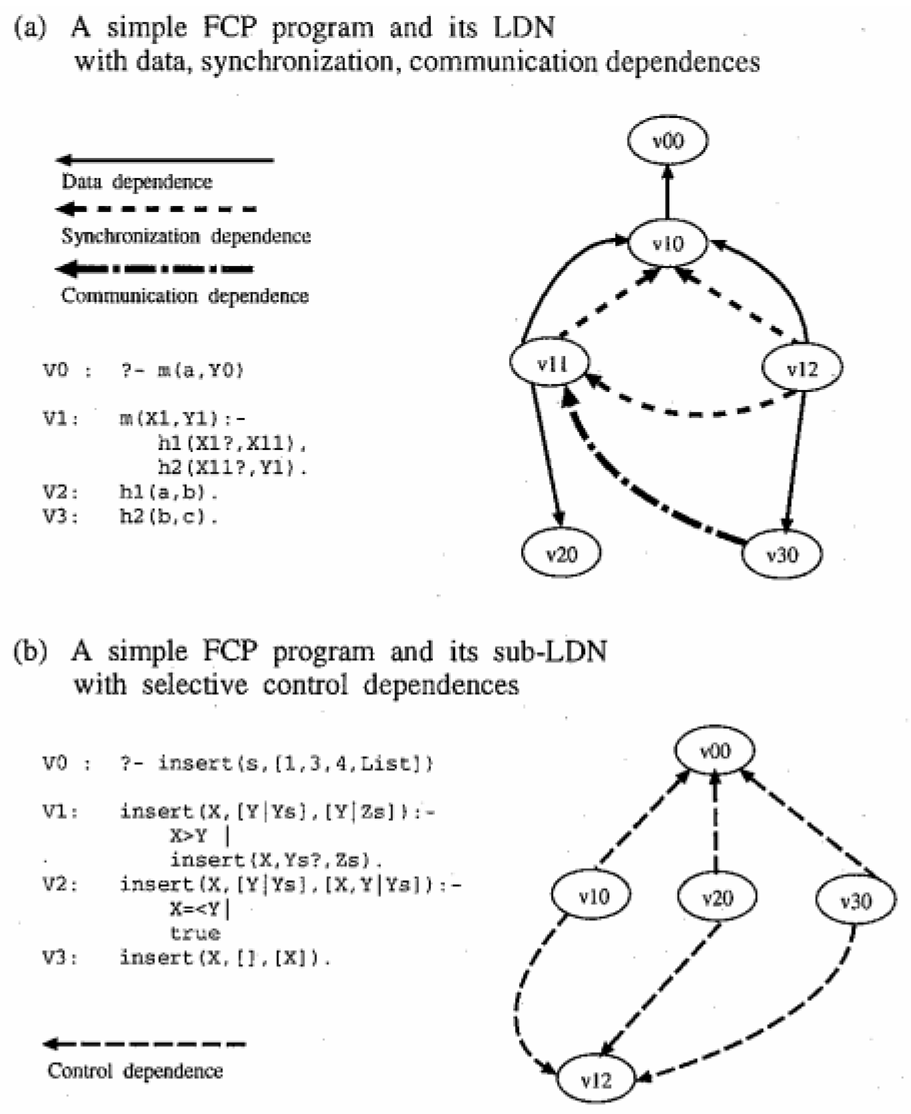
Definition 5.1 A *static slicing criterion* for a logic concurrent program is a 2-tuple (l, V), where l is a literal in the program and V is a set of variables used at l. The *static slice* $SS(l, V)$ of a concurrent logic program on a given static slicing criterion (l, V) consists of all literals in the program that possibly affect the beginning or end of execution of l and/or affect the values of variables in V. *Statically slicing* a concurrent logic program on a given static slicing criterion is to find the static slice of the program with respect to the criterion.

Note that once a concurrent logic program can be represented by its LDN, the static slicing problem of the program is reduced to be a simple reachability problem in the net.

Definition 5.2 A *dynamic slicing criterion* for a logic concurrent program is a quadruplet (l, V, H, I), where l is a literal in the program, V is a set of variables used at l, and H is a history of an execution of the program with input I. The *dynamic slice* $DS(l, V, H, I)$ of a concurrent logic program on a given dynamic slicing criterion $(l, V, H. I)$ consists of all literals in the program that actually affected the beginning or end of execution of l and/or affected the values of variables in V in the execution with I that produced H. *Dynamically slicing* a concurrent logic program on a given dynamic slicing criterion is to find the dynamic slice of the program with respect to the criterion.

Definition 5.3 A *static forward-slicing criterion* for a logic concurrent program is a 2-tuple (l, V), where l is a literal in the program and V is a set of variables used at l. The *static forward-slice* $SFS(l, V)$ of a concurrent logic program on a given static forward-slicing criterion (l, V) consists of all literals in the program that would be affected by the beginning or end of execution of l and/or affected by the value of v at l. *Statically forward-slicing* a concurrent logic program on a given static forward-slicing criterion is to find the static forward-slice of the program with respect to the criterion.

Note that once a concurrent logic program can be represented by its LDN, the static slicing problem of the program is reduced to be a simple reverse-reachability problem in the net.


(a) A simple FCP program and its LDN
with data, synchronization, communication dependences
Data dependence
Synchronization dependence
Communication dependence
v00
v10
v11
v12
v20
v30
(b) A simple FCP program and its sub-LDN
with selective control dependences
Control dependence
v00
v10
v20
v30
v12

```
V0 :   ?- m(a,Y0)

V1:    m(X1,Y1):-
           h1(X1?,X11),
           h2(X11?,Y1).
V2:    h1(a,b).
V3:    h2(b,c).
```

```
V0 :   ?- insert(s,[1,3,4,List])

V1:    insert(X,[Y|Ys],[Y|Zs]):-
           X>Y |
           insert(X,Ys?,Zs).
V2:    insert(X,[Y|Ys],[X,Y|Ys]):-
           X=<Y|
           true
V3:    insert(X,[],[X]).
```

Fig.4 The LDNs of two sample transformed FCP programs.

Definition 5.4 A *dynamic forward-slicing criterion* for a logic concurrent program is a quadruplet (l, V, H, I), where l is a literal in the program, V is a set of variables used at l, and H is a history of an execution of the program with input I. The *dynamic forward-slice* $DFS(l, V, H, I)$ of a concurrent logic program on a given dynamic forward-slicing criterion (l, V, H, I) consists of all literals in the program that actually affected the beginning or end of execution of l and/or affected the value of v at s in the execution with I that produced H. *Dynamically forward-slicing* a concurrent logic program on a given dynamic forward-slicing criterion is to find the dynamic forward-slice of the program with respect to the criterion.

Debugging is the process of locating, analyzing, and correcting bugs in a program by reasoning about causal relation between bugs and the error detected in the program, and has been a difficult part of software development. Program slicing is useful for debugging in the sense that it potentially allows users to ignore many statements that are irrelative to the error statements. For instance, if a program computes an erroneous value for variable v at statement s, only such statements contained in the slice with v have possibly affect the computation of that value, all statements which are not in the slice can be safely ignored. On the other hand, forward slices are also useful for debugging. For instance, during debugging, statement *s* is found to be incorrect. By making forward slices, we can find all statements which affected by *s*, this may be helpful in that how the error may be corrected.

Like debugging imperative programs, debugging logic programs is also a costly process in logic programming. There are a number of papers proposed and studied in that how to debug a logic program correctly and effectively, see [7]. One of the main achievements is the algorithmic debugging technique, which was first introduced by Shapiro [21]. The algorithmic debugging was the first attempt to lay a theoretical framework for program debugging and to partially automates the task of localizing a bug by comparing the *intended* program behavior with the *actual* program behavior. The intended behavior is obtained by asking the user whether or not a program unit (e.g., a procedure) behaves correctly. Using the answers given by the user, the location of the bug can be determined at the unit level. Following Shapiro's work, a number of studies have been made for Prolog, and have been extended to concurrent logic programs [7]. However, the big problem in algorithmic debugging is the that the number of queries may be very large. To solve this problem, some system uses heuristics to ask users more relevant questions first, and another way to reduce the number of queries is to use partial formal specifications as partial oracles [7]

Since program slicing is very useful in bug localization during debugging a program, and there are many methods which have been proposed and studied for debugging imperative programs by using program slicing techniques [1,11,25], we can expect that once some program slicing methods are proposed, the slicing for logic programs are also useful in debugging logic programs. As an example, when we debug a logic program by an algorithmic debugger, by combining algorithmic debugging with program slicing, the number of queries in the process may be reduced substantially. The similar methods have been proposed for imperative programs, and showed their powerful functions in debugging imperative programs automatically [11,12].

5.2 Testing

Testing is the process that executes a program with the intent to find errors. Although a number of testing motheds have been proposed for imperative programs, there is no testing mothed for concurrent logic programs until now [7]. Since the LDN of a concurrent logic program represents the control flows and data flow properties either within intraprocesses or between interprocesses in the program, it can be used to define the dependence-coverage criteria, i.e., test data selection rules based on covering program dependences, for testing concurrent logic programs. Issues on how to define and evaluate the dependence-coverage criteria should be studied in the future.

5.3 Understanding and Maintenance

One of the problems in software maintenance is that of the ripple effect, i.e., whether a code change in a program will affect the behavior of other codes of the program. To maintain a concurrent logic program, it is necessary to know which literals in which clauses will be affected by a modified literal, and which literals in which clauses will affect a modified literal. On the other hand, to understand the behavior of program, we usually want to know which literals in which clauses might affect a literal of interest, and which literals in which clauses might be affected by the execution of a literal of interest. The slicing and forward-slicing based on LDN of a program can satisfy these requirements.

5.4 Complexity Measurement

Metrics for measuring software complexity have many applications in software engineering. There are a number of complexity metrics proposed and studied for imperative programs [5,27], but, no complexity metric has been proposed for concurrent logic programs until now. Since measuring software complexity is an indispensable process in software development, it is necessary to propose some complexity metrics for measuring concurrent logic programs as well as sequential logic programs. Based on the LDN of a concurrent logic program, we can propose some complexity metrics for measuring concurrent logic programs. For instance, the metric defined by the sum of all primary program dependences holding between literals in a program can be used to measure the total complexity of the program, and the metric defined by the number of all synchronization and communication dependences in a program can be used to measure the complexity of concurrency in the program.

6 Concluding Remarks

We have presented a general framework for dependence analysis for concurrent logic programs, particularly for FCP programs. This framework derives from that of Cheng [3], developed for imperative concurrent programs, that has been naturally extended to analysis dependences for concurrent logic programs. Although here we presented the program dependences and the representation in term of FCP, a simple concurrent logic language, other versions for this framework for more complex concurrent logic languages are easy adaptable because they share their basic execution mechanisms with FCP.

Since dependence-based representations for imperative programs have played an important role in program understanding, debugging, testing, maintenance and so forth, we can expect that the literal dependence nets proposed in this paper are useful in a concurrent logic programming environment. Their significance depends on how we develop the representation themselves and apply them to practices of concurrent logic programming. On the other hand, how to develop a transformation tool that either transforms a concurrent logic program to its LDN efficiently or keeps the size of the LDN as small as possible without losing the necessary dependence information is also an important issue that influences the utilization of the LDN in a practical concurrent logic programming environment.

References

[1] H. Agrawal, R. Demillo, E. Spafford, "Debugging with Dynamic Slicing and Backtracking," Software-Practice and Experience, Vol.23, No.6, pp.589-616, 1993.

[2] J. Chang, A. M. Despain, D. Degroot, "AND-parallelism of Logic Programs Based on a Static Data Dependency Analysis," Digest of Papers, COMPCON 85, IEEE, New York, 1985.

[3] J. Cheng, "Process Dependence Net of Distributed Programs and Its Applications in Development of Distributed Systems," Proceedings of IEEE-CS 17th Annual COMPSAC, pp.231-240, U.S.A., November, 1993.

[4] J. Cheng, "Slicing Concurrent Programs - A Graph-Theoretical Approach,"in P. Fritzson (Ed.) "Automated and Algorithm Debugging," Lecture Notes in Computer Science, No.749, pp.223-240, Springer-Verlag, May, 1993.

[5] J. Cheng, "Complexity Metrics for Distributed Programs," Proceedings of IEEE-CS 4th Annual ISSRE, pp.132-141, U.S.A., November, 1993

[6] S. K. Debray, "Static Inference of Modes and Data Dependencies in Logic Programs," ACM TOPLAS, Vol.11,No.3, pp.418-450, 1987.

[7] M. Ducasse, J. Noye "Logic Programming Environments: Dynamic Program Analysis and Debugging," J. Logic Programming, Vol.19/20, pp.351-384, 1994.

[8] J.Ferrante, K.J.Ottenstein, J.D.Warren, "The Program Dependence Graph and Its Use in Optimization," ACM TOPLAS, Vol.9,No.3, pp.319-349, 1987.

[9] K. B. Gallagher, J. R. Lyle, "Using Program Slicing in Software Maintenance,"IEEE-CS TOSE, Vol.17, No.8, pp.751-761, 1991.

[10] S. Horwitz, T. Reps, "The Use of Program Dependence Graphs in Software Engineering," Proceedings of the 14th ICSE, pp.392-411, 1992.

[11] M. Kamkar, "Interprocedural Dynamic Slicing with Applications to Debugging and Testing," PhD thesis, Linkoping University, 1993.

[12] M. Kamkar, N. Shahmehri, P. Fritzson, "Bug Localization by Algorithmic Debugging and Program Slicing," LNCS, Vol.456, pp.60-74, Springer-Verlag, August 1990.

[13] A. King, P. Soper, "Schedule Analysis of Concurrent Logic Programs," Proceedings of International Joint Conference and Symposium on Logic Programming, pp.478-492, MIT Press, 1992.

[14] B. Korel, "Program Dependence Graph in Static Program Testing," Information Processing Letters, Vol.24, pp.103-108, 1987.

[15] B. Korel, "PELAS - Program Error-Locating Assistant System," IEEE-CS TOSE, Vol.14, No.9, pp.1253-1260, 1988.

[16] D. Kuck, R.Kuhn, B. Leasure, D. Padua. M. Wolfe "Dependence Graphs and Compiler and Optimizations," Conference Record of the 8th Annual ACM Symposium on Principles of Programming Languages, pp.207-208, 1981.

[17] K. J. Ottenstein, L. M. Ottenstein, "The Program Dependence Graph in a software Development Environment," ACM Software Engineering Notes, Vol.9, No.3, pp.177-184, 1984.

[18] A. Podgurski, L. A. Clarke, "A Formal Model of Program Dependences and Its Implications for Software Testing, Debugging, and Maintenance," IEEE-CS TOSE, Vol.16, No.9, pp.965-979, 1990.

[19] C. S. Hsieh, E. A. Unger, R. A. Mata-Toledo, "Using Program Dependence Graphs for Information Flow Control," J. Systems and Software, Vol.17, pp.227-232, 1992.

[20] D. C. Sehr, "Automatic Parallelization of Prolog Programs," PhD thesis, University of Illinois at Urbana-Chanpaign, October, 1992, CSRD Report 1288.

[21] E. Shapiro, "Algorithmic Program Debugging," MIT press, 1983.

[22] E. Shapiro, "The Family of Concurrent logic Programming Languages," ACM Computing Surveys, Vol. 21, No. 3, pp.412-510, September, 1989.

[23] E. Shapiro (Ed.), "Concurrent Prolog: Collected Papers," Vols. 1-2. MIT Press, 1987.

[24] R. Warren, M. Hermenegildo, S. K. Debray, "On the Practicality of Global Flow Analysis of Logic Programs," Proceedings of the Fifth International Conference on Logic Programming, pp.684-699, MIT Press, 1988.

[25] M. Weiser, "Programmers Use Slices When Debugging," CACM, Vol.25, No.7, pp.446-452, 1982.

[26] M. Weiser, "Program Slicing," IEEE-CS TOSE, Vol.10, No.4, pp.352-357, 1984.

[27] H.Zuse, "Software Complexity: Measures and Methods," Walter de Gruyter, 1990.