A Metrics Suite for Concurrent Logic Programs

Jianjun Zhao
Department of Computer Science
and Engineering
Fukuoka Institute of Technology
3-10-1 Wajiro-Higashi, Higashi-ku
Fukuoka 811-02, Japan
zhao@cs.fit.ac.jp

Jingde Cheng   Kazuo Ushijima
Department of Computer Science and
Communication Engineering
Kyushu University
Hakozaki 6-10-1, Fukuoka 812-81, Japan
{cheng,ushijima}@csce.kyushu-u.ac.jp

Abstract

A large body of research in the measurement of software complexity has been focused on imperative programs, but little effort has been made for logic programs. In this paper, a set of complexity metrics for concurrent logic programs are proposed, which are specifically designed to quantify the information flow of concurrent logic programs. These metrics are defined based on the argument dependence net (ADN) of a concurrent logic program which is an arc-classified digraph to explicitly represent various program dependences between arguments in the program. The proposed metrics can be used to measure the complexity of a concurrent logic program from various different viewpoints.

1 Introduction

Concurrent logic programming languages embody all the mechanisms necessary for modeling concurrency, communication, synchronization, and indeterminacy. It is powerful so that many important programming paradigms can be subsumed. For example, as one of the prominent features of concurrent logic programming languages, they allow us to describe interprocess communication with complicated protocols quite easily. Data structure with complicated data flow, such as streams of messages with reply boxes and streams of streams, can be expressed without any extensions to their simple, basic computation model. Furthermore, as a descendent of logic programming, concurrent logic programming also enjoys semantic clarity. Due to these advantages, many knowledge-based systems have been implemented in concurrent logic programming languages. For example, the concurrent logic programming language KL1 developed at Institute for New Generation Computer Technology (ICOT) has been successfully used as a kernel programming language for the Fifth Generation Computer Systems project of Japan and its follow-on project to develop various knowledge processing tools and application systems [11].

While concurrent logic programming languages are powerful for knowledge processing, programs written in concurrent logic programming languages may be very difficult to be understood, tested and maintained due to their lack of explicit control constructs and data flows. As more and more knowledge-based systems have been developed in concurrent logic programming languages, effective development and maintenance of these systems is becoming an increasingly important research topic.

Software metrics aim to measure the inherent complexity of software systems with a view toward predicting the overall project cost and evaluating the quality and effectiveness of the design. Software metrics have many applications in software engineering activities including testing, debugging, maintenance, and project management. In order for effective development and maintenance of concurrent logic programs, it is necessary to define some complexity metrics for estimating the complexity of concurrent logic programs. However, although a large body of research in measurement of software complexity has focused on imperative programs [2, 3, 4, 6, 9, 15], little effort has been made for logic programs. This situation is specially true for concurrent logic programs, since no complexity metric has been proposed for concurrent logic programs until now.

Markusz and Kaposi [5] proposed some complexity metrics for measuring the complexity of sequential logic programs based on counts of lexical items. O'Neal and Edwards [8] presented a language-independent intermediate representation for rule-based programs and defined some bulk measures which estimate complexity by examining aspects of program size, and rule measures which capture complexity based on the ways in which program rules interact with data and other rules. McCauley and Edwards [7] proposed some metrics which are defined based on the extended *and-or* graphic representation (a flowgraph model) of sequential logic programs. Their metrics capture graph attributes such as counts of nodes and edges, nesting depths of nodes, depth of the graph, lengths of the paths, and number of the paths.

In this paper, a set of complexity metrics for concurrent logic programs are proposed, which are specifically designed to quantify the information flow of concurrent logic programs. These metrics are defined based on the argument dependence net of a concurrent logic program which is an arc-classified digraph to explicitly represent various program dependences between arguments in the program. The proposed metrics can be used to measure the complexity of a concurrent logic

program from various different viewpoints. The primary idea of this work has been proposed in [13, 14], and this paper can be regarded as an outgrowth of our previous work.

The rest of the paper is organized as follows. Section 2 briefly introduces the concurrent logic programming language KL1. Section 3 defines three types of primary program dependences between arguments in concurrent logic programs and presents the argument dependence net for concurrent logic programs. Section 4 defines a set of complexity metrics for concurrent logic programs based on program dependences. Concluding remarks are given in Section 5.

2 Preliminaries

We assume that readers are familiar with the basic concepts of logic programs, and in this paper, we will use KL1 [12], a concurrent logic programming language based on Guarded Horn Clauses (GHC), as our target language. This language illustrates the basic mechanisms of concurrent logic programming.

A *term* is a variable, a constant, or a compound term $f(t_1, \ldots, t_n)$ where f is an n-ary function symbol and the t_i are terms, $1 \leq i \leq n$. An *atom* is of the form $p(t_1, \ldots, t_n)$, where p is an n-ary *predicate* symbol and the t_i are terms called *arguments*, $1 \leq i \leq n$. A *literal* is either an *atom* or the negation of an atom. The number of arguments of a literal is called its *arity*.

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, and $G_1, G_2, ..., G_n$ are guard test predicates. H is called the head of the clause, $G_1, G_2, ..., G_n$ are called the *guard* of the clause, and $B_1, B_2, ..., B_m$ are called the *body* of the clause, respectively. ":−", 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 :− $G_1, G_2, ..., G_n | true$. If both the guard and the body are empty the clause is called an *unit clause*, and is written simply as: H. A clause whose body includes exactly one goal is called an *iterative clause*. A clause with only negative literals is referred to as a *goal*. A *procedure* is a set of clauses each of which has the same predicate name and arity. A KL1 *program* is a finite set of guarded clauses.

Figure 1 shows a sample KL1 program called STACK which implements a stack function. The stream of the first argument of procedure stack receives the list of the stack commands, and the stream of the second argument outputs the results. The stack is maintained in the stream of the third argument. The possible input commands are:

- push(D) pushes the value of D into the top of the stack;

- pop gets the value from the top of the stack and outputs it;

- pop(N) gets N values from the top of the stack and outputs them as a list with the length N;

- reverse(N) gets N values from the top of the stack and outputs them as a reverse list with the length N.

In order to formally define primary program dependences between arguments in a concurrent logic program, we distinguish each argument, literal, and clause of the program. We assume that the clauses of a concurrent logic program are denoted by $C_1, C_2, \ldots, C_n$. The literals (including the guard test predicate) of a clause are numbered from left to right, such that the head is numbered by 0, the guard test predicate or the first body literal (if the guard is empty) is numbered by 1, and so on. The arguments of a literal are numbered from left to right by 1, 2, ...

To refer to an argument in a concurrent logic program, we use a method proposed by Boye *et al.* [1] which gives each argument of the program an unique label called *argument position*.

Definition 2.1 (Argument Position) *If C_i is a clause, the position of the kth argument of the jth literal is uniquely defined in the program by the tuple (C_i, j, p, k) such that p is the predicate name of the jth literal of C_i. Such a tuple is called an* argument position.

3 Program Dependences in Concurrent Logic Programs and Their Representation

Program dependences are dependence relationships holding between program elements in a program that are determined by the control flows and data flows in the program. Informally, if the computation of a program element directly or indirectly affects the computation of another program element, there might exist some program dependence between the elements.

Although program dependences are useful for compiler optimizations and development of software engineering tools for imperative programs, most of the work on program dependence analysis for logic programs still aims at generating efficient codes, rather than developing software engineering tools.

In this section, we propose a new program dependence model for concurrent logic programs. We present three types of primary program dependences named *sharing dependence, communication dependence*, and *unification dependence* between arguments in a concurrent logic program and a dependence-based representation named the *argument dependence net* (ADN), which explicitly represents the three types of primary program dependences in the program. As we will show in Section 4, an ADN can be used as an underlying model to develop complexity metrics for concurrent logic programs.

3.1 Primary Program Dependences

In a concurrent logic program, data can be transferred in two ways: either within a clause between two arguments sharing a common variable, or from one clause to another via unification. If we know the mode

```
C1:  stack([push(D)|I],O,S):-
         stack(I,O,[D|S]).

C2:  stack([pop|I],O,S):-
         O=[A|NO], pop(A,S,NS), stack(I,NO,NS).

C3:  stack([pop(N)|I],O,S):-
         O=[L|NO], pop(N,L,S,NS),stack(I,NO,NS).

C4:  stack([reverse(N)|I],O,S):-
         O=[L|NO], rev:rev(R,L),pop(N,R,S,NS),stack(I,NO,NS)

C5:  stack([],O,_):-
         O=[].

C6:  pop(A,[X|S],NS) :-
         A=answer(X),NS=S.

C7:  pop(A,[],NS):-
         A=empty, NS=[].

C8:  pop(N,L,S,OS) :-
         N>0 | L=[X|NL], pop(X,S,NS), NN=N-1,pop(NN,NL,NS,OS)

C9:  pop(0,L,S,OS) :-
         L=[], OS=S.
```

Figure 1: A sample KL1 program.

information of arguments in the program, we can further determine the direction that data is transferred, it means that we can determine data flows in the program. To represent such information in a concurrent logic program, we introduce two types of primary program dependences named *sharing dependence* which reflects data flows in a single clause due to sharing variables, and *communication dependence* which reflects data flows in a single clause due to interprocess communications. Moreover, according to the direction which data is transferred and the mode information of the arguments in a clause, we can further divide the sharing dependences into two categories: *backward-sharing dependence* which reflects the data-flow in a single clause from an argument of a head literal to an argument of a guard test predicate or a body literal, and *forward-sharing dependence* which reflects the data-flow in a single clause from an argument of a guard test predicate or a body literal to an argument of a head literal. In the following we give the formal definitions of these primary program dependences.

We assume that the mode information of each argument position in a concurrent logic program has been inferred by the algorithm proposed by Krishna Rao *et al.*, and has the type, *in* or *out* [10]. We use $\mathbf{M}(\alpha)$ to represent the mode of an argument α and $\Gamma(\alpha)$ to represent the set of all variables which appear in an argument α of the program.

Sharing Dependences

Definition 3.1 (Backward-Sharing Dependence)
Let P be a concurrent logic program and u, v be two literals of P such that u is a body literal (or a guard

test predicate) and v is a head literal of a clause C of P. Let α and α' be two argument positions such that $\alpha = (C, j, u, k)$ $(j \geq 1)$ and $\alpha' = (C, 0, v, k')$. α is backward sharing-dependent on α' iff all of the following conditions hold:

1. *$\Gamma(\alpha) \cap \Gamma(\alpha') \neq \phi$, i.e., there is at least one variable shared by α and α',*

2. *$\mathbf{M}(\alpha) = \mathbf{M}(\alpha') = in$.*

Example. Considering the clause $C8$ of the program shown in Figure 1. Let $\alpha = (C8, 0, pop, 3)$ represent the third argument "S" of the head literal pop(N,L,S,OS) and $\alpha' = (C8, 3, pop, 2)$ represent the second argument "S" of the second body literal pop(X,S,NS) of the clause, and α and α' are all input argument positions, i.e., $\mathbf{M}(\alpha) = \mathbf{M}(\alpha') = in$. α is backward sharing-dependent on α' since there is a shared variable S between α and α'.

Definition 3.2 (Forward-Sharing Dependence) *Let P be a concurrent logic program and u, v be two literals of P such that u is a head literal and v is a body literal (or a guard test predicate). Let α and α' be two argument positions such that $\alpha = (C, 0, u, k)$ and $\alpha' = (C, j, v, k')$ $(j \geq 1)$. α is forward sharing-dependent on α' iff all of the following conditions hold:*

1. *$\Gamma(\alpha) \cap \Gamma(\alpha') \neq \phi$, i.e., there is at least one variable shared by α and α',*

2. *$\mathbf{M}(\alpha) = \mathbf{M}(\alpha') = out$, and*

Example. Considering the clause $C8$ of the program shown in Figure 1. Let $\alpha = (C8, 0, pop, 4)$ represent the fourth argument "OS" of the head literal pop(N,L,S,OS) and $\alpha' = (C8, 5, pop, 4)$ represent the fourth argument "OS" of the fourth body literal pop(NN,NL,NS,OS) of the clause, and α and α' are all output argument positions, i.e., $\mathbf{M}(\alpha) = \mathbf{M}(\alpha') = out$. α is forward sharing-dependent on α' since there is a shared variable OS between α and α'.

Based on the Definitions 3.1 and 3.2, a sharing dependence between two arguments in a clause can be defined as either a backward-sharing dependence or a forward-sharing dependence. We have the following definition.

Definition 3.3 (Sharing Dependence) *Let DEP_{bs} be the set of all backward-sharing dependences of a concurrent logic program P, and DEP_{fs} be the set of all forward-sharing dependences of P. An argument α is sharing-dependent on another argument α' iff $(\alpha, \alpha') \in DEP_{bs} \cup DEP_{fs}$*

Communication Dependences

Notice that in addition to a shared variable between two arguments in the clause, our definition of a sharing dependence in a single clause also requires that one of the arguments belongs to the head literal and the other belongs to the body literal. As a result, our definition excludes the case that two arguments, which belong to two body literals, share a common variable. We introduce a new type of primary program dependences named *communication dependence* to represent this case. The communication dependence also reflects the data flow in a single clause, and more importantly, reflect the fact of interprocess communications between two body literals.

Definition 3.4 (Communication Dependence)
Let P be a concurrent logic program and u, v be two body literals of a clause C of P. Let α and α' be two argument positions such that $\alpha = (C, j, u, k)$ $(j \geq 1)$ and $\alpha' = (C, j', v, k')$ $(j' \geq 1)$. α is communication-dependent on α' iff all of the following conditions hold:

1. *$\Gamma(\alpha) \cap \Gamma(\alpha') \neq \phi$, i.e., there is at least one variable shared by α and α',*

2. *$\mathbf{M}(\alpha) = out$ and $\mathbf{M}(\alpha') = in$, and*

3. *for any $\alpha'' \in \mathbf{A}(w)$ where $w \in V$ is a vertex of N_{APCFN} that represents a head literal, there exists no sharing dependence between α'' and α, and also between α'' and α'.*

Intuitively, a communication dependence exists between two arguments belonging to two body literals in a clause that share a common variable such that the variable does not appear in any head literals of the clause.

Example. Considering the clause $C8$ of the program shown in Figure 1. Let $\alpha = (C8, 5, pop, 3)$ represent the third argument "NS" of the fourth body literal pop(NN,NL,NS,OS) of the clause, and $\alpha' = (C8, 3, pop, 3)$ represent the third argument "NS" of the second body literal pop(X,S,NS), and α is an output argument position, i.e., $\mathbf{M}(\alpha) = out$ and α' is an input argument position, i.e., $\mathbf{M}(\alpha') = in$. α is communication-dependent on α' since there is a shared variable OS between α and α' and the variable NS does not appear in the head literal pop(N,L,S,OS) of the clause.

Unification Dependences

Data as we mentioned above, can be transferred not only in a single clause but also between different clauses via unifications in a concurrent logic programs. The third type of primary program dependence named *unification dependence* are therefore introduced to capture such kind of information and reflect the data flow between arguments in different clauses in a concurrent logic program. Similar to sharing dependences, according to the direction which data is transferred and the mode information of the arguments between two clauses, we also divide the unification dependences into two categories: one which is called *backward-unification dependence* to reflect the data-flow from an input argument position of a head literal of a clause to an input argument position of a body literal of another clause, and the other which is called *forward-unification dependence* to reflect the data-flow from an output argument position of a body literal of a clause to an output argument position of a head literal of another clause.

Definition 3.5 (Backward-Unification Dependence)
Let P be a concurrent logic program and u, v be two literals of clause C of P such that u is a head literal and v is a body literal. Let α and α' be two argument positions such that $\alpha = (C, 0, p, k)$ and $\alpha' = (C', j, p, k)$ $(j \geq 1)$. α is backward unification-dependent on α' iff all of the following conditions hold:

1. *The unification of u and v does not fail, and*

2. *$\mathbf{M}(\alpha) = \mathbf{M}(\alpha') = in$.*

Example. Considering the two clauses $C4$ and $C8$ of the program shown in Figure 1. Let $\alpha = (C8, 0, pop, 3)$ represent the third argument "S" of the head literal pop(N,L,S,NS) of $C8$ and $\alpha' = (C4, 3, pop, 3)$ represent the third argument "S" of the third body literal pop(N,R,S,OS) of $C4$, and α and α' are two input argument positions, i.e., $\mathbf{M}(\alpha) = \mathbf{M}(\alpha') = in$. α is backward unification-dependent on α' since there exists a possible unification of the fourth literal pop(N,R,S,NS) of $C4$ and the head literal pop(N,L,S,OS) of $C8$.

Definition 3.6 (Forward-Unification Dependence) *Let P be a concurrent logic program and u, v be two literals*

of clause C of P such that u is a body literal and v is a head literal. Let α and α' be two argument positions such that $\alpha = (C, j, p, k)$ $(j \geq 1)$ and $\alpha' = (C', 0, p, k)$. α is forward unification-dependent *on α' iff all of the following conditions hold:*

1. *The unification of u and v does not fail, and*

2. $\mathbf{M}(\alpha) = \mathbf{M}(\alpha') = out$.

Example. Considering the two clauses $C4$ and $C8$ of the program shown in Figure 1. Let $\alpha = (C4, 3, pop, 2)$ represent the second argument "R" of the third body literal `pop(N,R,,S,NS)` of $C4$ and $\alpha' = (C8, 0, pop, 2)$ represent the second argument "L" of the head literal `pop(N,L,S,OS)` of $C8$, and α and α' are two output argument positions, i.e., $\mathbf{M}(\alpha) = \mathbf{M}(\alpha') = out$. α is forward unification-dependent on α' since there exists a possible unification of the fourth literal `pop(N,R,S,NS)` of $C4$ and the head literal `pop(N,L,S,OS)` of $C8$.

Based on the Definitions 3.5 and 3.6, a unification dependence between two arguments in two different clauses can be defined as either a backward-unification dependence or a forward-unification dependence. We have the following definition.

Definition 3.7 (Unification Dependence) *Let DEP_{bu} be the set of all backward-unification dependences of a concurrent logic program P, and DEP_{fu} be the set of all forward-unification dependences of P. An argument α in P is* unification-dependent *on another argument α' in P iff $(\alpha, \alpha') \in DEP_{bu} \cup DEP_{fu}$*

3.2 The Argument Dependence Net

Program dependence representations such as the *program dependence graph* (PDG) [9] for sequential imperative programs, have many applications in software engineering activities for imperative programs since one can use such representation to explicitly represent various primary program dependences in the programs. In order to use program dependences to develop software engineering tools for concurrent logic programs, here we present a similar representation for concurrent logic programs. The representation is named the argument dependence net which is an arc-classified digraph to represent all primary program dependences in a concurrent logic program. Roughly speaking, the argument dependence net is an extension of the PDG to the case of concurrent logic programs. The net of a concurrent logic program consists of vertices and arcs such that each vertex represents an unique argument position and each arc represents some dependence relationship between arguments in the program.

Definition 3.8 (Argument Dependence Net)
The argument dependence net (ADN) of a concurrent logic program P is an arc-classified digraph (V_A, Sha, Com, Uni), where:

- V_A *is the set of all argument positions of P;*

- *Sha is the set of sharing dependence arcs such that any $(\alpha, \alpha') \in Sha$ iff α is sharing-dependent on α';*

- *Com is the set of communication dependence arcs such that any $(\alpha, \alpha') \in Com$ iff α is communication-dependent on α';*

- *Uni is the set of unification dependence arcs such that any $(\alpha, \alpha') \in Uni$ iff α is unification-dependent on α'.*

Example. Figure 2 shows the ADN of the program in Figure 1. Because of the limitation of space, we only show a partial ADN of the program which is related to the fourth clause $C4$ of the program. The rest part of the ADN of the program can be drawn easily and is omitted in the figure. Moreover, we use thin solid arcs to represent sharing dependences, thick dashed arcs to represent communication dependences, and thin dashed arcs to represent unification dependences.

4 Metrics Based on Program Dependences

Since program dependences are dependence relationships holding between program elements in a program that are determined by control flows and data flows in the program, they can be regarded as one of intrinsic attributes of programs. Therefore it is reasonable to take program dependences as one of objects for measuring program complexity.

In this section, we define a set of complexity metrics in terms of program dependence relations to measure the complexity of a concurrent logic program from various viewpoints. Once the ADN representation of a concurrent logic program is constructed, the metrics can easily be computed in terms of the representation. The following notations are used for defining these metrics:

- $Du = Sha \cup Com \cup Uni$.

- $\sigma_{[1]=v}(R)$: the selection of binary relation R such that $\sigma_{[1]=v}(R) = \{(v1, v2)|(v1, v2) \in R \text{ and } v1 = v\}$.

- $|A|$: the cardinality of set A.

We first define some metrics for a concurrent logic program that concerns one or more types of program dependences in the program. These metrics can be used to measure various complexities of a concurrent logic program from a general viewpoint.

- $M_1 = |\mathbf{Com}|$: This metric is the number of all primary program dependence concerning interprocess communications in a concurrent logic program. It can be used to measure the complexity of interprocess communications in the program.

- $M_2 = |\mathbf{Uni}|$: This metric is the number of all primary program dependences concerning information flows between clauses via unifications in a
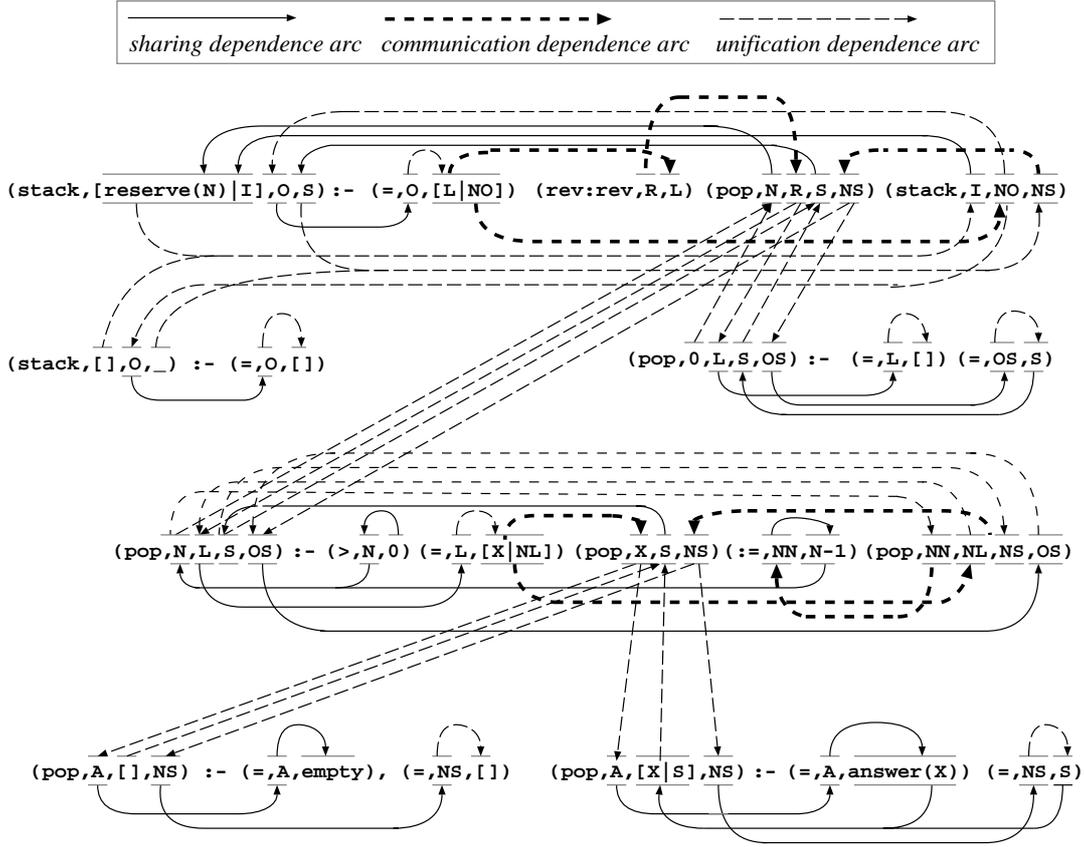
Figure 2: The partial ADN of the program in Figure 1.

concurrent logic program. It can be used to measure the complexity of information flows between clauses in the program.

- $M_3 = |\mathbf{Sha} \cup \mathbf{Com}|$: This metric is the number of all primary program dependences concerning information flows inside clauses due to shared variables in a concurrent logic program. It can be used to measure the complexity of information flows inside clauses of the program.

- $M_4 = |\mathbf{Du}|$: This metric is the number of all primary program dependences in a concurrent logic program. It can be used to measure the total complexity of the program.

Example. The following example shows how to compute the values of the metrics defined above based on the ADN of a concurrent logic program. To compute these metrics, we need to count the numbers of each type of dependence arcs respectively or the numbers of combination of the dependence arcs in the ADN. The values of the metrics for the sample program in Figure 1 are as follows:

- $M_1 = |Com| = 8$

- $M_2 = |Uni| = 24$

- $M_3 = |Sha \cup Com| = |Sha| + |Com| = 23 + 8 = 31$

- $M_4 = |Du| = |Sha \cup Com \cup Uni| = |Sha| + |Com| + |Uni| = 23 + 8 + 24 = 55$

In maintenance phases, when we have to modify an argument (literal), usually, we intend to know the information about how the modified argument (literal) intersect with other arguments (literals) in the program. This kind of information is very useful because it can tell us if the modified argument (literal) is a special point that connects with its environment more closely than other arguments (literals). If so, that means it is difficult to implement changes to the argument (literal) due to a large number of potential effects on other arguments (literals). We call such an argument (literal) the "most easily affected argument (literal)" of the program. To capture such attribute, we can define the following metric:

- $M_5 = \mathbf{max}\{|\sigma_{[1]=\alpha(\mathbf{D_u})}| \| \alpha \in \mathbf{V_A}\}$: This metric is the maximal number of arguments that an

argument is somehow dependent on in a concurrent logic program. It can be used to determine the "most easily affected" argument(s) in the program.

Example. The following example shows how to determine the "most easily affected" argument(s) in the program based on its ADN. To do so, we should find those vertices that contains the maximal numbers of dependence arcs in the ADN. By investigating the ADN in Figure 2 of the sample program in Figure 1, we can get $M_5 = max\{|\sigma_{[1]=\alpha(D_u)}|\,|\,\alpha \in V_A\}$ $= 3$, and the arguments which have the maximal value 3 in the program are as follows (represented by their argument positions): $(C1, 0, stack, 1)$, $(C1, 1, =, 2)$, $(C1, 3, pop, 1)$, $(C1, 3, pop, 2)$, $(C1, 3, pop, 3)$, $(C1, 3, pop, 4)$, $(C1, 4, stack, 2)$, $(C8, 0, pop, 1)$, $(C8, 0, pop, 2)$, $(C8, 0, pop, 4)$, $(C8, 2, =, 2)$, $(C8, 3, pop, 1)$, $(C8, 3, pop, 2)$, $(C8, 3, pop, 3)$.

As we observed, all the metrics defined above are absolute metrics. In general, the larger is a metric of a program, the more complex is the program. Moreover, some relative metrics should also be considered since they can measure the complexity of the program from different viewpoint. These new relative metrics can easily be obtained through dividing the above metrics by $|D_u|$ and $|V_A|$. Some relative metrics defined by this way are $|\mathbf{Uni}|/|\mathbf{D_u}|$, $|\mathbf{Com}|/|\mathbf{D_u}|$, $|\mathbf{Sha} \cup \mathbf{Com}|/|\mathbf{D_u}|$, and $\mathbf{max}\{|\sigma_{[1]=\alpha(\mathbf{D_u})}|\,|\,\alpha \in \mathbf{V_A}\}/|\mathbf{V_A}|$.

5 Concluding Remarks

In this paper we proposed a set of complexity metrics for concurrent logic programs which are specifically designed to quantify the information flow of concurrent logic programs. These metrics are defined based on the *argument dependence net* (ADN) of a concurrent logic program which is an arc-classified digraph to explicitly represent various program dependences between arguments in the program. The proposed metrics can be used to measure the complexity of a concurrent logic program from various different viewpoints. Although here we presented the approach in term of KL1, a simple concurrent logic language, other versions for this approach for other concurrent logic programming languages are easily adaptable because they share their basic execution mechanisms with KL1.

To demonstrate the effectiveness of the proposed metrics, we have implemented a program dependence analysis tool for KL1 programs [14]. The next step for us is to perform some experiments and collect data for evaluation. We hope a primary evaluation of these metrics will be available soon.

References

[1] J. Boye, J. Paakki, and J. Maluszyński, "Synthesis of Directionality Information for Functional Logic Programs," *Proceedings of the Third International Workshop on Static Analysis*, LNCS 724, Springer-Verlag, pp.165-177, 1993.

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

[3] N. E. Fenton, S. L. Pfleeger, "*Software Metrics: A Rigorous & Practical Approach,*" International Thomson Computer Press, 1997.

[4] M. Halstead, "*Elements of Software Science,*" Elsevier, North Holland, 1977.

[5] Z. Markusz and A. A. Kaposi, "Complexity Control in Logic-Based Programming," *The Computer Journal*, Vol.28, No.5, pp.487-495, 1985.

[6] T. McCabe, "A Complexity Measure," *IEEE Transaction on Software Engineering*, Vol.2, No.4, pp.308-320, 1978.

[7] R. A. McCauley and W. R. Edwards, "Analysis and Measurement Techniques for Logic-Based Languages," in A. Melton (ed.) *Software Measurement*, pp.93-113, International Thomson Computer Press, 1996.

[8] M. B. O'Neal and W. R. Edwards, "Complexity Measures of Rule-Based Programs," *IEEE Transaction on Knowledge and Data Engineering*, Vol.6, No.5, pp.669-680, 1994.

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

[10] M. R. K. Krishna Rao, D. Kapur, and R. K. Shyamasundar, "Proving Termination of GHC Programs," *Proceedings of the Tenth International Conference on Logic Programming*, pp.720-736, MIT Press, 1993.

[11] S. Uchida, "General Report of the FGCS Follow-on Project," *Proceedings of the International Symposium on Fifth Generation Computer Systems*, pp.1-9, ICOT, November 1994.

[12] K. Ueda and T. Chikayama, "Design of the Kernel Language for the Parallel Inference Machine," *The Computer Journal*, Vol.33, No.6, pp.494-500, 1990.

[13] Zhao, J. Cheng, and K. Ushijima, "Program Dependence Analysis of Concurrent Logic Programs and Its Applications," *Proceedings of 1996 International Conference on Parallel and Distributed Systems*, pp.282-291, IEEE Computer Society Press, 1996.

[14] J. Zhao, "Program Dependence Analysis of Concurrent Logic Programs and Its Applications," Ph.D Thesis, Dept. of Computer Science and Communication Engineering, Kyushu University, Japan, December 1996.

[15] J. Zhao, "Toward Measuring the Complexity of Software Architectures," *Research Bulletin of Fukuoka Institute of Technology*, Vol.30, No.2, March 1998 (to appear).