プログラミング-言語・基礎・実践- 8－3
(1992. 8. 20)

従属性に基づく並行プログラム表現

程 京 徳

九州大学 情報工学科

内容梗概

プログラム従属性とは、プログラムにおける制御の流れとデータの流れによって決められた、プログラムの実行文の間に存在する従属関係である。従って、それを用いてプログラムの挙動を表現することができる。従属性に基づくプログラム表現は、プログラムの最適化、並列化、理解、テスト、デバッグ、保守など多くのソフトウェア開発活動に応用されることができる。しかし、従来提案された従属性に基づくプログラム表現は、全て逐次プログラムのためのものであり、並行プログラムのために提案された従属性に基づくプログラム表現がいままでになかった。その主な原因は、プロセス間の同期と通信のような並行プログラムの挙動を表現するために、従来提案された制御従属性とデータ従属性だけを用いるのが不十分であることであろう。本論文は、従来提案された制御従属性とデータ従属性に加えて、選択従属性、同期従属性と通信従属性という三種類のプログラム従属性を新たに提案し、それらに基づいて、並行プログラムのプロセス従属ネットとプロセス支配ネットという二種類の従属性に基づくプログラム表現を提案する。また、プロセス従属ネットとプロセス支配ネットの応用も論ずる。

Dependence-Based Representations of Concurrent Programs

Jingde Cheng

Dept. of Computer Science and Communication Engineering, Kyushu University

Abstract

Program dependences are dependence relationships holding between statements in a program that are determined by control flow and data flow in the program, and therefore, they can be used to represent the program's behavior. A dependence-based program representation has many applications in various software development activities including program optimization, parallelization, understanding, testing, debugging, and maintenance. However, although a number of dependence-based program representations have been proposed for sequential programs, until recently, there is no dependence-based representation proposed for concurrent programs. A major reason for this situation is that using the usual control dependence and data dependence in sequential programs is inadequate to represent the full behavior of a concurrent program because of the existence of interprocess synchronization and communication in the program. In addition to the usual control and data dependences, this paper proposes three new types of basic program dependences in concurrent programs, named the selection dependence, synchronization dependence, and communication dependence, and two new program representations for concurrent programs, named the Process Dependence Net (PDN) and Process Influence Net (PIN). The paper also shows some potential applications of the representations in development of concurrent software.

1. Introduction

Program dependences are dependence relationships holding between statements in a program that are determined by control flow and data flow in the program, and therefore, they can be used to represent the program's behavior. There are two types of basic program dependences proposed in the literature, i.e., the control dependence that is determined by control flow of a program, and the data dependence that is determined by data flow of a program. Informally, a statement S is control dependent on the control predicate C of a conditional branch statement (e.g., an if statement or while statement) if the value of C determines whether S is executed or not. A statement S_2 is data dependent on a statement S_1 if the value of a variable computed at S_1 directly or indirectly has influence on the value of a variable computed at S_2 [8,17].

Since grasping program dependences between statements of a program is indispensable to many software development activities, a dependence-based program representation has many applications in various software development activities including program optimization, parallelization, understanding, testing, debugging, and maintenance [1,8,9,11,13,16-19]. For example, program dependence graph [8,16], which explicitly represents both control and data dependences in a sequential program, has been developed as an important program representation tool used in compiler construction and software testing, debugging, and maintenance. However, although a number of dependence-based program representations have been proposed for sequential programs, until recently, there is no dependence-based representation proposed for concurrent programs.

In general, a concurrent program consists of a number of processes, and therefore, it has multiple control flows and multiple data flows. These control flows and data flows are not independent because of the existence of interprocess synchronization and communication in the program. Moreover, a process in a concurrent program is usually allowed to nondeterministically select a communication partner among a number of processes ready for communication with the process. It is obvious that only using the usual control and data dependences is inadequate to represent the full behavior of a concurrent program.

In addition to the usual control and data dependences, this paper proposes three new types of basic program dependences in concurrent programs, named the *selection dependence*, *synchronization dependence*, and *communication dependence*, which are determined by interaction between multiple control flows and multiple data flows in the program, and two new program representations for concurrent programs, named the *Process Dependence Net* (PDN) and *Process Influence Net* (PIN), which are arc-labeled digraphs to represent the five types of basic program dependences in the programs explicitly. The paper also shows some potential applications of the representations in development of concurrent software.

2. 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 $(v_1,v_2) \in A$, v_1 is called the *initial vertex* of the arc and said to be *adjacent to* v_2, and v_2 is called the *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 a vertex v, denoted in-degree(v), is the number of predecessors of v, and the *out-degree* of a vertex v, denoted out-degree(v), is the number of successors of v. A *simple digraph* is a digraph (V, A) such that $(v,v) \notin A$ for any $v \in V$. □

Definition 2.2 An *arc-labeled digraph* is an n-tuple $(V, A_1, A_2, ..., A_{n-1})$ such that every (V, A_i) $(i = 1, ..., n-1)$ is a digraph. A *simple arc-labeled digraph* is an arc-labeled digraph $(V, A_1, A_2, ..., A_{n-1})$ such that $(v,v) \notin A_i$ $(i = 1, ..., n-1)$ for any $v \in V$. □

Definition 2.3 A *path* in a digraph (V, A) or an arc-labeled digraph $(V, A_1, A_2, ..., A_{n-1})$ is a sequence of arcs $(a_1, a_2, ..., a_\ell)$ such that the terminal vertex of a_i is the initial vertex of a_{i+1} for $1 \leq i \leq \ell - 1$, where $a_i \in A$ $(1 \leq i \leq \ell)$ or $a_i \in A_1 \cup A_2 \cup ... \cup A_{n-1}$ $(1 \leq i \leq \ell)$, and ℓ $(\ell \geq 1)$ 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 v_I-v_T path for short. A path in a digraph or an arc-labeled digraph is said to be *simple* if it does not include the same arc twice. A path in a digraph or an arc-labeled digraph is said to be *elementary* if it does not include the same vertex twice. □

Definition 2.4 A *nondeterministic parallel control flow net* is a 10-tuple $(V, N, P_F, P_J, A_C, A_N, A_{P_F}, A_{P_J}, s, t)$, where (V, A_C, A_N, A_P) is a simple arc-labeled digraph such that $A_C \subseteq V \times V$, $A_N \subseteq N \times V$, $A_{P_F} \subseteq P_F \times V$, $A_{P_J} \subseteq V \times P_J$, $N \subset V$ is a set of elements, called *nondeterministic selection vertices*, $P_F \subset V$ is a set of elements, called *parallel execution fork vertices*, $P_J \subset V$ is a set of elements, called *parallel execution join 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 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 $(v_1,v_2) \in A_C$ is called a *sequential control arc*, any arc $(v_1,v_2) \in A_N$ is called a *nondeterministic selection arc*, and any arc $(v_1,v_2) \in A_{P_F} \cup A_{P_J}$ is called a *parallel execution arc*. □

A usual (deterministic and sequential) control flow graph can be regarded as a special case of nondeterministic parallel control flow nets where

both nondeterministic vertex set N and parallel vertex set P and are the empty set.

Definition 2.5 Let u and v be any two vertices in a nondeterministic parallel control flow net. u *forward dominates* v iff every path from v to t contains u; u *properly forward dominates* v iff u forward dominates v and $u \neq v$; u *strongly forward dominates* v iff u forward dominates v and there exists an integer k ($k \geq 1$) such that every path from v whose length is greater than or equal to k contains u; u is the *immediate forward dominator* of v iff u is the first vertex that properly forward dominates v in every path from v to t. □

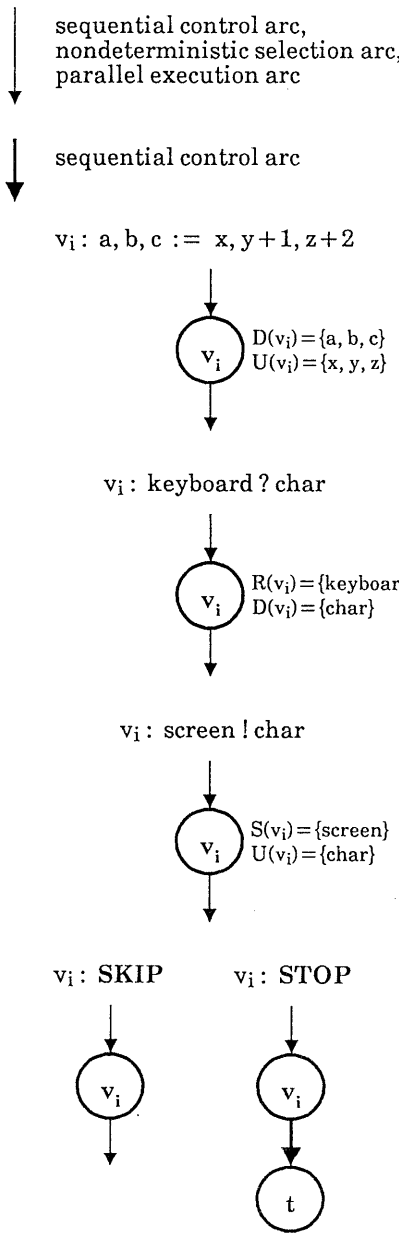

Fig. 1 Representations of primitive processes of Occam 2 programs

Definition 2.6 A *nondeterministic parallel definition-use net* is a 7-tuple (N_C, Σ_V, D, U, Σ_C, S, R), where N_C is a nondeterministic parallel control flow net ($V, N, P_F, P_J, A_C, A_N, A_{P_F}, A_{P_J}, s, t$), Σ_V is a finite set of symbols, called *variables*, D: $V \rightarrow 2^{\Sigma_V}$ and U: $V \rightarrow 2^{\Sigma_V}$ 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 Σ_C. □

Note that all above definitions are graph-theoretical, and therefore, they are independent of programming languages.

3. Nondeterministic Parallel Definition-Use Nets of Occam 2 Programs

We select Occam 2 [12] programs as the target programs for showing how to represent a concurrent program by a nondeterministic parallel definition-use net.

Fig. 1 shows various primitive processes of Occam 2 programs and their representations in a nondeterministic parallel definition-use net. Note that each unspecified arc in Fig. 1 should be replaced by a sequential control arc, nondeterministic selection arc, or parallel execution arc in a complete nondeterministic parallel definition-use net. Fig. 2 shows various constructions of Occam 2 programs and their representations in a nondeterministic parallel definition-use net.

In fact, Fig. 1 and Fig. 2 give the rules that can be used to transform an Occam 2 program into its nondeterministic parallel definition-use net. As an example, Fig. 3 shows a fragment of Occam 2 program and Fig. 4 shows its nondeterministic parallel definition-use net.

4. Basic Program Dependences in Concurrent Programs

Based on the nondeterministic parallel definition-use net of a concurrent program, we can define various types of basic program dependences in the program. 2

Definition 4.1 Let ($V, N, P_F, P_J, A_C, A_N, A_{P_F}, A_{P_J}, s, t$) be the nondeterministic parallel control flow net of a concurrent program, $u \in V$, and $v \in ((V - (N \cup P_F \cup P_F))$. u is *directly strongly control dependent* on v iff there exists a path from v to u such that the path does not contain the immediate forward dominator of v. u is *directly weakly control dependent* on v iff v has two successors v' and v" such that u strongly forward dominates v' but does not strongly forward dominates v". □

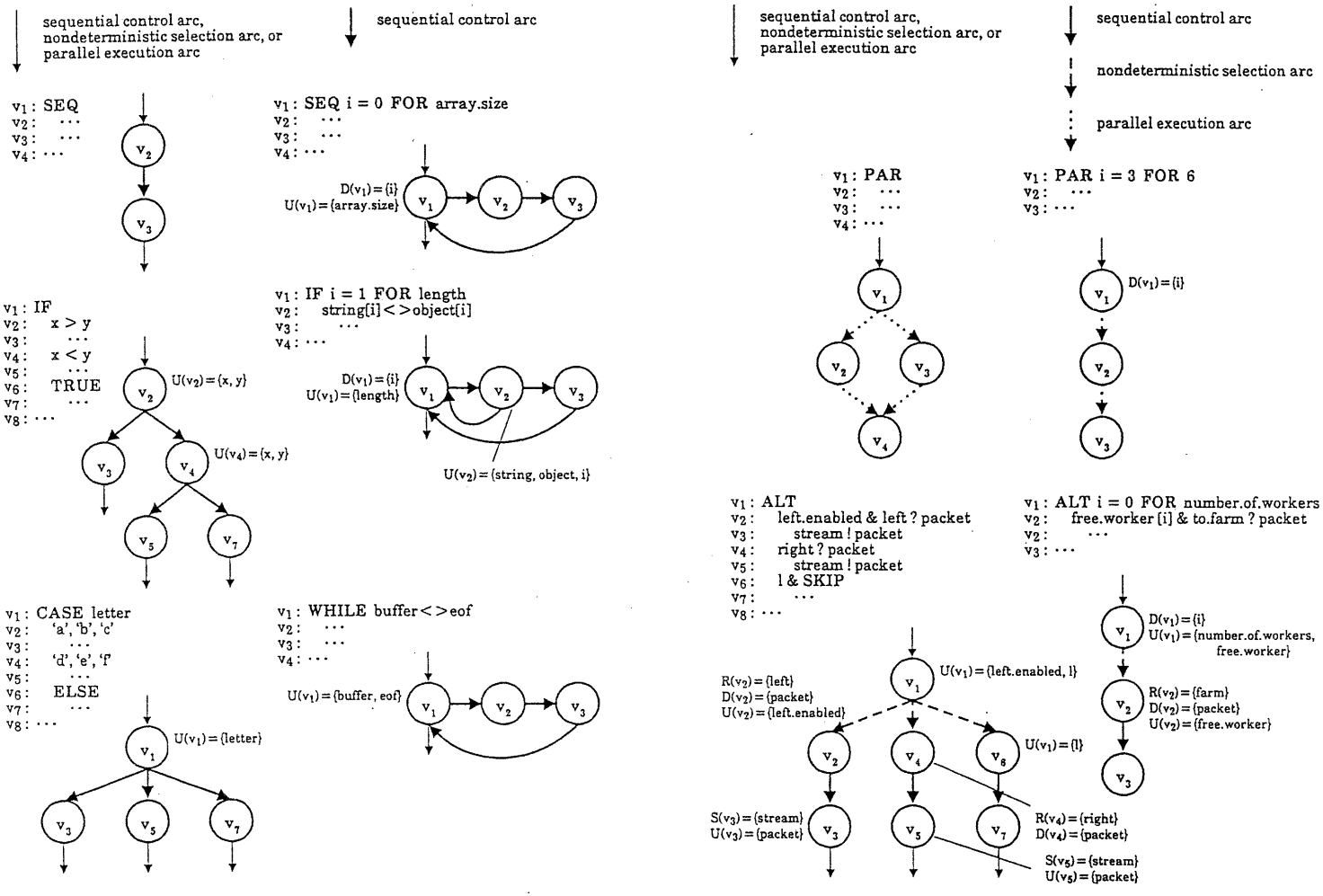

Fig. 2 Representations of constructions of Occam 2 programs in their nondeterministic parallel definition-use nets

```
v1  PAR
v2      SEQ
v3          input1 ? x; y
v4          IF
v5              (x<0) OR (y<0)
v6                  error1 ! 14 :: "minus operator"
v7                  ce ! 0.0
v8              y=0
v9                  error1 ! 11 :: "zero divide"
v10                 ce ! 0.0
v11             TRUE
v12                 c ! x/y
v13     SEQ
v14         input2 ? n
v15         sum := 0
v16         WHILE n<>0
v17             input2 ? data
v18             sum, n := sum+data, n-1
v19         ALT
v20             c ? factor
v21                 result ! sum*factor
v22             ce ? factor
v23                 error2 ! 14 :: "invalid factor"
v24 STOP
```

Fig. 3 A fragment of Occam 2 program

Note that according to the above definition, if u is directly strongly control dependent on v, then u is also directly weakly control dependent on v, but the converse is not necessarily true.

Informally, if u is directly strongly control dependent on v, then v must have at least two successors v' and v" such that if the branch from v to v' is executed then u must be executed, while if the branch from v to v" is executed then u may not be executed. If u is directly weakly control dependent on v, then v must have two successors v' and v" such that if the branch from v to v' is executed then u is necessarily executed within a fixed number of steps, while if the branch from v to v" is executed then u may not be executed or the execution of u may be delayed indefinitely. The difference between strong and weak control dependences is that the latter reflects a dependence between an exit condition of a loop and a statement outside the loop that may be executed after the loop is exited, but the former does not.

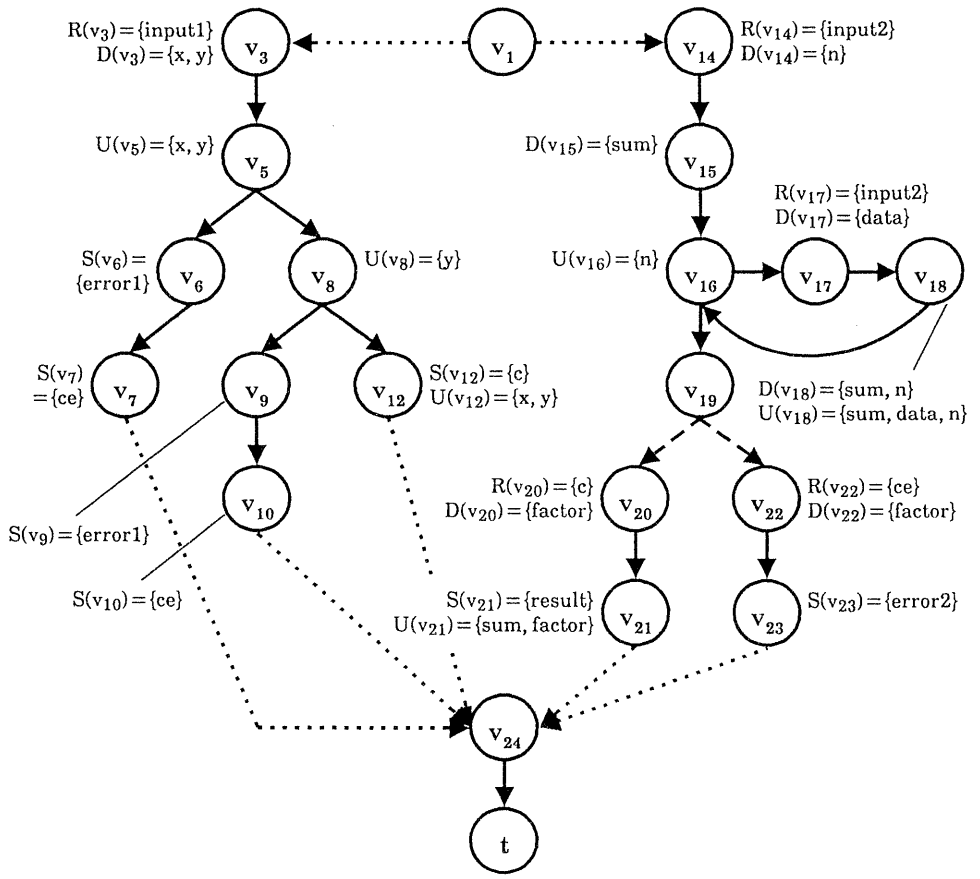

Fig. 4 The nondeterministic parallel definition-use net of Occam 2 program in Fig. 3

For example, in Fig. 4, vertices v_6, v_7, and v_8 are directly strongly (weakly) control dependent on vertex v_5, vertices v_9, v_{10}, and v_{12} are directly strongly (weakly) control dependent on vertex v_8, vertices v_{17}, v_{18}, and v_{16} are directly strongly (weakly) control dependent on vertex v_{16}, and vertex v_{19} is directly weakly control dependent on vertex v_{16} but not directly strongly control dependent on v_{16}.

Definition 4.2 Let $(V, N, P_F, P_J, A_C, A_N, A_{P_F}, A_{P_J}, s, t)$ be the nondeterministic parallel control flow net of a concurrent program, $u \in V$, and $v \in N$. u is *directly selection dependent* on v iff there exists a path from v to u such that the path does not contain the immediate forward dominator of v. □

Informally, if u is directly selection dependent on v, then v must have some successors such that if the branch from v to one of the successors is executed then u must be executed, while if other branch is executed then u may not be executed.

For example, in Fig. 4, vertices $v_{20} \sim v_{23}$ are directly selection dependent on vertex v_{19}.

Definition 4.3 Let $(N_C, \Sigma_V, D, U, \Sigma_C, S, R)$ be the nondeterministic parallel definition-use net of a concurrent program, and $P=((v_1,v_2), (v_2,v_3), \cdots, (v_{n-1},v_n))$ be a path in N_C. Then $D(P)=D(v_1)\cup D(v_2)\cup\ldots\cup D(v_n)$. □

Definition 4.4 Let $(N_C, \Sigma_V, D, U, \Sigma_C, S, R)$ be the nondeterministic parallel definition-use net of a concurrent program, and u and v be any two vertices in N_C. u is *directly data dependent* on v iff there is a path P from v to u in N_C such that $(D(v)\cap U(u)) - D(P) \neq \Phi$. □

Informally, if u is directly data dependent on v, then the value of a variable computed at v directly has influence on the value of a variable computed at u.

For example, in Fig. 4, vertices v_5, v_8, and v_{12} are directly data dependent on vertex v_3, vertices v_{16} and v_{18} are directly data dependent on vertices v_{14} and v_{18}, vertex v_{18} is directly data dependent on vertices v_{15} and v_{17}, and vertex v_{21} is directly data dependent on vertices v_{18} and v_{20}.

There are some efficient algorithms to compute the control and data dependences in a sequential program based on the control flow graph of the program [2,8,11,17]. Those algorithms can also be modified to compute the control, selection, and data dependences in a concurrent program.

Definition 4.5 Let $(N_C, \Sigma_V, D, U, \Sigma_C, S, R)$ be the nondeterministic parallel definition-use net of a concurrent program, and u and v be any two vertices in N_C. u is *directly synchronization dependent* on v iff any of the following conditions holds :

1) (v,u) is a parallel execution arc,

2) $S(v)=R(u)$, and

3) there exists a vertex v' such that v' is directly synchronization dependent on v, u properly forward dominates v', and $S(v")=\Phi$ and $R(v")=\Phi$ for any vertex v" in any v'-u path. □

Informally, if u is directly synchronization dependent on v, then the start or termination of execution of v directly determines whether or not the execution of u starts or terminates.

For example, in Fig. 4, vertices v_3, v_5, v_{14}, v_{15}, and v_{16} are directly synchronization dependent on vertex v_1, vertices v_{20} and v_{21} are directly synchronization dependent on vertex v_{12}, vertices v_{22} and v_{23} are directly synchronization dependent on vertices v_7 and v_{10}, and vertex v_{24} is directly synchronization dependent on vertices v_7, v_{10}, v_{12}, v_{21}, and v_{23}.

Definition 4.6 Let $(N_C, \Sigma_V, D, U, \Sigma_C, S, R)$ be the nondeterministic parallel definition-use net of a concurrent program, and u and v be any two vertices in N_C. u is *directly communication dependent* on v iff any of the following conditions holds:

1) there exist two vertices v' and v" such that u is directly data dependent on v', $R(v')=S(v")$, and v" is directly data dependent on v, and

2) there exist three vertices v', v", and v'" such that u is directly data dependent on v', v" is a successor of v', $R(v")=S(v'")$, and v'" is directly data dependent on v. □

Informally, if u is directly possibly communication dependent on v, then the value of a variable computed at v possibly directly has influence on the value of a variable computed at u.

For example, in Fig. 4, vertex v_{21} is directly communication dependent on vertex v_3.

5. Process Dependence Net and Process Influence Net

We now propose two dependence-based representations for concurrent programs, which are arc-labeled digraphs to represent the five types of basic program dependences in the programs explicitly. .

Definition 5.1 The *Process Dependence Net* (PDN) of a concurrent program is an arc-labeled digraph (**V, Con, Sel, Dat, Syn, Com**), where **V** is the vertex set of the nondeterministic parallel control flow net of the program, **Con** is the set of control dependence arcs such that any $(u,v) \in$ **Con** iff u is directly weakly control dependent on v, **Sel** is the set of selection dependence arcs such that any $(u,v) \in$ **Sel** iff u is directly selection 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 dependent 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. □

Note that the above definition of PDN is not constructive. A transformation algorithm is indispensable in order to transform a concurrent program into its PDN.

For example, Fig. 5 shows the PDN of Occam 2 program in Fig. 3.

Definition 5.2 The *Process Influence Net* (PIN) of a concurrent program is an arc-labeled digraph (**V, Con, Sel, Dat, Syn, Com**), where **V** is the vertex set of the nondeterministic parallel control flow net of the program, **Con** is the set of control influence arcs such that any (u,v)∈**Con** iff v is directly weakly control dependent on u, **Sel** is the set of selection influence arcs such that any (u,v)∈**Sel** iff v is directly selection dependent on u, **Dat** is the set of data influence arcs such that any (u,v)∈**Dat** iff v is directly data dependent on u, **Syn** is the set of synchronization influence arcs such that any (u,v)∈**Syn** iff v is directly synchronization dependent on u, and **Com** is the set of communication influence arcs such that any (u,v)∈**Com** iff v is directly communication dependent on u. □

It is obvious that the PIN of a concurrent program is a "reverse" of the PDN of that program.

6. Applications of PDN and PIN

As dependence-based representations of concurrent programs, the PDN and PIN have many potential applications in concurrent programming.

The most direct application of PDN is slicing concurrent programs because the explicit representation of various program dependences in a

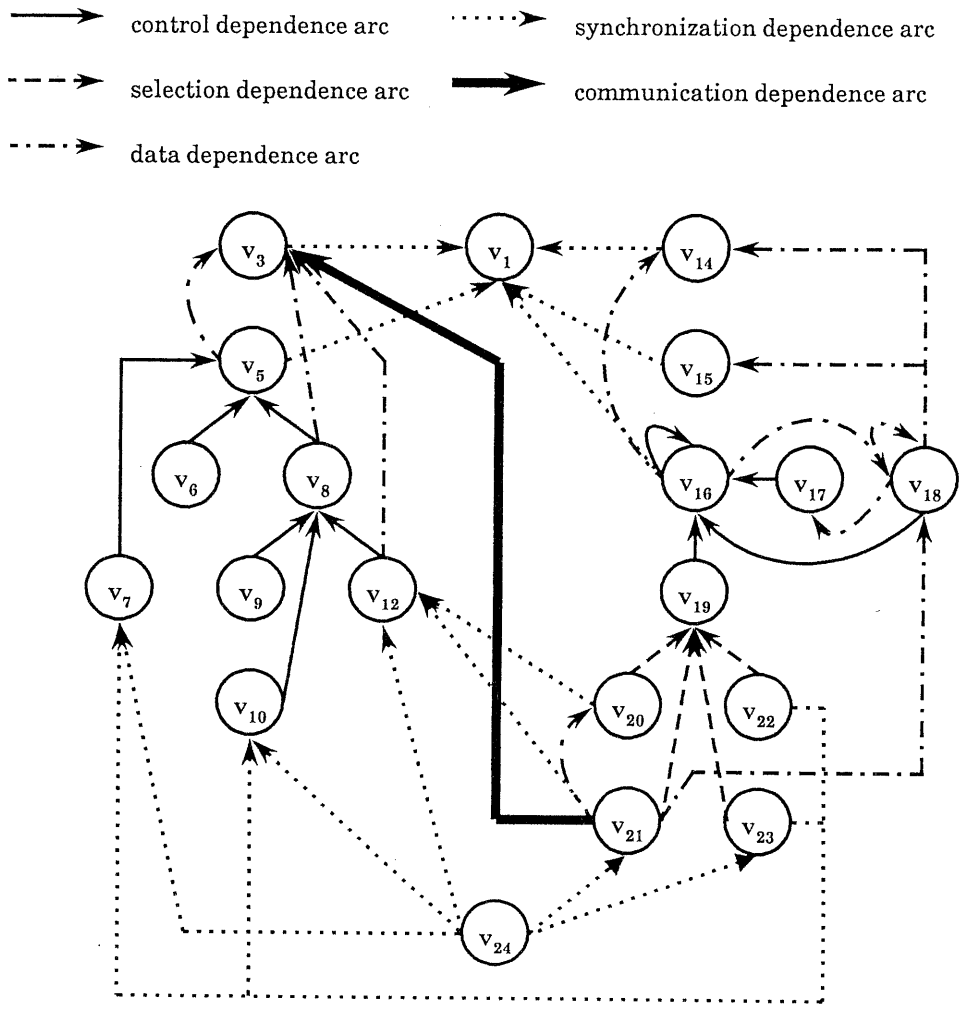

Fig. 5 The PDN of Occam 2 program in Fig. 3

concurrent program makes the PDN very ideal for constructing slices of the program.

Definition 6.1 A *static slicing criterion* of a concurrent program is a 2-tuple (s, V), where s is a statement in the program and V is a set of variables used at s. The *static slice* **SS**(s, V) of a concurrent program on a given static slicing criterion (s, V) consists of all statements in the program that possibly affect the beginning or end of execution of s and/or affect the values of variables in V. *Statically slicing* a concurrent program on a given static slicing criterion is to find the static slice of the program with respect to the criterion. □

Note that there is a difference between the concept of program slice given above for concurrent programs and that given in the literature [1,9,11,13,19] for sequential programs, i.e., the above definition includes a condition on the beginning or end of execution of s, and therefore, it is meaningful even if V is the empty set. This makes the concept useful in analysis of deadlocks and livelocks in concurrent programs.

It is obvious that once a concurrent program is represented by its PDN, the static slicing problem of the program is simply a vertex reachability problem in the net.

Definition 6.2 A *dynamic slicing criterion* of a concurrent program is a quadruplet (s, V, H, I), where s is a statement in the program, V is a set of variables used at s, and H is a history of an execution of the program with input I. The *dynamic slice* **DS**(s, V, H, I) of a concurrent program on a given dynamic slicing criterion (s, V, H, I) consists of all statements in the program that actually affected the beginning or end of execution of s and/or affected the values of variables in V in the execution with I that produced H. *Dynamically slicing* a concurrent program on a given dynamic slicing criterion is to find the dynamic slice of the program with respect to the criterion. □

Note that for a concurrent program, two different executions with the same input may produce different behavior and histories because of unpredictable rates of processes and existence of nondeterministic selection statements in the program. This is the reason why we use a program's execution history H produced with an input I to define the concept of dynamic slicing criterion.

The dynamic slicing problem of a concurrent program can be reduced to the vertex reachability problem in its PDN with the program's execution history information that can be collected by an execution monitor [7].

Definition 6.3 A *static forward-slicing criterion* of a concurrent program is a 2-tuple (s, v), where s is a statement in the program and v is a variable defined at s. The *static forward-slice* **SFS**(s, v) of a concurrent program on a given static forward-slicing criterion (s, v) consists of all statements in the program that would be affected by the beginning or end of execution of s and/or affected by the value of v at s. *Statically forward-slicing* a concurrent program on a given static forward-slicing criterion is to find the static forward-slice of the program with respect to the criterion. □

Definition 6.4 A *dynamic forward-slicing criterion* of a concurrent program is a quadruplet (s, v, H, I), where s is a statement in the program, v is a variable defined at s, and H is a history of an execution of the program with input I. The *dynamic forward-slice* **DFS**(s, v, H, I) of a concurrent program on a given dynamic forward-slicing criterion (s, v, H, I) consists of all statements in the program that are actually affected by the beginning or end of execution of s and/or affected by the value of v at s in the execution with I that produced H. *Dynamically forward-slicing* a concurrent program on a given dynamic forward-slicing criterion is to find the dynamic forward-slice of the program with respect to the criterion. □

It is obvious that once a concurrent program is represented by its PIN, the static forward-slicing problem of the program is simply a vertex reachability problem in the net, and the dynamic forward-slicing problem of the program can be reduced to the vertex reachability problem in the net with the program's execution history information.

We now discuss applications of the PDN and PIN in development of concurrent software. Some of them are direct applications of the PDN and PIN and others are applications of program slicing based on the PDN and PIN.

In understanding a concurrent program, we often want to know which statements in which processes might affect a statement of interest and/or which statements in which processes would be affected by the execution of a statement of interest. Obviously, once a concurrent program is represented by its PDN and PIN, the needs can be satisfied by slicing the program based on its PDN and/or forward-slicing the program based on its PIN.

Since the PDN and PIN of a concurrent program represents both control and data flow properties in every process and synchronization and communication properties in interprocess interaction in the program, it can also be used to define dependence-coverage criteria, i.e., test data selection rules based on covering program dependences, for testing concurrent programs. How to define and evaluate the dependence-coverage criteria is a challenging research problem since there is no effective coverage criterion for testing concurrent programs until now.

Static and dynamic slicing are useful in concurrent program debugging because they can be used to find all statements that possibly or actually caused the erroneous behavior of the execution of a concurrent program where an error occurs.

A program error is a difference between a program's actual behavior and the behavior required

by the specification of the program. A "bug" relative to an error is a cause of the error. Debugging is the process of locating, analyzing, and correcting bugs in a program by reasoning about causal relations between bugs and the error detected in the program. It begins with some indication of the existence of an error, repeats the process of developing, verifying, and modifying hypotheses about the bug(s) causing the error until the location of the bug(s) is determined and the nature of the bug(s) is understood, corrects the bug(s), and ends in a verification of the removal of the error [3,15]. In general, about 95% of effort in debugging has to be spent on locating and understanding bug because once a bug is located and its nature is understood, its correction is often easy to do [15]. Therefore, the most important problem in debugging is how to know which statements possibly and/or actually cause the erroneous behavior of the execution where an error occurs.

Debugging a concurrent program is more difficult than debugging a sequential program since a concurrent program in general has multiple control flows, multiple data flows, synchronization and communication among processes, and nondeterministic selections. Most current debugging methods and tools for concurrent programs provide programmers with only facilities to extract information from programs and display it in textual or visual forms, but no facilities to support the localization, analysis, and correction of bug in an automatic or semi-automatic manner [14]. Until recently, there is no systematic method used for bug location and analysis in debugging concurrent programs.

Having the PDN as a representation for concurrent programs, static and dynamic program slicing based on the PDN can provide us with a systematic method to support the bug location in debugging concurrent programs. Once we detected an error in an execution of a concurrent program that occurred at statement s, then we can find the static slice **SS**(s, V) of the program based on the PDN of the program. The static slice **SS**(s, V) covers all statements might cause the error occurred at statement s, i.e., all "possible candidates" of bugs. Since the PDN of a concurrent program covers all possible synchronization and communication dependences, a static slice of the program is perhaps still a large set of statements and include many statements that are actually irrelevant to the error being debugged. If we have an execution monitor that can collect execution history information of the program being debugged, then we can find the dynamic slice **DS**(s, V, H, I) of the program based on the PDN and the information that which statements are executed actually during the execution where the error occurs. The dynamic slice **DS**(s, V, H, I) covers all statements actually caused the error occurred at statement s in the execution, i.e., all "actual candidates" of bugs.

However, the static and dynamic slices of a program only cover those "candidates" of bugs but neither locate the bugs nor give some hints on the nature of the bugs. In order to develop more powerful debugging methods and tools to support bug localization, analysis, and correction in concurrent program debugging, we are constructing an entailmental logic calculus [4] for causal reasoning about bugs in concurrent programs and developing a knowledge-based approach to debugging concurrent programs. The PDN will serve in the approach as the basis for representing knowledge about causal relations in concurrent programs.

In concurrent program maintenance, it is necessary to know which statements in which processes would be affected by a statement modified in maintenance and/or which statements in which processes affect the modified statements. The needs can be satisfied by slicing or forward-slicing the program being maintained. The obtained slice will be very useful in maintenance of concurrent programs in the sense that they are helpful for us to know which statements are affected by the modified ones and which statements affect the modified ones.

Because the PDN and PIN of a concurrent program represents both control and data flow properties in every process and synchronization and communication properties in interprocess interaction in the program, it can also be used to define metrics for measuring complexity of concurrent programs. For example, we can consider the following dependence-based and slice-based metrics for measuring the complexity of a concurrent program: (1) how many statements and/or processes a statement or process in the program is directly dependent on, (2) how many statements and/or processes a statement or process in the program directly influences, (3) how long dependence path a process in the program or the program itself has, (4) how many statements the largest slice of the program has, (5) how many statements in a slice of the program are found only in that slice, (6) how many slices of the program includes some statements in common, and so on.

7. Concluding Remarks

We have proposed three new types of basic program dependences and two new program representations for concurrent programs. Although here we presented the program dependences and the representation in terms of Occam 2 programs, the concepts are general and easy to be applied to those concurrent programs written in other high-level concurrent programming languages such as Ada [5,6].

As concurrent program representations useful in understanding, testing, debugging, maintenance, and complexity measure/metrics of concurrent programs, the PDN and PIN may play an important

role in development of concurrent software. The significance of the representations in concurrent programming depends on how we develop the representations themselves and apply them to practices of concurrent programming. Having effective tools to transform a concurrent program into its PDN and PIN and to slice concurrent programs is crucial to the applications of the PDN and PIN in development of concurrent software. We are developing a group of tools including a tool to transform concurrent programs into their PDNs and PINs, a tool to slice concurrent programs based on their PDNs and PINs, a knowledge-base to store knowledge about program dependences and influences in concurrent programs, and a causal reasoning engine based on entailmental logic to reason about properties of bugs in concurrent programs. All of the tools will use the PDN and PIN as a unified representation for target concurrent programs.

There are some future research problems. For example, the five types of basic dependences presented in this paper do not cover all possible program dependences in a concurrent program. For example, a statement S_1 in a process may synchronization dependent on a statement S_2 in another process and S_2 may control dependent on a statement S_3. Obviously, there is some dependence between S_1 and S_3 that cannot be regarded as any of the five basic types of dependences. An important research problem is how to deal with all possible program dependences in a concurrent program. The size of the PDN and PIN of a practical concurrent program is crucial to the application of the net in practices. How to measure the size of a PDN or PIN and how to reduce their sizes on the condition that the dependence/influence information is preserved are also important research problems.

References

[1] H. Agrawal and J. R. Horgan, "Dynamic Program Slicing", Proc. ACM SIGPLAN'90, pp.246-256, 1990.

[2] A. V. Aho, R. Sethi, and J. D. Ullman, "Compilers: Principles, Techniques, and Tools", Addison-Wesley, 1986.

[3] K. Araki, Z. Furukawa, and J. Cheng, "A General Framework for Debugging", IEEE-CS Software, Vol.8, No.3, pp.14-20, 1991.

[4] J. Cheng, "Logical Tool of Knowledge Engineering: Using Entailment Logic rather than Mathematical Logic", Proc. ACM 19th Annual Computer Science Conference, pp.228-238, 1991.

[5] J. Cheng, "Task Dependece Net as a Representation for Concurrent Ada Programs", in J. van Katwijk (ed.), "Ada: Moving towards 2000" (Proceedings of the Ada-Europe Eleventh Annual International Conference), Lecture Notes in Computer Science, Vol.603, pp.150-164, Springer-Verlag, June 1992.

[6] J. Cheng, "The Tasking Dependence Net in Ada Software Development", ACM Ada Letters, Vol.12, July/August 1992.

[7] J. Cheng, K. Araki, and K. Ushijima, "Development and Practical Applications of EDEN – An Event-Driven Execution Monitor for Concurrent Ada Programs", Transactions of IPSJ, Vol.30, No.1, pp.12-24, 1989 (in Japanese).

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

[10]G. S. Goldszmidt, S. Temini, and S. Katz, "High-Level Language Debugging for Concurrent Programs", ACM TOCS, Vol.8, No.4, pp.311-336, 1990.

[11]S. Horwitz, T. Reps, and D. Binkley, "Interprocedural Slicing Using Dependence Graphs", ACM TOPLAS, Vol.12, No.1, pp.26-60, 1990.

[12]INMOS Limited, "occam 2 Reference Manual", 1988.

[13]B. Korel and J. Laski, "Dynamic Program Slicing", Information Processing Letters, Vol.29, No.10, pp.155-163, 1988.

[14]C. E. McDowell and D. P. Helmbold, "Debugging Concurrent Programs", ACM Computing Surveys, Vol.21, No.4, pp.593-622, 1989.

[15]G. J. Myers, "The Art of Software Testing", John Wiley & Sons, 1979.

[16]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.

[17]A. Podgurski and 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.

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

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