Slicing Concurrent Programs[*]
— A Graph-Theoretical Approach

Jingde Cheng

Department of Computer Science and Communication Engineering
Kyushu University
6-10-1 Hakozaki, Fukuoka 812, Japan
cheng@csce.kyushu-u.ac.jp

Abstract. This paper extends the notion of slicing, which was originally proposed and studied for sequential programs, to concurrent programs and presents a graph-theoretical approach to slicing concurrent programs. In addition to the usual control and data dependences proposed and studied for sequential programs, the paper introduces three new types of primary program dependences in concurrent programs, named the selection dependence, synchronization dependence, and communication dependence. The paper also propose a new program representation for concurrent programs, named the Process Dependence Net (PDN), which is an arc-classified digraph to explicitly represent the five types of primary program dependences in the programs. As a result, various notions about slicing concurrent programs can be formally defined based on the representation, and the problem of slicing a concurrent program can be simply reduced to a vertex reachability problem in its PDN representation.

1. Introduction

Debugging is an indispensable step in software development. 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. A bug may cause more than one error, and also, an error may be caused by more than one bug. Debugging a program is the process of locating, analyzing, and ultimately correcting bugs in the program by reasoning about causal relationships between bugs and the errors which have been detected in program. It begins with some

* This work is partly supported by The Ministry of Education, Science, and Culture of Japan under Grant-in-Aid for Scientific Research (C) No. 04650319 and a grant from The Kurata Foundation.

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, then corrects the bug(s), and ends in a verification of the removal of the error [2,19]. In general, about 95% of effort in debugging has to be spent on locating and understanding bug(s) because once a bug is located and its nature is understood, its correction is often easy to do [19]. Therefore, the most important problem in debugging is how to know which statements possibly and/or actually cause the erroneous behavior of a program.

Debugging a concurrent program is more difficult than debugging a sequential program because a concurrent program has multiple control flows, multiple data flows, and interprocess synchronization, communication, and nondeterministic selection. Almost all current debugging methods and tools for concurrent programs only provide programmers with facilities to extract some information from programs and display it in textual form or visual form, but no facilities to support the localization, analysis, and correction of bugs in an automatic or semi-automatic manner. Until now, there is no systematic method proposed and studied for bug location and analysis in debugging concurrent programs [8,11,18]. In order to satisfy the needs from the growing development and use of concurrent systems, it is urgent to establish a systematic and elegant debugging method for concurrent programs.

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. In the literature, there are two types of primary program dependences originally proposed and studied for optimization and parallelization of sequential programs, i.e., the control dependence and the data dependence [9,22]. 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.

Statically slicing a program on a criterion, originally introduced by Weiser, is a source-to-source transformation method to automatically reduce a program, by analyzing control and data dependences in the program, into a minimal form, which is called a static slice of the original program, such that the reduced program will produce the behavior satisfying the criterion [23,24]. The method is useful in program debugging in the sense that slicing a program can certainly narrow the domain on which reasoning about causal relationship between errors and bugs in the program is performed [13,15,22-24].

Based on Weiser's work, a number of program slicing techniques have been proposed and studied for sequential programs. Ottenstein and Ottenstein showed that once a program is represented by its program dependence graph (PDG for short), which is an arc-classified digraph to explicitly represent control and data dependences in the program [9], the problem of statically slicing the program can be simply reduced to a vertex reachability problem in its PDG representation [20]. Horwitz, Reps, and Binkley introduced a program representation, called the "system dependence graph," for programs with procedures and procedure calls and showed an algorithm for interprocedural static slicing of a program based on its system dependence graph representation [12]. Korel and Laski introduced the notion of a "dynamic slice" and proposed a method to compute a dynamic slice of a program by solving data flow equations over an execution trace of the program [17]. Agrawal and Horgan introduced a program execution trace representation, called the "dynamic dependence graph," and showed a method to compute a dynamic slice of a program based on its dynamic dependence graph representation [1]. Kamkar, Shahmehri, and Fritzson introduced a program execution trace representation, called the "dynamic dependence summary graph" for programs with procedures and procedure calls, and showed an algorithm for interprocedural dynamic slicing of a program based on its dynamic dependence summary graph representation [14].

However, until recently, there are few program slicing methods proposed and studied for concurrent programs. The present author 1991 introduced a program representation, called the "Process Dependence Net" for concurrent programs and discussed its various possible applications including slicing concurrent programs [3-5]. This is the first attempt to identify all primary program dependences in concurrent programs and represent them in a digraph for general applications. Duesterwald, Gupta, and Soffa introduced an execution trace representation, called the "distributed dependence graph" for distributed programs, and showed a parallel algorithm to compute dynamic slices of a distributed program based on its distributed dependence graph representation [10]. Korel and Ferguson also proposed a method to compute dynamic slices of a distributed Ada program by analyzing influences in and between multiple executed paths of the program [16].

This paper can be regarded as a further refinement and improvement to the author's some early work. The paper extends the notion of slicing to concurrent programs and presents a graph-theoretical approach to slicing concurrent programs. In addition to the usual control and data dependences proposed and studied for sequential programs, this paper introduces three new types of primary program dependences in concurrent programs, named the "selection dependence," "synchronization dependence," and "communication dependence," and a new program representation for concurrent programs, named the "Process Dependence Net" (PDN for short), which is an arc-classified digraph to explicitly represent the five types of primary program dependences in the programs. As a result, various notions about slicing concurrent programs can be formally defined based on the representation, and the problem of slicing a concurrent program can be simply reduced to a vertex reachability problem in its PDN representation.

The rest of this paper is organized as follows: Section 2 give some informal definitions about slicing concurrent programs, Section 3 introduces two program representations, named the “nondeterministic parallel control-flow net” and “nondeterministic parallel definition-use net,” for representing multiple control flows and multiple data flows in concurrent programs, Section 4 defines the five types of primary program dependences in a concurrent program formally based on its nondeterministic parallel control-flow and/or definition-use nets, Section 5 presents the PDN, defines those notions about slicing concurrent programs formally based on the PDN, and shows applications of the PDN to slicing concurrent programs.

2. Various Slices of Concurrent Programs

A major observation and/or belief underlying the author’s work on automated debugging is that if we have sufficient information about a program, its specification, and an execution of it where an error is detected, then, in principle, we should be able to debug the program by analyzing the error and reasoning about bug(s) caused the error based on the information and our knowledge and experiences without totally or partially executing the program time after time. The author considers that there are three key issues in automated debugging : (1) how to acquire, represent, and organize general knowledge about debugging in a general-purpose debugging system, (2) how to identify and capture sufficient information for a debugging problem, and (3) how to effectively and efficiently use the knowledge and information by automated reasoning to find bugs or plausible candidates of bugs.

Based on the above consideration, this paper focuses its attention on whether or not a slice of a concurrent program is adequate to a debugging problem rather than whether or not a slice of a concurrent program is executable.

We give some informal definitions about slicing concurrent programs as follows.

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 at s. ***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 notion of program slice given above for concurrent programs and that given in the literature 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 notion useful in analysis of synchronization errors in concurrent programs such as deadlocks and livelocks.

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 at s 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 [6]. This is the reason why we define the notion of dynamic slicing criterion using a specified execution history H of a concurrent program with an input I.

In debugging a concurrent program, once we found a bug or a plausible candidate of bug, in order to avoid that some new bugs are introduced in debugging the old bug, we often want to know that correcting the bug or candidate will affect which piece of the program, i.e., we want to know which statements in which processes would be affected by a statement if we modify it in debugging. The needs leads us to define some notions about forward-slicing concurrent programs as follows.

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 set of variables 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 values of variables in 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.

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 set of variables 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 values of variables in 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.

3. Nondeterministic Parallel Control-Flow and Definition-Use Nets

We now introduce the nondeterministic parallel control-flow net and nondeterministic parallel definition-use net for representing multiple control flows and multiple data flows in concurrent programs.

Definition 3.1 A ***digraph*** is an ordered pair ($\mathbf{V}$, $\mathbf{A}$), where $\mathbf{V}$ is a finite set of elements, called ***vertices***, and $\mathbf{A}$ is a finite set of elements of the Cartesian product $\mathbf{V}\times\mathbf{V}$, called ***arcs***, i.e., $\mathbf{A}\subseteq\mathbf{V}\times\mathbf{V}$ is a binary relation on $\mathbf{V}$. For any arc $(v_1,v_2)\in\mathbf{A}$, 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 ($\mathbf{V}$, $\mathbf{A}$) such that $(v,v)\notin\mathbf{A}$ for any $v\in\mathbf{V}$.

Definition 3.2 An ***arc-classified digraph*** is an n-tuple ($\mathbf{V}$, $\mathbf{A}_1$, $\mathbf{A}_2$, ..., $\mathbf{A}_{n-1}$) such that every ($\mathbf{V}$, $\mathbf{A}_i$) (i = 1, ..., n−1) is a digraph and $\mathbf{A}_i\cap\mathbf{A}_j = \Phi$ for i = 1, 2, ..., n−1 and j = 1, 2, ..., n−1. A ***simple arc-classified digraph*** is an arc-classified digraph ($\mathbf{V}$, $\mathbf{A}_1$, $\mathbf{A}_2$, ..., $\mathbf{A}_{n-1}$) such that $(v,v)\notin\mathbf{A}_i$ (i = 1, ..., n−1) for any $v\in\mathbf{V}$.

Definition 3.3 A ***path*** in a digraph ($\mathbf{V}$, $\mathbf{A}$) or an arc-classified digraph ($\mathbf{V}$, $\mathbf{A}_1$, $\mathbf{A}_2$, ..., $\mathbf{A}_{n-1}$) is a sequence of arcs (a_1, a_2, ..., a_L) such that the terminal vertex of a_i is the initial vertex of a_{i+1} for $1 \le i \le L-1$, where $a_i\in\mathbf{A}$ or $a_i\in\mathbf{A}_1\cup\mathbf{A}_2\cup...\cup\mathbf{A}_{n-1}$, and L ($L \ge 1$) 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.

Definition 3.4 A ***nondeterministic parallel control-flow net*** (CFN for short) is a 10-tuple ($\mathbf{V}$, $\mathbf{N}$, $\mathbf{P}_F$, $\mathbf{P}_J$, $\mathbf{A}_C$, $\mathbf{A}_N$, $\mathbf{A}_{PF}$, $\mathbf{A}_{PJ}$, $\mathbf{s}$, $\mathbf{t}$), where ($\mathbf{V}$, $\mathbf{A}_C$, $\mathbf{A}_N$, $\mathbf{A}_{PF}$, $\mathbf{A}_{PJ}$) is a simple arc-classified digraph such that $\mathbf{A}_C\subseteq\mathbf{V}\times\mathbf{V}$, $\mathbf{A}_N\subseteq\mathbf{N}\times\mathbf{V}$, $\mathbf{A}_{PF}\subseteq\mathbf{P}_F\times\mathbf{V}$, $\mathbf{A}_{PJ}\subseteq\mathbf{V}\times\mathbf{P}_J$, $\mathbf{N}\subset\mathbf{V}$ is a set of elements, called ***nondeterministic selection vertices***, $\mathbf{P_F} \subset \mathbf{V}$ ($\mathbf{N} \cap \mathbf{P_F} = \Phi$) is a set of elements, called ***parallel execution fork vertices***, $\mathbf{P_J} \subset \mathbf{V}$ ($\mathbf{N} \cap \mathbf{P_J} = \Phi$, and $\mathbf{P_F} \cap \mathbf{P_J} = \Phi$) is a set of elements, called ***parallel execution join vertices***, $\mathbf{s} \in \mathbf{V}$ is a unique vertex, called the ***start vertex***, such that in-degree(s) = 0, $\mathbf{t} \in \mathbf{V}$ is a unique vertex, called the ***termination vertex***, such that out-degree($\mathbf{t}$) = 0 and $\mathbf{t} \neq$ s, and for any $v \in \mathbf{V}$ ($v \neq \mathbf{s}$, $v \neq \mathbf{t}$), there exists at least one path from s to v and at least one path from v to t. Any arc $(v_1,v_2) \in \mathbf{A_C}$ is called a ***sequential control arc***, any arc $(v_1,v_2) \in \mathbf{A_N}$ is called a ***nondeterministic selection arc***, and any arc $(v_1,v_2) \in \mathbf{A_{PF}} \cup \mathbf{A_{PJ}}$ is called a ***parallel execution arc***.

A usual (deterministic and sequential) control flow graph can be regarded as a special case of CFN where $\mathbf{N}$, $\mathbf{P_F}$, $\mathbf{P_J}$, $\mathbf{A_N}$, $\mathbf{A_{PF}}$, and $\mathbf{A_{PJ}}$ are the empty set, respectively. A CFN can be used to represent the single control flow and/or multiple control flows in a sequential and/or concurrent program written in an imperative programming language such as C, Pascal, Ada, and Occam 2.

Definition 3.5 Let u and v be any two vertices in a CFN. u ***forward dominates*** v iff every path from v to $\mathbf{t}$ 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 to $\mathbf{t}$ whose length is greater than or equal to k contains u; u is called the **immediate forward dominator** of v iff u is the first vertex that properly forward dominates v in every path from v to $\mathbf{t}$; u is called the **last continuous forward dominator** of v iff u is the vertex such that any vertex in the path from v to u forward dominates v but a successor of u does not forward dominate v.

Definition 3.6 A ***nondeterministic parallel definition-use net*** (DUN for short) is a 7-tuple ($\mathbf{N_C}$, Σ_V, $\mathbf{D}$, $\mathbf{U}$, Σ_C, $\mathbf{S}$, $\mathbf{R}$), where $\mathbf{N_C}$ = ($\mathbf{V}$, $\mathbf{N}$, $\mathbf{P_F}$, $\mathbf{P_J}$, $\mathbf{A_C}$, $\mathbf{A_N}$, $\mathbf{A_{PF}}$, $\mathbf{A_{PJ}}$, $\mathbf{s}$, $\mathbf{t}$) is a CFN, Σ_V is a finite set of symbols, called ***variables***, $\mathbf{D}$: $V \rightarrow P(\Sigma_V)$ and $\mathbf{U}$: $V \rightarrow P(\Sigma_V)$ are two partial functions from $\mathbf{V}$ to the power set of Σ_V, Σ_C is a finite set of symbols, called ***channels***, and $\mathbf{S}$: $V \rightarrow P(\Sigma_C)$ and $\mathbf{R}$: $V \rightarrow P(\Sigma_C)$ are two partial functions from $\mathbf{V}$ to the power set of Σ_C.

A usual (deterministic and sequential) definition-use graph can be regarded as a special case of DUN where $\mathbf{N}$, $\mathbf{P_F}$, $\mathbf{P_J}$, $\mathbf{A_N}$, $\mathbf{A_{PF}}$, $\mathbf{A_{PJ}}$, Σ_C, $\mathbf{S}$, and $\mathbf{R}$ are the empty set, respectively. A DUN can be regarded as a CFN with the information concerning definitions and uses of variables and communication channels.

Note that the above definitions of CFN and DUN are graph-theoretical, and therefore, they are independent of any programming language.

We are developing a general-purpose system working on UNIX to compile target programs written in various imperative programming languages (at present, including C, Pascal, Ada, and Occam 2) into their CFN and DUN representations in textual forms [7]. Because of the limitation, here we do not discuss how to transform a program into its CFN and DUN representations.

As an example, Fig. 1 shows a fragment of Occam 2 program and Fig. 2 shows an arc-classified digraph representation of the DUN of the program.

```
PAR
  SEQ
    input ? x; y
    IF
      (x<0) OR (y<0)
        error1 ! 14 :: “minus operator”
        ce ! 0.0
      y = 0
        error ! 11 :: “zero divide”
        ce ! 0.0
      TRUE
        c ! x/y
  SEQ
    input2 ? n
    sum := 0
    WHILE n<>0
      input2 ? data
      sum, n := sum + data, n–1
    ALT
      c ? factor
        result ! sum*factor
      ce ? factor
        error2 ! 14 :: “invalid factor”
STOP
```

Fig. 1 A fragment of Occam 2 program

4. Primary Program Dependences in 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 among multiple control flows and interprocess communication among multiple data flows in the program. Moreover, a process in a concurrent program may 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 proposed and studied for sequential programs is inadequate for representing the complete behavior of a concurrent program. In order to slice concurrent programs by analyzing various dependences in the programs and construct slices of a concurrent program that includes sufficient information for debugging the program, we must identify all primary program dependences in concurrent programs at first.

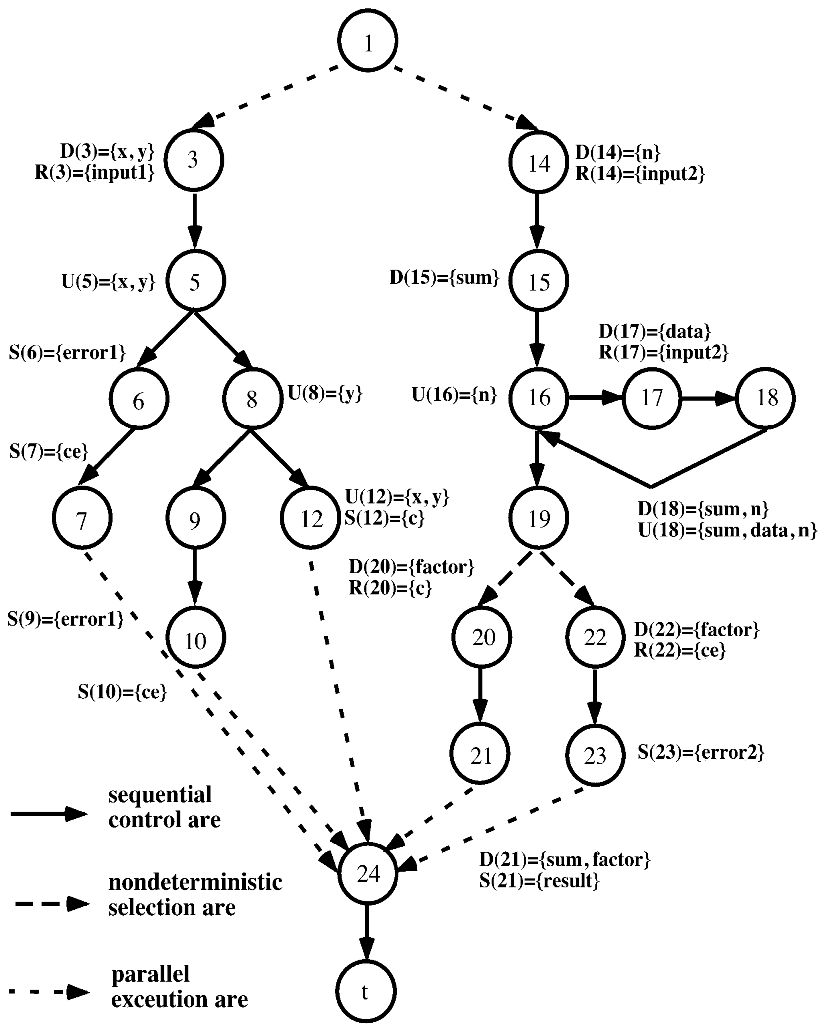

Fig. 2 An arc-classified digraph representation of the DUN of the program of Fig. 1

In addition to the usual control and data dependences, we now introduce three new types of primary program dependences in concurrent programs, named the ***selection dependence***, ***synchronization dependence***, and ***communication dependence***, which are determined by interactions between multiple control flows and multiple data flows in the programs. Based on the CFN and/or DUN of a concurrent program, we can define various types of primary program dependences in the program as follows.

Definition 4.1 Let ($\mathbf{V}$, $\mathbf{N}$, $\mathbf{P_F}$, $\mathbf{P_J}$, $\mathbf{A_C}$, $\mathbf{A_N}$, $\mathbf{A_{PF}}$, $\mathbf{A_{PJ}}$, $\mathbf{s}$, $\mathbf{t}$) be the CFN of a concurrent program, and $u \in \mathbf{V}$, $v \in ((\mathbf{V}-(\mathbf{N} \cup \mathbf{P_F} \cup \mathbf{P_J}))$ be any two vertices of the net. u is ***directly strongly control-dependent*** on v iff there exists a path P = $(v_1=v,v_2)$, (v_2,v_3), ..., $(v_{n-1},v_n=u)$ from v to u such that P does not contain the immediate forward dominator of v and there exists no vertex v' in P such that the path from v' to u 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 there exists a path P = $(v_1=v,v_2)$, (v_2,v_3), ..., $(v_{n-1},v_n=u)$ from v to u and any vertex v_i $(1 < i \leq n)$ in P strongly forward dominates v' but does not strongly forward dominate v".

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. For example, in Fig. 2, 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 ($\mathbf{V}$, $\mathbf{N}$, $\mathbf{P_F}$, $\mathbf{P_J}$, $\mathbf{A_C}$, $\mathbf{A_N}$, $\mathbf{A_{PF}}$, $\mathbf{A_{PJ}}$, $\mathbf{s}$, $\mathbf{t}$) be the CFN of a concurrent program, and $u \in \mathbf{V}$, $v \in \mathbf{N}$ be any two vertices of the net. u is ***directly selection-dependent*** on v iff (1) there exists a path P = $(v_1=v,v_2)$, (v_2,v_3), ..., $(v_{n-1},v_n=u)$ from v to u such that P does not contain the immediate forward dominator of v, and (2) there exists no vertex v_i $(1 < i < n)$ in P such that the path from vi to u does not contain the immediate forward dominator of v_i.

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 another branch is executed then u may not be executed. For example, in Fig. 2, vertices v_{20} ~ v_{23} are directly selection-dependent on vertex v_{19}.

The difference between the direct (strong or weak) control dependence and the direct selection dependence is that the former defines a kind of program dependence holding between the control predicate of a conditional branch statement and a statement whether it is executed is determined by the truth value of the control predicate, but the latter defines a kind of program dependence holding between a nondeterministic selection statement and a statement whether it is executed is determined by the nondeterministic selection.

Definition 4.3 Let ($\mathbf{N_C}$, $\mathbf{\Sigma_V}$, $\mathbf{D}$, $\mathbf{U}$, $\mathbf{\Sigma_C}$, $\mathbf{S}$, $\mathbf{R}$) be the DUN of a concurrent 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 $(\mathbf{D}(v)\cap\mathbf{U}(u))-\mathbf{D}(P') \neq \Phi$ where $\mathbf{D}(P') = \mathbf{D}(v_2)\cup \ldots \cup\mathbf{D}(v_{n-1})$.

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. 2, 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 [21]. Those algorithms can be modified to compute the control, selection, and data dependences in a concurrent program based on the DUN of the program [7].

Definition 4.4 Let ($\mathbf{N_C}$, $\mathbf{\Sigma_V}$, $\mathbf{D}$, $\mathbf{U}$, $\mathbf{\Sigma_C}$, $\mathbf{S}$, $\mathbf{R}$) be the DUN of a concurrent program, where $\mathbf{N_C}$ is the CFN ($\mathbf{V}$, $\mathbf{N}$, $\mathbf{P_F}$, $\mathbf{P_J}$, $\mathbf{A_C}$, $\mathbf{A_N}$, $\mathbf{A_{PF}}$, $\mathbf{A_{PJ}}$, $\mathbf{s}$, $\mathbf{t}$) 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 holds :

1) $(v,u)\in \mathbf{A_{PF}}\cup\mathbf{A_{PJ}}$, i.e., (v,u) is a parallel execution arc,

2) $\mathbf{S}(v) = \mathbf{R}(u)$, or

3) there exists a vertex v' such that v' is directly synchronization-dependent on v, u is the last continuous forward dominator of v', and $\mathbf{S}(v'') = \Phi$ and $\mathbf{R}(v'') = \Phi$ for any vertex v'' (excluding v') in the path from v' to u.

Informally, if u is directly synchronization-dependent on v, then the start and/or termination of execution of v directly determines whether or not the execution of u starts and/or terminates. For example, in Fig. 2, vertices v_3, v_5, v_{14}, v_{15}, and v_{16} are directly synchronization-dependent on vertex v_1, vertex v_{20} is directly synchronization-dependent on vertex v_{12}, vertex v_{22} is 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}.

The difference between the direct (strong or weak) control dependence and the direct synchronization dependence is that the former is irrelevant to the execution timing of a program but the latter is intrinsically relevant to the execution timing.

Definition 4.5 Let ($\mathbf{N}_C$, $\mathbf{\Sigma}_V$, **D**, **U**, $\mathbf{\Sigma}_C$, **S**, **R**) be the DUN of a concurrent program, and u and v be any two vertices of the net. u is ***directly communication-dependent*** on v iff 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.

Informally, if u is directly communication-dependent on v, then the value of a variable computed at v directly has influence on the value of a variable computed at u by an interprocess communication. For example, in Fig. 2, vertex v_{21} is directly communication-dependent on vertex v_3.

The difference between the direct data dependence and the direct communication dependence is that the direct data dependence is irrelevant to communication channels of a program but the direct communication dependence is intrinsically relevant to the channels.

5. Process Dependence Net and Concurrent Program Slicing

We now introduce the Process Dependence Net for representing the five types of primary program dependences in concurrent programs.

Definition 5.1 The ***Process Dependence Net*** (PDN for short) of a concurrent program is an arc-classified digraph (**V**, **Con**, **Sel**, **Dat**, **Syn**, **Com**), where **V** is the vertex set of the CFN 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.

A usual program dependence graph for sequential programs can be regarded as a special case of PDN. As an example, Fig. 3 shows an arc-classified digraph representation of the PDN of the Occam 2 program shown in Fig. 1.

Since program dependences in a program are implied by control and data flows in the program, an explicit representation of these program dependences is available only if we have some methods and tools to compute them by analyzing control and data flows in the program and then represent them in some form for general use. We are developing a general-purpose system working on UNIX to compile target programs written in various programming languages (at present, including C, Pascal, Ada, and Occam 2) into their PDN representations in textual forms [7].

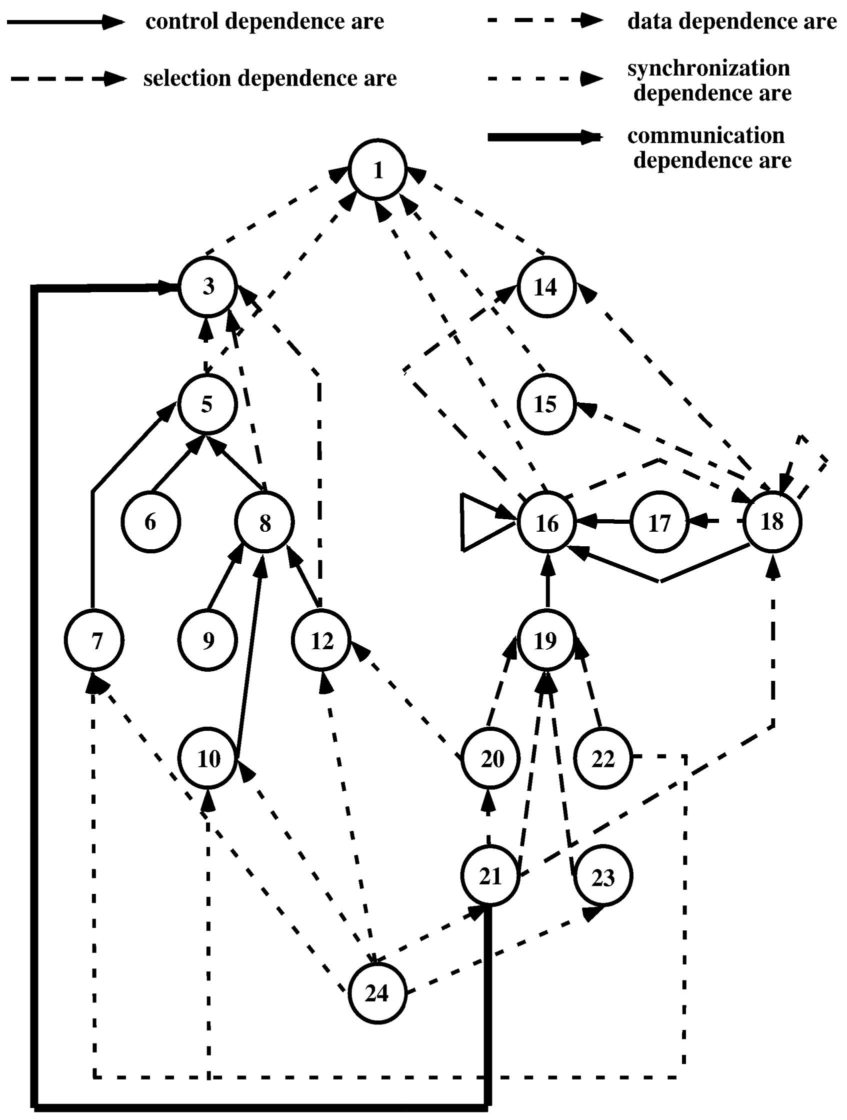

Fig. 3 An arc-classified digraph representation of the PDN of program of Fig. 1

We can also consider a representation of a concurrent program, named the “Process Influence Net,” as a “reverse” of the PDN of that program.

Definition 5.2 The ***Process Influence Net*** (PIN for short) of a concurrent program is an arc-classified digraph (**V**, **Con**, **Sel**, **Dat**, **Syn**, **Com**), where **V** is the vertex set of the CFN 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.

In Section 2, we have informally defined some notions about slicing concurrent programs. Now, having the PDN and PIN as dependence-based representations of concurrent programs, we can formally refine those notions based on the PDN and PIN.

Definition 5.3 Let P be a concurrent program, ($\mathbf{V}$, $\mathbf{N}$, $\mathbf{P_F}$, $\mathbf{P_J}$, $\mathbf{A_C}$, $\mathbf{A_N}$, $\mathbf{A_{PF}}$, $\mathbf{A_{PJ}}$, **s**, **t**) be the CFN of P, ($\mathbf{N_C}$, $\mathbf{\Sigma_V}$, **D**, **U**, $\mathbf{\Sigma_C}$, **S**, **R**) be the DUN of P, and (**V**, **Con**, **Sel**, **Dat**, **Syn**, **Com**) be the PDN of P. A ***static slicing criterion*** of P is a 2-tuple (s,V) such that s∈**V** and V = **U**(s). The ***static slice*** SS(s,V) of P on a given static slicing criterion (s,V) is a subset of vertices of **V**, SS(s,V)⊆**V**, such that for any v∈**V**, v∈SS(s,V) if and only if there exists a path from s to v in the PDN.

Obviously, based on the above definition, statically slicing a concurrent program on a given static slicing criterion, which is defined as to find the static slice of the program with respect to the criterion, is simply a vertex reachability problem in the PDN of the program. Therefore, having the PDN as a dependence-based representation of concurrent programs, the problem of statically slicing a concurrent program can be easily solved using a usual depth-first or breadth-first graph traversal algorithm to traverse the PDN representation of the program by taking the statement of interest as the start point of traversal.

For a concurrent program P and a given static slicing criterion (s,V), the static slice SS(s,V) of P consists of all statements in P that possibly affect the beginning or end of execution of s and/or affect the values of variables in V. Therefore, the information included in SS(s,V) should be sufficient to a debugging problem concerning s and variables used at s. However, there may be, of course, many statements in SS(s,V) that are completely irrelevant to the error being debugged. For example, a synchronization error detected at s may be independent of those statements included in SS(s,V) as the result of analyzing data dependence. In debugging a concurrent program, having a "partial and/or special" slice of the program rather than a "total and/or general" slice is often more convenient for us to effectively and efficiently debug the program. The above discussion leads us to define some kinds of special static slices of a concurrent program as follows.

Definition 5.4 Let (**V**, **Con**, **Sel**, **Dat**, **Syn**, **Com**) be a PDN. A path is called a ***control-dependent path*** if (u,v)∈**Con** for any arc (u,v) in the path; a path is called a ***selection-dependent path*** if (u,v)∈**Sel** for any arc (u,v) in the path; a path is called a ***data-dependent path*** if (u,v)∈**Dat** for any arc (u,v) in the path; a path is called a ***synchronization-dependent path*** if (u,v)∈**Syn** for any arc (u,v) in the path; a path is called a ***communication-dependent path*** if (u,v)∈**Com** for any arc (u,v) in the path.

Definition 5.5 Let P be a concurrent program and (**V**, **Con**, **Sel**, **Dat**, **Syn**, **Com**) be the PDN of P. The ***control-dependent static slice*** **Con-SS**(s,V) of P on a given static slicing criterion (s,V) is a subset of vertices of **V**, **Con-SS**(s,V)⊆**V**, such that for any v∈**V**, v∈**Con-SS**(s,V) if and only if there exists a control-dependent path from s to v in the PDN; the ***selection-dependent static slice*** **Sel-SS**(s,V) of P on a given static slicing criterion (s,V) is a subset of vertices of **V**, **Sel-SS**(s,V)⊆**V**, such that for any v∈**V**, v∈**Sel-SS**(s,V) if and only if there exists a selection-dependent path from s to v in the PDN; the ***data-dependent static slice*** **Dat-SS**(s,V) of P on a given static slicing criterion (s,V) is a subset of vertices of **V**, **Dat-SS**(s,V)⊆**V**, such that for any v∈**V**, v∈**Dat-SS**(s,V) if and only if there exists a data-dependent path from s to v in the PDN; the ***synchronization-dependent static slice*** **Syn-SS**(s,V) of P on a given static slicing criterion (s,V) is a subset of vertices of **V**, **Syn-SS**(s,V)⊆**V**, such that for any v∈**V**, v∈**Syn-SS**(s,V) if and only if there exists a synchronization-dependent path from s to v in the PDN; the ***communication-dependent static slice*** **Com-SS**(s,V) of P on a given static slicing criterion (s,V) is a subset of vertices of **V**, **Com-SS**(s,V) ⊆**V**, such that for any v∈**V**, v∈**Com-SS**(s,V) if and only if there exists a communication-dependent path from s to v in the PDN.

Obviously, for any concurrent program and any given static slicing criterion (s,V), the following holds: SS(s,V) = **Con-SS**(s,V)∪**Sel-SS**(s,V)∪**Dat-SS**(s,V)∪**Syn-SS**(s,V)∪**Com-SS**(s,V).

Moreover, some other kinds of special static slices also can be considered. For example, a static slice that is a combination of some kinds of the above special static slices, a static slice that only concerns a subset of variables used at a statement of interest rather than all variables used at the statement, a static slice that only concerns a subset of processes in a program rather than all process in the program, and so on. More detailed discussion on these topics is beyond the scope of this paper.

A dynamic slice of a concurrent program now can be formally defined as a subset of the corresponding static slice of the program as follows.

Definition 5.6 Let P be a concurrent program, (**V**, **N**, $\mathbf{P_F}$, $\mathbf{P_J}$, $\mathbf{A_C}$, $\mathbf{A_N}$, $\mathbf{A_{PF}}$, $\mathbf{A_{PJ}}$, **s**, **t**) be the CFN of P, ($\mathbf{N_C}$, $\mathbf{\Sigma_V}$, **D**, **U**, $\mathbf{\Sigma_C}$, **S**, **R**) be the DUN of P, and (**V**, **Con**, **Sel**, **Dat**, **Syn**, **Com**) be the PDN of P. A ***dynamic slicing criterion*** of P is a quadruplet (s,V,H,I) such that s∈**V**, V = **U**(s), and H⊆**V** is a subset of **V** which includes all actually executed statements in an execution of P with input I. The ***dynamic slice*** DS(s,V,H,I) of P on a given dynamic slicing criterion (s,V,H,I) is a subset of static slice **SS**(s,V), **DS**(s,V,H,I)⊆**SS**(s,V), such that for any v∈**SS**(s,V), v∈**DS**(s,V,H,I) if and only if there exists a path from s to v in the PDN and u∈H for any vertex u in the path.

It is trivial to compute a dynamic slice of a concurrent program based on the corresponding static slice of the program and the program's execution history information that can be collected by an execution monitor.

We can also define various special dynamic slices of a concurrent program similar to those special static slices of the program.

Those notions about statically and dynamically forward-slicing concurrent programs can be formally refined based on the PIN as follows.

Definition 5.7 Let P be a concurrent program, (**V**, **N**, $\mathbf{P_F}$, $\mathbf{P_J}$, $\mathbf{A_C}$, $\mathbf{A_N}$, $\mathbf{A_{PF}}$, $\mathbf{A_{PJ}}$, **s**, **t**) be the CFN of P, ($\mathbf{N_C}$, $\mathbf{\Sigma_V}$, **D**, **U**, $\mathbf{\Sigma_C}$, **S**, **R**) be the DUN of P, and (**V**, **Con**, **Sel**, **Dat**, **Syn**, **Com**) be the PIN of P. A ***static forward-slicing criterion*** of P is a 2-tuple (s,V) such that s∈**V** and V = **D**(s). The ***static forward-slice*** **SFS**(s,V) of P on a given static forward-slicing criterion (s,V) is a subset of vertices of **V**, **SFS**(s,V)⊆**V**, such that for any v∈**V**, v∈**SFS**(s,V) if and only if there exists a path from s to v in the PIN.

Definition 5.8 Let P be a concurrent program, (**V**, **N**, $\mathbf{P_F}$, $\mathbf{P_J}$, $\mathbf{A_C}$, $\mathbf{A_N}$, $\mathbf{A_{PF}}$, $\mathbf{A_{PJ}}$, **s**, **t**) be the CFN of P, ($\mathbf{N_C}$, $\mathbf{\Sigma_V}$, **D**, **U**, $\mathbf{\Sigma_C}$, **S**, **R**) be the DUN of P, and (**V**, **Con**, **Sel**, **Dat**, **Syn**, **Com**) be the PIN of P. A ***dynamic forward-slicing criterion*** of P is a quadruplet (s,V,H,I) such that s∈**V**, V = **D**(s), and H⊆**V** is a subset of **V** which includes all actually executed statements in an execution of P with input I. The ***dynamic forward-slice*** **DFS**(s,V,H,I) of P on a given dynamic forward-slicing criterion (s,V,H,I) is a subset of static forward-slice **SFS**(s,V), **DFS**(s,V,H,I)⊆**SFS**(s,V), such that for any v∈**SFS**(s,V), v∈**DFS**(s,V,H,I) if and only if there exists a path from s to v in the PIN and u∈H for any vertex u in the path.

It is obvious that once a concurrent program is represented by its PIN, the problem of statically forward-slicing the program is simply a vertex reachability problem in the net, which can be easily solved using a usual depth-first or breadth-first graph traversal algorithm to traverse the net by taking the statement of interest as the start point of traversal, and the problem of dynamically forward-slicing the program can be reduced to the vertex reachability problem in the net with the program's execution history information.

6. Concluding Remarks

We have introduced three new types of primary program dependences in concurrent programs and some new program representations for concurrent programs. We also defined some general notions about slicing concurrent programs and showed that once a concurrent program is represented by its dependence-based representations, the problem of slicing the program is simply a vertex reachability problem in the representations which can be easily solved using a usual graph traversal algorithm to traverse the representations by taking the statement of interest as the start point of traversal.

Static and dynamic slicing are useful in debugging concurrent programs because they can be used to find all statements that possibly or actually caused the erroneous behavior of an execution of a concurrent program where an error occurs. A static slice **SS**(s,V) covers all statements might cause the error occurred at statement s, i.e., all "possible candidates" of bugs. A 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. Therefore, we can say that slicing a concurrent program can certainly narrow the domain on which reasoning about causal relationship between errors and bugs in the program is performed.

However, static and dynamic slices of a concurrent 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 provide programmers with some automatic or semi-automatic manner for bug localization, analysis, and correction in concurrent program debugging, it is necessary to develop more powerful debugging methods and tools.

Acknowledgements

I am grateful to the anonymous referees and Prof. Peter Fritzson for their helpful comments on early drafts of this paper, to Dr. Mariam Kamkar for sending me the paper by Duesterwald, Gupta, and Soffa before this workshop, and to Prof. Bogdan Korel for showing me the paper by himself and Ferguson at the workshop.

References

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

[2] K. Araki, Z. Furukawa, and J. Cheng, “A General Framework for Debugging,” IEEE-CS Software, Vol.8, No.3, pp.14-20, 1991.
[3] J. Cheng, “Process Dependence Net: A Concurrent Program Representation,” Proc. JSSST 8th Conference, pp.513-516, September 1991.
[4] J. Cheng, “Task Dependence Net as a Representation for Concurrent Ada Programs,” in J. van Katwijk (ed.), “Ada: Moving towards 2000,” LNCS, Vol.603, pp.150-164, Springer-Verlag, June 1992.
[5] J. Cheng, “The Tasking Dependence Net in Ada Software Development,” ACM Ada Letters, Vol.12, No.4, pp.24-35, 1992.
[6] J. Cheng and K. Ushijima, “Partial Order Transparency as a Tool to Reduce Interference in Monitoring Concurrent Systems,” in Y. Ohno (ed.), “Distributed Environments,” pp.156-171, Springer-Verlag, 1991.
[7] J. Cheng, Y. Kasahara, M. Kamachi, Y. Nomura, and K. Ushijima, “Compiling Programs to Their Dependence-Based Representations,” Proc. 1993 IEEE Region 10 International Conference on Computer, Communication and Automation, to appear, October 1993.
[8] W. H. Cheung, J. P. Black, and E. Manning, “A Framework for Distributed Debugging,” IEEE-CS Software, Vol.7, No.1, pp.106-115, 1990.
[9] J. Ferrante, K. J. Ottenstein, and J. D. Warren, “The Program Dependence Graph and Its Use in Optimization,” ACM Transactions on Programming Languages and Systems, Vol.9, No.3, pp.319-349, 1987.
[10] E. Duesterwald, R. Gupta, and M. L. Soffa, “Distributed Slicing and Partial Re-execution for Distributed Programs,” Proc. 5th Workshop on Languages and Compilers for Parallel Computing, pp.329-337, August 1992.
[11] G. S. Goldszmidt, S. Temini, and S. Katz, “High-Level Language Debugging for Concurrent Programs,” ACM Transactions on Computer Systems, Vol.8, No.4, pp.311-336, 1990.
[12] S. Horwitz, T. Reps, and D. Binkley, “Interprocedural Slicing Using Dependence Graphs,” ACM Transactions on Programming Languages and Systems, Vol.12, No.1, pp.26-60, 1990.
[13] M. Kamkar, N. Shahmehri, and P. Fritzson, “Bug Localization by Algorithmic Debugging and Program Slicing,” Proc. PLILP ‘90, LNCS, Vol.456, pp.60-74, Springer-Verlag, August 1990.
[14] M. Kamkar, N. Shahmehri, and P. Fritzson, “Interprocedural Dynamic Slicing,” Proc. PLILP ‘92, LNCS, Vol.631, pp.370-371, Springer-Verlag, August 1992.
[15] B. Korel, “PELAS - Program Error-Locating Assistant System,” IEEE-CS Transactions on Software Engineering, Vol.14, No.9, pp.1253-1260, 1988.
[16] B. Korel and R. Ferguson, “Dynamic Slicing of Distributed Programs,” Appl. Math. and Comp. Sci., Vol.2, No.2, pp.199-215, 1992.
[17] B. Korel and J. Laski, “Dynamic Program Slicing,” Information Processing Letters, Vol.29, No.10, pp.155-163, 1988.
[18] C. E. McDowell and D. P. Helmbold, “Debugging Concurrent Programs,” ACM Computing Surveys, Vol.21, No.4, pp.593-622, 1989.
[19] G. J. Myers, “The Art of Software Testing,” John Wiley & Sons, 1979.
[20] 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.
[21] K. J. Ottenstein and S. Ellcey, “Experience Compiling Fortran to Program Dependence Graphs,” Software - Practice and Experience, Vol.22, No.1, pp.41-62, 1992.
[22] A. Podgurski and L. A. Clarke, “A Formal Model of Program Dependences and Its Implications for Software Testing, Debugging, and Maintenance,” IEEE-CS Transactions on Software Engineering, Vol.16, No.9, pp.965-979, 1990.
[23] M. Weiser, “Programmers Use Slices When Debugging,” Communications of ACM, Vol.25, No.7, pp.446-452, 1982.
[24] M. Weiser, “Program Slicing,” IEEE-CS Transactions on Software Engineering, Vol.SE-10, No.4, pp.352-357, 1984.