An Approach to Slicing Concurrent Ada Programs Based on Program Reachability Graphs

Xiaofang Qi, and Baowen Xu

Southeast University, Department of Computer Science & Engineering, 210096 Nanjing, China

Summary
Program slicing is an important technique applied in many software engineering activities, such as program debugging, testing, maintenance, measurement, reengineering and etc. This paper presents an effective representation for concurrent Ada programs, which is called task communication reachability graph (TCRG). Based on TCRG, we can precisely determine various dependences in concurrent Ada programs and construct a new dependence graph (MSPDG) which vertex is a pair composed of program state and statement. Dependence relation in MSPDG is precise and transitive. By traversing MSPDG, we can obtain high-precision slice for concurrent Ada programs. Compared with other high-precision slicing methods, the slice based on MSDG is more precise and its efficiency is higher in worst case.

Key words:
dependence analysis, slice, Ada, concurrent programs, reachability analysis

Introduction

The concept of program slice was originally introduced by Weiser [1]. A program slice consists of those statements of a program that potentially affect the values computed at some point of interest [2]. The task of finding program slice is called program slicing. In the past twenty years, program slicing has gradually become a well-known program analysis technique and has been widely applied in many engineering activities, such as program understanding, debugging, testing, maintenance, measurement, reengineering and etc [1-15]. So far, methods for slicing sequential programs are established [2,3], but due to nondeterministic behaviors of concurrent programs, there exist many difficulties in slicing concurrent programs, including how to represent executions of concurrent programs effectively, solve intransitivity problem of dependence relation between statements and obtain more precise slice efficiently [4-15].

Presently most researches into slicing concurrent programs employ concurrent program flow graph to represent executions of concurrent programs, based on this model determine various dependencies between statements and construct concurrent program dependence graphs, finally obtain slice by traversing dependence graphs [4 -15]. As for intransitivity problem of dependence relation, various traversing methods are proposed and slices with different precision are achieved. Cheng, Zhao and Hatcliff did not consider intransitivity problem, traversed program directly and hence got low-precision slice [5-9]. To get more precise slice, we proposed an algorithm to remove some redundant statements by computing the set of statements impossible to be included [10]. Krinke and Nanda determined whether traversed statements could be added according to control flow information [11-14]. Mohapatra has computed dynamic slices of concurrent object-oriented programs based on dependence graph [15].

In the above methods, the most precise slice can be obtained by using the method proposed by Krinke and Nanda (K-N method). Despite of that, there are still some problems in slicing concurrent programs. Firstly, as concurrent program flow graph is a simple connection of all control flow graphs each representing a single concurrent unit by appending edges representing interaction between concurrent units, it is difficult to precisely determine possible synchronization activities with the model. As we know, synchronization activities change the control flow of concurrent unit and influence the precision of various dependence analysis. More over, since concurrent program flow graph also can not effectively represent non-deterministic asynchronous communication sequence, it is inevitable to ignore the override of the data flow in a single concurrent unit because of variable redefinition in other concurrent unit. So it is impossible to analyze data dependence between concurrent units globally for concurrent program as a whole. The corresponding concurrent dependence graph is not only imprecise, but also is a simple connection of all

This work was supported by projects (60373066, 60425206, 60403016) supported by National Natural Science Foundation of China
Correspondence to: Baowen Xu, Department of Computer Science and Engineering, Southeast University, 210096 Nanjing, China. Email: bwxu@seu.edu.cn

Manuscript received August, 2005.
Manuscript reviced September, 2005.

dependence graphs each representing a single concurrent unit with synchronization and interference dependent edges [10-14]. Another critical issue is that K-N method has not solved intransitivity problem in essential. Intransitivity between statements is aroused by not distinguishing different executions of statements and hence not guaranteeing that simple connection of a dependence relation is a feasible dependence sequence. If transitive dependence is created, more precise slice would be expected to be obtained.

This paper is focus on computing high-precise slices for concurrent programs and is an extension of our early work [16]. Since Ada contains abundant concurrent facilities [17], research into dependence analysis and slicing techniques for it can be easily extended to other concurrent languages. We select Ada as our research language. In this paper, we present task communication graph, a concise and effective representation for concurrent Ada programs. Based on task communication graph, we construct a new dependence graph (MSPDG) which vertex is a pair composed of program state and statement. Dependence in MSPDG is precise and transitive. By traversing MSPDG, more precise slice will be obtained more efficiently compared with other high-precision slicing methods. The rest of sections are organized as follows. Section 2 introduces task communication graph, section 3 discusses various dependences in concurrent Ada programs and proves that dependences in MSPDG are transitive, section 4 gives our slicing algorithm based on MSPDG and related work, section 5 concludes this paper.

2. Task Communication Reachability Graph

Ada supports concurrent programming by task mechanism [17]. Tasks are concurrent units and main communication means among them are rendezvous, protected objects and share variables. As concurrent program flow graph can not meet the need of high-precise dependence analysis, in this section we propose a new representation for concurrent Ada programs, which is called task communication graph. Task communication graph is obtained by extending traditional reachability analysis. The cost of reachability analysis in statement level is high [18], so we simplify the representation for tasks and get task communication graph before constructing task communication reachability graph. For convenience to describe, we first introduce a few related basic concepts, notations and terminologies.

Definition 2.1 (Directed Graph) A directed graph G is a pair <N, E>, where N is a finite non-empty set of elements called nodes, $E \subseteq N \times N$, is the set of edges between nodes.

For any edge $(n_1, n_2) \in E$, n_1 is called a direct predecessor of n_2, and n_2 is called a direct successor of n_1, denoted by $Pred(n_1, n_2)$. A path from n_1 to n_k in G is a sequence of nodes $P=(n_1, \ldots, n_k)$ such that $(n_i, n_{i+1}) \in E$ for all $1 \leq i < k$. If there is a path from n_1 to n_k in G, n_1 is called a predecessor of n_k, and n_k is called a successor of n_1, denoted by $Pred^*(n_1, n_k)$.Generally, if we do not consider nest and creation of tasks, each task has only a sequential execution flow and can be represented by control flow graph.

```
        procedure MAIN is
            protected type SharedInt(InitVal: Integer) is
                procedure Write(NewInt: Integer);
                function Read return Integer;
             private
                IntData: Integer := InitVal;
            end SharedInt;

            task T is
                entry P(X: in Integer);
            end T;

            protected body SharedInt is
                procedure Write(NewInt: Integer) is
                begin
                    IntData := NewInt;
                end Write;
                function Read return Integer is
                begin
                     return IntData;
                end Read;
            end SharedInt;

S1          A: SharedInt(1);
S2          B: SharedInt(5);

            task body T is
                C: Integer;
            begin
S3              C:=10;
S4              A.Write(A.Read +1);  // A=A+1
S5              B.Write(10);             // B=10
S6              accept P(X: in Interger) do
S7                  C:= X+5;
                end P;
             end T;

        begin
S8          A.Write(B.Read);           // A=B
S9          T.P(A.Read);
        end MAIN;
```

Fig. 1 An example program

Definition 2.2 (Control Flow Graph) A control flow graph is a directed graph $Gc=<S, E, s_S, s_E>$, where S is the node set and represents statements or predicates, E is the edge set and represents the flow of control between nodes,

the start node s_S and the exit node s_E are two special nodes representing the beginning and the end of the program respectively.

If every path from node s_S to s_y passes through node s_x, s_x is called a predominator of s_y. If every path from s_x to s_E passes through s_y, s_y is called a postdominator of s_x.

A concurrent Ada program consists of one or more tasks. Each task proceeds independently and concurrently between the points where it interacts with other tasks. Statements, like new, select, entry call, accept, and access of protected objects or shared variables indicate such interactions. In this paper, these statements are called communication statements, interactions induced by synchronization or asynchronous communications are called communication activities, the program points where communication activities take place are called by communication points. As some communication statements, such as entry call and accept with accept body, correspond to two communication activities, they are replaced by two statements in order to describe conveniently. If s is an entry call or accept statement with accept body, s is replaced with s.s and s.e in corresponding communication points. For implicit task activation and termination, two statements, cobegin and coend, are added in corresponding communication points. Each segment extracted between communication points is called a task communication region. Given the CFG of a task, if we traverse all paths starting from the start node of the task or a statement that immediately follows a communication point, and end with communication statements, task communication regions of the task can be automatically extracted. If each task communication region is represented as a node (called TCG-node), edges between nodes represent the control flow of interactions and the corresponding directed graph is called task communication graph.

Definition 2.3 (Task Communication Graph) Task communication graph is a labeled directed graph $G_T = <N, E, n_S, F, L>$, where N is the set of nodes representing task communication regions, $E \subseteq N \times N$, is the set of edges representing communication activities, L is a function that assigns a label to each edge, the initial node n_S represents the region where the task initiates its execution, F is the final nodes where the task may finish its execution.

There are four kinds of labels for communication activities. For entry E, the starting and ending edges of the entry call(accept) are labeled with E.cs, E.ce (E.as, E.ae) or reduced as E.c, E.a for no accept body. The edge is labeled with $t \rightarrow (t_{d1}, t_{d2}, \ldots, t_{dr})$ if task $t_{d1}, t_{d2}, \ldots, t_{dr}$ are activated by parent task t in some activation. The edge is labeled with $t \leftarrow (t_{d1}, t_{d2}, \ldots, t_{dr})$ if master task t must wait for the terminations of task $t_{d1}, t_{d2}, \ldots, t_{dr}$ before its termination. Edges of access of protected objects or shared variables are labeled with its statement label. Fig 1 Gives an example of Ada program. Fig 2 shows CFGs and TCGs of task t1 and t2 of the program.

Suppose that a concurrent Ada program is composed of k tasks and the main program is regarded as the first task. The TCG of the ith task is denoted by $TCG_i = <N_i, E_i, n_S^i, F_i, L_i>(1 \le i \le k)$. According to label matching of communication activities in TCGs, we can make reachability analysis and construct task communication reachability graph. In task communication reachability graph, nodes, which are called TCRG-nodes or marks and correspond to reachable states of the concurrent program as a result of a sequence of communication activities, can be represented by a k-tuple of TCG-nodes where the ith component is a TCG-node in the ith task, and edges represent communication activities that make state transition happen. Reachability analysis is started from an initial mark denoted by m_S where $m_S = (n_S^0, \#, \ldots, \#)$ and # indicates that the corresponding task is inactive, successors are generated according to possible communication activities. In order to get all possible states, at one time only one communication activity is permitted to takes place. For any succeeding mark, it is different from its direct precedent mark only in variations of components where the corresponding communication activity happens. The following lemma 2.1 gives the generating rule.

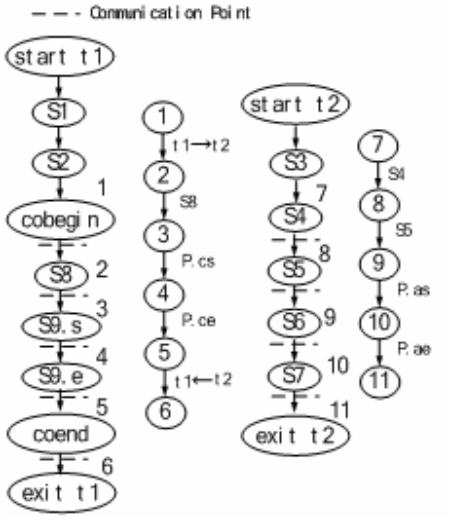

Fig. 2 CFG and TCG of task t1 and t2.

Lemma 2.1 Let m and m′ be two marks, m′ is a succeeding mark of m iff any one of the following

conditions holds ($1 \le i, j, p < k$) where k is the number of tasks:

(1) there exist i and j such that for all $p \neq i, j$, $m[p] = m'[p]$ and
  (i) $(m[i], m'[i]) \in E_i$ and $(m[j], m'[j]) \in E_j$
  (ii) $L(m[i], m'[i])$=E.cs and $L(m[j], m'[j])$=E.as, or $L(m[i], m'[i])$=E.ce and $L(m[j], m'[j])$=E.ae, or $L(m[i], m'[i])$=E.c and $L(m[j], m'[j])$=E.a ;

(2) there exist i, j1, j2, …, jr such that for all $p \neq i$, j1, j2, …, jr, $m[p]=m'[p]$ and
  (i) $(m[i], m'[i]) \in E_i$ and $L(m[i], m'[i]) = ti \rightarrow (t_{j1}, t_{j2}, \ldots, t_{jr})$
  (ii) For any j in j1, j2, …, jr, $m[j]=\# \wedge m'[j]=n_S{}^j$;

(3) there exist i, j1, j2, …, jr such that for all $p \neq i$, j1, j2, …, jr, $m[p]=m'[p]$ and
  (i) $(m[i], m'[i]) \in E_i$ and $L(m[i], m'[i])=t_i \leftarrow (t_{j1}, t_{j2}, \ldots, t_{jr})$
  (ii) For any j in j1, j2, …, jr, $m[j] \in F_j$ and $m'[j]=\#$;

(4) there exists i such that for all $p \neq i$, $m[p]=m'[p]$ and $(m[i], m'[i]) \in E_i$ and $L(m[i], m'[i])$ is a statement label.

The four conditions correspond to rendezvous, task activation, waiting for termination and access of protected objects or shared variables respectively. Fig3 shows TCRG of the program in Fig 1.

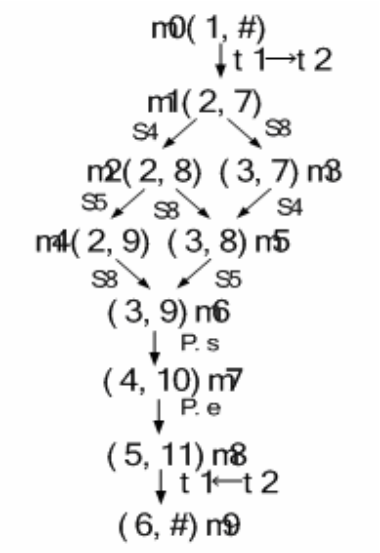

Fig.3 TCRG of the program in Fig.1 .

By TCRG which describes the flow of communication activities for concurrent programs as a whole, mutual control and data flow influences in tasks maybe precisely analyzed. Once the sequence of communication activities is established, the behavior of concurrent program is equivalent to the behavior of a sequential program. From this point of view, every path from the initial TCRG-node to a final TCRG-node corresponds to one sequential program and then TCRG is a representation of a combination of multiple sequential programs. Therefore, typical analysis techniques for sequential programs may be applied in TCRG.

3 Dependence Analysis Based on TCRG

In this section, we discuss dependence analysis based on TCRG. First we present a few concepts related to dependence analysis, then analyze various primary dependences in concurrent Ada programs and construct a dependence graph which node is a pair of program reachable state and statement, finally prove that dependence in the dependence graph is transitive.

3.1 Related concepts and property

In TCRG, due to different sequences of communication activities one statement may appear in different executions and might have different control or data flow information. As mentioned in section 1, not distinguishing different appearances of one statement in different executions may result in intransitivity problem. As TCRG is a sequential flow graph where each mark indicates one program state associated with communication activities, statement and mark can form to a pair to represent different executions of one statement. This pair is called M-S pair, represented by symbol Λ. We define the function $m(\Lambda)$ and $s(\Lambda)$ to return the state and statement component of Λ. For any mark m, each statement that appears in some component (TCG) of m may execute in the state of m and only these statements can be combined with m, i.e., any combination of m with other statements is meaningless.

Based on the concept of M-S pair, we define executable sequence to represent an execution of a concurrent program.

Definition 3.1 (Executable Sequence) Given the TCRG of a concurrent Ada program, G_T, an executable sequence in G_T is an ordered sequence of M-S pair $(\Lambda_1, \Lambda_2, \ldots, \Lambda_n)$ which forms a valid execution of the program.

All executable sequences in G_T is denoted by $ES(G_T)$. If one sequence of M-S pair is a subsequence of some executable sequence in $ES(G_T)$, it is called a feasible sequence of the program. In Fig 3, l_1=(<m0, start t1>, <m0, S1>, <m0, S2>, <m0, cobegin>, < m1, start t2>, <m1, S8>, <m1, S3>, <m3, S4>, <m5, S5>, <m6, S6>, <m6, S9>, <m7, S7>, <m8, coend>) is an executable sequence of the example. If we pick up some nodes in l_1 and form a

sequence (<m0, S1>, <m3, S4>, <m7, S7>) it is a feasible sequence.

In TCRG, program states are tightly associated with communication activities. Once a communication activity takes place in some program state, the corresponding program state will change. In some program states there exist multiple possible communication activities, so different program states will be reached after different communication activities. If one communication activity, denoted by s, happen in the state of m, we denote the generated state by Succ(m, s). In the analysis of data flow, only the data flow information related to s will reach Succ(m, s).

According to the above definitions, we can get the following property of feasible sequences.

Property 3.1 Given a concurrent program consisting of k tasks, let l be an ordered sequence of M-S pairs (Λ_1, Λ_2, …, Λ_n), l is a feasible sequence of the program iff

(1) for all i, $1 \le i < n$, either

(i) $m(\Lambda_i)=m(\Lambda_{i+1})$ or Pred*($m(\Lambda_i)$, $m(\Lambda_{i+1})$) if (Λ_i) is not a communication statement

or (ii)Succ($m(\Lambda_i)$,$s(\Lambda i)$)=$m(\Lambda_{i+1})$ or Pred*(Succ($m(\Lambda_i)$, $s(\Lambda_i)$), $m(\Lambda_{i+1})$) if $s(\Lambda_i)$ is a communication statement;

(2) for all t in the program,

$l|_t = (\Delta_1, \Delta_2, \ldots, \Delta_j) \Rightarrow \forall 1 \le p < j-1$: Pred*($s(\Delta p)$, $s(\Delta p+1)$)

where $l|_t$ is the subsequence of l in which all M-S pairs, which statement components do not appear in task t, have been removed.

3.2 Dependence Analysis

In General, there are two types of dependences, control dependence and data dependence. Informally, let Λ_1, Λ_2 be two M-S pairs, Λ_1 is control dependent on Λ_2 if whether Λ_1 can be executed or not depends on the execution of Λ_2, and Λ_1 is data dependent on Λ_2 if the execution of Λ_1 use variables defined in Λ_2. According to the cause of dependence, dependences in concurrent Ada programs can be further classified into common control and data dependence that exist like in sequential programs, intra and inter task synchronization control dependences and inter task data dependence.

Common control and data dependence, which are also called control and data dependence, happen in one task.

Definition 3.2 (Control Dependence) A M-S pair Λ_1 is control dependent on M-S pair Λ_2, denoted by CD(Λ_2, Λ_1), if the following hold,

(1) (Λ_2, Λ_1) is a feasible sequence;

(2) $s(\Lambda_2)$ is a predicate statement, $s(\Lambda_1)$ and $s(\Lambda_2)$ are statements from one task;

(3) there is a path P from $s(\Lambda_2)$ to $s(\Lambda_1)$ in the corresponding CFG such that $s(\Lambda_1)$ is a postdominator of each $s \neq s(\Lambda_2)$ in P and $s(\Lambda_1)$ is not a postdominator of $s(\Lambda_2)$.

Definition 3.3 (Data Dependence) A M-S pair Λ_1 is data dependent on M-S pair Λ_2, denoted by DD(Λ_2, Λ_1), if there exists a variable v such that the following hold,

(1) $s(\Lambda_2)$ and $s(\Lambda_1)$ are statements from one task;

(2) Λ_2 defines v and Λ_1 uses v;

(3) there exists an executable sequence from Λ_1 to Λ_2 on which v is not redefined.

In addition to the above dependences, there are other dependences induced by inter task communication activities. Rendezvous alternates the control flow of a program. In some scheduling, rendezvous may not be triggered or finished, which will make some of the succeeding statements wait until other tasks abort it. Consequently, whether these succeeding statements may execute or not depends on the execution of those statements that take part in the rendezvous. This dependence is called intra-task synchronization dependence.

Definition 3.4 (Intra-task Synchronization Dependence) A M-S pair Λ_1 is intra-task synchronization dependent on M-S pair Λ_2, denoted by SD(Λ_2, Λ_1), if the following hold,

(1) $s(\Lambda_2)$ is an entry call, accept statement ;

(2) $s(\Lambda_2)$ is a predominator of $s(\Lambda_1)$ in the corresponding CFG.

There exist two inter-task synchronization dependences. One is caused by rendezvous. Entry call and accept statements are mutually dependent on each other. The other is caused by task activation or termination. As task termination and implicit activation have no obvious dependence between statements, they can be ignored. Only explicit task activation is considered.

Definition 3.5 (Inter-task Synchronization Dependence) A M-S pair Λ_1 is inter-task synchronization dependent on M-S pair Λ_2, denoted by TSD (Λ_2, Λ_1), if the following hold,

(1) $s(\Lambda_1)$ is an entry call and $s(\Lambda_2)$ is an accept statement or vice versa;

(2) $s(\Lambda_1)$ is a task-begin statement and $s(\Lambda_2)$ is a *new* statement that dynamically creates the corresponding task.

In definition 3.5, considering that multiple entry call

statements might rendezvous with one accept statement in some program state, in each possible rendezvous the accept statement needed to be assigned a serial number for avoiding unnecessary indirect dependence between these entry call statements.

Inter task data dependence is generated as a result of data communications between tasks, such as rendezvous with parameters, access of shared variables and protected objects. For rendezvous with parameters, actual parameters can be substituted for formal parameters as every rendezvous is instantiated in TCRG. For access of protected objects, each data member is regarded as a variable and the set of definition and use of data members are summarized for the computation of inter task data dependence.

Definition 3.6 (Inter-task Data Dependence) A M-S pair Λ_1 is data dependent on M-S pair Λ_2, denoted by TDD (Λ_2, Λ_1), if there exists a variable v such that the following hold,

(1) $s(\Lambda_2)$ and $s(\Lambda_2)$ are statements from different tasks;
(2) Λ_2 defines v and Λ_1 uses v;
(3) there exists an executable sequence from Λ_1 to Λ_2 on which v is not redefined.

Based on the above definitions, we can define a dependence graph where nodes are M-S pairs for concurrent programs.

Definition 3.7 (M-S Pair Dependence Graph) Given the TCRG of a concurrent Ada program, a M-S Pair Dependence Graph is a directed graph G_D= <M, S, MS, E>, where M is the set of TSRG-nodes, S is the set of statements, MS is the set of nodes, MS$\subseteq$M×S, E is the set of edges, E={(Λ_i, Λ_j)| dep(Λ_i, Λ_j), Λ_i, $\Lambda_j\in$MS, dep$\in${CD, DD, SD, TSD, TDD}}.

As some executable sequences of a concurrent program have completely identical dependences except that program states of a few M-S pairs are different. Such executable sequences are called equivalent executable sequences. As we know, if the sequence of M-S pairs associated with communication statements is established, statements, which are not communication statements, can be executed in different program states generated as a consequence of communication activities that happen in other tasks. In such situation, the formed executable sequences are equivalent because it is impossible that statements, which are not communication statements, are directly dependent on communication statements from other tasks. Accordingly, only one of equivalent executable sequences to analyze is needed to be analyzed. In order to use TCRG conveniently, we choose those executable sequences where statements that are not communication statements are combined with the same program states as those of the immediate succeeding communication statement. In Fig 3, if we substitute <m3, S3> for <m1, S3> in l_1, we can get another executable sequence l_2=(< m0, start t1>, <m0, S1>, <m0, S2>, <m0, cobegin>, < m1, start t2>, <m1, S8>, <m3, S3>, <m3, S4>, <m5, S5>, <m6, S6>, <m6, S9>, <m7, S7>, <m8, coend>>. As S3 is not a communication statement, l_1 and l_2 is equivalent. And l_2 is chosen to be analyzed.

Control and synchronization dependences can be obtained by combining predicate and synchronization statements with the corresponding program states in TCRG. Data flow information, such as reach definition of variables, will be computed hierarchically. Along TCRG, we first select the task communication regions where the communication activity may happen, then compute their flow information along the subgraphs corresponding to the TCGs and combine it with program states, finally let the information flow into the succeeding TCRG-node related to communication activity. Based on the above information, data dependence can be easily determined.

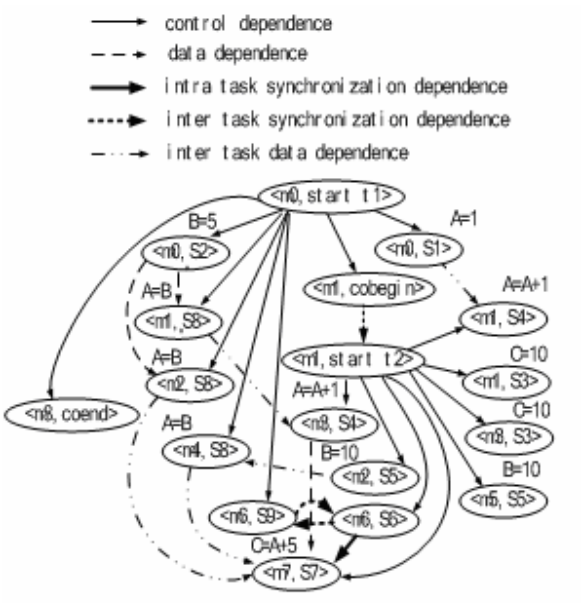

Fig4 MSPDG of the program in Fig.1 .

In contrast to conventional concurrent program dependence graph, MSPDG is built on TCRG. Dependence in concurrent programs is analyzed globally, so it is not only precise but also transitive. Below, we define transitive dependence and then give its proof.

Definition 3.8 (Transitive Dependence) Given the MSPDG of a concurrent Ada program, a M-S pair Λ_y is

transitive dependent on M-S pair Λx, denoted by Dep* (Λ_x, Λ_y), if the following hold,
(1) there exists a path $l = (\Lambda_x = \Lambda_1, \Lambda_2, \ldots, \Lambda_y = \Lambda_n)$in G_D;
(2) l is a feasible sequence of the program.

Theorem 3.1 The dependence relation in MSPDG is transitive.

Proof. Let $l=(\Lambda_x=\Lambda_1, \Lambda_2, \ldots, \Lambda_y=\Lambda_n)$ be an arbitrary path in MSPDG, according to definition 3.8, if we prove theorem 3.1, we only need to show that l is a feasible sequence of the program. As l is a path in MSPDG, according to the above definitions of dependence, it is easy to get l satisfies the first requirement of property 3.1. Below, we show it is also satisfies the second.

Let t be an arbitrary task of the program. When $l|_t = (\Delta_1, \Delta_2, \ldots, \Delta_j)$ is not empty, for all i, $1 \le i < j-1$:

(i) if Δ_i is adjacent to Δ_{i+1} in l, i.e., Δ_{i+1} is dependent on Δ_i, the dependence is an intra task dependence because $s(\Delta_i)$ and $s(\Delta_{i+1})$ are from one task t. From the definitions of intra task dependence, Pre*($s(\Delta_i)$, $s(\Delta_{i+1})$) will be derived;

(ii) if Δ_i is not adjacent to Δ_{i+1} in l, i.e., Δ_{i+1} is not dependent on Δ_i, let the path from Δ_i to Δ_{i+1} in l be $(\Delta_i, \Lambda_u, \ldots, \Lambda_{u+k}, \Delta_{i+1})$, since $s(\Lambda_u)$ and $s(\Lambda_{u+k})$ are not from task t, $s(\Delta_i)$, $s(\Lambda_u)$, $s(\Lambda_{u+k})$ and $s(\Delta_{i+1})$ should be communication statements. From the definitions of inter task dependences Pre*(Succ($m(\Delta_i)$, $s(\Delta_i)$), $m(\Delta_{i+1})$) can be derived. Let $(n_i, n_u, \ldots, n_{u+k}, n_{i+1})$ be a sequence which is formed by extracting the TCG-nodes in task t from $(\Delta_i, \Lambda_u, \ldots, \Lambda_{u+k}, \Delta_{i+1})$. Then we can find a path such that $(n_i, n_u, \ldots, n_{u+k}, n_{i+1})$ is a subsequence of the path. According to the construction of TCG Pre*($s(\Delta_i)$, $s(\Delta_{i+1})$) can be obtained.

As l satisfies property 3.1, l is a feasible sequence. Therefore, theorem 3.1 holds. □

4 Slicing Concurrent Ada Programs

In this section, we first present an algorithm for slicing concurrent Ada programs based on MSPDG and apply it in a case, then compare it with other slicing algorithms in precision and efficiency.

4.1 Slicing Algorithm Based On MSPDG

Let s be a statement, as it might execute in one or multiple program states, it will combine with these program states and form one or multiple M-S pairs. We denote the corresponding set by MS(s).

Definition 4.1 (Slice) Given a statement s, the slice of s is Slice(s)={ s′ | Dep*(<m′, s′>, <m, s>), <m, s>∈MS(s)}.

Theorem 3.1 shows that dependence relation in MSPDG is transitive. Therefore, based on it, computation of slice is a simple graph traversing problem. Fig 5 gives the corresponding algorithm.

As MSPDG is built on TCRG, which overcomes the drawbacks of traditional concurrent program flow graph, dependence in concurrent programs is analyzed globally. Hence, it is more precise than traditional dependence analysis. Based on MSPDG, high-precision slice can be obtained.

```
Input: MSPDG, slicing criterion s and MS(s)
Output: Slice(s) // the slice of s
Initialization: W = MS(s)
1 while W≠Φ do
2   Remove the next element Λfrom W;
3   for each (Λ′, Λ) in MSPDG do
4     if Λ′ is not visited
5       mark Λ′ visited, W = W∪{Λ′};
6       Slice(s) = Slice(s) ∪ {s(Λ′)};
      end if
     end for
  end while.
```

Fig. 5 An algorithm of slicing based on MSPDG

To compare with other slicing methods easily, we classify intransitivity problems into two types. The first type refers to the problem that dependence relation is intransitive because dependence sequence does not conform to the requirement of control flow, i.e., it is not a subsequence of a possible execution. In the program of Fig 1, S5→S8→S4 is a dependence sequence, but S4 is not dependent on S5 because it is not a subsequence of a possible execution. The second type refers to the problem that dependence relation is still intransitive although dependence sequence is a subsequence of a possible execution. In the program of Fig 1, S1→S4→S7 is a dependence sequence and it is a subsequence of a possible execution, however, there exits no execution in which S7 is dependent on S1.

Fig 4 shows the MSPDG of the example. For the first type of intransitivity problem, if we compute the slice of S4 based on Fig4, we will start from <m1, S4> and <m3, S4> and traverse it and get {<m1, S4>, <m0, S1>, <m0, start t1>, <m0, start t2>, <m3, S4>, <m0, cobegin>, <m1, S8>, <m0, S2>}. After removing the state component and

additional statements, like cobegin, we can get Slice(S4)={ start t1, start t2, S1, S2, S4, S8 } which does not include S5. Observe M-S pair dependence sequences <m2, S5>→<m4, S8> and <m1, S8>→<m3, S4>, we can find that distinguishing different execution of S8 effectively cut the intransitive dependence of S5→S8→S4. For the second type of intransitivity problem, if we compute the slice of S7, we traverse Fig 4 starting from <m7, S7>, and get {<m7, S7>, <m2, S8>, <m0, S2>, <m0, start t1>, <m3, S4>, <m0, start t2>, <m0, cobegin>, <m1, S8>, <m4, S8>, <m2, S5>, <m6, S6>, <m6, S9>}. Then we get Slice(S7)={ start t1, start t2, S2, S4, S5, S6, S7, S8, S9} which does not include S1. Observe M-S pair dependence sequences <m0, S1>→<m1, S4> and <m3, S4>→<m7, S7>, we may find that distinguishing different execution of S4 cut the intransitive dependence of S1→S4→S7.

Let P be a concurrent program with n statements, k tasks, and c_1 entry call and accept statements, c_2 statements of access of protected objects, shared variables and dynamical task creation. As every entry call and accept statement corresponds to two communication points, every statement of access of protected objects, shared variables or dynamical task creation corresponds to one communication point, every task corresponds to one task entry program point, and other two communication points, i.e., the program points where the task waits for activation and termination, the total number of TCG-nodes is $(2c_1+ c_2+3k)$ in the worst case according to the construction of TCG. Let the number of TCG-nodes from every task be p_1, p_2, … , p_k, the number of the generated TCRG-nodes is $p_1 \times p_2 \times \ldots \times p_k$ in the worst case. According to Churcy inequality, $p_1 \times p_2 \times \ldots \times p_k \le ((2c_1+ c_2+3k)/k)^k$. Consequently, the number of TCRG-nodes is up to $((2c_1+c_2)/k+3)^k$ at most. Hence, the number of nodes in MSPDG is $n((2c_1+ c_2)/k +3)^k$, and the worst complexity of slicing based on MSPDG is $O(n^2((2c_1+c_2)/k +3)^{2k})$.

4.2 Comparison with related methods

Methods based on concurrent program dependence graph can not address the second type of intransitivity problem because of imprecision of data flow analysis [5-14]. For the first type of intransitivity problem, different traversing methods may lead to different precision. Not considering intransitivity problem, Cheng's algorithm simply traverses PDN, program dependence net for concurrent Ada programs, Zhao's and Hatcliff's algorithm does so for concurrent Java programs [6-9]. Consequently, their methods do not solve the first intransitivity problem completely and precision of slice is low. We have present algorithms for slicing concurrent Ada programs [10]. This method can remove some redundant statements by computing the set of statements impossible to be included according to the information of control flow of all statements in dependence sequence. Since the set is a conservative result, our algorithm addresses the first intransitivity problem partially and the precision of the slice is increased in some extent. Krinke and Nanda determined whether traversed statements could be added according to precise control flow information. Their algorithms guarantee that each dependence sequence is a subsequence of a possible execution. So Krinke and Nanda's methods thoroughly address the first intransitivity problem. However, the precision of the slice obtained by their methods is still lower than our method based on MSPDG in that our method in this paper addresses two types of intransitivity problems.

Table 1: Comparison of Various Slicing Methods

Methods	Address the first problem	Address the second problem	Precision	Time Complexity
Cheng, Zhao, and Hatcliff	No	No	Low	$O(n^2)$
Chen, Xu	Partially Address	No	Middle	$O(n^4)$
Krinke, Nanda	Yes	No	High	$O(n^2(n/k)^{2k})$
Our method	Yes	Yes	Highest	$O(n^2((2c_1+ c_2)/k +3)^{2k})$

For convenience, table 1 compares the above slicing algorithms. In table 1, n indicates the number of statements in the program, k indicates the number of concurrent units, and c indicates the number of communication statements (c=c1+c2). From table 1, we can find that the most precise slice can be computed by our method proposed in this paper. Further, as c is much less n in common case, the efficiency of our method is higher in worst case than Krinke and Nanda's method..

5 Conclusions

In this paper, we have proposed task communication reachability graph for representing executions of concurrent Ada programs. Based on TCRG, we precisely determine various dependences in concurrent Ada programs and construct M-S pair dependence graph. As dependence relation in MSPDG is precise and transitive, more precise slice will be obtained more efficiently by

traversing MSPDG in contrast to other high-precision slicing methods. However, due to intrinsic complexity of concurrent programs, concurrent Ada programs that can be analyzed with our method in this paper should not include recursive procedures with communication activities and tasks with unbounded number [19].Although our method can get more precise slices than other previous methods, its efficiency is needed to improve. In future work, we plan to realize this goal along two ways. One is to develop a configurative method that combines strengths of our method and traditional polynomial algorithms and compute slices with different precision according to different requirements. The other is to reduce TCRG with keeping transitive property.

References

[1] Weiser, M. Program Slicing. IEEE Transactions on Software Engineering, 1984, 16(5): 498-509.

[2] Tip, F. A survey of program slicing techniques. Journal of programming language, 3(3), 1995.

[3] Horwitz, S. Interprocedural slicing using dependence graphs. ACM Transactions on Programming Languages and Systems, 1990, 12(1): 26-60.

[4] Chen Z.Q, Xu B.W, Zhao J. J. An overview of methods for dependence analysis of concurrent programs. ACM SIGPLAN Notices, 2002, 37 (8): 45–52.

[5] Cheng, J. Task dependence nets for concurrent systems with Ada 95 and its applications. ACM TRI-Ada International Conference, St. Louis, Missouri, USA: ACM Press, 1997, 67-78.

[6] Zhao, J.J. Multithreaded dependence graphs for concurrent Java programs. International Symposium on Software Engineering for Parallel and Distributed Systems, Los Angeles, California, USA: IEEE CS press, 1999, 13-23.

[7] Zhao, J.J, Li, B.X. Dependence based representation for concurrent Java programs and it's application to slicing. International Symposium on Future Software Technology 2004 (ISFST2004), Xian, China: 105–112.

[8] Hatcliff, J. A Formal Study of Slicing for Multi-threaded Programs with JVM Concurrency Primitives. International Symposium on Static Analysis 1999(SAS'99), September, Venice, Italy, LNCS1694: 1-18.

[9] Ranganath V.P, Hatcliff, J. Pruning the Detection of Interference and Ready Dependence for Slicing Concurrent Java Programs. Technical report, Computing and Information Sciences Department, Kansas State University, March 2005.

[10] Chen Z.Q, Xu B.W, Zhao J. J, Yang H.J. Static Dependency Analysis for Concurrent Ada 95 Programs. Ada-Europe 2002: 219-230.

[11]Krinke, J. Static slicing of threaded program. ACM SIGPLAN Notices, 1998, 33(7): 35-42.

[12]Krinke, J. Context-Sensitive Slicing of Concurrent Programs. ACM SIGSOFT Symposium on Foundations of Software Engineering 2003 held jointly with 9th European Software Engineering Conference (ESEC/FSE 2003), Helsinki, Finland, USA: ACM press, 2003, 178-187.

[13]Nanda, M.G, Ramesh, S. Slicing concurrent programs. ACM SIGSOFT Software Engineering Notes, 2000, 25(5): 180-190.

[14] Nanda, M.G. Slicing Concurrent Java Programs: Issues and Solutions. Ph.D. thesis, Indian Institute of Technology, Bombay, October 2001.

[15] Mohapatra, D. P, Mall, R, Kumar, R. Computing dynamic slices of concurrent object-oriented programs. J. Information and Software Technology, 2005, 47(12): 805-817.

[16] Qi, X. F, Xu, B.W. Dependence analysis of concurrent programs based on reachability graph and its applications. International Conference on Computational Science 2004 (ICCS 2004), May, Poland. LNCS3036: 405-408.

[17] ISO/IEC 8652: 1995(E). Ada reference manual-language and standard libraries.

[18] Pezze, M, Taylor, R. N, Young, M. Graph models for reachability analysis of concurrent programs. ACM Transactions on Software Engineering and Methodology, 1995, 4(2): 171-213.

[19] Ramalingam, G. Context-Sensitive Synchronization-Sensitive Analysis is Undecidable. ACM Transactions on Programming Languages and Systems, 2000, 22(2): 416-430.

Xiaofang Qi is a Ph.D. Candidate in the Department of Computer Science & Engineering at Southeast University. Ms. Qi's research focus is analysis techniques of concurrent programs. She has investigated issues in dependence analysis of program understanding, and slicing, etc.

Baowen Xu is a professor in the Department of Computer Science & Engineering at Southeast University. Dr. Xu's research focus is programming languages and Implementations, software testing and quality assurance technology, software reengineering, and intelligent software technology.