Dependence Analysis of Concurrent Programs Based on Reachability Graph and Its Applications

Xiaofang Qi and Baowen Xu

Department of Computer Science and Engineering, Southeast University,
210096 Nanjing, China
{xfqi, bwxu}@seu.edu.cn

Abstract. This paper presents task synchronization reachability graph(TSRG) for analyzing concurrent Ada programs. Based on TSRG, we can precisely determine synchronization activities in programs and construct a new type of program dependence graph, TSRG-based Program Dependence Graph(RPDG), which is more precise than previous program dependence graphs and solves the intransitivity problem of dependence relation in concurrent programs in some extent. Various applications of RPDG including program understanding, debugging, maintenance, optimization, measurement are discussed.

1 Introduction

As concurrent systems are intensively used day by day, approaches to analyze, comprehend, test and maintain concurrent programs are imperatively demanded. Since determining dependencies between statements is indispensable and crucial to such activities, dependence analysis gradually attracts many researchers to make efforts[1, 2]. Present studies on dependence analysis for concurrent programs are mostly based on concurrent program flow graph. With the model, Krinke and Nanda have computed dependence information of concurrent programs without synchronization[3, 4]. Zhao and Cheng have considered effects of synchronization. However, they analyzed synchronization activities merely by syntactical matching[1, 5]. This processing may produce spurious results leading to inaccurate dependence analysis in most case because some of these synchronization activities are possible to happen while some of them not. We have proposed an adapted MHP(May Happen in Parallel) algorithm to increase the precision of determining synchronization activities[6]. Unfortunately, this approach is still conservative because MHP algorithm only calculates a conservative approximation of MHP statement pairs. Reachability graph, recording all possible reachable states and describing executions of concurrent programs, includes various precise information related to dependence analysis[7, 8]. To improve the accuracy of dependence analysis, we employ reachability graph as the model for analysis and present a new method of dependence analysis for concurrent Ada programs.

M. Bubak et al. (Eds.): ICCS 2004, LNCS 3036, pp. 405–408, 2004.

2 Task Synchronization Reachability Graph

A concurrent Ada program consists of one or more tasks. Each task proceeds independently and concurrently between the points(called by synchronization points) where it interacts with other tasks by inter task synchronization activities during its lifecycle. Statements, like *new, entry call, accept, select, select-else*, indicate such synchronization activities. Each segment extracted between synchronization points is called a task region.

Definition 2.1. Task synchronization graph(TSG) is a labeled directed graph $G_T = <N, E, n_S, F, L>$, where N is the set of nodes corresponding to task regions, $E \subseteq N \times N$, is the set of edges representing synchronization activities, L is the mapping function, n_S is the initial node in which the statement *begin* appears, and F is the final nodes in which the statement *end* appears.

For a given 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. If task s_1, s_2, …, s_n are activated by parent task p in some activation, the edge is labeled with $(p>(s_1, s_2, \ldots, s_n))$. The edge labeled with $(m<(d_1, d_2, \ldots, d_n))$ specifies that master task m is to wait for the terminations of task $d_1, d_2, \ldots, d_n$ before its termination.

TSG emphatically describes synchronization and concisely represents the execution for single task. Task synchronization reachability graph gives the behavior of an entire concurrent program and is constructed from the TSGs of the tasks that compose the program. Suppose that a concurrent Ada program is composed of k tasks(the main program is processed as the first task) and the TSG of the *i*th task is denoted by $TSG_i = <N_i, E_i, n_S^i, F_i, L_i>$ $(1 <= i <= k)$, then a TSRG-node m is a k-tuple of TSG-nodes (m[1], m[2], …, m[k]) where $m[i] \in N_i \cup \{\perp\}$, $\perp$ indicates the corresponding task is inactive.

Definition 2.2. Task synchronization reachability graph(TSRG) is a labeled directed graph $G_R = <M, E, L, m_S, F>$, where M is the set of TSRG-nodes, one for indicating an execution state of the program, $E \subseteq M \times M$, is the set of edges, each corresponding to one possible inter task synchronization activity, L is the mapping function, m_S is the initial node, $m_S = (n_S^1, \perp, \ldots, \perp)$, F is the final nodes representing final state. There is an edge from m to m′ iff any of the following conditions holds(i, j, l = 1, 2, …, k) where k is the number of tasks:

(1) $\exists i\ ((m[i], m'[i]) \in E_i \wedge L(m[i], m'[i]) = i>(s_1, s_2, \ldots, s_n) \wedge$
$(\forall j(\ j = s_1, s_2, \ldots, s_n)\ m[j] = \perp \wedge m'[j] = n_S^j))$ $\quad (l \neq i,\ j,\ m[l] = m'[l])$

(2) $\exists i\ ((m[i], m'[i]) \in E_i \wedge L(m[i], m'[i]) = m<(d_1, d_2, \ldots, d_n) \wedge$
$(\forall j(\ j = d_1, d_2, \ldots, d_n)\ m[j] \in F_j \wedge m'[j] = \perp))$ $\quad (l \neq i,\ j,\ m[l] = m'[l])$

(3) $\exists i\ \exists j\ ((m[i], m'[i]) \in E_i \wedge (m[j], m'[j]) \in E_j \wedge$
$((L(m[i], m'[i]) = E.cs \wedge L(m[j], m'[j]) = E.as) \vee$
$(L(m[i], m'[i]) = E.ce \wedge L(m[j], m'[j]) = E.ae) \vee$
$(L(m[i], m'[i]) = E.c \wedge L(m[j], m'[j]) = E.a)))$ $\quad (l \neq i,\ j,\ m[l] = m'[l])$.

In definition 2.2, labels are similar to those in definition2.1 except that the starting(ending) of rendezvous are labeled with E.s(E.e) or reduced as E. Three conditions correspond to task activation, waiting for termination and rendezvous respectively. By TSRG, we can precisely determine synchronization activities and get more accurate MHP statement pairs.

3 TSRG-Based Program Dependence Graph and Its Applications

Considering that TSRG provides all global reachable states and one statement may appears simultaneously in more than one TSRG-nodes which may reside in different control flow branches of TSRG, then we propose a new paradigm of dependency between one statement binding with its TSRG-node and another.

Definition 3.1. TSRG-based program dependence graph (RPDG) of a concurrent Ada program is a directed graph $G_D = <M, S, MS, E>$, where M is the set of TSRG-nodes, S is the set of statements, $MS = M \times S$, is the set of RPDG-nodes, $E \subseteq MS \times MS$, is the set of edges, $E = \{(<m_1, s_1>, <m_2, s_2>) \mid Dep(<m_1, s_1>, <m_2, s_2>), Dep \in \{DepDc, DepAc, DepRc, DepCc, DepVc, DepSd, DepCd\}\}$.

In definition 3.1, various dependencies can be primarily classified into control and data dependencies. DepDc, DepAc, DepRc, DepCc, DepVc represent direct, activation, rendezvous, competence, virtual control flow dependencies respectively. Direct control flow dependency exists in task regions, similar to control dependency appearing in sequential programs. Activation, rendezvous, competence control dependency exist between task regions and are induced respectively by task activation, rendezvous, competence for a same entry. Virtual control dependency contributes to keep the connectivity of intra task control dependency on the border between task regions. Since statements of definition and reference on variables may execute concurrently or sequentially in some execution, we classify data dependencies into concurrent and sequential data dependency, denoted by DepSd, DepCd.

Dependencies in RPDG possess special property in transitivity, which do not appear in traditional program dependence graphs(PDG) where dependencies are defined between statements. Below, we analyze two main cases in concurrent programs where intransitive dependency happens in PDG:

(1) When multiple tasks compete for one resource (e.g. *accept* statement), only one of them can occupy and consume it. This will lead to several exclusive program segments from those tasks taking part in the competence, i.e., only one of the segments can be executed in one execution of the program. Obviously, it's impossible that there exists dependency among these exclusive segments.

(2) From one statement s_1 in $task_i$, the dependency propagates back into another statement s_2 in $task_i$ by inter task dependency sequence. When s_1 and s_2 appear in different branches of control flow or s_1 always executes before s_2 in any execution of $task_i$, s_1 is impossible to indirectly depend on s_2.

However, for such two cases imprecise transitivity of dependency sequence may be hindered in RPDG in some extent. In (1), there must exist multiple TSRG-nodes representing each competence and those exclusive segments will reside in different branches in TSRG. If s_1, s_2 are such exclusive statements and $Dep(<m_2, s_2>, <m_2, s>)$, $Dep(<m_1, s>, <m_1, s_1>)$ holds, then there's no transitive dependency between s_1 and s_2 because s appears respectively in two different TSRG-nodes m_1 and m_2 residing in different branches. Similar situation may happen in the former case of (2). Although the dependency is intransitive in the latter case of (2), we can say dependencies in RPDG is transitive in most of cases.

In addition to having better transitivity than traditional PDG, dependence analysis in RPDG are more accurate because of precisely detecting synchronization activities and MHP pairs by TSRG. Thus, RPDG may be used in various software engineering activities including program understanding, slicing, debugging, optimization, complexity measurement, maintenance and etc. Given a concurrent Ada program consisting of k tasks, n statements including c *entry call* and *accept* statements, the cost of RPDG is $O(n(2c/k+3)^k)$ in worst case.

4 Conclusions

Based on task synchronization reachability graph, we have constructed a new type of program dependence graph – TSRG-based Program Dependence Graph(RPDG) for concurrent Ada programs. RPDG is more precise than previous program dependence graphs and solves the intransitivity problem of dependence relation in concurrent programs in some extent. Nevertheless, we have mainly consider several primary aspects of task mechanism. Some constructs, such as communications by shared variables, protected object, arrays of tasks and etc, have not been discussed in detail. In the future work, we will promote our research more systematically and extend to other concurrent languages to facilitate analysis for more concurrent programs.

References

1. Cheng, J.: Task dependence nets for concurrent systems with Ada 95 and its applications. In: ACM TRI-Ada International Conference, St. Louis, Missouri, USA: ACM Press (1997) 67–78
2. Tip, F.: A Survey of Program Slicing Techniques. Journal of Programming Languages, Vol. 3 (1995) 121–189
3. Krinke, J.: Static slicing of threaded program. ACM SIGPLAN Notices, Vol. 7 (1998) 35–42
4. Nanda, M.G., Ramesh, S.: Slicing concurrent programs. ACM SIGSOFT Software Engineering Notes, Vol. 25 (2000) 180–190
5. Zhao, J.: Multithreaded dependence graphs for concurrent Java programs. In: International Symposium on Software Engineering for Parallel and Distributed Systems, Los Angeles, California, USA: IEEE CS press (1999)13–23
6. Chen, Z.Q., Xu, B.W.: An Approach to Analyzing Dependence of Concurrent Programs. Journal of Computer Research and Development, Vol. 39 (2002) 159–164
7. Qi, X.F, Chen, Z.Q., Xu, B.W.: A Petri Net Representation of Concurrent Ada Program and Its Application for Communication Slice. Journal of Nanjing University, Vol. 38 (2002) 37–42
8. Dwyer, M. B., Clarke, L. A.: A Compact Petri Net Representation and Its Implication for Analysis. IEEE Transaction on Software Engineering, Vol. 22 (1996) 794–811