A Technique for Parallel Reachability Analysis of Java Programs

Raghuraman Rangarajan , Sridhar Iyer and G. Sajith

Abstract – **Reachability analysis is a well-known static analysis tool for the verification of critical properties in concurrent programs and it involves the systematic enumeration of all possible global states of program execution. However, it suffers from the problems of exponential space and time complexities.**

The state explosion problem, i.e. the number of states generated for analysis increases exponentially with the number of concurrent threads of execution, has been tackled effectively by the apportioning technique in the context of concurrent object-oriented programs [4]. Apportioning reduces the problem of analysing a whole program into a number of smaller problems of individual and independent analysis of portions of the original program. This partitioning is not merely syntactic, but is also an abstraction of the program. However the time complexity of the analysis still remains exponential.

In this paper, a new technique is described that tackles the exponential time complexity of reachability analysis, in the context of Java programs. The state-transition graphs for each thread of execution are generated in parallel and these are subsequently combined to generate the complete state-space from which the reachability graph of the given program is determined. Proofs of safety of the technique and experimental results for the reachability analysis of Java programs are provided.

Raghuraman Rangarajan and Sridhar Iyer are with KReSIT, IIT Bombay, Email:{raghu,sri}@it.iitb.ernet.in
G. Sajith is with the CSE Dept, IIT Guwahati, Email:sajith@iitg.ernet.in

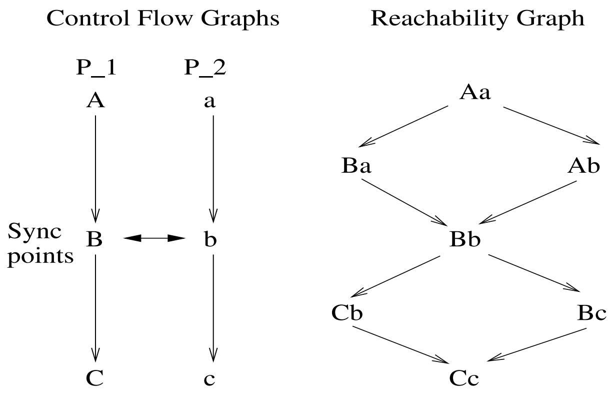

Fig. 1. Traditional reachability graph generation

I. Introduction

REACHABILITY analysis is an important and well-known tool for static analysis of concurrent programs and involves the systematic enumeration of all possible global states of program execution([6], [5]). The control-flow graphs of individual processes are modified to highlight the synchronization structure, abstracting away other details. The complete state-transition graph or the reachability graph is then constructed, thereby modeling the concurrent program as the set of all possible sequences. A simple algorithm for the generation of reachability graphs is described in figure 1.

Reachability analysis facilitates analysis of all possible partial orderings of the concurrent execution. It provides the same level of assurance as formal verification, for properties of the synchronization structure and its practical potential for analyzing non-trivial concurrent programs has been shown in [7]. However, it is well known that reachability analysis suffers from exponential space and time complexities.

As discussed subsequently, the space complexity of reachability analysis is $O(N^T)$, where T is the number of processes and N is the size of the control-flow graph for any given process. The time complexity of the analysis is $O(N^T)^2$. This is not practical for large concurrent programs and hence, strategies for mitigating the complexity of the analysis are necessary. Several techniques such as abstraction [1], partitioning [2] and apportioning [4] have been proposed for combating the state complexity. Although reduction in the number of states of the reachability graph reduces the time complexity, the time taken to analyse programs with large number of threads remains exponential and reachability analysis is still not practical for large programs [3].

In this paper, we describe a new algorithm which effectively reduces the time complexity, i.e. the time taken to generate the reachability graphs. The algorithm first generates an independent state-transition graph of each individual thread in the program. The algorithm then considers the cross-product of these state-transition graphs which is the complete state-space of the program execution. The reachability graph is a sub-graph of this cross-product and is arrived at by discarding unreachable states from this state-space. This is done by taking advantage of the synchronization properties of the system model (in this case, the locking mechanism). From the cross-product, tuples that correspond to multiple threads holding the same lock are discarded and the remaining are analysed with respect to the given error specification.

We outline proofs of safety for the technique and have shown experimentally that the algorithm brings about a significant decrease in the time complexity cost of reachability graph generation. This establishes the practicality of reachability analysis as a method for analysis of concurrent Java programs. Also, the generation of independent state-transition graphs may be distributed over several processors, thereby increasing the practical utility of the technique.

Section II gives the system model considered for this algorithm. The algorithm and an illustrative example are presented in section III. The safety and effectiveness of the algorithm are proved in section IV. The implementation results are given in section V and Section VI presents the conclusions.

1. $\forall$ i ε T : Generate R_i;
   where, T is the number of threads in the program and R_i is the reachability graph of thread i .
2. Consider R_G = all tuples (s_1 , s_2 , ... s_i ... , s_T);
   where, s_i is a state in R_i.
3. for each tuple in R_G do
   if $\exists l$ such that, l is in both x_i and x_j then
   Discard tuple;
   where, l is some lock
   and x_i and x_j are lock statuses of thread i and j respectively.

Fig. 2. Algorithm for parallel generation of reachability graph.

II. System Model

THE technique is applicable for analysis of Java programs having the following salient features :

- Concurrency is supported by having multiple threads of execution.
- All the threads are created at the start of program execution, with each thread being in its respective $entry$ state. This is denoted as the $initial$ state of the program.
- All threads have the same priority and the program terminates when all threads reach their $return$ state.
- The synchronization mechanism between the threads is by using locks. Mutually exclusive access to any data structure d is ensured by having a corresponding lock object $LOCK_d$, which supports two methods $lock$ and $release$.

The model permits concurrent execution of threads in different methods of an object, as well as concurrent execution of multiple threads in the same method. These may give rise to various errors during program execution. Some errors which can be detected by reachability analysis are : (i) method completion before release of acquired locks, (ii) trying to acquire (or release) the same lock twice, (iii) deadlocks due to cyclic waiting for acquiring of locks and cyclical invocations.

A. *Abstract Representation*

In order to generate the abstract representation of a concurrent Java program, $analysis\ points$ are defined to highlight the synchronization structure. The possible analysis points for any method m_x of a class c are :

- m_x_entry: This corresponds to the passive state of m_x, when there is no invocation upon it.
- m_x_begin: This corresponds to the active state of m_x, when it has been invoked and just after control is transferred to it. During program execution, the variable parameters of m_x are bound to actual object identifiers, at this point.
- m_x_return: This corresponds to the return of invocation from m_x, just before control is transferred back to its invoker. Multiple return points are merged to form a single return point.
- m_x_waiting_l_d: This corresponds to a point just before m_x invokes the lock method of l_d, the lock_object for shared data d. During concurrent execution, race conditions may prevail for 'acquiring' l_d. If l_d is already is already 'locked', then m_x waits till l_d is released.
- m_x_released_l_d: This corresponds to a point just after m_x invokes the release method of l_d. Any one of the other methods waiting for l_d may now acquire l_d.
- m_x_before_invocation_o.m_y: This corresponds to a point just before m_x invokes method m_y of the object denoted by o, resulting in a transfer of control to m_y. If m_y is already invoked by some other method, then m_x waits till m_y returns to its entry point.
- m_x_invoked_o.m_y: This corresponds to m_x having invoked m_y and waiting for its invocation to return from m_y.

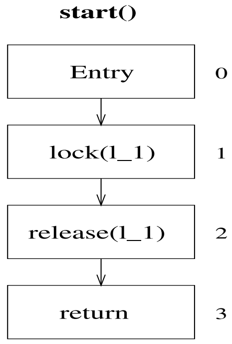

Fig. 3. An abstract example.

- m_x_after_invocation_o.m_y: This corresponds to a point just when control is transferred back to m_x , after the invocation returns from m_y.

III. The Algorithm

A. *Notations*

First a few notations :

- T : Number of threads in a program.
- EP: Entry point of a thread.
- $IS : < EP \ldots EP; OPEN \ldots OPEN >$; Initial state of a program.
- R_T : Reachability graph of a given program.
- R_i : State-transition graph of thread i.
- s_i : A state in R_i (analysis point of thread i).
- l_j : Status of lock j.

B. *Explanation*

The algorithm for parallel generation of reachability graph is described in figure 2. Given a program with T concurrent threads of execution, an initial state (IS), and a specification of the errors to be checked, the technique works as follows :

1. For each thread T_i, a state-transition graph R_i is generated, which corresponds to the execution of T_i while all other threads remain at their respective entry points (EP). The generation of the various R_is is carried out in parallel.
2. The cross product of these R_is (denoted as R_G) is considered. R_G consists of tuples of the form $(s_1, s_2, \ldots, s_i, \ldots, s_n)$ where, s_i is a state in R_i. Note that since R_G is an arbitrary cross product,

it may contain tuples corresponding to multiple threads holding the same lock.
3. Tuples in R_G that correspond to multiple threads holding the same lock are discarded. This is done by analysing the lock status of each s_i in the tuple. Since the tuples are independent of each other, this analysis can also be done in parallel.
4. The tuples in R_G are now analysed with respect to the given error specification.

C. Illustrative example

For the purpose of illustration consider an abstract program with two threads of execution, both having the same control-flow graph as shown in figure 3.

R_G for the given example is now generated as follows :
1. The structure of the state-transition graph for each thread (R_1 and R_2) is represented in table I. Each state in the graph consists of the analysis point in the control-flow graph, the current lock status and the successor state list. Both threads have identical state-transition graph.
2. R_G is a cross-product of R_1 and R_2. A partial R_G is shown in table II.
3. Tuples in which multiple threads hold the same lock are now discarded. Such a tuple is shown in table III. The tuples in R_T are valid states in R_G and are also shown in table III.

IV. Safety and Effectiveness

A. Safety

We have proved the following theorem for the safety of the algorithm. The safety of the algorithm is proved by showing that each state in R_T is present in R_G.

Theorem 1: The algorithm is safe for all programs in the system model.

This is shown as follows :
1. By construction, any analysis point (of a thread i) in a given tuple in R_T exists as a state in the corresponding R_i. That is, for any tuple in R_T : $(s_1, s_2, \ldots, s_i, \ldots, s_n)$, s_i exists in R_i.
2. Now, R_G consists of tuples of the form $(s_1, s_2, \ldots, s_i, \ldots, s_n)$ where, s_i is a state in R_i. Hence, all analysis points in each R_i are present in states in R_G.

From statements one and two it follows that any tuple in R_T also exists in R_G. Hence, the algorithm is safe for all programs in the chosen model.

B. Effectiveness

We show the effectiveness by comparing the reduction in the complexity of the new algorithm as compared to the traditional algorithm.

R_T has a state complexity of $O(N^T)$ where, N is the size of the control-flow graph for any thread and T is the number of threads. This may be seen from the fact that concurrent execution of a thread having p_1 analysis points with a thread having p_2 analysis points in their respectively control-flow graphs, gives rise to $(p_1.p_2)$ states in the reachability graph

In step two of the algorithm, R_G is considered to be a cross-product of R_is, where each R_i is of size N. Hence, R_G is also be bounded by a state complexity of $O(N^T)$.

Now in the worst case, each state in the reachability graph may be connected to every other state. Hence the time complexity, which is proportional to the number of edges is bounded by $O(N^T)^2$.

For calculating the time complexity of R_G, the cost of each step of the algorithm is calculated as follows :
1. Step one of the algorithm, i.e. construction of R_is, takes a time of $O(T.N)$.
2. Step two just considers the cross-product of all the R_is. The calculation of the size of cross-product takes a time of $O(T)$.
3. Step three considers tuples in R_G that correspond to multiple threads holding the same lock. The cost of finding whether multiple threads hold a lock is bounded by $O((T.K))$, where K is the number of locks in the program. Now, each tuple in R_G, which is of size $O(N^T)$, is analysed. Hence, the total cost of step three is $O(T.K.N^T)$.

The cost of the algorithm is then $O(T.N + T + T.K.N^T) \sim O(T.K.N^T)$. Now since each step is independent, it can be executed in parallel, hence reducing the time complexity to $O((T.K.N^T)/P)$, where P is the number of processors available.

TABLE I
STATE-TRANSITION GRAPH OF EACH THREAD (R_i).

Analysis Point	Successors	Lock Status
0	1	0
1	2	1
2	3	0
3	-	0

TABLE II
TUPLES IN PARTIAL R_G.

R_G tuple	Thread 1		Thread 2	
	Analysis Point	Lock Status	Analysis Point	Lock Status
1	0	0	1	1
2	1	1	1	1
3	3	0	3	0

TABLE III
DETECTION OF INVALID TUPLES.

R_G State	Thread 1		Thread 2		Summation of locks	Invalid tuple?
	Analysis Point	Lock Status	Analysis Point	Lock Status		
1	0	0	1	1	1	No
2	1	1	1	1	2	Yes
3	3	0	3	0	0	No

Even excluding the gain due to parallelization, the reduction in complexity consists of exponential terms and hence is significant. In the next section we give experimental results to quantify this reduction.

V. EXPERIMENTAL RESULTS

IN this section we provide the experimental results to compare the efficiency of the parallel algorithm (denoted as R_G) with respect to the traditional algorithm (denoted as R_T). Both the algorithms were implemented using C and C++, on a HP K-class server running HP-UX 11.0. While the generation of R_G may be implemented in parallel, only a sequential implementation was tested. The number of nodes in the reachability graphs, as well as the CPU time taken to generate these nodes are presented.

Table IV gives the comparative results for the well-known producer-consumer problem, while Table V presents the results for the dining philosophers problem. Table VI gives the results for a database application supporting *insert*(), *read*(), *modify*() and *delete*() operations on a sequential database file. Table VII gives the results for a simulation of traffic movement across a bridge such that the traffic flow at any given time is uni-directional.

The *Nodes* entry in each table gives the number of nodes in the reachability graph, while the *Time* entry gives the CPU time taken in hours: minutes: seconds. tenths.

VI. CONCLUSION

WE have presented a new algorithm for effective reachability analysis of concurrent programs. This leads to a significant reduction in time complexity as may be seen from the implementation results for various synchronization problems.

While the sequential implementation of the algorithm itself shows a significant reduction in the time taken to generate reachability graphs, a parallel implementation is likely to lead to an even greater reduction in the time complexity. Also, due to this reduction in time complexity, analysis of programs

TABLE IV

COMPARATIVE RESULTS FOR THE PRODUCER-CONSUMER PROBLEM.

Algorithm	2 prod 1 cons		2 prod 2 cons		3 prod 2 cons		3 prod 3 cons	
	Nodes	Time	Nodes	Time	Nodes	Time	Nodes	Time
R_T	1588	2.0	13679	3:35.5	114239	7:38:43.3	924807	-
R_G		0.150		1.84		34.15		9:10.391
Reduction %		92.5%		99.15%		99.87%		-

TABLE V

COMPARATIVE RESULTS FOR THE DINING PHILOSOPHERS PROBLEM.

Algorithm	2 phils		3 phils		4 phils		5 phils	
	Nodes	Time	Nodes	Time	Nodes	Time	Nodes	Time
R_T	707	0.49	16503	11:07.6	404671	-	9838799	-
R_G		0.066		0.843		31.175		20:15.47
Reduction %		86.53%		99.87%		-		-

TABLE VI

COMPARATIVE RESULTS FOR THE DATABASE APPLICATION.

Algorithm	2 users		3 users		4 users	
	Nodes	Time	Nodes	Time	Nodes	Time
R_T	4863	27.6	19603	5:10:04.8	2673627	-
R_G		0.082		3.040		2:49.964
Reduction %		99.70%		99.98%		-

TABLE VII

COMPARATIVE RESULTS FOR THE TRAFFIC PROBLEM.

Algorithm	2 cars		3 cars		4 cars		5 cars		6 cars	
	Nodes	Time	Nodes	Time	Nodes	Time	Nodes	Time	Nodes	Time
R_T	221	0.1	2444	9.2	26973	41:11.8	52836	-	420189	-
R_G		0.02		0.17		2.84		5.25		58.22
Reduction %		80%		98.15%		99.88%		-		-

with large number of threads is made practical. Development of a more rigorous and formal proof of correctness for the parallel algorithm and a parallel implementation are underway.

REFERENCES

[1] E. M. Clarke, O. Grumberg and D. E. Long. Model checking and abstraction. *Proceedings of ACM Symposium on Principles of Programming Languages*, pp. 343–354, ACM Press, 1992.

[2] O. Grumberg and D. E. Long. Model checking and modular verification. *In Concurr'91, J. C. M. Baeten and J. F. Groote (eds).*, pp. 250–265, published as LNCS-527, Springer-Verlag, 1991.

[3] S. Iyer, Raghuraman R. and A. Majumdar. Apportioning-based reachability analysis for concurrent Java programs. *In Proceedings of International Conference on Information Technology*, Tata-McGraw Hill, Bhubaneshwar, India, 1999.

[4] S. Iyer. Efficient reachability analysis for concurrent object-oriented programs. *Ph.D. Thesis*, Indian Institute of Technology, Bombay, 1997.

[5] C. E. McDowell. A practical algorithm for static analysis of parallel programs. *Journal of Parallel and Distributed Computing*, 515-536, 1989.

[6] R. N. Taylor. A general purpose algorithm for analyzing concurrent programs. *Communications of the ACM*, 26(5):362–376, May 1983.

[7] M. Young, R. N. Taylor, D. L. Levine, K. A. Nies and D. Broadbeck. A concurrency analysis tool suite for Ada programs. *ACM Transactions on Software Engineering and Methodology*, 4(1):65–106, Jan 1995.