Sequential and Parallel Reachability Analysis of Concurrent Java Programs

A Thesis Submitted

in Partial Fulfillment of the Requirements

for the Degree of

MASTER OF TECHNOLOGY

by

Raghuraman Rangarajan

to the

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

INDIAN INSTITUTE OF TECHNOLOGY, GUWAHATI

MARCH, 2000

Abstract

Concurrent programs are more difficult to test than sequential programs because of their non-deterministic behaviour. Reachability analysis is an important and well known tool for static analysis of concurrent programs. It involves the systematic enumeration of all possible global states of program execution. However, traditional algorithms to generate all reachable states of a concurrent program take exponential time and space.

Apportioning is a technique which is based on the idea of classification of program analysis points as *local* (having influence within a class) and *global* (having influence outside a class). Apportioning uses this classification to abstract away some analysis points, thus reducing the size of the reachability graph generated.

This report presents two algorithms for the generation of the reachability graph of a concurrent Java program. The algorithms are implemented for some of the apportioning-based reachability analysis tools. The results generated are used to verify the efficiency of apportioning as a tool for reachability analysis of concurrent Java programs. The first algorithm is a sequential implementation of the apportioning technique, while the second generates the reachability graph in parallel. While the sequential algorithm is used to demonstrate the reduction in the exponential space complexity of a reachability graph, the second algorithm attempts to mitigate the time complexity.

Contents

1 Introduction **1**

2 Literature Review **4**

2.1 Overview 4
2.2 Work done by Richard Taylor 4
2.3 Work done by Charles McDowell 7
2.4 Work done by Sridhar Iyer 8

3 Reachability Analysis of Concurrent Java Programs **9**

3.1 Overview 9
3.2 Analysis points 10
3.3 Algorithm for reachability graph generation 11
3.4 The apportioning technique 13
3.5 Advantages of apportioning 15
3.6 Some apportioning-based tools 16
3.6.1 OMEGA 16
3.6.2 ALPHA 16
3.6.3 BETA 17
3.7 Illustrative example 18
3.8 Error detection 21

4 Experiments Using Apportioning **22**

4.1 Overview 22
4.2 Concurrent Java programs 23
4.3 Implementation issues 23
4.4 Input and output formats 24
4.5 Module structure 27

4.6 Experimental results 28
4.6.1 Producer-consumer problem 28
4.6.2 Dining-philosopher problem 28
4.6.3 Database application 30
4.6.4 Traffic problem 32
4.7 Discussion 34

5 Reachability Graph Generation In Parallel 35
5.1 Introduction 35
5.2 Algorithm 36
5.3 Explanation 37
5.4 Example: producer-consumer problem 42
5.5 Experimental results 47
5.5.1 Producer-consumer problem 47
5.5.2 Dining-philosopher problem 47
5.5.3 Database application 49
5.5.4 Traffic problem 50
5.6 Discussion 50

6 Conclusion and Future Work 51

List of Tables

4.1 Experimental results for the producer-consumer problem. 28
4.2 Experimental results for the dining philosophers problem. 30
4.3 Experimental results for the database application. 31
4.4 Experimental results for the traffic simulation. 32

5.1 Tabularized form of the CFG of a producer/consumer. . 43
5.2 Reachability graph of a single thread of producer. 43
5.3 Reachability graph of a single thread of consumer. 44
5.4 Timing comparison for the producer-consumer problem (ALPHA). 47
5.5 Timing comparison for the producer-consumer problem (OMEGA). 48
5.6 Experimental results for the producer-consumer problem. 48
5.7 Timing comparison for the dining philosophers problem (ALPHA). 48
5.8 Timing comparison for the dining philosophers problem (OMEGA). 48
5.9 Experimental results for the dining philosophers problem. 49
5.10 Timing comparison for the database application (ALPHA). 49
5.11 Timing comparison for the database application (OMEGA). 49
5.12 Experimental results for the database application. 49
5.13 Timing comparison for the traffic problem (ALPHA). . . 50
5.14 Timing comparison for the traffic problem (OMEGA). . . 50
5.15 Experimental results for the traffic simulation. 50

List of Figures

2.1 Apportioning technique. 5
2.2 Taylor's algorithm. 6

3.1 Reachability graph generation. 11
3.2 Apportioning technique. 15
3.3 CFGs for the producer-consumer problem. 19
3.4 Illustration of ALPHA on the producer-consumer problem. 20

4.1 Input flowgraph format. 25
4.2 Reachability graph output format. 26
4.3 CFGs for the dining philosopher problem. 29
4.4 CFGs for the database problem. 31
4.5 CFGs for the traffic problem. 33

5.1 A barrier between two threads. 39
5.2 A deadlock condition. 40
5.3 CFG for the producer-consumer problem (with ids). 42
5.4 Representation of a reachability graph node. 44
5.5 Node generation process. 45
5.6 Error due to a deadlock. 46

Chapter 1

Introduction

Object-oriented programming is an approach that provides a way of modularizing programs by creating partitioned memory area for both data and functions that can be used as templates for creating copies of such modules on demand [3].

Object-oriented programming allows programmers to create modular, flexible programs with possibility of code reuse. Various features of object-oriented programming like abstraction (representation of essential features without including the details), encapsulation (the wrapping of data and methods into a single unit), inheritance (process by which objects of a class acquire the properties of another class) and polymorphism (ability to take more than one form), help in designing large and complex programs.

A concurrent program specifies two or more processes that execute in parallel and co-operate in performing a task [1]. Each process is a sequential program that executes a sequence of statements. Processes co-operate by communicating; they communicate using shared variables or message passing. When shared variables are used, one process writes into a variable that is read by another. When message passing is used, one process sends a message that is received by the other.

Concurrent object-oriented programs combine both the features of concurrency and the object-oriented paradigm.

Java is an object-oriented, multi-threaded, secure, platform independent language. Java programs can be run in both interpreted and

compiled form. The portable nature of a Java program allows it to be deployed across networks. Java provisions such as automatic garbage collection and elimination of pointers makes it an easy and fast language to learn. The object-oriented nature of Java allows programmers to develop code that can be easily reused in other applications, while the concurrent nature allows the development of multi-threaded programs which can be executed in parallel. These threads can access the same global data within a class and modify it.

Designers of such software programs are faced with a number of difficult verification problems like a program entering an infinite wait state or undesirable parallelism in the program. Hence the need for all the classical synchronization primitives in a concurrent Java program – locks, mutual exclusion and avoidance of deadlocks. Debugging a concurrent Java program involves the detection of any synchronization anomalies such as two or more processes updating the same variable (i.e. violation of mutual exclusion), and two or more processes waiting to acquire some resource forever (i.e. occurrence of deadlock).

Analysis of concurrent programs maybe be classified into *dynamic analysis* and *static analysis*. Dynamic analysis involve *run-time* monitoring of the system, for re-creating the program execution. However, monitoring introduces extraneous delays, making it difficult to recapture the erroneous behaviour. Static analysis technique avoids such problems since it does not require program execution. However, dynamic data and control-flow of a program cannot be handled by static analysis techniques.

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. 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. However, it suffers from "state-explosion", i.e., the number of states generated for analysis increases with the number of concurrent threads of execution.

A technique which tackles this drawback effectively is *Apportioning*.

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.

Partitioning of the program into its constituent classes is done by taking advantage of the modular structure of the object-oriented program. The abstraction itself is based on the idea of classification of synchronization points. The synchronization points can be classified as local (having influence within a class) and global (having possible influence without a class).

This thesis attempts to experimentally verify the usefulness and the efficiency of apportioning as a technique for the reachability analysis of concurrent Java programs. We do this by first experimentally comparing an apportioning-based tool with the traditional reachability graph generation method, using a sequential algorithm. We then present a parallel algorithm, which attempts to reduce the exponential time complexity associated with the problem and also present experimental results for the same.

The second chapter of the thesis describes some traditional methods of reachability analysis, while the third chapter explains the apportioning technique, which mitigates the problems associated with the traditional reachability graph generation methods. The apportioning technique is tested using various examples. The test results are presented in the fourth chapter. The fifth chapter presents a method for generating the nodes in the reachability graph of a concurrent Java program in parallel. Experimental results for different examples are also presented. The sixth chapter concludes the thesis.

Chapter 2

Literature Review

2.1 Overview

In this chapter, we review the various work done on reachability analysis of concurrent programs. We first discuss the work of Taylor, who proposed the *concurrency history* technique, based on the enumeration of all possible paths, for analysis of programs having task synchronization.

Next we discuss the work done by McDowell, who gives an algorithm for static analysis of parallel programs. Both the above mentioned techniques can be classified as traditional reachability analysis techniques. Both the techniques have exponential space and time complexity. We briefly describe a technique given by Iyer to reduce the exponential space required for reachability analysis. We discuss this technique in detail in the next chapter.

2.2 Work done by Richard Taylor

Richard N. Taylor gave a general purpose algorithm for analyzing concurrent programs in [12]. The algorithm addressed problems such as : how processes are synchronised, what determines when programs run in parallel and how errors are detected in the synchronization structure.

The algorithm generates the complete concurrency history for a pro-

gram. In doing so, all possible task synchronization points are determined, all possible infinite waits are decided.

The algorithm in essence simulates the synchronization aspects of program behaviour. The algorithm does not make any assumptions regarding the execution environment of the program that would influence the computation of the concurrency states. That is, it does not take into consideration, the number of processors, the relative speeds of the processors, and processor time-slicing schemes.

The complexity of the analysis $O(N^T)$, where T is the number of processes and N is the size of the control-flow graph for any given process. For large concurrent programs, the method will become impractical because of its high complexity and hence, strategies for mitigating the analysis is called for. Figure 2.1 gives a representation of the reachability graph generation.

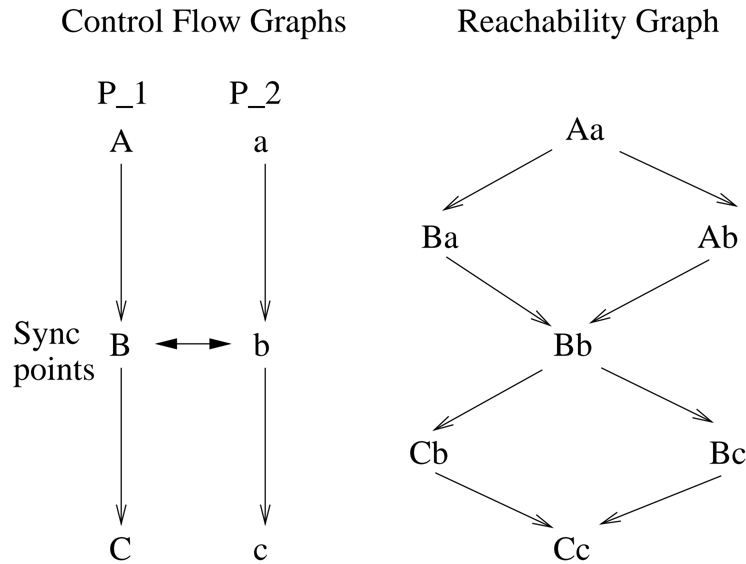


Figure 2.1: Apportioning technique.

A brief description of the algorithm is given in figure 2.2. The algorithm uses a breadth-first approach to generate all the states attainable by the program. The algorithm starts with the initial state of the program, and systematically enumerates all possible successors.

```
procedure Simulate;
    Worklist :=null;
    /* Push initial state on list and enter initial state in Unique State Table (UST) */

    while Worklist not empty loop
        q := first element on Worklist;

        /* States on the worklist are unexamined elements of the UST */

        for i in 1 . . T loop
            /* Check current node in task i to se if it is possible to generate
             * a successor state on the basis of task i's performing a rendezvous,
             * task activation, or the like
             */
            if moveable (i) then
                Create a new state z , reflecting task i's movement, as well as other
                tasks as required ( such as in the case of a rendezvous occuring);

                /* z is checked against the UST */
                if unique (z) then
                    Enter z in UST;
                    Push z onto Worklist;
                else
                    enter z in duplicate state table;
                end if;
        end loop;
    end loop;
end Simulate;
```

Figure 2.2: Taylor's algorithm.

The algorithm is initialized by placing the initial state on the Worklist and entering it as the first entry in the unique state table. During the subsequent execution, a state is taken from the worklist and examined in order to determine all possible successor states. Each successor state is checked against the UST. If it is a new state it is entered in the UST and then placed on the Worklist itself. If the state duplicates a state already in the UST, it is simply entered in the duplicate state table and is not examined any further. Thus the Worklist only contains states whose successors, if any, have never been determined. When a state is pulled off the worklist and examination determines that it has no successors, it is then entered into the terminal state table.

Taylor also employs a partitioning approach for *parceling* the analysis into biconnected components. Such an approach is applicable to programs composed of disjoint set of tasks and brings about a reduction in the size of the reachability graph generated.

2.3 Work done by Charles McDowell

Charles McDowell has given an algorithm for static analysis of parallel programs in [9]. This requires the determination of the reachable program states. A program state is a set of task states, the values of shared variables used for synchronization, and local variables that derive the values directly from synchronization operations.

To reduce the number of reachable states his algorithm merges a set of related states into a single *virtual state*. Using this approach, the analysis of concurrent programs becomes feasible as the number of virtual states is often orders of magnitude less than the number of reachable states.

The paper describes the reduction of the control flow graph of a concurrent program to the synchronization graph where only points of synchronization are represented as nodes. From this graph a *Canonical Concurrency History Graph* (CHG) is generated. In this graph, each node describes one concurrent reachable state of the program. Each edge corresponds to one task changing state.

McDowell uses abstraction as a method for *folding* many states in a reachability graph into a single state and hence is applicable to pro-

grams composed of identical sets of tasks. Such an abstraction results in a reduction in the size of the reachability graph generated.

2.4 Work done by Sridhar Iyer

Sridhar Iyer presents in [5], a technique called apportioning for the reachability analysis of concurrent object-oriented programs. Apportioning integrates two techniques for combating state-explosion, viz., *partitioning* and *abstraction*.

Partitioning decomposes the program into smaller components, so that analysis of the large program is replaced by the individual analysis of the smaller components. Abstraction hides out some pre-defined details of the program and hence reduces the size.

Apportioning effectively tackles the exponential problem, by classification of program analysis points. It is based on the idea of classification of program analysis points as local analysis points (LAP) and global analysis points (GAP). A LAP is said to have influence within a class while a GAP is said to have possible influence without a class.

The apportioning technique is explained in detail in the next chapter.

Chapter 3

Reachability Analysis of Concurrent Java Programs

3.1 Overview

A straightforward application of Taylor's algorithm for static analysis is unacceptable, because the amount of data that must be maintained is $O(N^T)$, where T is the number of tasks in a concurrent program. In case of Java, it is the number of threads created. N is a measure of the size of the tasks. Hence, modification to the algorithm is suggested in order to reduce the size of the reachability graph generated.

In this chapter we define analysis points in a program, which are used to generate the abstract representation which retains only the essential features of a concurrent object-oriented program. An algorithm for generating the reachability graph is then presented. This algorithm takes the control-flow graphs of a program as input and generates the corresponding reachability graph.

Subsequently, we view the apportioning technique which mitigates the exponential complexity of the traditional reachability analysis method and give an example to illustrate it.

3.2 Analysis points

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. This can be an LAP as well as a GAP.

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. This can be an LAP as well as a GAP.

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. This can be both a GAP and an LAP.

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. This can be an LAP only.

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. This can be an LAP only.

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. This can be an LAP or a GAP depending on the object o.

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. This can be an LAP or a GAP depending on the object o.

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. This can be an LAP or a GAP depending on the object o.

3.3 Algorithm for reachability graph generation

The following algorithm ([5]) generates the reachability graph of a program. It is called R_gen and takes 3 inputs : the set of class control-flow graphs, the set of objects (class instances) and the initial state of the system.

An abstract representation of R_gen is as shown in figure 3.1. C_i, M and RG stand for the control flowgraph of class i, the control flowgraph of the main program and the reachability graph generated, respectively.

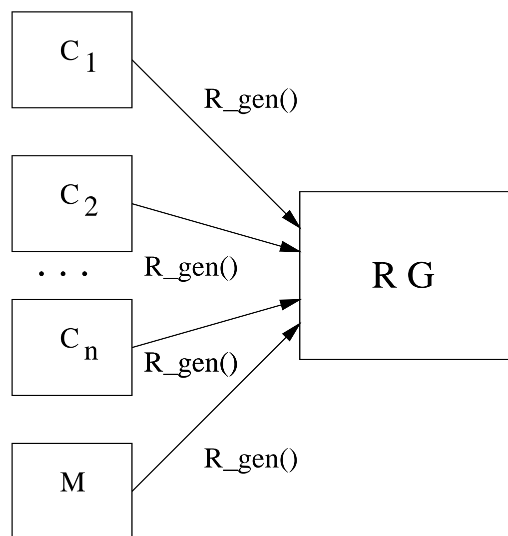


Figure 3.1: Reachability graph generation.

The algorithm is given below :

```
R_gen(C,O,I)
C : Set of class control-flow graphs
O : Set of objects
I : Initial state
{
  node_set N_r; /* Nodes in the reachability graph */
  edge_set E_r; /* Edges in the reachability graph */
  node current; /* Reachability graph node */
  node_set next; /* Successors of node current */
  node_set list; /* Nodes whose successors are yet to be determined
  */
  N_r = list = {I}; /* Initial state */
  repeat
    current = list.element[1]; /* Take the first node */
    list = list - current; /* Remove it from list */
    i = 1; /* Begin breadth-first generation */
    repeat /* For each field (method) in node current */
      next = find_successors(current, current.field[i]);
      repeat /* For each node in next */
        If (next.element[1] ∉ N_r i) then {
          N_r = N_r ∪ {next.element[1]};
          list = list ∪ {next} - {current};
        }
        /* Introduce appropriate edges into E_r */
        next = next - {next.element[1]}; /* remove the successor */
      until (elements(next) = 0); /* For all successors */
      i = i + 1;
    until (i = N); /* N is the number of methods in a node */
  until (elements(list) = 0); /* For all nodes */
}
```

The function *find_successors()* takes the current node as input and generates all its successors. The successors are generated with respect

to the following rules :

- If the current node is a m_x*_waiting_*l_d node and if the lock is *open*, then acquire the lock and generate the successor node.
- If the current node is a m_x*_released_*l_d node and if the lock is *closed*, then release the lock and generate the successor node.
- If the node is of any other type, then generate the successors.

3.4 The apportioning technique

Apportioning is based on the idea of classification of synchronization points. A synchronization point in any method can be classified either as *local analysis point (LAP)* that correspond to an interaction with another method of the same object, or *global analysis point (GAP)* that corresponds to an interaction with a method of a different object, or both.

Apportioning reduces the problem of analyzing 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. Partitioning of the program into its constituent classes is done by taking advantage of the modular structure of the concurrent object-oriented program.

Any apportioning-based reachability analysis algorithm proceeds as follows :

(a) Starting from an abstract representation of the program along with a classification of its analysis points, (i) a set of reduced classes each corresponding to a class in the program, with the GAP being abstracted out and (ii) a reduced version of the entire program, with the LAP being abstracted out, are generated.

(b) The reachability graphs for each of these reduced classes and the reduced program are generated.

(c) The error to be checked (expressed as a property of analysis points), is decomposed into a set of sub-properties, some having only LAP

(corresponding to the reduced classes) and another having only GAP (corresponding to the reduced program).

(d) Each reachability graph is analyzed for the corresponding sub-property.

A representation of apportioning is given in figure 3.2. The notations used in the figure are :

1. $GH()$: method to perform GAP hiding.
2. $LH()$: method to perform LAP hiding.
3. RLG: method to generate the local reachability graph of a class.
4. $RGG()$: method to generate the global reachability graph of a program.
5. C_i: control flowgraph of class i.
6. M: control flowgraph of the main program.
7. C_i': reduced control flow graph after GAP hiding.
8. C_i'': reduced control flow graph after LAP hiding.
9. $R_{C_i'}$: local reachability graph of class C_i.
10. R'': global reachability graph of program.

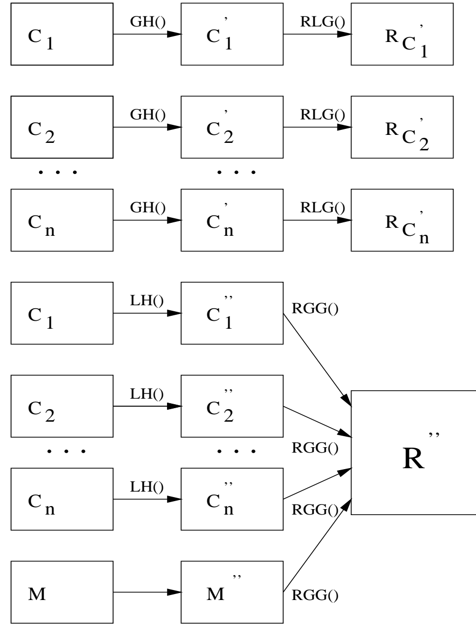


Figure 3.2: Apportioning technique.

3.5 Advantages of apportioning

The practical utility of the apportioning technique can be seen from the following observations :

The complexity (number of states generated for analysis) of traditional reachability analysis([12]), is $O(p^T)$, where T is the number of threads and p is the number of interactions for any thread. Extending such techniques to concurrent object-oriented programs by performing additional analysis for each class, would result in a complexity of $O(c(p_l)^m + (p)^T)$, where c is the number of classes, m the number of methods in each class, and p_l is the number of LAP in any method.

Apportioning mitigates this complexity to $O(cp_l^m + p_g^T)$, where p_g is the number of GAP in any method. Typically for many programs $p \approx (p_l + p_g)$. Although the complexity of the apportioned analysis is exponential, the amount of reduction is also composed of exponential terms, and is quite significant.

3.6 Some apportioning-based tools

Apportioning is a general technique which can be used to develop different algorithms, based on an appropriate classification for the analysis points. Classifying the analysis points differently before apportioning, gives rise to tools that are safe for different sub-classes of programs. Three such tools, OMEGA, ALPHA and BETA are discussed here.

3.6.1 OMEGA

OMEGA classifies the analysis points corresponding to global invocations, i.e., analysis points of the form m_x*_before_invocation_o.*m_y, m_x*_invoked_o.*m_y and m_x*_after_invocation_o.*m_y, as *GAP*.

All other analysis points, i.e., analysis points of the form m_x*_waiting_*l_d, m_x*_released_*l_d, m_x*_before_invocation_self.*m_y, m_x*_invoked_self.*m_y and m_x*_after_invocation_self.*m_y, belong to both *LAP* and *GAP*.

Thus, OMEGA is the representation of the traditional reachability analysis method. OMEGA can be said to be safe for all kinds of programs, while it is the least efficient of all the apportioning-based algorithms developed, since no abstraction is performed.

3.6.2 ALPHA

ALPHA classifies the analysis points in any method m_x as follows :

LAP: Analysis points of the form m_x*_waiting_*l_d, m_x*_released_*l_d, and analysis points m_x*_before_invocation_self.*m_y, m_x*_invoked_self.*m_y and m_x*_after_invocation_self.*m_y, belong to *LAP*.

GAP: Analysis points of the form m_x*_before_invocation_o.*m_y, m_x*_invoked_o.*m_y and m_x*_after_invocation_o.*m_y, where o is a variable object identifier, belong to *GAP*.

The analysis points m_x*_begin* and m_x*_return*, belong to both *LAP* and *GAP*.

ALPHA does not consider the analysis points corresponding to local invocations of a class as effecting the global outlook of the program and hence classifies them as *LAP*. Only the analysis points corresponding to global invocations get reflected in the global reachability graph.

Hence, ALPHA can be said to be safe for all programs in which a method does not invoke another method of the same object. This is because, all local invocations are classified as *LAP*, and so any cyclic invocation involving such invocations won't be caught. ALPHA can detect errors such as access to shared data without acquiring locks, method completion before release of locks and deadlocks.

The number of nodes ALPHA generates will generally be less than that of OMEGA, since OMEGA performs no apportioning. In the worst case, there might be no local invocations or shared memory access in the program, then ALPHA generates the same number of nodes as that of OMEGA.

3.6.3 BETA

BETA relaxes the condition given for ALPHA, that a method cannot invoke another method of the same object. It classifies analysis points as follows :

LAP: Analysis points of the form m_x*_waiting_*l_d and m_x*_released_*l_d, belog to *LAP*.

GAP: The analysis points of the form m_x*_before_invocation_o.*m_y, m_x*_invoked_o.*m_y and m_x*_after_invocation_o.*m_y, where o is an object identifier, belong to *GAP*.

The analysis points m_x*_begin*, m_x*_return*, m_x*_before_invocation_o.*m_y, m_x*_invoked_o.*m_y and m_x*_after_invocation_o.*m_y, belong to both *LAP* and *GAP*.

BETA relaxes the condition given in ALPHA, and allows the reflection of local invocations in the global reachability graph. Hence, other than the errors detected by ALPHA, BETA can also detect cyclic wait-

ing for return of local invocations and cyclic waiting involving a mixture of local and global invocations.

In general, BETA will generate more number of nodes than ALPHA. In its best case, it will generate the same number of nodes as ALPHA, i.e., if there are no local invocations present. In its worst case, BETA will generate the same number of nodes as OMEGA, considering the presence of no shared memory access.

3.7 Illustrative example

We illustrate the details of the ALPHA tool using a simple variation of the producer-consumer problem. The control flow graph of the producer-consumer problem is given in figure 3.3.

There are three classes in the producer-consumer problem. A producer class, a consumer class and a buffer class. The buffer class contains a bounded buffer and shared variables *l_first* and *l_last* to keep track of the bounds. The buffer class also has two methods, namely, *put()* and *get()* which can be invoked to perform their respective operations.

The producer class contains the method *produce()* which in turn invokes the *put()* method of the buffer object to put data into the buffer. While the consumer class defines a method *consume()* which in turn invokes the *get()* method of the buffer class.

The producer object invokes the *put()* method of the buffer object to place an item, while the consumer object invokes the *get()* method of the buffer to remove an item. The buffer is bounded and shared variables l_first and l_last are used to keep track of the items. The illustration of the ALPHA tool on the producer-consumer problem is given in figure 3.4.

Concurrent execution may give rise to the following errors : access of buffer data without proper *locks*, cyclic waiting among methods *put()* and *get()* for acquisition of *locks*, and termination of consumers before that of producers.

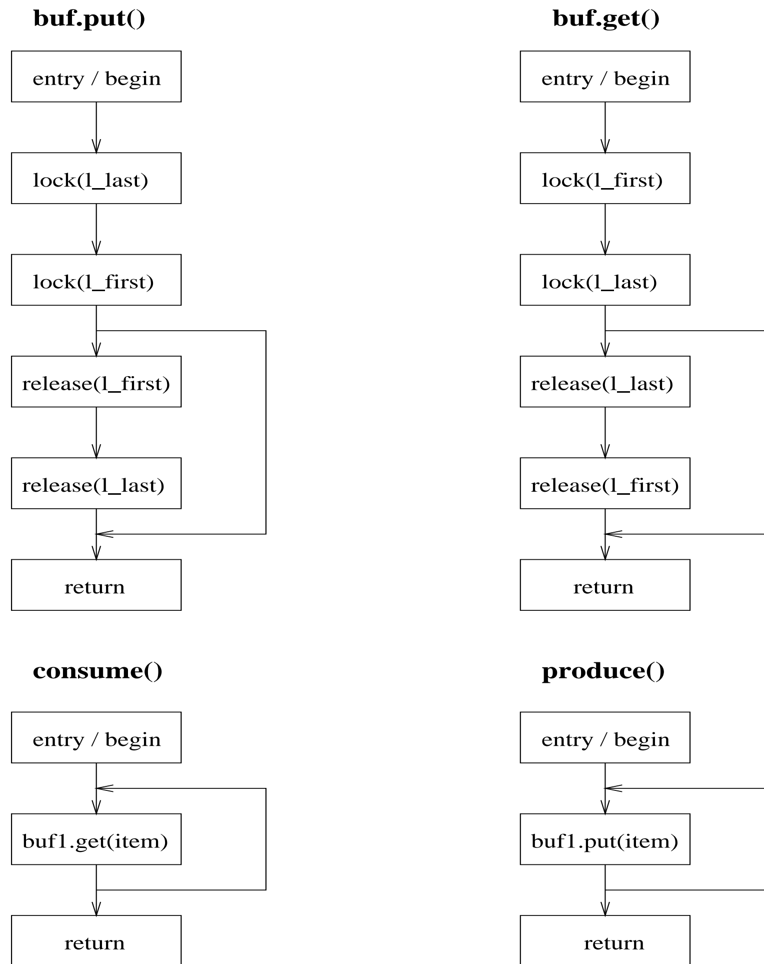


Figure 3.3: CFGs for the producer-consumer problem.

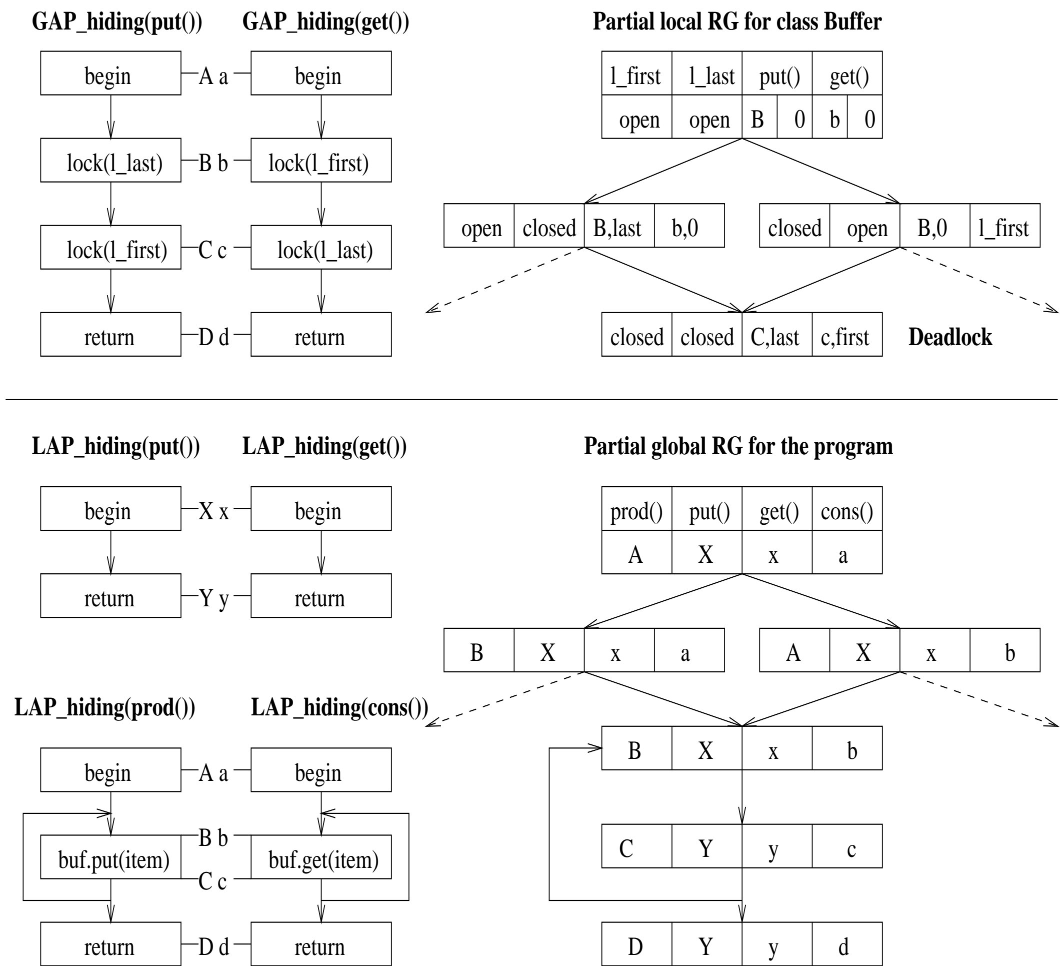


Figure 3.4: Illustration of ALPHA on the producer-consumer problem.

3.8 Error detection

A concurrent object-oriented program may exhibit a variety of errors during execution, such as deadlocks, improper access of shared data, overflow, livelocks, etc. In the context of reachability analysis a program is analyzed for the existence of a certain class of errors, viz., errors in the synchronization structure such as deadlocks. Analysis for errors such as those that depend on dynamic data values (overflow), livelocks, are beyond the scope of reachability analysis.

The implementation of this project focusses on errors that may be captured as properties of the global state of the program execution. Many errors such as improper access of shared data, deadlocks and incorrect termination belong to this category.

A synchronization anomaly exists if any of the reachability graphs of the program contains a concurrency state that has no successor and that state is not the terminated state. A parallel access anomaly exists if in any reachability graph there exists a state with at least two tasks such that both are accessing a shared data and at least one is attempting to modify it.

Chapter 4

Experiments Using Apportioning

4.1 Overview

We present in this chapter the implementation results of some reachability graph generation tools [1]. The results presented are the comparative analysis of one of the apportioning tools (ALPHA) with the traditional reachability analysis method (OMEGA). The implementation is an extension of the attempt made in [8]. The implementation was debugged, modified to allow program control flowgraphs with loops as input and then extended for the analysis of the different apportioning tools. Test results were then generated for different examples.

The program takes the input file name and the output file name as the command line arguments. The input file contains the control flowgraphs of the input program. The reachability graphs that are generated for each class and the global reachability graph are stored in the output file specified.

We first take a look at the support Java has for concurrent programming. We then discuss the various implementation issues of the program. The formats of the input control flowgraph and the output reachability graphs are then given. The structure of the implemented

[1] A concise version of this chapter has been published in [6].

program is then presented and finally, we present the implementation results for various examples, along with a discussion based on the results.

4.2 Concurrent Java programs

We discuss here a few concepts about the support Java has for concurrent programming.

Java supports concurrent programming with the help of threads. Java supports multithreading, i.e., multiple flows of control in a single program. A new thread can be created in Java in following ways :

- By creating a *Thread* class : Define a class that extends the *Thread* class and override its *run()* method with the code required by the thread.
- By converting a class to a thread : Define a class that implements the *Runnable* interface. The *Runnable* interface has only one method, *run()*, that is to be defined in the method with the code to be executed by the thread.

The approach to be taken depends on the whether the class is extending an other class or not. If the class requires extending an other class, then we have to implement the *Runnable* interface, since Java doesn't allow multiple inheritance.

Mutual exclusion in Java is provided by the use of the keyword *synchronized*. A method in a class itself can be declared as *synchronized* or a *synchronized* statement can be used to access the object in a thread-safe manner.

4.3 Implementation issues

The program implements the tools ALPHA, BETA and OMEGA using C++, for the analysis of Java programs having the following features :

- The program consists of a set of class definitions and a set of object declarations. All objects are created and initialized at

the start of program execution. The number of objects is fixed thereafter.

- Concurrency is supported by having multiple threads of execution. Creation of a new thread class is by declaring the new class as a subclass of the *Thread* class; the *Runnable* interface is not currently supported. All threads are created at the start of program execution, with each thread being in its *entry* state; the entry point of a thread is the *entry* of its *run()* method. All threads have the same priority and the program terminates when all threads reach their *return* state.

- The execution model is such that a thread enters an object when any of its methods are invoked and returns when the method is completed. Invocations are synchronous and the invoking method is blocked until the return of invocation.

- Threads communicate by using shared data within an object; a thread *waiting* and joining other threads is not currently supported. Synchronization for mutually exclusive access to shared data is supported by use of the keyword *Synchronized*.

4.4 Input and output formats

The flowgraphs of a program form the input to the reachability graph generator. The various information presented in the flowgraph are as follows :

1. Number of classes, their id's and objects.

2. Depiction of a class and its methods.

3. Details of each node in the flowgraph.

The format of the flowgraph input is given in figure 4.1.

Node_Id is a unique number that is used to distinguish nodes of a procedure. The *Type* of the node is one of the following : m_x*_begin*, m_x*_waiting_l*$_d$, m_x*_released_l*$_d$, m_x*function_call.*m_y and m_x*_return*.

1. Number of classes, their id's and objects

int Number_Of_Classes
Class_Name_1 Thread/Not_A_Thread   Number_Of_Objects   obj_1 obj_2 ... obj_n
Class_Name_2 Thread/Not_A_Thread   Number_Of_Objects   obj_1 obj_2 ... obj_n
.
.
.
Class_Name_n Thread/Not_A_Thread   Number_Of_Objects   obj_1 obj_2 ... obj_n

2. Depiction of a class and its methods

Class_Name_i
    Number_Of_Locks Lock_1 Lock_2 . . . Lock_n
    Number_Of_Methods Method_1 Method_2 . . . Method_n
    Method_1
        Node_1
        Node-2
        .
        .
        .
        Node_n
    Method_2
    .
    .
    .
    Method_n

3. Structure of a node

int Node_Id
int Type
union {
    char* LockName
    char* FunctionName
}
int Successors []

Figure 4.1: Input flowgraph format.

The various information needed for presenting the output reachability graphs are as follows :

1. The number of local reachability graphs.

2. Depiction of a reachability graph.

3. Details of all the nodes in the local and the global reachability graphs.

The format of both the local and the global reachability graph is the same and is as shown in figure 4.2. The array *Position* denotes the position of the threads (in their flowgraphs) from which the current node in the reachability graph had been created. The array *LockStatus* shows the status of the locks at that given node.

1. Number of local reachability graphs

int Number_Of_Graphs

2. Depiction of a reachability graph

Class_Id class_id
Node_1
Node_2
.
.
.
Node_n
End of reachability graph

3. Structure of a node

int Node_Id
int Position [Number_Of_Threads]
int LockStatus [Number_Of_Locks]

Figure 4.2: Reachability graph output format.

4.5 Module structure

A brief description of the module structure is as follows. The program first gets the input control flow graphs for all the classes, generates the local reachability graphs for each class and then generates the global reachability graph for the program. It then prints out the node details of all the graphs generated. A high-level view of the modules implemented for performing the operations described in section 3.4 (figure 3.2) is as follows :

- *Main*() invokes *GetInput*() for reading the given program control-flow graph. *GetInput*() iteratively calls *GetClass*() to get the details of each class C_i, such as the class name, objects, locks and details of the procedures in C_i.

  These details are got using the functions *GetName()*, *GetObjectNames()*,*GetLockNames()*, *GetProcedureNames()* and *GetNodesOfProcedure()*. The *GetNodesOfProcedure()* iteratively gets all the node details of a procedure, like identity of a node, type of the node, analysis point type, information about the node and the node's successors.

- *Main*() then invokes *GenLRGraph*() to generate the local reachability graphs for each class C_i. *GenLRGraph()* first abstracts away the global analysis points (*GAP*) using the function *HideGAP()* and then calls the function *GenerateGraph()* to generate the reachability graph in a breadth first manner.

  GenerateGraph() uses *FindSuccessors* to identify all possible successors of a given reachability graph node. *FindSuccessors* follows the rules explained in section 3.3. It uses a queue to collect the successors found. At each iteration, an element from the queue is taken and its successors generated.

- Subsequently, *Main*() invokes *GenGRGraph*() to generate the global reachability graph for the given program. *GenGRGraph*() implements the operations *LAP_hiding*() and *GenerateGraph*(), in a manner similar to that of *GenLRGraph*(), described above.

- Finally, *Main*() invokes *PrintGraph*() to output the details of the reachability graphs generated, such as the total number of nodes in each graph and the details of each node according to the format given in section 4.4.

4.6 Experimental results

In order to show the efficiency of ALPHA v/s OMEGA, we compare the number of nodes in the reachability graphs generated by them, as well as the CPU time taken to generate these nodes, on a HP K-class server running HP-UX 11.0.

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

4.6.1 Producer-consumer problem

Table 4.1 gives the comparative results of ALPHA and OMEGA, for the producer-consumer problem described in section 3.7.

Tool	1 prod 1 cons		2 prod 1 cons		2 prod 2 cons		3 prod 2 cons		3 prod 3 cons	
	Nodes	Time	Nodes	Time	Nodes	Time	Nodes	Time	Nodes	Time
OMEGA	203	0.1	1588	2.0	13679	3:35.5	114239	7:38:43.3	-	-
ALPHA	75	0.1	255	0.2	1335	3.6	7815	3:27.5	46595	3:50:27.8
Reduction %	63.05%		83.94%		90.24%		93.16%		-	

Table 4.1: Experimental results for the producer-consumer problem.

4.6.2 Dining-philosopher problem

The dining-philosopher problem illustrates contention for resources in a concurrent program. Briefly, there are n philosophers seated for dinner, with one fork between every adjacent pair of philosophers. The philosophers alternatively think and eat. In order to eat, each philosopher picks up two forks, first the one to the right and then the one to the left. The forks are subsequently released.

A representation of the problem is given in figure 4.3
and its description is as follows :

- A *fork* class provides the methods *up()* and *down()* for taking up and putting down a fork. On invocation, *up()* waits until the fork is free and then assigns it to the invoking philosopher. The method *down()* releases the fork taken by the philosopher. The shared variables *f_sts* and *f_own*, record the fork status and the identity of the owning philosopher, respectively. The *lock l_sts* is used to ensure mutual exclusion to the above mentioned variables.

- A *philosopher* class defines *l_fork* and *r_fork* for recording identities of the fork objects. The method *work()*, thinks and eats

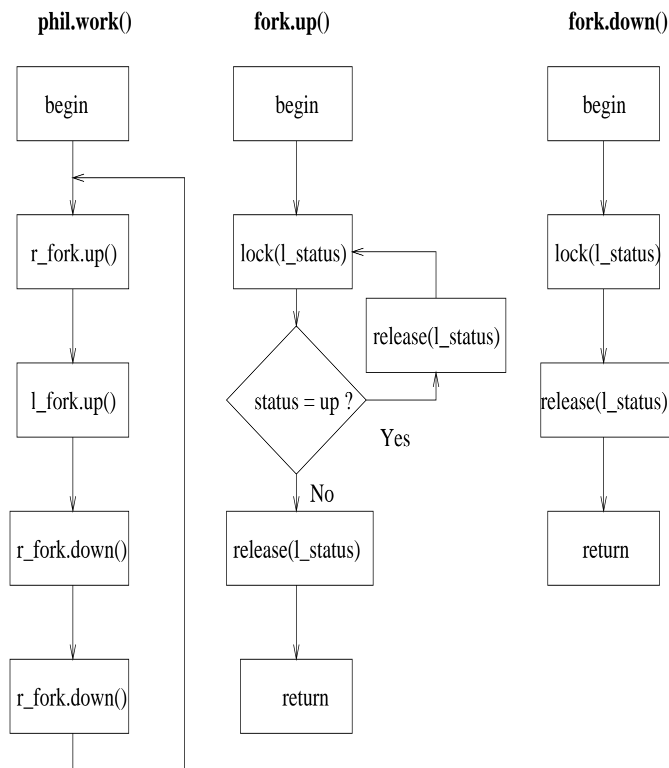


Figure 4.3: CFGs for the dining philosopher problem.

iterativelyand it invokes the *up()* and *down()* methods of the *fork* object to take up and put down the forks.

Table 4.2 presents the results for the dining philosophers problem.

Tool	2 phils		3 phils	
	Nodes	Time	Nodes	Time
OMEGA	707	0.49	16503	11:07.6
ALPHA	371	0.1	5879	1:12.1
Reduction %	47.52%		64.37%	

Table 4.2: Experimental results for the dining philosophers problem.

4.6.3 Database application

A database file supports operations for inserting, deleting, and accessing records. An unsorted sequential access file is considered for simplicity. Any request is handled by scanning all the records sequentially and permitting access when the stored key matches the given key. In the object-oriented framework, the sequential file may be viewed as a collection of record objects, each storing one key value with associated data and providing methods for their access.

Figure 4.4 gives a representation of the database application and its description is as follows :

- A *record* class provides the methods *insert()*, *read()*, *modify()* and *delete()*, which may be invoked for performing the respective database operations. The shared variables *r_key*, *r_data* and *r_next* store the key value, associated data and the identity of the successor record respectively. The *lock l_key* ensures mutual exclusion to these variables.

  On invocation, *insert()* inserts the given key and data if the stored key is *null*, else it invokes the *insert()* of the next record. Similarly, *read()*, *modify()* and *delete()* perform their respective operations and invoke the same operation of the next record on account of failure.

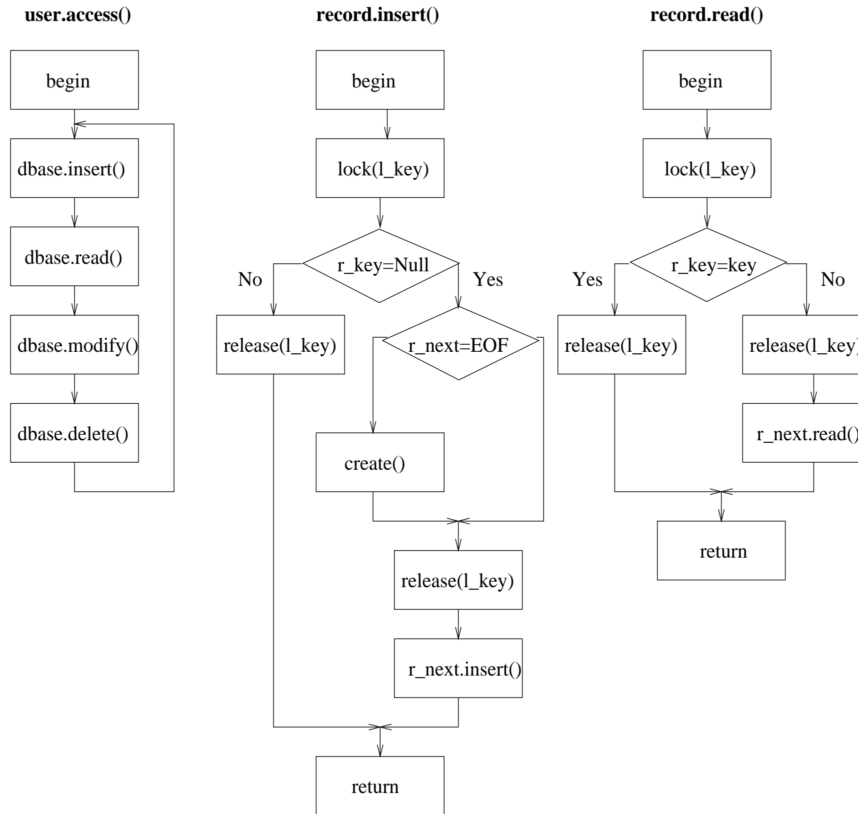


Figure 4.4: CFGs for the database problem.

- A user class provides a method *access()* that invokes the methods of the *record* class.

Table 4.3 gives the results for a database application.

Tool	2 users		3 users	
	Nodes	Time	Nodes	Time
OMEGA	4863	27.6	19603	5:10:04.8
ALPHA	3471	24.1	8979	1:36.1
Reduction %	28.62%		54.20%	

Table 4.3: Experimental results for the database application.

4.6.4 Traffic problem

The traffic problem involves movement of traffic across a bridge such that at any time, traffic is moving in only one direction. There is also a maximum limit on the number of vehicles that can be on the bridge at a time. Figure 4.5 gives a representation of the traffic problem and its explanation is as follows :

- The class *Bridge* contains the shared variables *l_dir* and *l_num*, which allow access to the variables controlling the direction of travel (*dir*) and the number of vehicles traveling (*num*), respectively.

  The *Bridge* class also contains a method *cross()*, which allows a object of *Car* class to cross the bridge by accessing the above mentioned shared variables. The *cross()* method checks if the current direction of travel is the same as that of its direction, and if the number of cars are less than the maximum allowed, and crosses the bridge if both conditions are satisfied.

- The *Car* class has a method called *start()*. When the car wants to cross the bridge, the method *start()* invokes the *cross()* method of the *Bridge* class.

Table 4.4 gives the results for a simulation of traffic movement across a bridge.

Tool	2 cars		3 cars		4 cars		5 cars	
	Nodes	Time	Nodes	Time	Nodes	Time	Nodes	Time
OMEGA	221	0.1	2444	9.2	26973	41:11.8	-	-
ALPHA	50	0.01	230	0.1	1310	3.6	7776	3:31.52
Reduction %	77.37%		90.59%		95.14%		-	

Table 4.4: Experimental results for the traffic simulation.

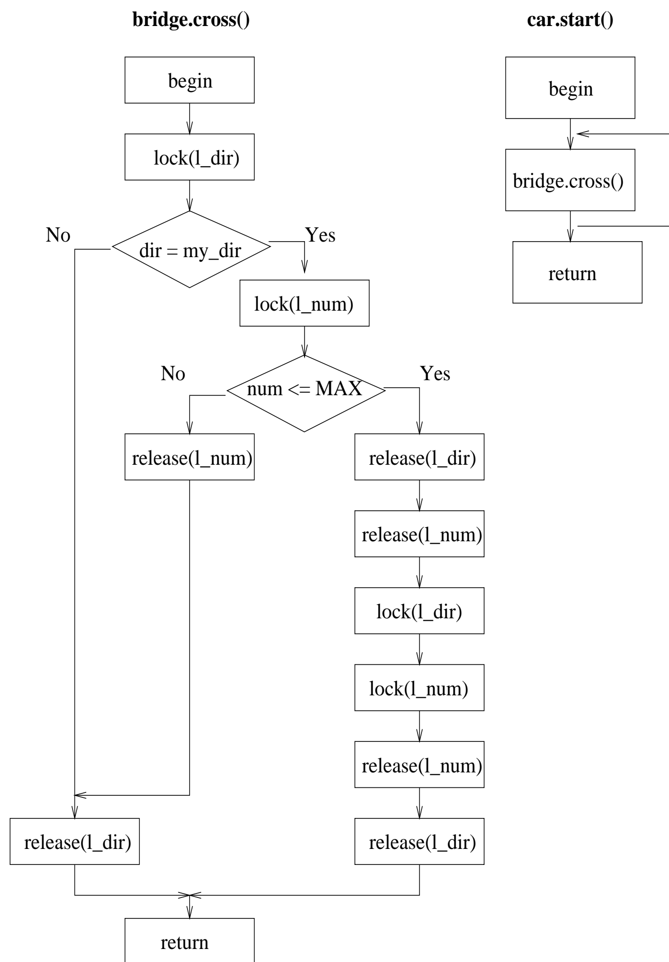


Figure 4.5: CFGs for the traffic problem.

4.7 Discussion

It is evident from the results shown, that the percentage of reduction in complexity due to the apportioning-based analysis increases as the number of threads increases. This is because, for the traditional analysis (OMEGA), the local analysis of each of the methods gets reflected repeatedly in the global outlook. This does not happen in the case of the apportioning-based analysis (ALPHA). An increase in the number of threads or the number of shared objects, would make the difference more substantial.

Chapter 5

Reachability Graph Generation In Parallel

5.1 Introduction

From the results of the sequential reachability graph generation, we infer that the generation of the reachability graph of a concurrent program takes both exponential time and space. Thus, for analysis of programs with a large number of threads a faster way of generation of reachability graphs must be found. Even though apportioning reduces the number of states of the reachability graph, thus reducing both time and space, a large number of threads will still take exponential time to analyse.

In order for such a tool to be practically useful, it must be able to generate the reachability graphs in a reasonable amount of time for a program with large number of threads. In this chapter, a parallel algorithm is described which attempts to reduce at-least, the time taken to generate the reachability graphs.

5.2 Algorithm

The algorithm is as follows :

1. /* Generate the reachability graph of each single thread of execution in the program, using the algorithm given in section 3.3. */

   $\forall i \in T$: Generate R_i ;

   where,
   T is the set of threads in the program and
   R_i is the reachability graph of thread i.

2. /* Generate the cross-product of the sets of nodes in each of the above reachability graphs */

   $Generate\ G = all\ tuples\ (\ s_1, s_2, \ldots\ s_i\ \ldots\ , s_n\ )$;
   where,
   s_i is a state in R_i.

3. /* Remove all invalid nodes from the combination, checking for violation of the mutual exclusion principle and the deadlock condition */

   (a) /* Catch errors due to violation of the mutual exclusion principle. */

   If $\exists\ l$ such that $l \in x_i, l \in x_j$ then
   n = invalid;
   where,
   l is some lock,
   s_i and s_j are states of threads i and j respectively,
   x_i and x_j are the lock statuses of s_i and s_j, and
   n is some node in graph G containing states s_i and s_j.

(b) /* Catch errors due to violation of the deadlock condition. */

If $\exists\ l_1 \in x_i^{'}, \exists\ l_2 \in x_j^{'}$ such that $l_1 \in x_j, l_2 \in x_i, l_1 \neq l_2$ then
n = invalid;
where,
l_1, l_2 are some locks,
x_i and x_j are the lock statuses of s_i and s_j,
$x_i^{'}$ and $x_j^{'}$ are the lock statuses before the action of s_i and s_j,
s_i and s_j are states of threads i and j respectively,
n is some node in graph G containing states s_i and s_j.

5.3 Explanation

The algorithm works as follows :

- The first step of the algorithm generates the reachability graph of each thread. The reachability graph of a single thread of execution, enumerates all possible states achievable by it. This is the set of states which a thread can contribute to the global reachability graph. The states of this thread which are actually present in the global reachability graph will be a subset of this set.

  This step catches the following errors in programs :

  1. Method completion before release of acquired locks.
  2. Methods in an object waiting for acquisition of locks, in a cyclic manner (i.e. a deadlock).
  3. Trying to acquire the same lock twice.
  4. Trying to release the same lock twice.
  5. Cyclical invocations involving local, global or a mixture of local and global invocations

- The second step of the algorithm generates all possible combination of nodes which can be achieved by the threads running concurrently.

- We now have all the possible nodes in the program, a subset of which will form the reachability graph. In order to arrive at the reachability graph, we have to remove all the spurious states. This step removes all the invalid, i.e, unreachable nodes by checking for the following error conditions :

  1. **Errors occurring due to violation of the mutual exclusion principle** :

     A mutually exclusive area in a program is a region of isolation, where only a single thread can execute at a time. In the second step of the algorithm we have taken into consideration all possible combinations of nodes which can occur in the program.

     In such a case, some nodes will be generated which may have more than one thread in a mutually exclusive region. Access is allowed to a mutually exclusive region through the acquiring of some locks. Hence, if two or more threads are shown to be present in a mutually exclusive region, they will all be holding the same locks needed for mutual exclusion. Hence, a summation of locks will show such an error.

     A thread acquiring a lock, which results in it entering a mutually exclusive region, can be thought of as some kind of a *barrier*, where other threads have to wait until the thread holding the lock for that region releases it.

     Once the thread holding the lock releases it, another thread (which was waiting on that lock) can proceed ahead by acquiring the lock. In other words, we can say that a *barrier* is made up of a thread releasing a lock at one end and one or more threads waiting on that lock at the other end (figure 5.1). Hence, two or more threads cannot co-exist across any *barrier*, since it would mean that more than one thread holds the same resource.

     Our check for violation of mutual exclusion would be then to check if there exists threads across any barrier. If so, declare the state as invalid or unreachable. The check is done by summing up the lock statuses of all threads and finding out

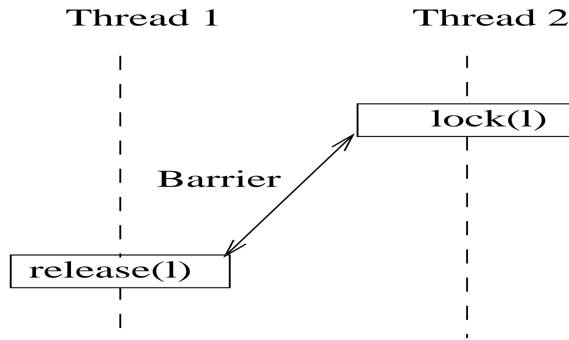


Figure 5.1: A barrier between two threads.

if more than one thread holds the same lock. This is done as follows :

If $\exists\ l$ such that $l \in x_i, l \in x_j$ then,
n = invalid;
where,
l is some lock,
s_i and s_j are states of threads i and j respectively,
x_i and x_j are the lock statuses of s_i and s_j and
n is some node in graph G containing states s_i and s_j.

2. **Errors occurring due to violation of the deadlock condition** :

   A deadlock occurs when two or more threads cyclically wait for resources. The program cannot proceed further ahead from this point of execution. In terms of a *barrier*, a deadlock can be considered as a set of *barriers* crossing each other.

   As mentioned before, no two threads can co-exist across a *barrier*. So now by *crossing each other*, we mean that these *barriers* itself cannot co-exist, because they exist across each other.

We had mentioned earlier that, a *barrier* is made up of one thread releasing a lock (which it had formerly acquired) and one or more threads waiting to acquire this lock. Now a deadlock is a state were all threads wait cyclically for locks, i.e., each of these threads are at one end of a *barrier* (the *waiting for the lock* end). The corresponding other end in each *barrier*, that of a thread releasing a lock will never occur, since a program cannot proceed ahead of a deadlock.

Suppose the threads ignore such a deadlock, manage to get their required resources and proceed further. These are now threads with states across two or more *barriers* and their resources overlap. Any node which follows, will then have a sum of locks greater than the total number of locks.

Now let us consider a node where, all of the threads involved in the deadlock have reached a state where they are all releasing the corresponding locks involved in the deadlock (the locks which they had spuriously acquired). Such a node is shown in figure 5.2. If such a node does not exist (the program may not specify for some of the locks acquired), i.e., one or more of the threads does not have such a release, then anyway that thread holds the lock forever (after it initially acquires it) and hence our first condition of violation due to

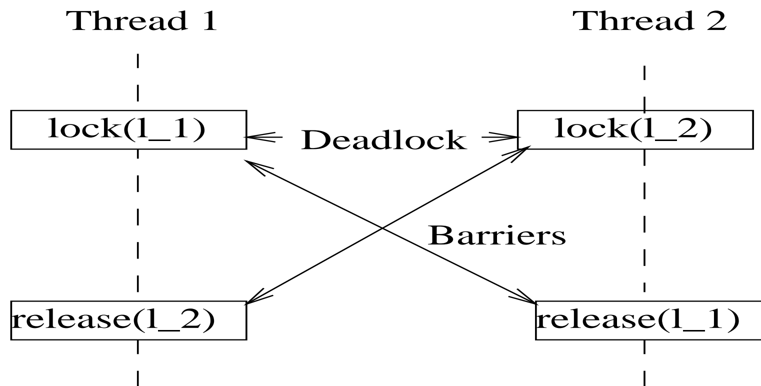


Figure 5.2: A deadlock condition.

mutual exclusion would catch that error.

At such a state, each of the lock is held by two threads (a violation of the mutual exclusion condition). Now, the lock status of the node is the result of the operations performed by each of the threads. So the threads release the locks involved in the deadlock, and hence, even though the lock status formed by the effect of all these threads does not violate the mutual exclusion condition, the node so formed is an invalid node, because none of the threads could have reached that state.

In essence, we can capture all spurious states resulting from the violation of the mutual exclusion condition except for the above described nodes, whose parents violate the mutual exclusion condition, but still the nodes result in a valid lock status.

Considering such a case for two threads, we can represent the error checking as :

If $\exists\ l_1 \in x_i^{'}, \exists\ l_2 \in x_j^{'}$ such that $l_1 \in x_j, l_2 \in x_i, l_1 \neq l_2$ then
n = invalid;
where,
l_1, l_2 are some locks,
x_i and x_j are the lock statuses of s_i and s_j and
$x_i^{'}$ and $x_j^{'}$ are the lock statuses before the action of s_i and s_j,
s_i and s_j are states of threads i and j respectively,
n is some node in graph G containing states s_1 and s_2.

The above mentioned errors are the only errors that can occur when analysing any concurrent program which uses shared memory. Hence, by catching these errors we can safely conclude that, the remaining nodes are error-free and hence reachable.

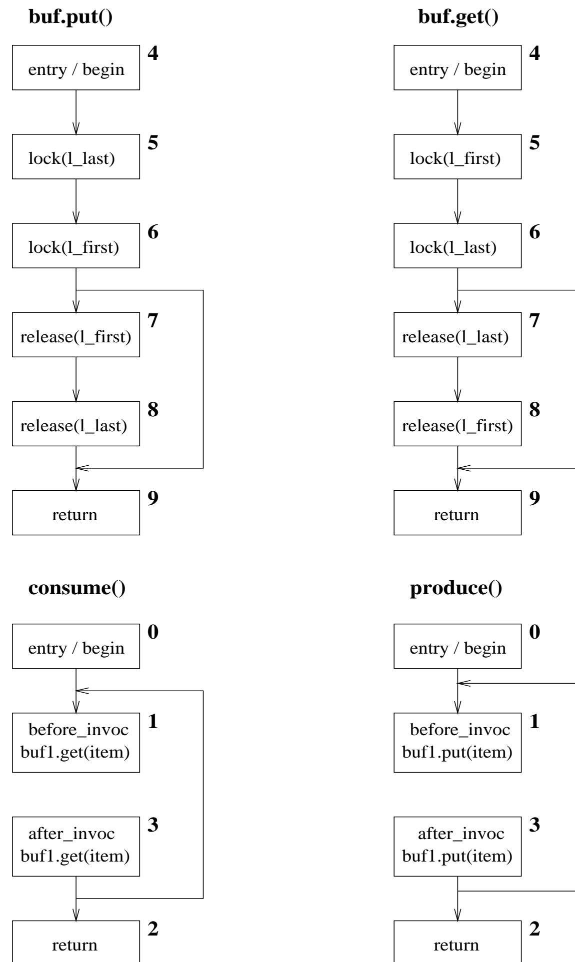


Figure 5.3: CFG for the producer-consumer problem (with ids).

5.4 Example: producer-consumer problem

We explain the algorithm now, with the help of the producer-consumer problem. We assume a program consisting of a single thread each of producer and consumer. Figure 5.3 gives the control flowgraph for the producer-consumer problem given in section 3.7 with each state identified. Table 5.1 gives the tabularized form of the control flowgraph.

Node	Successors
0	1
1	4
2	-
3	2,1
4	5
5	6
6	7,9
8	9
9	3

Table 5.1: Tabularized form of the CFG of a producer/consumer.

Node	Lock states before	Lock states after	Successors	Old node id
0	00	00	1	-
1	00	00	4	-
2	00	00	-	-
3	00	00	2,1	-
4	00	00	5	-
5	00	10	6	-
6	10	11	7,10	-
7	11	10	8	-
8	10	00	9	-
9	00	00	3	-
10	11	11	11	9
11	11	11	12,13	3
12	11	11	-	2
13	11	11	14	1
14	11	11	-	4

Table 5.2: Reachability graph of a single thread of producer.

1. In the first step of the algorithm we generate the reachability graphs of all threads. The reachability graph of the producer and the consumer is given in table 5.2 and table 5.3 respectively.
2. In the second step, we generate the combination of all states in each of the reachability graph generated, i.e., the combinations :

$$\{ \ (0,0); \ (0,1); \ \ldots \ (1,0); \ (1,1); \ \ldots \ (14,14) \ \} \ .$$

3. Now we remove the errors due to the violation conditions specified. Figure 5.4 shows the format of a node in the reachability

Node	Lock states before	Lock states after	Successors	Old node id
0	00	00	1	-
1	00	00	4	-
2	00	00	-	-
3	00	00	2,1	-
4	00	00	5	-
5	00	01	6	-
6	01	11	7,10	-
7	11	01	8	-
8	01	00	9	-
9	00	00	3	-
10	11	11	11	9
11	11	11	12,13	3
12	11	11	-	2
13	11	11	14	1
14	11	11	-	4

Table 5.3: Reachability graph of a single thread of consumer.

graph. The figure 5.5 shows how the nodes generated, are checked for violations. A node is checked for validity by adding up the lock statuses (after the effect of each thread) of each thread state. Invalid states occurring due to violation of the mutual exclusion condition are also shown.

An error occurring due to violation of the deadlock condition is shown in figure 5.6. Here, both the producer and the consumer threads are releasing a lock which they could not have acquired (i.e. a state after a deadlock). The error is caught by checking

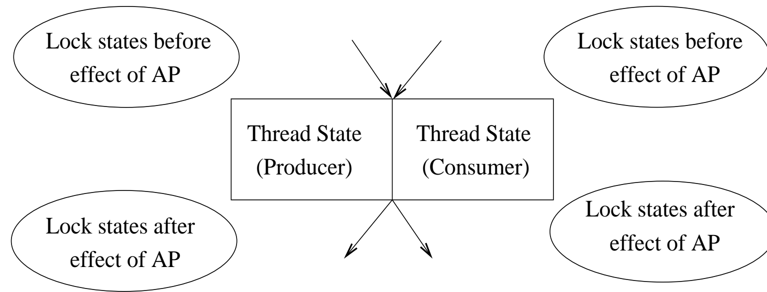


Figure 5.4: Representation of a reachability graph node.

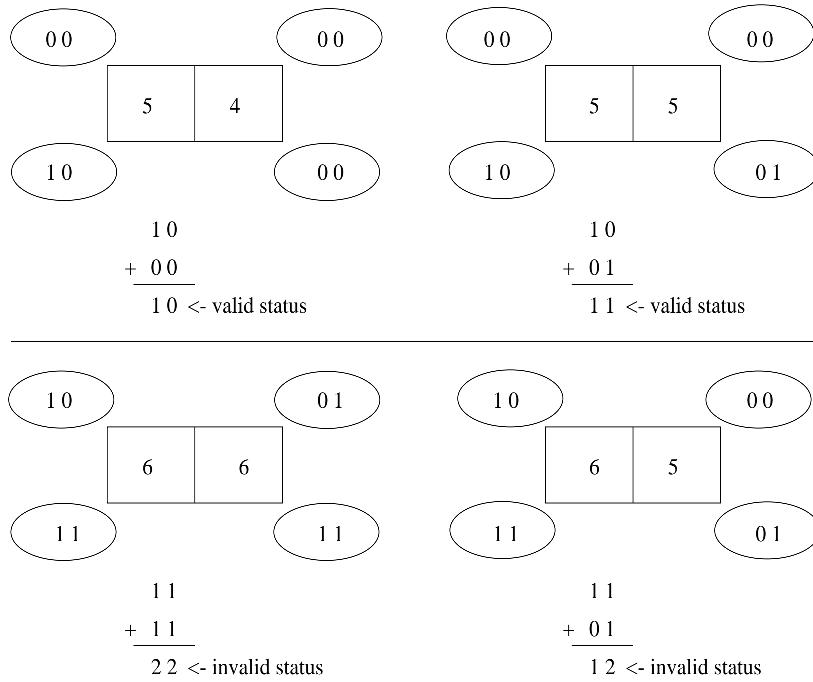
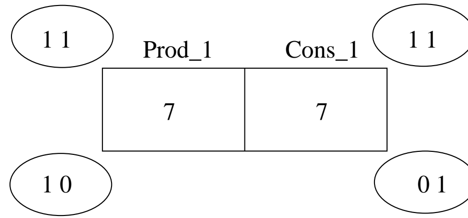


Figure 5.5: Node generation process.



1 1 <- Status before effect of release(l_first) :Prod_1	1 0 <- Status after effect of release(l_first) :Prod_1
+ 0 1 <- Status after effect of release(l_last) :Cons_1	+ 1 1 <- Status before effect of release(l_last) :Cons_1
2 1 <- Invalid Status	2 1 <- Invalid Status

Figure 5.6: Error due to a deadlock.

whether the state of the lock remains the same, even after the release of the locks.

We do this by rolling back the status of the producer thread, i.e., adding the lock status of the thread before the effect of the release action. We then do the same for the consumer thread. Both these statuses violate the deadlock condition, hence invalidating the node.

5.5 Experimental results

In order to show the efficiency of the parallel algorithm with respect to the sequential method of generation, we provide the results of the same comparison of ALPHA v/s OMEGA. The algorithm was implemented using C and C++, on a HP K-class server running HP-UX 11.0. While the algorithm is a parallel algorithm, the implementation runs sequentially, since the machine used lacked the parallel run-time libraries. We compare the number of nodes in the reachability graphs generated by them, as well as the CPU time taken to generate these nodes. We also provide a comparison of the timings of both the algorithms given in this thesis.

$ALPHA_n$ and $OMEGA_n$ respectively are the results of the new algorithm for $ALPHA$ and $OMEGA$. The *Nodes* entry in each table gives the sum of the number of nodes in the local and the global reachability graphs, while the *Time* entry gives the CPU time taken in hours:minutes:seconds.tenths.

5.5.1 Producer-consumer problem

Table 5.4 and 5.5 give the comparison of the time taken by both the algorithms for $ALPHA$ and $OMEGA$. Table 5.6 gives the comparative results of $ALPHA$ and $OMEGA$, for more threads.

Tool	2 prod 1 cons		2 prod 2 cons		2 prod 2 cons		3 prod 3 cons	
	Nodes	Time	Nodes	Time	Nodes	Time	Nodes	Time
ALPHA	255	0.2	1335	3.6	7815	3:27.5	46595	3:50:27.8
$ALPHA_n$		0.051		0.094		0.430		2.861
Reduction %		49%		97.4%		88.06%		99.98%

Table 5.4: Timing comparison for the producer-consumer problem (ALPHA).

5.5.2 Dining-philosopher problem

Table 5.7 and 5.8 give the comparison of the timings of both the algorithms for $ALPHA$ and $OMEGA$. Table 5.9 presents the results for

Tool	2 prod 1 cons		2 prod 2 cons		2 prod 2 cons		3 prod 3 cons	
	Nodes	Time	Nodes	Time	Nodes	Time	Nodes	Time
OMEGA OMEGA$_n$	1588	2.0 0.150	13679	3:35.5 1.84	114239	7:38:43.3 34.15	46595	- 9:10.391
Reduction %		92.5%		99.15%		99.87%		-

Table 5.5: Timing comparison for the producer-consumer problem (OMEGA).

Tool	3 prod 3 cons		4 prod 4 cons		5 prod 4 cons	
	Nodes	Time	Nodes	Time	Nodes	Time
OMEGA	924807	9:10.391	-	-	-	-
ALPHA	46595	2.361	1679655	2:9:612	60466215	1:43:20.980
Reduction %	99.99%		-		-	

Table 5.6: Experimental results for the producer-consumer problem.

more philosophers.

Tool	2 phils		3 phils	
	Nodes	Time	Nodes	Time
ALPHA ALPHA$_n$	371	0.1 0.059	5879	1:12.1 0.265
Reduction %		41%		99.63%

Table 5.7: Timing comparison for the dining philosophers problem (ALPHA).

Tool	2 phils		3 phils	
	Nodes	Time	Nodes	Time
OMEGA OMEGA$_n$	707	0.49 0.066	16503	11:07.6 0.843
Reduction %		86.53%		99.87%

Table 5.8: Timing comparison for the dining philosophers problem (OMEGA).

Tool	4 phils		5 phils		6 phils	
	Nodes	Time	Nodes	Time	Nodes	Time
OMEGA	404671	31.175	9838799	20:15.47	-	-
ALPHA	105023	7.342	1889615	3:25.251	340122288	1:24:5.0
Reduction %	74.05%		80.79%		-	

Table 5.9: Experimental results for the dining philosophers problem.

5.5.3 Database application

Table 5.10 and 5.11 give the comparison of the timings of both the algorithms for $ALPHA$ and $OMEGA$. Table 5.12 gives the results for more users in a database application.

Tool	2 users		3 users	
	Nodes	Time	Nodes	Time
ALPHA	3471	24.1	8979	1:36.1
ALPHA_n		0.059		0.268
Reduction %		99.75%		99.75%

Table 5.10: Timing comparison for the database application (ALPHA).

Tool	2 users		3 users	
	Nodes	Time	Nodes	Time
OMEGA	4863	27.6	19603	5:10:04.8
OMEGA_n		0.082		3.040
Reduction %		99.70%		99.98%

Table 5.11: Timing comparison for the database application (OMEGA).

Tool	4 users	
	Nodes	Time
OMEGA	2673627	2:49.964
ALPHA	108123	5.540
Reduction %	96.1%	

Table 5.12: Experimental results for the database application.

5.5.4 Traffic problem

Table 5.13 and 5.14 give the comparison of the time taken by both the algorithms for $ALPHA$ and $OMEGA$. Table 5.15 gives the results for more number of cars.

Tool	2 cars		3 cars		4 cars	
	Time	Reduction	Time	Reduction	Time	Reduction
ALPHA ALPHA$_n$	50	0.01 0.01	230	0.1 0.02	1310	3.6 0.07
Reduction %		0%		80%		98.06%

Table 5.13: Timing comparison for the traffic problem (ALPHA).

Tool	2 cars		3 cars		4 cars	
	Time	Reduction	Time	Reduction	Time	Reduction
OMEGA OMEGA$_n$	221	0.1 0.02	2444	9.2 0.17	26973	41:11.8 2.84
Reduction %		80%		98.15%		99.88%

Table 5.14: Timing comparison for the traffic problem (OMEGA).

Tool	5 cars		6 cars	
	Nodes	Time	Nodes	Time
OMEGA	52836	5.25	420189	58.22
ALPHA	7790	0.43	46670	3.020
Reduction %	85.28%		88.89%	

Table 5.15: Experimental results for the traffic simulation.

5.6 Discussion

The results shown above are that of the sequential implementation of the algorithm. The sequential implementation itself shows a significant reduction in the time taken to generate the reachability graph. A parallel implementation would show an even greater reduction in the time complexity.

Chapter 6

Conclusion and Future Work

The aim during the course of this work was to experimentally verify apportioning as an effective and practically viable tool for reachability analysis of a concurrent Java program. The effectiveness of the technique was proved by showing a reduction in space as compared with that of the traditional reachability analysis algorithm. For the chosen experiments, a typical reduction of 67% was observed in the number of nodes generated. The reduction in nodes increased with increasing number of threads in the program.

Still, sequential reachability graph generation takes both exponential time and space. Towards this end, we developed an algorithm for generation of the reachability graph in parallel. The results of this algorithm as compared with that of the sequential approach showed more than 86% decrease in analysis time. Also, due to this reduction in analysis time, we could analyse programs with a larger number of threads, hence proving the practical validity of the attempt.

Some avenues for possible extension of this work are :

1. Accept a wider range of programs for analysis.

2. Implementation can be extended to other apportioning-based tools like BETA.

3. A more rigorous verification of the parallel algorithm.

Bibliography

[1] G. R. Andrews. Concurrent programming, principles and practice. *Addison-Wesley*, 1991.

[2] K. Arnold and J. Gosling. The Java^{TM} programming language, 2nd ed. *Addison-Wesley*, 1998.

[3] E. Balagurusamy. Programming with Java. *Tata-McGraw Hill*, 1998.

[4] D. Ince. Object oriented software engineering with C++. *McGraw Hill*, 1991.

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

[6] 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.

[7] J. JàJà. An Introduction to parallel algorithms. *Addison-Wesley*, 1992.

[8] A. Majumdar. Reachability graph generation of concurrent Java programs. Indian Institute of Technology, Guwahati, 1999.

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

[10] Parallel programming guide for HP-UX systems, 1st ed. *Hewlett-Packard*, 1998.

[11] R. S. Pressman. Software engineering - a practitioner's approach, 3rd ed. *McGraw Hill*, 1992.

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

[13] R. N. Taylor and L. J. Osterweil. Anomaly detection in concurrent software by static data flow analysis. *IEEE Transactions. Software Engineering*, 265 - 278, 1980.