Apportioning : A Technique for Efficient Reachability Analysis of Concurrent Object-Oriented Programs

Sridhar Iyer.

Dept. of Computer Science
IIT Guwahati
Guwahati, INDIA
email : sri@iitg.ernet.in

S. Ramesh.

Dept of Computer Science
IIT Bombay
Mumbai, INDIA
ramesh@cse.iitb.ernet.in

Abstract

The object-oriented paradigm has been found to be useful for the construction of large and complex concurrent systems. Reachability analysis is an important and well-known tool for static (pre-run-time) analysis of concurrent programs. However, direct application of traditional reachability analysis to concurrent object-oriented programs has many problems, such as incomplete analysis for reusable classes (not safe*) and increased computational complexity (not* efficient*).*

We have proposed a novel technique called apportioning*, for effective reachability analysis of concurrent object-oriented programs, that integrates the techniques of* abstraction *(considering a reduced representation of the system) and* partitioning *(dividing the system into smaller units). The given program is apportioned into a reduced version of each of its classes, as well as a reduced version of the program. The error to be checked is also decomposed into appropriate sub-properties for checking in the reachability graphs corresponding to the apportioned program.*

We have developed a number of apportioning-based algorithms, having different degrees of safety and effectiveness. In this paper, we present the details of one of these algorithms.

1 Introduction

The object-oriented paradigm provides support for the construction of modular and reusable program components and hence is attractive for the design of large and complex distributed systems.

Reachability analysis is an important and well-known static analysis tool for the pre-run-time verification of critical properties in concurrent programs, such as freedom from deadlock. The control-flow graphs of individual processes are modified to highlight the synchronization structure, abstracting away other details. Subsequently the complete state-transition graph of the execution, known as the *reachability graph*, is constructed, thereby modeling the concurrent program as the set of all possible execution sequences ([7], [6]). For properties of the synchronization structure in concurrent programs, reachability analysis provides the same level of assurance as formal verification. The practical potential of reachability analysis for analyzing non-trivial concurrent programs has been shown in [9], and a simple algorithm for the generation of reachability graphs is described in figure 1.

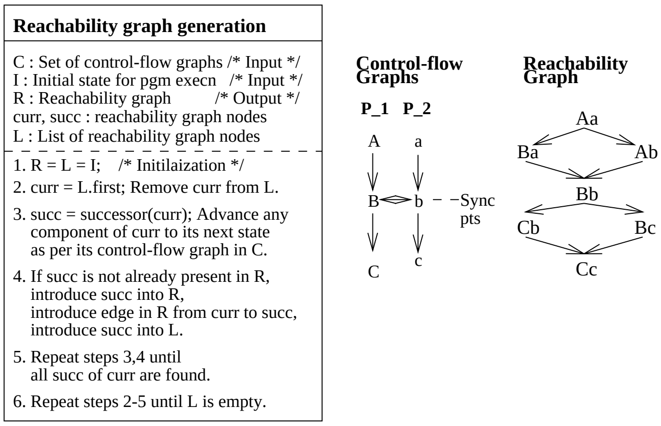

Figure 1: Traditional reachability graph generation

Direct application of traditional reachability analysis to concurrent object-oriented programs, is not *safe*, i.e., it fails to capture all the errors in classes that are exported to libraries, since it deals only with the sequence of interactions present in the given program.

A straight-forward extension of traditional reachability analysis for performing *safe* analysis of concurrent object-oriented programs would be : perform additional analysis for each class to ensure it to be 'error-free' before exporting it to libraries. However, such an extension is not *efficient*, since it aggravates the computational complexity of the analysis, which is already known to be exponential due to "state-explosion".

An analysis technique besides being *safe* and *efficient*, should also be *effective*, in the sense that it should not reject too many correct programs. From our experience, we believe that a single *general-purpose* strategy that is safe for all programs and at the same time, efficient and effective also, is unlikely to exist; any 'one-size-fits-all' analysis strategy is likely to be either ineffective or inefficient.

In this paper we propose a new and powerful technique called *apportioning*, for analysis of concurrent object-oriented programs. Apportioning integrates two well-known techniques for combating state-explosion, viz., *abstraction* [1] and *partitioning* [3]. It relies on a classification of program analysis points as local (having influence within a class) and global (having possible influence outside a class). The given program is *apportioned* into an abstract version for each of its classes, containing only local analysis points for the classes, as well as an abstract version of the whole program containing only global analysis points. The error to be checked (expressed as a property over analysis points), is also decomposed into a number of sub-properties, one corresponding to each abstract portion of the program. The analysis involves generating the reachability graphs for each abstract portion and checking them for the corresponding sub-property using any standard technique.

Apportioning is a general technique using which a number of specific analysis algorithms can be developed, by choosing an appropriate classification for the analysis points. This classification also influences the *safety* (detection of all errors in the program), *efficiency* (reduction of complexity) and *effectiveness* (amount of spurious errors reported) of the algorithms.

Using apportioning, we have developed a number of analysis algorithms that are safe for different classes of programs and having various degrees of efficiency and effectiveness. In this paper we discuss one of the simplest of these algorithms, which we call ALPHA and outline the development of other algorithms. We have formally shown the safety of ALPHA for all concurrent object-oriented programs not having local invocations; other local interactions, such as shared variables, are permitted. We also give examples to illustrate the efficiency and effectiveness of ALPHA. To the best of our knowledge, there is no instance of reachability analysis being applied to concurrent object-oriented programs.

2 The Apportioning Technique

The basic idea of apportioning is similar to *partitioning* : reduce the problem of analyzing a whole program into a number of smaller problems, of individual and independent analysis of portions of the original program. But unlike partitioning, the portions are not mere syntactic sub-units of the program being analyzed but are also their *abstractions*.

While the modular structure of concurrent object-oriented programs is exploited to partition a program into its constituent classes, the abstraction is based upon a novel idea of classification of synchronization points. A synchronization point in any method can be classified either as *local analysis point* (LAP) that corresponds 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*.

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.

The practical utility of apportioning can be seen from the following observations : The complexity (number of states generated for analysis) of traditional reachability analysis [7], 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 $O(c(p_l)^m + (p_g)^T)$, where p_g is the number of GAP in any method. Typically for many programs, $p \approx (p_l + p_g)$. Hence although the complexity of the apportioned analysis is also exponential, the amount of reduction is also composed of exponential terms, and is quite significant.

In our work, we focus on the analysis for errors that correspond to properties of states in the reachability graph. Our approach for implemention of apportioning-based reachability analysis is shown in figure 2 and discussed in the following sections.

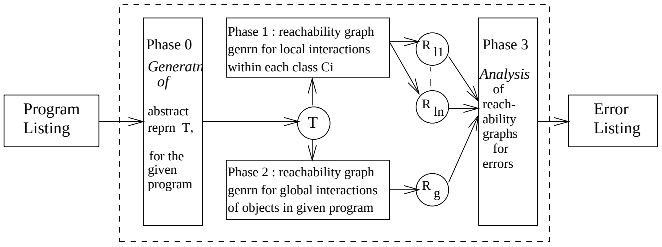

Figure 2: Apportioning-based reachability analysis

3 ALPHA : An efficient apportioning-based analysis algorithm

ALPHA applies to concurrent object-oriented programs [8] having the following features :

- A program has a set of class definitions, a set of object declarations and a MAIN object specification. The MAIN object creates a fixed number of threads and objects (using appropriate constructs) at the start of program execution and invokes appropriate objects to start concurrent execution. The program terminates when all threads return to MAIN and terminate. Threads can not be created or destroyed by other objects.

- 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.

  Different methods of an object may execute concurrently and the model supports a *lock* feature, to enable mutually exclusive access to the data shared by an object's methods.

The above model is kept simple, for ease of illustration. Nevertheless, it is powerful enough to encompass many concurrent object-oriented systems, such as COOL [5]. Extensions of ALPHA to models that support inheritance, bounded creation and destruction of objects and bounded multiple concurrent invocations of the same method, are discussed in [4].

The model supports *object-level* concurrency (concurrent execution of methods in different objects) as well as *method-level* concurrency (concurrent execution of different methods in the same object). This may give rise to various *data access errors*, such as access of shared data without appropriate *locks* and *synchronization errors*, such as cyclic waiting among methods for acquisition of locks and return of local or global invocations.

We illustrate the details of ALPHA using a simple variation of the producer-consumer problem, shown in figure 3.

3.1 Input to ALPHA

The input to ALPHA is a specification of the program to be analyzed, along with a specification of the error to be checked.

Phase 0 creates the abstract representation (using standard techniques) from the program specification. The abstract representation, as illustrated in figure 3, consists of :

- M_x : The *method control-flow graph* for each method m_x.

- C_i : The *class control-flow graph* for each class c_i. $C_i = [\{M_1, \ldots, M_{\gamma_i}\}, \{\ell_1, \ldots, \ell_{\delta_i}\}]$, where γ_i is the number of methods and δ_i is the number of *locks* defined in c_i.

- T : The *program control-flow graph*. $T = [\{C_1, \ldots, C_\alpha\}, \{o_{1,1}, \ldots, o_{\alpha,\beta_\alpha}\}, IS]$, where α is the number of classes, β_i is the number of objects of class c_i ($o_{i,j}$ is the object corresponding to the j^{th} instance of c_i), and IS is the initial state for program execution. IS specifies the methods that are *active* at the start of program execution and implicitly captures the concurrency (number of threads) in the program.

The error to be checked is specified as a property of the global state of the program. This state property is specified as a boolean formula P_t (in disjunctive

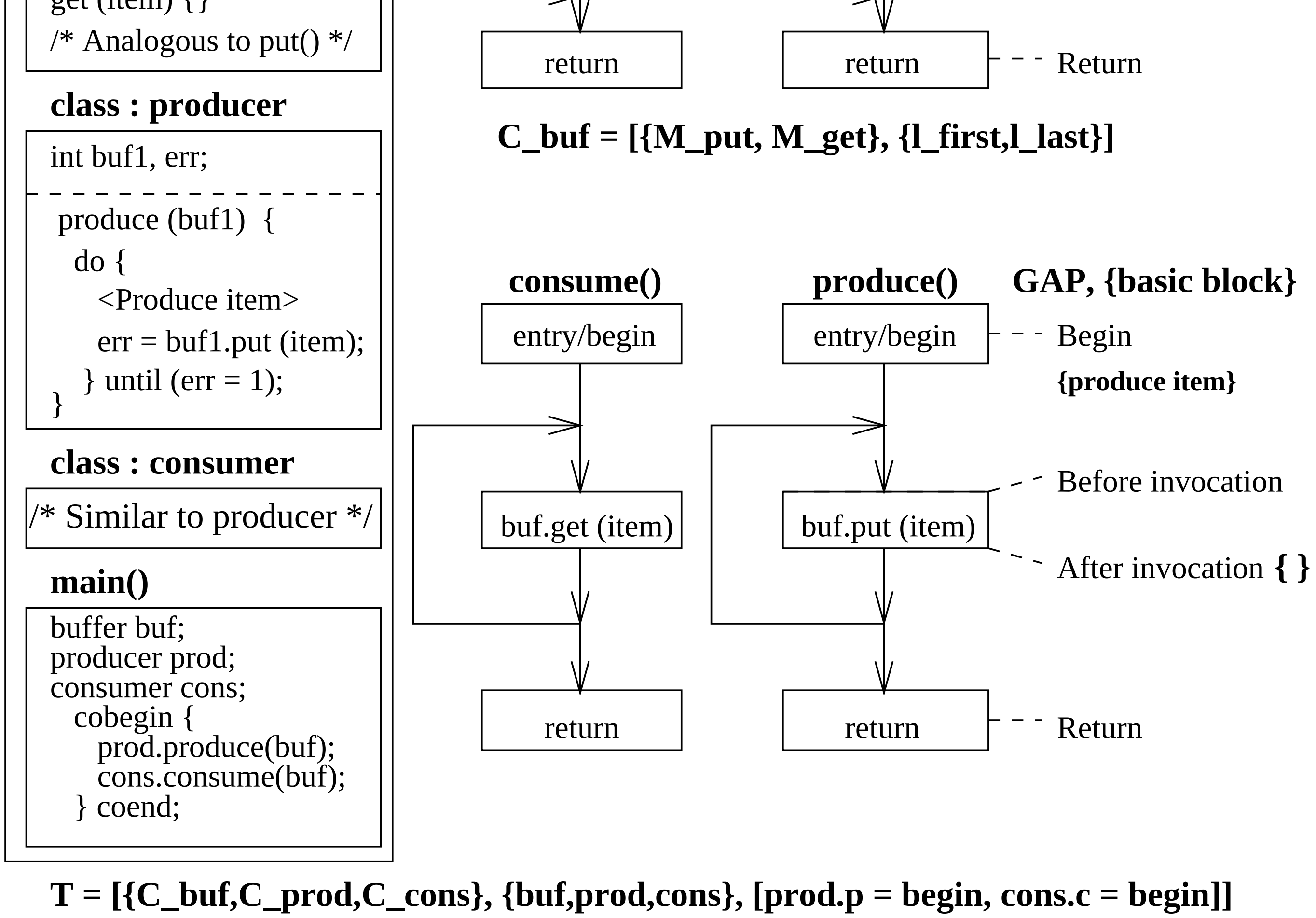

Figure 3: Program flow-graph for producer-consumer example

normal form), involving propositions of analysis points of methods. Thus $P_t = P_t^1 \vee P_t^2 \vee \ldots \vee P_t^{\epsilon}$, where each $P_t^k (1 \leq k \leq \epsilon)$, is a conjunction of propositions of the form $m_x\_AT\_a$ (*method* m_x *is at analysis point* a).

3.2 Classification of analysis points

ALPHA classifies the analysis points corresponding to access of shared data and local invocations as LAP and those corresponding to global invocations as GAP. The analysis points *entry, begin* and *return* are classified as both LAP and GAP.

3.3 Phase 1

Phase 1 has two steps : For each method, an operation called $GAP\_hiding()$ is applied to abstract out all non-LAP, resulting in a reduced graph. Subsequently, for each class C_i, a reachability graph R_{l_i} is generated from the reduced graphs, *assuming concurrent execution of the methods.*

For any method M, all LAP of M are retained in $M' = GAP\_hiding(M)$. But every non-LAP g of M is removed in M', with any edge from/to g being replaced by an edge from $L\_pred_M(g)$ ($nearest\_LAP\_predecessors$ of g), to $L\_succ_M(g)$ ($nearest\_LAP\_successors$ of g), as illustrated in figure 4. Multiple edges between any two nodes in M' are represented only once and M' has fewer nodes and edges than M. The existence of $L\_pred_M(a)$ and $L\_succ_M(a)$, for any analysis point a in M, can be easily shown by induction. Now $GAP\_hiding(C_i) = [\{GAP\_hiding(M_1), \ldots, GAP\_hiding(M_{\gamma_i})\}, \{\ell_1, \ldots, \ell_{\delta_i}\}]$.

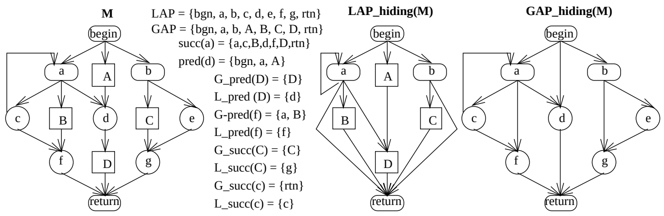

Figure 4: Hiding on method control-flow graphs

R_{l_i} is given by $R\_gen(\{GAP\_hiding(C_i)\}, \{o_{i,1}\}, IS_{l_i})$, where

$R\_gen()$ is the reachability graph generation algorithm described in figure 5, that uses the rules described in figure 6 to determine successors of nodes. Only one instance of C_i ($o_{i,1}$) is considered and IS_{l_i}, the initial node for R_{l_i}, has all methods at their respective *entry* points. Each node in R_{l_i} has fields to record the status of each *lock* and method in $GAP\_hiding(C_i)$, and the structure of R_{l_i} is illustrated in figure 7.

Note that Rule $S1$ for $R\_gen()$ enables methods to make a non-deterministic transition from *entry* (passive state before invocation) to *begin* (active state upon invocation), thereby capturing all possible concurrent execution of the methods.

```
R_gen(C, O, I)      /* Reachability graph generation. */
C : Set of class control-flow graphs.
O : Set of objects (class instances).
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 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) then {
       N_r = N_r ∪ {next.element[1]};
       list = list ∪ {next} − {current};
     }
     ≪ Introduce appropriate edges into E_r ≫;
     next = next − {next.element[1]}; /* remove 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 */
}
```

Figure 5: Algorithm for reachability graph generation

3.4 Phase 2

Phase 2 also has two steps : For each method, an operation called $LAP\_hiding()$ is applied to abstract out all non-GAP, resulting in a reduced graph. Subsequently, a reachability graph R_g is generated from the reduced graphs.

For any method M, $LAP\_hiding(M)$ is complementary to $GAP\_hiding(M)$ described in Phase 1. All GAP of M are retained in $M' = LAP\_hiding(M)$, and every non-GAP of M is removed in M', with appropriate changes to the corresponding edges.

$LAP\_hiding(C_i) = [\{LAP\_hiding(M_1), \ldots, LAP\_hiding(M_{\gamma_i})\}, -]$, and $LAP\_hiding(T) = [\{LAP\_hiding(C_1), \ldots, LAP\_hiding(C_\alpha\}), \{o_{1,1}, \ldots, o_{\alpha,\beta_\alpha}\}, IS]$.

```
find_successors(node, field)
/* Successors of node, are determined by atomic application of
  one of the following rules to the analysis point field.
  There is a successor for node corresponding to each
  immediate_successor of field.
  A set of successors for node is returned. */
{ /* Assuming unique identifiers for methods */
  apt1 : field.m_x_sts;    /* Present analysis point of m_x */
  apt2 : next_node.m_x_sts; /* Successor point for m_x */

  Rule S1 : If (apt1 = entry) then apt2 = begin;
        /* S1 is used only in Phase 1 and not in Phase 2 */
  Rule S2 : If (apt1 = begin) then
        apt2 = M_x.immediate_successor(apt1);
  Rule S3 : If ((apt1 = wait_ℓ_d) ∧ (node.ℓ_d_sts = open)) then
        next_node.ℓ_d_sts = closed;
        next_node.m_x_lks = node.m_x_lks ∪ {ℓ_d};
        apt2 = M_x.immediate_successor(apt1);
  Rule S4 : If (apt1 = released_ℓ_d) then
        next_node.ℓ_d_sts = open;
        next_node.m_x_lks = node.m_x_lks − {ℓ_d};
        apt2 = M_x.immediate_successor(apt1);
  Rule S5 : If ((apt1 = before_invoke_m_y) ∧ (m_y_sts = entry))
        then apt2 = invoked_m_y; next_node.m_y_sts = begin;
        ≪ Bind parameters of m_y, to objects ≫
  Rule S6 : If ((apt1 = invoked_m_y) ∧ (m_y_sts = return)) then
        apt2 = after_invoke_m_y; next_node.m_y_sts = entry;
  Rule S7 : If (apt1 = after_invoke_m_y) then
        apt2 = M_x.immediate_successor(apt1);
  Rule S8 : If (apt1 = return) then apt2 = entry;
}
```

Figure 6: Determining successors of reachable nodes

R_g is given by $R\_gen(LAP\_hiding(T))$. All instances of all classes are considered and the initial node for R_g, is IS, as defined in T. Each node in R_g has fields to record the status of each method of each object in T, and the structure of R_g is similar to that of R_{l_i} (figure 7).

3.5 Phase 3

Phase 3 again has two steps : For the given state property to be checked, say P_t, it generates a number of sub-properties each involving either purely LAP or GAP and then checks each of these sub-properties on the appropriate reachability graphs R_l or R_g.

Given that $P_t = P_t^1 \vee P_t^2 \vee \ldots \vee P_t^\epsilon$, where each $P_t^k (1 \le k \le \epsilon)$, is a conjunction of analysis point propositions. For every P_t^k, P_g^k is the sub-predicate having the propositions of P_t^k that involve GAP. $P_{l_{i,j}}^k$ is the sub-predicate having the propositions of P_t^k that involve LAP of the methods in object $o_{i,j}$. Corresponding to each P_t^k, $P_{l_{i,j}}^k (1 \le j \le \beta_i)$ is checked in $R_{l_i} (1 \le i \le \alpha)$ and P_g^k is checked in R_g, using standard algorithms.

For the producer-consumer example, consider a deadlock error, which can be represented as a state property P_t given by $P_t = (prod.invoked\_put) \wedge$

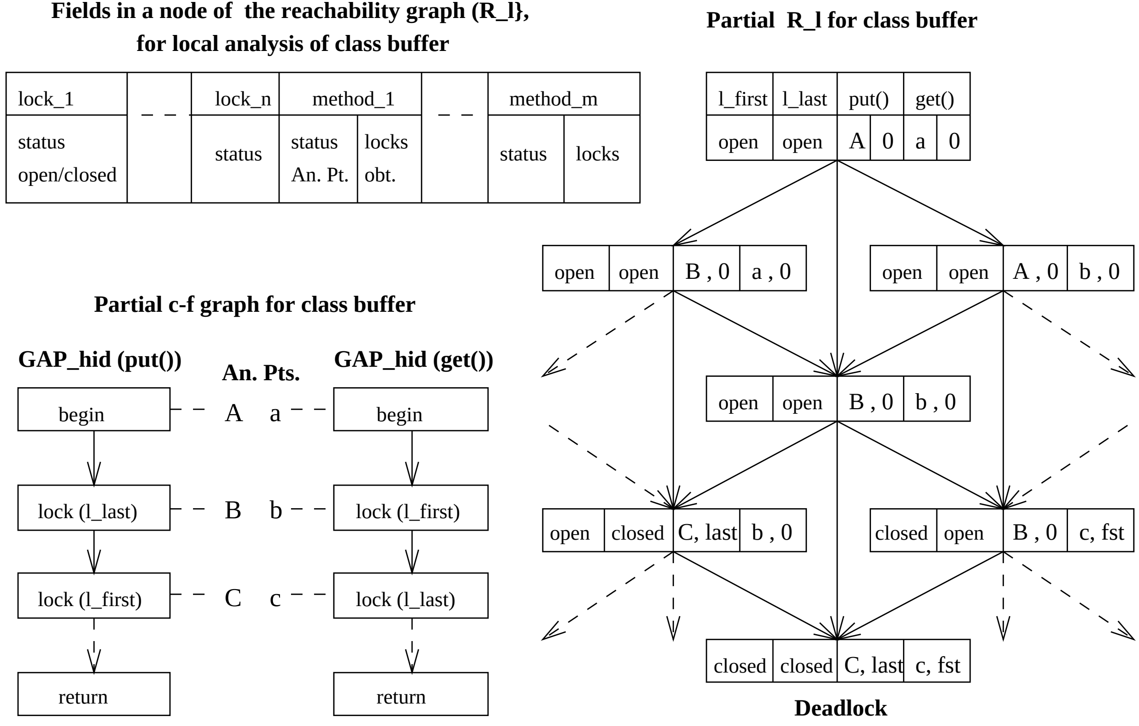

Figure 7: Reachability graph generation in Phase 1

$(put.lock\_first) \wedge (cons.invoked\_get) \wedge (get.lock\_last)$. Corresponding sub-predicates are : $P_g = (prod.invoked\_put) \wedge (cons.invoked\_get)$, and $P_{l_{buf}} = (put.lock\_first) \wedge (get.lock\_last)$. Now when P_g is satisfied in R_g and $P_{l_{buf}}$ is satisfied in $R_{l_{buf}}$, the deadlock is detected.

4 Safety, Efficiency and Effectiveness of ALPHA

The main criteria that should be satisfied by ALPHA is *safety*, i.e., it should reject all programs that are rejected by a corresponding non-apportioned reachability analysis algorithm. A more formal definition is : Given a program S and property P, the apportioning $< \{S_i | i = 1, \ldots, n\}, \{P_i | i = 1, \ldots, n\} >$ of $< S, P >$, is safe provided, whenever S satisfies P, each S_i satisfies the corresponding P_i.

ALPHA is *efficient* if the complexity of checking if each S_i satisfies the corresponding P_i, is an order of magnitude lower than checking if S satisfies P; including the cost of apportioning S and P.

ALPHA is *effective* if it rejects only those programs that are rejected by the corresponding non-apportioned algorithm.

4.1 Safety

Now we prove the following :

Theorem 1 *ALPHA is safe for programs that do not have local invocations, i.e., for programs wherein a method does not invoke another method of the same object.*

Let R_t denote a non-apportioned reachability graph for the given program. Now theorem 1 immediately follows from the following two lemmas :

Lemma 1 : For any state property P_t^k that holds in R_t, the corresponding property $P_{l_{i,j}}^k (1 \leq j \leq \beta_i)$ holds in $R_{l_i} (1 \leq i \leq \alpha)$.

Lemma 2 : For any state property P_t^k that holds in R_t, the corresponding property P_g^k holds in R_g.

A state property P holds in reachability graph R iff there exists a state s in R such that s satisfies the predicate for P ($s \models P$).

We prove lemmas 1 and 2 by showing that for any node s in R_t, corresponding nodes exist in R_l and R_g. Let s have fields $[s^1, \ldots, s^\lambda]$. We define the following operations :

$proj_{i,j}(s)$: This is the *projection* of the fields in s that correspond to the entries for object $o_{i,j}$.

$proj_g(s)$: This is the *projection* of the fields in s that correspond to the analysis points of all methods of all objects.

$G\_hide(s)$: This is a set of nodes s_1, such that if $s^k (1 \leq k \leq \lambda)$ is LAP or if s^k is status of *lock*, then $s_1^k = s^k$, and if s^k is GAP, then $s_1^k = a$, where $a \in L\_pred_{M_k}(s^k)$.

$L\_hide(s)$: This is a set of nodes s_1, such that if $s^k (1 \leq k \leq \lambda)$ is GAP or if s^k is status of *lock*, then $s_1^k = s^k$, and if s^k is LAP, then $s_1^k = a$, where $a \in G\_pred_{M_k}(s^k)$.

Now lemmas 1 and 2 are proved using the following lemmas 3 and 4 respectively,

Lemma 3 : For any state s in R_t, there exists $s_l \in G\_hide(proj_{i,j}(s))$ such that s_l is in R_{l_i}.

Lemma 4 : For any state s in R_t, there exists $s_g \in L\_hide(proj_g(s))$ such that s_g is in R_g.

Lemmas 3 and 4 are proved by induction on the length of the path connecting the root of R_t to s. A more detailed treatment of the proofs is given in [4].

4.2 Efficiency

In order to show the efficiency of ALPHA, we compare the number of nodes in the reachability graph generated by it, with that of a similar algorithm, which we call OMEGA, that classifies *all* LAP as GAP also; *OMEGA corresponds to the extension of traditional reachability analysis to concurrent object-oriented programs.*

Table 1 shows the comparison of ALPHA and OMEGA for the producer-consumer example (concurrent execution of threads t_1 and t_2 having p_1 and p_2 distinct analysis points, results in $(p_1 \times p_2)$ nodes in the reachability graph). Table 2 shows their comparison for another standard example, the dining-philosophers problem. The comparison for more examples, such accessing a stack, a database file, is given in [4].

4.3 Effectiveness

The following result for effectiveness of ALPHA follows directly from the definitions of sub-properties :

Theorem 2 *ALPHA effective when for any P_t^k, either $P_g^k = P_t^k$ or $P_{l_{i,j}}^k = P_t^k$.*

Thus ALPHA is effective for all data access errors, cyclic acquisition of *locks* and global invocation deadlocks in the program. ALPHA is not effective for arbitrary mixed interactions.

5 More apportioning-based algorithms

As mentioned earlier, by varying the classification of analysis points, many algorithms that are safe for different classes of programs and having different degrees of efficiency and effectiveness, can be generated. These algorithms have the same actions for Phase 0 and Phase 1, and differ only in Phase 2 and Phase 3.

At one end of the spectrum is OMEGA, which corresponds to an extension of traditional reachability analysis to concurrent object-oriented programs. It classifies *all* LAP as GAP also and is safe for all programs in the chosen model. It is the most effective of apportioning-based algorithms but least efficient. At the other end of the spectrum is ALPHA, which classifies only *entry*, *begin* and *return* analysis points as being both LAP and GAP, and is safe for the subclass of programs that do not have local invocations. It is the most efficient of these algorithms but may be the least effective.

Some of the other algorithms that lie between this spectrum between ALPHA and OMEGA are :

BETA : BETA classifies the LAP corresponding to all local invocations as being GAP also and is safe for all programs in the chosen model. It is less efficient than ALPHA but more effective.

GAMMA : GAMMA classifies the LAP corresponding to only those local invocations that directly or indirectly result in a global invocation, as being GAP also, and is safe for all programs in the chosen model. GAMMA is more efficient than BETA but less effective.

More details of these algorithms and their proofs are given in [4].

6 Conclusions

The apportioning technique developed here, has some similarities (and differences) with some reachability analysis approaches in the literature. McDowell [6] employs an abstraction approach for "folding" many states in a reachability graph into a single state. However, such an approach is applicable only to programs composed of identical sets of tasks.

Taylor [7] employs a partitioning approach for "parceling" the analysis into biconnected components, applicable to programs composed of disjoint sets of tasks. Yeh and Young [10] describe an approach called *compositional reachability analysis* that allows one to analyze individual portions of a large system and hierarchically compose partial analysis results. Cheung and Kramer [2] describe a technique for incorporating context constraints into compositional reachability analysis. However, their use of process algebraic structures results in the generation of larger graphs for the subsystems, than would be generated by a straight-forward reachability graph generation. In apportioning, the abstraction of analysis points is performed *prior* to generation of the reachability graphs itself and not just before composing subsystems into larger systems. As a result, the graphs generated for the subsystems are likely to be smaller. Apportioning also exploits the similarity among subsystems (such as objects belonging to the same class), to generate a single reachability graph for all such subsystems.

We make the following observations regarding apportioning : (i) the efficiency increases as $p \to (p_l + p_g)$, where p is the total number of analysis points and p_l, p_g are the number of LAP and GAP, respectively and (ii) the efficiency increases as the number of threads

	A. Number of Analysis Points				B. Nodes in Reachability Graph			
Method $\rightarrow$	prod()	cons()	buffer		Case $\rightarrow$	1-producer	2-prod	3-prod
$\downarrow$ AP			put()	get()	$\downarrow$ Graph	1-consumer	2-cons	3-cons
LAP (p_l)	2	2	6	6	OMEGA.R_l †	$(2+2+6^2)=40$	40	40
GAP_ω (p)	4	4	6	6	OMEGA.R_g ‡	$(4+6)^2=100$	4096	123904
GAP_α (p_g)	4	4	2	2	ALPHA.R_g	$(4+2)^2=36$	1024	25600
	Reduction for Apportioning					45.72%	74.27%	79.31%

† Number of nodes in R_l is same for ALPHA also.
‡ In case of 2 or more producers, only one prod() thread can be active in buf.put(), at any given time.

Table 1: Comparison of efficiency for producer-consumer example

	A. Analysis Points			B. Nodes in Reachability Graph			
Method $\rightarrow$	work()	fork		Case $\rightarrow$	2-philosophers	3-phil	4-phil
$\downarrow$ AP		up()	down()	$\downarrow$ Graph			
LAP (p_l)	2	4	4	OMEGA.R_l †	$(2+4*4)=18$	18	18
GAP_ω (p)	10	4	4	OMEGA.R_g	$(26*26)-2(4*4)-2(4*4)=612$	17480	$456,848$
GAP_α (p_g)	10	2	2	ALPHA.R_g	$(18*18)-2(2*2)-2(2*2)=306$	5808	$104,944$
	Reduction for Apportioning				48.57%	66.76%	77.03%

Table 2: Comparison of efficiency for dining-philosopher example

increases (since the analysis of local interactions is not repeated). Although the complexity of apportioned analysis is also exponential in the number of threads, the best-case reduction achieved over traditional analysis is quite significant, since it is also composed of exponential terms.

We also note that the identification of program analysis points is a common feature of many concurrency analysis techniques, such as dynamic analysis, formal verification etc. Now the idea of classifying analysis points may be extended to these analysis technique also and hence the idea of apportioning may be generalized as : (i) Identify and classify the program analysis points, (ii) Generate the execution models for local and global analysis and (iii) Decompose the property for checking and appropriately analyze the models.

In this paper we have shown the practical viability of the apportioning technique, by computing the number of states in the reachability graphs, for a few examples. Prototype implementation of the algorithms to enable further experimentation and to rigorously explore the practical utility of apportioning for larger programs, is being undertaken.

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] S. C. Cheung and J. Kramer. Contextual local analysis for design of distributed systems. *Journal of Automated Software Engineering*, 2(1):5–32, Mar 1995.

[3] 0. 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.

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

[5] R. Lea, C. Jacquemot, and E. Pillevesse. COOL : System support for distributed programming. *Communications of the ACM*, 36(9):37–46, Sep 93.

[6] C. E. McDowell. A practical algorithm for static analysis of parallel programs. *Journal of Parallel and Distributed Computing*, 6(3):515–536, June 1989.

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

[8] P. Wegner. Concepts and paradigms of object-oriented programming. *OOPS Messenger*, 1(1):8–87, Aug 1990.

[9] 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.

[10] W. J. Yeh and M. Young. Compositional reachability analysis using process algebra. *In Proceedings of the Symposium on Software Testing, Analysis and Verification*, pp. 49-59, ACM SIGSOFT, ACM Press, Oct 1991.