Apportioning-based Reachability Analysis of Concurrent Java Programs

Sridhar Iyer
School of Information Technology
IIT Bombay
Mumbai, INDIA
email : sri@it.iitb.ernet.in

Raghuraman R.
Dept of Computer Science
IIT Guwahati
Guwahati, INDIA
raghu@iitg.ernet.in

Anirban Majumdar
Wipro-MDC
Wipro Infotech Ltd.
Bangalore, INDIA
anirbanm@wipinfo.soft.net

Abstract

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

Apportioning is a new technique that effectively tackles this drawback of reachability analysis, in the context of concurrent object-oriented programs. It is based upon a simple but powerful idea of classification of program analysis points as local (having influence within a class) and global (having possible influence outside a class). Different choices for this classification give rise to tools that are efficient for different sub-classes of programs.

In this paper, we discuss the implementation of one of these tools, for the analysis of concurrent Java programs and present the experimental results for its efficiency.

1 Introduction

The Java programming language [1] has wide acceptance for developing concurrent object-oriented applications in distributed systems. Although the techniques for debugging concurrent programs have been extensively studied, there is little work on analysis and verification of concurrent object-oriented programs.

The techniques for debugging concurrent programs may be broadly classified into *dynamic analysis* and *static analysis* [7]. Dynamic analysis techniques correspond to *run-time* debugging and typically rely on monitoring the system, for re-creating the program execution. However, they are vulnerable to *probe effect* [3] (monitoring introduces extraneous delays, making it difficult to recapture the erroneous behavior). Static analysis techniques do not require program execution and avoid the problems due to execution monitoring. However, they cannot cater to the dynamic nature of the data and control-flow, and may report *spurious errors*, such as errors in impossible program paths.

Reachability analysis is a well-known static analysis tool for the verification of critical properties in concurrent programs, such as deadlock freedom. 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 (*reachability graph*), is constructed, thereby modeling the concurrent program as the set of all possible execution sequences [9]. A simple algorithm for the generation of reachability graphs is described in figure 1.

Reachability graph generation

C : Set of control-flow graphs /* Input */
I : Initial state for pgm execn /* Input */
R : Reachability graph /* Output */
curr, succ : reachability graph nodes
L : List of reachability graph nodes

1. R = L = I; /* Initilaization */
2. curr = L.first; Remove curr from L.
3. succ = successor(curr); Advance any component of curr to its next state as per its control-flow graph in C.
4. If succ is not already present in R, introduce succ into R, introduce edge in R from curr to succ, introduce succ into L.
5. Repeat steps 3,4 until all succ of curr are found.
6. Repeat steps 2-5 until L is empty.

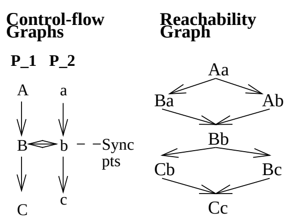

Figure 1: Traditional reachability graph generation

Reachability analysis facilitates analysis of all possible partial orderings of the concurrent execution. It provides the same level of assurance as formal verification, for properties of the synchronization structure and its practical potential for analyzing non-trivial concurrent programs has been shown in [10].

Direct application of reachability analysis to concurrent object-oriented programs is not *safe*, in the sense that it may not 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 to 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"; the complexity is typically $O(p)^T$, where T is the number of participating processes and p is the size of the control-flow graph for any given process.

In [6] we reported the development of a new technique called *apportioning*, to effectively tackle this drawback. Apportioning relies on a classification of synchronization points in any method into either *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. Given an abstract representation of the program along with a classification of its analysis points, an apportioning-based reachability analysis tool does the following :

(a) Generates (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.

(b) Generates the reachability graphs for each of these reduced classes and the reduced program.

(c) Partitions the error to be checked (expressed as a property of analysis points), 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). Subsequently it checks each sub-property in the appropriate reachability graph.

A high-level design for such tools is shown in figure 2 and the details are discussed in the following section.

Apportioning is a general technique that enables the generation of many tools that are *safe* for different subclasses of programs and having varying degrees of *efficiency*; different classifications for the analysis points gives rise to different tools. Classifying all the analysis points as GAP, results in a tool, OMEGA, which corresponds to the extension of traditional reachability analysis to concurrent object-oriented programs. While OMEGA is safe for all such programs, it is the least efficient of the tools developed. Another tool, ALPHA, classifies only the analysis points for method invocation across object boundaries as GAP, and all other

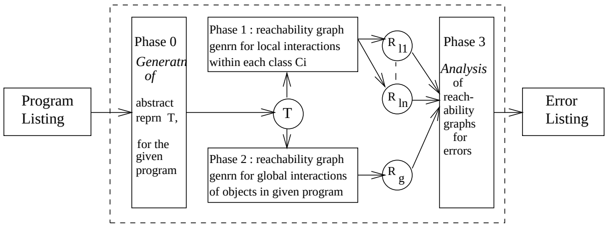

Figure 2: Apportioning-based reachability analysis

analysis points as LAP. ALPHA is the most efficient of these tools and is safe for programs not having local invocations; other local interactions, such as shared variables, are permitted. In this paper, we present the detailed design of OMEGA and ALPHA and the experimental results for comparing their efficiency.

Section 2 discusses the implementation of the tools, Section 3 presents their proofs of safety and Section 4 gives the experimental results for efficiency comparison. Section 5 outlines other apportioning-based tools and Section 6 concludes with a comparison of related work.

2 Implementation of apportioning based analysis

We have implemented prototype versions of OMEGA and ALPHA using C++, for the analysis of Java programs having the following features :

- A 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, i.e., at the entry of its *start*() 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 typically supported by means of *locks*; a *Lock* class exporting the methods *get*() and *release*() is provided. The keyword *Synchronized* may also be used for synchronization of variables.

The above model is kept simple, for ease of illustration. Extensions of the tools to support inheritance and bounded creation and destruction of objects, are discussed in section 5.

2.1 Detailed design

The detailed design of apportioning-based reachability analysis tools is shown in figure 3 and is as follows : Phase 0 creates the abstract representation (using standard compiler techniques) from the program listing. The abstract representation, 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* for providing mutually exclusive access to the data 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 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).

Phase 1 has two steps (corresponding to (a).(i) and (b) above) : For each method M, an operation called $GAP\_hiding(M)$ is applied to abstract out all non-LAP g of M, resulting in a reduced graph M'; any edge from/to g is replaced by an edge from the *nearest_LAP_predecessors* of g, to the *nearest_LAP_successors* of g. Thus $GAP\_hiding(C_i) = [\{GAP\_hiding(M_1), \ldots, GAP\_hiding(M_{\gamma_i})\}, \{\ell_1, \ldots, \ell_{\delta_i}\}]$. Subsequently for each C_i, an operation

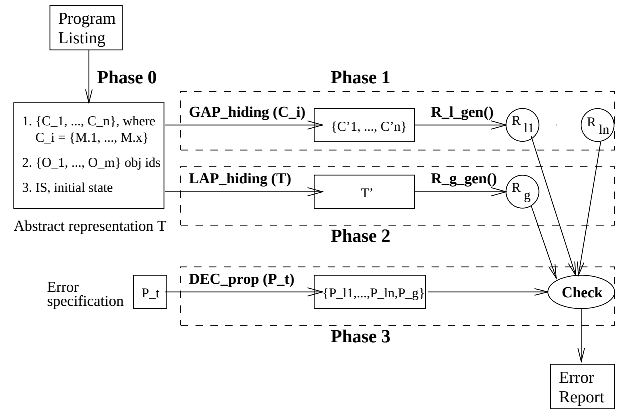

Figure 3: Detailed view of apportioning

$R_l\_gen(\{GAP\_hiding(C_i)\}, \{o_{i,1}\}, IS_{l_i})$, is applied to generate a reachability graph R_{l_i}. 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.

Phase 2 also has two steps (corresponding to (a).(ii) and (b) above) : For each method M, an operation called $LAP\_hiding(M)$ is applied to abstract out all non-GAP of M, resulting in a reduced graph M''. $LAP\_hiding(M)$ is complementary to $GAP\_hiding(M)$ described above. Now $LAP\_hiding(C_i) = [\{LAP\_hiding(M_1), \ldots, LAP\_hiding(M_{\gamma_i})\}, -]$, and $LAP\_hiding(T) = [\{LAP\_hiding(C_1), \ldots, LAP\_hiding(C_\alpha)\}, \{o_{1,1}, \ldots, o_{\alpha,\beta_\alpha}\}, IS]$. Subsequently for the given program T, an operation called $R_g\_gen(LAP\_hiding(T))$, is applied to generate a reachability graph R_g. All instances of all classes are considered and the initial node for R_g, is IS, as defined in T.

Phase 3 again has two steps (corresponding to (c) and (d) above) : For the given state property to be checked, P_t, which is a boolean formula as described above, a number of sub-properties each involving either purely LAP or GAP, are generated. For each P_t^k, the operation $DEC\_prop(P_t^k)$ decomposes P_t^k into (i) a sub-property P_g^k having the propositions of P_t^k that involve GAP. and (ii) a set of sub-properties of the form $P_{l_{i,j}}^k$, having the propositions of P_t^k that involve LAP of the methods in object $o_{i,j}$. Subsequently for each P_t^k, $P_{l_{i,j}}^k (1 \leq j \leq \beta_i)$ is checked in $R_{l_i} (1 \leq i \leq \alpha)$ and P_g^k is checked in R_g, using standard graph traversal and error-detection algorithms. P_t^k is taken to be satisfied if P_g^k holds in R_g and each $P_{l_{i,j}}^k$ holds in the corresponding R_{l_i}.

2.2 Module structure

A high-level view of the modules implemented for performing the operations described above, is as follows :

- *Main*() invokes *GetInput*() for reading the given program control-flow graph T. *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.

- *Main*() then invokes *GenLRGraph*() to generate R_{l_i} for each class C_i. *GenLRGraph*() in turn calls *HideGAP*() and *GenerateGraph*() to implement the operations $GAP\_hiding(C_i)$ and $R\_l\_gen(C_i)$, respectively. *GenerateGraph*() implements the generation of a reachability graph in a breadth-first manner. It uses *FindSuccessors*() to identify all possible successors of a given reachability graph node.

- Subsequently, *Main*() invokes *GenGRGraph*() to generate R_g for the given program T. *GenGRGraph*() implements $LAP\_hiding(T)$ and $R\_g\_gen(T)$, 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.

2.3 Input

The input to the tools is an abstract representation of the Java program to be analyzed and a specification of the error to be checked. Presently, the abstract representation is created manually by inspecting the given program. Implementation of the pre-processor (Phase 0) for automatically generating this representation from a given program, is partially complete.

Now, ALPHA employs the following classification for the analysis points of any method m_x in object O :

LAP : Analysis points of the form $m_x\_waiting\_\ell_d$, $m_x\_released\_\ell_d$, where ℓ_d is the *lock* for shared data d in O, and analysis points of the form $m_x\_before\_invocation\_self.m_y$, $m_x\_invoked\_self.m_y$ and $m_x\_after\_invocation\_self.m_y$, belong to LAP.

GAP : 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 a variable object identifier, belong to GAP.

Analysis points m_x_entry, m_x_begin and m_x_return, belong to both LAP and GAP.

OMEGA classifies all the above LAP as belonging to both LAP and GAP.

2.4 Illustrative example

We illustrate the details of the tools using a simple variation of the producer-consumer problem. 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 $first$ and $last$ are used to keep track of the items. 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.

The abstract representation, reachability graphs generated in Phase 1 and Phase 2, as well as the decompostion of a deadlock error in Phase 3, is shown in figure 4.

3 Proofs of safety

We say that '*Program S has error e*' (denoted as $S \models e$) if an execution of program S exhibits the error e. In general, reachability graphs may be analyzed for properties that do not necessarily correspond to errors. Hence, we say that a state property P holds in reachability graph R (denoted as $R \models P$), iff there exists a state s in R such that s satisfies the predicate for P (denoted as $s \models P$).

Now given a program S and a state property to be checked P, a reachability analysis tool T generates a reachability graph for S (denoted as R_s) and detects the presence of P if $R_s \models P$. T is said to *reject* S if $R_s \models P$. T is said to be *safe* if it rejects all erroneous programs, i.e., for any error e that can be represented as a state property P, $(S \models e) \Rightarrow (R_s \models P)$.

Given a program S and a state property to be checked P, an apportioning-based reachability analysis tool A apportions S into smaller units, say $S_1, \ldots, S_n$, decomposes P into sub-properties, say $P_1, \ldots, P_n$, and checks if the reachability graph for each S_i (denoted as R_{s_i}) satisfies ($\models$) the corresponding P_i. A is said to *reject* S if $R_{s_i} \models P_i$ for some S_i and P_i.

Now OMEGA may be seen to be *safe* for all programs in the chosen model, since it enumerates all possible global states of the program execution; all the LAP are classified as GAP also, hence no abstraction is performed in Phase 2 and R_g is the *traditional reachability graph* for program execution. Also in Phase 3, the sub-property P_g is the given state property P_t itself and is checked directly in R_g.

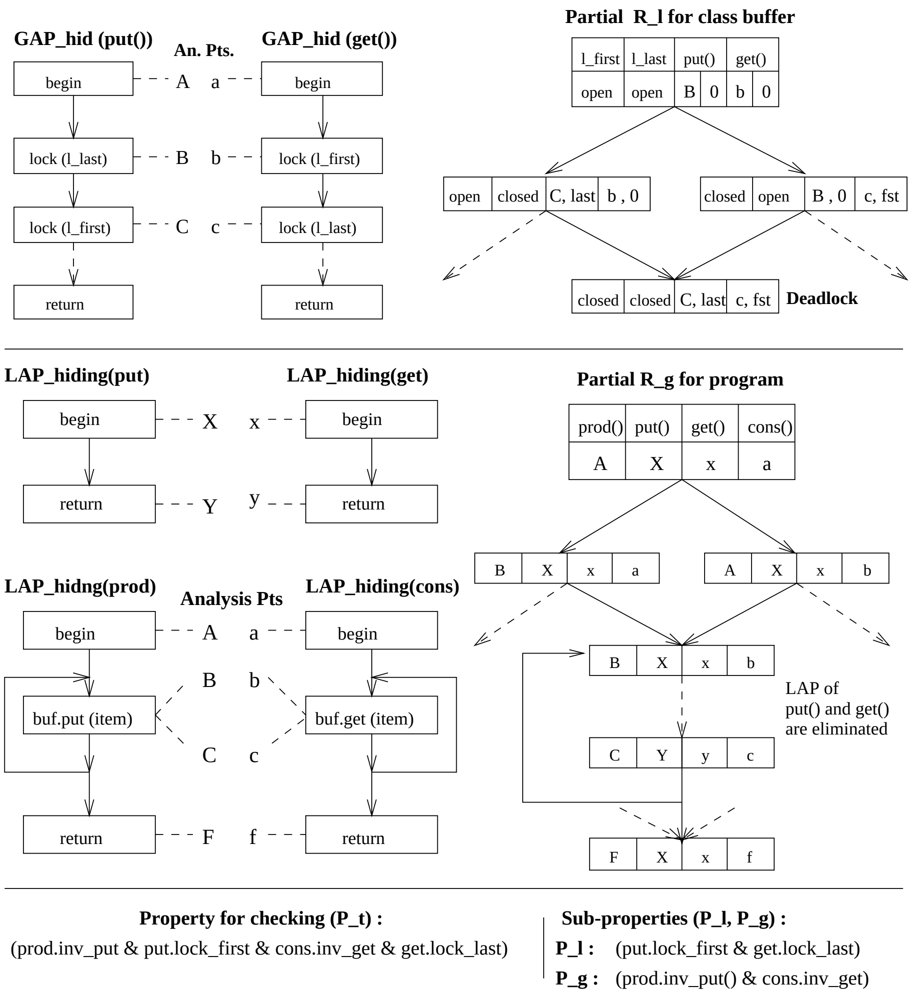

Figure 4: Illustration of ALPHA on producer-consumer example

Thus OMEGA corresponds to the direct extension of traditional reachability analysis to concurrent object-oriented programs and hence we use OMEGA as *the basis for proving the safety and comparing the efficiency* of other tools. We define the safety of other tools relative to OMEGA, as : An apportioning-based tool A is *safe* with respect to OMEGA, provided A rejects S whenever OMEGA rejects S. More formally, a given apportioning $< \{S_i | i = 1, \ldots, n\}, \{P_i | i = 1, \ldots, n\} >$ of $< S, P >$, is safe provided whenever $R_s \models P$, $R_{s_i} \models P_i$ for each i.

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.*

To prove theorem 1, let R_t denote the reachability graph generated by OMEGA, for a given program. 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.

We prove lemmas 1 and 2 by showing that for any node s in R_t, corresponding nodes exist in R_{l_i} and R_g. A more detailed treatment of the proofs, is given in [5].

4 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 their execution times on a Sun SPARC workstation running SunOS 5.7.

Table 1 gives the comparative results of ALPHA and OMEGA, for the producer-consumer problem described earlier, while Table 2 presents the results for the well-known dining philosophers problem. Table 3

Tool	1 prod 1 cons		2 prod 1 cons		2 prod 2 cons	
	No. of Nodes	Time	No. of Nodes	Time	No. of Nodes	Time
ALPHA	(36 + 39)	0.1	(216 + 39)	0.7	(1296 + 39)	10.9
OMEGA	(139 + 39)	0.4	(1161 + 39)	7.7	(9490 + 39)	5:52.8
Reduction %	57.87%	-	78.75%	-	85.99%	-

Table 1: Experimental results for the producer-consumer problem

Tool	2 phils		3 phils	
	No. of Nodes	Time	No. of Nodes	Time
ALPHA	(324 + 47)	1.1	(5832 + 47)	2:37.3
OMEGA	(660 + 47)	1.3	(16456 + 47)	15:38.3
Reduction %	47.52%	-	64.37%	-

Table 2: Experimental results for the dining philosophers problem

gives the results for a database application supporting *insert*(), *read*(), *modify*() and *delete*() operations on a sequential database file. Table 4 gives the results for a simulation of traffic movement across a bridge such that the traffic flow at any given time is uni-directional.

The *No. of Nodes* entry in each table gives the sum of the number of nodes in R_g and R_l, while the *Time* entry gives the CPU time taken in hours:minutes:seconds.tenths.

5 More apportioning-based tools

As mentioned earlier, at one end of the spectrum of apportioning-based analysis tools is OMEGA, that is safe for all programs in the model but is the least efficient, while at the other end is ALPHA, that is the most efficient but safe only for programs not having local invocations. BETA and GAMMA are other tools that lie along this spectrum and address this drawback of ALPHA.

BETA classifies the analysis points corresponding to local invocations, viz., $m_x\_before\_invocation\_self.m_y$, $m_x\_invoked\_self.m_y$ and $m_x\_after\_invocation\_self.m_y$, as belonging to both LAP and GAP. The classification of the other analysis points is the same as that of ALPHA. BETA is safe for all programs in the model. In the best case, the number of states generated by BETA is the same as that of ALPHA, while in the worst case, it is the same as that of OMEGA.

GAMMA is an optimization of BETA that classifies analysis points corresponding to only some local invocations as both LAP and GAP. A *method invocation graph* for each class, is used to identify local invocations that lead to a global invocation, either directly or transitively. Only such invocations are classified as both LAP and GAP. The classification of the other analysis points is the same as that of ALPHA. GAMMA is also safe for all programs in the model. In the best case, the number of states generated by GAMMA is the same as that of ALPHA, while in the worst case, it is the same as that of BETA.

We have also designed extensions to these tools for the analysis of programs having the following additional features :

Inheritance: Single, non-strict inheritance [8] is supported. A *subclass* may inherit data structures and methods from one *superclass* only and the subclass methods may access the data defined by the superclass, only through invocation of the methods defined in the superclass.

Dynamic Objects : Creation and destruction of objects is supported by assuming an appropriate bound on the total number of of objects in the system, at any given time. Dynamic creation and destruction are regarded as global invocations on the corresponding *constructor* and *destructor* methods provided by the class.

Prototype implementation of these tools for further experimentation, is underway.

6 Conclusions

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 [9], 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

Tool	2 users		3 users	
	No. of Nodes	Time	No. of Nodes	Time
ALPHA	(324 + 3147)	34.5	(5832 + 3147)	2:25.5
OMEGA	(1716 + 3147)	40.6	(16456 + 3147)	6:09:17.2
Reduction %	28.62%	-	54.20%	-

Table 3: Experimental results for the database application

Tool	2 vehicles		3 vehicles		4 vehicles	
	No. of Nodes	Time	No. of Nodes	Time	No. of Nodes	Time
ALPHA	(4 + 6)	0.1	(8 + 6)	0.1	(16 + 6)	0.1
OMEGA	(27 + 6)	0.1	(108 + 6)	0.3	(405 + 6)	0.3
Reduction %	69.70%	-	87.72%	-	94.65%	-

Table 4: Experimental results for the traffic simulation

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 quite significant.

Some of the other recent techniques proposed for tackling state-explosion are, compositional reachability analysis using process algebra [11] and the incorporation of local context constraints into compositional reachability analysis [2]. These techniques are applicable for the analysis of Ada-like programs and allow one to analyze individual portions of a large system and hierarchically compose the partial analysis results. 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. Also, it is not clear if these techniques exploit the similarities among subsystems to avoid repeated analysis. Recently some work towards slicing of concurrent object-oriented programs has also been carried out [12]. Slicing is however more useful for program debugging and reverse engineering rather than for property verification.

Finally, we observe that the execution time of apportioning-based tools may be improved by efficiently indexing the generated states, as well as by distributing the generation of the different reachability graphs to different machines, on a distributed computing platform such as PVM [4]. Although such efforts do not have any effect on the number of states generated, they are likely to increase the practical utility of the tools.

References

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

[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] J. Gait. A debugger for concurrent programs. *Software - Practice and Experience*, 15(6):539–554, June 1985.

[4] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek and V. Sunderam. PVM: Parallel Virtual Machine - A user's guide and tutorial for networked parallel computing. *MIT Press*, 1994.

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

[6] S. Iyer and S. Ramesh. Analysis of concurrent object-oriented programs. *In Proceedings of International Conference on Information Technology* Tata-McGraw Hill, Bhubaneshwar, India, 1998.

[7] C. E. McDowell and D. P. Helmbold. Debugging concurrent programs. *ACM Computing Surveys*, 21(4):593–622, Dec 1989.

[8] A. Taivalsaari. On the notion of inheritance. *ACM Computing Surveys*, 28(3):438–479, 1996.

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

[10] 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.

[11] 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.

[12] J. Zhao. Slicing of Concurrent Java Programs, *In Proceedings of the Seventh IEEE conference on Program Comprehension*, May 1999.