Apportioning: A technique for efficient reachability analysis of concurrent object-oriented programs

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

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

Abstract

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

Reachability analysis is an important and well-known tool for static analysis of critical properties in concurrent programs, such as deadlock freedom. It involves the systematic enumeration of all possible global states of program execution and provides the same level of assurance as formal verification, for properties of the synchronization structure in 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 safe and efficient reachability analysis of concurrent object-oriented programs, that 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). Given a program and a classification of its analysis points, reachability graphs are generated for (i) an abstract version of each class in the program having only local analysis points and (ii) an abstract version of the whole program having only global analysis points. The error to be checked is decomposed into a number of sub-properties, which are checked in the appropriate reachability graphs.

Different choices for the classification of analysis points, provide the flexibility to have many algorithms that are safe and efficient for different sub-classes of programs. We have developed a number of apportioning-based algorithms, having different degrees of safety and efficiency. In this paper, we present the details of one of these algorithms, formally show its safety for an appropriate class of programs and present experimental results to demonstrate its efficiency for various examples.

Keywords

Concurrent programs, Object-oriented programming, Static analysis, Reachability analysis.

1 Introduction

The object-oriented paradigm provides support for the construction of modular and reusable program components, and has proved to be useful for the design of large and complex systems. One important feature of this paradigm is that classes may be exported to libraries and programs may incorporate these classes on the basis of their functionality alone; a class is an encapsulation of some private state information (data) and a set of procedures (methods) that manipulate the data, while an *object* is an execution instance of a class [29]. Concurrency is incorporated into object-oriented systems by having multiple threads, that execute the same/different methods of the same/different objects. The concurrent object-oriented paradigm has been found to be useful for describing large distributed systems [1], but there is little work on the analysis and verification of concurrent object-oriented programs [6]. Recently some work towards slicing of concurrent object-oriented programs has been carried out [34, 33]; slicing is useful for program debugging and reverse engineering rather than for property verification.

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 ([27], [5]). 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.
```

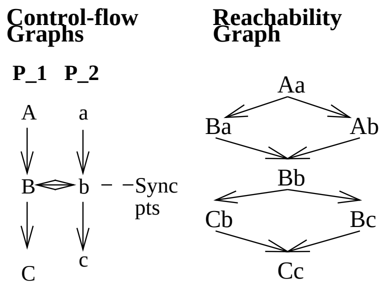

Figure 1: Traditional reachability graph generation

Reachability analysis avoids the problems associated with dynamic (run-time) analysis of concurrent program execution and facilitates analysis of all possible partial orderings of program execution [22]. For properties of the synchronization structure in concurrent programs, reachability analysis provides the same level of assurance as formal verification [31], and its practical potential for analyzing non-trivial concurrent programs has been shown in [5] and [31].

To the best of our knowledge, there is no literature on the application of reachability analysis to concurrent object-oriented programs. Direct application of traditional reachability analysis to concurrent object-oriented programs is not *safe*, since it may fail to capture all the errors in classes that are exported to libraries. This is because traditional reachability analysis deals only with the sequence of interactions present in the given program and does not consider other possible interactions within the methods of an exported class. 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 number of states generated for analysis increases exponentially with the number of concurrent execution threads).

In this paper, we propose a new technique called *apportioning*, for performing *safe* and *efficient* reachability analysis of concurrent object-oriented programs. Apportioning relies on a classification of program analysis points into local (having influence within a class) and global (having possible influence outside a class). The given program is *apportioned* into (i) an abstract version for each of its classes, containing only local analysis points (LAP) for the methods, and (ii) an abstract version of the whole program containing only global analysis points (GAP). 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 standard techniques.

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 ([27], [20]) is $O(p)^T$, where T is the number of threads and p is the number of interactions for any thread. As discussed subsequently, extending such techniques to concurrent object-oriented programs by performing additional analysis for each class, results 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. We provide formal proofs of *safety* and experimental results to demonstrate the *efficiency* of apportioning-based reachability analysis.

An analysis technique besides being *safe* and *efficient* should also be *effective*, in the sense that it should not reject too many correct programs. Generally, effectiveness and efficiency are inversely proportional; a less effective solution is more efficient. 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.

Apportioning is a general technique that enables the generation of many algorithms that are *safe* for different classes of programs and having varying degrees of *efficiency* and *effectiveness*; different strategies for classification of the analysis points into LAP and GAP lead to different algorithms. We have developed a number of apportioning-based algorithms, having different degrees of safety, efficiency and effectiveness.

In this paper, we present the details of one of these algorithms, which we call ALPHA, and outline the development of other algorithms. ALPHA is applicable for the analysis of programs that (i) define classes (directly as well as through inheritance), (ii) export the defined classes to libraries for use by other programs and (iii) define an application using these classes. We have developed formal criteria for the safety and effectiveness of reachability analysis algorithms and have shown the safety of ALPHA for concurrent object-oriented programs not having local invocations; other local interactions, such as shared variables, are permitted. We have implemented a prototype version of ALPHA for analysis of concurrent programs written in a subset of the Java programming language and we present experimental results to demonstrate its efficiency and effectiveness.

We use the term reachability analysis to mean reachability analysis techniques a' la [27], in which finite-state models of program execution, extracted from program source code, are used for the checking of specific properties of the synchronization structure, such as freedom from deadlock, mutual exclusion, etc. Reachability analysis for high-level design and protocols ([25], [14]) as well as the general approach of using a powerful language like temporal logic as advocated in [23], have been extensively studied but are not addressed here. However we believe that the ideas and approach proposed in this paper can be extended to such techniques also.

2 Preliminaries

Concurrent program execution implies the existence of multiple processes that communicate with each other by means of appropriate channels. The processes may execute on the same or different processors and communicate by a variety of means, ranging from shared memory to sending messages on the physical network connecting the processors [1].

The non-determinism and synchronization involved in the execution of concurrent programs, may give rise to *race* conditions between the interacting processes. Hence in addition to errors found in sequential programs, concurrent programs may have *data access errors*, such as violation of mutual exclusion during simultaneous access of shared variables, *synchronization errors*, such as deadlocks and *temporal errors*, such as calls to terminated program components.

Also, as a result of the non-determinism, concurrent programs may have different execution sequences at different times. Hence the undesirable behavior may not appear for all executions. Attempts to re-create the execution instance by monitoring the program, introduces extraneous delays (known as the *probe effect* [11]) and contributes to the difficulty in reproducing the erroneous behavior.

Analysis techniques for concurrent programs may be broadly classified into :

Static analysis : These techniques do not require program execution and perform *compile-time* analysis for critical program properties. These include dataflow analysis [10], reachability analysis [31], and formal verification [23].

Dynamic analysis : These techniques require program execution and correspond to *run-time* debugging of the program behavior. These range from simple extensions of sequential techniques [4], to complex techniques that rely on monitoring the system, to re-create the program execution [22], or to ensure conformance with pre-specified intended program behavior [15].

Approaches to combine both static and dynamic analysis have also been studied [9].

Static analysis techniques avoid the problems arising due to execution monitoring, such as the probe effect and management of monitoring data. They may be easily automated and can cater to analysis of specific execution paths. However since the underlying program details are abstracted away, static analysis cannot cater to the dynamic nature of the control-flow and may report *spurious errors*, such as errors in impossible program paths, that do not correspond to any real error during program execution. Also, exact analysis for dynamic data values, is beyond the scope of static analysis.

Dynamic analysis techniques do not report any spurious errors and can cater to the dynamic nature of the control-flow and data values during program execution. However they are vulnerable to the probe effect and are unable to satisfactorily deal with the non-determinism in concurrent programs [22]. Also, when the execution is monitored against a pre-specified intended behavior, additional difficulty is caused due to delays involved between occurrence of an event, its recognition and reaction to the event. Also, analysis of a single monitored execution instance, is insufficient to provide strong assurance of the program properties.

In this paper, we focus on reachability analysis, which is discussed in detail in the following section.

3 Reachability analysis

One of the earliest works in reachability analysis is the *concurrency history* technique proposed by Taylor [27], based on enumeration of all possible execution paths, for analysis of programs having rendezvous-like synchronization. The program call graph representing the invocation structure of subprograms, and the program flow-graph annotated with the synchronization constraints, are used to generate a state-transition graph called the *concurrency history*, for program execution. Each state in the concurrency history is an ordered tuple, indicating the synchronization point reached by each task in the program. The edges in the concurrency history correspond to one or more tasks making a transition from one synchronization point to the next and indicate which states may follow each other during program execution. Once the complete history is generated, all possible rendezvous are determined, all possible infinite waits are detected and parallel actions are determined.

Apt [2] applies the same enumerative technique in a formal language theory framework, for proving the correctness of CSP programs. Appelbe and McDowell [5], employ a concurrency history technique for the detection of data access errors in Fortran programs with shared memory based concurrency extensions. Young et. al. [31], describe an application of reachability analysis to Ada programs, wherein the reachability graph is checked against a specification of acceptable behaviors and violations of the specification are reported.

As noted earlier, the outcome of any static analysis technique is approximate, since the analysis is done at compile-time. It has only limited capability for dealing with dynamic references and may report *spurious errors* that are the result of infeasible program paths and do not correspond to real errors in the program. An approach based on symbolic execution, for detecting infeasible execution paths, is described by Young and Taylor [30].

3.1 Complexity of reachability analysis

The main drawback of reachability analysis is the exponential complexity due to "state-explosion", i.e., the size of the reachability graph grows as the product of the sizes of the individual control-flow graphs.

Typically the analysis complexity is $O(p)^T$, where T is the number of participating processes and p is the size of the control-flow graph for any given process. It may be easily seen that exhaustive reachability analysis for large concurrent programs is not practical and effective strategies for mitigating the analysis complexity are essential.

In general, the techniques for mitigating the exponential complexity of reachability analysis can be broadly classified as : *abstraction-based* and *partitioning-based.* Abstraction hides out some details about states, resulting in identification of states that are different from the original model, consequently reducing the size of the model [8]. Partitioning decomposes the model into its smaller components, so that analysis of the large model is replaced by individual analysis of smaller models [12].

McDowell [20] employs an abstraction approach for "folding" many states in a reachability graph into a single state, applicable to programs composed of identical sets of tasks. Valmari [28] describes a technique of partial ordering that avoids the generation of all paths formed by the interleaving of the same set of transitions.

Taylor [27] employs a partitioning approach for "parceling" the analysis into biconnected components, applicable to programs composed of disjoint sets of tasks. Yeh and Young [32] describe an approach called *compositional reachability analysis* that allows one to analyze individual portions of a large system and hierarchically compose partial analysis results. The technique is based on process algebra, and is applicable for analysis of Ada-like programs. Reachability graph representations of subsystems are independently derived and hierarchically combined to form representations of successively larger systems. Restriction and hiding operations in the algebra are used to isolate internal details of the subsystems. Graf and Steffen [13] propose a technique of *context constraints* to abstract behavior restrictions imposed on a subsystem by its neighboring processes due to the need for co-ordination. Cheung and Kramer [7] describe a technique for automatically deriving the context constraints and incorporating them into compositional reachability analysis, thereby improving its efficiency.

Other non-enumerative approaches include symbolic techniques such as the constrained expression approach, that represents possible interactions between tasks by sets of inequalities [3] and the use of binary decision diagrams [21].

To the best of our knowledge, the use of reachability analysis techniques for the analysis of concurrent object-oriented programs has not been investigated. In the next section we give an overview of the apportioning technique developed in this paper.

4 The apportioning technique

Apportioning integrates the ideas of *partitioning* [12] and *abstraction* [8], to mitigate the complexity of reachability analysis for concurrent object-oriented programs. 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 (referred to as an *analysis point*) is classified as :

- *Local Analysis Point (LAP)*: A *LAP* corresponds to an interaction with another method of the same object (such as, shared data access), and/or
- *Global Analysis Point (GAP)*: A *GAP* corresponds to an interaction with a method of a different object (such as, invocation).

Some analysis points (such as, method entry and method exit) may be classified as both *LAP* and *GAP*.

Now the inputs to any apportioning-based reachability analysis algorithm are :

1. An abstract representation of the program (typically in the form of control-flow graphs for the constituent classes),

2. A classification of analysis points into LAP and/or GAP, and

3. The error to be checked (expressed as a property of analysis points).

Typically, the abstract representation is an explicit input, while the classification of analysis points and the specification of errors to be checked are implicit inputs built into the algorithm.

Any apportioning-based reachability analysis algorithm proceeds as follows :

(a) Given the program control-flow graph and the classification of analysis points :

  (i) For each class in the program, the GAP are abstracted out from the corresponding class control-flow graph and a set of reduced class control-flow graphs is generated.

  (ii) For the entire program, the LAP are abstracted out from the program control-flow graph and a reduced control-flow graph is generated.

(b) The reachability graphs for each of these reduced class control-flow graphs and the reduced program control-flow graph 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.

An implementation of apportioning-based reachability analysis is shown in figure 2. Subsequent to modeling the language dependent program specification in terms of an abstract representation (Phase 0), Phases 1 and 2 model the execution of the program by means of reachability graphs. Phase 1 abstracts away the global interactions from a class and creates a reachability graph for the concurrent execution of all its methods (steps (a).(i) and (b)). Phase 2 abstracts away local interactions and creates a reachability graph for program execution (steps (a).(ii) and (b)). Finally, Phase 3 performs the analysis of the program using the reachability graphs generated in Phase 1 and Phase 2 (steps (c) and (d)). A detailed description of these phases with respect to a specific system model, is given subsequently.

4.1 Advantages of apportioning

The modularity of the apportioning phases and the clear separation of modeling from analysis have their own obvious advantages. Also, apportioning alleviates the problems in extending reachability analysis to concurrent object-oriented programs mentioned earlier, as follows :

Safety of classes exported to libraries: Phase 1 generates the complete reachability graph for the interactions within each class. Thus in addition to the interactions in the given program, all possible paths for concurrent execution of the methods, are also analyzed.

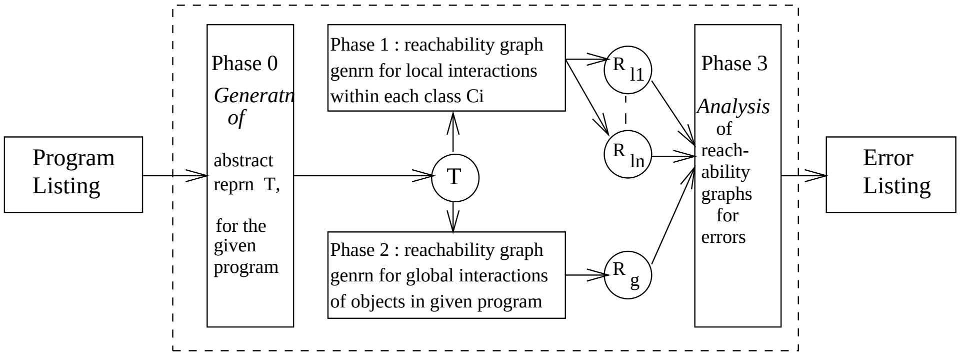

Figure 2: Apportioning-based reachability analysis

Efficient analysis of similar interactions: Phase 2 abstracts away the *LAP* before generating the reachability graph for the program. This ensures that the analysis of local interactions within the methods of an object, is performed only *once* at a class-level, and is not repeated for each object of the class.

Reduction in complexity: Typically, the complexity (number of states generated) of traditional reachability analysis for concurrent programs is, $O(p)^T$, where T is number of threads and p is the number of interactions for any thread. Extending traditional analysis to concurrent object-oriented programs by performing additional analysis for each class, results in $O(p_l)^m$ additional states being generated for each class, where m is the number of methods in each class and p_l is the number of *LAP* in any method. Hence, the complexity of performing reachability analysis of concurrent object-oriented programs along traditional lines is $O(c(p_l)^m + (p)^T)$, where c is the number of classes.

As a result of efficient analysis of similar interactions, apportioning reduces the 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 apportioned analysis is also exponential, the reduction is also composed of exponential terms, and is quite significant. Experimental results showing the practical savings due to apportioning are given in section 10.

We also make the following observations :

- Apportioning does not depend on the semantics of the interactions in the given program to mitigate analysis complexity, unlike the traditional techniques described in [27] and [20]. It exploits the inherent structure of the object-oriented paradigm, to apportion any given program at the syntactic level itself.

- Apportioning integrates ideas of partitioning-based and abstraction-based approaches, unlike techniques that are pure partitioning [12] or pure abstraction [8]. It not only partitions the program for analysis into independent sub-units but also performs abstraction on these sub-units.

- Apportioning is a general technique using which a number of analysis algorithms can be developed, by choosing an appropriate classification for the analysis points. Different algorithms are safe for different classes of programs and have varying degrees of efficiency and effectiveness. Some of these algorithms are outlined in section 6.

Apportioning is also orthogonal to techniques for tackling state-explosion (techniques that attempt to reduce T, the number of threads considered for analysis), and may be used in conjunction with such techniques.

In the next section we describe our programming model for the development of apportioning-based reachability analysis algorithms.

5 The programming model

Since reachability analysis does not require program execution, there is very little dependence on the characteristics of the underlying operating system. Hence no assumptions need to be made regarding the physical processors, their inter-connection and the communication channels.

5.1 Characteristics

Some standard concurrent object-oriented features [29] supported by the model are :

- A program consists of a set of class definitions and a *main* routine. A class defines some data structures and a set of methods (*private* as well as *public*). An object is an execution instance of a class. Each class provides a *constructor* method that is invoked to create and initialize an object of that class. The *main* routine invokes appropriate objects to start program execution.

- Program execution follows the *passive object model*, i.e., a thread of execution enters an object when any of its methods is invoked and returns when the method is completed; a thread may leave the object temporarily upon subsequent invocation of some other object.

- Objects support *medium-grain* concurrency, i.e., different methods of an object may execute concurrently. Multiple concurrent invocations of the *same* method are serviced sequentially in non-deterministic order.

- Method invocation follows the *client-server model* with waiting semantics, i.e., invocations are synchronous and the invoking method is blocked until the return of invocation. The parameters for the call are passed-by-value.

For easy illustration of the apportioning technique, we make the following assumptions :

- Each object in the system has a unique identifier. We denote by $o_{i,j}$, the object corresponding to the j^{th} instance of class c_i, and $o_{i,j}.m_x$ is the unique identifier for method m_x in object $o_{i,j}$.

- Objects are *static* entities, i.e., *all* objects are created at the start of program execution and the number of objects remains fixed thereafter. Extension to support dynamic creation and destruction of objects is discussed in section 11.

- The number of execution threads is fixed. At the start of program execution, the *main* routine creates a fixed number of threads (using appropriate constructs) and invokes relevant objects to start concurrent execution. The program terminates when all threads return to *main* and terminate.

- Mutually exclusive access to shared data among an object's methods is by means of *locks*. For any arbitrary data structure d, the declaration $LOCK_d$ implicitly creates a *lock_object*, having two methods *lock* and *release* associated with it. A method may access d only after 'acquiring' the corresponding *lock*, i.e., after invoking the *lock* method provided by $LOCK_d$.

- A method of an object may invoke another method of the same object. We use the term *local invocation* to denote those invocations that are performed using explicit *self-reference*. However, there is no inheritance among classes. Extension to support inheritance is discussed in section 11.

The above model is kept simple, for ease in illustrating the apportioning technique. Nevertheless, it is powerful enough to encompass a variety of concurrent object-oriented systems, such as COOL [19] and Amoeba [24]. Prototype implementation and experimental results for apportioning-based analysis of concurrent programs written in a subset of the Java programming language are discussed in section 10.

5.1.1 Typical program errors

The model supports concurrency at the *object-level*, i.e., concurrent execution of methods in different objects, as well as concurrency at the *method-level*, i.e., concurrent execution of different methods in the same object. Hence the concurrent execution may give rise to errors such as : (i) *data access errors* due to access of shared data without appropriate *locks* and method completion before *release* of acquired locks, (ii) *synchronization errors* due to cyclic waiting among methods for acquisition of locks and return of local or global invocations, and (iii) *temporal errors* due to incorrect terminations. Some of these program errors are illustrated in figure 3.

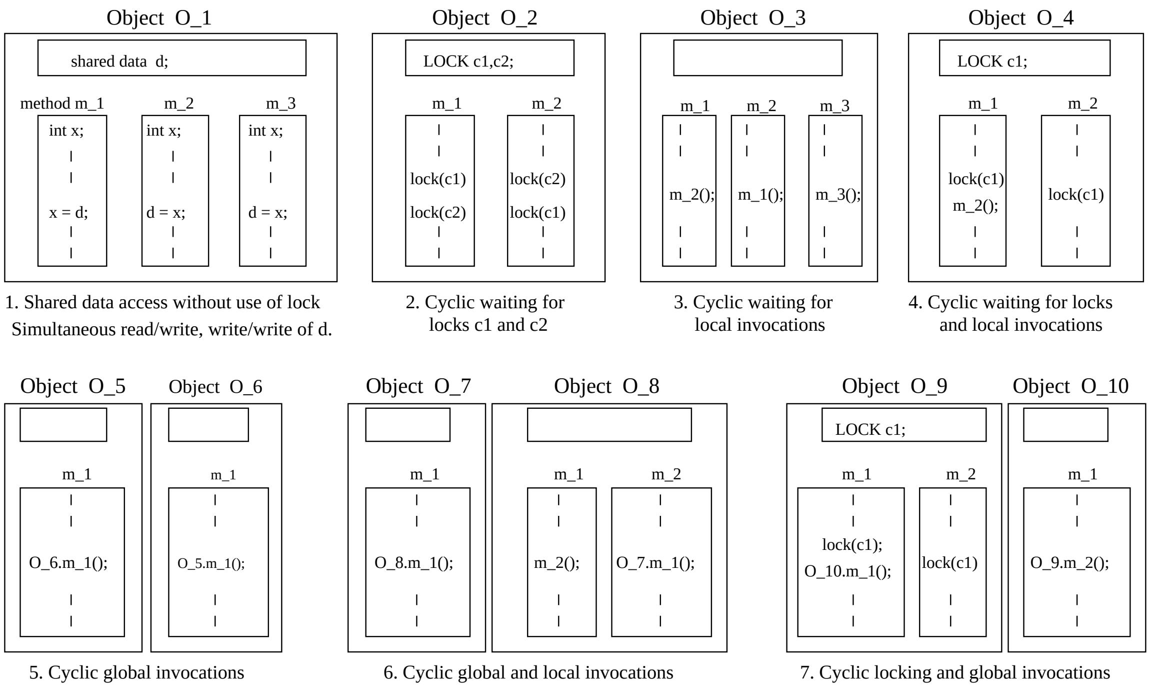

Figure 3: Typical program errors

5.1.2 Illustrative example : Producer-Consumer

For concrete illustration during the development of apportioning-based reachability analysis algorithms, we use as a running example, a simple variation of the producer-consumer problem (see figure 4). 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 are used to keep track of the items. The methods *put*() and *get*() of the buffer, access these shared variables by means of appropriate *locks*. The *main* routine creates the producer, consumer and buffer objects by implicit declaration, creates threads and concurrently invokes *produce*() and *consume*(). A thread of execution in *produce/consume*(), leaves it upon invocation of *put/get*(), performs the *put/get*() operations on the buffer and returns to *produce/consume*() upon completion. The thread returns to *main* and terminates when the buffer is full/empty.

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.

5.2 Abstract representation

Informally, a program is represented for analysis as a *program control-flow graph* which consists of : (i) A set of *class control-flow graphs*, one corresponding to each class in the program, (ii) A set of identifiers for all the objects declared in the program and (iii) The initial state for

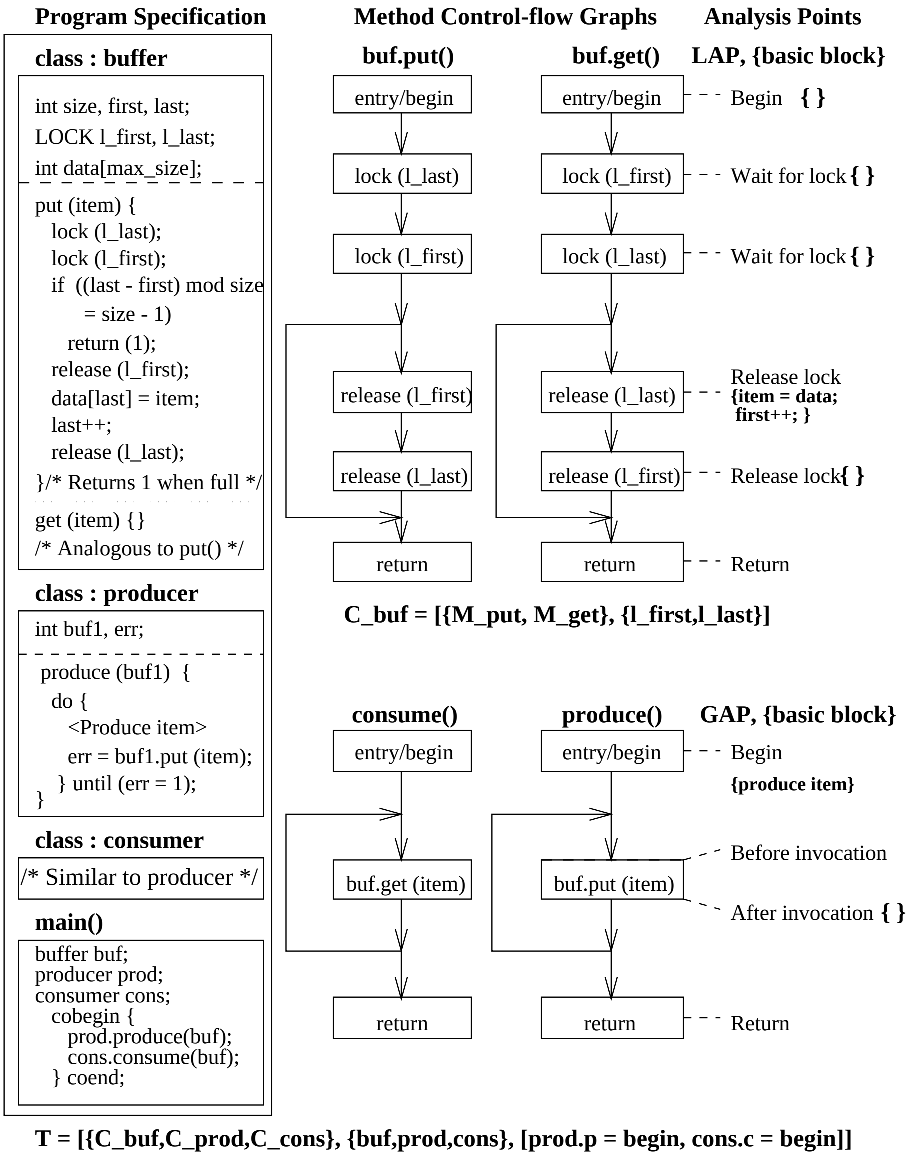

Figure 4: Program flow-graph for producer-consumer example

program execution.

Each class control-flow graph consists of : (i) A set of *method control-flow graphs*, one corresponding to each method defined in the class and (ii) a set of identifiers for all the *locks* declared in the class. Each method control-flow graph consists of : (i) A set of nodes called *analysis points*, corresponding to the various synchronization states of the method during program execution, (ii) A set of edges that capture the flow of control between the various analysis points, and (iii) Zero or more *basic blocks* associated with each analysis point to capture the actions between that analysis point and the next.

The initial state for program execution is an ordered tuple specifying the methods that are *active* at the start of program execution. Figure 4 shows the analysis points, associated basic blocks and the control-flow graphs for the producer-consumer example.

In order to formally describe the abstract representation, we use the following notations :

α	: Number of classes in the given program; $\{c_1, \ldots, c_\alpha\}$ are the classes, $\alpha \geq 1$.
β_i	: Number of instances (objects) of a given class c_i; $\beta_i \geq 1$.
γ_i	: Number of methods in a given class c_i; $\gamma_i \geq 1$.
δ_i	: Number of *locks* in a given class c_i.
$o_{i,j}$	: Object corresponding to the j^{th} instance of class c_i.
ℓ_d	: The *lock* for shared data d, accessed by methods of a class.

5.2.1 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.

- 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 parameters (variables) in m_x would be bound to actual values (object identifiers) at this point.

- m_x_return : This corresponds to the return of invocation from m_x, just before control is transferred back to its invoker. Multiple return points are merged to form a single return point.

- $m_x\_waiting\_\ell_d$: This corresponds to a point just before m_x invokes the *lock* method of ℓ_d, the *lock_object* for shared data d. If ℓ_d is already 'acquired' by some other method, then m_x waits till ℓ_d is released.

- $m_x\_released\_\ell_d$: This corresponds to a point just after m_x invokes the *release* method of ℓ_d. Any one of the other methods waiting for ℓ_d may now acquire ℓ_d.

- $m_x\_before\_invocation\_o.m_y$: This corresponds to a point just before m_x invokes method m_y of the object denoted by the identifier o, resulting in a transfer of control to m_y. If m_y is already invoked by some other method, then m_x waits till m_y returns to its *entry* point.

- $m_x\_invoked\_o.m_y$: This corresponds to m_x having invoked m_y and *waiting* for its invocation to return from m_y.

- $m_x\_after\_invocation\_o.m_y$: This corresponds to a point just when control is transferred back to m_x, after the invocation returns from m_y.

5.2.2 Basic blocks

Associated with each analysis point is a set of basic blocks, containing zero or more elements, corresponding to the actions between that analysis point and the next. Analysis points m_x_entry, m_x_return, $m_x\_before\_invocation\_o.m_y$ and $m_x\_invoked\_o.m_y$ have zero basic blocks associated, while analysis points m_x_begin, $m_x\_waiting\_\ell_d$, $m_x\_released\_\ell_d$ and $m_x\_after\_invocation\_o.m_y$ may have more than one basic block associated with them. In case of loops in the control-flow, the same basic block may be associated with more than one analysis point.

5.2.3 Method control-flow graph

The method control-flow graph for any method m_x is $M_x = (N_x, E_x)$ where,

- N_x is a set of nodes, each corresponding to an analysis point (with its associated basic blocks) in m_x.
- E_x is a set of directed edges, such that there is an edge from node n_1 to n_2, if n_2 follows n_1 in the control-flow and there is no other analysis point between n_1 and n_2.

For any node a in M_x, there is a path from m_x_begin to a and from a to m_x_return. M_x captures all possible control states in the execution of m_x.

5.2.4 Class control-flow graph

The class control-flow graph for any class c_i is $C_i = [\{M_1, \ldots, M_{\gamma_i}\}, \{\ell_1, \ldots, \ell_{\delta_i}\}]$, where

- $\{M_1, \ldots, M_{\gamma_i}\}$ is the set of method control-flow graphs for the methods in c_i.
- $\{\ell_1, \ldots, \ell_{\delta_i}\}$ is the set of identifiers of the *locks* in c_i.

C_i captures all the relevant information for any object of class c_i.

5.2.5 Program control-flow graph

The program control-flow graph for the given program (denoted as T) is
$T = [\{C_1, \ldots, C_\alpha\}, \{o_{1,1}, \ldots, o_{1,\beta_1}, \ldots, o_{\alpha,1}, \ldots, o_{\alpha,\beta_\alpha}\}, IS]$, where

- $\{C_1, \ldots, C_\alpha\}$ is the set of class control-flow graphs for the classes defined and pre-defined library classes referenced by the program.
- $\{o_{1,1}, \ldots, o_{\alpha,\beta_\alpha}\}$ is the set of all object identifiers in the program.
- IS is initial state for program execution and is an ordered tuple $\ll o_{i,j}.m_x \gg$, where $o_{i,j}.m_x$ is any method that is *active* at the start of program execution.

T captures all relevant information for the generation of reachability graphs, for all programs in the chosen model.

5.2.6 Phase 0: Generation of the program control-flow graph

Phase 0 is the preprocessing phase that generates the program control-flow graph T for the given program and has the following steps :

1. Generation of a *method control-flow graph* for each method by identifying its *analysis points* and associating *basic blocks* with each analysis point.

2. Generation of the *class control-flow graph* for each class by identifying the *locks* and methods declared in the class.

3. Generation of the *program control-flow graph* by identifying the *objects* and classes declared in the program and determining the *initial state* for program execution.

Since objects are *static* entities, the set of object identifiers is easily determined by examining the declarations in the *main*() routine. The initial state is also determined by examining the invocations in the *main*() routine.

Phase 0 is implemented as a preprocessing phase to the analysis, using standard compiler-writing techniques. The details of Phase 0 are not addressed here since they are fairly straightforward and do not form the main focus of this paper. Phase 0 is the only language dependent part of the analysis and makes the subsequent phases independent of the underlying system details, as well as the program listing.

5.3 Specification of errors

We focus 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.

We capture the global state of program execution in terms of analysis points of methods, for all the objects in the program. For every analysis point a in a method m_x, we associate a proposition m_x_AT_a, which is TRUE when the execution of m_x is at a. Now any (error) property of the global state can be expressed as a boolean formula over such propositions. We assume errors to be formulae in *disjunctive normal form* involving only *positive* propositions. More precisely, suppose P_t is such a formula, then :

- $P_t = P_t^1 \vee P_t^2 \vee \ldots \vee P_t^\epsilon$, where ϵ is the number of disjunctions.

- Each $P_t^k (1 \leq k \leq \epsilon)$, is a conjunction of propositions, $P_t^k = P_t^{k_1} \wedge P_t^{k_2} \wedge \ldots \wedge P_t^{k_\theta}$, where θ is the number of conjunctions.

- Each proposition $P_t^{k_p} (1 \leq p \leq \theta)$ is of the form $o.m\_AT\_a$, where o is an object identifier, m is a method identifier and a is an analysis point.

Note that the use of negation-free propositions is not restrictive for the following reason : a given program has only a finite number of analysis points and any negative proposition can be expressed as disjunctions of positive propositions involving other analysis points.

As noted earlier, any apportioning-based algorithm hinges on the classification of analysis points into LAP and GAP. The right choices for classification leads to *safety* (all errors detected by a non-apportioned algorithm are detected), *efficiency* (reduced complexity) and *effectiveness* (no additional spurious errors are reported). In the next section, we describe some of the apportioning-based algorithms developed and subsequently present the details of one of them.

6 Apportioning-based algorithms

Varying the classification of analysis points into LAP and GAP, enables the generation of several apportioning-based algorithms, that are safe for different classes of programs and having different degrees of efficiency and effectiveness. A brief description of some of these algorithms is as follow :

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 (including those corresponding to $locks$ and local invocations), are classified as both LAP and GAP.

Since all the LAP are also classified as GAP, no abstraction is performed in Phase 2 and the reachability graph generated is the same as the *traditional reachability graph* for program execution. Thus *OMEGA corresponds to the direct extension of traditional analysis to concurrent object-oriented programs.*

OMEGA is further discussed in section 8.2. We show that OMEGA is safe for all programs in the chosen model and the most effective of the algorithms developed; however, it is the least efficient. We use OMEGA as the basis for comparing the efficiency and effectiveness of the other apportioning-based algorithms.

ALPHA : ALPHA classifies analysis points corresponding to $locks$, i.e., analysis points of the form $m_x\_waiting\_\ell_d$, $m_x\_released\_\ell_d$ and analysis points corresponding to local invocations, i.e., analysis points of the form $m_x\_before\_invocation\_self.m_y$, $m_x\_invoked\_self.m_y$, $m_x\_after\_invocation\_self.m_y$, as LAP. Analysis points corresponding to global invocations are classified as GAP. Only analysis points of the form m_x_entry, m_x_begin and m_x_return, are classified as being both LAP and GAP.

ALPHA is safe for the sub-class of programs that do not have local invocations and is the most efficient of the algorithms developed. A detailed discussion of ALPHA is given in the following sections.

BETA : BETA classifies the analysis points corresponding to *locks* as LAP, and analysis points corresponding to global invocations as GAP. All other analysis points (including those corresponding to local invocations), are classified as both LAP and GAP.

In [16], we have shown that BETA is safe for all programs in the chosen model, and is more effective but less efficient than ALPHA.

GAMMA : GAMMA is an optimization of BETA, and its classification of analysis points differs from that of BETA as follows : Instead of classifying *all* the analysis points corresponding to local invocations as both LAP and GAP, only those local invocations that directly or indirectly result in a global invocation, are classified as both LAP and GAP. An appropriate *method invocation graph* is generated and used to determine such invocations.

BETA and GAMMA are mentioned here only to illustrate how different choices in the classification of analysis points lead to different characteristics in the algorithms. Detailed discussions of BETA and GAMMA, and their proofs of safety and effectiveness, are beyond the scope of this paper and may be found in [16].

Also, different algorithms being safe for different sub-classes of programs, given a program for analysis, an appropriate algorithm may be chosen by determining the sub-class to which the given program belongs. An approach for incorporating apportioning-based algorithms into a tool-suite to enable choice of an 'ideal' analysis tool for a given program, is described in [17].

In the following sections we discuss the detailed design of ALPHA, formally prove its safety and evaluate its efficiency.

7 Detailed design of ALPHA

The input to ALPHA is a program control-flow graph T and an error property P_t. ALPHA classifies the analysis points in any method m_x as follows :

LAP : Analysis points of the form $m_x\_waiting\_\ell_d$, $m_x\_released\_\ell_d$, 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 : 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.

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

An overview of the different phases of analysis has been presented in section 4 (figure 2). A more detailed view of these phases is given in figure 5 and we now describe them in detail. We use the following notations :

M	: A method control-flow graph.
a	: A node in M (analysis point).
a_0	: Node in M corresponding to $begin$.
a_z	: Node in M corresponding to $return$.
$\mathcal{N}(M)$	: Set of all nodes in M.
$\mathcal{N}_l(M)$	: Set of all nodes in M that are LAP.
$\mathcal{N}_g(M)$	: Set of all nodes in M that are GAP.

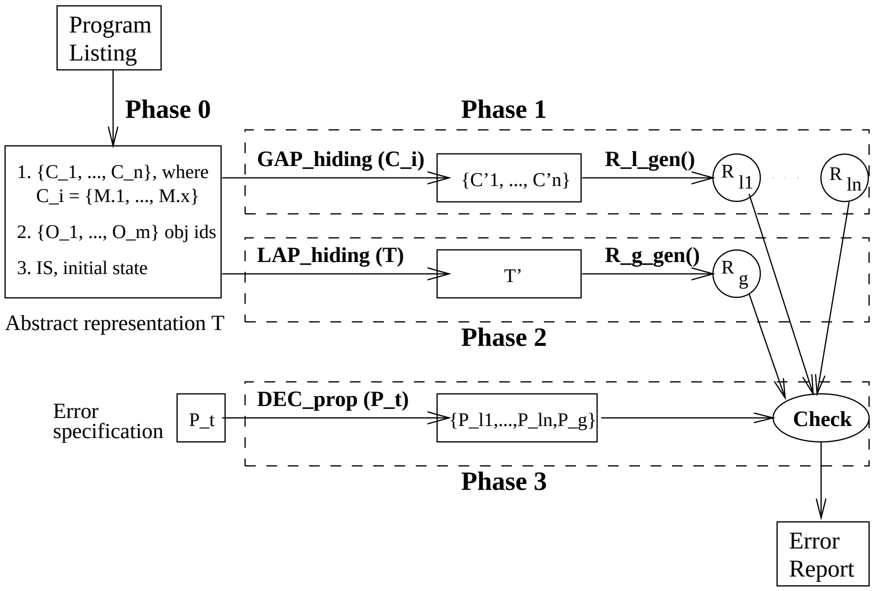

Figure 5: Detailed view of apportioning

7.1 Phase 1

Phase 1 takes the program control-flow graph T as input and produces a set of reachability graphs (denoted as R_l) as output. For each class C_i in T, Phase 1 performs the following actions :

1. Abstraction of non-LAP from the method control-flow graphs in C_i.

   For any method control-flow graph M in C_i, the operation $GAP\_hiding(M)$ (illustrated in figure 6), removes all non-LAP from M and results in a graph M' having only LAP. For any analysis point a in M, if a is an LAP then $GAP\_hiding(M)$ retains a in M'. If a is not an LAP then M' has edges from the nearest $LAP\_predecessors$ of a to its nearest $LAP\_successors$.

2. Generation of reachability graph R_{l_i}, corresponding to C_i.

   For the abstract method control-flow graphs in C_i (obtained by $GAP\_hiding()$), the operation $R_l\_gen()$ generates the reachable states for all possible interleaving of the concurrent execution of the methods in C_i and results in the reachability graph R_{l_i}. Each node in R_{l_i} has fields to record the status of each *lock* and each method in C_i, as illustrated in figure 7.

For the producer-consumer example, $GAP\_hiding()$ of the methods $put()$ and $get()$, and R_l for the class *buffer*, are shown in figure 7. A more formal description of $GAP\_hiding()$ and $R_l\_gen()$ is given below.

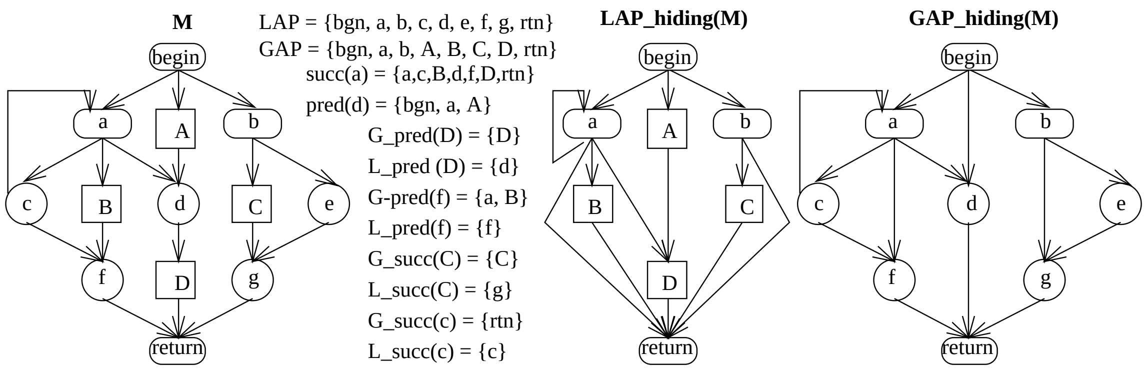

Figure 6: Operations on method control-flow graphs

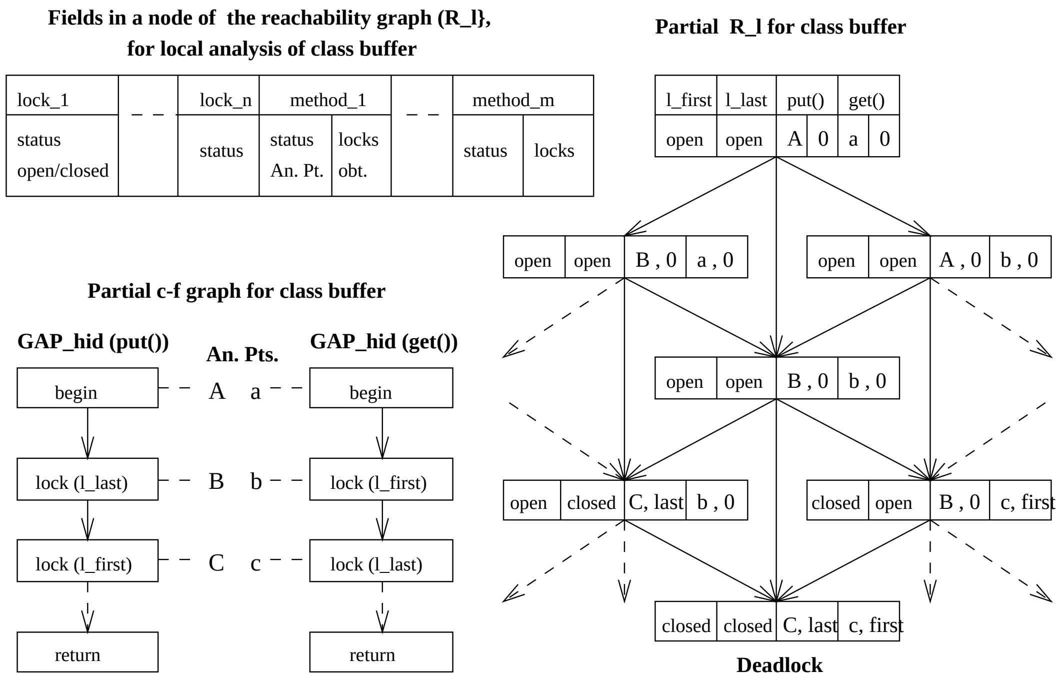

Figure 7: Reachability graph generation in Phase 1 for producer-consumer example

7.1.1 GAP_hiding()

$GAP\_hiding(M)$, the abstraction of non-LAP from M, is defined by means of the following operations :

$pred_M(a)$: For any node a in $\mathcal{N}(M)$, $pred_M(a)$ is the set of *predecessors* of a. Any node a' in $\mathcal{N}(M)$ is a *predecessor* of a if there is a path in M from a' to a. $pred_M(a_0)$ is a_0 itself and the nature of M ensures that $pred_M(a)$ is non-empty; there is a path from *begin* to any node in M.

$L\_pred_M(a)$: For any node a in $\mathcal{N}(M)$, $L\_pred_M(a)$ is the set of nearest *LAP_predecessors* of a. If a is in $\mathcal{N}_l(M)$, $L\_pred_M(a) = \{a\}$. If a is not in $\mathcal{N}_l(M)$, $L\_pred_M(a)$ is the set of all a' such that a' is in $\mathcal{N}_l(M)$, $a' \in pred_M(a)$ and there is no other node in $\mathcal{N}_l(M)$ along any path from a' to a. $L\_pred_M(a)$ can be shown to be non-empty by induction on the length of any path from a_0 to a. The details of the proof may be found in [16].

$succ_M(a)$: For any node a in $\mathcal{N}(M)$, $succ_M(a)$ is the set of *successors* of a. Any node a'' in $\mathcal{N}(M)$ is a *successor* of a if there is a path in M from a to a''. $succ_M(a_z)$ is a_z itself and the nature of M ensures that $succ_M(a)$ is non-empty; there is a path from any node in M to *return*.

$L\_succ_M(a)$: For any node a in $\mathcal{N}(M)$, $L\_succ_M(a)$ is the set of nearest *LAP_successors* of a. If a is in $\mathcal{N}_l(M)$, $L\_succ_M(a) = \{a\}$. If a is not in $\mathcal{N}_l(M)$, $L\_succ_M(a)$ is the set of all a'' such that a'' is in $\mathcal{N}_l(M)$, $a'' \in succ_M(a)$ and there is no other node in $\mathcal{N}_l(M)$ along any path from a to a''. $L\_succ_M(a)$ can be shown to be non-empty by induction on the length of any path from a to a_z. The details of the proof may be found in [16].

$GAP\_hiding(M)$: $GAP\_hiding(M)$ *abstracts* away all nodes not in $\mathcal{N}_l(M)$ and results in M'. The nodes in M' are the nodes in $\mathcal{N}_l(M)$. The edges in M' are such that, if there is an edge in M between a_1, a_2, where $a_1, a_2 \in \mathcal{N}_l(M)$, then there is an edge in M' between a_1 and a_2. For all other nodes a in M ($a \in \mathcal{N}(M) - \mathcal{N}_l(M)$), edges are introduced in M', from every node in $L\_pred_M(a)$ to each node in $L\_succ_M(a)$. Note that by definition, all nodes in $L\_pred_M(a)$ and $L\_succ_M(a)$, belong to $\mathcal{N}_l(M)$ and hence are present in M'. Multiple edges between any two nodes in M', are replaced by a single edge.

Recall that $C_i = [\{M_1, \ldots, M_{\gamma_i}\}, \{\ell_1, \ldots, \ell_{\delta_i}\}]$. Now $GAP\_hiding(C_i)$ performs $GAP\_hiding(M)$, for each M in C_i. The fields in C_i corresponding to ℓ, are retained. Hence $GAP\_hiding(C_i) = C'_i$, where $C'_i = [\{GAP\_hiding(M_1), \ldots, GAP\_hiding(M_{\gamma_i})\}, \{\ell_1, \ldots, \ell_{\delta_i}\}]$.

7.1.2 Generation of R_{l_i}

For any class C_i, the reachability graph R_{l_i} is : $R_{l_i} = R_l\_gen(\{GAP\_hiding(C_i)\}, \{o_{i,1}\}, IS_{l_i})$, where $R_l\_gen()$ is the algorithm described in figure 8, which uses the rules described in figure 9, to determine successors of nodes. $GAP\_hiding(C_i)$ is as described earlier, $o_{i,1}$ is a single instance of C_i, and IS_{l_i} the initial state for R_{l_i} is $(\forall x(m_x\_sts = entry)) \wedge (\forall d(\ell_d\_sts = open))$. Note that Rule $S1$ enables methods to make a non-deterministic transition from *entry* to *begin*, thereby capturing all possible concurrent execution.

```
R_gen(C, O, I)      /* Reachability graph generation algorithm. */
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 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) 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 */
}
```

Figure 8: Algorithm for reachability graph generation

The structure of R_{l_i} is, $R_{l_i} = (N_l, E_l)$ where,

- N_l is the set of nodes, each corresponding to an interleaving of the nodes of the method control-flow graphs that constitute C_i. Each node has fields to record the status for each *lock* and method of C_i. Field ℓ_d_sts records the status of *lock* ℓ_d as *closed* if any method has acquired ℓ_d and *open* otherwise. Field m_x_sts records the LAP reached by m_x while field m_x_lks records the set of *locks* acquired by m_x.

- E_l is the set of directed edges such that there is an edge from node n_1 to n_2, if n_2 represents a state that may be reached from n_1, by a single transition for some method

```
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; /* For Phase 1 only */
   Rule S2 : If (apt1 = begin) then apt2 = M_x.immediate_successor(apt1);
   Rule S3 : If ((apt1 = waiting_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_invocation_m_y) ∧ (m_y_sts = entry)) then
                  apt2 = invoked_m_y; next_node.m_y_sts = begin;
                  // Bind parameters of m_y, to objects gg
   Rule S6 : If ((apt1 = invoked_m_y) ∧ (m_y_sts = return)) then
                  apt2 = after_invocation_m_y; next_node.m_y_sts = entry;
   Rule S7 : If (apt1 = after_invocation_m_y) then
                  apt2 = M_x.immediate_successor(apt1);
   Rule S8 : If (apt1 = return) then apt2 = entry;
}
```

Figure 9: Rules for determining successors of reachability graph nodes

m_x in n_1, i.e., a transition in $GAP\_hiding(M_x)$ from any analysis point to one of its $immediate\_successors$.

R_l is the set of R_{l_i} for all classes C_i defined in T; $R_l = \{R_{l_1}, \ldots, R_{l_\alpha}\}$.

7.2 Phase 2

Phase 2 takes the program control-flow graph T as input and produces a reachability graph (denoted as R_g) as output, by performing the following actions :

1. Abstraction of non-GAP from the method control-flow graphs in T.
   For any method control-flow graph M in class C_i, the operation $LAP\_hiding(M)$ (illustrated in figure 6), removes all non-GAP from M and results in a graph M' having only GAP. For any analysis point a in M, if a is a GAP then $LAP\_hiding(M)$ retains a in M'. If a is not a GAP then M' has edges from the nearest $GAP\_predecessors$ of a to its nearest $GAP\_successors$.

2. Generation of reachability graph R_g.

For the abstract method control-flow graphs in T (obtained by $LAP\_hiding(T)$), the operation $R_g\_gen()$ generates the reachable states for execution of the given program and results in a reachability graph R_g. Each node in R_g has fields to record the status of each method of each object in T, as illustrated in figure 10.

For the producer-consumer example, $LAP\_hiding()$ of the methods $put()$, $get()$, $prod()$, $cons()$, and R_g for the program, are shown in figure 10. A more formal description of $LAP\_hiding()$ and $R_g\_gen()$ is given below.

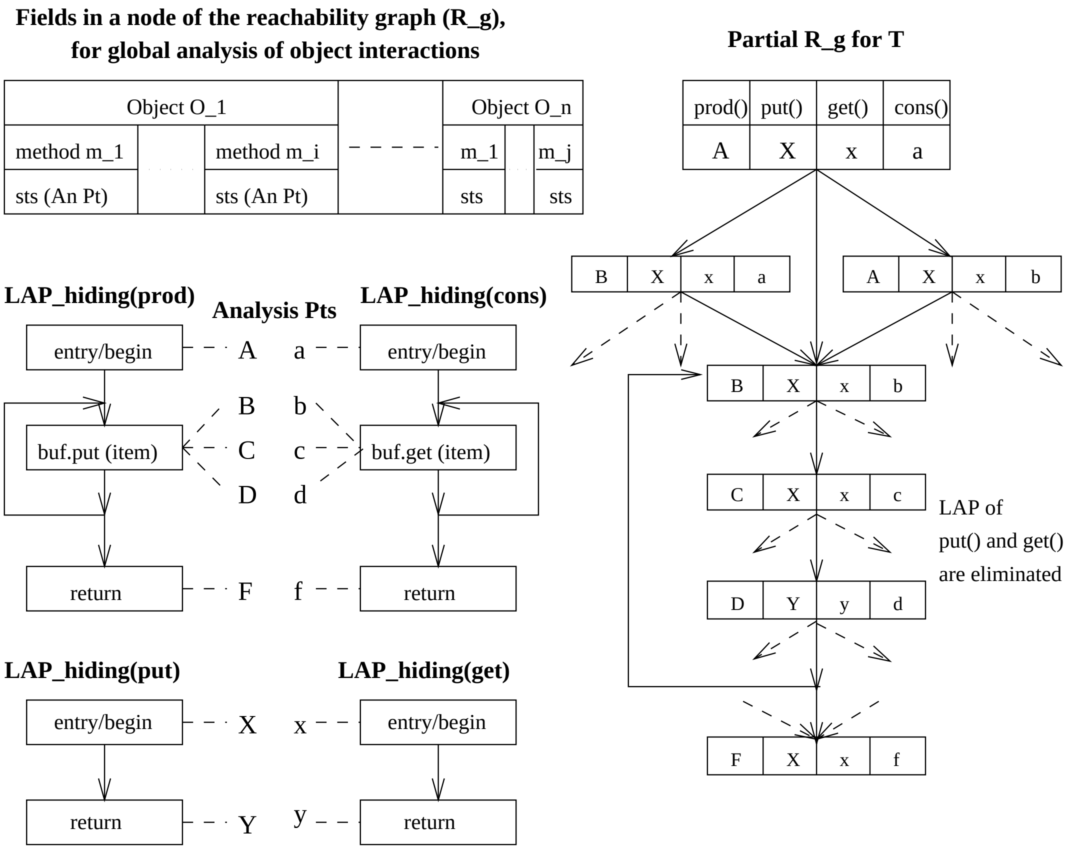

Figure 10: Reachability Graph for Phase 2 for producer-consumer example

7.2.1 LAP_hiding()

$LAP\_hiding(M)$, the abstraction of non-GAP from M, is defined by means of the following operations :

$G\_pred_M(a)$: For any node a in $\mathcal{N}(M)$, $G\_pred_M(a)$ is the set of nearest $GAP\_predecessors$ of a. If a is in $\mathcal{N}_g(M)$, $G\_pred_M(a) = \{a\}$. If a is not in $\mathcal{N}_g(M)$, $G\_pred_M(a)$ is the set of all a' such that a' is in $\mathcal{N}_g(M)$, $a' \in pred_M(a)$ and there is no other node in $\mathcal{N}_g(M)$ along any path from a' to a. $G\_pred_M(a)$ may be shown to be non-empty along the lines of $L\_pred_M(a)$.

$G\_succ_M(a)$: For any node a in $\mathcal{N}(M)$, $G\_succ_M(a)$ is the set of nearest $GAP\_successors$ of a. If a is in $\mathcal{N}_g(M)$, $G\_succ_M(a) = \{a\}$. If a is not in $\mathcal{N}_g(M)$, $G\_succ_M(a)$ is the set of all a'' such that a'' is in $\mathcal{N}_g(M)$, $a'' \in succ_M(a)$ and there is no other node in $\mathcal{N}_g(M)$ along any path from a to a''. $G\_succ_M(a)$ may be shown to be non-empty along the lines of $L\_succ_M(a)$.

$LAP\_hiding(M)$: $LAP\_hiding(M)$ *abstracts* away all nodes not in $\mathcal{N}_g(M)$ and results in M'. The nodes in M' are the nodes in $\mathcal{N}_g(M)$. The edges in M' are such that if there is an edge in M between a_1, a_2, where $a_1, a_2 \in \mathcal{N}_g(M)$, then there is an edge in M' between a_1 and a_2. For all other nodes a in M ($a \in \mathcal{N}(M) - \mathcal{N}_g(M)$), edges are introduced in M', from every node in $G\_pred_M(a)$ to each node in $G\_succ_M(a)$. Note that by definition, all nodes in $G\_pred_M(a)$ and $G\_succ_M(a)$, belong to $\mathcal{N}_g(M)$ and hence are present in M'. Multiple edges between any two nodes in M', are replaced by a single edge.

$LAP\_hiding(C_i)$ is $LAP\_hiding(M)$, for each M in C_i. The fields in C_i corresponding to ℓ_d_sts (status of *locks*) are discarded. Hence $LAP\_hiding(C_i) = C_i'$, where
$C_i' = [\{LAP\_hiding(M_1), \ldots, LAP\_hiding(M_{\gamma_i})\}, -]$.

$LAP\_hiding(T)$ is $LAP\_hiding(C_i)$, for each C_i in T. The fields in T corresponding to the set of object identifiers and the initial state are retained. Hence, $LAP\_hiding(T) = T'$, where $T' = [\{LAP\_hiding(C_1), \ldots, LAP\_hiding(C_\alpha\}), \{o_{1,1}, \ldots, o_{\alpha,\beta_\alpha}\}, IS]$.

7.2.2 Generation of R_g

For any given T, the reachability graph R_g is : $R_g = R_g\_gen(LAP\_hiding(T))$, where $R_g\_gen()$ is the algorithm described in figure 8, which uses the rules described in figure 9, to determine *successors* of nodes. $LAP\_hiding(T)$ is as described earlier. Note that Rule $S1$ is not applicable in $R_g\_gen()$.

The structure of R_g is, $R_g = (N_g, E_g)$ where,

- N_g is the set of nodes, each corresponding to the various execution states for T. Each node has a field to record the status for each object $o_{i,j}$ in T. The field for $o_{i,j}$ has subentries to record the status for each method m_x of $o_{i,j}$ and the object identifiers passed as parameters to m_x. The entry $o_{i,j}.m_x\_sts$ records the GAP reached by m_x. The entries in the set $o_{i,j}.m_x\_par$ record the object identifiers passed as parameters to m_x.

- E_g is the set of directed edges such that there is an edge from node n_1 to n_2, if n_2 represents a state that may be reached from n_1, by a single transition for some method m_x in n_1, i.e., a transition in $LAP\_hiding(M_x)$ from any analysis point to one of its *immediate_successors*.

7.3 Phase 3

Phase 3 takes the reachability graphs $\{R_{l_1}, \ldots, R_{l_\alpha}\}$ (generated in Phase 1), R_g (generated in Phase 2) and the error property P_t to be checked, as input and produces an error report, by performing the following actions :

1. Decomposition of property P_t into sub-properties.
   As defined earlier (section 5.3), P_t is a boolean formula of the form, $P_t = P_t^1 \vee P_t^2 \vee \ldots \vee P_t^\epsilon$, 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). For each P_t^k, the operation $DEC\_prop(P_t^k)$ (illustrated in figure 11), decomposes P_t^k into :

   (i) A sub-property P_g^k having only those propositions of P_t^k that involve GAP, and

   (ii) A set of sub-properties $P_l^k = \{P_{l_{1,1}}^k, \ldots, P_{l_{1,\beta_1}}^k, \ldots, P_{l_{\alpha,1}}^k, \ldots, P_{l_{\alpha,\beta_\alpha}}^k\}$, such that any sub-property in P_l^k (say $P_{l_{i,j}}^k$), has only those propositions of P_t^k that involve LAP of methods in object $o_{i,j}$.

2. Checking of sub-properties in the appropriate reachability graphs.
   For each P_t^k, the sub-property P_g^k is checked in R_g and each $P_{l_{i,j}}^k (1 \leq j \leq \beta_i)$ is checked in $R_{l_i} (1 \leq i \leq \alpha)$, using standard 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}.

For the producer-consumer example, a deadlock property involving the methods $put()$, $get()$, $prod()$ and $cons()$, and its corresponding sub-properties generated by $DEC\_prop()$, are shown in figure 11. A more formal description of $DEC\_prop()$ is given below.

For the producer-consumer example (figure 4),
Let $P_t = (prod\_AT\_invoked\_buf.put) \wedge (buf.put\_AT\_waiting\_1\_first) \wedge$
$(cons\_AT\_invoked\_buf.get) \wedge (buf.get\_AT\_waiting\_1\_last)$.
Since there are no disjunctions in P_t, $k = 1$ and $P_t^1 = P_t$ itself.
Now $P_g^1 = (prod\_AT\_invoked\_buf.put) \wedge (cons\_AT\_invoked\_buf.get)$, is checked in R_g.
And $P_{l_{buf}}^1 = (put\_AT\_waiting\_1\_first) \wedge (get\_AT\_waiting\_1\_last)$, is checked in $R_{l_{buf}}$.

Figure 11: Property decomposition in Phase 3 for the producer-consumer example

7.3.1 DEC_prop()

$DEC\_prop(P_t)$ the decomposition of P_t is defined by means of the following operation :

$DEC\_prop(P_t^k)$: For any P_t^k in P_t, $DEC\_prop(P_t^k)$ (described in figure 12) generates the set of sub-properties $\{P_{l_1}^k, \ldots, P_{l_\alpha}^k, P_g^k\}$. Each $P_{l_i}^k (1 \leq i \leq \alpha)$ is the set $\{P_{l_{i,1}}^k, \ldots, P_{l_{i,\beta_i}}^k\}$. Each $P_{l_{i,j}}^k$ is the conjunction of the proposition TRUE and the propositions in P_t^k that involve

LAP of methods in object $o_{i,j}$. P_g^k is the conjunction of the proposition TRUE and the propositions in P_t^k that involve GAP.

$DEC\_prop(P_t)$ is $DEC\_prop(P_t^k)$ for all P_t^k in P_t. Thus,
$DEC\_prop(P_t) = \{DEC\_prop(P_t^1), \ldots, DEC\_prop(P_t^\epsilon)\}$.

```
DEC_prop(P)      /* Decomposition of property P. */
P : property. /* Conjunction of propositions. */
{
property P_g; /* Sub-property generated by DEC_prop(P) */
properties P_l; /* Set of sub-properties generated by DEC_prop(P) */
proposition p; /* Has the form o.m_AT_a (object.method_AT_analysis point). */
class_id c_i; /* Identifier for class */
object_id o_{i,j}; /* Identifier for object (j^th instance of c_i) */
property P_{l_{i,j}}; /* Sub-property for object o_{i,j} */
properties P_{l_i}; /* Set of sub-properties of all objects of class c_i */

  k = 1;
  repeat    /* For each proposition in P */
   p = P.propsn[c]; /* Take the k^th proposition */
    If (p.a is GAP) then { /* Check the analysis point in p */
     ≪ Include p in P_g ≫
    }
    If (p.a is LAP) then {
     ≪ Determine o_{i,j}, the object identifier in p.o ≫
     ≪ Determine m_x, the method identifier in p.m ≫
     ≪ Include m_x_AT_a in P_{l_{i,j}} ≫
     P_{l_i} = P_{l_i} ∪ P_{l_{i,j}}
    }
   k = k + 1;
  until (k = E); /* E is the number of propositions in P */
}
```

Figure 12: DEC_prop(P) in Phase 3

7.3.2 Checking of sub-properties

We say that 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. Now checking if P_t holds in the apportioned reachability graphs $\{R_{l_1}, \ldots, R_{l_\alpha}, R_g\}$, is done as follows :

1. $DEC\_prop(P_t)$ is generated.
2. For each $DEC\_prop(P_t^k)$ in $DEC\_prop(P_t)$, each $P_{l_{i,j}}^k$ is checked in R_{l_i} and P_g^k is checked in R_g.

$DEC\_prop(P_t^k)$ is satisfied if each $P_{l_{i,j}}^k$ holds in the corresponding R_{l_i}, and P_g^k holds in R_g. P_t holds in $\{R_{l_1}, \ldots, R_{l_\alpha}, R_g\}$, if any of $\{DEC\_prop(P_t^1), \ldots, DEC\_prop(P_t^\epsilon)\}$ is satisfied.

For checking the deadlock property (P_t^1) in the producer-consumer example (figure 11, P_g^1 is checked in R_g and $P_{l_{buf}}^1$ is checked in $R_{l_{buf}}$. Now from figure 10, it can be seen that the state represented as ¡C; X; x; c¿ in R_g, satisfies P_g^1. Also from figure 7, it can be seen that the state represented as ¡closed; closed; C,last; C,first¿ in $R_{l_{buf}}$, satisfies $P_{l_{buf}}^1$. Hence $DEC\_prop(P_t^1)$ is satisfied and as a result P_t holds in $\{R_{l_{buf}}, R_g\}$.

Standard graph traversal and error-detection algorithms are used for checking if a given predicate is satisfied in any node of the reachability graph.

8 Safety and Effectiveness

We say that '*Program S has error e*' (denoted as $S \models e$) if an execution of program S exhibits the error e. Given a program S and an error to be checked e, a general analysis algorithm G, would perform the following actions : (i) generate an execution model of S (denoted as M_S^G), (ii) model the error e as a property (denoted as P_e^G) and (iii) check whether P_e^G holds in M_S^G (denoted as $M_S^G \models P_e^G$). We say that G *rejects* S if $M_S^G \models P_e^G$.

The main criteria that should be satisfied by G is *safety* : G is *safe* if it rejects all erroneous programs, i.e., $(S \models e) \Rightarrow (M_S^G \models P_e^G)$; if an error exists in the program, then it will be detected by the analysis.

A trivial algorithm that rejects all programs is always safe. Hence we define an additional criteria, *effectiveness* : G is *effective* if it rejects only erroneous programs, i.e., $(M_S^G \models P_e^G) \Rightarrow (S \models e)$; if an error is detected by the analysis, then it will exist in the program. Any error that is detected by the analysis but does not exist in the program, is termed as a *spurious error*.

In this paper, we have confined ourselves to reachability analysis, and to errors that can be represented as properties of the global state (expressed as predicates in *disjunctive normal form*).

8.1 Criteria for reachability analysis algorithms

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 algorithm H generates a reachability graph for S (denoted as R_s) and detects the presence of P if $R_s \models P$. H is said to *reject* S if $R_s \models P$.

Note that while H may be *safe*, it is not always *effective*; reachability analysis being a static analysis technique has only limited capability for dealing with dynamic references and may report spurious errors that are the result of infeasible program paths.

Hence we define a notion of *relative effectiveness* for reachability analysis algorithms as : An algorithm H_1 is *more effective* than another algorithm H_2, for analysis of a given program S, if H_1 reports fewer spurious errors than H_2.

As mentioned earlier in section 6, we use the algorithm OMEGA, as the basis for comparing other apportioning-based algorithms.

8.2 Safety and effectiveness of OMEGA

Recall that OMEGA classifies the analysis points corresponding to global invocations as GAP and all other analysis points as both LAP and GAP. Since all the LAP are classified as GAP also, no abstraction is performed in Phase 2, and R_g is itself the *traditional reachability graph* for program execution. Also, the sub-property P_g in Phase 3, is the given state property P_t itself, and is checked in R_g. Hence any state property that may be detected by traditional reachability analysis, also holds in OMEGA. Thus OMEGA is *safe* for all programs in the chosen model.

Also, OMEGA is the most *effective* of the apportioning-based algorithms, since the given error property P_t is not decomposed, and is itself the sub-property P_g that is checked in R_g. OMEGA is also the least *efficient* of the apportioning-based algorithms, since no abstraction is performed in Phase 2 before generating R_g.

Hence we use OMEGA as *the basis for proving the safety and comparing the efficiency and effectiveness* of the other algorithms. These criteria are formalized below.

8.3 Criteria for apportioning-based algorithms

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

We define the safety and effectiveness of A with respect to OMEGA as :

Safety : 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.

Effectiveness : A is *effective* with respect to OMEGA, provided whenever A rejects S, OMEGA also rejects S. More formally, a given apportioning $< \{S_i | i = 1, \ldots, n\}, \{P_i | i = 1, \ldots, n\} >$ of $< S, P >$, is effective provided whenever each $R_{s_i} \models P_i$, $R_s \models P$.

In the next section we present the proofs of safety and effectiveness for ALPHA.

9 Safety and effectiveness of ALPHA

9.1 Safety

The safety of ALPHA is shown by the following theorem :

Theorem 1 *ALPHA is safe for programs that do not have local invocations, i.e., for programs that do not have invocations of the form $self.method()$.*

Now both ALPHA and OMEGA generate the same reachability graphs $R_l = \{R_{l_1}, \ldots, R_{l_\alpha}\}$ in Phase 1. In Phase 2, ALPHA abstracts out non-GAPs and generates the reachability graph R_g, while OMEGA does not perform any abstraction and generates the traditional reachability graph (denoted as R_t). Using the definition of safety in section 8.3, we show the safety of ALPHA as :

Lemma 1 *For any state property P_t that holds in R_t, the decomposition $DEC\_prop(P_t)$ holds in $\{R_l, R_g\}$.*

In order to prove lemma 1, we use the following two lemmas :

Lemma 2 *For any P_t^k that holds in R_t, each property $P_{l_{i,j}}^k inDEC\_prop(P_t^k)$ holds in R_{l_i}.*

Lemma 3 *For any P_t^k that holds in R_t, the property $P_g^k inDEC\_prop(P_t^k)$ holds in R_g.*

We prove lemmas 2 and 3 by showing that for any node s in R_t, corresponding nodes exist in R_l and R_g. Let s have fields $[s^1, \ldots, s^\lambda]$; each field is an analysis point. We define the following operations on s :

$proj_{i,j}(s)$: This is the *projection* of the fields in s that correspond to the entries for object $o_{i,j}$. The fields in s that correspond to entries for other objects are removed. Recall that the entry for $o_{i,j}$ records the status of all its *locks* and analysis points of all its methods.

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

$G\_hide(s)$: 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 2 and 3 are proved using the following two lemmas respectively,

Lemma 4 *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 5 *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 4 and 5 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 may be found in [16].

9.2 Effectiveness

From the definition of effectiveness in section 8.3, it can be seen that an apportioning-based algorithm A, is effective with respect to OMEGA, iff P_t is itself a sub-property in the decomposition $DEC\_prop(P_t)$ and all other sub-properties are TRUE.

The following result for effectiveness of ALPHA is a direct consequence of the above observation and the definition of $DEC\_prop(P_t)$ in section 7.3 :

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

It follows from this theorem that ALPHA is effective for all data access errors, cyclic acquisition of *locks* and global invocation deadlocks in the program. ALPHA is also effective for arbitrary properties that involve either only LAP or only GAP. ALPHA is not effective for properties involving a mixture of LAP and GAP.

10 Efficiency of ALPHA

We have implemented prototype versions of ALPHA and OMEGA using C++, for analysis of concurrent programs written in a subset of Java. For the sake of simplicity, we have implemented the *synchronized* feature of Java using a class *Lock*, as described in our programming model.

In order to show the eficiency 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. The results for the producer-consumer example, are shown in Table 1. 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 in hours:minutes:seconds.tenths.

The number of nodes generated by the algorithms may also be theoretically estimated using the following observations :

- The total number of threads is the sum of the numbers of producers and consumers, since a thread of execution in *produce/consume*() itself performs the corresponding *put/get*() operation on the buffer.

- Concurrent execution of a thread t_1 having p_1 distinct analysis points, with a thread t_2 having p_2 analysis points, results in $(p_1 \times p_2)$ nodes in the reachability graph.

- A transition from the analysis point m_y_entry (passive state of m_y) to m_y_begin (active state of m_y) occurs either spontaneously at start of program execution, or in conjunction with a corresponding transition from $m_x\_before\_invocation\_m_y$ to $m_x\_invoked\_m_y$. Similarly the transition from m_y_return to m_y_entry occurs in conjunction with the transition from $m_x\_invoked\_m_y$ to $m_x\_after\_invocation\_m_y$. Hence analysis points of the form m_x_entry and $m_x\_invoked\_m_y$ should not be enumerated while theoretically estimating the number of nodes.

It can be easily verified that the numbers of nodes generated by the experiments are quite close to their respective theoretical estimates.

Table 2 presents the experimental results for the well-known dining philosophers problem: There are n philosophers seated for dinner, with one fork between every two of them. When hungry, each philosopher picks up two forks, one to the right and one to the left, and subsequently releases them. A *fork* class supports (i) a method *up()*, which waits until the fork is free and then assigns it to the invoking philosopher and (ii) a method *down()*, which releases the fork if it has been taken by the invoking philosopher.

From Tables 1 and 2, we note that while exact values for the execution times are of limited importance, the *percentage of reduction for ALPHA v/s OMEGA*, in the number of nodes as well as in execution times, is extremely significant.

More experimental results for comparing the efficiency of ALPHA v/s OMEGA, for implementations of a database application and a traffic simulation problem are reported in [18]. In the next section, we describe some extensions to ALPHA and present more examples.

Algorithm	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%	-

† Number of nodes in R_l is 39 for both ALPHA and OMEGA.

Table 1: Experimental results for the producer-consumer example

Algorithm	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%	-

† Number of nodes in R_l is 47 for both ALPHA and OMEGA.

Table 2: Experimental results for the dining philosophers example

11 Extensions to ALPHA

We have extended ALPHA for analysis of concurrent object-oriented programs having the features of fine-grain concurrency, dynamic creation and destruction of objects and to support inheritance. A brief description of these extensions is as follows :

Fine-grain Concurrency : ALPHA-F is the extension of ALPHA to support bounded fine-grain concurrency, i.e., multiple concurrent invocations of the same method in an object. Phase 1 of the analysis remains unchanged. The structure of the nodes in R_g (Phase 2) is extended to also allow recording the status of the analysis points for *each* of the multiple concurrent invocations of a method. An appropriate bound on the number of such invocations is assumed.

Table 3 gives a comparison of the complexity of ALPHA-F with that of OMEGA-F, for various number of users for a hardware stack. (An n-element hardware stack is constructed by pipelining n individual stack elements, each storing only one data item and supporting push() and pop() operations. When a push() is performed on the i^{th} element, it stores the pushed data. Any data already stored is pushed to the $(i+1)^{th}$ element; pop() is performed analogously. *Multiple invocations of push() and pop() on the same element, are possible*).

Dynamic Objects : ALPHA-D is the extension of ALPHA to support bounded dynamic creation and destruction of objects. A class provides *constructor* and *destructor* methods which are invoked for the creation and destruction of objects belonging to that class. Such invocations are regarded as *global* invocations. Hence Phase 1 of the analysis remains unchanged. The structure of the nodes in R_g (Phase 2) is extended to also allow recording an additional status for each object, viz., *live*. Only the objects declared in the program are *live* at the start of program execution. During reachability graph generation in Phase 2, whenever an analysis point corresponding to the invocation of a constructor method is encountered, the status of the corresponding object is changed to *live*, and remains so until an invocation of the corresponding destructor method is encountered. An appropriate bound on the total number of objects in the system, at any given time is assumed.

Table 4 compares the complexities of ALPHA-DF and OMEGA-DF for various number of users accessing a sequential database file. (A sequential database file supports operations for insert, read, modify and delete of records. Requests are served by sequentially scanning all the records. *Multiple concurrent invocations on the same record as well as dynamic creation/destruction of records during execution, are possible*).

Inheritance : ALPHA-I is the extension of ALPHA to support inheritance, i.e., the definition of new classes on the basis of existing ones. Single, non-strict inheritance ([29], [26]), 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. The analysis points corresponding to *super-reference* are classified as GAP. Hence, Phase 1 remains unchanged. The structure of the nodes in R_g (Phase 2) is extended to also allow recording the status of the analysis points for *each* of the inherited methods in an object. An appropriate *inheritance hierarchy graph* is used to capture the inheritance hierarchy of the classes.

Table 5 gives a comparison of the complexities of ALPHA-IDF with OMEGA-IDF, for a random access file for various number of users. (A sorted random access file is defined using *inheritance* on some of the methods of the sequential file. As in the case of sequential file access, *fine-grain concurrency and dynamic objects are possible*).

A more detailed discussion of these extensions is given in [16].

Algorithm	2-users	3-users	4-users
ALPHA-F	(196 + 18) †	(2744 + 18)	(38416 + 18)
OMEGA-F	(324 + 18)	(5832 + 18)	(104976 + 18)
Reduction %	37.43%	52.79%	63.39%

† Number of nodes in R_l is 18 for both ALPHA-F and OMEGA-F.

Table 3: Comparison of efficiency for hardware stack operations

Algorithm	2-users	3-users	4-users
ALPHA-DF	(784 + 258) †	(21952 + 258)	(614656 + 258)
OMEGA-DF	(1296 + 258)	(46656 + 258)	(1679616 + 258)
Reduction %	32.95%	52.66%	63.40%

† Number of nodes in R_l is 258 for both ALPHA-DF and OMEGA-DF.

Table 4: Comparison of efficiency for sequential file access

Algorithm	2-users	3-users
ALPHA-IDF	(3600 + 278) †	(216000 + 278)
OMEGA-IDF	(4624 + 278)	(314432 + 278)
Reduction %	20.89%	31.28%

Table 5: Comparison of efficiency for random access file

12 Conclusions

We have developed a new technique called apportioning for reachability analysis of concurrent object-oriented programs and have described in detail, ALPHA, an efficient apportioning-based algorithm that is safe for programs not having local invocations. We have also shown apportioning to be useful technique for generating many algorithms having varying degrees of efficiency and effectiveness. An approach for incorporating such algorithms into a tool-suite to enable choice of an 'ideal' tool for a given program, is described in [17].

We make the following observations regarding apportioning :

- The efficiency of apportioning 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.
- The efficiency of apportioning increases as the number of threads increases. This is due to the fact that for traditional analysis (OMEGA.R_g), the analysis of local interactions in objects is performed repeatedly for various threads, while for apportioned analysis (ALPHA.R_g), it is performed only once.
- Although the number of states generated by traditional and apportioned analysis are both exponential in the number of threads, the percentage of reduction (in the number of states as well as in execution time) due to apportioning, is quite significant. This is due to the fact that the amount of reduction is also composed of exponential terms.
- Optimization approaches for traditional reachability analysis, such as the use of efficient representations, are equally applicable to apportioned analysis, maintaining the significance of the reduced complexity.
- The task of generating the apportioned reachability graphs may be easily distributed among multiple processors in order to further reduce the execution time. While this does not have any effect on the total number of states generated, the reduction in execution time is likely to increase the practical utility of the technique.

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. Thus the general idea of apportioning can be extended to other techniques for analysis of concurrent object-oriented programs also.

References

[1] G. R. Andrews. Paradigms for process interactions in distributed systems. *ACM Computing Surveys*, 23(1):49–90, Mar 1991.

[2] K. R. Apt. A static analysis of CSP programs. *Proceedings of Workshop on Progam Logic, published in Lecture Notes in Computer Science, vol. 164*, Springer-Verlag, 1984.

[3] G. S. Avrunin, U. A. Buy, J. C. Corbett, L. K. Dillon and J. C. Wileden. Automated analysis of concurrent systems with the constrained expression toolset. *IEEE Transactions on Software Engineering*, 17(11):1204–1222, Nov 1991.

[4] E. Adams and S. S. Muchnick. Dbxtool: A window-based symbolic debugger for Sun workstations. *Software - Practice and Experience*, 16(7):653–669, July 1986.

[5] W. F. Appelbe and C. E. McDowell. Integrating tools for debugging and developing multitasking programs. *Proceedings of the ACM SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging, published in ACM SIGPLAN NOTICES*, 24(1):78–88, Jan 1989.

[6] J. P. Bahsoun, S. Merz, and C. Servieres. Modular description and verification of concurrent objects. *In proceedings of Workshop on Object-Based Parallel and Distributed Computation*, pp. 168-186, LNCS 1107, Springer, June 1995.

[7] 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.

[8] 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.

[9] J.-D. Choi, B. P. Miller and R. H. B. Netzer. Techniques for debugging parallel programs with flowback analysis. *ACM Transactions on Programming Languages and Systems*, 13(4):491–530, Oct 1991.

[10] D. Callahan and J. Subhlok. Static analysis of low level synchronization. *Proceedings of the ACM SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging, published in ACM SIGPLAN NOTICES*, 24(1):100–112, Jan 1989.

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

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

[13] S. Graf and B. Steffen. Compositional minimization of finite state systems. *In Proceedings of International Conference on Computer-Aided Verification*, LNCS 531, pp. 186–196, Springer-Verlag, 1990.

[14] G. J. Holzmann. Design and Validation of Computer Protocols. *Prentice-Hall, New Jersey*, 1991.

[15] W. Hseush and G. E. Kaiser. Modeling concurrency in parallel debugging. *Proceedings of the ACM SIGPLAN/SIGOPS Workshop on Principles and Practices of Parallel Programming, published in SIGPLAN Notices*, pages 11–20, Mar 1990.

[16] S. Iyer. Efficient reachability analysis for concurrent object-oriented programs. *Ph.D. Thesis*, Indian Institute of Technology, Bombay, 1998.

[17] S. Iyer and S. Ramesh. A tool-suite for reachability analysis of concurrent object-oriented programs. *In Proceedings of Joint Asia-Pacific Software Engineering Conference and International Computer Science Conference*, IEEE Computer Society, HongKong, 1997.

[18] S. Iyer, R. Raghuraman and A. Majumdar. Apportioning-based analysis of concurrent Java programs. *In Proceedings of International Conference on Information Technology*, Bhubaneshwar, India, Dec 1999.

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

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

[21] K. L. McMillan. Symbolic model checking : An approach to the state-explosion problem. *PhD Thesis*, Carnegie-Mellon University, 1992.

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

[23] Z. Manna and A. Pnueli. Temporal verification of reactive systems. *Springer-Verlag*, 1995.

[24] Sape J. Mullender, Guido van Rossum, Andrew S. Tanenbaum, Robert van Renesse, and Hans van Staveren. Amoeba: A distributed operating system for the 1990s. *Computer*, 23(5):44–53, May 1990.

[25] K. K. Sabnani, A. M. Lapone and M. U. Uyar An algorithmic procedure for checking safety properties of protocols *IEEE Transactions on Communications* 37(9):940–948, Sep 1989.

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

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

[28] A. Valmari A stubborn attack on state explosion. *Proceedings of International Conference on Computer-Aided Verification*, LNCS 531, pp. 156–165, Springer-Verlag, 1990.

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

[30] M. Young and R. N. Taylor. Combining static concurrency analysis with symbolic execution. *IEEE Transactions on Software Engineering*, 14(10):1499–1511, Oct 1988.

[31] 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.

[32] 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.

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

[34] J. Zhao, J. Cheng and K. Ushijima. Static slicing of concurrent object-oriented programs, *In Proceedings of the 20th IEEE Annual Intl. Computer Software and Applications Conference*, Aug. 1996.

Authors' Biographies

Sridhar Iyer

Sridhar Iyer received the B.Tech and PhD degrees in Computer Science and Engineering at the Indian Institute of Technology, Bombay, India in 1989 and 1998 respectively. Currently he is working as an Assistant Professor in the School of Information Technology at Indian Institute of Technology, Bombay. He has also been a faculty member in the Dept of Computer Science and Engineering at Indian Institute of Technology, Guwahati. His current research interests include: Mobile Agents, Ad-hoc Networks and Analysis of Distributed Programs.

S. Ramesh

S. Ramesh received the B.E. degree in Electronics and Communication Engineering from Indian Institute of Science, Bangalore, India in 1981 and the Ph.D. degree in Computer Science and Engineering from Indian Institute of Technology, Bombay, India in 1987. Currently he is working as Professor in the Department of Computer Science and Engineering, Indian Institute of Technology, Bombay. He has held visiting research positions in Techniche Universitat Eindhoven, TIFR Mumbai, GMD Bonn and Univ. of New Southwales Sydney. His current research interests include: Formal methods for design and verification of programs, Distributed Computing, Protocol Engineering and Embedded Real Time Systems.