Distributed Apportioning Based Reachability Analysis of Concurrent Java Programs

Inderdeep Singh
School of Information Technology
IIT Bombay, INDIA.
Email: inder@it.iitb.ernet.in

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

ABSTRACT

Reachability analysis is an important and well-known tool for static analysis of critical properties in concurrent programs, such as freedom from deadlocks. 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).

Apportioning is a technique that overcomes these limitations and enables safe and efficient reachability analysis of concurrent object-oriented programs. Apportioning 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.

In this paper we present the development of ARA, an apportioning based tool for analysis of concurrent Java programs. Some of the main features of ARA are: varying the classification of analysis points, distributing the generation of reachability graphs over several machines, and the use of efficient data structures, to further reduce the time required for reachability analysis. We also present our experience with using ARA for the analysis of several programs.

1 INTRODUCTION

The Java programming language has gained wide acceptance for the development of distributed applications, Although the techniques for debugging concurrent programs have been extensively studied, there is little work on analysis and verification of concurrent object-oriented programs. 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 [5, 6]. Traditional reachability analysis suffers from combinatorial-explosion i.e., the number of states generated for analysis increases exponentially with the number of concurrent threads of execution.

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 straightforward 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.

In an earlier paper [1], we reported the development of apportioning which abstracts local and global components from the input program flow graph, and performs analysis on these abstracted components. The abstraction is based on classifying the flow graph states into *global*, i.e., those that transfer control flow to an instruction beyond the scope of the given class, and *local*, i.e., those that transfer control flow to another instruction in the same class.

This work was supported by CFDVS, IIT Bombay.

The former are referred to as GAP *(Global Analysis Points)* and they form the global component (p_g) of the flow graph, while the latter are referred to as LAP *(Local Analysis Points)*, and they form the local component (p_l) of the flow graph.

The practical utility of the apportioning technique can be seen from the following observations: The complexity (number of states generated) of traditional reachability analysis [5, 6] is $O(p)^T$, where T is the number of threads and p is the number of interactions for any thread. Extending such techniques to concurrent object-oriented programs by performing additional analysis for each class, 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. A more detailed discussion of apportioning along with its proofs of safety and effectiveness may be found in [1].

In this paper, we present ARA, an apportioning based reachability analysis tool for Java programs. ARA uses efficient data structures and distributed generation of reachability graph over several machines for further reduction of the time taken for analysis. ARA has been implemented as a standalone application executing on a single (Windows/Linux) host, as well as a client-server application executing on multiple hosts (Windows/Linux). We have used ARA successfully for performing reachability analysis of applications such as, implementations of the producer-consumer problem and a GUI based circuit designer. To the best of our knowledge, there is no other tool for reachability analysis of concurrent Java programs.

2 APPORTIONING

As mentioned earlier, the main drawback of reachability analysis is its exponential complexity, $O(p)^T$, where T is the number of participating processes and p is the size of the control-flow graph for any given process. In 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 [8], consequently reducing the size of the model. Partitioning decomposes the model into smaller components [7], so that analysis of the large model is replaced by individual analysis of smaller models. Though several analysis techniques have been proposed for concurrent programs [4,9], there is no literature on the use of reachability analysis for concurrent object-oriented programs.

2.1 The apportioning technique

Apportioning integrates the ideas of partitioning [7] 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).
- *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.

The inputs to an apportioning-based reachability analysis tool are: (i) an abstract representation of the program (in the form of control-flow graphs), (ii) a classification of analysis points into LAP/GAP, and (iii) the query 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 query specification are implicit inputs built into the tool.

As shown in Fig 2.1, apportioning-based reachability analysis proceeds as follows:

(a) Given the program control-flow graph and the classification of analysis points:
   1. 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.
   2. 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 this reduced class control-flow graphs and the reduced program control-flow graph is generated.
(c) The query 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.

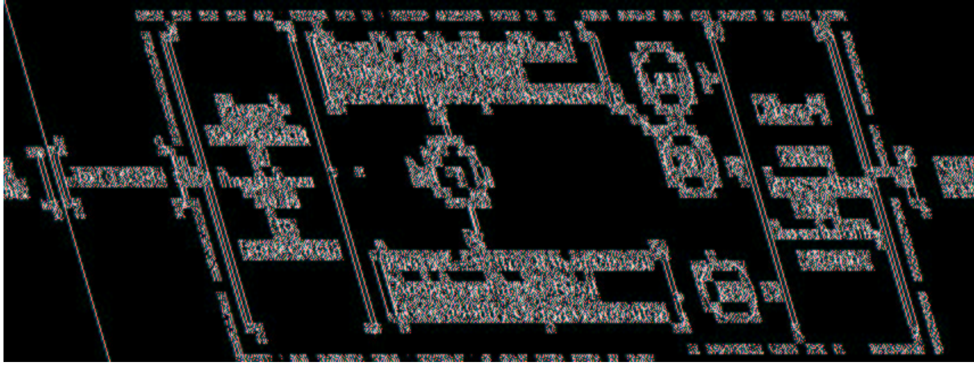

Fig 2.1 Apportioning based reachability analysis

2.2 Advantages of apportioning

The main advantages of apportioning are:

- *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.
- *Reduction in complexity:* Typically, the complexity (number of states generated) of traditional reachability analysis for concurrent programs is $O(p)^T$, apportioning reduces the complexity to $O(c(p_l)^m + (p_g)^T)$, where $p \approx (p_l + p_g)$.

A more detailed discussion of apportioning, its advantages and proofs of correctness, is given in [1].

In the next section, we describe the design and implementation of ARA, an apportioning based tool for the reachability analysis of multi-threaded Java programs.

3 ARA: DESIGN AND IMPLEMENTATION

3. 1 System Model:

ARA is applicable for performing reachability analysis of multi-threaded Java programs, with the following restrictions:

- *Objects need to be explicitly declared*: For example, a function call like '*new classobj();*' made with a nameless object is not supported. It needs to be rewritten as '*classobj obj=new classobj();*'.
- *Threads need to be explicitly created*: For example, *Thread t[]=new Thread[10]; t[k].start();* is not supported as the value of subscript 'k' is dynamic assigned.
- 'Inner classes' are not supported.

3.2 Analysis modes in ARA

ARA supports the following in built analysis modes:-

a) **OMEGA :** OMEGA classifies the analysis points corresponding to global invocations as GAP and all other analysis points as both LAP and GAP. OMEGA corresponds to the extension of traditional reachability analysis to concurrent object oriented programs.
b) **ALPHA :** ALPHA classifies the analysis points corresponding to 'lock acquire' (entry point to a synchronized method), 'lock release' (exit from a synchronized method) and all local function calls as LAP. All global invocations are GAP.
c) **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 local function calls

are classified as both LAP and GAP.

d) **GAMMA:** GAMMA is an optimization of BETA, as follows:- Instead of classifying all the local function calls as LAP and GAP, only those local function calls that directly or indirectly result in a global invocation, are classified as both LAP and GAP.

More details on OMEGA, ALPHA, BETA and GAMMA may be found in [1].

In addition to the in built analysis modes, ARA also supports user defined specification of analysis points. The user can specify in an APS file, the classification of analysis points types into LAP or GAP.

3.3 Overview of ARA

ARA's input is a concurrent Java program and reachability queries, and its output is the query results. ARA has been implemented as a standalone application as well as with a client - server based paradigm, as shown in Fig 3.1.

The standalone version is implemented in VC++ on the Windows platform and in C++ on the Linux platform. It performs local reachability analysis, global reachability analysis, and query processing. The application is single threaded, and provides a GUI for query building (see section 3.6.3).

The client server version is an extension of the standalone versoin and it distributes the generation of reachability graphs onto multiple machines. The client is written in Java (Swing) and is a multithreaded application which connects to multiple servers concurrently. Each server is implemented in C++, for Linux as well as Windows platforms. The GUI features of the client are similar to that of the standalone version.

Both versions of ARA provide in built classification of analysis points (OMEGA, ALPHA, BETA, GAMMA), and also allow the user to provide a different classification of analysis points. The detailed design of the client-server version is given in the following sections.

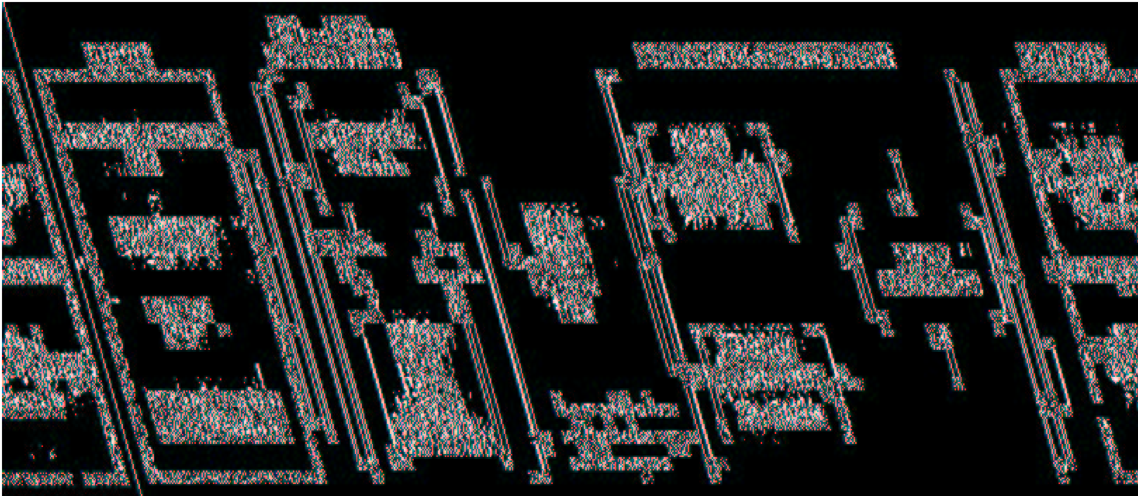

Fig 3.1 Design of ARA

3.4 Inputs to ARA

The inputs to ARA are:

- *Java source files* : The location of source code of the program to be analyzed.
- *Server addresses*: The IP addresses of the servers are specified in a file.
- *Analysis mode*: The user could select an in built analysis mode or specify a different classification of analysis points in an APS file.
- *Query specification*: The user could use the GUI to build a query or write it to a query file. Queries could be checked concurrently with reachability graph generation, or they could be checked on reachability graphs that were generated earlier.

3.5 Parsing and flow graph generation

This phase parses the given Java code to generate flow graphs from the classes, methods, objects and synchronized constructs in the program. The parser is written using Lex and Yacc and the flow graph generator is written in C. The parser invokes the flow graph generator whenever it encounters instructions

related to the control flow of the input program, such as loops, conditional statements, break, continue, function calls, entry and exit from methods. After parsing of each method, the flow graph generator runs a *backpatching* algorithm to assign successors to every node of the flow graph of that method. The ARA client invokes this phase with appropriate parameters, to generate the flow graph for any given program.

3.6 Client side processing

The ARA client is a multithreaded application that provides a GUI for the user interaction with the back end servers. The client has a main thread and several child threads which perform the following actions:

3.6.1 Main thread actions

1) The main thread generates the flow graph from the Java source code by invoking the parser and flow graph generator.
2) The main thread reads server address file and initiates a new child client thread corresponding to every server. Each child thread locates an available server, using the algorithm in Fig 3.2.
3) The main thread now runs a load management algorithm (Fig 3.3), and apportions the tasks of reachability graph generation, among the set of available online servers.

```
Algorithm for locating an available server
//returns a client corresponding to an available server in the online state
Client* available_online_server()
int servers_online   // servers_online >0 if at least one server is online, initialized by main thread
   while servers_online!=0
     for all client threads do
       if client's corresponding server is not online   do
           establish TCP connection with server()
       else if   client's corresponding server is online and is not busy
                return client
       end if
       if   client's corresponding server is online
            servers_online=1;
       end if
     end for
   end while
end
```

Fig 3.2 Algorithm for locating an available server

```
Algorithm for load management
Void load_manager()
  while Global_analysis_not_done && for_all_classes_ local_not_done ()  && more than 0 servers running do
        client=available_online_server ()

     if  global analysis not done
        assign_task_to_server("global analysis", program_flow_graph_file);
     else  assign_task_to_server ("local analysis",class_flow_graph);
     end if
  end while
end
```

Fig 3.3 Algorithm for load distribution

3.6.2 Child thread actions

1) The child threads establish a socket with one of the servers as explained in Fig 3.2 and 3.3.
2) Depending upon the task to be performed (i.e. query processing, local reachability analysis, global reachability analysis), the client runs a protocol to communicate the appropriate task parameters to the server, and then waits for a *task completed token* from the server. While waiting for this token, the

client receives the server's *status update packets* giving the percentage of the task completed by the server, and updates the GUI to indicate the server status (see Fig 3.4).

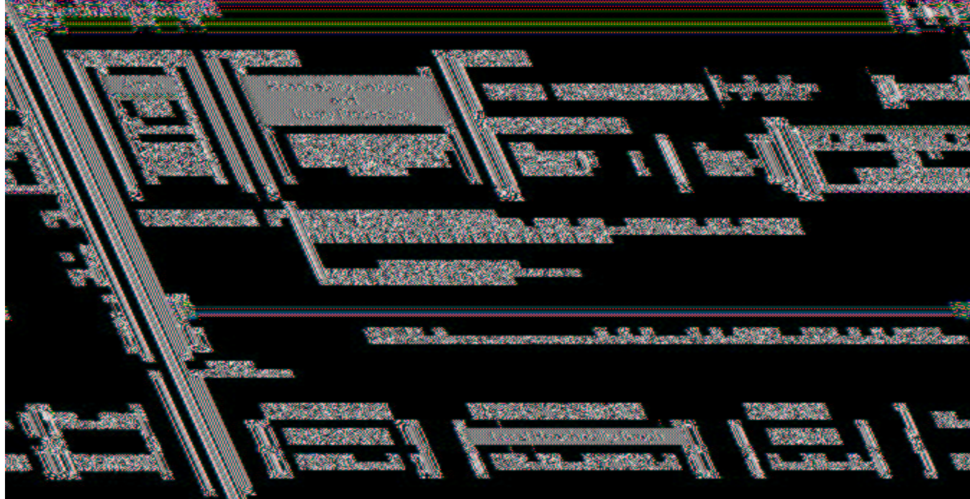

Fig 3.4 Client GUI showing the status of reachability analysis

3.6.3 Query building

We use the conjunctive normal form (CNF) syntax to represent queries. A typical query is represented as >($thread_1$_$state_{11}$ | $thread_1$_$state_{12}$ | … | $thread_1$_$state_{1n}$) &…& ($thread_m$_$state_{m1}$ | $thread_m$_$state_{m2}$ | …. | $thread_m$_$state_{mn}$) <. The 'state of a thread' is the analysis point of the method & class at which the thread is currently executing. The client incorporates a GUI to help the user build queries, as shown in Fig 3.5

Fig 3.5 Query building

3.7 Server side processing

Each server side component is a single threaded application which establishes a server socket and connects to at most one client at a time. The addresses of servers is maintained in a server address file, which is accessible to the client side component. Each server provides the following services:-

- *Local reachability analysis:* concurrent execution of the methods in a given class (section 3.7.4).
- *Global reachability analysis:* concurrent execution of all threads in the given program (section 3.7.5).
- *Query processing:* checking for state properties in the appropriate graphs (section 3.7.6).

3.7.1 Main thread actions

1) The server runs as an infinite loop. In each iteration, the server provides one of the above services.
2) The server accepts a TCP connection from a client, and executes the appropriate service routine.
3) The server periodically sends server's *status update packets*. These packets are used by the client application to maintain a progress indicator on the GUI.

3.7.2 Reachability graph representation

The main data structure required for reachability analysis is a graph denoted as (V,E), where V is the set of reachability graph nodes and E is the edges representing the adjacent reachable nodes to a particular node. The major operations on this structure during reachability analysis or query processing are *searching* and *insertion*. Generally we would use a breadth first search (BFS) algorithm for performing these operations. However, the time complexity of BFS increases linearly with the size of the graph. On the other hand, the time complexity of an AVL tree increases logarithmically for these operations. Since the number of nodes in a reachability graph is very high, we explored the use of an AVL tree for representing the graph. To get a feel of the actual numbers and processing time involved, we performed an experiment to compare the time for search/insert operation of BFS with an AVL tree, whose results are given in Fig 3.6. We chose to use an AVL tree structure for representation of the reachability graph, whose details are given in the next section.

Fig 3.6 AVL tree Vs breadth first search on graph data structure

3.7.3 Reachability graph data structure in ARA

The composite data structure used by ARA (see Fig 3.7) involves an AVL tree backbone, which classifies the reachability graph nodes with the use of a '*key*' for every node. Thus the reachability graph structure is superimposed over the AVL tree and all operations over the graph can now be made over the AVL tree. When a node is generated during reachability analysis, a *key* is also generated using the contents of that node. This key is used to search for an appropriate node of the AVL tree. If such a key already exists, the reachability graph node is appended to a linked list within the AVL tree node. If such a key does not already exist, a new AVL tree node is created and the reachability graph node is added to its linked list. Finally the AVL tree is balanced using an appropriate rotation algorithm. The task of inserting a node into the AVL tree is a recursive one and balancing of the AVL tree is conducted before returning from the recursive call. Thus the balancing is confined to that branch of the AVL tree along which a node was added, and the balancing takes at most 2-3 pointer rotation steps. The search/insert operations thus have a *logarithmic time complexity rather than a linear one,* and the reachability graph edges are also retained.

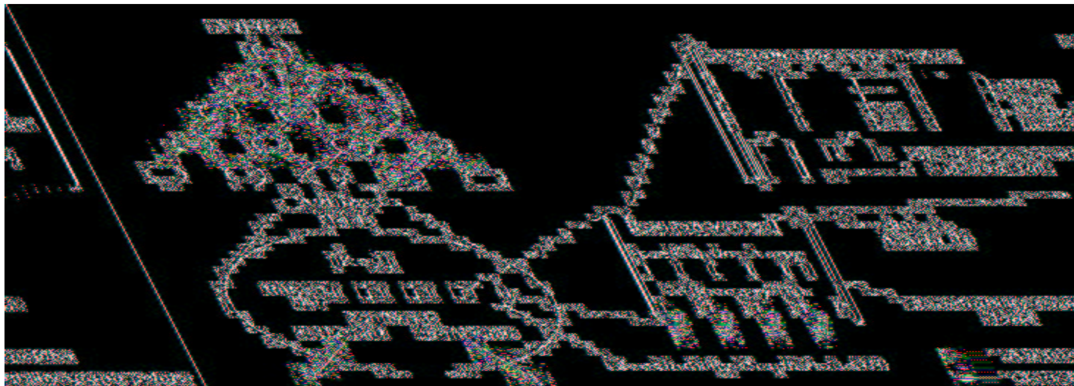

Fig 3.7 Data structure for representation of reachability graph

3.7.4 Local reachability graph generation

This phase abstracts out the GAP from the given class flow graph and generates reachability graph for this class. Each class is analyzed separately by assuming that every method of the class is executed concurrently. This phase generates a file containing the GAP abstracted graph and another file containing the reachability graph. The phase is written in C++ (Linux & Windows) and runs on the server side.

3.7.5 Global reachability graph generation

This phase abstracts out the LAP from the given program flow graph and generates reachability graph for the entire program. This phase involves the analysis of the main thread of the given program, along with all those threads which are invoked by the main thread. Unlike local analysis, the threads flow of control in global analysis may traverse the class boundaries. This phase generates a file containing the LAP abstracted graph and another file containing the reachability graph. The phase is written in C++ (Linux & Windows) and runs on the server side.

3.7.6 Query processing

Each query specified by the user is decomposed into sub-queries as follows: One sub-query is constructed to include all the GAP 'thread_states' from the given query. For each class in the program, a sub-query is constructed to include its LAP 'thread_states' from the given query. More details of query processing may be found in [1]. Each sub-query is then checked in the appropriate reachability graph.
ARA provides the options of processing queries concurrently with generation of the reachability graph or on graphs that are already generated. Each server sends periodic *status update packets* to the client, while processing queries (see Fig 3.4)

3.8 ARA outputs

ARA generates the following output files:

- Flow graph file: contains the flow graphs of given Java program.
- LAP abstracted flow graph: contains flow graphs after abstraction of local analysis points (LAP).
- GAP abstracted flow graph: contains flow graphs after abstraction of global analysis points (GAP).
- Local reachability graphs: contains the local reachability graph for each class in the program.
- Global reachability graph: contains the global reachability graph for the entire program.
- Query response: contains the responses to the queries.

4 CONCLUSIONS

We have described the design and implementation of ARA, a distributed apportioning based reachability analysis tool, for multi-threaded Java programs. We have used ARA to check for deadlocks and other state properties, in implementations of the producer-consumer problem and a GUI based circuit designer.

We are in the process of using ARA to test other real-life Java applications, as well as extending the constructs supported by ARA. We expect to shortly make ARA available for download from http://www.cfdvs.iitb.ac.in

5 ACKNOWLEDGEMENTS

We are thankful to the Centre for Formal Design and Verification of Software (CFDVS), IIT Bombay, for providing the necessary resources to carry out this project.

6 REFERENCES

[1] S. Iyer and S. Ramesh. Apportioning: A technique for efficient reachability analysis of concurrent object- oriented programs. IEEE Transactions on Software Engg. Accepted for publication, 2000.
[2] 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.
[3] G. J. Holzmann. Design and Validation of Computer Protocols. Prentice-Hall, New Jersey, 1991.
[4] 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.

[5] R. N. Taylor. A general purpose algorithm for analysing concurrent programs. Communications of the ACM, 26(5):362 376, May 1983.
[6] C. E. McDowell. A practical algorithm for static analysis of parallel programs. Journal of Parallel and Distributed Computing, 6(3):515 536, June 1989.
[7] 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.
[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] 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.