Analysis of Concurrent Software by Cooperative Application of Static and Dynamic Techniques

Richard N. Taylor

Technical Report #196

Department of Information and Computer Science
University of California, Irvine
Irvine, California 92717

April 7, 1983

Analysis of Concurrent Software by Cooperative Application of Static and Dynamic Techniques

Richard N. Taylor

Programming Environment Project
Department of Information and Computer Science
University of California, Irvine

Abstract

Stand-alone techniques for the analysis and testing of the synchronization structure of concurrent programs have recently been developed. These techniques are able to detect, for example, task blockage, including deadlock. Static analysis provides firm results, but has limited applicability and is potentially expensive. Dynamic analysis makes fewer assumptions, but its assurances are not as strong. This paper presents strategies whereby the two can be employed jointly to advantage. Dynamic analysis can be used to further investigate results from static analysis, and vice versa. Their joint use can be facilitated by an appropriate implementation, some principles for which are outlined.

1. Introduction

It seems almost universally accepted that developing concurrent software is more difficult than developing sequential software. This difficulty manifests itself particularly in the verification and testing process. Execution of a sequential program can be viewed as the process of stepping through linear text, branching as necessary. Understanding (and thus verifying) concurrent execution is more difficult as several texts are simultaneously involved. Moreover, transitions between program states are determined not only by traditional semantics, but also by notions of synchronization, speed of physical processors, and the external real-time environment. It seems most appropriate, therefore, to develop verification and testing (V&T) tools and techniques that aid in examining these additional program characteristics, and the consequences of their presence. The techniques considered in this paper focus specifically on questions of synchronization, deadlock, and determination of activities that may occur in parallel. More general aspects of functional correctness are not considered.

Recently, relevant V&T techniques have been advanced in two areas: static analysis and dynamic analysis. Particularly useful in examining concurrent software is static analysis, as it does not require execution of the code. A suitably constructed analyzer is able to examine all possible sequences of events, under all circumstances. If it is able to demonstrate the required (structural) properties of a program, then one may have strong confidence in the program indeed. This confidence is independent of changing real-time conditions and the actions of an unpredictable scheduler.

Dynamic analysis presents a contrast: a single test can only demonstrate the correctness of a single path for a single set of test data, under a specific set of external conditions (real-time environment, physical processors, scheduling algorithm, and the like). What is more, with sequential software instrumentation may be used during repeated tests to progressively isolate the source of an error. With concurrent software, however, instrumenting the code may upset the timing such that new errors are introduced or old ones eliminated.

Unfortunately dynamic analysis cannot be dismissed so lightly. Static analysis makes several assumptions and often has several limitations, any of which may lessen its utility. (Not to mention the obvious fact that static analysis cannot even begin to address functional correctness: it is only concerned with "shallower" structural properties). Dynamic analysis is therefore a necessary adjunct to static analysis, seeking to make up for these deficiencies.

The purpose of this paper is to briefly overview the developments in these two areas, then proceed to indicate the desirability and possibility of their integration. Their combined application would be synergistic, exceeding the benefits resulting from a stand-alone application of the tools.

To give the discussion a specific reference, attention will be restricted to analysis of programs written in Ada* [Ichb82]. It should be kept in mind, however, that the results achieved, as well as the problems encountered, are equally pertinent to many other concurrent programming languages, such as HAL/S [Mart77] or CSP [Hoar78].

The following section will detail some of the capabilities of both static and dynamic analyzers of concurrent software. Section 3 then explores some of the possibilities for conceptually integrating the techniques. A few implementation issues associated with the physical integration of the techniques are considered in Section 4. Section 5 is the conclusion.

2. Stand-alone Capabilities

2.1. Analysis Objectives

Programs written in Ada may consist of an arbitrary number of simultaneously executing asynchronous tasks. Tasks may be synchronized, wherever desired, by means of rendezvous. If, at any time, requests for several rendezvous

*Ada is a trademark of the U.S. Department of Defense (AJPO).

are pending, "non-deterministic" choice of one is possible through use of the select statement. Task termination occurs upon reaching the end of a task, through task abortion, or upon execution of the terminate statement.

Some key analysis questions associated with concurrent Ada programs are easy to see. Since the rendezvous is the only mechanism for synchronization, do the "desired" rendezvous, and only the desired rendezvous, always occur? Are the rendezvous points correctly positioned such that no activities may execute in parallel that should not? Is the structure of the program such that it will never deadlock? Is the synchronization structure of the program overconstrained such that performance bottlenecks result?

These are the topics addressed by the techniques below.

2.2. Static Analysis

The static analysis technique that is the object of our discussion is described in detail in [Tayl83]. The technique presented there provides information regarding three aspects of an Ada program's synchronization structure: identification of all the rendezvous that are possible, detection of any task blockages (deadlocks) that may occur, and a listing of all program activities that may occur in parallel.

Described first are the limitations of the technique. As with several other static analysis tools, the technique is defined with respect to a graph model of programs. In particular each Ada program unit (subprogram, task, package, or generic) defines a flowgraph: each statement in the unit is represented by a node in the graph; each transfer of control is represented by a directed edge. A path through the graph represents a sequence of statements. Not all paths through the graph correspond to executable sequences of statements, of course. Determination of the possible (or impossible) paths requires examination of the program logic - the semantics of the branch conditions, etc. This determination is in the realm of symbolic execution or formal verification. Since the domain of this discussion is static analysis, it therefore has an inherent inaccuracy: namely, all paths through a flowgraph are assumed executable.

Three other limitations on the analysis quality exist. First, static analysis is only accurate when individual program objects (like tasks or entries) can be identified statically; program features potentially causing dynamic identification, such as access values (pointers) and subscripts, are inadequately handled, and are therefore excluded from further discussion in this subsection.

Second, because of the specific techniques used, the number of tasks created during execution must be bounded. Third, since the analysis conducted is independent (ignorant) of the target execution environment, the implications of delay statements, non-zero execution times, and scheduler algorithms are not taken into account. (This restriction may be viewed as an asset, of course: the results produced do not rely on any possibly erroneous assumptions about the target environment.)

One of the issues addressed later in this paper is how the effect of these restrictions can be mitigated.

Turning now to the analysis technique itself, its central notion is the concurrency state. The algorithm of [Tayl83] determines all the concurrency states a program can possibly enter. Each state displays the next synchronization-related activity to occur in each of a system's tasks. A (legal) sequence of states presents a history of synchronization activities for a class of program executions. These states contain or infer the desired information regarding rendezvous, deadlock, and parallel activities.

Concurrency states are defined in terms of state nodes.

Definition

A state node c_i in a flowgraph is a node that corresponds to any of the following statements: entry call, accept, select, delay, abort, task begin, task end, and subprogram begin, subprogram end, subprogram call, block begin, block end, but only where the subprogram or block may directly or indirectly perform a tasking activity.

A finer degree of statement granularity is actually required than indicated by this definition, but that detail is not necessary for this discussion.

The *successors* of a state node are defined as follows. Let G be a flowgraph for a unit of program S.

Definition

The set of successor nodes of state node c_i, denoted succ(c_i), is the set of all state nodes for which there exists a path p from c_i to c_j in G such that there is no state node c' on p between the first node of the path, which is c_i, and the last node on the path, c_j.

These definitions present an abstraction of a sequential unit of an Ada program. The model retains specification of the possible sequences of tasking related activities. Other details are omitted. Succ(c_i) represents the set of all *tasking* activities that may possibly follow execution of c_i. Note that many activities unrelated to tasking may intervene between c_i and any one of its successors: assignments, procedure calls, branches, etc.. These are omitted from the model: only the constraints they impose on the possible orderings of tasking activities are retained. (This omission is a tacit assumption of zero execution time for these statements.)

Concurrency states can now be defined. Let T be the number of tasks occurring during execution of program S.

Definition

A concurrency state C is an ordered T-tuple (c_1, c_2, ..., c_T) where each c_i, $1 \leq i \leq T$, is a state node in some flowgraph of a unit of S, or else c_i is the marker "inactive". Each c_i denotes either the next state node to be executed in task i, or that task i is inactive.

A successor relationship exists between concurrency states. Let C= (c_1, ..., c_T) and C' = (c_1', ..., c_T') be two concurrency states for program S.

Definition

The set of successor states of concurrency state C, denoted succ(C), is the set of all C' such that

1) for all i, $1 \leq i \leq T$, either
   i) c_i' is an element of succ(c_i)
   ii) $c_i' = c_i$
   iii) c_i = "inactive" and c_i' = begin task i
   or iv) c_i = end task i and c_i' = "inactive"
2) there exists at least one c_j', $1 \leq j \leq T$, which represents application of case i), iii), or iv) above,
3) adherence to the tasking semantics of Ada is reflected in the application of the four cases above to the definition of each c_i', including selection of c_i' from succ(c_i).

If succ(C) is the empty set, then C is said to be a *terminal state.* The third part of the definition informally expresses the requirement that the successor relationship preserves the meaning of the rendezvous concept, the select mechanism, task termination in the presence of dependency relationships, and so forth. This is done subject to the general limitations of the static analysis model. (As used by the analysis algorithm some components of concurrency states have annotations to aid fulfillment of this requirement.)

Informally, the successors of a concurrency state are all those concurrency states which may follow occurrence of C in some execution. Let C' be a member of succ(C). If state C arises during execution of program S, then, with respect to synchronization activities, it is possible for S to directly progress to state C'. At least one task in the system must advance to a successor node; no task may advance further than that. Advancement to C' may involve the activation or termination of a task.

Last some concepts related to sequences of concurrency states are defined. A concurrency *history* is a sequence of concurrency states, where each element of the sequence C_k is a member of succ(C_{k-1}), for all $k > 1$. A history begins with the "initial" state of a system S, (begin <<main program>>, "inactive", ..., "inactive"). A history is thus a sequence of snapshots of the execution of S, starting with system invocation. A *proper history* is a finite history of which all elements are unique, save possibly the final element of the sequence. That is, "loops" in the concurrency history are prohibited. A *complete history* of a program S is the set of all proper histories of S.

The analysis technique of [Tayl83] can, in effect, produce the complete history for a program. Subject to the accuracy limitations described earlier, this complete history provides the information desired about the program's synchronization structure. When the complete history is generated the analysis describes all possible program actions under all possible external (real-time and implementation) conditions. Any possible deadlocks appear immediately as terminal states that have at least one active task. The complete set of all possible rendezvous can be determined from a single scan of the complete history, as can the set of possible parallel actions. Moreover, since the algorithm generates histories, the sequence of tasking activities leading to any state (such as a deadlock) can be readily seen. This history information can be used as a vital part of further investigation into the nature of the anomaly. (Such as investigation to ensure that the history does not involve an unexecutable path.)

Perhaps not so obvious, however, is the performance information derivable from the complete history. Performance bottlenecks can be detected, for instance, by scanning the history for states in which many tasks are all blocked on the same entry call. Alternatively, the history could be scanned for sequences of states in which one task is consistently found to be waiting through several state transitions before its service request is satisfied. Examination of history subsequences may also yield performance estimates. The number and type of synchronization activities occurring between states A and B could be noted. This information could be used, with other environmental information, as the basis for a performance estimate. Examination of different subsequences from A to B would yield upper and lower bounds on the number of interactions or delays occurring.

A final note concerning this static analysis technique is in order. As if the limitations on the applicability and accuracy of the analysis weren't enough, the algorithm is $O(n^T)$, where T is the number of tasks in the system, and n is the number of state nodes. Potentially a very large number of states may be generated, and such generation may take considerable time. This represents a further drawback which motivates the integrated approach described in Section 3.

2.3. Dynamic Analysis

Sophisticated dynamic analysis techniques for sequential software appeared in the late 1960's. Techniques applicable to concurrent software have only recently appeared, however, perhaps indicating again the additional complexities associated with concurrent systems.

Dynamic analysis techniques can be classified, somewhat arbitrarily, into schemes for aiding in documentation of run-time events, error monitoring, debugging, and testing. Different implementation techniques are appropriate for the different schemes. Referring for the moment to sequential software, the first category typically includes techniques for such things as tracing variable values and maintaining execution frequency counts. The second category, error monitoring, includes techniques that monitor for violation of implicit specifications of intent, e.g. array bounds violations or division by zero. Debugging refers to the process of isolating the source of an observed error.

Techniques for aiding this process typically include breakpoint and program state modification facilities. Testing, as used here, refers to the process of dynamically comparing a program to explicit specifications of intent, such as embedded program assertions.

The analysis objectives indicated in Section 2.1 cover topics in three of these four areas: only debugging is omitted. We will therefore describe advances in dynamic analysis techniques for concurrent software in the three remaining categories.

Substantial results have recently been achieved with respect to error monitoring. These results were originally presented in [Germ82], and most recently, in an updated form, in [Helm82]. These papers describe a technique for monitoring for deadness errors in Ada tasking. Using the terminology of static analysis, a deadness error is a terminal concurrency state having at least one active task.

The basic idea of the technique is to transform an Ada program P into another Ada program P' such that P and P' have the same set of possible deadness errors, but, during execution, P' will detect the imminency of a deadness error, report the condition, and allow the possibility of evasive action. The transformation accomplishes this by adding a monitor task to P which maintains, essentially, the current concurrency state. Each task in the system updates the monitor, telling it the task's next tasking activity, such as entry call, accept completion, or task completion. By doing a one state lookahead the monitor can detect a deadness error "just before it happens" and can thus raise an exception in the offending task, again "just before" the error would occur.

The concepts used in the definition of the deadness monitoring technique are quite similar to those used above for static analysis and will not be shown here. The dynamic technique is not beset by the same restrictions as the static technique however. The dynamic technique functions correctly in the presence of nearly all Ada82 tasking constructs. The use of access values and subscripts presents no problems. (One of the more interesting aspects of the implementation is in the unique identification of each task.) Furthermore no spurious errors are reported.

The instrumentation process itself is potentially efficient; some parts of the monitor task do not even require recompilation with each new program to be monitored.

Some interesting issues arise with regard to run-time efficiency though. Since dynamic analysis is being discussed it is clear that the impact of the real-time environment is felt, including the effect of delay statements. Unfortunately the error monitoring instrumentation imposes a potentially substantial amount of overhead. Not only is another task included in the program (the monitor) but the number of entry calls occurring leaps by a factor of three or more. Sensitive timing criteria may therefore be unsatisfiable. Then too, as mentioned earlier, the overhead induced by the instrumentation may cause an observed phenomenon to disappear (under the same set of external conditions) though the *potential* for that error still remains.

The prime limitation of error monitoring, of course, is that a batch of error free runs does not allow one to infer much about the correctness of the program, even with respect to the limited scope of deadness errors. A change in the underlying implementation, such as a new scheduler, may cause a crop of errors to appear, even when the program is run on the old test data.

The apparatus used to perform this error monitoring can be easily augmented to document many run time events of interest. Since the monitor is notified of all rendezvous, task initiations, terminations, and the like, it is a simple chore for it to produce a trace listing of these events. The implementation described in [Helm82] in fact allows this. Such tracing is analogous to the presentation of concurrency histories by the static analyzer.

Clearly there are many simple instrumentation schemes capable of generating other kinds of tracing information. These will not be considered here.

Lastly we briefly mention the subject of testing based on run time comparison of the code with embedded assertions. From the viewpoint of (only) being concerned with synchronization structure, it is necessary to express, in the assertions, the properties desired. Unfortunately an assertion language capable of expressing the desired properties has not been advanced. A significant related development, however, is the specification of the Anna language [Krie82]. Anna is a language for annotating Ada programs. Sufficient expressive power is provided to allow full specification of a sequential program's intended functionality. Anna currently omits any annotations for Ada's tasking constructs, but work towards its completion is ongoing.

3. Technique Integration

It should be clear from the foregoing descriptions that the static analysis and dynamic analysis techniques have complementary characteristics. Static analysis results can be definitive and informative, accounting for all possible program actions. But the application of the technique is limited to a subset of Ada and some of the analysis results require scrutiny to determine if the reported phenomena are indeed possible. Dynamic analysis has fewer limitations, but the meaning of the results obtained is not so clear (unless an error is discovered).

This complementary character of the individual techniques suggests that if they are applied in concert, several of their deficiencies may be eliminated. Several suggestions are made below, indicating the nature of this joint application. With regard to dynamic analysis the prime focus will be on error monitoring, though the other aspects will be addressed too.

The most obvious joint use of the techniques is to employ dynamic analysis in the further investigation of phenomena detected by the static analyzer. It may be, for example, that the particular scheduler used or the semantics of the guards on select statements preclude a (statically) reported deadlock from occurring. Dynamic analysis could monitor these conditions, watching for the error during testing.

Certain concurrency states may not be possible for other reasons: the static analyzer may have assumed the executability of an unexecutable path, for instance. In this case the concurrency histories may be of use in attempting to develop test data to force execution of the path. More generally the concurrency histories provide a guide to the development of a battery of test cases. (Other automated tools, such as symbolic executors, may be useful aids in this process as well.) In the restricted sense of testing a program's synchronization structure, the complete concurrency history can be used as a yardstick in evaluating the thoroughness of a test regimen. This appears to be potentially one of the most useful applications of the concurrency history information.

Furthermore, dynamic analysis could be used to investigate the performance properties of a program that were highlighted by the static analyzer. The queue length for entries could be monitored to see how often the bottlenecks predicted by static analysis occur.

A second obvious use of the joint application is to employ dynamic analysis in the investigation of aspects of the program inadequately handled by the static analyzer. That is, it would be appropriate to emphasize test cases which particularly exercise tasks that are objects of access variables, or entries that are subscripted.

It is also possible for static analysis results to help in reducing the overhead incurred by the dynamic techniques. If a subset of a program's task can be definitely shown to be error free, then the instrumentation used for their monitoring may be reduced or eliminated.

In a more speculative vein, it seems that the static analysis results could be used to guide limited instrumentation of timing-sensitive real-time programs. By examining the list of actions that can occur in parallel, for example, or perhaps a report of rendezvous that are guaranteed to always occur, it may be possible to identify places where instrumentation can be placed without upsetting critical timing.

Potentially one of the most significant problems with static analysis is the large number of concurrency states that a program may have. There may be so many that it would be infeasible to generate them all. Under such circumstances it may be possible for dynamic analysis to aid in the process of pruning the static analysis: cutting off exploration of uninteresting (hopefully unexecutable) histories.

Two approaches bear examination here. First, unit testing may provide solid data on minimal/maximal execution times for program segments. Further, data may be obtained on the scheduler used for a particular implementation. These could be supplied to the static analyzer such that histories violating either premise would not be explored. The results obtained would not be general, of course, but they would perhaps be useful.

A second approach is to include assertions in the Ada program indicating certain desired properties of the synchronization structure. Such assertions could make statements such as "task. T is terminated" or "tasks T1 and T2 are currently engaged in a rendezvous." The static analyzer would regard these statements as true, and would not explore histories contradicted by the assertions. Subsequent dynamic analysis would then monitor the assertions during testing. Although this again may be a practical approach, it is clear that the results of this kind of static analysis would not be reliable. Violation of an assertion during testing (or operation!) would invalidate any assurances previously produced.

Dynamically produced tracing information may also provide a basis for cooperation. If the dynamic monitor emits sufficient information to construct a time-stamped log of concurrency states (or an improved version of them - we regard the current definition as temporary until some experience is gathered), then a variety of useful information could be (statically) derived from it. Clearly derivable are a graph of clock time versus tasks active, statement of average entry queue lengths, the relative frequency with which different select alternatives are chosen, and statement of the actual delay occurring at delay statements (as compared with the minimum time required). Speculating once more, it seems that capturing the concurrency states during execution could even enable interactive debugging of the synchronization structure. Each state captures the essential properties of the program: one could "back up" to a given condition (concurrency state), change the program, then resume execution based on information in the state. Perhaps more realistically, it would certainly be possible to use static analysis to examine all the possible successors of a concurrency state that had been captured dynamically. Used in this way the static analyzer would not generate the complete history for a program, only the the histories rooted at the state supplied by dynamic analysis.

4. Implementation Issues

Section 3 considered the ways in which the two analysis techniques are logically related. Some potential also exists for an integrated implementation of the two.

The framework within which such an implementation should occur is, obviously, a full Ada programming environment, such as Arcturus [Stan81]. Joint application of these techniques is clearly not sufficient to meet all the V&T demands of a project. The tools must be applied in a situation where other techniques can conveniently be brought to bear as required.

In this context an immediate opportunity for integration is through use of an intermediate program representation. Neither technique needs to, or should, deal with source text representations. Reductions of a program (by the static analyzer) or transformations of a program (by the dynamic analyzer) can more efficiently be applied to, say, an attributed parse tree. This remains true when embedded assertions are used by the techniques.

Potential also exists for directly aiding joint use of the techniques.

Several of the ideas in the preceding section imply a back-and-forth application of the techniques. Only a few histories or even a partial history may be produced by the static analyzer before it is appropriate to perform some dynamic analysis. After that, additional directed use of the static tool may be desired. This would be promoted by storing the concurrency histories in a "database" that would allow additions, deletions, and so forth. Further integration would be aided by using an interface program to hide the contents of the database. As particular histories were required, by the user or the dynamic analyzer, they would be retrieved if currently stored or else automatically generated. The key is that the static tool must not be monolithic.

Joint use of the tools would also clearly be promoted by relieving the user of mechanically transferring information between tools. For example, if it was desired to automatically generate test data on the basis of a concurrency history, then the transfer of the detailed history to the data generator should not require manual intervention.

Finally, it should be the goal of any implementation to provide a user interface such that not only are the detailed mechanisms of the tools hidden, but the methods as well. That is, the user should be able to concentrate on asking questions about the behavior of a program without being concerned with how the answers can be developed. Though this goal may not be immediately achievable, developers should not lose sight of it.

5. Conclusion

Two techniques for analyzing the synchronization structure of concurrent programs have been overviewed and compared in this paper. Each technique, static analysis and dynamic analysis, has its own strengths and weaknesses. Fortunately these characteristics are complementary in several respects. Joint application of the techniques is therefore both natural and advantageous. Some specific ways in which the techniques can operate synergistically were presented. Finally a few implementation principles were outlined.

We conclude by urging that the techniques be incorporated into an Ada programming environment such that their joint use is facilitated. Until better techniques for the V&T of concurrent software come along, the approach presented here is both worthwhile and feasible.

Acknowledgements

Several of the implementation ideas expressed here originated with the Toolpack project [Oste82]. This work was sponsored (in part) by the Defense Advanced Research Projects Agency of the United States Department of Defense under contract MDA-903-82-C-0039 to the Irvine Programming Environment Project. The views and conclusions contained herein are those of the author and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the United States Government.

References

[Germ82]
German, Steven M., David P. Helmbold, and David C. Luckham. Monitoring for deadlocks in Ada tasking. Proceedings of the AdaTEC Conf. on Ada, Arlington, VA (October 4-8, 1982), pp. 10-25.

[Helm82]
Helmbold, David P. and David C. Luckham. Techniques for runtime detection of deadness errors in Ada tasking. Preliminary Draft, Computer Systems Laboratory, Stanford University, Stanford California.

[Hoar78]
Hoare, C.A.R. Communicating sequential processes. *Communications of the ACM,* Vol. 21, No. 8 (August 1978), pp. 666-677.

[Ichb82]
Ichbiah, Jean D., et.al. Reference Manual for the Ada Programming Language (Draft Revised MIL-STD 1815). United States Department of Defense, July 1982.

[Krie80]
Krieg-Brueckner, Bernd and David C. Luckham. Anna: Towards a language for annotating Ada programs. *Sigplan Notices,* Vol. 15, No. 11 (December 1980), pp. 128-138. (Proceedings of the symposium on the Ada language.)

[Krie82]
Krieg-Brueckner, Bernd, David C. Luckham, Friedrich W. von Henke, Olaf Owe. Reference manual for Anna, a language for annotating Ada programs (Draft). Supercedes [Krie80].

[Mart77]
Martin, Fred H. HAL/S - The avionics programming system for shuttle. Proc. AIAA Conf. Computers in Aerospace, Los Angeles, CA (November 1977), pp. 308-318.

[Oste82]
Osterweil, Leon J. Toolpack - An experimental software development environment. Proc. 6th Intl. Conf. on Software Engineering, Tokyo, Japan (September 1982), pp. 166-175.

[Stan81]
Standish, Thomas A. Arcturus - An advanced highly integrated programming environment. In *Software Engineering Environments,* H. Hunke, editor, North Holland, 1981.

[Tayl83]
Taylor, Richard N. Static analysis of concurrent Ada programs. *Communications of the ACM,* 1983 (to appear). Also Technical Report Number DCS-10-IR, Department of Computer Science, University of Victoria (May 1981).