Verifying Properties of Distributed Systems: Prospects for Practicality

Lori A. Clarke
Leon J. Osterweil
University of Massachusetts Amherst

Abstract

Industry is rapidly embracing distributed systems. Although there are many advantages to distribution, such systems are certainly more difficult to understand and thus more vulnerable to errors. Unfortunately, insufficient thought is being given to how to ensure the reliability of such systems. Whereas dynamic testing approaches provide increased confidence in the reliability of sequential systems, for distributed systems even the repeatability of a given test execution result cannot be counted upon. Alternative validation approaches, such as formal verification and static analysis, are usually complicated to use and often intractable. In this talk we describe a static analysis tool, called *FLAVERS*, that addresses these limitations and offers a promising approach for validating distributed systems..

FLAVERS was designed to be used by typical developers on real distributed software to prove important user-defined properties about the behavior of such systems. Industrial developers have been successful in applying *FLAVERS* to a diverse set of projects. But these successes also indicated hurdles that remain before *FLAVERS*, or similar techniques, can be expected to see wide acceptance. These hurdles, and our approaches for overcoming them, will be described in this talk.

Keywords: static analysis, data flow analysis, program verification, distributed systems

Introduction

Increasing numbers of industrial systems are being implemented as distributed systems, largely because distribution can improve execution speed and response time, can increase robustness, and can provide the flexibility needed to facilitate component-based(plug and play) approaches to software development. There has been considerable interest in developing glue technology or middleware to facilitate interoperability (e.g., [2] and [12]) to support component-based development. Unfortunately, there has been far less interest in the critical issue of how these distributed systems are be validated to assure that they will perform acceptably.

Because of non-deterministic computation and the potential for the parallel execution of the statements in different threads to execute in various relative orderings with respect to each other, it is extremely difficult to write such systems without inadvertently introducing subtle errors, such as race conditions and deadlock. These characteristics also make it more difficult to reason about such systems in order to gain Confidence that they will always perform as required, and will never execute dangerous disallowed sequences of activities. Testing is the usual approach to gaining such confidence. But because distributed systems may execute quite differently on two different runs using the same input data, a successful test execution provides fewer assurances for a distributed system than for a sequential one Thus, if distributed systems are to be widely and successfully used with confidence, better

methods for raising confidence in those systems must be developed and transitioned to industry.

In this paper we describe *FLAVERS* [7], a system that helps verify user-specified properties of sequential and distributed systems. *FLAVERS* performs a static, global analysis of a system, using relatively efficient data flow analysis techniques, similar to those used in optimizing compilers. The goal of a *FLAVERS* analysis is to determine if a specified property will hold on all or no executions of a system. If *FLAVERS* determines that the property is valid, then the property is guaranteed to hold on all possible executions of the system, for all possible input data. If *FLAVERS* cannot determine definitively that the property holds, then the user is shown a path through the system, whose execution would violate the property. If this path is executable, then *FLAVERS* has found an error that the developer must fix. Sometimes, however, the path does not correspond to a legitimate execution of the system, in which case additional information can be supplied to *FLAVERS* to help it to suppress consideration of this path.

FLAVERS has been successfully used in several research experiments as well as on a few industrial applications. Some of these experiments have shown how it can be applied to the example (and contrived) programs that are used repeatedly in the literature [4]. Others have demonstrated how it can be applied to communication protocols to verify properties that require modeling the external environment [13]. Since *FLAVERS* works on a very generic model of execution, others have been used to show that it could be applied to high-level software architectures [15]. Industrial colleagues have demonstrated the successful application of *FLAVERS* to Advanced Distributed Simulation systems and to the verification of *High Level Architectural* (HLA) requirements [18].

One of our goals was to develop a technique that could be used outside the research laboratory. We wanted to develop an approach that was efficient and scaleable, that could be applied to real programming languages (there are implementations for Ada and C++, with support for Java underway), and that was easy enough to be used by the stronger members of a typical industrial software development project. Although *FLAVERS* has overcome many of the limitations of other verification approaches, and in many ways is close to satisfying our goal, we have discovered that there are several other technical hurdles that must be addressed before *FLAVERS* can be more readily transitioned to industry. In the remainder of this paper, we first provide an overview of *FLAVERS*, so the reader has a better understanding of how this approach works. We then describe the obstacles that we have encountered in our laboratory or observed when working with industrial users and outline our plans to remove, or at least diminish, these obstacles.

Related Work

Testing executes a program to determine whether its actual runtime behaviors conform to its expected behaviors. Since the number of possible program executions is usually excessively large, it is impractical to use testing to demonstrate the absence of errors. Moreover, for distributed systems, testing does not even guarantee that the system will consistently work on the validated test cases. Formal verification employs mathematical reasoning to prove the absence of errors by showing analytically that a program satisfies or contradicts a given property. Formal verification techniques can formulate and prove intricate properties but require a significant amount of mathematical expertise, and hence are human intensive and error prone. Static analyses can also demonstrate the absence of certain classes of errors without actually executing a program. Unlike formal verification, they entail little human intervention, making them less costly and less error-prone to use.

Static analysis of distributed systems is usually based upon reachability analysis, necessary condition analysis, or data flow analysis.

Reachability analysis enumerates all possible execution states, which in the worst case is exponential in the number of tasks [19]. Several approaches have been suggested for optimizing such approaches [3, 5, 9, 10, 13, 14]. These approaches significantly improve the feasibility of reachability methods, but in general are still prohibitively expensive to use with industrial-sized distributed systems.

Necessary condition analysis encodes the property and the necessary conditions for execution as linear inequalities whose solutions determine satisfaction of the specification [1]. While this technique has exponential bounds on its running time, it has been successfully applied to a number of programs [6].

While the bulk of data flow analysis research has been aimed at optimization, recent work has applied data flow analysis to the verification of distributed software [17,11]. Data flow analysis is promising in that it usually has a low-order polynomial bound on the computation time. Most of the work in this area has been directed at the analysis of restricted classes of behaviors, such as deadlock. Olender and Osterweil demonstrated how data flow analysis could be used to verify user-specified properties of sequential systems [15]. *FLAVERS* extends their approach to distributed systems. In addition, *FLAVERS* addresses one of the major limitations of static analysis techniques in that it provides a systematic approach for sharpening the results.

Overview of *FLAVERS*

FLAVERS, **Fl**ow **A**nalysis for **VER**ifying **S**ystems, can be used to verify user-specified properties about sequential or distributed programs. Using *FLAVERS*, an analyst must first define a set of program events of interest and then formulates the properties to be checked as sequences of those events. Properties are specified in the form of *quantified regular expressions* (QREs)[8, 15]. These expressions consist of the alphabet of the property, which is the set of events referenced in the property, an indicator of whether the property should hold on all or on no executions of the program, and a regular expression that describes the event sequences. After the analyst writes a property in the QRE language, it is submitted to *FLAVERS* for a syntactic check and a translation into a *deterministic finite state automaton* (DFSA) form. The reasoning component of *FLAVERS* only uses this DFSA representation of the property, thus any specification language that could be translated into a DFSA could be used to represent properties.

FLAVERS must determine if there are any executions of the program that could violate the property specification. Thus, if a property specifies that a sequence of events should hold on all executions of the program, *FLAVERS* needs to determine if there are any executions for which this sequence does not hold. In order to determine what a program's execution sequences are, a graph model of the program is created where the nodes must be annotated appropriately with events that come from the alphabet of the property. For example, if the property is describing the order of inter-task interactions, any node in the program that represents an inter-task interaction would need to have an annotation indicating which task interaction occurs when the code associated with this node is executed. If we are interested in the order that certain procedure calls could occur, then each node associated with such an invocation would need to be annotated with an event that indicated that such a procedure is called. By carefully modeling the program and selecting the granularity of the nodes, any arbitrary action in the program could be associated with an event that is used in a property.
.

Given a property and a program representation annotated with the events in the property, *FLAVERS* uses data flow analysis techniques to determine whether the property indeed holds. The results of an analysis may indicate that the property holds on *all* executions of the program, *no* executions of the program, or *some* executions of the program. The first two results are called *conclusive*, while the latter is called *inconclusive*. One of the reasons for an inconclusive result is that an error has been found, and thus the property does not hold over some executions of the program. Another reason, however, could be the potential imprecision of the model of the program that was used during the analysis. As is typically the case with static analyses, to assure that the results are always *onservative*, the model of the program overestimates the possible executable behaviors of the program. If the results of an analysis are inconclusive because the property holds on all real executions of the program but does not hold on some infeasible executions of the model, then the reported violation is *spurious.*

All static analysis approaches are hampered by spurious results. *FLAVERS* addresses this problem by allowing the analyst to add control and data flow information incrementally to the analysis to increase the precision. This information is added via *feasibility constraints***,** which are also represented by DFSAs. Each feasibility constraint models an aspect of the program behavior that is not captured precisely in the model of the program. For example, the analyst might decide that the results are inconclusive because the value of some variable used as the sentinel in a conditional statement affects the precision of the results. With the help of automated tools provided by *FLAVERS*, the analyst can build an automaton representing the behavior of that variable. The resulting feasibility constraint is incorporated in the next analysis run. Since the modeled behavior of this variable restricts the set of all possible behaviors of the program, the overall program behavior is represented with a greater degree of accuracy, thereby reducing the chance of a spurious result. We have found that typically with a few iterations, analysts can find an error in the program or successfully establish a conclusive result.

Feasibility constraints are a very general concept that can be used to extend our modeling capabilities. Feasibility constraints can be used not only to improve the model of the program but they can also be used to model aspects of the overall systems that are not explicitly represented in the available code. For example, feasibility constraints can be used to create assumptions about the behavior of the environment in which the program executes. Or to represent the behavior of missing software components. When these components become available, one should prove that each corresponds to the feasibility constraints used to model its behavior. Thus, *FLAVERS* supports a natural compositional approach to verifying large systems.

Hurdles to Adoption

Although static analysis approaches offer numerous advantages over the more traditional dynamic testing approaches, success in applying static analysis entails meeting some significant challenges. In this section we identify some of these challenges and indicate how we are addressing them. These challenges must be met successfully in order to expedite more widespread adoption of this promising, and increasingly essential, static analysis approach to verifying properties of distributed systems.

Property specification.

Careful and precise specification of the property to be verified is clearly an essential part of dataflow analysis. As noted above, in *FLAVERS* the property is specified by means of a QRE, a construct that offers sufficient semantic power to express a wide range of important

properties of concurrent and distributed systems. Our experience, however, indicates that it is difficult for novice (and even experienced) users to write QRE's that precisely and correctly express desired properties. We have noticed that experienced users rather quickly evolve idioms that are effective in capturing the nuances needed for precise specification of properties, and that these idioms often are clearly described using structured natural language.

Thus, for example, the property, "The elevator never moves with its doors open," is surprisingly tricky to express precisely and correctly using event sequence formalisms such as QRE's. The desired behavior is that, for all execution sequences, once an "open door" event occurs, there can be any sequence of events, except a "move elevator" or "close door" event, followed by a "close door" event, which can then be followed by any sequence of events except a "move elevator" or "open door" event, followed by a "move elevator" event. Translation of this natural language specification into the precise QRE notation represents an additional challenge, and consequent adoption barrier. In our work we are identifying a structured natural language designed to support the use of the idioms that we have found most useful, and that are particularly straightforwardly translatable into QRE's. This work will build upon some recent work that is attempting to identity common specification patterns for several different specification languages [8]. We plan to develop a translation system based on generative grammars that will support the automatic translation of comfortable natural language property specifications into precise and correct QRE's.

System Annotation

Another obstacle to effective exploitation of static dataflow analysis is the need to correctly annotate the graph representation of the system being analyzed with indications of the exact locations at which all of the events of interest occur. It is critically important that no event locations be overlooked, as this might lead to erroneous analytic results. It is also critically important that all annotations reflect events that can actually occur and that they all be correctly placed. In general it is possible that an event of interest may potentially correspond to an arbitrarily long and complex sequence of system code. Thus, the problem of unerringly identifying all possible events can be arbitrarily difficult.

Our experience, however, has indicated that many important properties are often expressed in terms of events that are relatively easy to identify from direct and straightforward examination of source code. Thus, for example, many of the properties that we have studied are defined in terms of events all of which occur as procedure invocations. Often the procedure invocation itself is the event of interest. Sometimes invocation of a procedure with a particular parameter is the event of interest. In such cases it is easy to conceive of an automated tool that can be of considerable help in assuring that the needed annotations are carried out completely and correctly.

We are currently developing such a tool. Based upon the notion of a concordance, this tool uses parsing technology to identify such syntactic tokens as keywords and identifiers in programming language source text. The tool then supports users in specifying simple configurations of these tokens and identifying them with specific named events. The tool then works with our front-end technology to annotate the graph representation of the source text with the events that correspond to the configurations that the user has specified. While some types of annotations may be difficult or impossible to specify using our tool, our experience, nevertheless, suggests that most will be automatable in this way. This will leave the user free to concentrate attention on a smaller number of annotations, thereby increasing the chance that all annotations will be done correctly. This should reduce user worries about incorrect annotations undermining the accuracy of the analyses and, therefore, reduce this obstacle to adoption.

Evaluating Data Flow Anomaly Reports

While the automatic generation of diagnostic reports is one of the positive features of static dataflow analysis, the reports can, nevertheless, sometimes be cryptic and potentially misleading. Thus, recall that *FLAVERS* will report the possibility of an anomaly if the property can be violated along some path through the graph representation of the program. In addition, as noted above, the violation may be on a path that cannot be executed. Thus, reports of possible illegal executions must always be viewed with some skepticism, until the path or paths causing them have been specified and studied for executability. Indeed, early dataflow analyzers tended to produce large quantities of reports of possible illegal behaviors, and it was not unusual for many of them to have been based upon unexecutable paths. This was an obstacle to adoption of these early systems.

Thus, in our work we have attempted to augment *FLAVERS* with additional tools designed to help the analyst separate such spurious results from real error phenomena. Our first step was to create a tool that generates paths along which illegal sequences of events occur and makes specifications of these paths available to the human user. Tracing one such path is a straightforward, but not completely trivial task. The work of this tool is complicated, however, by the fact that in most cases there are infinitely many such paths (arising, for example, from the possibility of infinitely many different numbers of iterations of program loops). Identification of a small number of such paths that are "significantly" different from each other seems to be a difficult task to which more work must be devoted. But the current tool has still proven to be of substantial help to analysts who wish to understand how illegal sequences of events might possibly occur in their systems.

As already noted, obliging the human analyst to determine whether paths are unexecutable, and therefore irrelevant to the analysis, has proven to be an irritant that has been an obstacle to adoption of this technology. While it is impossible to write a system that always unerringly determines the executability of an arbitrary sequence of program statements (it is equivalent to the Halting Problem), our experience has nevertheless indicated that it is often quite straightforward to demonstrate the unexecutability of many of the paths in the sorts of programs we have seen. Thus, as has been described above, we have augmented *FLAVERS* with feasibility constraint capabilities for pruning from consideration large classes of unexecutable paths through our graph representations. Our early experience with this capability suggests that iteratively adding additional information results in an incremental reduction in the number of spurious reports. Unfortunately, as more information is modeled, the worst case bound on the analysis increases. In practice, however, this additional information reduces the search space and thus often results in a reduction, not an increase, in the execution time. The major hurdle, therefore, is to help users view the paths that are associated with property violations and to help them formulate effective feasibility constraints. We expect that the availability of such capabilities will result in significant reduction in the number of spurious diagnostic messages, with a consequent lowering of this important barrier to adoption.

Understanding Data Flow Analysis

While the preceding discussion has identified specific technical issues that we are addressing in order to reduce resistance to adoption of this technology, it has not addressed a more fundamental difficulty, namely the widespread ignorance of the very nature of static analysis as a complement to the more traditional dynamic testing approaches. The first prototype demonstrations of the viability of static dataflow analysis for program error diagnosis were carried out nearly 25 years ago [16]. Yet, we have noted that surprisingly few actual practitioners know about this technology. Indeed, we have been startled to find

that practitioners often question the very possibility of this type of analysis. It appears that the dynamic testing paradigm is so deeply ingrained in many programmers' consciousness that it has become the fixed perspective from which all other quality determination approaches are considered. Because testing is aimed at identifying the presence of faults, and is essentially incapable of being used to demonstrate the absence of faults, many programmers seem to refuse to even believe that such demonstrations are even possible.

Approaches to addressing this fundamental problem seem to require going beyond the devising of new tools and technologies. We believe that the fundamental difficulty here stems from lack of a clear grasp of the fundamental goals and requirements of testing and analysis. All too many programmers seem to view the running of test cases and examination of results as goals unto themselves, rather than as means to an end. All too many programmers seem to have not even thought about what that end might be. As a consequence all too many testing activities are directionless, leaving testers unable to articulate either what they have learned or established at the conclusion of a testing activity or even what they were trying to learn. It is our contention that programmers should be attempting to demonstrate the absence of errors from their programs, and to establish that their programs either do or do not have certain important properties and characteristics. Dynamic testing can encourage and support such beliefs, but can generally not establish them as proven truths. Thus, we believe that educating those who must demonstrate program quality about the value of establishing goals and requirements for testing will help to increase interest in static analysis, and lead to eventual acceptance of this approach as a viable complement to dynamic testing.

Integrating Static Analysis

Acceptance of static analysis as a complement may be a first step towards more widespread adoption of static analysis, but adoption is not likely to come until its capabilities have been effectively synergized as complements to those of dynamic testing. Our experience with static analysis has suggested strong synergies with dynamic testing. Thus, for example, we do not suggest that static dataflow analysis be the first diagnostic technique employed in attempts to demonstrate the absence of errors from a program. Dynamic testing should generally be performed first, as the generation of rough preliminary test data sets, and evaluation of results of test execution, is usually straightforward and effective in finding many errors. But, after incrementally thorough dynamic testing regimes have failed to turn up errors, it is often the case that static dataflow analysis can and should be used to demonstrate that errors are absent. Correspondingly, as incremental modeling for the purpose of sharpening dataflow analysis becomes increasingly difficult, it is reasonable to consider the use of dynamic testing to explore the executability of paths along which illegal event sequences have been determined.

As our understanding of the strengths and weaknesses of static dataflow analysis has increased, it has become clear that it needs to be effectively integrated with dynamic testing. In recent work we have started to study the definitions of specific testing/analysis processes that cleanly and effectively synergize these two approaches. By promulgating definitions of these integrated testing and analysis processes, and demonstrating their efficiency and effectiveness, we hope to be able to establish both the importance and the practicality of static dataflow analysis, and to use these process definitions to lower barriers to adoption still further.

References

[1] 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,* Vol.17 n.11, pp. 1204-1222 November ,1991.

[2] D. J. Barrett, L. A. Clarke, P. L. Tarr and A. E. Wise, "A Framework for Event-Based Software Integration ", *ACM Transactions on Software Engineering and Methodology (TOSEM)*, Vol. 5, n. 4, pp. 378-421, October, 1996.

[3] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, L .J. Hwang, "Symbolic Model Checking: States and Beyond", *Proceedings of the Fifth Annual IEEE Symposium on Logic in Computer Science*, 1990.

[4] A. T. Chamillard, G. A. Avrunin and L. A., Clarke. *An Empirical Comparison of Static Concurrency Analysis Techniques*, Department of Computer Science, University of Massachusetts",TR 96-84, May, 1996.

[5] S. C. Cheung, J. Kramer, "Compositional Reachability Analysis of Finite-State Distributed Systems with User-Specified Constraints", *SIGSOFT*, pp.140-150, 1995.

[6] J. C. Corbett, "Evaluating deadlock detection methods for concurrent software", *IEEE Transactions on Software Engineering,* Vol. 22, n. 3, pp.161-180, March, 1996.

[7] M. Dwyer and L. A. Clarke , "Data Flow Analysis for Verifying Properties of Concurrent Programs", "*Proc. of the Second ACM SIGSOFT Symposium on the Foundations of Software Engineering*", pages 62-75, New Orleans, December, 1994.

[8] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett, "Property Specification Patterns for Finite-State Verification", *Technical Report 97-049*, Department of Computer Science, University of Massachusetts at Amherst, August ,1997.

[9] P. Godefroid and P. Wolper, "Using Partial Orders for the Efficient Verification of Deadlock Freedom and Safety Properties", *Proceedings of the Third Workshop on Computer Aided Verification*, pp. 417-428, July, 1991.

[10] G. J. Holzmann, P .Godefroid, and D. Pirottin, "Coverage preserving reduction strategies for reachability analysis", *Proc. 12th Int. Conf on Protocol Specification, Testing, andVerification, INWG/IFIP*, Orlando, Florida, 1992.

[11] S. P. Masticola and B. G. Ryder, "A Model of Ada Programs for Static Deadlock Detection in Polynomial Time", *Proceedings of the Workshop on Parallel and Distributed Debugging ACM*, pp. 97-107. May, 1991.

[12] M. J. Maybee, D. H.Heimbigner, and Leon J. Osterweil, "Multilanguage Interoperability in Distributed Systems: Experience Report ", *Proceedings of the Eighteenth International Conference on Software Engineering*, pp. 451-463, Berlin, Germany, March 25-30, 1996.

[13] G. Naumovich, L. A. Clarke, and L. J. Osterweil, "Verification of Communication Protocols Using Data Flow Analysis". in *Proceedings of SIGSOFT'96: Fourth*

Symposium on the Foundations of Software Engineering, pages 93-105, San Francisco, October 1996.

[14] G. Naumovich, G. S. Avrunin, L. A. Clarke, and L. J. Osterweil , "Applying Static Analysis to Software Architectures", *Proceedings of the European Software Engineering Conference combined with the SIGSOFT Symposium on Foundations of Software Engineering*, Zurich, Switzerland , September, 1997.

[15] K. M. Olender and L. J. Osterweil , "Interprocedural Static Analysis of Sequencing Constraints", *ACM Transactions on Software Engineering and Methodology*, pp. 21-52, Janaury, 1992.

[16] L. J. Osterweil and L. D. Fosdick, "DAVE - A Validation, Error Detection, and Documentation System for Fortran Programs", *Software Practice and Experience*, pp. 473-486. October, 1976.

[17] J. H. Reif and S. A. Smolka, "Data Flow Analysis of Distributed Communicating Processes", *International Journal of Parallel Programming*, Vol. 19, n. 1, 1990.

[18] SAIC, "Advanced Interoperability Technology Development: Investigating Static Data Flow Analysis for Advanced Distributed Simulation Verification, Final Technical Report", *Science Applications International Coproration*, Orlando, Florida, May, 1997.

[19] R.N. Taylor, "Complexity of Analyzing the Synchronization Structure of Concurrent Programs", *Acta Informatica*, Vol. 19, pp. 57-84, 1983.

[20] A. Valmari, "A Stubborn Attack on State Explosion: Computer-Aided Verification 90" In. E. M. Clarke and R. P. Kurshan (Eds.), *Number 3 in DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pp. 25-41, American Mathematical Society, Providence , RI, 1991.

[21] W. J. Yeh and M. Young, "Compositional Reachability Analysis Using Process Algebra", *Proceedings of Symposium on Testing, Analysis and Verification*, October, 1991.