Architecture-level Dependence Analysis in Support of Software Maintenance

Judith A. Stafford and Alexander L. Wolf

Software Engineering Research Laboratory
Department of Computer Science
University of Colorado
Boulder, CO 80309 USA
{judys,alw}@cs.colorado.edu

1 Introduction

Software maintenance is the most costly phase of software development. Two factors contributing to the high cost are the late detection of system failures and the increasing difficulty of understanding a system as it ages. The advent of formal architecture description languages (ADLs) provides an opportunity to reason about the correctness of a system at the earliest stages of development as well as at a high level of abstraction throughout its life.

ADLs capture information about a system's components and how those components are interconnected. Some also capture information about the possible states of components and about the component behaviors that involve component interaction; behaviors and data manipulations internal to a component are typically not considered at this level. Techniques have been developed for architectural analysis that can reveal problems such as potential deadlock and component mismatches [2, 12, 14, 18].

In general, there are many kinds of questions one might want to ask at an architectural level for maintenance purposes as varied as reuse, reverse engineering, fault localization, impact analysis, regression testing, and workspace management. These kinds of questions are similar to those currently asked at the implementation level and answered through static dependence analysis techniques applied to program code. It seems reasonable, therefore, to apply similar techniques at the architectural level, either because the program code may not exist at the time the question is being asked or because answering the question at the architectural level is more tractable than at the implementation level.

Our research takes a broad view of dependence relationships that is appropriate to the concerns of architectures and their attention to component interactions. In particular, both the structural and the behavioral relationships among components expressed in current-day formal ADLs, such as Rapide [13] and Wright [2], are considered. We are developing an architecture-level dependence analysis technique, called *port-to-port chaining*, and implementing the technique in a tool called Aladdin [24].

2 Chaining Through Aladdin

As mentioned above, there are many kinds of questions asked at the architectural level. These include the following.

1. If this component is to be reused in another system, which other components are also required?
2. If this component is communicating through a shared repository, with what other components does it communicate?
3. For a given component with a given safety rating, are there components in the system that it could affect that have a higher safety rating?
4. If a change is made to this component, what is the minimal set of test cases that must be rerun?
5. If a failure occurs, what is the minimal set of components that must be inspected during the debugging process?

These questions share the common theme of identifying the components of a system that a particular component either is dependent on or supports in some way. Thus, Aladdin allows an analyst to pose queries about what architectural elements can affect or be affected by a specified architectural element. The answer to such a query is a *chain* of dependent component communication ports given in the form of a directed graph that the analyst can use in studying the system.

This work was supported in part by the National Science Foundation under grant CCR-97-10078 and by the Air Force Materiel Command, Rome Laboratory, and the Defense Advanced Research Projects Agency under Contract Number F30602-94-C-0253 and F30602-98-2-0163. The content of the information does not necessarily reflect the position or the policy of the U.S. Government and no official endorsement should be inferred.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ISAW3 Orlando Florida USA
Copyright ACM 1998 1-58113-081-3/98/11...$5.00

The first three questions in the list above are architecture-based questions answerable through analysis of the architectural description of the system alone. Questions 4 and 5 are questions about the implementation that can be answered with the aid of architecture-level analysis. They require the ability to maintain a correspondence between elements of the implementation and elements in the architectural description. Maintaining the correspondence is made difficult by the fact that the architectures reflected in implementations tend to drift from their description in ADLs. At least three different approaches to providing this correspondence are being investigated: code generation [25, 27], architecture refinement [16], and architecture recovery [5, 15, 17]. All have restrictions that make them less than completely useful. The value of architectural description throughout the life of a system will not be fully realized until there is support for precise, bi-directional mappings between the implementation and the architectural description.

When designing a static dependence analysis tool, one has a choice between finding all possible dependencies through exhaustive search, such as is done when a program dependence graph (PDG) [7] is constructed, or by identifying dependencies related to a particular item of interest as needed in a demand-driven approach [6]. Information gathered in a specific search can be cached and used when applicable as a basis for future searches. Aladdin uses a demand-driven approach, constructing chains in response the queries made by a user.

Architecturally, Aladdin is similar to ProDAG [22], an implementation-level dependence analysis tool for Ada and C++ programs. ProDAG uses a the former approach to identifying dependencies based on the program dependence relationships defined by Podgurski and Clarke [20]. Dependence analysis is performed by both ProDAG and Aladdin in a two-step process. First, a language-specific intermediate representation is created, and then language-independent analysis is performed over this representation. In Aladdin, the representation consists of a set of cells, where each cell represents the set of dependences that could exist between a given pair of architectural elements.

In general, dependence analysis works by gleaning information from some formal description, and is based on inspections of pairs of elements, the relationships between them, and the way their interactions contribute to overall system behavior. The structural and behavioral relationships that exist between elements appearing in both program code and architecture descriptions appear to be quite consistent. Examples of structural relationships are textual inclusion and inheritance. Examples of behavioral relationships are various kinds of flows, such as causality, temporal ordering, and state transition.

There appear to be two primary differences between implementation and architectural dependence analysis. First, consider the kinds of elements that are the objects of the analysis. In implementation dependence analysis one reasons in terms of variables, statements, and procedures. In architecture dependence analysis one reasons in terms of components, connections, and ports. Second, and perhaps more importantly, the execution models of implementation languages and ADLs are quite different. Implementation dependence analysis typically (if not exclusively) is defined for sequential, procedural languages. The trend in ADLs is toward the use of concurrent, event-based execution models. To our knowledge, static dependence analysis techniques for such models have not previously been defined. The analysis of posets in Rapide is a dynamic analysis technique. The current version of Aladdin operates on architecture descriptions written in Rapide, which embodies an event-based execution model. In the future we will be looking at Wright and the C2 ADL, both of which also have event-based execution models.

3 Refining Chains

A difficulty associated with static dependence analysis is finding the right balance between the relative simplicity and lower cost of conservative analysis and the increased complexity and cost associated with increased precision. The development of efficient and precise dependence analysis algorithms continues as an active area of research in the programming languages and software engineering communities (e.g., [3, 8, 10, 19]).

One aspect of this research is the investigation of means for increasing the precision of chains. The ability to model internal component behavior provides such an opportunity. In an entirely structural architectural description, such as that shown on the left side of Figure 1, each component is regarded as a black box and only the component interface and possible port-to-port connections between components are described. When some description of the internal component behavior is provided it is possible to make some observations about the dependences among entrance and exit ports of a component's interface as shown in the enhanced architecture on the right side of Figure 1. These internal connections provide opportunity for refining dependence chains through improving our knowledge of which entrance ports could have contributed to the stimulation of a particular exit port.

As an example, The solid lines in Figure 1 show that parts of the architecture that would be included in a query about what parts of the architecture could have been involved in the stimulation of the circled port. When the intra-component connections are included one is able to completely ignore the dashed component.

An exit port may be stimulated directly as a response to the stimulation of an entrance port of the component's interface. This is the situation depicted in Figure 1. Or it may be influenced by the internal behavior and data manipulations of the component. Aladdin uses a summarization algorithm to safely abstract away the details of the component's internal behavior resulting in connections only between ports of components' external interfaces.

Additional methods for improving the precision and efficiency of Aladdin's analysis include:

- improving precision of chains through the evaluation of patterns of events used as triggers and constraints on behavioral and state transitions, and
- the application of data flow analysis techniques to variables used as guards and communication elements, as well as

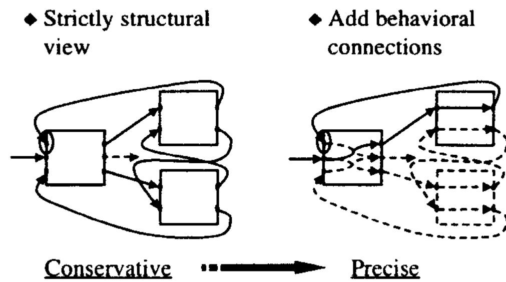

Figure 1: Intra-component Connections Improve Precision.

- improving efficiency by caching chains as they are created for use as bases for calculating responses to later queries.

4 Related Research

Our work on architecture dependence analysis builds on prior work in three primary areas: novel approaches to slicing, traditional dependence analysis techniques, and applications of static concurrency analysis tools to architecture descriptions.

Sloane and Holdsworth [23] suggest new applications for program slicing [26], in which the basis for analysis includes aspects other than traditional data and control flow. They present syntactically based generalized slicing for analysis of non-imperative programs. We agree with the spirit of this work and are pursuing a similar goal, but in the particular context of software architectures and, primarily, event-based execution models.

The work most closely associated with our research is Zhao's use of a system dependence net to slice architectural descriptions written in the Wright ADL [28]. His method produces a reduced architectural description containing just the lines of ADL code that could be associated with a particular slicing criterion. While the goals of his research are very similar to ours, the techniques are quite different. His focus is on the actual lines of code as is the case with program slicing. The object is to produce a reduced description that describes an architecture with the reduced functionality. While we use an architectural description to determine components and interface ports that can be ignored or that must be included for certain purposes, we do not focus on the description itself, but rather the more abstract nature of the components and the connections among them.

Considerable work has been done in the study and use of dependence relationships among variables and statements at the implementation level. For example, Ferrante, Ottenstein, and Warren [7] introduced the program dependence graph (PDG) for use in compiler optimization. Harrold and Soffa [10] have studied alias and interprocedural analysis of C and C++ programs.

Representation schemes for program/system dependencies that are similar to our tabular representation have been used in program optimization and for system requirements analysis. Aho, Sethi and Ullman [1] describe a program representation where bit vectors are used to compactly represent "gen" and "kill" sets for each statement. Grady [9] uses an N-square representation to examine system coupling when assigning functionalities to components during initial system decomposition. Pomakis and Atlee [21] use a tabular notation similar to the one found in SCR [11] to specify feature behaviors. While all of these methods use a representation similar to our tabular representation, the analyses applied to the representations are different and are for different purposes.

Chang and Richardson [4] introduce techniques for creating dynamic specification slices. This approach uses traditional slicing criteria, whereas our work involves exploring relationships at the architectural level, where additional criteria are defined.

Naumovich et al. [18] apply INCA and FLAVERS, two static concurrency analysis tools used for proving behavioral properties of concurrent programs, to an Ada translation of a description of the gas station problem that was written in the Wright ADL. Their approach is to create a concurrent program that can simulate the intended concurrent behavior of the system. Our work is aimed at developing general dependence analysis techniques that may contribute to the enhancement of the static analyses already provided by these tools.

References

[1] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers principles, techniques, and tools.* Addison-Wesley, Reading, MA, 1986.

[2] R. Allen and D. Garlan. A Formal Basis for Architectural Connection. *ACM Transactions on Software Engineering and Methodology*, 6(3):213–249, July 1997.

[3] D.F. Bacon, S.L. Graham, and O.J. Sharp. Compiler Transformations for High-Performance Computing. *ACM Computing Surveys*, 26(4):345–420, December 1994.

[4] J. Chang and D. J. Richardson. Static and Dynamic Specification Slicing. In *Proceedings of the Fourth Irvine Software Symposium*, Irvine, CA, April 1994.

[5] J.R. Cordy and K.A. Schneider. Architectural Design Recovery Using Source Transformations. In *CASE'95: Workshop on Software Architecture*, Toronto, Canada, July 1995.

[6] E. Duesterwald, R. Gupta, and M.L. Soffa. Demand-driven Computation of Interprocedural Data Flow. In *Symposium on Principles of Programming Languages (POPL'95)*, pages 37–48, San Francisco, January 1995.

[7] J. Ferrante, K.J. Ottenstein, and J.D. Warren. The Program Dependence Graph and Its Use in Optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, July 1987.

[8] D.W. Goodwin. Interprocedural Dataflow Analysis in an Executable Optimizer. In *Proceedings of the ACM SIGPLAN '97 Conference on Programming Language Design and Implementation*, pages 122–133, Las Vegas, Nevada, June 1997.

[9] J.O. Grady. *System Requirements Analysis.* McGraw-Hill, Inc., 1993.

[10] M.J. Harrold and M.L. Soffa. Efficient Computation of Interprocedural Definition-Use Chains. *ACM*

Transactions on Programming Languages and Systems, 16(2):175–204, March 1994.

[11] K. Heninger. Specifying Software Requirements for Complex Systems: New Techniques and Their Applications. *IEEE Transactions on Software Engineering*, 6(1):2–12, January 1980.

[12] P. Inverardi, A.L. Wolf, and D. Yankelevich. Checking Assumptions in Component Dynamics at the Architectural Level. In *Proceedings of the Second International Conference on Coordination Models and Languages*, number 1282 in Lecture Notes in Computer Science, pages 46–63. Springer-Verlag, September 1997.

[13] D.C. Luckham, J.J. Kenney, L.M. Augustin, J. Vera, D. Bryan, and W. Mann. Specification and Analysis of System Architecture Using Rapide. *IEEE Transactions on Software Engineering*, 21(4):336–355, April 1995.

[14] J. Magee, N. Dulay, S. Eisenbach, and J. Kramer. Distributed Software Architectures. In *Proceedings of the Fifth European Software Engineering Conference*, number 989 in Lecture Notes in Computer Science, pages 137–153. Springer-Verlag, September 1995.

[15] N.C. Mendonca and J. Kramer. Developing an Approach for the Recovery of Distributed Software Architectures. In *Proceedings of the Sixth International Workshop on Program Comprehension*, pages 28–36, June 1998.

[16] M. Moriconi, X. Qian, and R. A. Riemenschneider. Correct Architecture Refinement. *IEEE Transactions on Software Engineering*, 21(4):356–372, April 1995.

[17] G.C. Murhpy and D.N. Notkin. Lightweight Lexical Source Model Extraction. *TOSEM*, 5(3):262–292, July 1996.

[18] G. Naumovich, G.S. Avrunin, L.A. Clarke, and L.J. Osterweil. Applying Static Analysis to Software Architectures. In *Proceedings of the Sixth European Software Engineering Conference*, number 1301 in Lecture Notes in Computer Science, pages 77–93. Springer-Verlag, 1997.

[19] H. Pande, W. Landi, and B. Ryder. Interprocedural Def-Use Associations for C Systems with Single Level Pointers. *IEEE Transactions on Software Engineering*, 20(5):385–403, May 1994.

[20] A. Podgurski and L.A. Clarke. A Formal Model of Program Dependencies and its Implications for Software Testing, Debugging, and Maintenance. *IEEE Transactions on Software Engineering*, 16(9):965–979, September 1990.

[21] K.P. Pomakis and J.M. Atlee. Reachability Analysis of Feature Interactions: A Progress Report. In *Proceedings of the 1996 International Symposium on Software Testing and Analysis (ISSTA '96)*, pages 216–223. ACM SIGSOFT, January 1996.

[22] D.J. Richardson, T.O. O'Malley, C.T. Moore, and S.L. Aha. Developing and Integrating ProDAG in the Arcadia Environment. In *SIGSOFT '92: Proceedings of the Fifth Symposium on Software Development Environments*, pages 109–119. ACM SIGSOFT, December 1992.

[23] A.M. Sloane and J. Holdsworth. Beyond Traditional Program Slicing. In *Proceedings of the 1996 International Symposium on Software Testing and Analysis (ISSTA '96)*, pages 180–186. ACM SIGSOFT, January 1996.

[24] J.A. Stafford, D.J. Richardosn, and A.L. Wolf. Aladdin: A Tool for Architecture-Level Dependence Analysis of Software Systems. Technical Report CU-CS-858-98, Department of Computer Science, University of Colorado, Boulder, Colorado, April 1998.

[25] S. Vestal. *MetaH Programmer's Manual*. Honeywell, Inc., Minneapolis, MN, 1996.

[26] M. Weiser. Program Slicing. In *Proceedings of the 5th International Conference on Software Engineering*, pages 439–449. IEEE Computer Society, March 1981.

[27] G. Zelesnik. Unicon Reference Manual. Technical Report CMU-CS-97-TBD, Carnagie Mellon Univeristy, 1997.

[28] J. Zhao. Applying Slicing Techniques to Software Architectures. In *The Fourth IEEE International Conference on Engineering of Complex Computer Systems*, Monterey, August 1998.