

# Architecture-level Dependence Analysis for Software Systems

Judith A. Stafford<sup>†</sup>, Debra J. Richardson<sup>‡</sup>, and Alexander L. Wolf<sup>†</sup>

<sup>†</sup>Department of Computer Science  
University of Colorado  
Boulder, CO 80309 USA  
+1.303.492.5263  
{judys,alw}@cs.colorado.edu

<sup>‡</sup>Dept. of Information and Computer Science  
University of California  
Irvine, CA 92697 USA  
+1.714.824.7353  
djr@ics.uci.edu

## ABSTRACT

Software architecture description languages provide a means to formally describe software systems at a high level of abstraction. They capture the high-level structure and/or behavior of the system, thus providing a basis for course-grain static analyses. Dependence analysis has been used as a basis for program optimization, debugging, and testing. We are developing a dependence analysis technique, called *chaining*, for use with formal architectural descriptions, and implementing the technique in a tool called Aladdin.

## KEYWORDS

Dependence Analysis, Software Architecture, Architecture Description Languages

## 1 Introduction

Software architecture descriptions model systems at high levels of abstraction. They capture information about a system's components and how those components are interconnected. Some architectural descriptions 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.

Formal software architecture description languages (ADLs) allow one to reason about the correctness of software systems at a correspondingly high level of abstraction. Techniques have been developed for architecture analysis that can reveal such problems as potential deadlock and component mismatches [3, 15, 17, 20]. In general, there are many kinds of questions one might want to ask at an architectural level for 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.

The traditional view of dependence analysis is based on control and data flow relationships associated with functions and variables (e.g., [13]). This type of analysis was originally applied to compiler optimization [2] in order to determine safe code restructuring. The development of efficient and precise dependence analysis algorithms continues as an active area of research among compiler designers (e.g., [4, 9, 13]). The application of dependence analysis techniques to aid program understanding and impact analysis has been widely studied (e.g., [19, 21, 23]). The creation of both static and dynamic program slices has been a focus for many researchers (e.g., [28, 30]). Sloane and Holdsworth [25] suggest generalizing the concept of program slicing and suggest the potential for slicing non-imperative programs.

Our research takes a broader view of dependence relationships that is more 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 [16] and Wright [3] are considered. We are developing an architecture-level dependence analysis technique, called *chaining*, and implementing the technique in a tool called Aladdin. Aladdin is similar in concept to ProDAG [24], which is an implementation-level dependence analysis tool for Ada and C++ programs. Dependence analysis is performed by both ProDAG and Aladdin in a two-step process. First, an 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 relationships that could

This work was supported in part by the National Science Foundation under grant CCR-97-10078 and by the Air Force Material Command, Rome Laboratory, and the Defense Advanced Research Projects Agency under Contract Number F30602-94-C-0253. 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.

exist between a given pair of architectural elements. This set is queried in order to construct chains of dependent elements.

## 2 Dependence Relationships

Dependence relationships at the architectural level arise from the connections among components and the constraints on their interactions. These relationships may involve some form of control or data flow, but more generally involve *source structure* and *behavior*. Source structure (or structure, for short) has to do with static source specification dependencies, while behavior has to do with dynamic interaction dependencies. Some examples of architectural relationships are:

### Structural Relationships

- **Textual Inclusion**—The specification for a component may be created from numerous source modules that are textually combined.
- **Import/Export**—The specification for a component may describe the information exported and imported between source modules (e.g., with a module interconnection language).
- **Inheritance**—The specification for a component may be created through inheritance from other source modules.

### Behavioral Relationships

- **Temporal**—The behavior of one component precedes or follows the behavior of another component.
- **State-based**—The behavior cannot happen unless the system, or some part of the system, is in a specified state.
- **Causal**—The behavior of one component implies the behavior of another component.
- **Input/Output**—A component requires/provides information from/to another component.

The structural dependencies allow one to locate source specifications that contribute to the description of some state or interaction. The behavioral dependencies allow one to relate states or interactions to other states or interactions. Both structural and behavioral dependencies are important to capture and understand when analyzing an architecture.

The particular kinds of dependencies that can be established among components are heavily influenced by the primitive features of the ADL. For instance, in CHAM,

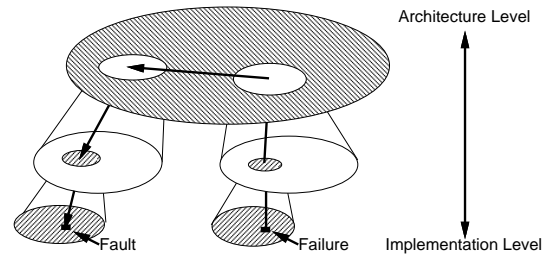


Figure 1: Architecture-Assisted Fault Localization.

only behavioral relationships are modeled, and those relationships are based on synchronous input and output ports through which data flow.<sup>1</sup> As mentioned above, Rapide specifications can involve both structural and behavioral relationships and typically involve event-based interactions.

## 3 Dependence-Related Questions

There are many kinds of questions one might want to ask at an architectural level. Below, we list some questions that are answerable through analysis of a formal architecture description.

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. If a change is made to this component, what other components might be affected?
4. If a change is made to this component, what is the minimal set of test cases that must be rerun?
5. If the source specification for a component is checked out into a workspace for modification, which other source specifications should also be checked out into that workspace?
6. If a failure of the system occurs, what is the minimal set of components of the system 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. The first three questions are architecture-based questions and are answerable through analysis of the formal

<sup>1</sup>Connectors can be introduced to model other kinds of behavioral relationships, but these are still based on, or built from, the primitive concept of the synchronous port.

description of the system alone. Questions 4, 5, and 6 are questions about the implementation that will be answered with the aid of architecture-level analysis. These will require the ability to map certain elements of the *current* implementation, such as a line of code or a test case, to the *current* version of the architecture, as illustrated in Figure 1. Architectures erode due to changes in a system’s implementation that are not reflected in the documentation and higher-level specifications. Two approaches to providing precise mappings are code generation [27, 29, 31] and architecture recovery [6, 19].

#### 4 Creating and Using Chains

As mentioned earlier, chains represent dependence relationships in an architectural specification. The individual chain links within a chain associate elements of an architecture that are directly related, while a chain of dependencies includes associations among architectural elements that are indirectly related. Chaining is the process of applying dependence algorithms to architectural descriptions in order to create these sets of related components and/or elements.

The chaining method was inspired by a study of an architectural description of the ADAGE avionics architecture given in Rapide [18]. The ADAGE example is large and complex, containing three levels of architecture and built from 30 components communicating through over 300 ports. The study involved the intentional introduction of a fault into the specification and the development of a (manual) process for uncovering the fault. The resulting process was to trace back through the system dependencies identified from the Rapide specification. Using the process, fewer than 25 percent of the system’s components required examination.

Aladdin implements the portion of the process that derives chains from an architectural specification. It also provides a set of queries over the chains that aid in debugging the architecture. We have used Aladdin in the analysis of a Rapide specification for the gas station example [26].<sup>2</sup>

We define three types of chains: affected-by, affects, and related. These are illustrated in Figure 2 and represent the directionality of the relationship. An affected-by relationship indicates that the element of interest could be affected by the elements linked together in the chain. Conversely, an affects chain contains the set of elements on which the element of interest could potentially have

<sup>2</sup>It could be argued that the gas station example is not representative of software architecture specifications, although it is widely used in the architecture literature. It has the advantage of being well known and compact, and does in fact exhibit features that would appear in a “real” architecture specification. In general, there appears to be a dearth of good architecture specification examples, both large and small.

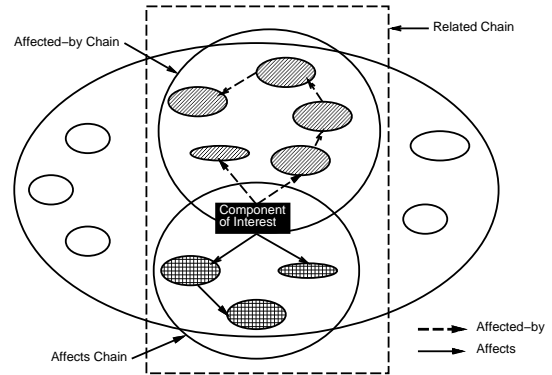


Figure 2: Chains.

an effect. For a particular element, related chains are the union of affected-by and affects chains. Different types of chains are created to answer different kinds of questions. Question 1 in Section 3 could be answered by constructing an affected-by chain based on imports and includes relationships. As other examples, one could answer questions 2 and 6 by building affected-by chains; question 4 could be answered using affects chains, and related chains could be used to answer question 5. Depending on the type of change being made, question 3 could be answered by building any one of the chains.

Aladdin uses a table to record the dependence relationships of an architecture. The table has  $m$  rows and  $n$  columns;  $m$  is the number of communication ports in the architecture plus any inputs from the environment, and  $n$  is the number of communication ports in the architecture plus any emissions to the environment, such as warning messages. The relationships associated with connections and communication ports in an architecture induce dependence relationships. The columns in the table represent the dependent in the relationship and the rows represent the source (or object) of the dependence. For instance, if  $a$  is dependent on  $b$ , then the cell at column  $a$  and row  $b$  details that relationship. The cell may, for example, reflect the existence of a direct connection, such as a remote function call, it may indicate sharing of data, it may indicate interaction by means of event notification and subscription, or it may contain a combination of relationships between the source and destination elements. In general, for a given architecture description language, it is necessary to understand the various ways in which two architectural elements can be related so that the dependence table can be constructed.

Once the system’s relationships are recorded in the table, a transitive closure over the dependencies for a particular element can be performed in order to construct

a chain, as depicted in Figure 2. For instance, in the gas station example, chaining over the table allows the discovery of unused ports, specification coding errors, and investigation of how the replacement of a component would affect the rest of the system.

## 5 Related Research

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

Structural dependencies are investigated at the source level for use in tools such as *makedepend* that examines code to automatically derive the file dependencies (e.g., `#include` in the C environment) used in Make files. Murphy and Notkin [19] make use of source structure information as well as call graphs to extract a source model of a system. Our approach applies this concept to architecture description languages and combines the information with behavioral dependencies.

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 [11] 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 in the program and logical operations are performed on these bit sets to determine statement dependencies. Grady [10] uses an N-square representation to examine system coupling when assigning functionalities to components during initial system decomposition. Pomakis and Atlee [22] use a tabular notation similar to the one found in SCR [12] to specify feature behaviors. This is used in conjunction with a graphical representation to study possible feature interactions. 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.

Sloane and Holdsworth [25] suggest new applications for program slicing, 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.

Chang and Richardson [5] introduced techniques for cre-

ating dynamic specification slices. This approach uses traditional slicing criteria, whereas our work involves exploring relationships at the architectural level, where additional criteria are defined.

Zhao [32] is investigating the use of a system dependence net to slice architectural descriptions written in the Wright ADL. The work described in this initial paper is similar in nature to our work on chaining but is preliminary and the details of his method for determining related components are unstated.

Naumovich et al. [20] 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.

## 6 Future Work

Future plans for Aladdin include improving the precision of the analysis by improving the intra-component behavior summarizations. We intend to incorporate more features of the Rapide language, to extend our definition of chaining to other languages (including CHAM [14], Acme [8], and Wright [3]), and to test our approach against more complex architectures.

Historically, testing has concentrated on the implementation of the system, which has meant that it is considered fairly late in the development process. Eventually, we intend to incorporate chaining into a complete life cycle software analysis and testing environment, such as the TAOS environment [23]. TAOS includes dependence analysis and testing at the implementation level, with some support for specification-based test case generation and result checking. Integrating architecture analysis techniques such as chaining would round out the life cycle support for analysis and testing.

## REFERENCES

- [1] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers principles, techniques, and tools*. Addison-Wesley, Reading, MA, 1986.
- [2] F.E. Allen. Control Flow Analysis. *SIGPLAN Notices*, pages 1–19, July 1970.
- [3] R. Allen and D. Garlan. A Formal Basis for Architectural Connection. *ACM Transactions on Software Engineering and Methodology*, 6(3):213–249, July 1997.
- [4] 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.

- [5] J. Chang and D. J. Richardson. Static and Dynamic Specification Slicing. In *Proceedings of the Fourth Irvine Software Symposium*, Irvine, CA, April 1994.
- [6] J.R. Cordy and K.A. Schneider. Architectural Design Recovery Using Source Transformations. In *CASE'95: Workshop on Software Architecture*, Toronto, Canada, July 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. Garlan, R. Monroe, and D. Wile. ACME: An Architecture Description Interchange Language. In *Proceedings of CASCON '97*. IBM Center for Advanced Studies, November 1997. To appear.
- [9] 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.
- [10] J.O. Grady. *System Requirements Analysis*. McGraw-Hill, Inc., 1993.
- [11] 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.
- [12] K. Heninger. Specifying Software Requirements for Complex Systems: New Techniques and Their Applications. *IEEE Transactions on Software Engineering*, 6(1):2–12, January 1980.
- [13] S. Horwitz, T. Reps, and D. Binkley. Interprocedural Slicing Using Dependence Graphs. *ACM Transactions on Programming Languages and Systems*, 22(1):26–60, January 1990.
- [14] P. Inverardi and A.L. Wolf. Formal Specification and Analysis of Software Architectures using the Chemical Abstract Machine Model. *IEEE Transactions on Software Engineering*, 21(4):373–386, April 1995.
- [15] 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.
- [16] 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.
- [17] 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.
- [18] W. Mann, F. C. Belz, and P. Corneil. A Rapide-1.0 Definition of the ADAGE Avionics System. Technical Report CSL-TR-93-585, Stanford University, 1993.
- [19] G.C. Murhpy and D.N. Notkin. Lightweight Lexical Source Model Extraction. *TOSEM*, 5(3):262–292, July 1996.
- [20] 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*. Springer-Verlag, 1997. To appear.
- [21] 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.
- [22] 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.
- [23] D.J. Richardson. TAOS: Testing with Analysis and Oracle Support. In *Proceedings of the 1994 International Symposium on Software Testing and Analysis (ISSTA '94)*, pages 138–153. ACM SIGSOFT, August 1994.
- [24] 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.
- [25] 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.
- [26] J.A. Stafford, D.J. Richardson, and A.L. Wolf. Chaining: A Software Architecture Dependence Analysis Technique. Technical Report CU-CS-845-97, University of Colorado, September 1997.
- [27] RAPIDE Design Team. Draft: Rapide 1.0 Architecture Language Reference Manual. July 1997.
- [28] P. Tonella, G. Antoniol, R. Fiutem, and E. Merlo. Flow Insensitive C++ Pointers and Polymorphism Analysis and its Application to Slicing. In *19th International Conference on Software Engineering*, pages 433–443, Boston, April 1997.
- [29] S. Vestal. *MetaH Programmer's Manual*. Honeywell, Inc., Minneapolis, MN, 1996.
- [30] M. Weiser. Program Slicing. In *Proceedings of the 5th International Conference on Software Engineering*, pages 439–449. IEEE Computer Society, March 1981.
- [31] G. Zelesnik. Unicon Reference Manual. Technical Report CMU-CS-97-TBD, Carnegie Mellon University, 1997.
- [32] J. Zhao. Using Dependence Analysis to Support Software Architecture Understanding. *New Technologies on Computer Software*, pages 135–142, September 1997.