

# Top-Down Analysis for Bottom-Up Development

Judith A. Stafford<sup>†</sup> and John D. McGregor<sup>‡</sup>

<sup>†</sup>Department of Computer Science  
Tufts University  
Medford, MA 02155 USA  
jas@cs.tufts.edu

<sup>‡</sup>Department of Computer Science  
Clemson University  
Clemson, SC 29634 USA  
johnmc@cs.clemson.edu

## 1 Introduction

Component-based software engineering (CBSE) is the construction of software systems from software components. Software components are independently deployable units of executable code that can be assembled into a system based on their externally visible properties, which are made available to system developers via the components interface specifications. Software components provide a means for large-scale reuse of intellectual property as well as coding, testing, and analysis effort. Benefits expected from developing an industry in CBSE include: reduced system development and maintenance costs, reduced time-to-market, and the availability of a marketplace of components that are bought and sold in massive quantities much like any other product. Component-based systems are constructed from a blend of locally developed and commercially available pieces enabling incremental development of complex systems. However, the full potential for software component reuse and a component marketplace can only be realized if tools and techniques are available to support the development and maintenance of these systems [2]. In this paper we address the specific problem of creating analytic assets, i.e. assets that summarize information about components that is required for compositional analysis. As a first step toward being able to create the analytic assets needed for component-based systems, we describe our in identifying and packaging analytic assets for program procedures for use in compositional program analysis.

Analytic asset based analysis is hierarchical and can be applied at the statement level, the procedure level, the component level or the assembly level. What is important is that the items at the level of analysis will be treated as black boxes after their initial analysis. For example, the control dependence graph for an assembly of components can be constructed without having access to the internals of the individual components. Each component is packaged with a set of analytic assets that, among other things, includes the information necessary to compose its control dependence subgraph with that of the other components in the assembly. Then when the black boxes are aggregated the control dependence graph is composed from the information attached to each component. Our goal is to be able to use these assets to support a variety of compositional reasoning tasks.

## 2 Approach

Composition of software from parts has long been used as a means to increase the tractability and efficiency of building and maintaining software systems. Composition at its simplest is the creation of programs out of procedures and extends up to the composition of software systems from components that are, themselves, software systems. Compositional approaches to software development allow one to consider the system as a hierarchy of software elements from program statements to software components. Current practice in software analysis requires repeated contextual analysis of the parts and access to source code.

Compositional approaches to software analysis must be developed to provide scalable support for analysis, testing, and evolution of increasingly large and complex software systems and to support analysis of component-based software for which access to source code cannot be assumed. In this section we describe our work in progress towards developing a hierarchy of dependence analysis techniques to support impact analysis of software systems that are comprised of pre-existing parts. Our approach is based on developing dependence analysis algorithms that produce *analytic assets*, which can be composed through the application of compositional analysis techniques that are applied at the next higher level of the composition hierarchy. Using this approach it is possible to avoid reevaluation of parts and access to source code by the assembler is not required.

Our intuition for this work arose from the recognition that formal, mathematically-based languages called architecture description languages (ADLs), which were first introduced in the early 1990s, provide support for analysis similar to support for static analysis provided by programming languages [25, 27]. The importance of early life cycle application of quality attribute analysis is evidenced by the architecture evaluations that are currently applied in practice in the informal (less than mathematically rigorous) setting of the Architecture Tradeoff Analysis Method, architecture reviews and other forms of architecture evaluation [5]. ADLs support unambiguous description of software in terms of its major computational elements, connectors that support interaction among them, and their configuration. Researchers have developed techniques similar to those used to analyze computer programs to analyze the code used to describe software architectures. Benefits include early identification of design anomalies, impact analysis to support informed tradeoff analysis [27], performance analysis to identify potential bottlenecks and missed deadlines [14, 25] and analysis of liveness and safety properties of systems [3, 13, 18]. Our prior work in this area [27] demonstrated that precision in the results produced by the use of these techniques can be improved with specification of the internal properties of software elements (property requirements imposed by the architect on developers), which may eliminate the need to consider some potential input-to-output pathways in the software elements and thereby identification of fewer potential communication and stimulation pathways among distant elements of the architecture.

Our approach to hierarchical dependence analysis is based on the observation that interdependencies among software elements exist due to communication among elements and stimulation of one element by another; and that the process of identifying dependencies differs depending on the means of communication and stimulation. At the lowest level, which we take as a unit comprised of a sequence of program statements where communication and stimulation occur in the form of transfer of control from statement to statement and variable assignment and use. When these programs are decomposed into a main and a set of procedures, then procedure call is an added form of stimulation and parameter passing is an added form of communication. At the component assembly level communication and stimulation generally take the form of remote procedure call, message passing, and event stimulation.

These various sources of dependencies provide different opportunities for improving precision of dependence analysis, which at its most conservative (i.e. assuring no dependencies are missed) assumes there is possibility for all elements to affect each other. Our work explores the causes of dependencies among elements at various levels of the hierarchy and the development of algorithms to exploit information about these dependencies to improve the precision (i.e. include fewer false dependencies) of analysis at the next

higher level of composition.

Past work in the area of inter-procedural dependence analysis is limited in its support for creating analytic assets due to reliance on the availability of control flow graphs at the time of program analysis [9, 10, 12, 16]. In our current work, we are bridging the gap between program analysis and architecture analysis through the identification of a hierarchy of input-to-output pathway analysis techniques that support precise impact analysis at any level of the composition hierarchy. In subsections 2.1 and 2.2 we demonstrate our approach through the description of work in architecture dependence analysis, which depends on the availability of analytic assets in the architectural elements for increased precision, and the development of compositional control dependence program analysis (CCDA). CCDA allows us to identify analytic assets, which are associated with procedures and subsequently used multiple times during program analysis. Our future work involves similarly identifying the makeup of analytic assets for software elements that employ more complex forms of communication and stimulation, to support analysis of components and assemblies of components, which can be analyzed similarly to software architectures.

## 2.1 Architecture Dependence Analysis

Assemblies of software components can be described similarly to software architectures described using architecture description languages (ADLs) in terms of the components and their interconnections. We have developed a prototype tool, Aladdin, for performing architecture dependence analysis that allows us to obtain information that can be used to study the dependence relationships among ports (i.e. component sources and sinks) of an architecture. Aladdin takes architectural descriptions as inputs and, in response to queries by a user, produces *pathways* of direct dependencies rooted at a specified port in the architecture. Aladdin uses a three-phase approach to identify architecture dependence pathways. In Phase I, Aladdin computes and records all potential dependence pathways between component sinks and sources. In Phase II connections among ports of different components are identified. And finally, in Phase III, the intracomponent pathways are combined with the architectural connections to identify architecture pathways. A graphical depiction of the results of these three phases is shown in Figure 1. Aladdin allows a user to query the architecture dependence structure to identify inter-component dependencies.

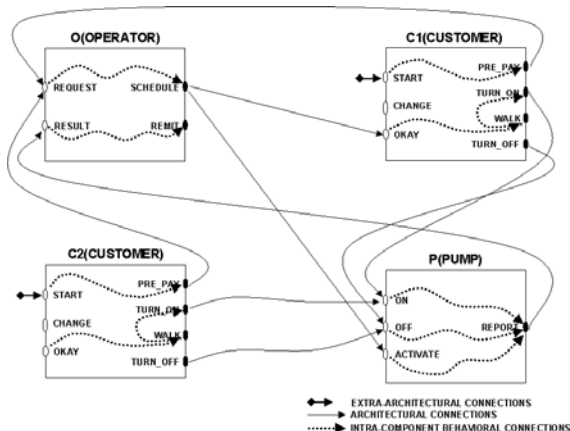


Figure 1: Gas Station Architecture.

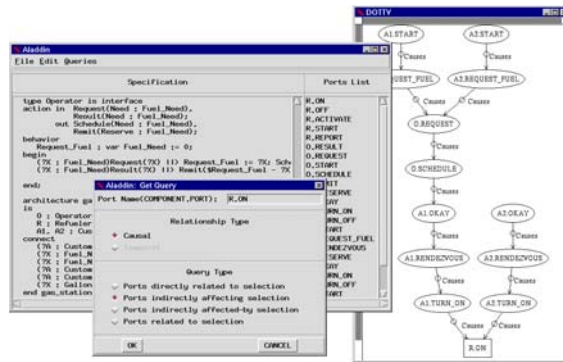


Figure 2: Screen Shot of Aladdin in Use.

Aladdin’s interface and its use to analyze the gas station architecture depicted in Figure 1 is shown in Figure 2. Architectural dependence pathways rooted at the Pump’s `On` port are shown to the right in the figure.

Aladdin has been carefully designed to separate language-specific processing tasks from language-independent analysis tasks, and so can be tailored for use with multiple languages. For example, besides its use with Rapide [17] as exhibited in Figure 2, Aladdin also works with Acme [7]. The abstract syntax tree (AST) representation of an architectural specification is translated into a *dependence table* that is used to compute indirect dependencies. The portion of Aladdin that builds the ASTs for the Rapide and Acme variants were obtained from the language definition teams, the Rapide Design Team at Stanford University and the Acme research group at Carnegie Mellon University. Aladdin can be made to work with any language for which an AST builder is available. We are currently searching for a suitably expressive composition language.

## 2.2 Creating Analytic Assets

The goal of our research is to identify the minimal information that must be known about a software element, in the case of the work reported in this section a procedure, in order to identify program dependencies without access to the code of procedures so that we could understand the fundamental requirements of producing these kinds of analytic assets for larger grained software elements. For sequential programs conservative sets of dependencies among program statements can be identified statically using the control flow graph as a base. During the 1970s and 1980s data flow algorithms, and dependence analysis algorithms were developed to identify the potential interdependencies among program statement in single procedure programs [1, 22, 28, 29]. In the 1990s attention turned toward multi-procedure programs and other types of control structures [6, 8, 11, 12, 16, 20, 21, 4, 24, 23]. The goal of each of these works, and the many other unmentioned works in this area, was to create a more efficient, more precise, more conservative, or more powerful algorithm to statically identify dependencies among program statements. Our research is stimulated and informed by this large body of work.

In the remainder of this section we describe a model for identifying control dependence analytic assets for multi-procedure sequential programs. The guiding principles followed during the design of this model were that it be a rigorously defined, language independent, and compositional. Our approach differs from other approaches to CFG-based inter-procedural analysis techniques in that we recognize that the forward

dominance relationship [15] among program statements, which is used to define control dependencies [22] does not apply to statement-to-statement relationships in programs comprised of multiple procedures. This observation allows us to produce a simpler algorithm than earlier inter-procedural control dependence algorithms and that produces reusable analysis results. We defined a forward dominance forest for representation of forward dominance relationships in procedures of multi-procedure programs, which allows us to segregate procedure control dependence information without the need to retain the control flow graph, and use this as the basis for our control dependence algorithm. The compositional control dependence analysis (CCDA) model supports the identification of intra-procedural dependencies in isolation; at program composition time the inter-procedural implications of the intra-procedural dependencies are computed.

Inter-procedural communication among statements in a program creates several challenges for the dependence analyst. These challenges fall into two areas: consideration of the effects of calling context and failure to return from a procedure call. These result in very different types of dependence-related problems:

- Ignorance of calling context results in identification of unnecessarily imprecise sets of dependences.
- Ignorance of the potential for non-return from procedure calls introduces the possibility of creating non-conservative sets of dependences.

Our definition of inter-procedural control dependence reflects the recognition of these two concerns. Informally, if  $u$  and  $v$  are statements in a multi-procedure program  $\mathcal{P}$ ,  $v \in P$  where  $P \in \mathcal{P}$ , is control dependent on  $u$  if the number of times  $v$  is executed with respect to the same invocation of  $P$  or the number of times  $P$  is invoked can be affected by a decision made at  $u$ .

The first condition for control dependence is the same condition that applies to uni-procedure programs. However, when this condition is considered for multi-procedure programs, then the control dependence of  $v$  may rest on a statement in another procedure; for example, if  $v$  follows a procedure call from inside a while loop, then its execution depends upon the resumption of  $P$  after the procedure call. The second condition is equivalent to considering a procedure call to be an entry to a region of code in which all statements are dependent on the condition that determines whether entry to the region is executed. We call these two types of dependence *resumption* and *invocation* dependence. Additionally, we define *dependence inheritance* and *potential control dependence*, which are required to support composition.

These control dependence concepts in combination with an enhanced model of uni-procedure control dependence, which we refer to as procedure control dependence, form the foundation for our model. The procedure control dependence algorithm produces the following control dependence analytic assets: procedure control dependence graph, a set of interrupted flow arcs that represent the relationship between a call statement and the statement that immediately follows the call, and a binary indication of whether the procedure's forward dominance relationship can be captured as a forward dominance tree. Our model, depicted in Figure 3, uses this information in combination with the call graph for the program, applies the notions of invocation, resumption, potential, and inherited control dependence and produces a conservative set of interdependencies for the program.

A full discussion and description of the model is beyond the scope of this paper and the reader is directed to [26] for details. The important things to note are: first, each procedure is analyzed only one time and the information produced is reused without modification in any context; second, the procedure control dependence graph does not reveal the source code for the procedure; third, the algorithm and information required to identify program dependencies are different from those required to identify the procedures' analytic assets. These are the characteristics for compositional analysis at all levels in the composition hierarchy.

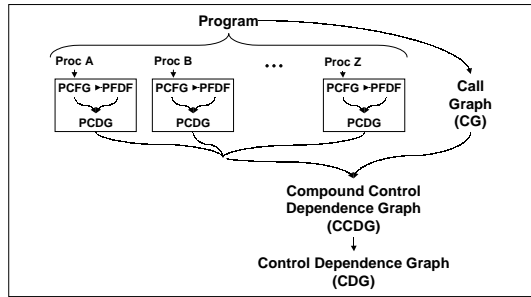


Figure 3: Compositional Model of Program Dependencies.

### 3 Conclusion and Future Work

We have presented an overview of our work in compositional dependence analysis. Our hierarchical approach to static dependence analysis provides a scalable means for identifying dependencies among components of software assemblies. Identification of dependencies among components can be used to support a variety of software maintenance activities. We are currently working on using intra-procedural analysis assets to support intra-component dependence analysis and identification of component dependence analytic assets. We are also investigating architectural-based regression testing (ABRT) techniques for component-based systems. The techniques are based on dependence analyses that are conducted based on information in the architecture description for the system under test. Next steps include identification and validation of structural and functional testing criteria for architectural descriptions of assemblies of components. Structural criteria will be based on coverage of architectural elements and, using the same operational profile techniques that we have used in reliability measurement [19] in combination with compositional control dependence analysis, we will use a probabilistic approach to identify pathways through an assembly and focus regression testing effort on those pathways that are most likely to exhibit failure.

### References

- [1] F.E. Allen and J. Cocke. A Program Data Flow Analysis Procedure. *Communications of the ACM*, 19(3), March 1976.
- [2] P. Allen. Reuse in the component marketplace. *The Cutter Edge*, 2001.
- [3] R. Allen, D. Garlan, and J. Ivers. Formal Modeling and Analysis of the HLA Component Integration Standard. In *Proceedings of the ACM SIGSOFT Sixth International Symposium on the Foundations of Software Engineering*, pages 70–79. ACM SIGSOFT, November 1998.
- [4] G. Bilardi and K. Pingali. A Framework for Generalized Control Dependence. In *ACM SIGPLAN '96 Conference on Programming Language Design and Implementation (PLDI'96)*, pages 291–300, Philadelphia, Pennsylvania, May 1996. Uses loop dominator forest to efficiently compute CD.
- [5] P. Clements, R. Kazman, and M. Klein, editors. *Evaluating Software Architectures: Methods and Case Studies*. Addison-Wesley, Reading, Massachusetts, 2001.

- [6] E. Duesterwald, R. Gupta, and M.L. Soffa. A Practical Framework for Demand-Driven Interprocedural Data Flow Analysis. *ACM Transactions on Programming Languages and Systems*, 19(6):995–1030, November 1997.
- [7] D. Garlan, R. Monroe, and D. Wile. ACME: An Architecture Description Interchange Language. In *Proceedings of CASCON '97*, pages 169–183. IBM Center for Advanced Studies, November 1997.
- [8] D. Grunwald and H. Srinivasan. Data Flow Equations for Explicitly Parallel Programs. In *ACM Conference on Principles of Parallel Processing*, San Diego, May 1993.
- [9] M.J. Harrold and N. Ci. Reuse-Driven Interprocedural Slicing. In *Proceedings of the 1998 International Conference on Software Engineering*, pages 74–83. ACM SIGSOFT, May 1998.
- [10] M.J. Harrold, G. Rothermel, and S. Sinha. Computation of Interprocedural Control Dependence. In *Proceedings of the 1998 International Symposium on Software Testing and Analysis (ISSTA '98)*, pages 11–20. ACM SIGSOFT, March 1998.
- [11] J. Hatcliff, J. Corbett, M. Dwyer, S. Sokolowski, and H. Zheng. A Formal Study of Slicing for Multi-threaded Programs with JVM Concurrency Primitives. In *Static Analysis Symposium (SAS'99)*, Venice, Italy, September 1999.
- [12] S. Horwitz, T. Reps, and D. Binkley. Interprocedural Slicing Using Dependence Graphs. *ACM Transactions on Programming Languages and Systems*, 12(1):26–60, January 1990.
- [13] P. Inverardi, A.L. Wolf, and D. Yankelevich. Static Checking of System Behaviors Using Derived Component Assumptions. *ACM Transactions on Software Engineering and Methodology*, 9(3):239–272, July 2000.
- [14] M.H. Klein, T. Ralya, B. Pollak, R. Obenza, and M.G. Harbour. *A Practitioner's Handbook for Real-Time Analysis*. Kluwer Academic Publishers, 1993.
- [15] T. Lengauer and R.E. Tarjan. A fast algorithm for finding dominators in a flowgraph. *ACM Transactions on Programming Languages and Systems*, 1(1):121–141, July 1979.
- [16] J.P. Loyall and S.A. Mathisen. Using Dependence Analysis to Support the Software Maintenance Process. In *Proceedings of the 1993 International Conference on Software Maintenance*, pages 282–291. IEEE Computer Society, September 1993.
- [17] D.C. Luckham and J. Vera. An Event-based Architecture Definition Language. *IEEE Transactions on Software Engineering*, 21(9):717–734, September 1995.
- [18] J. Magee, J. Kramer, and D. Giannakopoulou. Analysing the Behaviour of Distributed Software Architectures: A Case Study. In *Fifth IEEE Workshop on Future Trends of Distributed Computing Systems*, pages 240–247, October 1997.
- [19] J.D. McGregor, J.A. Stafford, and I. Cho. Measuring and Reporting Component Reliability . In *Proceedings of the 1st ACIS International Conference on Software Engineering Research and Applications*. LNCS, 2003.
- [20] G. Naumovich, G.S. Avrunin, and L.A. Clarke. Data Flow Analysis for Checking Properties of Concurrent Java Programs. In *Proceedings of the 21st International Conference on Software Engineering*, pages 399–410, Los Angeles, May 1999.
- [21] 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.
- [22] A. Podgurski and L.A. Clarke. A Formal Model of Program Dependences and its Implications for Software Testing, Debugging, and Maintenance. *IEEE Transactions on Software Engineering*, 16(9):965–979, September 1990.
- [23] S. Sinha and M.J. Harrold. Analysis and Testing of Programs with Exception Handling Constructs . *IEEE Transactions on Software Engineering*, 26(9):849–871, September 2000.
- [24] S. Sinha, M.J. Harrold, and G. Rothermel. System-Dependence-Graph-Based Slicing of Programs with Arbitrary Interprocedural Control Flow. In *Proceedings of the 1999 International Conference on Software Engineering*, pages 432–441. Association for Computing Machinery, May 1999.

- [25] B. Spitznagel and D. Garlan. Architecture-Based Performance Analysis. In *Proceedings of the 1998 Conference on Software Engineering and Knowledge Engineering*, pages 146–151, San Francisco, California, June 1998.
- [26] J.A. Stafford. A Formal, Language-Independent, and Compositional Approach to Interprocedural Control Dependence Analysis. Technical Report CU-CS-907-00, Department of Computer Science, University of Colorado, Boulder, Colorado, August 2000.
- [27] J.A. Stafford and A.L. Wolf. Architecture-Level Dependence Analysis for Software Systems. *International Journal of Software Engineering and Knowledge Engineering*, 11(4):431–453, August 2001.
- [28] F. Tip. A Survey of Program Slicing Techniques. *Journal of Programming Languages*, 3:121–189, 1995.
- [29] M. Weiser. Program Slicing. In *Proceedings of the 5th International Conference on Software Engineering*, pages 439–449. IEEE Computer Society, March 1981.