Annotating Components to Support Component-Based Static Analyses of Software Systems

Judith A. Stafford and Alexander L. Wolf

University of Colorado
Boulder, CO 80309 USA
{judys,alw}@cs.colorado.edu

University of Colorado
Department of Computer Science
Technical Report CU-CS-896-99 December 1999

ABSTRACT

COTS components are typically viewed as black-boxes; input and output information is supplied in their interfaces. In this paper we argue that interfaces provide insufficient information for many analysis purposes and suggest analysis-related annotations be supplied with components. We center our discussion on static dependence analysis. We use an extensible language to annotate components and perform dependence analysis over these descriptions. We propose that component annotations be certified, thereby providing a base for certifiable analysis.

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. 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.

1 Introduction

Component-based development is intended to increase reliability and evolvability of systems while at the same time decrease the cost of system development. These benefits are expected because a component can be developed and tested in isolation, and then, based on its interface and stated functionality, used in a variety of systems. Kozaczynski [6] presents three component views that are relevant to differing concerns during different phases of system development: a design view, an implementation view, and a deployment view. We propose adding a fourth view to this list: an analysis view to support static analyses. Prior work has shown that static analysis of system behavior based solely on component interfaces is inadequate to diagnose incorrect component interactions [1, 3]. The proposed analysis view will provide the necessary information for reasoning about component interactions beyond the compatibility of their interfaces.

Knowledge of certain behavior- and resource-related attributes is used as a basis for choosing among similar software components. This is not unlike the information provided through parts annotations, such as voltage requirements and model number, used in the building and repair of hardware systems, automobiles, electrical systems, and plumbing systems. Consider the purchase of a simple light bulb. The interface of most light bulbs is identical; however the choice of which to purchase involves tradeoffs related to the voltage, wattage, and price of the light bulb. All of these things are listed on the light bulb as attributes of the bulb. They are guaranteed to be correct, and thus a person feels secure in making a decision about which bulb to purchase to meet their specific needs.

Various types of system analyses require specific types of knowledge about software components. It is useful to associate such properties as performance ratings, resource requirements, safety levels, and security levels with components. The specification of component quality attributes is suggested by Han [5]. He identifies the need to understand how to characterize the quality attributes for a component as well as the need to understand how to use such attributes as a part of overall system analysis. The MetaH [9] architecture description language (ADL) allows a system architect to assign specific types of attributes to architectural components, and then to perform various types of analyses as early as the design phase of software development. The set of annotation types supported in MetaH is limited to those that support analyses specifically tailored for developing real-time, fault tolerant systems with primary emphasis on avionics applications. COTS component annotations must provide support for a more broad spectrum of analyses.

We propose the use of an extensible language that supports the definition of annotation types. We use Acme [4] developed at Carnegie Mellon University for this purpose. Acme supports the description of component and connector interfaces, their internal properties, and their interconnections. Acme is an interchange language for architectural descriptions written in various ADLs. Property types are defined as needed for analysis or interchange purposes to capture the semantics of different ADLs; it is this feature of the language that makes Acme especially useful for our purposes. We use Acme to describe COTS component interfaces and define analysis-related annotations as Acme properties. Annotation types are suggested by developers of analysis techniques and included as component properties at the discretion of component developers.

2 Pathway Annotations

There are many types of annotations that would be useful to certify and supply with components in support of analyses such as performance tradeoff analysis and secure information flow analysis [8]. As an example, suppose the set of annotations associated with components includes

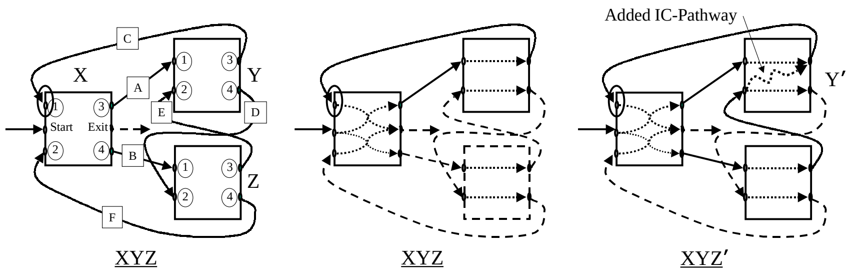


Figure 1: Intra-component pathways support identification of dependence sets.

some performance rating, say average time to complete a calculation, and that another annotation declares the amount of disk space required to install the component; these annotations can be used in performance tradeoff analysis. If an application is typically to be used in an environment where disk space is abundant but processing speeds are slow, one could choose to use the component that uses more disk space but performs its function more efficiently.

We are applying software dependence analysis techniques to component-based systems. Dependence analysis is the study of how one element of a system can affect or be affected by other elements of the system. Component interfaces alone do not provide sufficient information for identifying useful sets of potentially interacting components. We annotate components with information required for performing component-level dependence analysis. This information is in the form of pathways connecting component input ports to output ports; pathways capture the potential for an input to affect an output in some way. The behavioral relationships among the input and output ports of a component define the interaction behavior of that component. It is important to note that the interaction behavior is not intended to capture the component's functional behavior. For example, the description of how a server interacts with its clients is independent of the computation carried out by the server on behalf of its clients. Many software maintenance activities require knowledge about the potential for interaction among elements of a system; examples of such applications are impact analysis, fault localization, system re-engineering, and regression test set selection. Software dependence analysis is used as a source of information in support of these and similar activities.

Most applications of dependence analysis techniques require that conservative sets of dependencies be identified; that is, at least all potential dependencies must be included in the dependency set. If one is to be conservative when performing component-level dependence analysis based on component interfaces alone, one must assume that each and every input has the potential to impact every output. This assumption can result in computation of very imprecise dependence sets. The simple example shown in Figure 1 illustrates the improvement in precision that is gained when input-to-output pathways are included in the information used to determine possible dependencies. Figure 1 contains three diagrams. The boxes in each diagram represent components in a system; two systems are represented: system `S` composed of components `X`, `Y`, and `Z` and system `S′` in which we have replaced component `Y` with a its variant, `Y′`. The ports into and out of each component are named as shown in the circles inside each component in the diagram to the left. The arrows in the figure represent the ability for a box, or some port into or out of that box, to communicate with another box in the diagram; we call them connectors and refer to them by the names given in the squares over each arc in the diagram to the left. The solid arcs in this figure denote arcs that must be traversed in order to identify a conservative set of dependencies representing the parts of the system that have potential to affect port 1 of component `X`.

In the view of system `S` shown to the left in Figure 1, the intra-component pathways are unknown. The lack of information about the internal port-to-port interaction of the component forces the analysis to include ports of all components of the system in the dependency set. If, however, one knows for certain that particular input-output pairs are independent, as illustrated in the diagram in the center, the precision of the analysis is greatly improved.

In support of component-level dependence analysis, we use intra-component dependence analysis to identify potential interactions among component inputs and outputs and call the resulting dependencies *intra-component pathways* or ic-pathways. These pathways can be supplied as annotations with COTS components in support of system-wide dependence analysis. We are developing a formal model of dependencies for multi-procedure programs that will support certification of ic-pathway identification by independent certification houses. Such certification is required for use of COTS components in application domains where issues such as security and safety are of great importance. In such areas, potential for information flow among component inputs and outputs must be well understood and documented.

As mentioned above, the behavioral relationships among ports of a component define its interaction behavior. It is important to understand this when choosing among different components that provide similar functionality. We use a simple example architecture to convince the reader that minor changes in the internal dependence structure of a component have potential to result in major changes to the system-level dependence structure. We replace one component with another having an identical interface and a single additional pathway. Recall that the solid lines in the box and arrow diagrams of Figure 1 represent the connectors and components of systems `S` and `S'` that will be included in a dependence set representing the parts of the system that are identified as having potential to affect port 1 of component `X`. The ic-pathway represented by the squiggle in the diagram to the right in Figure 1 has been added to component `Y`. When the dependence set is calculated using component `Y'`, ports associated with connectors `B` and `E` and component `Z` are added to the set of dependencies.

As a real life example, one can imagine such a situation arising in the development of two components designed to work within a banking system. Suppose that `Get_Account_Balance` is a type of component. Components of this type take a bank-user's account number, checking account information, and overdraw-protection account (protection account for short) information as inputs and returns the user's debt to the protection account as well as the user's available cash balance as outputs. Now suppose that `AB1` and `AB2` are components of this type. `AB1` determines the user's available cash balance by examining the current checking account balance; `AB2` on the other hand, considers available credit in the protection account to be available cash. `AB2` returns the sum of the checking account balance and available protection as the user's available cash balance. This additional connection between the input of the protection account information and the available balance output results in additional dependencies on all components required for managing the protection account. If an incorrect balance is reported, these components must be included in the set of components that could have contributed to the miscalculation.

3 Automated Dependence Analysis

In this section we provide a brief overview of Acme [4] and discuss the composition of an Acme description of our example system. We then introduce Aladdin [8], a tool that we have developed to identify chains of dependencies in architectural descriptions, and discuss its use in reasoning about the effects of replacing component `Y` with the variant component, `Y'`, that includes one additional pathway.

The development of ADLs is still at a point where there is a general lack of agreement on

```
Component X = {
    Port Start;
    Port 1;
    Port 2;
    Port 3;
    Port 4;
    Port Exit;
    Property paths = {
        [src="Start"; target="3";
                      relationship="causes"]
        [src="Start"; target="4";
                      relationship="causes"]
        [src="1"; target="Exit";
                      relationship="causes"]
        [src="2"; target="Exit";
                      relationship="causes"]
    };
};
```

Figure 2: Acme description of component X.

the full set of linguistic concepts required to describe software architectures. Nevertheless, there is an emerging consensus on a core set of concepts that primarily have to do with the structural aspects of software architectures. Recognizing this emerging consensus, the designers of the Acme language are attempting to represent a useful intersection of existing ADLs as a means to support some degree of interoperability among their associated tools. Additional goals for the language include providing a descriptive standard for architectural tools, assistance in the development of new ADLs, and a language that is accessible to most system developers.

The language provides seven basic constructs for describing software architectures. To some extent, these constructs serve as a "least-common denominator" and, therefore, some important language-specific concepts cannot be directly represented. As mentioned above, the orientation of the constructs is toward structure, so the missing concepts center on behavior; this is the same issue that arises in specification of COTS components. An additional Acme construct, *Property*, allows an architect (or developer of an ADL-to-Acme translator) to describe properties that may be useful for analysis purposes, but not representable using the other constructs. We define a "path" property that is used by Aladdin to reason about the dependencies in a system. A path property is specified to indicate the ability of an input to contribute to the stimulation of an output in some way. Figure 2 shows the Acme description of component `X` of system `S`. The inputs and outputs are listed as ports and annotations of the ic_pathways are specified as `path` properties after using the Acme keyword "Property".

Aladdin [8] is a tool developed at the University of Colorado that identifies dependencies in software architectures. It can be used as a stand-alone tool or in conjunction with ADLs. It was designed to be easily integrated with ADL tool kits developed elsewhere, and is currently available for use in analyzing Acme and Rapide [7] architectural descriptions.

Figure 3 shows the use of Aladdin to study the effects of adding the new pathway to component Y. If one thinks of an architectural description as a set of boxes and arrows in a diagram as we did in Section 2, then one can think about Aladdin as walking forwards or backwards from a given box, traversing arrows either from heads to tails or vice versa. In Aladdin, the arrows are called *links* and the process of walking (i.e., performing a transitive closure) over the links is called *chaining*.

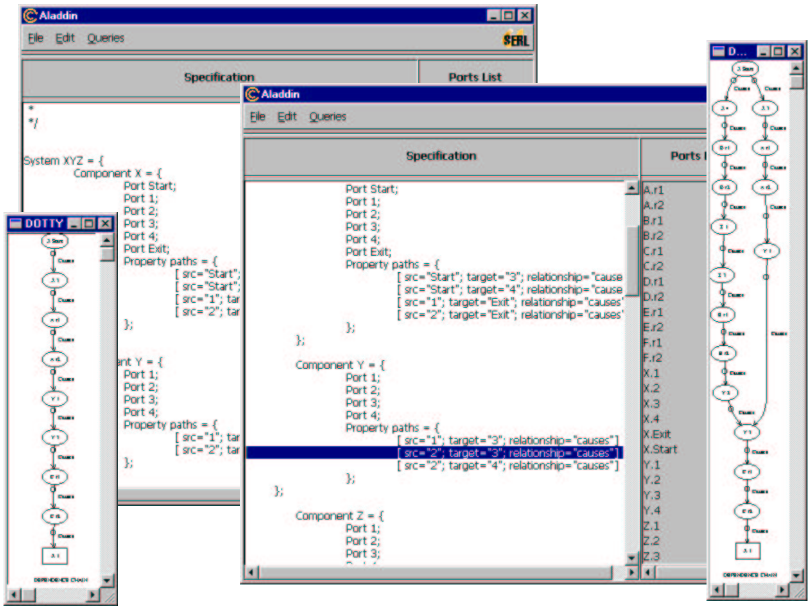
Aladdin's analysis is performed on demand in response to an analyst's query. A query might request information about the existence of anomalous dependence relationships, or might request information about the parts of the system that could affect or be affected by a specific component port. Figure 3 contains a combined screen dump from two uses of Aladdin to analyze the two variants of our example system. An Acme description of system S is displayed in the left pane of the main Aladdin window to the left in the figure and the Acme description of S′ is shown in the main Aladdin window to the right. The right pane of each window displays the list of component ports that have been identified from the respective descriptions.

The analyst can select to perform any of several queries. The analyst can choose to see a list of ports with no source or those with no target, which are two kinds of port-related anomalies. Ports with no source or no target may indicate an unspecified connection or they may indicate a function of the component that is not used in this system. The analyst can also choose to create a chain. If "Queries" is selected, then the window "Get Query" appears. The analyst selects a query, in this case wanting to see a chain of all the ports in the architecture that could causally affect port X.1. Aladdin uses dotty [2], a graph layout tool, to display the resultant chains. The windows on the left and right of the main Aladdin window show chains of dependencies for port X.1 of two variants of system S. A chain is displayed as a directed graph rooted at the rectangular vertex representing the specified port of interest, in these cases the vertex X.1 at the bottom of the graph. The arcs are labeled with a relationship type and represent direct dependence relationships between pairs of ports. The vertices of the graph represent all ports that could cause the port of interest, X.1, to be activated.

The branch in the dependency chain for X.1 shown in the dotty window to the right in the figure represents the additional dependencies that are identified when the ic-pathway Y.2⟶Y.3 is added to component Y. Aladdin makes such differences visually accessible to the analyst. Compare the chain of dependencies of X.1 computed without benefit of the knowledge of ic-pathways that is shown in Figure 4. In this case X.1, the port of interest, is shown at the top and a cycle of possible dependencies is indicated by the fact that arcs enter as well as exit from the vertex representing X.1. Clearly either chain shown in Figure 3 is more useful for analysis purposes.

4 Conclusions

Component annotations provide a means for reasoning about important system properties early in the development process as well as when considering replacing one component with another during system evolution. We envision that designers of analysis tools will specify component annotations required to support their particular kind of analysis. If a component manufacturer wishes for their component to be included in a system over which this type of analysis is to be performed, the annotations must be provided and certified by an independent certification agency. In this COTS world, system builders can reason about interoperability of system components with confidence. We use Acme, an extensible modeling language, to describe component interfaces and annotations. We perform automated dependence analysis over systems composed of Acme-described components to determine far-reaching affects associated with making changes to individual components.



```
*
*/

System XYZ = {
        Component X = {
                Port Start;
                Port 1;
                Port 2;
                Port 3;
                Port 4;
                Port Exit;
                Property paths = {
                        [ src="Start";
                        [ src="Start";
                        [ src="1"; tar
                        [ src="2"; tar
                };

nt Y = {
                Port 1;
                Port 2;
                Port 3;
                Port 4;
                Property paths = {
                        [ src="1"; tar
                        [ src="2"; tar
                };
```

```
                Port Start;
                Port 1;
                Port 2;
                Port 3;
                Port 4;
                Port Exit;
                Property paths = {
                        [ src="Start"; target="3"; relationship="cause
                        [ src="Start"; target="4"; relationship="cause
                        [ src="1"; target="Exit"; relationship="causes'
                        [ src="2"; target="Exit"; relationship="causes'
                };
        };

        Component Y = {
                Port 1;
                Port 2;
                Port 3;
                Port 4;
                Property paths = {
                        [ src="1"; target="3"; relationship="causes"]
                        [ src="2"; target="3"; relationship="causes"]
                        [ src="2"; target="4"; relationship="causes"]
                };
    };

        Component Z = {
                Port 1;
                Port 2;
                Port 3;
```

Figure 3: The use of Aladdin to identify differences in dependence chains.

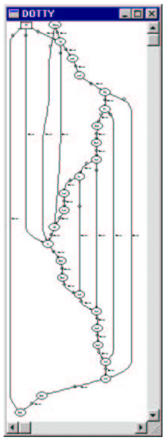


Figure 4: The dependence chain for port `X.1` when ic-pathways are unknown.

REFERENCES

[1] D. Compare, P. Inverardi, and A.L. Wolf. Uncovering Architectural Mismatch in Dynamic Behavior. *Science of Computer Programming*, 33(2):101–131, February 1999.

[2] E.R. Gansner, E. Koutsofios, S.C. North, and K.-P. Vo. A Technique for Drawing Directed Graphs. *IEEE Transactions on Software Engineering*, 19(3):214–230, March 1993.

[3] D. Garlan, R. Allen, and J. Ockerbloom. Architectural Mismatch: Why Reuse is So Hard. *IEEE Software*, 12(6):17–26, November 1995.

[4] 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.

[5] J. Han. An Approach to Software Component Specification. In *Proceedings of the 1999 International Workshop on Component Based Software Engineering*, pages 97–102, May 1999.

[6] W. Kozaczynski. Composite Nature of Component. In *Proceedings of the 1999 International Workshop on Component Based Software Engineering*, pages 73–77, May 1999.

[7] D.C. Luckham and J. Vera. An Event-based Architecture Definition Language. *IEEE Transactions on Software Engineering*, 21(9):717–734, September 1995.

[8] J.A. Stafford, D.J. Richardson, 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.

[9] S. Vestal. *MetaH Programmer's Manual Version 1.27.* Honeywell, Inc., Minneapolis, MN, 1998.