Architecture-Level Dependence Analysis for Software Systems

Judith A. Stafford[†] and Alexander L. Wolf[‡]

[†]Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA 15213 USA
jas@sei.cmu.edu

[‡]Department of Computer Science
University of Colorado
Boulder, CO 80309 USA
alw@cs.colorado.edu

University of Colorado
Department of Computer Science
Technical Report CU-CS-913-00 December 2000

A version of this report appears in the
International Journal of Software Engineering and Knowledge Engineering
Vol. 11, No. 4, August 2001, pp. 431-452.

ABSTRACT

The emergence of formal software architecture description languages provides an opportunity to perform analyses at high levels of abstraction, as well as early in the development process. Previous research has primarily focused on developing techniques such as algebraic and transition-system analysis to detect component mismatches or global behavioral incorrectness. In this paper we motivate the utility and describe the challenges in developing a different kind of analysis for use at the architectural level, namely dependence analysis. Various kinds of dependence analyses have been used widely at the implementation level to aid program optimization, anomaly checking, program understanding, testing, and debugging. However, the languages used for architectural description offer quite different features than the languages for which traditional dependence analysis techniques have been developed. We describe our initial approach to architecture-level dependence analysis and illustrate that approach through a prototype tool we have built, called Aladdin, to automatically perform the analysis.

1 Introduction

Software architecture descriptions are intended as models of systems at high levels of abstraction [5, 27, 30, 37]. They capture information about a system's components and how those components are interconnected. Some software architecture descriptions also capture information about possible states of components and about the component behaviors that involve component interaction; states and behaviors internal to a component are typically not considered at the architectural level. Formal notations for software architecture description, often referred to as *architecture description languages* (ADLs), allow one to reason about properties of software systems at correspondingly high levels of abstraction. The analysis techniques that have been developed for these languages have, in general, focused primarily on correctness properties, such as liveness and safety [2, 17, 22, 25].

But there are many other kinds of questions in addition to correctness that one might want to ask at an architectural level for purposes as varied as localizing faults, determining the impact of changes, minimizing regression tests, reengineering a system, reusing components, and managing a developer's workspace. Here we list a small number of examples.

1. Are there any components of the system that are never needed by any other components of the system?
2. If this component is to be reused in another system, which other components of the system are also required?
3. Which components of the system contribute to this piece of functionality?
4. What are the potential effects of dynamically replacing this component?
5. If this component is communicating with other components through a shared repository, with what other components could it be communicating?
6. 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?
7. If a change is made to this component, what other components might be affected?
8. If a change is made to this component, what is the minimal set of test cases that must be rerun?
9. 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 either affect or are affected by a particular component in some way. In fact, these kinds of questions are similar to those currently asked at the implementation level and answered through a technique known as *dependence analysis* applied to program code [1, 3, 10, 11, 16, 24, 26, 28, 29, 38]. It seems reasonable, therefore, to apply a similar technique 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 [36].

The traditional view of dependence analysis, formulated in the context of imperative implementation languages, is based on control and data flow relationships associated with functions and variables. In particular, control and data dependencies are identified by examining the flow of control through a program as well as the locations of definitions and uses of the program's variables.

The challenge in applying dependence analysis to architectural descriptions is to recast control and data flow relationships in terms of abstract components and their interactions. Although there is no commonly accepted and precise definition of what constitutes a component, the software architecture community is coming to consensus that components should be modeled as loosely coupled, nondeterministic, concurrent processes that communicate and synchronize through event interactions. Rapide [20] and Wright [2] are two ADLs representative of this approach. In both these languages, components have *ports*, each one of which *"defines a logical point of interaction between the component and its environment"* [2], where that interaction is either the receipt of some stimulus or the generation of some response modeled uniformly as *events*. The interaction behavior of the components is specified by modeling the possible sequences of stimulation and response events. In Rapide this is done by giving a partially ordered event set (called a *poset*), while in Wright this is done through a variant of CSP [14]. In any case, to identify dependencies among components, we must use the behaviors to identify the control and data relationships among communication ports. This abstract view of component interaction is quite different from the traditional notion of imperative program behavior, thus requiring a reformulation of the dependence analysis technique.

In this paper we motivate the utility and describe the challenges in developing a dependence analysis technique for use with ADLs. We describe our initial approach to architecture-level dependence analysis and illustrate that approach through a prototype tool we have built, called Aladdin, to automatically perform the analysis. We conclude by suggesting promising avenues for future research.

2 Architectures and Dependencies

In this section we begin by introducing the basic terminology of architectural description used in this paper. We then provide a brief overview of the Rapide architecture description language [20] as an illustration of architecture-level behavioral specification. Following this overview we introduce the conceptual foundations for architecture-level dependence analysis and argue why the specification of behavior in modern-day ADLs makes the development of dependence analysis techniques a challenge.

2.1 Terminology

Informally, an *architecture* is a set of components and the interactions among them.[1] Figure 1 provides a graphical depiction of the most basic structural elements of an architectural description. The following list gives fairly common definitions for these elements, as well as several other related terms that we use in this paper:

- *component*—a unit of a system that plays some well-defined functional role and having an interface through which it interacts with other components;
- *port*—a logical point of interaction between a component and its environment;
- *in port*—a port through which a component receives stimuli from its environment;
- *out port*—a port through which a component generates stimuli, possibly in response to some stimuli from its environment;

[1] Some definitions of architecture include a first-class notion of connectors as well as components [2, 27]. For the purposes of this paper, we do not make that distinction.

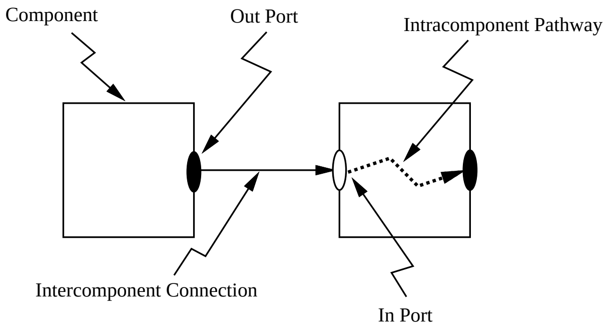


Figure 1: Architectural Terms.

- *intercomponent connection*—a directed relationship between an out port of one component and an in port of a different component;
- *intracomponent pathway*—a directed relationship between an in port and an out port of the same component;
- *event*—the change in the behavioral state of a component, including the generation or receipt of a stimulus;
- *system*—a set of interacting components.

For simplicity of presentation, we do not consider hierarchical relationships among components (i.e., a notion of *subcomponent*) in this paper.

If P_x is an out port of component C_x and P_y is an in port of component C_y, then an intercomponent connection from P_x to P_y indicates the ability for an event associated with P_x to directly stimulate an event associated with P_y. If P_i and P_j are two ports of component C, then an intracomponent pathway from P_i to P_j indicates the ability for an event associated with P_i to directly or indirectly stimulate an event associated with port P_j as specified by the interaction behavior of C. This interaction behavior of C is meant to capture the way in which it interacts with its environment and not with how it carries out its functional role.

2.2 Overview of Rapide

Rapide is a powerful language that represents well the sophistication of modern-day ADLs. For purposes of this paper, we present only a brief overview of the language, and restrict ourselves to the features used in a simple example, the familiar "gas station" problem,[2] which is shown in Figure 2 and which we discuss in detail in Section 3.

Components are defined in terms of their interfaces. Three types of components are described in Figure 2: a pump, a customer, and an operator. Interfaces are structured into separate sections that specify different aspects of the component's behavioral interaction with other components. Two sections are of relevance here. An *action* section contains the declaration of *in* and *out*

[2] It could be argued that the gas station problem is not representative of software architecture specifications, although it is widely used in the architecture literature [21, 25]. 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.

```
type Dollars is integer; -- enum 0, 1, 2, 3 end enum;
type Gallons is integer; -- enum 0, 1, 2, 3 end enum;

type Pump is interface
action in  On(), Off(), Activate(Cost : Dollars);
       out Report(Amount : Gallons, Cost : Dollars);
behavior
    Free : var Boolean := True;
    Reading, Limit : var Dollars := 0;
    action In_Use(), Done();
begin
    (?X : Dollars)(On ∼ Activate(?X)) where $Free ||> Free := False; Limit := ?X; In_Use;;
    In_Use ||> Reading := $Limit; Done;;
    Off or Done ||> Free := True; Report($Reading);;
end Pump;

type Customer is interface
action in  Okay(), Change(Cost : Dollars);
       out Pre_Pay(Cost : Dollars)Okay(), Turn_On(), Walk(), Turn_Off();
behavior
       D : Dollars is 10;
begin
       start ||> Pre_Pay(D);;
       Okay ||> Walk;;
       Walk ||> Turn_On;;
end Customer;

type Operator is interface
action in  Request(Cost : Dollars), Result(Cost : Dollars);
       out Schedule(Cost : Dollars), Remit(Change : Dollars);
behavior
       Payment : var Dollars := 0;
begin
       (?X : Dollars)Request(?X) ||> Payment := ?X; Schedule(?X);;
       (?X : Dollars)Result(?X) ||> Remit($Payment - ?X);;
end;

architecture gas_station() return root
is
    O : Operator;
    P : Pump;
    C1, C2 : Customer;
connect
    (?C : Customer; ?X : Dollars) ?C.Pre_Pay(?X) ||> O.Request(?X);
    (?X : Dollars) O.Schedule(?X) ||> P.Activate(?X);
    (?X : Dollars) O.Schedule(?X) ||> C1.Okay;
    (?C : Customer) ?C.Turn_On ||> P.On;
    (?C : Customer) ?C.Turn_Off ||> P.Off;
    (?X : Gallons; ?Y : Dollars)P.Report(?X, ?Y) ||> O.Result(?Y);
end gas_station;
```

Figure 2: Rapide Specification of the Gas Station Architecture.

actions, which specify the component's ability to observe or generate particular events. Implicitly declared actions represent events generated in the environment of the system that are watched for in an interface. The event `start` of the customer interface in Figure 2 is an example of an implicitly declared action. A *behavior* section, which may contain local declarations, describes how the component reacts to observed events and generates events associated with an out action. Behaviors are defined in an event pattern language. Patterns are sets of events together with their partial ordering, which is represented by a so-called *poset*.

Component types are instantiated and then connected to form architectures. The *architecture* declaration at the bottom of Figure 2 instantiates one operator, one pump, and two customers. The semantics of connections between architectural elements are specified through rules. Connection rules have a trigger, an operator, and a body. Rapide uses four kinds of connections in connection rules. The only one of concern for our example here is the agent connection (written syntactically as "`||>`"). In an agent connection, the observation of the pattern described in the trigger asynchronously generates the events in the body.

The behavior section of an interface contains state transition rules that are similar in structure to connection rules. Thus, a transition rule is composed of a trigger, an operator, and a body. The agent operations described above are also used as operators in the transition rules. The trigger may be a pattern or a boolean expression, while the body may be a state assignment or it may generate a poset. Conceptually, when the appropriate pattern of events occurs, the events in the body of the rule are triggered. In the gas station example, the body of behaviors are either state assignments or simply the generation of a single event.

Rapide provides placeholders for use in patterns and expressions. These are designated with the symbol "`?`". Placeholders are used in comparisons, dynamic generation of components, as iterators, or to bind the values of parameters. In the case of dynamic creation of components, a placeholder serves as a universal quantifier. For instance, in the gas station example

(?C : Customer; ?X : Dollars) ?C.Pre_Pay(?X) `||>` O.Request(?X);

is interpreted to mean that "for any component of type `customer`, there is to be an agent connection between the customer's `Pre_Pay` action and the operator's `Request` action, where the number of dollars in the `Request` action is bound to the number of dollars in the `Pre_Pay` action".

2.3 Architecture-Level Dependence Analysis

If a system's architectural description is available early in the development process, then it provides a basis from which to reason about the system before effort is expended on later development phases and artifacts. If a precise mapping is maintained between the architectural description and the implementation, then the high-level nature of the architectural description can provide a reliable abstract basis upon which to perform tractable analyses of the implementation.

The degree to which both early and high-level analyses based on architectural descriptions provide benefit depends directly on the sophistication of the language used to capture the architecture. Dependence analysis can be applied to simple "box and arrow" diagrams of software architectures, but can then yield only coarse-grained information about structural system properties. This is illustrated in Figure 3, which shows the four components of the gas station architecture and the relationships among those components based on the specified connections among named ports. By itself, the interconnection structure offers only a gross understanding of architectural dependencies. In general, when performing a transitive closure over the connections in such a description, one is likely to end up encountering all the components in the architecture. Conversely, dependence

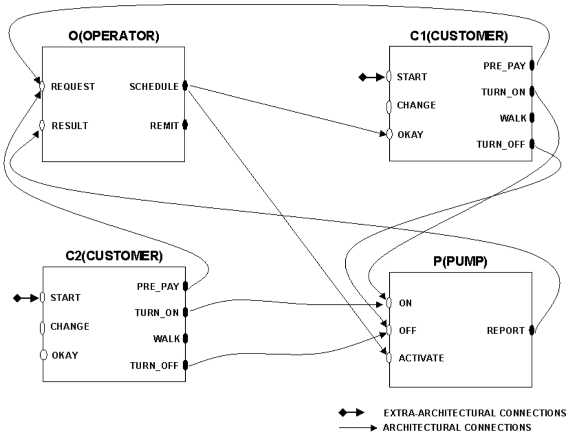


Figure 3: Intercomponent View of Gas Station Dependencies.

analysis applied to descriptions that include detailed behavioral information about component interactions have the potential to yield correspondingly detailed results. Figure 4 shows a view the gas station architecture of Figure 3 enhanced by an analysis of intracomponent pathways.

In the established formulation of dependence at the implementation level, the two major contributors to dependencies are control and data. In that context, a *control flow graph* (CFG) is normally used as the basis for calculating the dependencies. A CFG is a directed graph in which the vertices represent program elements (usually statements or basic blocks) and the arcs represent the potential for control to flow from the arc's source to the arc's target. Typically, the vertices of the CFG are annotated with information about the definition and use of variables in the program. This combination of control and data information is then used to identify control and data dependencies among the program statements. The *program dependence graph* (PDG) [8, 28] combines control and data dependence information into a single, compact representation. A dependence between two vertices in a PDG is classified as *indirect* when the dependence relationship exists transitively through intermediate vertices. If the relationship does not involve transitive dependencies, then it is said to be a *direct* dependence.

The realm of architectural description also involves both control and data relationships, and here too we can base the analysis on some sort of representation of control. But the nature of control in ADLs is quite different from that of control in the languages that, to date, have been the targets of dependence analysis. In particular, modern-day ADLs such as Rapide and Wright are *event based*, which has the following implications: (1) unlike synchronous procedure call or task rendezvous, event interactions are asynchronous and events may or may not be received and (2) unlike the simple conditional statements that govern procedure invocation, complex event patterns can be used to constrain when a port may be stimulated. While these characteristics could be achieved through explicit programming in traditional analysis targets (e.g., C, C++, and Ada), the fact that they are first-class semantic concepts in ADLs makes them important objects of analysis.

The key to architectural dependence analysis lies in determining ways to minimize the set of potential dependencies. One approach is to create a representation of the direct intercomponent dependencies, and then refine this by including summarizations of intracomponent dependencies, thereby limiting the number of potential pathways among the in and out ports within a component. The results of such an approach are reflected in Figure 4. For example, the analysis reveals that a

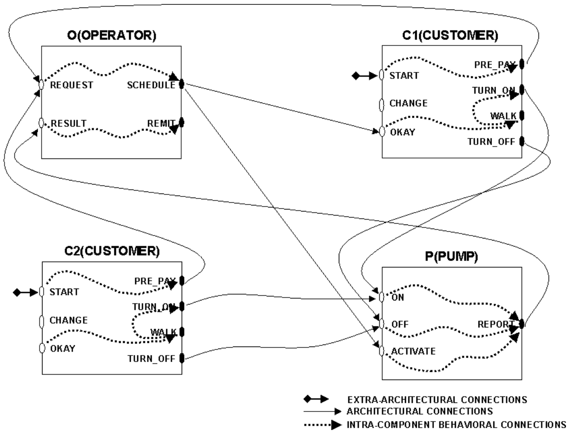


Figure 4: Enhanced View of Gas Station Dependencies.

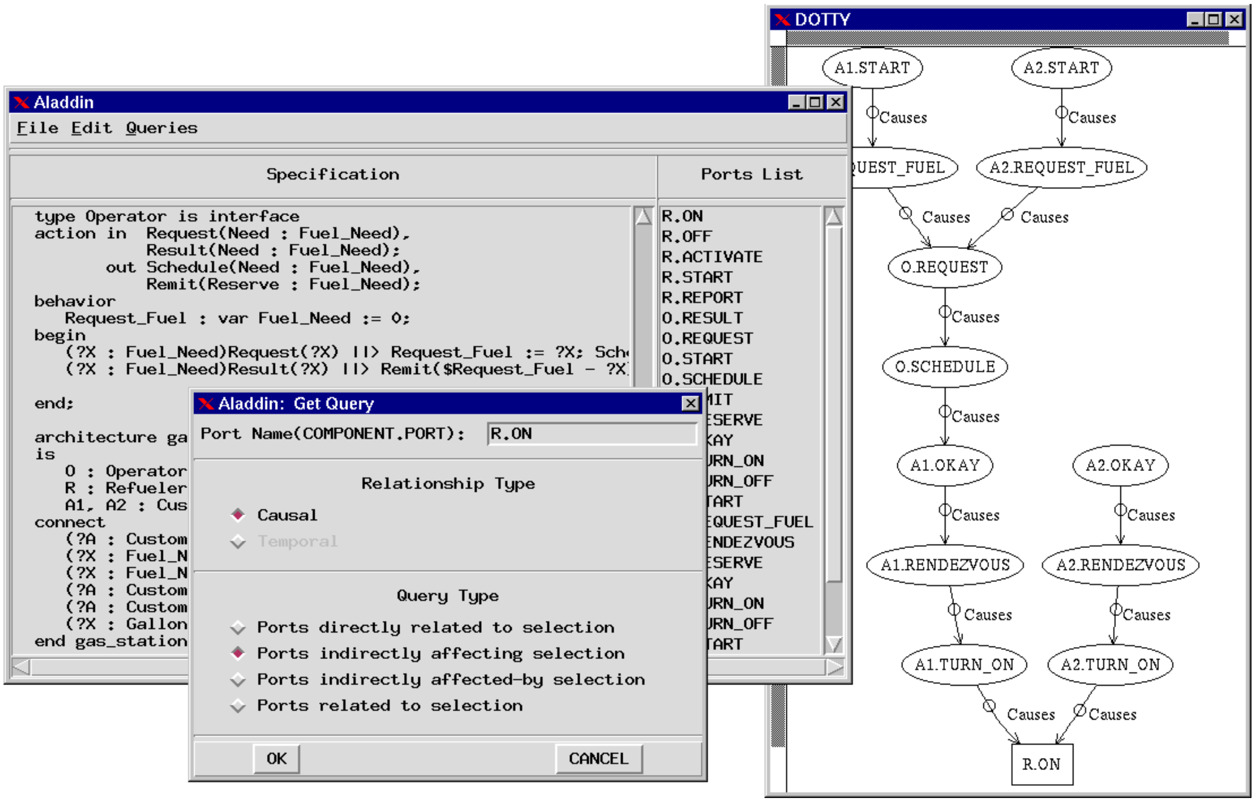
response through component `C1`'s out port `Pre_Pay` implies a stimulus through its in port `Start` and not through either of its in ports `Okay` or `Change`, a fact not discernible from Figure 3. There are also a number of critical anomalies evident only from the analysis result depicted in Figure 4, which we discuss below. The ability to specify intracomponent interface behavior is a key distinguishing feature of modern-day ADLs. Furthermore, the constraints on component interactions imposed by event patterns introduce additional opportunities to explore ways in which to refine the analysis of event-based interactions.

3 A Prototype Architecture-Level Dependence Analysis Tool

Aladdin is a prototype tool for performing architecture-level dependence analysis. The main window of its user interface is shown to the left in Figure 5. The purpose of using Aladdin is to obtain information that can be used to study the dependence relationships among ports of an architecture as an aid to answering questions such as those listed in Section 1. For instance, in the gas station architecture described in Section 2.2, Aladdin exposes the existence of unused ports, helps locate specification coding errors, and supports investigation of how the replacement of a component would affect the rest of the system.

Aladdin takes architectural descriptions as inputs and, in response to queries by a user, produces a *chain* of direct dependencies rooted at a specified port in the architecture. Aladdin uses a two-phase algorithm to identify dependence chains. In Phase I, Aladdin computes and records all potential direct dependencies. Phase II is executed in response to a user request for the dependencies that are related, either directly or indirectly, to a specific port. The user does this through the interface shown in the center of Figure 5. The output of Phase II is a dependence chain such as the one shown to the right in Figure 5.

Aladdin has been carefully designed to separate language-specific processing tasks from language-independent analysis tasks, and so can be tailored for use with other languages. For example, besides its use with Rapide, Aladdin has also been made to work with Acme [9]. An 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 Rapide Design Team at Stan-



```
type Operator is interface
action in  Request(Need : Fuel_Need),
           Result(Need : Fuel_Need);
       out Schedule(Need : Fuel_Need),
           Remit(Reserve : Fuel_Need);
behavior
   Request_Fuel : var Fuel_Need := 0;
begin
   (?X : Fuel_Need)Request(?X) ||> Request_Fuel := ?X; Sch
   (?X : Fuel_Need)Result(?X) ||> Remit($Request_Fuel - ?X

end;

architecture ga
is
   O : Operator
   R : Refueler
   A1, A2 : Cus
connect
   (?A : Custom
   (?X : Fuel_N
   (?X : Fuel_N
   (?A : Custom
   (?A : Custom
   (?X : Gallon
end gas_station
```

```
R.ON
R.OFF
R.ACTIVATE
R.START
R.REPORT
O.RESULT
O.REQUEST
O.START
O.SCHEDULE
MIT
ESERVE
KAY
URN_ON
URN_OFF
TART
EQUEST_FUEL
ENDEZVOUS
ESERVE
KAY
URN_ON
URN_OFF
TART
```

Aladdin: Get Query

Port Name(COMPONENT.PORT): R.ON

Relationship Type

● Causal
○ Temporal

Query Type

○ Ports directly related to selection
● Ports indirectly affecting selection
○ Ports indirectly affected-by selection
○ Ports related to selection

Figure 5: Screen Shot of Aladdin in Use.

ford University and the Acme research group at Carnegie Mellon University, respectively, and then directly incorporated into Aladdin. In the future, we expect to obtain other such ADL front ends in order to provide dependence analysis for a variety of languages.

In the remainder of this section we detail the generation of an Aladdin dependence table and our use of Aladdin to analyze the Rapide specification for the gas station architecture shown in Figure 2.

3.1 Construction of the Dependence Table

For a given ADL, it is necessary to understand the various ways in which any two architectural elements can be related so that a dependence table for a particular architecture can be constructed. This tabular representation could alternatively be viewed as a dependence graph. In a Rapide description, the relationship types associated with connection and transition rule operators induce dependence relationships. The table shown in Figure 6 represents the dependence relationships of the gas station example. Notice that this table captures the relationships depicted in Figure 4.

A dependence table has m rows and n columns, where m is the number of ports in the architecture plus any implicitly declared interactions with the environment (e.g., the action `start`), and n is the number of ports in the architecture plus any emissions to the environment. A cell in the dependence table represents the presence or absence of a dependence relationship between the pairs of architectural elements that define the column and row in which the cell resides. The columns

	O				P				C1						C2					
	OUT		IN		OUT	IN			OUT				IN		OUT				IN	
	Sch(D)	Rem(D)	Req(D)	Res(D)	Rep(G,D)	On	Off	Act(D)	PP(D)	T_On	Walk	T_Off	Okay	Chg(D)	PP(D)	T_On	Walk	T_Off	Okay	Chg(D)
O																				
OUT																				
Sch(D)								x					x							
Rem(D)																				
IN																				
Req(D)	x																			
Res(D)		x																		
P																				
OUT																				
Rep(G,D)				x																
IN																				
On					x															
Off					x															
Act(D)					x															
C1																				
OUT																				
PP(D)			x																	
T_On						x														
Walk										x										
T_Off							x													
IN																				
Okay											x									
Chg(D)																				
start									x											
C2																				
OUT																				
PP(D)			x																	
T_On						x														
Walk																x				
T_Off							x													
IN																				
Okay																	x			
Chg(D)																				
start															x					

Figure 6: Gas Station Dependence Table.

in the table represent the dependent in the relationship and the rows represent the source of the dependence. For instance, if a is dependent on b, then the cell at column a and row b details that relationship.

In the case of the gas station architecture of Figure 2, there is an agent connection between the `Turn_On` action for each customer and the `On` action of the pump, where the `On` action is the body, and thus the dependent. Therefore, this connection is mapped to a causal relationship and is recorded in the table cells (C1.T_On,P.On) and (C2.T_On,P.On) in Figure 6.

While composite event patterns in triggers and bodies of rules can be complex, they do not need to add complexity to the tabular representation. This is because, in general, there is no way to statically determine which of the events associated with actions in the trigger will be the cause of a given triggering. Thus, all are recorded as sources for the relationship. For example, in the last rule of the behavior section of the `Pump` interface of the gas station example shown in Figure 2, either the event `Off` or the internally generated event `Done` can stimulate the generation of a report. Actions `On` and `Activate` are the only possible external events associated with stimulation of `Done` so the syntax of this rule results in `Report` being identified as being dependent on `On` and `Activate`, as well as on `Off`. This is a conservative approach and we are investigating methods for reducing the generation of false dependencies based on the semantics of the pattern language used by Rapide.

3.2 Summarizing Internal Behavior

The local declaration of variables and actions in the behavior section of a Rapide component interface specification may be involved in the eventual connection of an in port to an out port. However, the details of this internally generated behavior is below the level of abstraction that is of interest to us for building dependence chains. Thus, Aladdin applies a summarization algorithm in order to safely abstract away these low-level details. In the gas station example, the pump has several local declarations; Aladdin summarizes the effects of their inclusion, thus producing the intracomponent pathways shown in Figure 4.

The most conservative approach to identifying dependencies between in and out ports of a component is to assume that each in port has the potential to affect each out port. This is equivalent to viewing the component as a black box. It is often the case that we can be more precise than that by performing intracomponent analysis and identifying the potential pathways from in ports to out ports. This approach, however, requires that an expensive analysis be performed. As a compromise, we have developed a summarization algorithm that allows us to reduce the number of intracomponent connections.

As an example of an opportunity to apply our scheme, consider the behavior associated with the `Pump` interface in the gas station example. This interface contains the internal actions `In_use` and `Done`. These actions are used to describe the transformational behavior of a `Pump` component. The first rule in the behavior section can be read roughly as follows. When both `On` and `Activate` events are observed and the variable `Free` is "true", then do the following sequence of things: set the variable `Free` to "false", set the variable `Limit` to the value of the input parameter `Cost`, and then emit the signal that the pump is `In_use`. The double semicolon signifies the end of a sequence of actions. The action `In_use` is an internal action and, therefore, its occurrence is only observed within the scope of `Pump`. When `In_use` is observed, then the internal variable `Reading` is set to the value of `Limit` and the internal action `Done` signals the completion of pumping. Once this event is observed the value of the variable `Free` is set back to "true" and a report is generated that contains the value of the variable `Reading`. The trigger for `Report` reveals an alternative stimulus for generation of a report. If the external event `Off` is observed at any time before the internal

```
Begin SUMMARIZE(out_port)
src_ports[] = GET_SRC_PORTS(out_port)
loop for each src_port in src_ports[]
  if src_port is an in port
    then CREATE_LINK(src_port, target_port)
  else if src_port is an internal port
    in_ports[] = GET_IN_PORTS(component)
      for each in_port in in_ports[]
        rels[] = GET_CELL()
        for each rel in rels[]
          if rel.src = in_port and
              rel.target is an internal port
            CREATE_LINK(rel,target_port)
          endif
        endfor
      endfor
  endif
endloop
End SUMMARIZE
```

Figure 7: Internal Behavior Summarization Algorithm.

event `Done`, then a report is generated with the current value of the variable `Reading`, which will be equal to $0 unless pumping was completed, in which case it would be equal to the value of `Limit`.

The basis for our approach to summarization lies in the fact that events can be either internally or externally visible, but we do not want to concern ourselves with the internal events. So we ask ourselves how can we ignore internal events and still gain some benefit from the fact that they contribute to the creation of intracomponent pathways? The answer to this question leads to an elegant summarization algorithm, presented in Figure 7, that can result in significant reductions in the numbers of identified intracomponent pathways, depending on the transformational relationships within the component. While the gas station example provides an opportunity to describe the intuition behind the algorithm, the behavior associated with its components is such that there is no gain in reduction of intracomponent pathways. However, the benefit of lowered cost of analysis is realized.

The intuition behind our algorithm is based on the following observations. If an internal port is being used as a stimulant, then it must appear in the body of some other transition rule. The trigger of that rule must contain an in and/or another internal action. In either case, the internal action must be the target of stimulation originating at some external `in` action. Therefore, links are constructed between each in port associated with an event that directly stimulates any internal event and, additionally, all out actions associated with events that are directly stimulated by internal events. This algorithm is conservative and may, in fact, result in the creation of several false links. As mentioned in Section 2, we are investigating means to exploit patterns used as constraints, triggers, and guards on transitions in order to improve this summarization algorithm as well as to develop other means for improving the precision of chains.

3.3 Creating Chains of Dependencies

Figure 6 is the result of the Aladdin table builder processing the Rapide specification for the gas station example. Once this representation is available, Aladdin builds dependence chains in response to user queries. Building dependence chains that capture how one component *affects* another component involves creating links by beginning at the row labels and locating the related column label, whereas building dependence chains that capture how one component is *affected by* another component involves linking in reverse, from column to row labels.

Consider a query to uncover the cause of the `P.Activate` event in the gas station example. The transitive closure that constructs an affected-by chain of events begins at the columns and looks to the related events in the rows. The `O.Schedule` event is the only possible source of the `P.Activate` event. This relationship is transitive and, assuming that all possible prior events occurred, we repeat the process for each of the related events. In this case, only the `O.Schedule` event is directly caused by `O.Request`, which is generated by any one or both of the `C1.Pre_Pay` or `C2.Pre_Pay` events, which in turn may only be preceded by the `start` event of the `Customer` interface.

Construction of an affects chain that has the `O.Schedule` action as its first link begins by checking the cells that have entries in the `O.Schedule` row. The `P.Activate` and `C1.Okay` are the only columns that have entries for this row. These relationships are also transitive, so the chain is constructed in a similar, though reversed manner, to the affected-by chain.

3.4 Using Aladdin

The Query function of Aladdin is accessible by way of the interface shown in the center of Figure 5. This interface can be used to access the following information about the various ports appearing in an architectural description.

- ports with no destination;
- ports with no source;
- ports directly related to a particular port;
- ports indirectly affected by a particular port;
- ports indirectly affecting on a particular port;
- ports indirectly related to a particular port; and
- cycles involving a particular port.

Figure 8 shows the results of running the first two kinds of queries against the gas station specification. The circles in the figure indicate the anomalies that were revealed by these analyses. Of course, only the engineer can determine whether these anomalies are actual faults in the specification. For instance, it is possible that an unused event has been included in an interface because it is expected to be needed in the future, not because it is a misconnection.

The architectural description for the gas station contains a serious error. In particular, it is never possible for the second customer to pump gas. The first customer does not suffer this problem. So, after this problem is discovered through simulation of the architecture, the engineer uses Aladdin to query for the events that could lead to the `P.On` event. The resulting chain is shown in Figure 9 by highlighting the relevant dependence relationships. The actual output from Aladdin is given in the window shown to the left in Figure 5. The chain reveals that when the second customer pays for gas, the first customer is given credit and could pump again if still in the gas station, while the second customer is never given the okay to pump. That is, the chain back through the

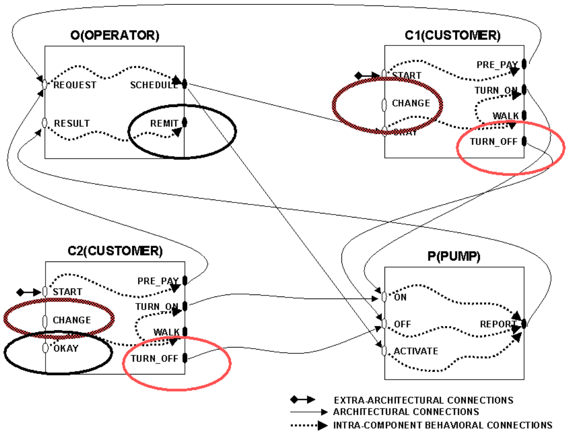


Figure 8: Gas Station Anomalies.

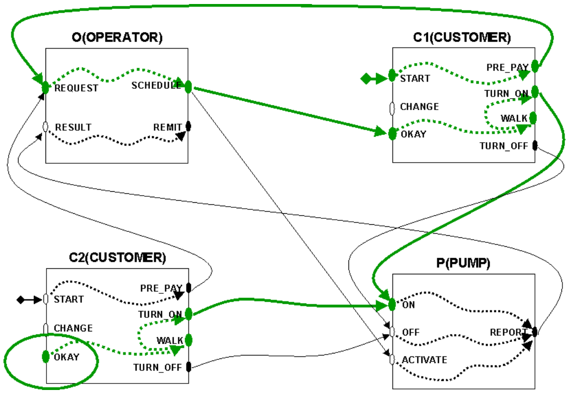


Figure 9: Gas Station Fault Localization.

`C2.Okay` event does not lead to a legal start state for the system. Additionally, by comparing the two subchains, we can see that the `C1.Okay` event is generated no matter who paid for the gas.

With this information in hand, the engineer can examine the architectural specification in Figure 2, look at the connections involving the scheduling of customers, and find that `C1` is the only customer that receives the `Schedule` event from the operator. The information tells the engineer that there is a fault in the connection based on the `O.Schedule` action in the architecture; the fault is then repaired.

This simple example already demonstrates the leverage that architecture-level dependence analysis can give to an engineer. The analysis can be performed early in the life cycle and helps to reveal a significant fault in the system. One would expect additional errors to be introduced as the system is refined and implemented. Thus, detection and correction of errors during the first stages of development produce overall savings in development costs.

4 Related Research

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

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 [8] introduced the program dependence graph for use in compiler optimization. Podgurski and Clarke [28] proved that the combination of data and control dependencies produces a conservative set of behavioral dependencies. Others have focused on problems associated with procedure calls, variable aliases, and concurrency-related control mechanisms [6, 12, 13, 15, 18, 19]. Recently, work has been directed at improving the precision and efficiency of program dependence analysis [31, 32, 35].

Our notion of dependence chaining is intended to support a range of analysis applications, including what would amount to an architectural slice. Program slicing was originally introduced by Weiser [38] as an aid to program debugging. Sloane and Holdsworth [33] suggest generalizing the concept of program slicing and show the potential in syntax-based slicing of non-imperative programs.

Recent work in program slicing has focused on using slices to improve the efficiency of model checking. Two such approaches explore slicing of programs written in VHDL [7] and Promela [23]. The concurrency model of these languages closely resemble that found in ADLs such as Rapide and Wright. The approach taken in each case is to interpret a reduced set of the concurrency-related control constructs of the language in terms of sequential program constructs, representing them in the system dependence graph of Horwitz et al. [15]. While these approaches produce a large number of false dependencies, the results have proven useful for their intended purpose of improving the efficiency of model checking.

Zhao [39] describes an approach to slicing architectural descriptions written in Wright for the purposes of identifying reusable subarchitectures. The work he describes is similar in nature to our work, but details of the method for determining related components are unstated.

Naumovich et al. [25] 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 analyses already provided by these tools.

5 Conclusion

In this paper we have introduced the general challenges of performing dependence analyses on software architecture descriptions, and presented a dependence analysis technique that we developed as a vehicle for exploring these challenges. We have demonstrated the technique through an example application of a prototype tool, Aladdin, that provides automated support for reasoning about the dependence relationships among the components of a system described in a modern-day architecture description language. The example illustrates the important leverage that software developers can gain from the effort they must expend in documenting a system's architecture in a formal notation.

In addition to the benefits discussed in this paper, software architecture descriptions have further promise in decreasing the cost and improving the quality of software development. Yet, before these benefits can be realized, there must be additional research carried out. Below we identify several such areas of research, going beyond only those to support dependence analysis.

- Automated analysis requires that developers provide some form of machine-processable description of the software architecture. Therefore, development environments that support compilation and simulation of architectural descriptions need to be made available. The tool set supporting Rapide is an early example of such an environment.
- Automated analysis techniques similar to those available to programmers will need to be developed for machine-processable ADLs. We have described one such technique in this paper, and mentioned others concerned with establishing correctness. Other types of analyses, such as performance analysis [4, 34], would also be useful, and would provide further incentive to developers to expend the effort to create formal descriptions.
- Methods for avoiding, or recovering from, *architectural drift* [27] need to be developed. Architectural drift is the widely recognized phenomenon in which the embodiment of an architecture tends to vary from its original form over time. Architecture-based analyses are useful only in as much as an implementation reflects the structural and behavioral properties of the architectural description that might be used as the basis for analysis.
- Developing automated techniques for mapping between various architectural views would improve the extent to which one can reason about the impact of making a change to some artifact of the development process, be that a software artifact or a workflow artifact. For instance, if a deadline is approaching and a developer wants to stall development of a specific component, the full impact of this decision can be assessed by analyzing a combination of an architectural view of work assignments and the structure of the software.
- Given mappings among architectural views, an algebra for reasoning about the relationships associated with various architectural views would support definition of extended notions of dependencies among artifacts associated with software development.

Advances in any of the areas listed above would increase the applicability and utility of software architecture analysis in general, and architecture-level dependence analysis in particular.

Acknowledgments

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. The work of J. Stafford was also sponsored in part by the Software Engineering Institute, a federally funded research and development center sponsored by the U.S. Department of Defense.

References

[1] F.E. Allen. Control Flow Analysis. *SIGPLAN Notices*, 5(7), July 1970.

[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] S. Balsamo, P. Inverardi, and C. Mangano. An Approach to Performance Evaluation of Software Architectures. In *Proceedings of the 1998 Workshop on Software and Performance*. IEEE Computer Society, October 1998.

[5] L. Bass, P. Clements, and R. Kazman, editors. *Software Architecture in Practice*. Addison-Wesley, Reading, Massachusetts, 1998.

[6] J. Cheng. Slicing Concurrent Programs—A Graph-Theoretical Approach. In *Automated and Algorithmic Debugging*, number 749 in Lecture Notes in Computer Science, pages 223–240. Springer-Verlag, 1993.

[7] E.M. Clarke, M. Fujita, S.P. Rajan, T. Reps, S. Shankar, and T. Teitelbaum. Program Slicing of Hardware Description Languages. In *Proceedings of the 10th IFIP WG10.5 Advanced Research Working Conference on Correct Hardware Design and Verification Methods*, September 1999.

[8] 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, October 1987.

[9] 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.

[10] 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. ACM SIGPLAN, June 1997.

[11] 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.

[12] 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.

[13] J. Hatcliff, J.C. Corbett, M.B. Dwyer, S. Sokolowski, and H. Zheng. A Formal Study of Slicing for Multi-threaded Programs with JVM Concurrency Primitives. In *Proceedings of the Static Analysis Symposium*, number 1694 in Lecture Notes in Computer Science, pages 1–18. Springer-Verlag, 1999.

[14] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, Englewood Cliffs, New Jersey, 1985.

[15] S. Horwitz, T. Reps, and D. Binkley. Interprocedural Slicing Using Dependence Graphs. In *Proceedings of the ACM SIGPLAN '88 Conference on Programming Language Design and Implementation*, pages 35–46. ACM SIGPLAN, July 1988.

[16] 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.

[17] 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.

[18] J. Krinke. Static Slicing of Threaded Programs. In *Proceedings of the ACM SIGPLAN/SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, pages 35–42. Association for Computer Machinery, July 1998.

[19] 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.

[20] 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.

[21] N. Madhav. Testing Ada 95 Programs for Conformance to Rapide Architectures. In *Proceedings of Ada-Europe '96*, number 1088 in Lecture Notes in Computer Science, pages 123–134. Springer-Verlag, June 1996.

[22] 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.

[23] L. Millett and T. Teitelbaum. Slicing Promela and its Applications to Model Checking, Simulation, and Protocol Understanding. In *Proceedings of the SPIN '98 Workshop*, pages 75–83, November 1998.

[24] G.C. Murhpy and D.N. Notkin. Lightweight Lexical Source Model Extraction. *ACM Transactions on Software Engineering and Methodology*, 5(3):262–292, July 1997.

[25] 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 Held Jointly with the 5th ACM SIGSOFT Symposium on Foundations of Software Engineering*, number 1301 in Lecture Notes in Computer Science, pages 77–93. Springer-Verlag, 1997.

[26] 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.

[27] D.E. Perry and A.L. Wolf. Foundations for the Study of Software Architecture. *SIGSOFT Software Engineering Notes*, 17(4):40–52, October 1992.

[28] 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.

[29] 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.

[30] M. Shaw and D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice-Hall, Englewood Cliffs, New Jersey, 1996.

[31] 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.

[32] 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 Computer Machinery, May 1999.

[33] 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.

[34] B. Spitznagel and D. Garlan. Architecture-Based Performance Analysis. In *Proceedings of the 10th International Conference on Software Engineering and Knowledge Engineering*, June 1998.

[35] 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.

[36] J.A. Stafford and A.L. Wolf. Architecture-Level Dependence Analysis in Support of Software Maintenance. In *Proceedings of the Third International Software Architecture Workshop*, pages 129–132, November 1998.

[37] J.A. Stafford and A.L. Wolf. Software Architecture. In G.T. Heineman and W.T. Councill, editors, *Component-Based Software Engineering: Putting the Pieces Together*. Addison-Wesley, Reading, Massachusetts, 2001.

[38] M. Weiser. Program Slicing. *IEEE Transactions on Software Engineering*, 10(4):352–357, July 1984.

[39] J. Zhao. A Slicing-Based Approach to Extracting Reusable Software Architectures. In *Proceedings of the 4th European Conference on Software Maintenance and Reengineering*, pages 215–223. IEEE Computer Society, February 2000.