Aladdin: A Tool for Architecture-Level Dependence Analysis of Software Systems

Judith A. Stafford[†], Debra J. Richardson[‡], and Alexander L. Wolf[†]

[†]Department of Computer Science
University of Colorado
Boulder, CO 80309 USA
{judys,alw}@cs.colorado.edu

[‡]Dept. of Information and Computer Science
University of California
Irvine, CA 92697 USA
djr@ics.uci.edu

University of Colorado
Department of Computer Science
Technical Report CU-CS-858-98 April 1998

ABSTRACT

The emergence of formal 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 present Aladdin, a tool that implements chaining*, a static dependence analysis technique for use with architectural descriptions. Dependence analysis has been used widely at the implementation level to aid program optimization, anomaly checking, program understanding, testing, and debugging. We investigate the definition and application of dependence analysis at the architectural level. We illustrate the utility of chaining, through the use of Aladdin, by showing how the technique can be used to answer various questions one might pose of a Rapide architecture specification.*

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.

1 Introduction

Software architecture descriptions are intended as models of systems at high levels of abstraction. They capture information about a system's components and how those components are interconnected. Some software architectures also capture information about the possible states of components and about the component behaviors that involve component interaction; behaviors and data manipulations internal to a component are typically not considered at the architectural level.

Formal software architecture description languages (ADLs) allow one to reason about the correctness of software systems at a correspondingly high level of abstraction. Techniques have been developed for architecture analysis that can reveal such problems as potential deadlock and component mismatches [3, 15, 18, 20]

In general, there are many kinds of questions one might want to ask at an architectural level for purposes as varied as specification error detection, reuse, reengineering, reverse engineering, fault localization, impact analysis, regression testing, and workspace management. For example, one might want to find out which components of a system could receive notification of a particular event generated by some component. Or, one might want to minimize the number of components that must be examined in order to find the source of a system failure. Or, one might like to anticipate which components of a system would need to be retested when one component is replaced by a new one.

These kinds of questions are similar to those currently asked at the implementation level and answered through static dependence analysis techniques applied to program code. It seems reasonable, therefore, to apply similar techniques at the architectural level, either because the program code may not exist at the time the question is being asked or because answering the question at the architectural level is more tractable than at the implementation level.

The traditional view of dependence analysis is based on control and data flow relationships associated with functions and variables (e.g., [13, 21, 30]). This type of analysis was originally applied to program optimization [2] in order to determine safe code restructuring. The development of efficient and precise dependence analysis algorithms continues as an active area of research among compiler designers (e.g., [4, 9]). The application of dependence analysis techniques to aid program understanding and impact analysis has also been widely studied (e.g., [19, 22, 24]).

Our research takes a broader view of dependence relationships that is more appropriate to the concerns of architectures and their attention to component interactions. In particular, both the structural and the behavioral relationships among components expressed in current-day formal ADLs, such as Rapide [16] and Wright [3], are considered. We are developing an architecture-level dependence analysis technique, called *chaining*, and implementing the technique in a tool called Aladdin. Aladdin is similar in concept to ProDAG [25], which is an implementation-level dependence analysis tool for Ada and C++ programs. Dependence analysis is performed by both ProDAG and Aladdin in a two-step process. First, an intermediate representation is created, and then language-independent analysis is performed over this representation. In Aladdin, the representation consists of a set of cells, where each cell represents the set of relationships that could exist between a given pair of architectural elements. This set is queried in order to construct chains of dependent elements.

We begin our presentation of chaining and Aladdin with a brief review of questions that might reasonably be asked at the architectural level and answered in part by dependence analysis. These questions further motivate the need for architecture-level dependence analysis. We then describe the concept of chaining and the architectural relationships that underlie the concept. This is followed by a description of how chains are created and used by Aladdin. The current version of Aladdin works on specifications written in the Rapide ADL, and thus the examples given below are cast in terms of that particular language. We then review related work and conclude with a discussion of our future plans, which include the adaptation of Aladdin to specifications written in Acme [8] and extension to other ADLs.

2 Motivating Questions

There are a variety of questions that should be answerable by an examination of a formal architecture description. Here we list a small number of those questions as motivation for the study of architecture-level dependence analysis.

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 through a shared repository, with what other components does it communicate?
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. They involve issues of correctness, impact of change, and traceability.

The first seven are questions answerable through analysis of a formal description of a system's architecture. Questions 8 and 9 are actually questions about the implementation, but ones that can better be answered with the aid of architecture-level analysis. Thus, the last two questions depend on a mapping between elements of the current version of the implementation and elements of the current version of the architecture, as illustrated in Figure 1. Architectural specifications and system implementations tend to drift apart if not carefully managed. Two approaches intended to support precise mappings are code generation [29, 31] and architecture recovery [6, 19].

3 Chaining

Chaining is a dependence analysis technique we propose for use at the architectural level. The individual chain links within a chain associate architectural elements that are directly related,

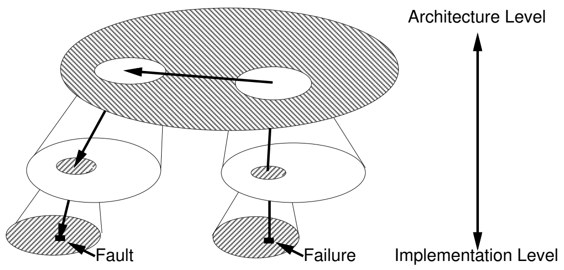

Figure 1: Architecture-Assisted Fault Localization.

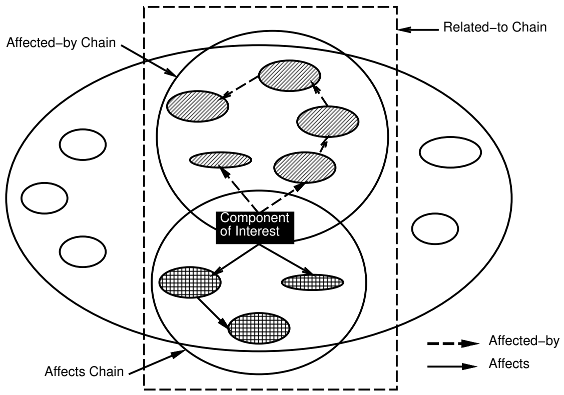

Figure 2: Types of Chains Relevant to a Component.

while a chain of dependencies includes associations among architectural elements that are indirectly related. Chaining is the process of applying dependence algorithms to architectural descriptions in order to create these sets of related elements.

We consider four kinds of elements in a software architecture.

- *Data elements*: contain the data to be transformed, communicated, or otherwise used by components.

- *Processing (or behavioral) elements*: provide the means for components to transform data.

- *Connecting elements*: provide the means for components to interact. These elements are also called ports.

- *State elements*: contain the (current) state of components in terms of both data and processing.

From the perspective of a component C, we find it useful to describe three types of chains (see Figure 2).

- *Affected-by chains*: consist of the set of components and/or their elements that could potentially affect an element of C. These are the elements that C is affected by.

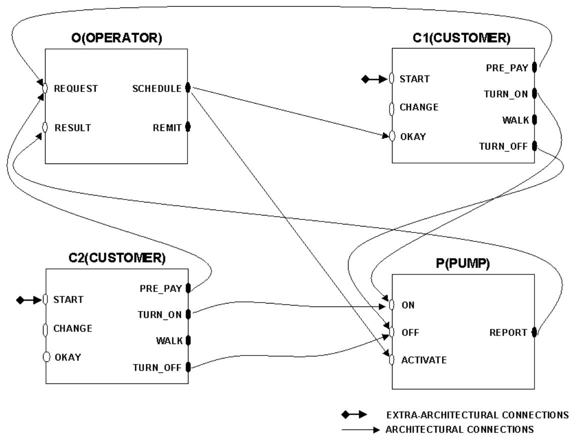

Figure 3: Intercomponent View of Gas Station Dependencies.

- *Affects chains*: consist of the set of components and/or their elements that could be affect by C. These are the elements that C affects.
- *Related-to chains*: consist of the set of components and/or their elements that may affect or be affected by an element of C. This chain is the combination of the affected-by and the affects chains for elements of C.

Different types of chains are created to answer different kinds of questions. For example, question 2 in Section 2 could be answered by constructing an affected-by chain based on certain structural relationships. As other examples, one could answer questions 5 and 9 by building affected-by chains, question 8 by building affects chains, and question 6 by building related-to chains. Depending on the type of change being made, question 7 could be answered by building any one of the chains.

Traditional approaches to architecture specification focus on the basic interconnection structure of components. By itself, the interconnection structure offers only a gross understanding of architectural dependencies. This is illustrated in Figure 3, which shows the four components of an example architecture and the dependencies among those components based on the specified connections among named ports. The example is the familiar gas station problem,[1] which we discuss in detail in Section 4.

The depiction in Figure 3 resembles the box and arrow diagrams traditionally used to describe software architectures. Only a very coarse-grained, conservative analysis can be performed when considering just this intercomponent view of an architecture, resulting in the capture of false dependencies. These false dependencies are due to the lack of information about how the interface behaviors of the components refine intercomponent interactions. In Figure 4 we show a more precise view of dependencies enhanced by an analysis of intracomponent behaviors. For example, the analysis reveals that an output through component `C1`'s port `Pre_Pay` implies an input through its port `Start` and not through its port `Okay`, a fact not discernable from Figure 3. There are also a number of critical anomalies evident only from the analysis result depicted in Figure 4. We discuss

[1]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 (e.g. [17, 20]). It has the advantage of being well known and compact, and does in fact exhibit features that would appear in a "real" architecture specification. In general, there appears to be a dearth of good architecture specification examples, both large and small.

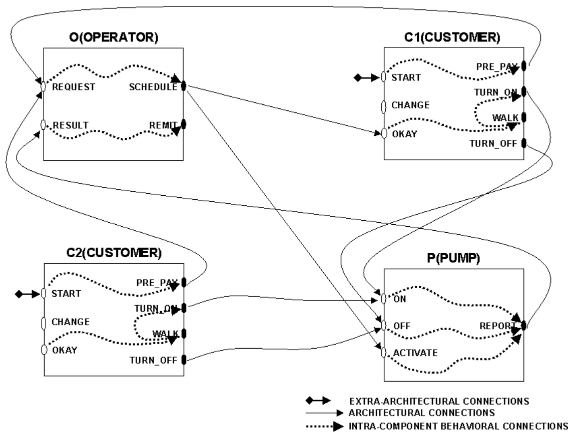

Figure 4: Enhanced View of Gas Station Dependencies.

these anomalies in Section 4. The ability to specify intracomponent interface behavior is a key distinguishing feature of modern-day ADLs.

Underlying a chain are, of course, specific dependence relationships. Traditionally, we think of dependence relationships at the level of program code, where control and data flow are the most prominent aspects to study. Dependence relationships at the architectural level arise from the connections among components and the constraints on their interactions. These relationships may involve some form of control or data flow, but more generally they involve *source structure* and *behavior*. Source structure (or structure, for short) has to do with static source specification dependencies, while behavior has to do with dynamic interaction dependencies. Below we give some examples.

Structural Relationships

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

Behavioral Relationships

- **Causal**—The behavior of one component implies the behavior of another component.
- **Input/Output**—A component requires/provides information from/to a second component.
- **Temporal**—The behavior of one component is specified to precede or follow the behavior of another component.
- **State-based**—A component cannot be in a particular state unless a second component is in a particular state.

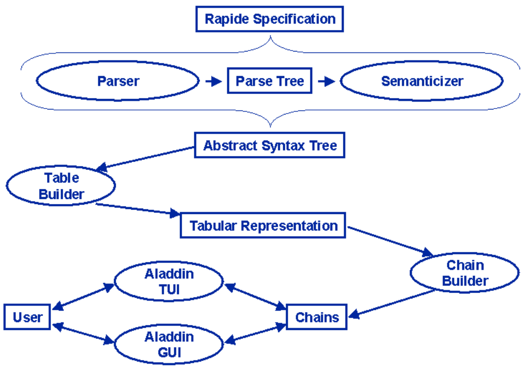

Figure 5: Aladdin Architecture.

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

The particular kinds of dependencies that can be established among components are heavily influenced by the primitive features of the ADL. For instance, in CHAM [14], only behavioral relationships are modeled, and those relationships are based on synchronous input and output ports through which data flow.[2] As mentioned in Section 1, Rapide specifications can involve both structural and behavioral relationships and typically involve event-based interactions.

4 Aladdin

Aladdin is a prototype tool for performing architecture-level dependence analysis. As shown in Figure 5, it takes as input an architectural specification and produces chains representing dependence relationships. Aladdin also provides a set of queries over the chains (through both a graphical and a textual user interface) that aid in answering questions such as those listed in Section 2. Currently Aladdin works only on Rapide specifications, but it has been carefully designed to separate language-specific processing tasks from language-independent analysis tasks. The boundary is at the so-called *table builder*, which translates an abstract syntax tree representation of an architectural specification into a common dependence table representation. In fact, the portion of Aladdin above the table builder in the figure was obtained from the Rapide research group at Stanford and directly incorporated into Aladdin. In the future, we expect to obtain other such ADL front ends for use with Aladdin.

The cells of the table built from the abstract syntax tree represent the set of relationships that exist between pairs of elements in the architecture. In general, for a given ADL, it is necessary to understand the various ways in which any two architectural elements can be related so that the dependence table can be constructed. This tabular representation could alternatively be viewed as a dependence graph, where multiple edges between node pairs represent the sets of relationships between row and column labels.

[2] Connectors can be introduced to model other kinds of behavioral relationships, but these are still based on, or built from, the primitive concept of the synchronous port.

Once the relationships are recorded in the table, a transitive closure over specific (but not necessarily homogeneous) dependencies for a particular element can be performed in order to construct a chain. For instance, in the gas station architecture mentioned above, chaining over the table allows the discovery of unused ports, specification coding errors, and investigation of how the replacement of a component would affect the rest of the system. Below, we detail our use of Aladdin in analyzing the Rapide specification for this architecture, which is shown in Figure 6. A close variant of this specification is available in the Rapide 1.0 Toolset distribution.

In this section, we begin by briefly reviewing the Rapide language so that our example can be understood. We then describe the general method for constructing the tabular representation, and follow this by describing the method for using the table to produce chains.

4.1 Overview of Rapide

Rapide is a powerful and complex language. For purposes of this paper, we present only a brief overview of the language, and restrict ourselves to the features used in our example. A full description of the language is available in the Rapide language manuals [27].

Components are defined in terms of their interfaces. Three types of components are described in Figure 6: a pump, a customer, and an operator. Interfaces are structured into separate sections that specify different aspects of the component's behavior and its interactions with other components. Two sections are of relevance here.

An *action* section contains the declaration of *in* and *out* actions, which specify the component's ability to observe or emit particular events. Implicitly declared actions represent events generated in the environment of the system that are emitted by or watched for in an interface. The event `start` in the first transition rule of the customer interface in Figure 6 is an example of an implicitly declared action.

A *behavior* section, which may contain local declarations, describes the computation performed by the component, including how the component reacts to in actions and generates out actions. Computations are defined in an event pattern language [28]. Patterns are sets of events together with their partial ordering, which is represented by a so-called *poset*. Posets are the basis for the dynamic analyses, such as simulation, provided with the Rapide tool set. Thus, they nicely complement the static analyses provided by Aladdin.

Component types are instantiated and then connected to form architectures. The *architecture* declaration at the bottom of Figure 6 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.

In our restricted use of Rapide, connections in an architecture may be made only between pattern lists and connection sets, where a pattern list is a list of one or more patterns to be observed and a connection set is either an expression or a pattern. Furthermore, the gas station example makes use of only the agent connection.

The behavior section 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 agents described above are also used as operators in the transition rules. The trigger may be a pattern or a boolean expression. The body may be a state assignment or it may generate a poset. Patterns of events may be watched for in the computation. When a pattern is observed, it triggers the events in the body of the rule. In the gas station example, the body of behaviors are either state assignments or the generation of a single event.

Rapide provides placeholders for use in patterns and expressions. These are designated with a "?". 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 as meaning "for each 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".

4.2 Construction of the Dependence Table

As mentioned above, Aladdin uses a table to represent the dependence relationships of an architecture. The relationships associated with connection and transition rule operators induce these dependence relationships. Figure 7 represents the dependence relationships of the gas station example. Notice that it captures in tabular form the relationships depicted in Figure 4.

A dependence table has m rows and n columns, where m is the number of communication ports in the architecture plus any implicitly declared interactions with the environment (e.g., the action `start`), and n is the number of communication ports in the architecture plus any emissions to the environment. The cells of the table contain the set of possible relationships between the pairs of architectural elements that define the column and row in which the cell resides. The columns in the table represent the dependent in the relationship and the rows represent the source (or object) of the dependence. For instance, if a is dependent on b, then the cell at column a and row b details that relationship. The cell may, for example, reflect the existence of a direct connection, it may indicate sharing of data, it may indicate interaction by means of event notification and subscription, or it may contain a combination of relationships between the source and dependent elements.

A table builder for a specific ADL must map the relationships that can be modeled in the language to the relationships described in Section 3 and recognized by the Aladdin chain builder. For example, the initial version of Aladdin maps dependencies induced by the agent connection to the causal relationship. In the case of the gas station architecture of Figure 6, 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 7.

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 actions in the trigger will be the cause of a given triggering. Thus, all are recorded as sources for the relationship. This is a conservative approach and we are investigating methods for reducing the generation of such false dependencies.

The local declaration of variables and actions in the behavior section of a component interface specification may be involved in the eventual connection of an incoming port to an outgoing port. This internally generated behavior is below the level of abstraction that is of interest to us for building chains, although it is necessary for Rapide's other, dynamic analyses. Aladdin applies a summarization algorithm in order to safely abstract away these behaviors. For instance, in the gas station example, the pump has several local declarations. Aladdin summarizes the effects of their inclusion, thus producing the intra-component behavioral connections shown in Figure 4.

```
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  O(), 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 6: Rapide Specification of the Gas Station Architecture.

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

Figure 7: Gas Station Dependence Table.

4.3 Using Aladdin

Figure 7 is the result of the Aladdin table builder processing the Rapide specification for the gas station example. Once this representation is available, chains can be built so that the Aladdin user can perform queries to examine properties of the specification. Affects chains involve creating links by beginning at the row labels and locating the related column label, whereas affected-by chains are constructed by linking 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. These relationships are transitive and, as mentioned above, we are assuming that all possible prior events occurred, so 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.

The Aladdin user interface can be used to access the following information about the various ports[3] appearing in a Rapide specification:

[3]Recall that "port" is used in this paper as a general term to refer to any means through which a component can communicate with other components in the system or with the environment of the system.

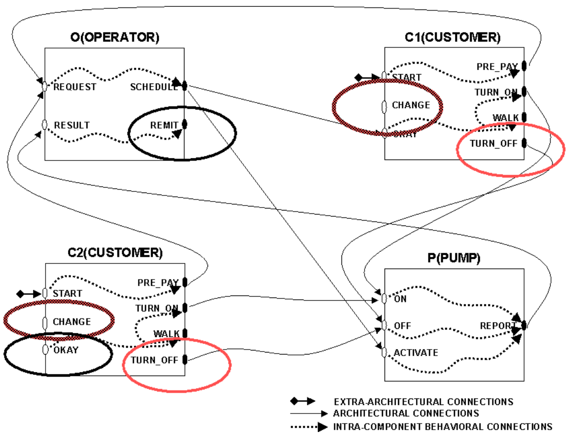

Figure 8: Gas Station Anomalies.

- 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. The engineer would use 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 shown in Figure 10. The chain shows 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 `C2.Okay` event does not lead to a legal start state for the system. Additionally, by comparing the two sub-chains, 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 6, 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.

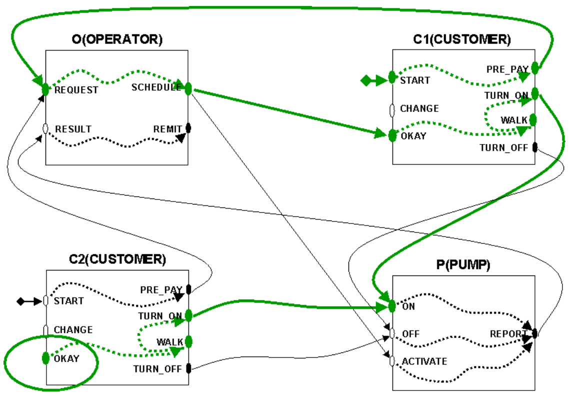

Figure 9: Gas Station Fault Localization.

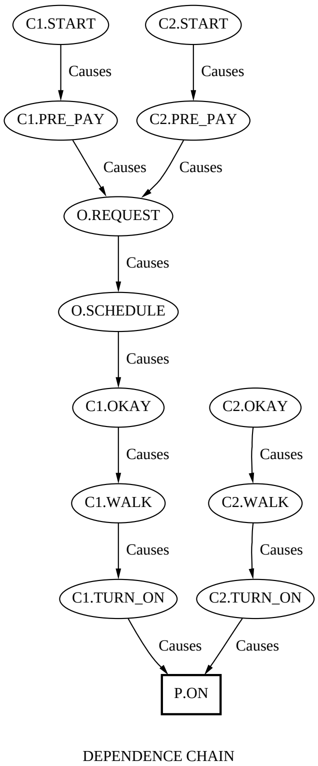

Figure 10: Aladdin-Generated Affected-By Chain.

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 help reveal significant faults in a system. One would expect additional errors to be introduced as the system is refined and implemented. But the early detection and correction of errors during the first stages of development produce overall savings in development costs.

5 Related Research

This work 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.

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

Considerable work has been done in the study and use of dependence relationships among variables and statements at the implementation level. For example, Ferrante, Ottenstein, and Warren [7] introduced the program dependence graph (PDG) for use in compiler optimization. Harrold and Soffa [11] have studied alias and interprocedural analysis of C and C++ programs.

Representation schemes for program/system dependencies that are similar to our tabular representation have been used in program optimization and for system requirements analysis. Aho, Sethi, and Ullman [1] describe a program representation where bit vectors are used to compactly represent "gen" and "kill" sets for each statement in the program. Logical operations are performed on these bit sets to determine statement dependencies. Grady [10] uses an N-square representation to examine system coupling when assigning functionalities to components during initial system decomposition. Pomakis and Atlee [23] use a tabular notation similar to the one found in SCR [12] to specify feature behaviors. This is used in conjunction with a graphical representation to study possible feature interactions. While all of these methods use a representation similar to our tabular representation, the analyses applied to the representations are different and are for different purposes.

Our notion of 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 [30] as an aid to program debugging. Sloane and Holdsworth [26] suggest generalizing the concept of program slicing and show the potential in syntax-based slicing of non-imperative programs. We agree with the spirit of this work and are pursuing a similar goal, but in the particular context of software architectures.

Chang and Richardson [5] introduced techniques for creating dynamic specification slices. This approach uses traditional slicing criteria, whereas our work involves exploring relationships at the architectural level, where additional criteria are defined.

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

Naumovich et al. [20] apply INCA and FLAVERS, two static concurrency analysis tools used for proving behavioral properties of concurrent programs, to an Ada translation of a description of the gas station problem that was written in the Wright ADL. Their approach is to create a concurrent program that can simulate the intended concurrent behavior of the system. Our work is aimed at developing general dependence analysis techniques that may contribute to the enhancement of the analyses already provided by these tools.

6 Conclusion

The main contribution of this paper is the introduction of architecture-level dependence analysis for both the structural and behavioral aspects of a system.

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

Historically, testing has concentrated on the implementation of the system, which has meant that it is considered fairly late in the development process. Eventually, we intend to incorporate chaining into a complete life cycle software analysis and testing environment, such as the TAOS environment [24]. TAOS provides dependence analysis and testing at the implementation level, with some support for specification-based test case generation and result checking. Integrating architecture analysis techniques such as chaining would round out the life cycle support for analysis and testing.

REFERENCES

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

[2] F.E. Allen. Control Flow Analysis. *SIGPLAN Notices*, pages 1–19, July 1970.

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

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

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

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

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

[8] D. Garlan, R. Monroe, and D. Wile. ACME: An Architecture Description Interchange Language. In *Proceedings of CASCON '97*, pages 169–183. IBM Center for Advanced Studies, November 1997.

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

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

[11] M.J. Harrold and M.L. Soffa. Efficient Computation of Interprocedural Definition - Use Chains. *ACM Transactions on Programming Languages and Systems*, 16(2):175–204, March 1994.

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

[13] S. Horwitz, T. Reps, and D. Binkley. Interprocedural Slicing Using Dependence Graphs. *ACM Transactions on Programming Languages and Systems*, 22(1):26–60, January 1990.

[14] P. Inverardi and A.L. Wolf. Formal Specification and Analysis of Software Architectures using the Chemical Abstract Machine Model. *IEEE Transactions on Software Engineering*, 21(4):373–386, April 1995.

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

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

[17] 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.

[18] 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.

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

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

[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 Dependencies and its Implications for Software Testing, Debugging, and Maintenance. *IEEE Transactions on Software Engineering*, 16(9):965–979, September 1990.

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

[24] 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.

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

[26] 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.

[27] RAPIDE Design Team. Draft: Guide to the Rapide 1.0 Language Reference Manuals. July 1997.

[28] RAPIDE Design Team. Draft: Rapide 1.0 Pattern Language Reference Manual. July 1997.

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

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

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

[32] J. Zhao. Using Dependence Analysis to Support Software Architecture Understanding. *New Technologies on Computer Software*, pages 135–142, September 1997.