Chaining: A Software Architecture Dependence Analysis Technique

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-845-97 September, 1997

ABSTRACT

The emergence of formal architecture description languages provides an opportunity to perform analyses at high levels of abstraction. 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 describe chaining*, a technique similar in concept and application to program slicing, in which the goal is to reduce the portions of an architecture that must be examined by an architect for some purpose, such as testing or debugging. In chaining, links represent the dependence relationships that exist in an architectural specification. Links connect elements of the specification that are directly related, producing a chain of dependencies that can be followed during analysis. We illustrate the utility of chaining 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 architectures 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 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 [2, 7, 9, 13].

In general, there are many kinds of questions one might want to ask at an architectural level for purposes as varied as reuse, reengineering, fault localization, impact analysis, regression testing, and even 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 know 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.

In this paper, we introduce *chaining*, a dependence analysis technique we are developing for software architectures. In chaining, links represent the dependence relationships that exist in an architectural specification. Links connect elements of the specification that are directly related, producing a chain of dependencies that can be followed during analysis. The traditional view of dependence analysis is based on control and data flow relationships associated with functions and variables [1, 4, 5, 12, 15, 16]. We take a broader view of dependence relationships that is more appropriate to the concerns of architectures and their attention to component interactions. In particular, we look at both the structural and the behavioral relationships among components expressed in current-day formal architecture description languages, such as Rapide [8] and Wright [2].

We begin with a brief review of relevant questions that might reasonably be asked at the architectural level as a way to further motivate our investigation of static 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, including the representation scheme we employ for recording dependence relationships. For purposes of illustration, the discussion is cast in terms of a particular architecture description language, namely Rapide. We then review related work and conclude with a discussion of our future plans.

2 MOTIVATING QUESTIONS

As mentioned above, there are a variety of questions that should be answerable by an examination of a formal architecture description. Here we list several such questions as motivation for the study of architectural dependence analysis.

1. Which components could have contributed to the particular state of a component?
2. Which components make use of this particular state of a component?
3. Which components could have been used if this state of the system is reached?
4. If this component is to be reused in another system, which other components of the system are also required?
5. Which components of the system contribute to this piece of functionality?
6. What are the potential effects of dynamically replacing this component?
7. If this component uses a blackboard connector, what other components will it be communicating with?
8. If a change is made to this component, what other components might be affected?
9. 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?

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. The idea of collecting together related portions of a system, in an effort to reduce the amount of information that must be examined for some purpose, recalls the notion of program slicing introduced by Weiser [22]. Sloane and Holdsworth [19] suggest generalizing the concept of program slicing and show the potential in slicing non-imperative programs.

Our notion of chaining is intended to support a range of analysis applications, including what would amount to an architectural slice. Unlike previous approaches to slicing, which are based on statements and variables, the basis for architectural slices is more general architectural relationships, as discussed in the next section.

3 CHAINING

Chaining is a dependence analysis technique we propose for use at the architectural level. In chaining, links represent the dependence relationships that exist in an architectural specification. Links connect elements of the specification that are directly related, producing a chain of dependencies that can be followed for further analysis.

For the purposes of this paper, we offer the following definition of a software architecture: A *software architecture* is a set of components and the connections among them.

A *component* is comprised of four types of *elements*

- *Component interaction elements*: provide the means for connection to or interaction with another component. These elements may also be called ports.
- *Data elements*: contain the data to be transformed by, provided by or used by the component.
- *Processing (or behavioral) elements*: provide the means for transforming or transmitting the data and/or state of the component.

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

Given component C that contains elements we describe three types of chains:

- *Affected-by* chains: consist of the set of components and/or their elements that could potentially affect the behavior of a processing element, the state of a state element, and/or the value of a data element of C.
- *Affects* chains: consist of the set of components and/or the processing elements, state elements, and data elements that could be impacted by a change of C.
- *Related* chains: consist of the set of components and/or their elements that may affect or be affected-by the behavior of a processing element, the state of a state element, and/or the value of a data element of C; this chain is the combination of the affected-by and the affects chain for elements of C.

As examples, an affected-by chain would contribute to answering question 1 from above, while an affects chain would contribute to answering question 8. A related chain would contribute to answering question 6.

Underlying chains are dependence relationships between components. 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 between 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
  - *Includes:* The specification for a component may be created from numerous source modules that are textually combined.
  - *Import/Export:* The specification for a component may describe the information exported and imported between source modules (e.g., with a module interconnection language).
  - *Inheritance:* The specification for a component may be created through inheritance from other source modules.
- Behavioral Relationships
  - *Temporal:* The behavior of one component precedes or follows the behavior of another component.
  - *Causal:* The behavior of one component implies the behavior of another component.
  - *Input:* A component requires information or stimulation from another component.
  - *Output:* A component provides information or stimulation to another component.

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. The behavioral dependencies (of which control and data flow are examples) allow one to relate states or interactions to other states or interactions.

The particular kinds of dependencies and chains that can be established among components are heavily influenced by the primitive features of the architecture description language. For instance, in CHAM [6] and Wright, only behavioral relationships are modeled, and those relationships are based on synchronous input and output ports[1] through which data flow. Darwin [9] and Rapide specifications can involve both structural and behavioral relationships. The structural relationships in these languages derive from concepts of import, export, and inheritance. Rapide behavioral specifications are particularly interesting in that they can involve event-based interactions, as we illustrate in the next section.

4 CREATING AND USING CHAINS

As mentioned above, chains represent dependence relationships in an architectural specification that can be followed during analysis. Individual links within a chain associate components and/or component elements of an architecture that are directly related, while a chain of dependencies associates components and/or component elements that are indirectly related. We develop a matrix representation for architectures and apply a straightforward algorithm to the matrix in order to discover chains of related component elements.

We have used chaining to discover an intentionally introduced fault in the ADAGE avionics architecture, which is written in Rapide [10]. The ADAGE example is a two-level nested architecture consisting of an outer-level architecture plus four sub-architectures. The four sub-architectures together contain 30 components communicating through nearly 100 ports. It is, therefore, too large to be presented here. Instead, we illustrate our technique using another Rapide architecture specification, the well-known gas station example.[2] The Rapide code for the gas station, which is shown in Figure 2, is taken from a teaching example available in the Rapide 1.0 Toolset. This example unfortunately does not involve structural relationships, but is sufficient to give the flavor of our approach.

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

4.1 Overview of Rapide

Rapide is intended to aid in the development and maintenance of large systems. Rapide itself is very powerful and complex. For the purposes of this paper, we present only a brief overview of

[1] In both languages, 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.

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

```
type FACE is interface
provides
     ...
requires
     ...
service
     ...
action
     in  I() ;
     out O() ;
behavior
     X : integer;
begin
     I ||> O;
constraint
     ...
end FACE
```

```
architecture ARCH return sometype is
      I1, I2:  FACE;
connect
      ...
constraint
      ...
end ARCH
```

Figure 1: Skeleton Rapide Interface Architecture.

the language features used in our example. A full description of the language is available in the Rapide Language Manuals [20].

The architecture description for the gas station is provided in the July 1997 release of the Rapide Toolset. There are seven versions of the example included in the release. We are using the second version.

Within this paper, we are limiting our view of Rapide to the following features and concepts: architectures, computations, connections, events, interfaces, `in` and `out` actions, posets, placeholders, and triggers.

Components are defined in interfaces. The interfaces are comprised of `provides`, `requires`, `service`, and `action` declaration sections, a `behavior` section that may contain local declarations, and a `constraint` section. `action` sections contain the declaration of `in` and `out` actions, which declare the ability to observe or emit particular events. Figure 1 shows the skeleton of an interface, FACE, and an architecture, ARCH, where `I` and `O` are declared as actions. Not all architectures and interfaces contain all of the possible sections. In fact, the gas station example does not have `provides` , `requires`, or `service` sections in any of its interfaces, nor are there any `constraint` sections.

For our purposes, we are interested in the behavioral view that is provided at the architecture level. Computations are defined in a pattern language [21]. Patterns are sets of events together with their partial ordering. The representation for the partial orders are called *posets*. Posets can be used to evaluate the correctness of the architecture.

Rapide uses four symbols to represent connections in connection rules. Connection rules have a trigger, an operator and a body.

- *Agent* connection ("||>"): The observation of the pattern described in the trigger asynchronously generates the events in the body.
- *Basic* connection ("`to`"): A basic connection connects two functions, objects, or actions.

The connection will be made between a `provides` or `in` element of a component and and a `requires` or `out` element of the other component. Basic connections provide a means of synchronous communication in Rapide.

- *Pipe* connection ("=>"): The observation of the pattern described in the trigger asynchronously generates the events in the body. The difference between a pipe and an agent is that a pipe creates dependencies between all events generated in prior triggerings of the rule.

- *Service* connection ("`to`"): Service connections are a special case of basic connections which connect a service and a dual of that service. Services provide a means of bundling connectors together. A dual of a service has the reverse of the `provides`, `requires`, `in`, and `out` actions from the service. Service connections provide a means of synchronous communication.

Pipes and agents are also used as operators in the transition rules defined in the `behavior` section of an interface. A transition rule is composed of a trigger, an operator, and a body. The trigger may be a pattern or a boolean expression. The body may be a state assignment or a poset generator. An event or set of events generated during a simulation is added to the computation of that simulation. Patterns of events may be watched for in a poset. 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.

In our restricted use of Rapide, connections in an architecture may be made 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.

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 every customer, there is to be a pipe 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 Matrix

As mentioned above, we use a matrix to represent the dependence relationships of an architecture. The matrix is $\mathbf{m} \times \mathbf{n}$, where $\mathbf{m}$ is the number of ports in the architecture plus any implicitly declared actions, and $\mathbf{n}$ is simply the number of ports in the architecture. Implicitly declared actions represent events generated in the environment of the system that are watched for within an interface. The `start` event in the first transition rule of the customer interface of the gas station specification in Figure 3 is an example of an implicitly declared action.

The relationships associated with connections and state transition rule operators in an architecture educe a dependence relationship. In the dependence matrix, the columns 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

```
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 2: A Rapide Interface Architecture.

cell may, for example, reflect the existence of a direct connection, such as a remote function call, it may indicate sharing of data, or it may indicate interaction by means of event notification and subscription. In general, for a given architecture description language, it is necessary to understand the various ways in which two components and their elements can be related so that the dependence matrix can be constructed.

For example, one could identify the following direct dependence relationships derived from Rapide transition rule and connection operators:

- *Agent* ("||>"): The element(s) of the body are dependent on the element(s) of the trigger.
- *Basic* ("`to`"): The `requires` or `in` element is dependent on the `provides` or `out` element of the connection .
- *Pipe* ("=>"): The element(s) of the body are dependent on the element(s) of the trigger and the element(s) of the trigger are dependent on the element(s) of the body in order to capture possible dependencies on events generated by prior triggerings of the rule.
- *Service* ("`to`"): A service connection creates two ("symmetric") dependencies. Each member of the relationship must be recognized as both the source and the dependent in the relationship because a service can contain a combination of the `provides`, `requires`, `in` and `out` port types.

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 recorded in the matrix cells (C1.T_On,P.On) and (C2.T_On,P.On) in Figure 3.

The method for constructing the matrix is a two step process:

- STEP 1: Build the matrix frame. Determine and record all the communication ports for the architectural components and use these as the column and row labels of the matrix.
- STEP 2: Record dependencies in the cells. Mark the appropriate cell of the matrix for each connection and transition rule in the architecture and transition rule in component interfaces.

For the gas station example, we took these steps in building the matrix shown in Figure 3.

STEP 1.

As stated above, the first step is to build the matrix frame. Ports in Rapide are defined in the `provides`, `requires`, `service` and `action` sections of component interfaces. We construct the matrix by first identifying the components of the system, and then identifying the ports within their interfaces.

A. The gas station architecture is found at the bottom of Figure 2. The components of the architecture, which are listed in its top section, are an operator: `O`, a pump: `P`, and two customers: `C1` and `C2`.

B. The next stage of Step 1 is to check the `provides`, `requires`, `action`, and `service` sections of the interfaces for each component to determine what its ports are. In the gas station architecture, there are no `provides`, `requires`, or `service` sections in any of the interfaces. Each interface defines all its ports in its `action` section. The `Operator` interface watches for `Request` and `Result` events and outputs `Schedule` and `Remit` events. These are listed as column and row labels of the matrix shown in Figure 3.

C. It is possible for implicitly declared events to be watched for by a component. These events would be evident because they would appear in a trigger but would not be defined in the component interface. In our gas station example, the `start` event of the `Customer` interface is an implicitly declared event and as such will be included as a row label.

STEP 2.

After the frame of the matrix has been built, the connections and transitions are recorded in the matrix cells.

A. The first stage of Step 2 is to record the connections defined in the architecture interface, which is done by looking at the `connect` section of the architecture at the bottom of Figure 2. The first connection recorded for this example is the agent connection between each customer's `Pre_Pay` action and the operator's `Request` action. The use of a placeholder for the customer tells us that the connection is to be made for each component of type `Customer` in the architecture. This connection is recorded in the matrix in Figure 3 as a "||>" in the (C1.PP,O.Req) and (C2.PP,O.Req) cells. The rest of the connections are recorded in a similar manner.

B. Next, the transitions defined in the behavioral part of the component interfaces are recorded, which is done by looking at the `behavior` section of each interface. The relationship between the actions in the trigger and the body of the transition rules are recorded in the appropriate cell of the matrix. Some notable features of the behavior in the gas station interfaces are:

   - `start` appears as the trigger of the Customer `Pre_Pay` action. `start` is an implicitly declared event that is the beginning of a given execution of the system.
   - In the `Customer` interface, `Walk` is declared to be an `out` action. It is used both in the body of a transition rule and as a trigger. The appearance of an `out` action in a trigger indicates that the body will only be triggered by locally generated `Walk` events.
   - The interface of the pump contains local declarations. The local variables and actions are used to define the internal behavior that would have to occur between the observation of the `On` $\sim$ `Activate` pattern in the computation and the, locally visible, emission of the `Done` event, which could trigger the `Report` event. Because we restrict our view of behavior to the architectural level, a summary of the internal behavior of the pump provides us with the dependence of `Report` on `On` and `Activate`, and conversely, since the connection is a pipe, the dependence of `On` and `Activate` on `Report`.[3] These relationships

[3] If, in fact, we chose to model the internal behavior, these internal events could be included in the matrix frame and their behavior modeled in the same way as that at the architectural level. The summarization at the architectural level produces a direct dependence where there would otherwise be an indirect dependence.

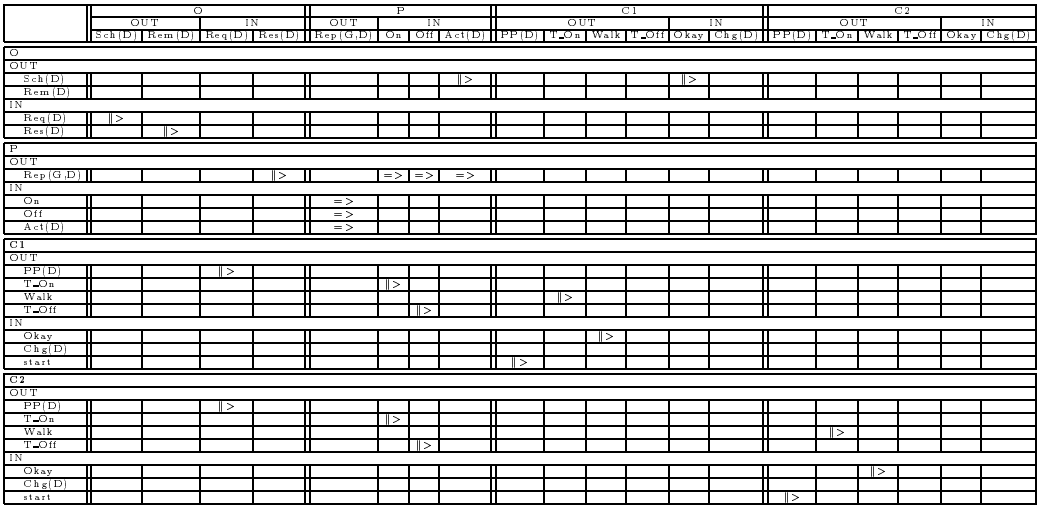

Figure 3: Gas Station Dependence Matrix.

are recorded in the appropriate cells of the matrix with a "=>", since the operator is a pipe.

While composite patterns in triggers and bodies of rules can be complex, they do not need to add complexity to the matrix representation. This is because, in general, there is no way to tell, statically, which of the actions in the trigger are 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 false dependencies.

Once the system's relationships are recorded in the matrix, one can construct chains to answer questions about the represented architecture. 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.

In the gas station matrix, construction of an affected-by chain of events that may have caused the `P.Activate` event, begins at the columns and look 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 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 may only be preceded by the `start` event of the `Customer` interface.

For example, 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.

Chaining can be used to answer questions about an architecture. The types of chains constructed vary depending upon the question being asked. For instance, affects chains answer questions about which events cause other events, while affected-by chains answer the opposite. Some interesting questions and their chain-based answers are:

1. *What components would need to be retested if we were to decide to replace the pump?* In this case we are interested in the components that could receive information from the pump, thus we want to construct an affects chain. We begin by looking at the rows associated with the `out` ports of the pump and chain from there as follows:

   (a) P.Report $\Longrightarrow$[4] O.Result
   (b) O.Result $\Longrightarrow$ O.Remit
   (c) O.Remit $\Longrightarrow$

   The `O.Remit` action does not appear in a trigger therefore, this is the last link of this chain. For this architecture, the pump and the operator need to be retested. Of course, one might wonder about the correctness of the architecture when it contains `out` actions that are not subscribed for, so in asking this question we have uncovered a possible fault in the architecture.

2. *Are there any actions for which there are no subscriptions?* In this case we look for blank rows in the matrix. If there are no entries in a row, no components watch for that action. In the gas station architecture, there are no subscribers for the `C1.Change`, `C2.Change`, or `O.Remit` actions. Taking a quick check of the `connect` section of the gas station architecture reveals the fact that `O.Remit` is not used as a trigger in any connection. This connection is included in other versions of the architecture provided with the Rapide 1.0 Toolset.

   If the blank row is an `in` action then it is probably an anomaly. If it is an `out` action, it may be a case of planning for the future when a component may want to watch for this event. In any case, it is good to be able to consider the ramifications of an event being made available and going unused.

3. *What series of events could contribute to the Turn_On event for C2?* This architecture contains a serious error. In particular, it is never possible for the second customer to pump gas. The first customer does not suffer from this dilemma. Thus, one could compare the affects chains that lead to the `C1.Turn_On` and `C2.Turn_On` events to find where they differ.

   (a) C1 affects chain
      i. C1.Turn_On $\Longleftarrow$ C1.Walk
      ii. C1.Walk $\Longleftarrow$ C1.Okay
      iii. C1.Okay $\Longleftarrow$ O.Schedule
      iv. O.Schedule $\Longleftarrow$ O.Request
      v. O.Request $\Longleftarrow$ (C1.Pre_Pay or C2.Pre_Pay)
   (b) C2 affects chain
      i. C2.Turn_On $\Longleftarrow$ C2.Walk
      ii. C2.Walk $\Longleftarrow$ C2.Okay
      iii. C2.Okay $\Longleftarrow$

[4] $\Longrightarrow$ may be read as "causes"

We can see from comparing the chains, that the problem is that the `C2.Okay` event never is generated, but the `C1.Okay` event is generated no matter who paid for the gas.

With this information at hand, we examine the architecture 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 system's architect that there is a fault in the connection based on the `O.Schedule` action in the architecture.

5 RELATED WORK

To our knowledge, the work described in this paper is the first attempt to define dependence analysis of specifications written in current-day architecture description languages. Nevertheless, the work builds on previous and related work in three primary areas: traditional dependence analysis techniques; novel approaches to slicing; and applications of static concurrency analysis tools to architecture descriptions. In this section, we review representative work in these areas.

Various tools have been developed to trace the structural dependencies within program code. An example is the tool *makedepend*, which examines code to automatically derive the file dependencies (e.g., `#include` in the C environment) used in Make files. Our approach applies this concept to architecture description languages and combines the information with behavioral dependencies.

ProDAG [18] is a program dependence analysis toolset that performs statement-level dependence analysis. ProDAG allows one to create and access various predefined relationships originally identified by Podgurski and Clarke [16]. The technique of chaining raises these ideas to the architectural level, as well as incorporating the notion of structural dependence.

Sloane and Holdsworth [19] suggest advanced applications for Weiser's concept of program slicing [22], in which the basis for analysis includes aspects other than traditional data and control flow. They describe a generalized slicing tool that treats slicing as tree-marking manipulations of program syntax trees. Their concept of syntactically based generalized slicing allows them to contemplate the slicing of non-imperative programs. They demonstrate this by describing several tools that could be built to aid in the understanding of formal compiler specifications. We agree with the spirit of this work and, in some sense, are pursuing a similar goal, but in the particular context of software architectures.

Oda and Araki [14] first introduced the concept of static specification slicing for specifications written in Z. Chang and Richardson [3] extend this work with the introduction of techniques for creating dynamic slices. In both of these efforts, the value of a variable in a particular predicate is used as the slicing criteria, whereas we are exploring relationships at the architectural level, where the concept of a variable does not exist.

Zhao, Cheng, and Ushijima [23] propose the system dependence net (SDN) as a representation of concurrent object-oriented programs. The SDN is used to find slices of CC++ (Concurrent C++) programs, using the value of a variable or return value of a method call at a particular statement as the slicing criteria. They point out that new types of dependence relationships should be considered when slicing concurrent object-oriented programs. This is similar to our recognition that there are new types of dependence relationships to consider in analyzing architecture descriptions. However, they restrict slicing to the statement level, using the limited criteria of variable and method call return values.

Naumovich et al. [13] apply INCA and FLAVERS, two static concurrency analysis tools used for proving behavioral properties of concurrent programs, to an Ada translation of the Wright description of the gas station problem. The focus of this work is on demonstrating that existing static concurrency analysis techniques can be applied to software architectures to help prove, or disprove, the satisfaction of certain behavioral properties. Their approach, however, amounts to creating 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, in fact, contribute to the enhancement of the static analyses already provided by these tools.

6 CONCLUSION

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

Our dependence matrix representation for architectures is general enough to support analysis capabilities beyond what is described in this paper. For instance, it can provide a view of dependence relationships at a higher level of abstraction than that presented here by determining dependences between the components themselves, thus considering the components as the labels on the rows and columns of the matrix. Moreover, although here we describe chaining for a subset of Rapide, our approach can be easily adapted to many other architecture description languages. This is because the application of chaining is not language dependent, although the construction of the dependence matrix is somewhat determined by the constructs of the language. This is similar to traditional program dependence analysis, where the representation of the program and the def-use annotations depends on the programming language, but the dependence analysis itself does not. Thus, we intend to incorporate more features of the Rapide language, to extend our definition of chaining to other languages (including CHAM [6], C2 [11], Darwin [9], and Wright [2]), and to test our approach against more complex architectures.

We are interested in investigating heuristics for reducing the conservativeness of chaining through patterns. In Rapide specifications, for example, there is information available such as which combination of placeholders is required by a dependent, that could be used to reduce the numbers of actions to be linked into a chain. We are also continuing to investigate the summarization of internal component behavior. We are debating the benefits of incorporating these local actions into the matrix or summarizing their effects.

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 [17]. TAOS includes dependence analysis and testing at the implementation level, but also has support for using specifications in the testing process for test generation and result checking. Integrating architecuture analysis techniques, such as chaining and related techniques, would round out the life cycle support for analysis and testing.

Given the similarity between chaining and traditional program slicing, we expect comparisons to be made between the two. There is growing skepticism in the programming languages and software engineering research communities as to the value of slicing software systems, since slices have not been shown to result in significant reductions in program size. It is our belief that the savings from applying such techniques will be greatest when applied to large systems, since these systems should be more loosely coupled. But computing statement level slices for large systems is

likely to be impractical. Architectural level slicing may prove a practical alternative. Moreover, architectural level analysis offers the possibility to detect faults early in the software life cycle.

REFERENCES

[1] H. Agrawal, R. A. DeMillo, and E. H. Spafford. An Execution-Backtracking Approach to Debugging. *IEEE Software*, pages 21–26, May 1991.

[2] R. Allen and D. Garlan. Formalizing Architectural Connection. In *Proceedings of the 16th International Conference on Software Engineering*, pages 71–80. IEEE Computer Society, May 1994.

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

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

[5] 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.

[6] 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.

[7] 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*. Springer-Verlag, 1997. To appear.

[8] 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.

[9] 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.

[10] W. Mann, F. C. Belz, and P. Corneil. A Rapide-1.0 Definition of the ADAGE Avionics System. Technical Report CSL-TR-93-585, Stanford University, 1993.

[11] N. Medvidovic, R. N. Taylor, and Jr. E. J. Whitehead. Formal Modeling of Software Architectures at Multiple Levels of Abstraction. In *Proceedings of the California Software Symposium 1996*, pages 28–40, Los Angeles, CA, April 1996.

[12] C.T. Moore, T.O. O'Malley, D.J. Richardson, S.H.L. Aha, and D.A. Brodbeck. ProDAG: A Program Dependence Graph System. Technical report, Department of Information and Computer Science, University of California at Irvine, 1990.

[13] G. Naumovich, G.S. Avrunin, L.A. Clarke, and L.J. Osterweil. Applying Static Analysis to Software Architectures. In *Proceedings of the Sixth European Software Engineering Conference*. Springer-Verlag, 1997. To appear.

[14] T. Oda and K. Araki. Specification Slicing in Formal Methods of Software Development. In *Proceedings of the Seventeenth Annual International Computer Software and Applications Conference*, pages 313–319. IEEE Computer Society Press, November 1993.

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] 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.

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

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

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

[23] J. Zhao, J. Cheng, and K. Ushijima. Static Slicing of Concurrent Object-Oriented Programs. In *IEEE-CS 20th Annual International Computer Software and Applications Conference*, pages 312–320, 1996.