The Application of Dependence Analysis to Software Architecture Descriptions

Judith A. Stafford[1], Alexander L. Wolf[2], and Mauro Caporuscio[3]

[1] Department of Computer Science
Tufts University
Medford, MA 02155 USA
jas@cs.tufts.edu
[2] Department of Computer Science
University of Colorado
Boulder, Colorado 80309-0430 USA
alw@cs.colorado.edu
[3] Dipartimento di Informatica
Università dell'Aquila
I-67010 L'Aquila, Italy
caporusc@univaq.it

Abstract. As the focus of software design shifts increasingly toward the architectural level, so too are its analysis techniques. Dependence analysis is one such technique that shows promise at this level. In this paper we briefly describe and illustrate the application of dependence analysis to architectural descriptions of software systems.

1 Introduction

Traditionally, software architectures are described using informal, natural-language documents. Box and arrow diagrams are often used to bring more precision to the descriptions, but while they can reveal some ambiguous and missing properties, they are not capable of modeling all the information provided in the natural-language specification, such as system behavior. Formalization, as applied to software development at the architectural level, involves the application of mathematically based modeling languages to capture structural and behavioral properties of the components of a system. Above all, these languages provide support for rigorous analysis of a system early in the life cycle and/or at high levels of abstraction. Additionally, a formally described software architecture can serve as a vehicle for precise and unambiguous communication among the stakeholders in a system, and can provide a means to accurately capture domain-specific properties in ways that support domain-specific architectural generalizations.

The goal of formally describing and analyzing the structure and behavior of a software system is not new. Formal approaches have been proposed and used in various phases of software development and maintenance for as long as people have recognized the challenges of software engineering. Formal design notations

M. Bernardo and P. Inverardi (Eds.): SFM 2003, LNCS 2804, pp. 52–62, 2003.

and their associated analyses, in particular, were a major focus of research in the 1970s and early 1980s. Results ranged from techniques for describing and analyzing module interconnection, which were intended to address static properties of component structure and import/export relationships, to techniques for describing and analyzing concurrent processes, which were intended to address dynamic properties of component interaction behavior.

Software architecture is but the latest framework within which researchers are trying to attain the goal of formal system description and analysis. Its emphasis is on unifying and extending earlier techniques for description and analysis, and in applying the resulting new techniques in the context of modern-day software practice. Unification is coming about from considering how the component structure of a system can be used to modularize the description and analysis of behavioral properties such that those descriptions and analyses can be performed in a more tractable, compositional manner. Extensions are being explored that are enhancing the typing of components and their interfaces to account for dynamic interaction behaviors. And, finally, the application of formal approaches is benefiting from the rapidly growing industry interest in system development based on large-grain component assembly rather than on small-grain component programming.

2 Formal Architectural Analysis

Research in architectural analysis centers on determining which specific properties are appropriate for this level of analysis, and on developing techniques to carry out those analyses. The premise underlying this work is that the confidence gained through analysis at an architectural level will translate into confidence in other levels of the system.

Many techniques for analyzing software systems have been developed over the past decades. Most, however, are ineffective for analyzing large systems. This is particularly true for techniques aimed at analyzing concurrent systems, where state explosion problems are especially acute. To make techniques for these situations more tractable, traditional specification and analysis techniques have been enhanced in a variety of ways. Software architecture can be seen as another approach to attacking the problem by providing a particular method for abstraction and modularization.

Automated analysis techniques can differ in the levels of assurance they provide. In general, the techniques trade off efficiency and tractability against precision and completeness. For instance, it may be possible to guarantee some properties only under certain assumptions or conditions. Carefully chosen, those assumptions and conditions can match well with the context in which the system is anticipated to operate, and thus the analysis can provide useful information.

A desirable characteristic of any imprecise or incomplete analysis technique used to examine a property is that it give no false positive results concerning that property. In other words, it should never indicate the absence of a problem when, in fact, there is a problem. On the other hand, it is reasonable to allow a

technique to indicate the possible presence of a problem, even if none truly exists, and defer further analysis to some other automated analysis technique or to the human. This characteristic is commonly referred to as *conservatism*. Clearly, the most conservative analysis technique is one that indicates the possible presence of an error in all situations. Such an absurd technique, while highly efficient (it can be implemented using a constant function), is not of use. One goal of analysis research is to increase the precision of conservative techniques such that they are both efficient and useful.

3 Dependence Analysis

Dependence analysis involves the identification of interdependent elements of a system. It is referred to as a "reduction" technique, since the interdependent elements induced by a given inter-element relationship forms a subset of the system. It has been widely studied for purposes such as code restructuring during optimization, automatic program parallelization, test-case generation, and debugging. Dependencies can be identified based on syntactic information readily available in a formal specification. This type of analysis generally ignores state information, but may incorporate some knowledge of the semantics of a language to improve the precision of the results [6].

Dependence analysis as applied to program code is based on the relationships among statements and variables in a program. Techniques for identifying and exploiting dependence relations at the architectural level have also been developed [8,13,14,15]. 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 involve source structure and behavior. Source structure (or structure, for short) has to do with system dependencies such as "imports", while behavior has to do with dynamic interaction dependencies such as "causes". Structural dependencies allow one to locate source specifications that contribute to the description of some state or interaction. Behavioral dependencies allow one to relate states or interactions to other states or interactions. Both structural and behavioral dependencies are important to capture and understand when analyzing an architecture.

4 Example: Aladdin

Aladdin [9] is a tool that identifies dependencies in software architectures. It was designed to be easily adapted for use with a variety of architectural description languages and has been demonstrated on the languages Acme [4] and Rapide [10].

If one thinks of an architectural description as a set of boxes and arrows in a diagram, where the arrows represent the ability for a box, or some port into or out of that box, to communicate with another box in the diagram, then one can think about Aladdin as walking forwards or backwards from a given box, traversing arrows either from heads to tails or vice versa. In Aladdin, the arrows

are called *links* and the process of walking (i.e., performing a transitive closure) over the links is called *chaining*.

If there is no knowledge about how a box's input ports behaviorally relate to it output ports, then a forward (backward) walk must include leaps from each input (output) port that is reached to all output (input) ports. In that case, the analysis is essentially being performed in a conservative manor at the component level, which can lead to a high degree of false dependencies. If, instead, the designer makes a precise statement about how input and output ports are related, presumably using an appropriately rich architecture description language, then Aladdin can take advantage of this information to produce a more precise reduction set.

The behavioral relationship among the input and output ports of a component define the interaction behavior of that component. It is important to note that the interaction behavior is not intended to capture the functional behavior of the component. For example, the description of how a server interacts with its clients is independent of the computation carried out by the server on behalf of its clients. Aladdin uses a summarization algorithm operating on the description of a component's interaction behavior to identify possible relationships between pairs of input and output ports. The resulting connections are called *transitional connections*.

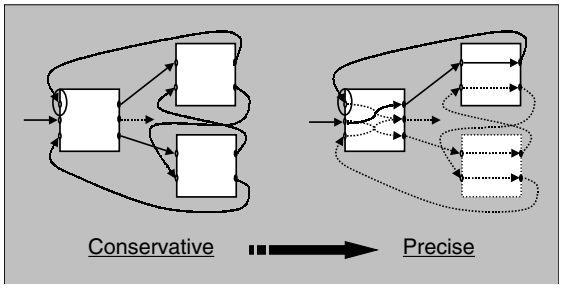

Fig. 1. Increasing Precision of Dependence Analysis.

Figure 1 illustrates the improvement in precision that can be gained when transitional connections are included in the information used to determine possible dependencies. The solid arcs in this figure denote arcs that must be traversed in order to identify a conservative set of dependencies. In the view of the system shown on the left, the transitional connections are unknown. Therefore, when tracing back from the circled port, one must assume that any stimulus applied to input port could have contributed to a response on any output port. The lack of information on the interaction behavior of the component forces the analysis to include all components of the system in the dependency set. The existence of the transitional connections in the view of the system on the right provides

information that allows the analysis to eliminate the component connected only by the dashed arcs.

Rather than constructing a complete dependence graph, Aladdin's analysis is performed on demand in response to an analyst's query. The query might request information about the existence of certain specific kinds of anomalous dependence relationships, or might request information about the parts of the system that could affect or be affected by a specific port in the architecture. A view of Aladdin's interface is shown in Figure 2. A file containing a Rapide architectural specification is selected using the file menu. In this figure a specification for a variant of the familiar gas station example was selected. The specification is displayed in the left pane of the main Aladdin window. The right pane displays the list of component ports that have been identified from the architectural description.

Rapide is a high-level, event-based simulation language that provides support for the dynamic addition and deletion of predeclared components. Rapide descriptions are composed of type specifications for component interfaces and architecture specifications for permissible connections among the components of a system.

System behavior is described through architectural connection rules, state transition rules, and patterns of events required to generate events that activate the rules. System behavior can be simulated through execution of the Rapide description. The results of a simulation of system behavior can be studied using a representation called a *poset*. A poset is a partially ordered set of events captured during a single simulation of a system.

Components are defined in terms of their interfaces. Three types of components are described in Figure 3, which is the Rapide description of the gas station problem. The component types are a pump, a customer, and an operator. In this simple example we see that interfaces specify several aspects of the component's interactions with other components. The declaration of *in* and *out* actions 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 3 is an example. Behaviors, which may involve local variables, describe 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 [12], where a pattern is a set of events together with their partial ordering. The partial order of events is represented as a poset.

The analyst can instruct Aladdin to perform any of several queries. The queries window shown at the top left in the figure appears when the analyst selects the "Queries" menu item. The analyst can choose to see a list of ports with no source or those with no target, which are two kinds of port-related anomalies. The small window to the right of the window "Queries" contains a list of all the ports in the specification that do not have targets. Ports with no

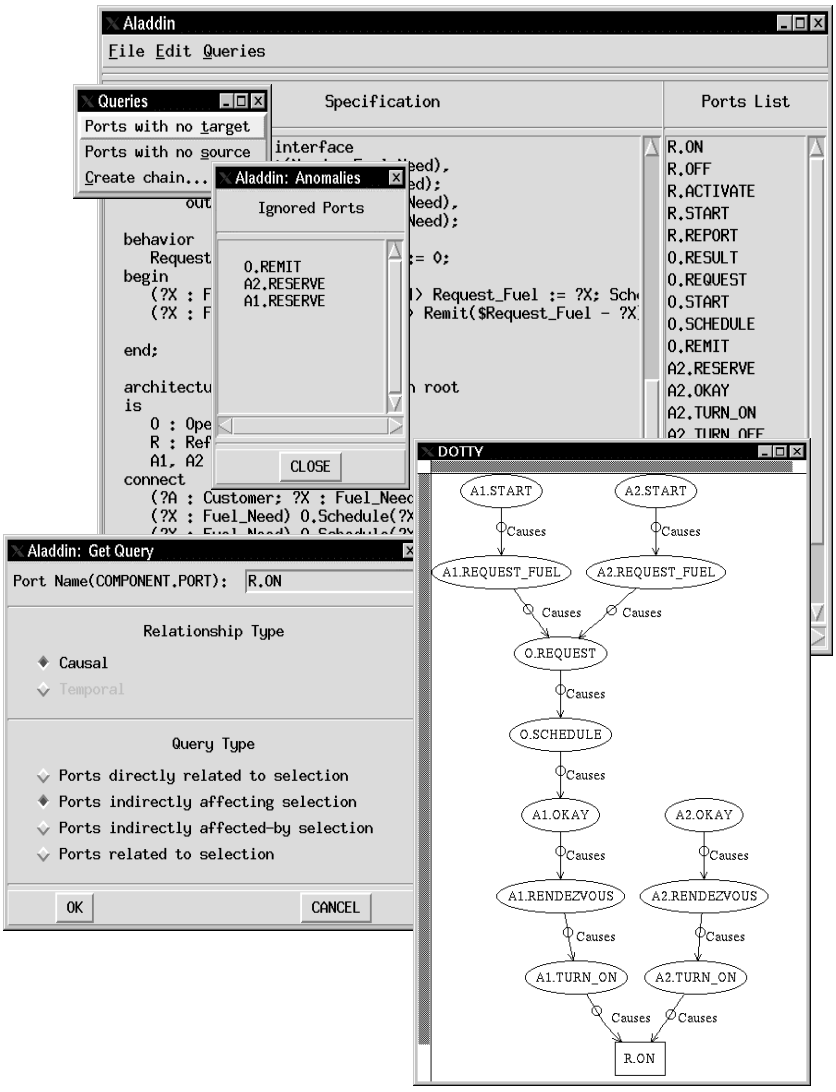

Fig. 2. Use of Aladdin to Identify Anomalies and Perform Port-Based Queries.

source or no target may indicate an unspecified connection or they may indicate a function of the component that is not used in this particular architecture.

The analyst can also choose to create a chain. If "Create chain. . ." is selected, then the window "Get Query" appears. The analyst selects a query, in this case the analyst wanted to see a chain of all the ports in the architecture that could causally affect port `R.ON`. Dotty [3], a graph layout tool, is used to display 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;
```

Fig. 3. Rapide Description of the Gas Station Example [11].

resultant chain, which appears in the window "Dotty". The chain is displayed as a directed graph rooted at the node representing the specified port of interest, in this case the node `R.ON` at the bottom of the graph. The arcs are labeled with a relationship type and represent direct (or perhaps summarized) dependence relationships between pairs of ports. The nodes of the graph represent all ports that could cause, directly or indirectly, the port of interest, the event `R.ON`, to be triggered.

This query was performed in order to help identify the cause of a failure in a Rapide simulation of the gas station. In the simulation it was discovered that `A2` was never allowed to refuel. The cause of this is apparent from viewing the chain, and in fact could have been discovered through running an anomaly check prior to simulation, since the event `A2.OKAY` has no source. Through examination of the chain, the analyst determines that the problem occurs because `O.REQUEST` must record the source of a request so that the appropriate `OKAY` can be triggered.

Aladdin takes advantage of the behavior section of Rapide interface definitions. Aladdin applies a summarization algorithm to the behavioral description in order to identify the transitional connections in the Rapide description. Aladdin can also be used in conjunction with Rapide's simulation tools. If a specification error is detected during a simulation, Aladdin can be used to identify a reduced set of description elements.

As another example, consider the architecture depicted in Figure 4. The components and relationships shown in this figure represent the architecture of a software system called MobiKit [1], which supports the mobility of clients of a distributed publish/subscribe service. Clients of the system first "move out" from one location and then "move in" to a new location. Figure 5 shows a portion of a forward chain resulting from this architecture. The analysis reveals a lack of coordination in the architecture. For example, a mobile client can perform a *moveIn* operation before the *moveOut* is completed.

Aladdin can also be used independently of any particular architecture description language. The analyst can manually define links by using, for example, an informal graphical notation. When all the connections have been identified, the analyst can make queries about the relationship of specific ports to other ports in the architecture, as described above. In this way it supports Jackson and Wing's notion of "lightweight formal methods" [5] in a manner similar to Feather's use of a database [2].

5 Conclusion

As the focus of software design shifts increasingly toward the architectural level, so too are its analysis techniques. Dependence analysis is one such technique that shows promise at this level. For dependence analysis to most effective, however, designers must employ sophisticated, behavior-oriented architectural description languages. As it turns out, the model underlying these languages tends to be that of concurrent, compositional, event-based computation, not the traditional

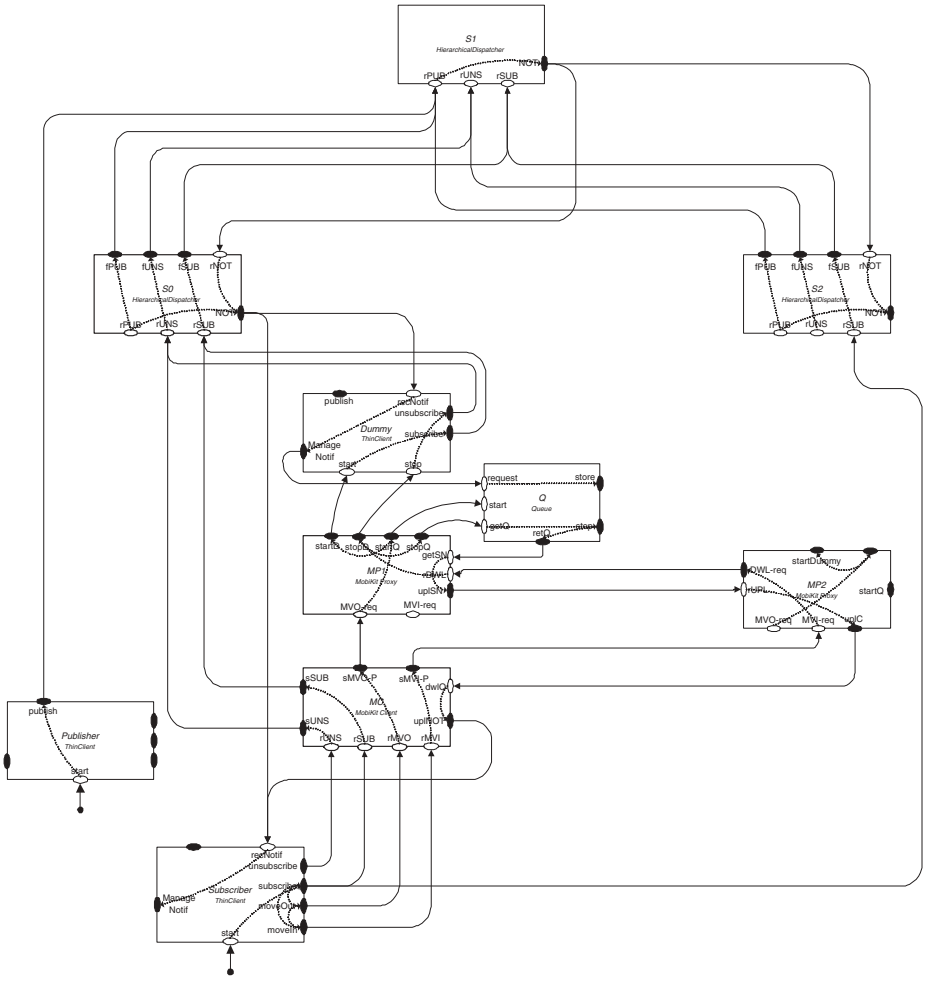

Fig. 4. Architecture of MobiKit.

basis for dependence analysis. Early work is beginning to emerge to extend the theoretical foundation of dependence analysis [7], but much remains to be done.

Acknowledgments

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

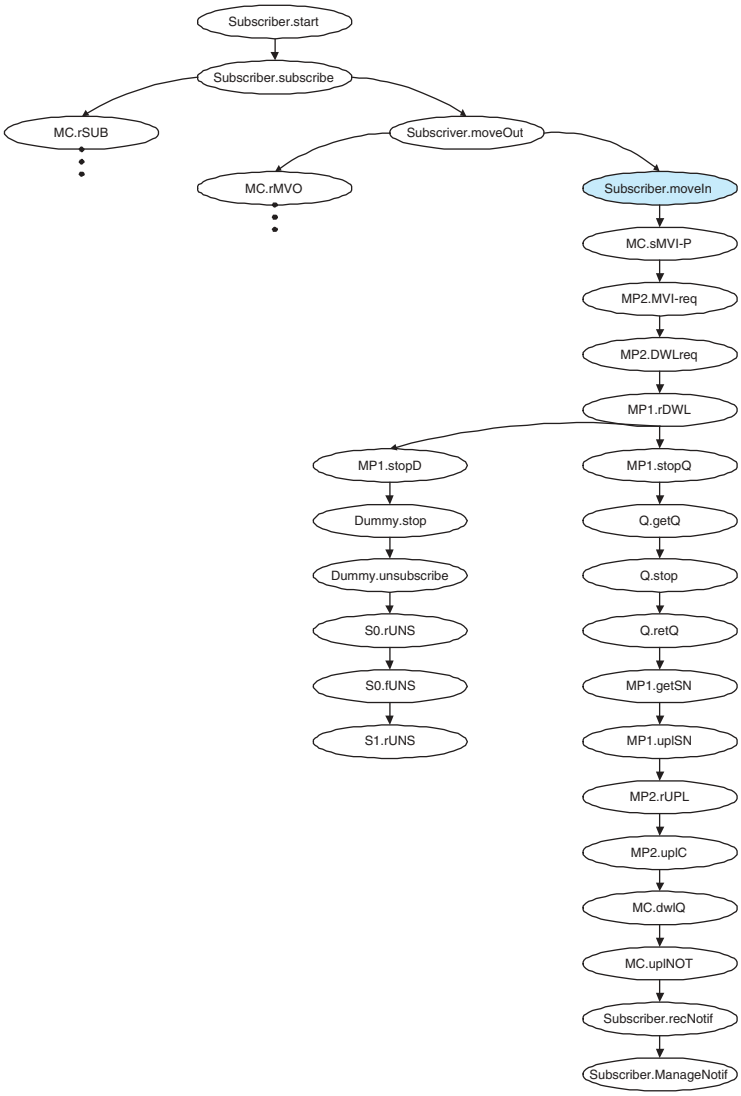

Fig. 5. Portion of a Chain Derived from the MobiKit Architecture.

the Air Force Material Command, Rome Laboratory, and the Defense Advanced Research Projects Agency under Contract Number F30602-00-2-0608. 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 M. Caporuscio was supported in part by the MIUR National Research Project SAHARA.

References

1. M. Caporuscio, A. Carzaniga, and A.L. Wolf. Design and Evaluation of a Support Service for Mobile, Wireless Publish/Subscribe Applications. *IEEE Transactions on Software Engineering*. To appear.
2. M.S. Feather. Rapid Application of Lightweight Formal Methods for Consistency Analyses. *IEEE Transactions on Software Engineering*, 24(11):949–959, November 1998.
3. E.R. Gansner, E. Koutsofios, S.C. North, and K.-P. Vo. A Technique for Drawing Directed Graphs. *IEEE Transactions on Software Engineering*, 19(3):214–230, March 1993.
4. D. Garlan, R. Monroe, and D. Wile. ACME: An Architecture Description Interchange Language. In *Proceedings of CASCON '97*, pages 169–183. IBM Center for Advanced Studies, November 1997.
5. D. Jackson and J.M. Wing. Lightweight Formal Methods. *Computer*, 29(4):21–22, April 1996.
6. 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.
7. J.A. Stafford. *A Formal, Language-Independent, and Compositional Approach to Control Dependence Analysis*. PhD thesis, University of Colorado, Boulder, Colorado, USA, August 2000.
8. 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.
9. J.A. Stafford and A.L. Wolf. Architecture-Level Dependence Analysis for Software Systems. *International Journal of Software Engineering and Knowledge Engineering*, 11(4):431–452, August 2001.
10. RAPIDE Design Team. Draft: Guide to the Rapide 1.0 Language Reference Manuals. July 1997.
11. RAPIDE Design Team. Draft: Rapide 1.0 Architecture Language Reference Manual. July 1997.
12. RAPIDE Design Team. Draft: Rapide 1.0 Pattern Language Reference Manual. July 1997.
13. S. Vestal. *MetaH Programmer's Manual Version 1.27*. Honeywell, Inc., Minneapolis, MN, 1998.
14. M.E.R. Vieira, M.S. Dias, and D.J. Richardson. Analyzing Software Architectures with Argus-I. In *Proceedings of the 2000 International Conference on Software Engineering*, pages 758–761. Association for Computer Machinery, June 2000.
15. J. Zhao. Using Dependence Analysis to Support Software Architecture Understanding. *New Technologies on Computer Software*, pages 135–142, September 1997.