Enriching Component Interfaces with Checkable Dependence Specifications *
(Technical Report #2005-5)

Venkatesh Prasad Ranganath Georg Jung
John Hatcliff
Department of Computing and Information Sciences
Kansas State University †

Matthew B. Dwyer
Department of Computer Science and Engineering
University of Nebraska ‡

June 6, 2005
revised: September 10, 2005

Abstract

Component middleware frameworks such as the CORBA Component Model (CCM) and Enterprise Java Beans (EJB) are increasingly being applied to address challenges involved in building large-scale distributed systems. We seek to provide foundations and tools for model-driven development of such systems in which architectural models serve as highly leveragable abstractions that form a substrate into which a variety of forms of essential behavioral properties are woven. Our vision emphasizes a synergistic integration of analysis and analysis clients at multiple levels of abstraction in a system including both the architectural model (component interfaces and connections) level and the source code (component implementation) level.

Dependence and flow properties have proven to be useful behavioral abstractions that can be leveraged in a variety of ways at both the architectural model level and code level. However, previous works provide no mechanism for connecting architectural-level dependences to code-level dependences, e. g., for the purpose of guaranteeing soundness of model-level specifications by checking that dependences present in implementations conform to model-level specifications – a task that is especially challenging when working with sophisticated component frameworks like CCM.

In this paper, we propose a layered approach to enriching component interfaces specifications to include a variety of forms dependence and flow information. We have implemented conformance checks between layers of component specifications using a flexible Java dependence analysis framework, and we report on our experiences applying the framework to a collection of CCM component specifications and implementations.

*This work was supported in part by the U.S. Army Research Office (DAAD190110564), DARPA/IXO's PCES program (AFRL Contract F33615-00-C-3044), NSF (CCR-0306607, CCF-0429149, CCF-0444167), and Lockheed Martin.

†Manhattan KS, 66506, USA. {rvprasad, jung, hatcliff}@cis.ksu.edu
‡Lincoln, NE, USA. dwyer@cse.unl.edu

1 Introduction

Component middleware frameworks such as the CORBA Component Model (CCM) and Enterprise Java Beans (EJB) are increasingly being applied to address challenges involved in building large-scale distributed systems. In these frameworks, systems are constructed from loosely-coupled reusable components facilitated by platform-independent execution semantics and reusable services that coordinate how application components are composed and deployed. A major advantage of such frameworks is that they enable a high degree of modularity which is achieved through a clear separation of concerns. This separation cleanly isolates different stages of the development process and divides business logic from infrastructure, allowing substantial parts of a system's implementation to be synthesized directly from interface specifications and infrastructure configuration options. It also enables business logic elements to be exchanged over different systems or to be updated or modified separately. For these reasons, component-based development is beginning to take hold in real-time safety/mission-critical domains as a mechanism for incorporating non-functional aspects such as real-time, quality-of-service, and distribution. Moreover, development of such systems is increasingly model-centric with models forming the basis of defining component interfaces and describing component composition to form system assemblies.

In our vision of model-driven development, such architectural models serve as highly leveragable abstractions that form a substrate into which a variety of forms of requirements, middleware policies, and crucial behavioral and quality of service information are woven. This vision includes a *synergistic integration of analysis and analysis clients at multiple levels of abstraction in a system including the component interface level and source code levels.* In this vision, higher-level model-based information can be used as *specifications* against which lower-level implementations are checked for conformance. Conversely, lower-level implementations can be analyzed to discover fundamental semantic and performance characteristics which can be lifted to the model-level and leveraged in a number of ways such as development visualization/understanding, configuration of model-level attributes, optimization of underlying middleware services, etc.

Dependence information is one type of information that is highly-leverageable in the context of component-based model driven development. At the source code level, techniques for capturing, inferring, and applying dependence information have been used in a number of settings including program slicing, program subsetting, impact analysis, program understanding [15], debugging [5], partial evaluation [3], compiler optimizations [4] such as global scheduling, loop fusion, code motion etc. At the architectural model level, the utility of dependence specifications (also called *flow* or *influence* specifications in this context) [17] has been recognized to the extent that industry standards such as the Architecture Analysis and Design Language (AADL) [1] includes such specifications as language primitives. For example, architectural dependence information can be utilized for purposes similar to source level dependences including impact analysis, security-oriented information flow analysis, as well as tasks needed for real-time applications including end-to-end timing and latency analysis, and resource management based on operational flows.

We have developed a number of model-level analyses that work off of architectural dependences in the Cadena framework – a sophisticated IDE that we have constructed for supporting development of component-based systems built using industry standard component technology and deployed on standards-based middleware frameworks [8]. Working with engineers at Boeing and Lockheed Martin, these analyses have been used to automate a variety of tasks associated with development of component-based real-time embedded systems including seeding priorities for effect handlers, detecting design flaws such as circular event dependencies, aiding schedulability analysis, and synthesizing appropriate configuration settings for underlying middleware services.

Although dependence specifications are quite useful both at the architecture/model level and source code level, what is currently missing in previous work is the ability to "link" these levels of abstraction. Specifically, we believe that substantial synergistic effects can be obtained by creating an ability to map dependence information between architectural models and source code. This would enable, for example, developers working at the architectural level to browse and query architectural

dependences and in interesting/troublesome cases "drill down" to determine how higher level dependences are realized at the implementation level. Developers working at the code level would be able to take advantage of the compositional natures of these specifications by automatically checking that intra-component dependences in lower-level implementations conform to higher-level specifications, and then "zooming out" to work with sound model-level abstractions for end-to-end reasoning about of systems composed by connecting components.

While establishing a mapping between model- and code-level dependence information might seem straight-forward at first glance, a number of challenging issues arise when working with realistic and widely-used frameworks like CCM and EJB. These include accommodating different programming models, different threading models, variances in standard component implementation skeletons, and reconciling the tension between the desire for abstract black-box specifications and exposing crucial implementation features (e.g., component mode variables) that play a significant role in controlling component behavior. Almost all existing work on architecture dependences fails to take into account subtleties associated with these issues. Adequate accounting for these issues when attempting to map between models and code seems to require richer architecture dependence specification forms than those proposed in previous work.

In this paper we present a framework for specifying and checking notions of dependence in which dependence information can be mapped across multiple levels of abstraction in component-based systems (component interface, component model/infrastructure features, Java source code). The contributions of this work are as follows.

- We identify a collection of dependence notions that support the type of reasoning and analyses described earlier and (unlike much previous work) are robust enough to be meaningful and semantically clear in both model-level and code-level usage.
- We present a layered dependence specification framework that captures common features of standard component model infrastructures and describes how dependence information can be mapped from component interfaces through the infrastructure model to source code.
- We describe how Indus (our Java program slicer) can be used to check that dependences in a Java implementation conform to model-level dependence specifications – thus providing guarantees of soundness for information that can be leveraged in a variety of ways in a model-driven component/software-product-line development process.
- We validate the effectiveness of the above approach by applying our framework to several different examples of realistic component-based systems.

In summary, these new capabilities move us closer to achieving our vision of model-driven development for real component middleware frameworks in which dependence information is automatically mapped between models and implementations and leveraged in a variety of ways.

The rest of this paper is organized as follows. Section 2 provides a brief overview of CCM (the component middleware framework that we focus on in this paper) and describes the factors that must be accommodated in an effective mapping of dependence information between architecture models and code. Section 3 presents our layered approach for capturing dependence specifications at the component interface and component infrastructure levels. Section 4 describes how these specifications are related to Java code-level notions of dependence and how code-level dependence specifications can be checked against model-level specifications. Section 5 reports on our experience in applying these techniques to examples of representative avionics mission control and command systems provided by Boeing and Lockheed Martin. Section 6 discusses related work, and Section 7 concludes and presents directions for future work.

2 Background

In this section, we give an overview of CCM and an example of a simple CCM component that we will use to motivate our dependence-based specification and checking framework. We then discuss concerns that must be addressed in any framework that attempts to relate model-level dependences, which we term *influences*, and code-level dependences.

2.1 CORBA Component Model

In the CCM architecture, a system is realized as a collection of components. Each component has a *component interface* consisting of one or more *ports* that are used to connect to other components. CCM components *provide* interfaces to clients on ports referred to as *facets*, and *use* interfaces provided by other clients on ports referred to as *receptacles*. Components *publish* events on ports referred to as *event sources*, and *consume* events on ports referred to as *event sinks*. Each port is unidirectional, and one can think of the facet ports and event sink ports as *input ports* in the sense that these ports represent the servicing/accepting end in a communication, while the receptacle and event sink ports can be thought of as *output ports* since they represent the calling/sending end of a communication.

The CCM *interface definition language* (IDL) is used to define component interfaces consisting of named ports as well as interface and event types used as port types. Interface port types are simply a collection of methods signatures (ala Java interfaces) whereas event port types describes the types of elements in event payloads. Thus, interface connections allow a component c_1 that uses an interface of type i from component c_2 to synchronously invoke a method from i on c_2. Event connections allow a component c_3 to publish an event that will be asynchronously dispatched and handled by a subscribing component c_4.

To illustrate the different notions of influence we are interested in, we take the example of the "Lazy-Active" component as used in aviation systems by the Boeing company (figure 1). The Lazy-Active component is designed to bridge the gap between high-frequency sensors and low-frequency processing components. It features two event ports ($\rightarrow \alpha$ consumes an event and $\beta \rightarrow$ publishes an event), and two interface ports ($\gamma \leftrightarrows$ provides an interface to clients and $\leftrightarrows \delta$ uses an interface provided by another component). The receipt of an event on port α signifies that new sensor data is available. The Lazy-Active component reacts with setting an internal *flag* to *stale*, indicating that the cached *data* is not valid any longer. In addition it publishes an event on port β to inform downstream components that the new data is available. In parallel, but not necessarily with the same frequency, the component serves data-requests on port γ by calling method $f()$ which returns data from the cache. During such data-requests, if the flag is set to *stale* then Lazy-Active retrieves data by invoking a method, $g()$, on port δ and updates the cache before serving the data.

2.2 Concerns

Programming Model The underlying programming model of the component implementation framework dictates how various features of the component model are provided and realized. For example, CCM supports segmented (a federation of objects cater the services provide by the component) and monolithic (a single object caters the services provided by the component) implementations. In the latter case, the relationship between various interfacial elements of the components is spread across multiple objects as opposed to a single object acting as the component. Clearly, conformance checking of such an implementation is impossible without knowing the how various features of a component model are realized in the underlying programming model.

Behavioral Specifications Past efforts [2, 6, 12] pertaining to software architecture specifications via ADL dealt with behavioral specification in architectural components. The semantics/behavior of architectural components were captured via relationships between the component's interfacial

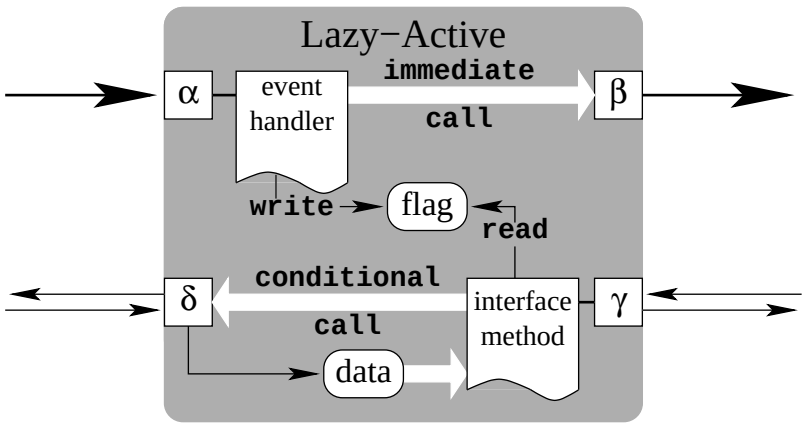


Figure 1: The Lazy-Active Component

elements. Recently, we supported a very similar notion for components in Cadena [8]. However, it was unclear how to derive structural ordering for the purpose of component implementation synthesis. Also, as these specifications are not mapped to component implementation frameworks (in terms of the mapping of component model elements to programming model element and the arity of such a mapping), it is unclear how these specifications would be leveraged to perform conformance checking.

Execution models In a sequential execution model, the behavior associated with a port is contained within the servicing thread. However in a concurrent execution model, the behavior of a port may span multiple threads. In such cases, conformance checking needs to be aware of the nature of the execution model and consider the effect of concurrency on the behavior for the purpose of checking.

Black Box v/s White Box An issue in model driven approaches to component-based system development is that developers tend to specify component interfaces and follow up with a progression of refined component models to arrive at an implementation tied to a component implementation framework. From an engineering perspective, support for model refinement should be able to check each progression for conformance. This removes the burden of checking system assemblies for conformance provided the assembly only uses specification conformant components and the assembly has been checked based on the specifications of the involved components. Our experience with industrial partners confirms these observations.

3 Component-level Influence

In this section, we describe a layered approach for enriching basic component interface descriptions. This approach is designed to support the separation of concerns that is often found in large component-based development projects where a designer defines component interfaces, a developer implements components, and a system integrator assembles a system from a collection of component implementations. By enriching interfaces, we provide a means for designers to communicate additional checkable constraints to implementors and for integrators to exploit more refined analysis results for assemblies of components enriched with those constraints.

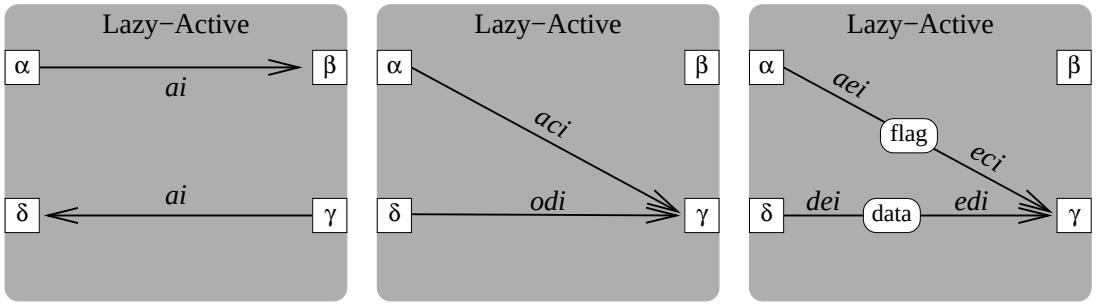
Component interfaces descriptions, as expressed in IDL for example, serve as the basic layer in our approach and for consistency, in this paper, we will present our approach in terms of the CCM.

While CCM distinguishes between message-based ports (i.e., event sources/sinks) and operation-based ports (i.e., facets/receptacles), we only need to distinguish between *input* and *output* ports. In Figure 1, α and γ are input ports while β and δ are output ports.

Additional layers of component specification involve describing relationships between the execution of implementations associated with ports. Specifically, we capture whether one aspect of a component implementation can *influence* the behavior of another aspect of a component implementation. We use the term influence, instead of dependence, because in performing design we have found it more natural to think about the potential future behavior of a component, whereas in other applications, like debugging, notions of dependence history are more natural.

Our goal is to be able to capture such relationships in black-box specifications that are implementation-independent. To achieve this, we define an abstract component execution model. Components may have data *elements* that store the state of a component instance. Component functionality is invoked when an input port is *activated* (i.e., a message is received or a synchronous call is made). Conceptually, each input port activation has associated with it several sequences of *actions*, consisting of internal component operations or activations of output ports, among which one is selected based on input and component data values.

We distinguish between *relational* and *trace-based* component interface specifications. A relational specification describes the influence of one port or element on another port or element. In contrast, a trace-based specification describes the sequencing of port and element actions. Conceptually, trace-based specifications carry more information and are closer to implementation. In this paper, we consider relational specifications since they represent the initial steps one would take in refining component interface specification. The refinement of relational to trace-based specifications, which can be captured as state-machines, will be treated in future work.



(a) activation influence (b) data/control-data influence (c) data/control influence

Figure 2: The Lazy-Active Component

3.1 Influences

We introduce two layers of influence specification. The first, which relates how activation of an input port can influence activation of an output port, is called the *port relation layer* (P); it is defined in terms of a simplified component execution model that elides details about actions and elements. The second layer, which is called the *port-element relation layer* (PE), relates activation of a port to modifications or tests of a component data element. The PE layer is a refinement of the P layer in the sense that all of the relations definable in P are definable in PE.

3.1.1 Port Relation Layer

In this layer, we express influences between port activations and between parameter values passed in port activations. There are three primary forms of influence in this layer.

An *activation influence* ($\overset{ai}{\rightarrow}$) captures whether the activation of an input port can subsequently lead to the the activation of an output port. Input ports are activated externally, thus we do not capture the activation of input ports. Figure 2(a) summarizes the activation influences in the Lazy-Active example. $\alpha \overset{ai}{\rightarrow} \beta$ since the activation of α (upon receiving a message) causes the activation of β (resulting in the sending of a message). Similarly, $\gamma.f() \overset{ai}{\rightarrow} \delta.g()$ as the call on the method, f, associated with the interface γ leads to a call to method, g, associated with the output interface δ; the . operator is used to resolve specific methods on an interface port.

Activation influences cannot capture the relationship between the data passed via ports. We introduce *data influences* to capture such relationships. In the Lazy-Active example, we capture the fact that the output data on $\delta.g()$ is used to calculate the output data on $\gamma.f()$ as an *output data influence*, which is denoted as $\delta.g() \overset{odi}{\rightarrow} \gamma.f()$ and illustrated in Figure 2(b). Similarly, the influence of an operand to an operation on an input port on an operand to an operation on an output port is captured as *input data influence* ($\overset{idi}{\rightarrow}$).

Data passed on ports may influence the choice of actions that is executed by the component. This is captured by two forms of *control influences*. An *input control influence* ($\overset{ici}{\rightarrow}$) captures the fact that data passed on an input port can determine the sequence of actions executed by the component; in contrast, a lack of input control influence indicates that input data is processed by a fixed sequence of actions that calculate an output value based on the input. Similarly, *output control influence* ($\overset{oci}{\rightarrow}$) captures the influence of data provided by an output port on the action sequence of an input port. Our component execution model only captures the actions sequences associated with input ports, thus the target of both forms of influences are input ports. In the Lazy-Active example, if the data received on α influenced (via *flag*) the choice of actions executed when $\gamma.f()$ is activated then $\alpha \overset{ici}{\rightarrow} \gamma.f()$ as illustrated in Figure 2(b). It is also possible that the event of activation (and not the received data) of α may influence (via *flag*) the choice of actions executed when $\gamma.f()$ is activated. Such influences are captured as *activation control influence* ($\overset{aci}{\rightarrow}$). Hence, in the above case, $\alpha \overset{aci}{\rightarrow} \gamma.f()$. Similarly, the event of activation of a port i may influence the data provided by another port j. We capture this form of influence as *activation data influence* ($\overset{adi}{\rightarrow}$).

3.1.2 Port-Element Relation Layer

In this layer, we refine the main influence notions of the P layer to explicitly capture the influence of and on data elements that encode the state of a component instance. This allows developer to specify that relationships between ports are indirect thereby allowing for finer reasoning about component assemblies.

The influences introduced in P layer are applicable at the PE layer. We introduce *element influences* to capture the relationship between the ports and the data elements of a component. The first two forms of element influence, *data element influences* ($\overset{dei_o}{\rightarrow}$ and $\overset{dei_i}{\rightarrow}$) and *element data influences* ($\overset{edi_i}{\rightarrow}$ and $\overset{edi_o}{\rightarrow}$), capture the influence of the input/output data available from a port on a data element and vice versa. The influence of data elements on the choice of actions associated with input ports is captured by *element control influence* ($\overset{eci}{\rightarrow}$). An *activation element influence* ($\overset{aei}{\rightarrow}$) captures the fact that a port activation can lead to modification of a component element. The fact that an element is used in the action sequence responding to an input port activation to determine the activation of an output port is captured as a *element activation influence* ($\overset{eai}{\rightarrow}$); note that this influence relates a pair of an input port and element with the influenced output port.

In the Lazy-Active example, the following PE influence specifications reflect an enriched component interface which is partially illustrated in Figure 2(c). Activation of α may change the value of *flag*: $\alpha \overset{aei}{\rightarrow}$ *flag*. The value of *flag* may determine the sequence of actions executed in $\gamma.f()$: *flag* $\overset{eci}{\rightarrow} \gamma.f()$. The output of a call to method g on port δ may change the value of *data* : $\delta.g() \overset{dei_o}{\rightarrow}$ *data*. The value of *data* may determine the value returned by method f on port γ :

$data \overset{edi_o}{\rightarrow} \gamma.f()$. Finally, *flag* may determine if g is invoked on port δ in an activation of $\gamma.f$: $(\gamma.f(), flag) \overset{eai}{\rightarrow} \delta.g()$ which is a refinement of the data-independent activation influence illustrated in in Figure 2(a).

3.2 Definitions

In this subsection, we provide formal definitions for the various notions of influences introduced above and define the relation between influences in P and PE layers thereby demonstrating that it is a refinement.

We begin with some notation used in defining influences. We denote the ports of a component as follows: I_m are the set of message-based input ports, O_m are the set of message-based output ports, I_{op} are the set of operations on the operation-based input ports, and O_{op} are the set of operations on operation-based output ports.

$\mathcal{E}$ is the set of data elements in the component. For convenience, we consider a single domain of values for data elements, D. $d_e(e : \mathcal{E}) : D$ returns the data associated with the element e. $d_i(i : (I_m \cup I_{op})) : D$ returns the data provided to the port or operation i. $d_o(o : (O_m \cup O_{op})) : D$ returns the data provided by the port or operation o.

To abstractly model the processing of data within components we define $t(d : D) : D$ as an n-ary function where all inputs except d are elided. We use $t(()e)$ to express functional dependence on a data element e.

We are interested in modeling the effect of data on the action sequences executed in response to input port activation. Actions can be either port activations, definition of or reference to a component element, e_{def} or e_{ref}, or hidden internal actions within the component. We are not, however, interested in modeling the action sequences themselves. Note that there may be many possible action sequences for a given input port activation with a specific sequence determined based on parameter values and component state. Let $a(i : I_m \cup I_{op})$ denote the set of all such sequences for an input activation; we interpret sequences as sets of actions when convenient. Let $a_{e=v}(i : I_m \cup I_{op})$ denote the set of sequences for an input port activation when the data state of the component or port parameter is consistent with the constraint $e = v$, where $e \in \mathcal{E}$ or e is a parameter to i. To abstractly model the effect of a data value on the action sequences associated with an input port activation we define a predicate $a(d : D, i : I_m \cup I_{op}) : \{true, false\}$ as:

$$\exists v_1, v_2 \in D \mid v_1 \neq v_2 \wedge a_{d=v_1}(i) \cap \overline{a_{d=v_2}(i)} \neq \emptyset$$

Thus, the predicate $a(d, i)$ indicates whether data element d is capable of controlling the action sequence executed in response to i's activation.

Based on these definitions, Tables 1 and 2 contain definitions of P and PE layer influences, respectively.

The definition of *activation influence* in Table 1 does not capture the requirement that o should always be activated after i is activated. This can be captured by changing the existential quantifier to an universal quantifier. This is true for other influences too. We characterize this aspect of the definition as the *strength* of the specification. Based on our experience, existential and universal quantifiers are insufficient as the model is refined. Hence, in our framework, we envision that the developers will start with these definitions and adjust the strength requirement as the system evolves.

Influence	**Type**	**Definition**
activation	$\overset{ai}{\rightarrow}: (I_m \cup I_{op}) \times (O_m \cup O_{op})$	$i \overset{ai}{\rightarrow} o \implies \exists \sigma \in a(i) \mid o \in \sigma$
input data	$\overset{idi}{\rightarrow}: (I_m \cup I_{op}) \times (O_m \cup O_{op})$	$i \overset{idi}{\rightarrow} o \implies t(d_i(i)) = d_i(o)$
ouput data	$\overset{odi}{\rightarrow}: (O_m \cup O_{op}) \times (I_m \cup I_{op})$	$o \overset{odi}{\rightarrow} i \implies t(d_o(o)) = d_o(i)$
input control	$\overset{ici}{\rightarrow}: (I_m \cup I_{op}) \times (I_m \cup I_{op})$	$i \overset{ici}{\rightarrow} j \implies a(t(d_i(i)), j)$
output control	$\overset{oci}{\rightarrow}: (O_m \cup O_{op}) \times (I_m \cup I_{op})$	$o \overset{oci}{\rightarrow} i \implies a(t(d_o(o)), i)$
activation data	$\overset{adi}{\rightarrow}: (I_m \cup I_{op} \cup O_m \cup O_{op}) \times (I_m \cup I_{op} \cup O_m \cup O_{op})$	$\overset{adi}{\rightarrow} \equiv \overset{aei}{\rightarrow} \circ \overset{edi_i}{\rightarrow} \cup \overset{aei}{\rightarrow} \circ \overset{edi_o}{\rightarrow}$
activation control	$\overset{aci}{\rightarrow}: (I_m \cup I_{op}) \times (I_m \cup I_{op})$	$\overset{aci}{\rightarrow} \equiv \overset{aei}{\rightarrow} \circ \overset{eci}{\rightarrow}$

Table 1: P Layer Influence Definitions

Influence	**Type**	**Definition**
element data input	$\overset{edi_i}{\rightarrow}: \mathcal{E} \times (O_m \cup O_{op})$	$e \overset{edi_i}{\rightarrow} o \implies t(d_e(e)) = d_i(o)$
element data output	$\overset{edi_o}{\rightarrow}: \mathcal{E} \times (I_m \cup I_{op})$	$e \overset{edi_o}{\rightarrow} i \implies t(d_e(e)) = d_o(i)$
data element input	$\overset{dei_i}{\rightarrow}: (I_m \cup I_{op}) \times \mathcal{E}$	$i \overset{dei_i}{\rightarrow} e \implies t(d_i(i)) = d_e(e)$
data element output	$\overset{dei_o}{\rightarrow}: (O_m \cup O_{op}) \times \mathcal{E}$	$o \overset{dei_o}{\rightarrow} e \implies t(d_o(o)) = d_e(e)$
element control	$\overset{eci}{\rightarrow}: \mathcal{E} \times (I_m \cup I_{op})$	$e \overset{eci}{\rightarrow} i \implies a(t(d_e(e)), i)$
activation element	$\overset{aei}{\rightarrow}: I_m \cup I_{op} \times \mathcal{E}$	$i \overset{aei}{\rightarrow} e \implies \forall \sigma \in a(i) \mid e_{def} \in \sigma$
element activation	$\overset{eai}{\rightarrow}: ((I_m \cup I_{op}) \times \mathcal{E}) \times (O_m \cup O_{op})$	$(i, e) \overset{eai}{\rightarrow} o \implies (\exists \sigma \in a(i) \mid e_{ref} \in \sigma \wedge o \notin \sigma)$ $\wedge (\exists \sigma \in a(i) \mid e_{ref} \in \sigma \wedge o \in \sigma)$

Table 2: PE Layer Influence Definitions

3.2.1 Intra-layer Relation Conformance

Based on the above definitions, all P layer influences can be realized with PE layer influences as follows.

$$
\begin{array}{lcl}
\overset{ai}{\rightarrow} & \Longleftarrow & \overset{eai}{\rightarrow} \\
\overset{idi}{\rightarrow} & \equiv & \overset{dei_i}{\rightarrow} \circ \overset{edi_o}{\rightarrow} \\
\overset{odi}{\rightarrow} & \equiv & \overset{dei_o}{\rightarrow} \circ \overset{edi_i}{\rightarrow} \\
\overset{ici}{\rightarrow} & \equiv & \overset{aei}{\rightarrow} \circ \overset{eci}{\rightarrow} \\
\overset{oci}{\rightarrow} & \equiv & \overset{dei_o}{\rightarrow} \circ \overset{eci}{\rightarrow} \\
\overset{adi}{\rightarrow} & \equiv & \overset{aei}{\rightarrow} \circ \overset{edi_i}{\rightarrow} \cup \overset{aei}{\rightarrow} \circ \overset{edi_o}{\rightarrow} \\
\overset{aci}{\rightarrow} & \equiv & \overset{aei}{\rightarrow} \circ \overset{eci}{\rightarrow}
\end{array}
$$

Even though $\overset{aci}{\rightarrow}$ and $\overset{adi}{\rightarrow}$ do not explicitly reference component data, and can thus be thought of as a P influence, they can only be defined with the use of PE influences. Given these equivalences, it is clear that a PE layer specification can be checked to see if it is a refinement of a P layer specification (i.e., that all of the relations at the PE layer are contained in the equivalent relations derived from the native P layer relations).

Algorithms for checking conformance between P and PE layer specifications and component implementations are discussed in Section 4.3. Those same techniques can be used to check conformance between PE and P layer specifications.

3.2.2 Expressing Enriched Component Interface Specifications

The definitions in this section should be viewed as an assembly language of influence relations. We do not expect developers to express component influences by writing textual descriptions relating ports and elements. We envision extensions to component-based IDEs, such as Cadena, in which developers will select features of components and be presented with the options for relating those features through influences. Furthermore, we believe that experience using these primitive constructs will lead to the discovery of patterns of primitive constructs. Subsequently, most influence specifications will be expressed using much higher-level derived influence patterns.

4 Program-Level Dependence

In this section, we review notion of program dependences, describe how component influences can be mapped to program dependences in component implementations, and present a dependence-based approach for conformance checking of influence specifications against component implementations.

4.1 Dependences

Conceptually, program dependences capture the relationship between program points and program variables. Data and control dependences are the most basic notions and are based on the aspect of chains of assignments (i.e., data flow) and control structuring (i.e., control flow) in the program. Their (paraphrased) definitions are given below.

Definition 1 *Given two program points m and n in a program P, m is data dependent on n ($n \overset{dd}{\rightarrow} m$) if the definition of a variable at n reaches the use of the same variable at m.*

Definition 2 *Given two program points m and n in a program P, m is control dependent on n ($n \overset{cd}{\rightarrow} m$) if n has more than one immediate successor and the execution of m can be delayed by choosing one of these successors to be executed after n.*

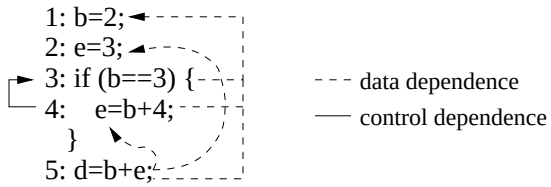


Figure 3: Program dependences

We illustrate these concepts with the small example in Figure 3. The occurrence of *b@5* (i.e., variable `b` at line 5) is data dependent on the definition of *b@1* (at line 1) as this definition *will* reach line 5. In contrast, the occurrence of *e@5* is data dependent on the definition of e at lines 4 and 2 as either of these definitions *may* reach line 5. These dependences, along with $b@1 \overset{dd}{\rightarrow} e@4$, are *direct*, but we we can combine chains of dependences, such as $b@1 \overset{dd}{\rightarrow} e@4$ and $e@4 \overset{dd}{\rightarrow} e@5$, to assert that *e@5* is *indirectly* data dependent on *b@1*. We denote chains of dependences using $\overset{dd^*}{\rightarrow}$, for chains of length 0 or more, or $\overset{dd^+}{\rightarrow}$, for chains of positive length; any dependence definition can be *iterated* in this way. Depending on the value of `b`, the conditional at line 3 may cause the execution of line 4 to be delayed (infinitely in this case), thus $3 \overset{cd}{\rightarrow} 4$. There are no other control in this example.

4.2 Realizations

Clearly, influence and dependence are similar as they capture the same sort of relations but in different abstractions, i.e, specification and implementation. Hence, it is possible to realize influence specifications in terms of program dependences. As we are dealing with the transition from component interface specification to component implementation, such a realization enables influence conformance checking.

For the purpose of simplicity, we consider a procedural sequential programming model in which to implement the component and the implementation framework. We also introduce the following terms in conjunction with the component model and the programming model.

- f_i is the procedure invoked when a message is received through an input port i or an operation $f()$ is serviced on an input port i,
- f_o is the procedure invoked when a message is sent through an output port o or an operation $f()$ is initiated on an output port o,
- $a \overset{calls}{\rightarrow} b$ indicates that procedure a directly calls procedure b,
- $\rho_i(f)$ is the program point that reading the arguments to a procedure f,
- $\rho_o(g)$ is the program point that returns the value while exiting from a procedure g,
- $\theta(d)$ is the programming model representation of a model level data element,
- $\mathcal{P}(f)$ is the program points in the procedure f and procedures reachable from f,
- v_p is the program variable v at program point p, and
- $a \overset{c^*d^*}{\longrightarrow} b = a \overset{cd^*}{\rightarrow} v_p \overset{dd^*}{\rightarrow} b$.

Given the above terms and assuming that the data elements are atomic, the semantics of influences can be provided in terms of program dependences as given below.

$$[\![i \overset{dei_i}{\rightarrow} e]\!] = \rho_i(f_i) \overset{dd^*}{\rightarrow} v_p \overset{c^*d^{**}}{\longrightarrow} \theta(e) \quad (1)$$

$$[\![e \overset{edi_i}{\rightarrow} o]\!] = \theta(e) \overset{dd^*}{\rightarrow} v_p \overset{c^*d^{**}}{\longrightarrow} \rho_i(f_o) \quad (2)$$

$$[\![o \overset{dei_o}{\rightarrow} e]\!] = \rho_o(f_o) \overset{dd^*}{\rightarrow} v_p \overset{c^*d^{**}}{\longrightarrow} \theta(e) \quad (3)$$

$$[\![e \overset{edi_o}{\rightarrow} i]\!] = \theta(e) \overset{dd^*}{\rightarrow} v_p \overset{c^*d^{**}}{\longrightarrow} \rho_o(f_i) \quad (4)$$

$$[\![e \overset{eci}{\rightarrow} i]\!] = \theta(e) \overset{dd^*}{\rightarrow} v_p \overset{c^*d^{**}}{\longrightarrow} v_q \overset{cd^+}{\rightarrow} v_r \in \mathcal{P}(f_i) \quad (5)$$

$$[\![i \overset{aei}{\rightarrow} e]\!] = \theta(e) \in \mathcal{P}(f_i) \quad (6)$$

$$[\![(i,e) \overset{eai}{\rightarrow} o]\!] = \theta(e) \in \mathcal{P}(f_i) \wedge \theta(e) \overset{dd^*}{\rightarrow} v_p \overset{c^*d^{**}}{\longrightarrow} v_q \overset{cd^+}{\rightarrow} \rho_i(g_o) \quad (7)$$

$$[\![i \overset{ai}{\rightarrow} o]\!] = f_i \overset{calls^+}{\rightarrow} g_o \quad (8)$$

4.2.1 Subtleties

In case of structured data elements, it is possible for a data element a to affect another data element b that is contained in data element c. If in such a case, c is the data being returned by a port then a does influence the data returned by the port. In terms of program dependence, this case is captured by $a \overset{dd}{\rightarrow} c.b$ and not by $a \overset{dd}{\rightarrow} c$. However, the above semantics do not capture the former case. This can be addressed by extending the above semantics to capture the effect on an element of a complex data element as an effect on the complex data element. For example,

$$[\![e \overset{edi_i}{\rightarrow} o]\!] = \theta(e) \overset{dd^*}{\rightarrow} v_p \overset{c^*d^{**}}{\longrightarrow} v_q \wedge (\exists a \in \mathcal{E}.a \overset{edi_i}{\rightarrow} o \wedge \theta(a) \overset{dd^*}{\rightarrow} v_q).$$

Note that control dependence is used to realize data influence and data dependence is used to realize control influence. This is due to the subtle difference between influences and program dependences – influences capture the effect of data and control on both data and control flow while dependences capture the effect data flow and control flow in isolation.

The basic dependence definitions do not specify how to handle concurrency, i.e, what if m and n belong to different threads in the same program? Since concurrency is common in modern day programs, various forms of concurrency-aware program dependence such as *interference dependence* [10] and *ready dependence* [7] have been proposed to handle data flow aspects and control flow aspects, respectively, in the presence of concurrency. Hence, concurrency can be handled by extending data dependence and control dependence to subsume interference dependence and ready dependence, respectively.

4.3 Conformance checking

Given the above realizations, it is possible to mechanically check if the implementation conforms to P and PE layer influence specification. We designed a general graph exploration algorithm to check for conformance between the component implementation and various layers of specifications. This algorithm is similar to the one proposed by Liu et. al. [11].

In our design, the programming language counterpart of the influence semantics is coded up as a finite automaton with the calls and dependence relations represented as CALLS and CD or DD transition labels, respectively. Depending on the influence being checked, either the call graph or a multi-graph formed by combining various dependence graphs is selected. Figure 4 illustrates a finite automaton that recognizes rooted paths of the form $\overset{dd\,cd\,dd}{\rightarrow\rightarrow\rightarrow}$ and how a conformant dependence path in the program from Figure 3 can be recognized by the automaton.

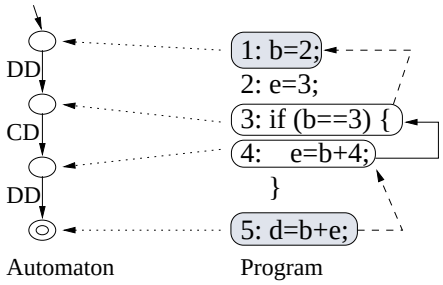


Figure 4: Conformance checking

Based on the implementation framework of the component model, the port level entities are mapped to their programming language counterpart. This mapping is vendor specific and it can be provided in a machine readable form by the vendor of the implementation framework. Usually the programming model entities are either `return` statements or variable/parameter reading/writing expressions. However, it is possible that data elements can be represented via *get* and *set* methods. In such cases, it suffices to capture dependences with respect to the invocation of such procedures. For simplicity, we convert both graphs into directed labeled graphs.

4.3.1 Algorithm

We refer to a path starting with node m as a *m-rooted path*. Given a path p in a labeled graph and an automaton a, we say p is *a-conformant* if a accepts the string of labels in p.

Our algorithm (given in Figure 5) accepts a directed labeled graph g and an automaton a along with two nodes s and e from g. It explores g to discover *s-rooted a-conformant* paths. The algorithm maintains the current state t of the automaton corresponding to each path p being explored. Upon visiting a node m, the algorithm creates *p'*, the path being explored, by extending p with m. If m is e and the state t associated with *p'* is a final state in a then *p'* is an *s-rooted a-conformant* path. Hence, the number of *s-rooted a-conformant* paths and the lower bound on the number of possible *s-rooted* paths are incremented by 1. Otherwise, the algorithm checks if *p'* can be extended to n, an immediate successor of m reachable via label l, such that the extended path is *a-conformant*, i.e, a transition labeled l exists from s to *s'* in a. If so, the algorithm records *(n,s',p')* and proceeds with the exploration. The algorithm terminates when there are no paths that can be extended to be *a-conformant*.

To avoid infinite loops, paths that lead to a node already in the path are not further explored. This does not lead to incorrect results as all closures in the finite automaton are non-finite. Also, as the algorithm does not consider paths ending with a cycle, it is *non-termination insensitive* [16], i.e, it does not consider infinite paths (divergence), due to loops, as possible non-conformant paths. This can be remedied by incrementing the lower bound on the number of possible paths when a node existing on the current path is visited and by considering the variant of control dependence that accounts for divergence while creating dependence graphs.

Upon termination, the algorithm returns the *conformance index*, the ratio of the number of *s-rooted a-conformant* paths to the lower bound on the number of possible *s-rooted* paths. A non-zero conformance index indicates conformance, i.e, a path in the source code between the program points represented by the nodes s and e realizes the corresponding influence. A conformance index of 1 indicates that all execution paths between the two program points of interest realize the corresponding influence.

Multiple paths can exist between the program points of interest in the dependence/call graph of the component implementation and all or some of these may realize the influences based on the realizations. This is a situation similar to that described in Section 3.2. However, in this case, the

```
CHECK(g, a, s, e)
 1  g : graph representation of the implementation
 2  a : conformance automaton
 3  s : a node from g
 4  e : a node from g
 5  w : a worklist of triples
 6  w = (s, initState(a), [])
 7  matched = 0
 8  possible = 0
 9  while w ≠ ∅
10  do (m, t, p) = remove(w)
11     p' = p · [m]
12     if isInFinalState(a, t) and m = e
13        then matched = matched + 1
14             possible = possible + 1
15        else for each (n, l) in succs(g, m)\p'
16             do if canPerformTransitionLabelled(d, l)
17                   then t' = resultingState(a, t, l)
18                        w = w ∪ {(n, t', p')}
19                   else possible = possible + 1
20  return matched/possible
```

Figure 5: Conformance Index Calculation

precision of the check or the value of conformance index can be improved by using θ mapping that is sensitive to the instances of data elements in the influence specification and their representation at various program points in the programming model.

5 Experience and Discussion

We have applied our approach and conformance checker to a collection of component models and implementations drawn from realistic industrial systems. In this section we describe our experiences and discuss some preliminary conclusions we can draw from the data we have collected..

5.1 Setup and Execution

We selected 4 components from systems specified by Boeing and Lockheed Martin; the Lazy-Active component presented in Section 2 was one among the selected components. These components along with complete systems containing the components were modeled in Cadena by Cadena's developers (who are not among the authors on this paper) and skeletal implementations were automatically generated via the backend provided by OpenCCM [14], a Java implementation of CCM. Subsequently, Cadena developers studied the C++ implementations of the systems provided by Boeing and Lockheed Martin, coded up semantically equivalent business logic in Java, and integrated it into the generated component implementations.

It is not uncommon for components to provide simple *set* and *get* ports to clients. The control and data flows in such methods are trivial and any dependence analysis could be used to check influence conformance, including ours. To assess our conformance checking algorithm, we selected the most complex operation and message handling methods in the four component implementations with the most complex control flow; there were 3, 5, 3, and 6 methods chosen from the four components, respectively. These 17 methods ranged from 6-10 on McCabe's cyclomatic complexity measure and

3-10 on the NPath complexity [13] measure. The number of data elements in the selected components ranged from 2 to 5. Our experience suggests these methods reflect a realistic level of component method complexity.

We implemented the algorithm described in the previous section as a conformance checker in Java on top of Indus [9], a toolkit for Java program analysis, understanding, customization, and adaption. Indus provides a rich collection of concurrency-aware program analyses such as object flow analysis, escape analysis, call graph analysis, and a collection of dependence analyses. It also provides a concurrent program slicing library that builds off of these analyses and serves as the base for the first publicly available program slicer for Java and Eclipse. The rich support for various data structures along with the high modularity, precision, and configurability of these analyses eased our implementation. Indus also provides a generic implementation of parametric query engine, called Kaveri, that can be leverage to easily realize the conformance checker. At the time of writing, incompleteness in the implementation of Kaveri forced us to write parts of the checker by hand [1].

For each component, we (1) manually identified PE layer influence specifications; the number of specifications was distributed unevenly across the four components (3, 19, 5 and 13 respectively). Then we (2) manually identified program points in the corresponding implementations that represented the model elements captured in the influence specifications. Finally, we (3) executed the checker with these program points as input to extract the conformance index. Step (3) involved analysis of the component's implementation to calculate the call graph and the program dependence information that was later used by the checker. The execution environment comprised of Java platform v1.4.2 with 64MB of maximum heap space running on a 1.4GHz Pentium running Linux.

5.2 Results and Discussion

In these case studies, the output of the checker matched the intended PE layer influence specifications in all cases. The checker returned non-zero conformance index when a conformant path did exist in the implementation while it returned a zero conformance index when no such path existed. For the non-zero conformance index values, the indices ranged between 0.004 and 0.75 across the entire set of conformance checks for the four components. Step (3) for all runs finished in less than 30 seconds. We believe that these results bode well for broad application of conformance checking of enriched component interface specifications against implementations. The run-time is sure to be reduced as we bring the optimized Kaveri checker online. We note several important issues encountered in this work that ought to guide follow-on work.

(1) In OpenCCM, each component is deployed in a container. This container manages any interactions for the component. Hence, each activation of an output port is via invocation of the corresponding method on an component specific extension plugged into the container and then the container eventually dispatches the message/operation to the target component. So, while checking for activation influence, we provided the invocation site, where the activation is handed over to the container, as the end point instead of the invoked method. This does not lead to incorrect results as it is based on the semantics of OpenCCM. Also, this enables modular and faster checking as the implementation of other components interacting with the component being checked need not be analyzed.

(2) Neither our algorithm nor our checker considered return paths via backtracking. This affected the checking of activation influence by yielding conformance indices that were below 1 in several case studies where they could have been 1. However, this can be trivially remedied by extending the algorithm to consider procedural return paths while checking for activation influence.

(3) In CCM, the component interface specification is provided in the IDL3. The CORBA3 specification provides a translation scheme from IDL3 language to IDL2. Similarly, various specifications to compile IDL2 to Java and other programming languages exists. OpenCCM relies on these specifications to compile a component interface into Java classes and methods. Likewise, we relied on

[1] We anticipate that the checker will be implemented completely within Kaveri as a collection of parametric query checks by the time camera ready copy is due for ASE.

these translations steps in step (2) of the case studies.
(4) All of the selected components were implemented using the monolithic implementation model since the version of OpenCCM used to implement these components did not support segmented implementations. Similarly, all components used a sequential execution model.

The results from these case studies along with our observation indicate that our approach is automatable and robust even when dealing with the complexity of industrial strength component models and middleware infrastructure. The results indicate that the concerns pertaining to *programming model* and *behavioral specifications* (both mentioned in Section 2.2) can be addressed by our approach. The use of concurrency-aware dependences addresses the *execution models* concern while the refinement-based approach with well-defined realization rules between refinements for the sake conformance check remedy the *Black Box v/s White Box* concern.

6 Related Work

Various architecture description languages (ADL) have been developed to specify software architectures. Garlan et.al [6] developed a style-based approach for developing software architecture based on *Wright*, an architecture description language, along with the support for configuration rules that constrained the assembly of components. Later, Allen et.al [2] proposed an approach based on a variant of CSP to specify the behavior and, hence, the meaning of Wright specifications. In another effort, Luckham et.al [12] developed *Rapide*, an event-based concurrent OO language for prototyping system architectures, that supported partial-ordering based mechanism to capture component behavior and interactions via events in a concurrent setting. Unlike our effort, these efforts were focused on design and analysis of the system architectures.

Stafford et.al [17] proposed how to facilitate dependence analyses of software architectures to detect design anomalies and enable design comprehension. They also demonstrated that such analyses can be automated. Our effort is similar to this as it is based on influence, the dual of dependence, and motivated by the application of concept of program dependence to component-based systems. However, our effort differs as it is focused on enabling refinement-based implementation synthesis and conformance checking. In a similar effort, Zhao [18] introduced the notion of architectural slicing to extract sub-architectures from a given software architecture. This effort was motivated by program slicing (based on program dependence) and focused on architecture comprehension and reuse as opposed to our end purposes.

7 Conclusion and Future Work

Based on our experience, we identified key concerns in Section 2.2 that affect model driven development of component-based systems in terms of implementation synthesis and conformance checking. To address these concerns, we focused on the mechanism to specify the behavior of components. As a first step, inspired by previous efforts, we developed a layered approach that relied on refinement of behavioral specifications based on the notion of influence to enable a smooth transition from component interface specifications to implementations. The use of refinement also enabled conformance checking of the implementation against the specification.

Our initial experiments indicate that our approach enables conformance checking of components based on component models that are as feature-rich as CCM. They also indicate that the generation of mappings between various refinements of influence specification can be automated.

As future work, we believe it will be straightforward to extend our Java slicer to automatically infer model-based dependence information from existing code-level implementations. Our interaction with industrial partners has shown that this capability is very important for obtaining leverageable model information in situations where engineers are incorporating previously developed components or where they are reluctant to devote energy to writing such model level specifications directly. In

addition, we plan to evaluate how our approach applies to components that use a concurrent execution model and segmentation in their implementations. Finally, we would like to investigate how adding additional layers to our refinement structure to capture further information about influence ordering would facilitate synthesis of business logic implementations (e.g., control flow skeletons) from model level specifications.

References

[1] Sae architecture and analysis design language. This specification is available at http://www.aadl.info/.

[2] R. Allen and D. Garlan. A formal basis for architectural connection. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 6(3):213–249, 1997.

[3] L. O. Anderson. *Program Analysis and Specialization for the C Programming Languages* . PhD thesis, DIKU, University of Copenhagen, DIKU, University of Copenhagen, Universit et sparken 1, DK-2100, Copenhagen ∅, Denmark., May 1994.

[4] J. Ferrante, K. J. Ottenstein, and J. O. Warren. The Program Dependence Graph and It's Use in Optimization. *ACM Transaction of Programming Languages and Systems*, 9(3):319–349, July 1987.

[5] M. A. Francel and S. Rugaber. The Relationship of Slicing and Debugging to Program Understanding. In *Proceedings of the 7th IEEE International Workshop on Program Comprehension (IWPC'99*, pages 106–113, 1999.

[6] D. Garlan, R. Allan, and J. Ockerbloom. Exploiting style in architectural design environments. In *Proceedings of SIGSOFT'94: The Second ACM SIGSOFT Symposium on the Foundations of Software Engineering*. ACM Press, December 1994.

[7] J. Hatcliff, J. C. Corbett, M. B. Dwyer, S. Sokolowski, and H. Zheng. A Formal Study of Slicing for Multi-threaded Programs with JVM Concurrency Primitives. In *Proceedings on the 1999 International Symposium on Static Analysis (SAS'99)*, Lecture Notes in Computer Science, Sept 1999.

[8] J. Hatcliff, W. Deng, M. B. Dwyer, G. Jung, and V. P. Ranganath. Cadena: An Integrated Development, Analysis, and Verification Environment for Component-based Systems. In *Proceedings of the 2003 International Conference on Software Engineering (ICSE'03)*, May 2003. This software is available at http://cadena.projects.cis.ksu.edu.

[9] G. Jayaraman, V. P. Ranganath, and J. Hatcliff. Kaveri: Delivering Indus Java Program Slicer to Eclipse. In *Proceedings of the Fundamental Approaches to Software Engineering, FASE 2005*. Springer-Verlag, April 2005. This software is available at http://indus.projects.cis.ksu.edu.

[10] J. Krinke. Static Slicing of Threaded Programs. In *Proceedings ACM SIGPLAN/SIGFSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE'98)*, pages 35–42, Montreal, Canada, June 1998. ACM SIGPLAN Notices 33(7).

[11] Y. A. Liu, T. Rothamel, F. Yu, S. D. Stoller, and N. Hu. Parametric regular path queries. In *Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation (PLDI'04)*, pages 219 – 230. ACM, ACM Press, 2004.

[12] D. C. Luckham, J. J. Kenney, L. M. Augustin, J. Vera, D. Ryan, and W. Mann. Specification and analysis of system architecture using rapide. *IEEE Transantions in Software Engineering*, 21(4):336–355, April 1995.

[13] B. A. Nejmeh. NPATH: A measure of execution path complexity and its applications. *Communications of the ACM*, 31(2):188 – 200, February 1988.

[14] OpenCCM, a Java implementation of CCM. This software is available at http://openccm.objectweb.org.

[15] A. Podgurski and L. Clarke. A formal model of program dependences and its implications for software testing, debugging, and maintenance. *IEEE Transactions on Software Engineering*, 16(8):965–979, 1990.

[16] V. P. Ranganath, T. Amtoft, A. Banerjee, M. B.Dwyer, and J. Hatcliff. A New Foundation For Control-Dependence and Slicing for Modern Program Structures. In *Programming Languages and Systems, Proceedings of 14th European Symposium on Programming, ESOP 2005*. Springer-Verlag, April 2005. Extended version is available at http://projects.cis.ksu.edu/docman/?group_id=12.

[17] J. A. Stafford and A. L. Wolf. Architecture-level Dependence Analysis for Software Systems. *International Journal of Software Engineering and Knowledge Engineering*, 11(431-452):4, August 2001.

[18] J. Zhao. Applying Slicing Technique to Software Architectures. In *Proceedings of the 4th IEEE International Conference on Engineering of Complex Computer Systems*, pages 87–98, 1998.