Analyzing Software Architecture Based on Statechart Semantics

Marcio Dias Marlon Vieira Debra Richardson
Information and Computer Science
University of California, Irvine
Irvine, CA 92697-3425 USA
{mdias, mvieira, djr}@ics.uci.edu

Abstract

High assurance architecture-based and component-based software development relies fundamentally on the quality of the components of which a system is composed and their configuration. Analysis over those components and their integration as a system plays a key role in the software development process. This paper describes an approach to develop and assess architecture and component-based systems based on specifying software architecture augmented by statecharts representing component behavioral specifications. The approach is applied for the C2 style and associated ADL and is supported within a quality-focused environment, called Argus-I, which assist specification-based analysis and testing at both the component and architecture levels.

1. Introduction

A current trend in software engineering is architecture-based and component-based software development, where the high-level structure and behavior of a large software system is specified and the system is configured of components and their connections. Quality assurance of such a system relies fundamentally on the quality of the components of which the system is composed and their configuration. Yet little quality assurance technology exists for this architecture-based and component-based software development paradigm.

To predictably and reliably build complex systems by composing components, components must be analyzed not only independently but also in the context of their connection to and interaction with other components. Analysis should be coordinated, therefore, at the software architecture level of abstraction, where components, connectors, and their configuration are better understood and intellectually tractable. Analysis at the level of system architecture may address such behavioral qualities as functional correctness, performance, and timing (e.g., allowable order of operations, real-time guarantees) as well as structural quality.

To support both structural and behavioral architecture-based analysis, not only must the structure of the architectural configuration and the components be specified, as is supported by all architecture descriptions languages (ADLs), but also the behavior of those components and their interactions must be described. Many ADLs fail to provide a mechanism for behavioral semantics. Here, we discuss augmenting ADLs with statechart for specifying component behavior. We have defined this integration for C2SADEL, the ADL for C2-style architectures, yet we believe the approach is more general and effective with a number of ADLs. We support this approach to architecture- and component-based software development within a quality-focused environment, called Argus-I, which provides a comprehensive toolkit facilitating iterative and evolvable analysis throughout architectural specification and implementation. Both structural and behavioral analyses are accomplished by a synergistic combination of static and dynamic techniques.

The remainder of this paper is organized as follows. Section 2 motivates our work, while Section 3 discusses related work in component description and architecture-based analysis. In Section 4, statecharts are described. The Argus-I environment and its analysis capabilities are described in Section 5, while an example is presented in Section 6. Finally, we conclude by discussing our philosophical approach as well as future work.

2. Motivation

Perry and Wolf define architecture as follows: "Architecture is concerned with the selection of architectural elements, their interactions, and the constraints on those elements and their interactions necessary to provide a framework in which to satisfy the requirements and serve as a basis for the design" [16]. In the years since this seminal paper, there has been much interest in software architecture among researchers, resulting in two notable contributions. First, the research community has converged on a set of three fundamental elements as forming the basis for discourse about architecture: the component (a unit of computation or data storage), the connector (an entity that facilitates communication between components) and the configuration (a topological arrangement of components and connectors) [11]. Second, the community has demonstrated the importance of architectural styles as embodiments of engineering experience, allowing identification and exploitation of structural and behavioral commonality among related applications [19]. Apart from these two contributions, most research in software architecture has focused on formal specification languages, called architecture description languages or ADLs, for describing architectures at a high level of abstraction [1, 5, 8, 10, 13, 20, 22], along with simple architectural analysis techniques – that is, primarily syntactic analysis applied in isolation to ADL models [11]. In fact, there has been relatively little research attempting to fulfill the original vision of Perry and Wolf, particularly work that provides means for analyzing an architecture to demonstrate that it indeed does provide a framework for satisfying requirements and serving as a dependable basis for design. This is one primary motivation for our work.

To support sophisticated architecture-based analysis, behavioral semantics must be considered. In spite of the importance of structural description at both architecture and component levels, behavioral specification enables more informative analysis. Although architecture-based analysis places stringent new requirements on architecture description techniques, it has numerous benefits. For one, it enables explicitly focusing on architectural defects rather than relying on other test strategies to detect these defects. Moreover, architecture-based analysis can begin much earlier in the development process than usual (after implementation and during system integration), thereby detecting defects early in the software lifecycle, when they are less costly to fix and more likely to be fixed unerringly. Furthermore, since an architecture description can be reused to develop multiple systems, the analysis costs, which are high relative to the rest of development, can be amortized across the family of systems.

Each ADL embodies a particular approach to architecture specification. All ADLs address structural architecture and component description, although there is a wide variation in the behavioral aspects each ADL is able to address, including: functional behavior, timing, allocation of resources, performance, fault-tolerance, and so on [2].

C2 [13] is a component- and message-based architectural style for constructing flexible and extensible software systems. A C2-style architecture is a hierarchical network of concurrent components linked together by connectors (or message routing devices) in accordance with a set of style rules. The ADL associated with C2, C2SADEL, provides the capability to specify component functionality in first-order logic, using invariants and operation pre- and post-conditions [12], but there is no provision for explicitly defining component behavior.

The lack of semantic definition in C2 for component behavior limits the prospects for enlightening analyses. To face this problem, we integrated statechart semantics to describe component behavior with the C2 architectural model. As proposed by Harel [6], statechart semantics extend basic finite-state automata with many features, including an instantaneous broadcast communication mechanism and the ability to specify timing constraints.

We believe that statecharts contribute much added value in this role. In particular, the formal semantic theory used for functional behavior description should be simple so as to help the software architect rather than hinder him or her with needless confusion. The statechart model is fundamentally graphical in form and hence visually appealing, yet unlike many graphical design notations, statecharts have a precise formal semantics [6]. Furthermore, both *sequentiality* and *concurrency* of activities may be represented in a uniform way and *real time constraints*, such as for event and activity scheduling, are easily included. Another strong reason is that as part of the Unified Modeling Language (UML) standard defined by Object Management Group (OMG) for object-oriented modeling [15], the statechart model is well accepted in industry. This facilitates a smooth transition for designing and implementing components as object-oriented classes. For all of these reasons, we chose statecharts as the modeling notation to integrate with C2, thereby providing a rich abstraction to describe software architecture.

While formal statechart semantics defines component behavior, C2 semantics define the topology and relations among components. Architectural behavior is characterized in terms of significant events (messages) that take place when components are in determined states. A state defines a component's behavior by how it reacts to certain events – the events it produces that pass through the architecture. The combined abstraction allows us to promote sophisticated analysis.

3. Related Work

Our work is related to other ADLs that describe component behavioral semantics and most closely to the analysis capabilities provided for their architectural formalism. Here, we overview related work in both areas.

3.1. Component Specification

We observe that analysis tools for most existing ADLs tend to view architectures statically and current support for dynamic modeling and analysis is scarce. The ability to perform more sophisticated analysis of software architecture directly depends on the ability to model behavior. To this end, ADLs have employed different specification mechanisms. ACME [5] and UniCon [20] allow component behavioral information to be specified in property lists, but these have no pre-defined semantics. MetaH [22] allows specification of component implementation semantics with path declarations that describe sequencing behaviors of objects. Wright [1] defines component semantics using CSP as the underlying model. In Rapide [9], each component specification has an associated *behavior*, which is defined via state transition rules that generate partially ordered sets of events (posets). Darwin [10] uses the ð-calculus as its underlying semantic model, whereby a system is described as a collection of independent processes that communicate via named channels. In more recent work [3], Darwin provides for modeling component behavior by labeled transition systems (LTSs) described in the FSP specification language. A system is modeled as a collection of LTSs, which are interacting finite-state machines.

It is difficult to argue whether the statechart model is better than CSP, posets, the ð-calculus, or LTSs to specify component behavior. Statecharts do fit this task well and have many advantages, including those described above, especially wider use and acceptance in the software industry.

3.2. Architecture-based Analysis

Analysis of architectures may be performed statically, without execution, or dynamically, at runtime; certain types of analysis can be performed both statically and dynamically. Here, we describe some analysis techniques that make use of architectural information and are related to our work.

Dynamic architecture-based analysis is concerned with demonstrating the software architecture's run-time behavior in response to selected inputs and conditions. Examples of dynamic analysis are simulation, testing, debugging, and assertion checking. Dynamic analysis techniques require "executing" an architecture in one of two ways: (1) the system implemented based on the architecture is run, or (2) the architecture itself is simulated. Certain analyses, such as reliability assessment, are more meaningful or only possible on the implementation. Such techniques are limited, however, since they are not applicable until the system has been implemented. Performing analysis on the architecture itself allows early defect detection, which has been shown much more cost-effective than revealing architectural defects after they have fully impacted the system implementation. Dynamic analysis is a necessary part of software verification, because some qualities are better assessed using dynamic techniques and only dynamic techniques can analyze in the operational environment.

Due to the complexity of software systems, dynamic analysis alone is insufficient to verify architecture functionality – that is, the architecture is fully functional or free of defects. In fact, all dynamic analysis techniques rely on sampling [25], which means that correctness cannot be verified. Static techniques are crucial, therefore, to offset the limitations of dynamic analysis. Static architecture-based analysis is concerned with suggesting problems or finding defects by examining a software architecture specification without execution. Examples of static analysis are data flow, dependence and reachability analysis, and model checking. Static analysis techniques are capable of performing consistency and completeness checks over a software architecture, demonstrating whether certain properties are satisfied (such as liveness and safety properties), and assessing certain concurrent and distributed aspects of an architecture (such as the potential for deadlock or starvation). Architectures can also be analyzed statically for adherence to design heuristics and style rules.

3.2.1. Architecture simulation. Simulation is a powerful tool for dynamically analyzing complex systems prior to their implementation. Simulation is useful for a wide variety of purposes, including detecting defects, optimization, or simply system understanding. Rapide [9] supplies this capability. The Rapide simulator runs a user-defined test case over the architecture, generating a trace of events together with the causal event history and their timing. These traces may reveal defects in dynamic behavior not easily exposed by static analyses.

3.2.2. Architecture-based dependence analysis. Dependence between pairs of elements in an architectural configuration can be analyzed either statically or dynamically. The transitive closure of direct dependence relationships creates sequences of dependencies, or indirect dependence, similar to program slices [24]. Aladdin [21] is a static dependence analysis tool that identifies direct relationships based upon input/output, temporal, state-based, and causal dependence and implements a technique called chaining for indirect dependence relationships. Aladdin has been implemented for Rapide and ACME.

3.2.3. Architecture-based model checking. Model checking is a static analysis technique that constructs a finite model of the system (e.g., from the composite component descriptions according to the architecture configuration) and checks it against a set of desired properties – standard qualities or specific system requirements. Wright [1] uses FDR [4] (a commercial specification-checker for CSP) as the core of its consistency checking tool. Darwin applies the compositional reachability analysis technique in TRACTA [3] to analyze behavior of concurrent and distributed systems. The LTS of a complete system, computed from those of its subsystems, is model checked against desired properties.

3.2.4. Architecture conformance testing. Testing the system for conformance to its architecture helps to ensure that architectural drift has not occurred during system implementation and evolution. To ascertain architecture conformance [8], both the aspects of

the architecture to be tested and the expected behavior must be identified. This can be done via an architecture conformance oracle consisting of a trace specification of expected events and their causal and timing relationships. Using such an oracle requires a mapping between elements in the architecture specification and those in the implemented system. Rapide provides event pattern mappings to define how a system is related to a reference architecture. The mapped executions are checked for conformance to the reference architecture's constraints.

4. Modeling Component Behavior with Statecharts

There are several works on statechart, and most of them have tried to understand and modify the semantics of the original Harel statecharts [6]. In [26], David Harel explains the statecharts semantics for situations where some overlapping states exist, which was not included in the initial statecharts idea. In [28], Beeck presents a useful comparison of all the major variants of statecharts developed till 1994, saying that almost every statecharts variant models concurrency by interleaving. In [27], Harel again tries to explain the statecharts semantics that they originally implemented in STATEMATE, and which they have modified since then to accommodate users' demands. Lilius and Porres [29] discuss the formalization of UML statecharts, presenting a clear description of semantics for statecharts in UML. Such a description is needed as a reference model for implementing tools for code generation, simulation, and verification. We extended UML statecharts [15] to model behavioral semantics for architectural components. A statechart represents the sequences of states that a component goes through during its life in response to events (incoming request or notification messages). A transition is triggered by an event under certain conditions.

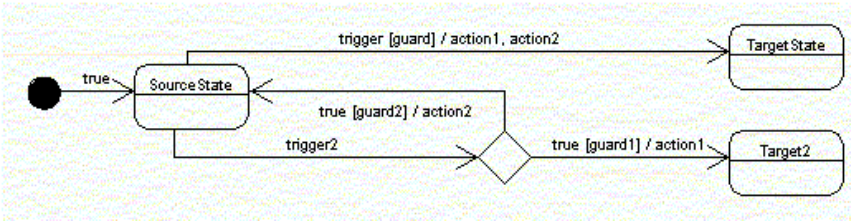

Figure 1. Simple Statechart

The basic elements of a statechart are *states*, *composite states*, and *transitions*. The following paragraphs will introduce how these elements are used in our approach; refer to Figure 1.

A *state* represents a situation in the life of a component when it satisfies some conditions, may perform some actions, and/or waits for some events. An *initial state* appears in the top-level of a statechart; the transition from an initial state may be labeled with an action generated upon component creation; otherwise, it must be unlabeled. A *final state* represents the completion of activity in the enclosing state and triggers a transition on the enclosing state labeled by the completion event, if such a transition is defined. Reaching the final state firing an action with the keyword *_Exit* means that the application has finished and the system terminates.

A *composite state* decomposes a state into two or more concurrent substates or into mutually exclusive disjoint substates. A state may only be decomposed in one of these two ways. Any substate of a composite state can be decomposed in either way.

A *transition* is a relationship between two states indicating that when the specified *trigger* event occurs, the component in the source state to the target state and performs the associated *action* provided that the specified *guard* is satisfied. When this occurs, the transition is said to "fire."

An *event* is an occurrence that may trigger a state transition. Events are related to the component interface, more specifically to the messages that arrive at the component and are understood by the interface. For example, in the C2 context, events are requests and notifications. If a message reaches a component and does not trigger any transition, it is

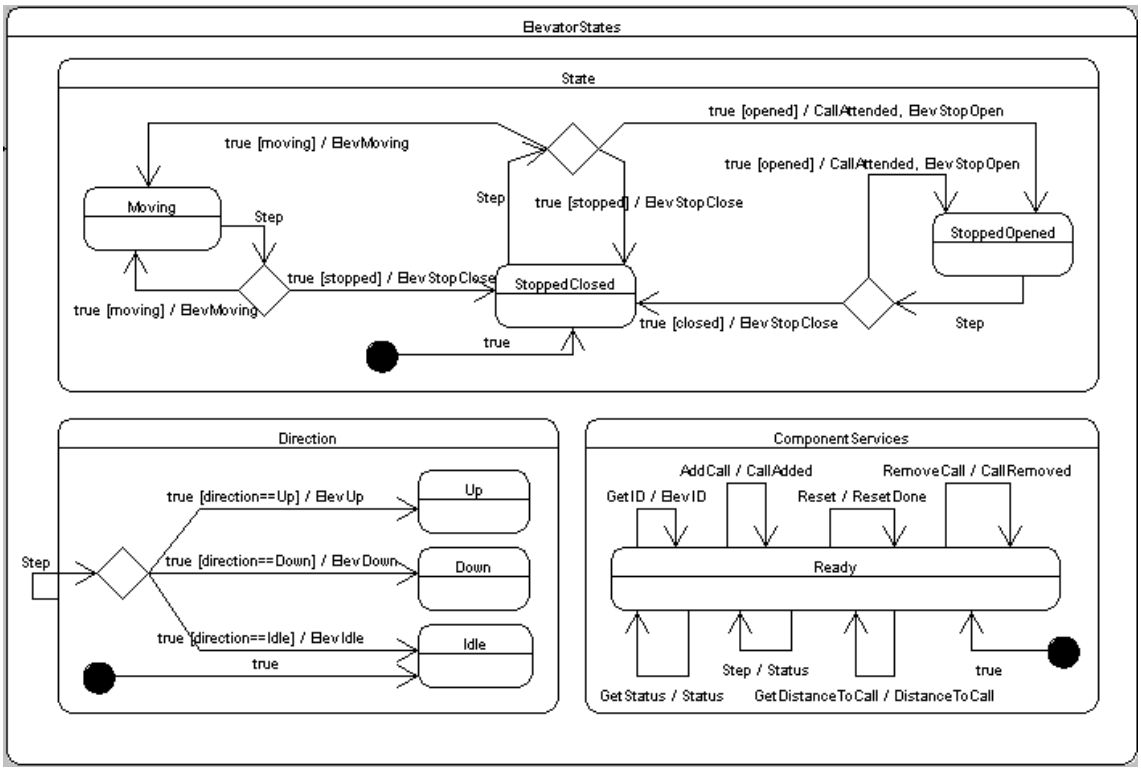

Figure 2 Elevator Component Statechart

ignored by the statechart. If it triggers more than one transition within the same sequential region, only one will fire. An event name may appear more than once per state if the guard conditions are different; the choice may be nondeterministic if guard conditions are not specified. Predefined triggers denoted by "*true*" and "*else*" may be defined for a transition: *true* enables the transition to be taken immediately when the component reaches the source state; *else* enables the transition if there is no transition from the current state to be taken after an event arrives at the component.

A *guard* condition is a boolean expression written in terms of parameters of the triggering event, and attributes or pre/post conditions on the component. The guard condition may also involve tests of concurrent states of the current machine, or explicitly designated states of some reachable component (for example, elevator "*in* Moving" and "*not in* Idle"). A simple transition may be extended to include a tree of *branches*, or decision symbols (indicated by the diamond). This is equivalent to a set of individual transitions, one for each path through the tree, whose guard condition is the "and" of all of the conditions along the path. The predefined guard "*else*" may be used for at most one outgoing transition; this transition is enabled if all the guards labeling the other transitions are false. The "*exception*" guard used to define an internal error condition not identifiable with a logical expression.

An *action* is executed if and when a transition fires. It is written in terms of messages that the architectural component sends when the transition is fired. The action expression may be an action sequence comprising a number of distinct actions including actions that explicitly generate events, such as sending notification signals or invoking operations.

Figure 2 presents a statechart description for an elevator component (used in the larger context of an Elevator system in Section 6). There are three concurrent states: *Direction*, *State*, and *Services*. "*Services"* describes those services provided by the component that are independent of the current direction or movement. For instance, *CallAdded* is always sent by this component when it receives *AddCall*. *Direction* indicates the current elevator direction (*Up*, *Down*, or *Idle*). *Movement* indicates if the elevator is *Moving*, *Stopped* with the doors

Opened, or *Closed*. The initial state of an *Elevator* component (just after creation) is (*Idle*, *StoppedClosed*, *Ready*).

5. Argus-I

In considering the development of Argus-I, we investigated the suitability of various analysis techniques for software architectures so as to include a complimentary set. We knew in advance that we wanted the toolkit to analyze architectural elements as well as topology, which presented us with a broad range of techniques to consider. One outcome of this exploration is that Argus-I provides support for numerous analysis techniques, from type checking to model checking and simulation. A developer or analyst must pick and choose among these techniques by considering the risk associated with the system and the role of specific elements in higher risk functions. The magnitude of the analysis effort should be commensurate with this risk – that is, while all architectural components should be validated, more critical components should be subject to more thorough analysis. Likewise, the size and complexity of the software architecture is an important factor in establishing the appropriate level of architecture-based analysis.

Furthermore, we wanted specification and analysis to go hand in hand so as to permit the architecture to be incrementally and naturally checked. Argus-I was implemented using the framework developed by Argo/UML [18]. This environment integration between Argus-I and Argo/UML allows not only easy data exchange but also a more consistent and integrated design process, including support for both architecture and detailed object-oriented design activities.

5.1. Argus-I Process

Although Argus-I does not enforce any specific process, Figure 3 presents guidance to understand the interaction between the specification and analysis capabilities inside the Argus-I context.

The developer starts by specifying the architecture and its elements (components and connectors). This specification may occur in an incremental fashion through the creation of new items or the reuse/evolution of existing components. During this incremental specification, the developer should use the available analysis tools to incrementally assess the quality of the software architecture.

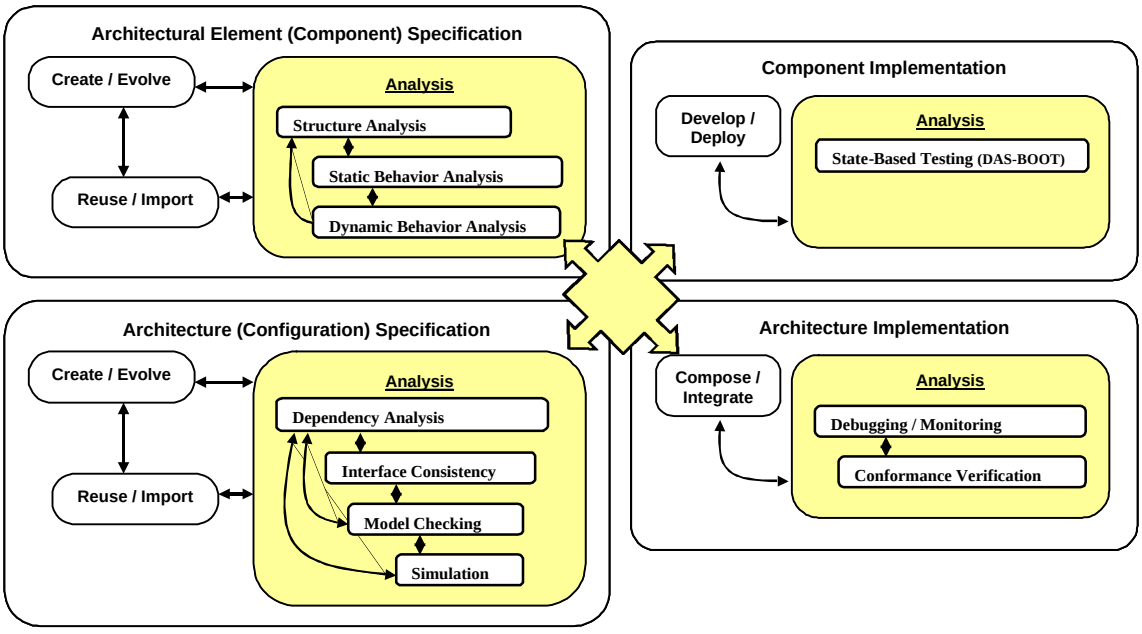

Figure 3. Argus-I Process

Specification analysis tools work at both component and architecture levels. At the component level, structural and behavioral analyses check component integrity. At the architecture level, dependence analysis, model checking, and simulation serve to evaluate the architecture specification.

Argus-I does not explicitly provide support for the coding activity. During implementation, however, developers should use the implementation analysis tools to incrementally verify not only each component individually but also whether components behave as desired when integrated with other components.

5.2. Architectural Element (Component) Specification Analysis

Argus-I provides capabilities to evaluate both structure and behavior for individual architectural components with *structural analysis*, *model checking* and *simulation* tools. These capabilities assist the software architect in improving component behavior independently of the configuration.

Structure Analysis

During structural analysis, a collection of `type checking` rules is applied over each component specification to determine whether each component is "well" structured. Any parts of a component not well specified are presented with an explanation of the rules violated. The type-checking rules generally embody common sense inference and consistency rules. For instance, interfaces must be mapped to an operation and operations must be mapped to an interface. In the current version, `type checking` rules are enforced during component and message type specification. Argus-I can export component specifications to C2SADEL.

Static Behavior Analysis

Argus-I analyzes component behavior statically by syntactic and semantic checking of statecharts (using critics from Argo/UML [17]) and reachability analysis. Aspects such as conflicting transitions (nondeterminism) and consistency between the component interface and the state machine are verified. Model checking is applied to further analyze the component specification, by translating the statechart specification into Promela (the input language of the SPIN verification system [7], using an approach similar to MOCES [14]). SPIN is then used to check the logical consistency of the statechart model against proprieties expressed in linear time temporal logic.

Dynamic Behavior Analysis

Dynamic behavior analysis consists of statechart simulation upon an event trace. The user selects a list of events to drive execution. The statechart engine executes and the user can follow the triggered transitions and actions to validate the statechart.

5.3. Architecture (Configuration) Specification Analysis

We built *dependence analysis* and *simulation* tools and integrated *model checking* into Argus-I for architectural analysis. These tools explore structural and behavioral dependence among components, validate the configuration through extensive simulations, and verify of specific properties using model checking. The results facilitate refining and correcting the architecture before the system is implemented.

Dependence Analysis

Dependence relationships at the architectural level arise from the connections among components and the constraints on their interactions. The core of an architectural style comes from how components/connectors are organized. For instance, the structural dependence among C2 components are determined by analyzing the characteristics of each component (its level in the architectural topology, notifications to which it will react and requests that it can send). These characteristics expose control and data dependence. In the C2 style, control dependence between two components is not explicit, because components are not aware of

which components with whom they interact. Control dependence exits, however, between a component that sends a message (request or notification) and any component that understands and potentially receives that message. Data dependence in the C2 style is identified by the parameters passed in a message by the component that defines the parameter to the component that uses it.

Interface Consistency

Performing structural analysis over component interfaces and the architecture configuration enables detection of architectural defects, such as, interface mismatch, missing interfaces and unused services.

Model Checking

Model checking can be performed at the architectural level as well as the component level. At the architectural level, Argus-I creates a "super statecharts model" integrating all components' statecharts (concurrently) subject to the constraints of the architectural configuration. The resultant model is converted to Promela, and SPIN is invoked to check desired properties.

Simulation

Simulation provides a sophisticated and accurate way for visualizing the functionality of a software architecture. By simulating the architectural model, it is possible to detect architectural defects, like a component that receives an unexpected event, or a state transition that should generate an action but does not. Based on simulation results, the user may improve the architecture.

Another capability provided by Argus-I's simulation tool is predicting system performance by how often each component, message, connector, or component interface is invoked for particular simulations.

Simulation is done by interpreting components' behavior while following the architectural topology – thus, all components' statechart engines run in parallel. The result is a trace, which can be visualized either graphically or textually. A trace may indicate an *error*, which occurs when a component during the simulation receives an unexpected event, or a *warning*, representing an anomalous event that is not definitely erroneous (for example, the trace presents a warning when a component accepted an expected message but no effect was produced).

5.4. Component Implementation Analysis

Argus-I carries out component implementation analysis thought state-based testing using DAS-BOOT [23]. Detailed explanation of DAS-BOOT is beyond the scope of this paper; due to space limitations we do not consider this activity in the example that follows.

State-based Component Testing

DAS-BOOT is a specification-based testing tool that verifies the behavior of a Java class based on and against a statechart diagram modeling how objects of that class should behave. DAS-BOOT receives as input a Java class to be tested, a state-based specification of the class behavior, and a FSM coverage criterion from which it produces as output test driver-oracles that automatically test the Java class against the component specification according to the coverage criterion.

5.5. Architecture Implementation Analysis

Architecture implementation analysis can be done in many different ways. We adopted a dynamic approach involving execution of the architecture implementation. Although our major concern is verifying conformance between architecture specification and implementation, the actual behavior of the components can be unpredictable. This unpredictability demands support for monitoring and debugging, providing finer control over the components and architecture during execution.

Debugging and Monitoring

Architecture-based debugging and monitoring acts at a higher level of abstraction than does typical program debugging to control and follow system execution. In the C2 style, this abstraction level is well defined by message exchange between components.

Debugging at the architectural level, the user can validate the sequence of events that occurs during execution by visualizing event traces for the architecture as well as for individual components. Following event traces, the user can gain better understanding of components and architecture misbehavior and, consequently, better reason about the cause of architectural defects.

Utility of this event monitoring is enhanced by enabling the user with further control over execution. Argus-I's debugger provides mechanisms to interrupt messages at pre-determined points ("breakpoints"), allowing the user to follow execution more carefully and control it interactively, and to modify, add or remove messages received by or sent from a component.

Argus-I's "component inspector" offers capabilities similar to "watch" and "object inspector" mechanisms present in program level debuggers. The component inspector enables the user to access the internal state of a component so as to understand and verify component behavior. The user can browse through the component data structure and modify data values.

Argus-I facilitates performance analysis by providing the user with statistical measures of component and interface usage, based on the number of events exchanged during the execution.

Conformance Verification

When verifying consistency between specification and implementation at the architectural level, both the conformance of the architectural topology as well as the conformance of each component and connector of the architecture must be considered.

Conformance to the architectural topology is based on dynamic configuration "enforcement" – i.e., the architecture is dynamically built based on information from the *specification* and its *mapping* to the implementation. This guarantees that the actual architecture topology is consistent with its specified configuration.

Conformance verification for components is dynamically performed with respect to two distinct (but associated) aspects: interface and behavior. As the names suggest, interface conformance means the implemented component's interface is consistent with the specified interface, while behavioral conformance means that component behavior satisfies the behavior specification (as embodied in the statechart). Checking for both is performed based on the messages exchanged and captured during monitored execution.

Conformance verification identifies all component interfaces not used during execution, as well as all used interfaces not specified. This verifies inconsistency/incompleteness in both directions, from specification to implementation and vice-versa. For those interfaces that were specified and used during execution, any mismatch between specification and implementation is identified, including message type, direction, and parameters.

Behavioral conformance verification is based on parallel execution of all components' statecharts. At runtime, each component instance has a statechart engine that executes based upon incoming messages. Incoming messages trigger transitions and outgoing messages are compared to the expected effects. At any time, the user can follow the execution history and check those messages that were responsible for each step, as well as those messages that were not expected/specified in the statechart.

Conformance verification requires not only monitored execution of the implementation but also mapping events in the implementation to the specification. This representation mapping must be defined by the user before verification; Argus-I provides support for defining the mapping. Implementation must be based on the C2 framework.

6. EXAMPLE

In this section, we illustrate the analysis capabilities of Argus-I within the context of an elevator control system. First, we briefly explain the problem context, and then present the architecture and components specification. Finally, analyses are performed and discussed.

6.1. Elevator Problem

The elevator system assigns elevator cars to attend users' calls, which are made from inside an elevator car or outside on a floor. For an inside call, the user is requesting a specific elevator to move to a particular floor and only the specific car is assigned to attend the call. For an outside call, the user is requesting an elevator to pick up at a particular floor and the system can assign any elevator to attend the call.

The component types used in this system are:

- *Elevator:* represents the model for one elevator car in the system; the number of such components determines the number of elevator cars.
- *ElevatorWindow:* represents the view for each car.
- *ControlWindow:* represents the control view for the building independent of any elevator car.
- *Scheduler:* assigns an elevator to attend a user call; if there is no scheduler, all elevator cars are assigned to an external call, and after it is attended, outstanding cars are unassigned.
- *Clock:* represents the timing of actions; a single clock component in the configuration means the system is synchronous; multiple clock components means the system is asynchronous.

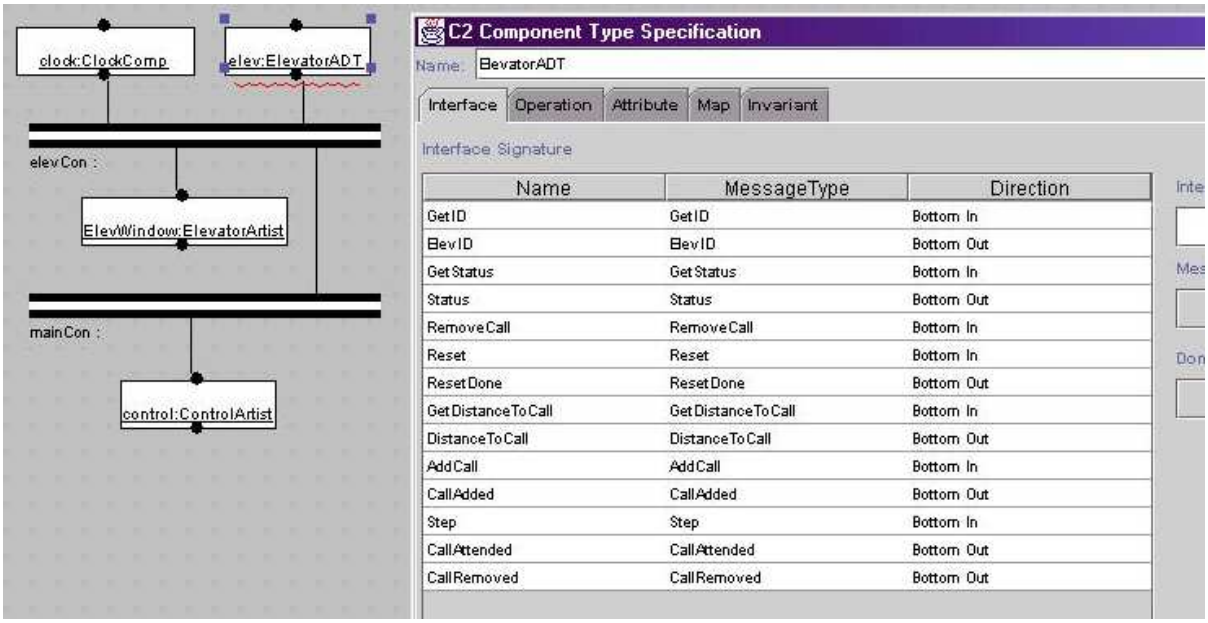

Figure 4. Architectural configuration for Elevator system and *Elevator* component interface specification

With these components, different architectural configurations can be specified and analyzed. Figure 4 shows the simplest architectural configuration in the background. This architecture is composed of one *ControlWindow* (lowest layer), one *ElevatorWindow* (middle layer), one *Elevator*, and one *Clock* (highest layer). Since there is only one elevator to attend all calls, the *Scheduler* is not necessary. Other configurations are also considered in this section and different analysis results are discussed.

In Figure 2 (in Section 4), the *Elevator* behavior is specified using statecharts, where concurrent states can be observed. Observe, for instance, that after receiving *AddCall* (trigger) in state (*StoppedClosed, Idle, Ready*), the Elevator component is supposed to send an event *CallAdded* and transition to state (*Moving, Up, Ready*).

6.2. Component Specification Analysis

Static structural analysis checks the consistency between component interface specification and the structural aspects of it's behavior specification. This ensures that all events referenced in a component's statechart are defined in the interface specification and that each interface defined there is used in the statechart. For instance, it ensures that the interface *AddCall* of the *Elevator* component has been used as a trigger in the statechart model that describes the component behavior.

Static behavioral analysis over the component employs model checking to determine whether a component's behavior always satisfies user-defined properties. For example, suppose the user wants to ensure that an *Elevator* cannot be in states *Moving* and *Idle* at the same time; this property is represented as a Promela never clause:

```
! [] ! (in_Moving && in_Idle)
```

Model checking returns *claim violated*, meaning that *Moving* and *Idle* are not mutually exclusive (the states can be active at the same time). This occurs because the guards used fail to consider the active states of the component in deciding which transition should fire.

6.3. Architecture Specification Analysis

Structural dependence analysis is one capability for analyzing the architectural specification. In the C2-style, components have dynamic interfaces, and the "physical" connections between components are not based on their interfaces. C2 components can even modify their interfaces "on-the-fly". Dependence analysis, in this context, determines the logical dependence between component interfaces. In an architecture configuration with two elevators and two clocks, for instance, dependences are shown graphically in Figure 5. *ControlWindow* depends on *Elevator* for the interfaces *AddCall*, *RemoveCall* and *CallAttended*. On the other hand, *ElevatorWindow* is found to depend on *Elevator* for the interfaces *AddCall* and *CallAttended* among others, but not for *RemoveCall*,. Although both *ControlWindow* and *ElevatorWindow* show the same connection to *Elevator* in the architectural configuration, further dependence analysis incorporating component structure and behavior determines that *ElevatorWindow* does not recognize *RemoveCall*.

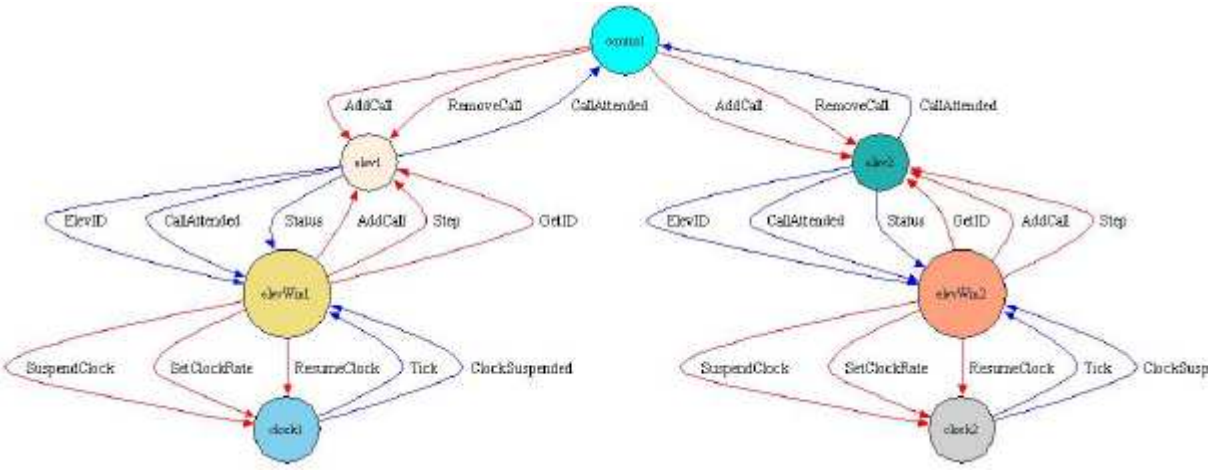

Figure 5. Dependence analysis of architecture with two elevators

Static structural analysis also identifies interfaces that are not used in the architectural configuration under analysis. For instance, *Elevator* provides an interface *DistanceToCall*, that returns the actual distance from the current state of the elevator to the call, considering not only the difference between floors but also the call direction and other calls to which the elevator must attend. This service is useful for an optimizing *Scheduler* that considers the actual distance before selecting an Elevator to attend to a call. A simpler *Scheduler*

component does not need this interface, and it would not be used in the architecture configuration.

At the architecture level, model checking provides more power and more verification options than at the component level. Now we can verify whether or not one component can reach a state based on the events generated by other components. For instance, suppose the user wants to ensure that the *Clock* component cannot be *deactivated* when the *Elevator* is *moving*. The following Promela never clause expresses this property:

```
! [] ! (Elevator_in_Moving && Clock_in_Deactivated)
```

This property is not violated, thus the desired behavior has been verified.

Architectural simulation is based on the combined statecharts for all components in the architectural configuration. Again, the user provides a sequence of events, this time for the entire system. Events are also generated by actions specified in the statecharts and their effect propagated through the architecture. The user can reason about the trace of events generated, such as that shown in the event graph in Figure 6. In this figure, we see the effect of component *Elevator2* receiving *RemoveCall* is to send *CallRemoved*, which is received by *ElevatorWindow2*, but this message is not understood, so we see a *warning arrow*. Simulation continues until there are no events left, at which point the simulator sends the event "*true*" for all components. After this, if no component generates an event, simulation stops. Otherwise, simulation continues as before. This enables determining whether the simulation completes (reaches a final state) or reaches a deadlock situation.

Architectural simulation also provides statistics about the number of events generated

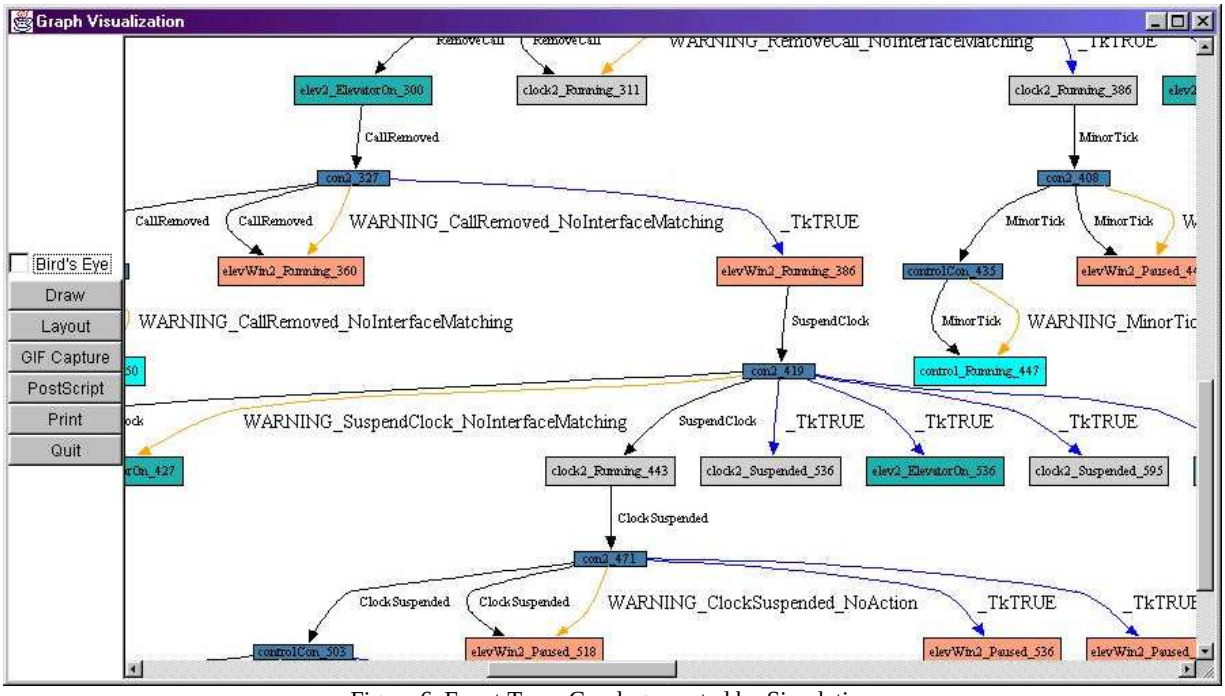

Figure 6. Event Trace Graph generated by Simulation

for each component and for each interface; this information is useful for predictive performance evaluation as well as determining where further analysis or optimization might be fruitful. For instance, after simulating an architectural configuration with two elevators, we might see the following percentage of usage for each component's interface:

ControlWindow	26.65%	ElevatorWindow (N)	13.71%
Elevator (N)	11.51%	Clock (N)	11.45%

With this information, we identify those components that may be time-critical or represent bottlenecks and, consequently, should be subject to further evaluation and

performance improvement. Here, special attention should be given to the *ControlWindow* component.

6.4. Architecture Implementation Analysis

Argus-I supports debugging at the architectural level during execution of the implemented system. The user can toggle breakpoints for incoming and outgoing messages in a component, and also remove, add or even edit messages (modify parameters) when execution is stopped. Figure 7 illustrates behavioral conformance verification during system execution; here, the user is viewing the *Elevator* component being verified against its behavioral specification in the context of the executing architecture. Verification confirms that the actual events the component has received (IN events in the upper left panel) and sent (OUT events in the lower left panel) are the same as those specified in the component's statechart. The current possible transition to fire is displayed in the upper right panel, and the history of fired transitions recognized from the actual events is displayed in the lower right panel.

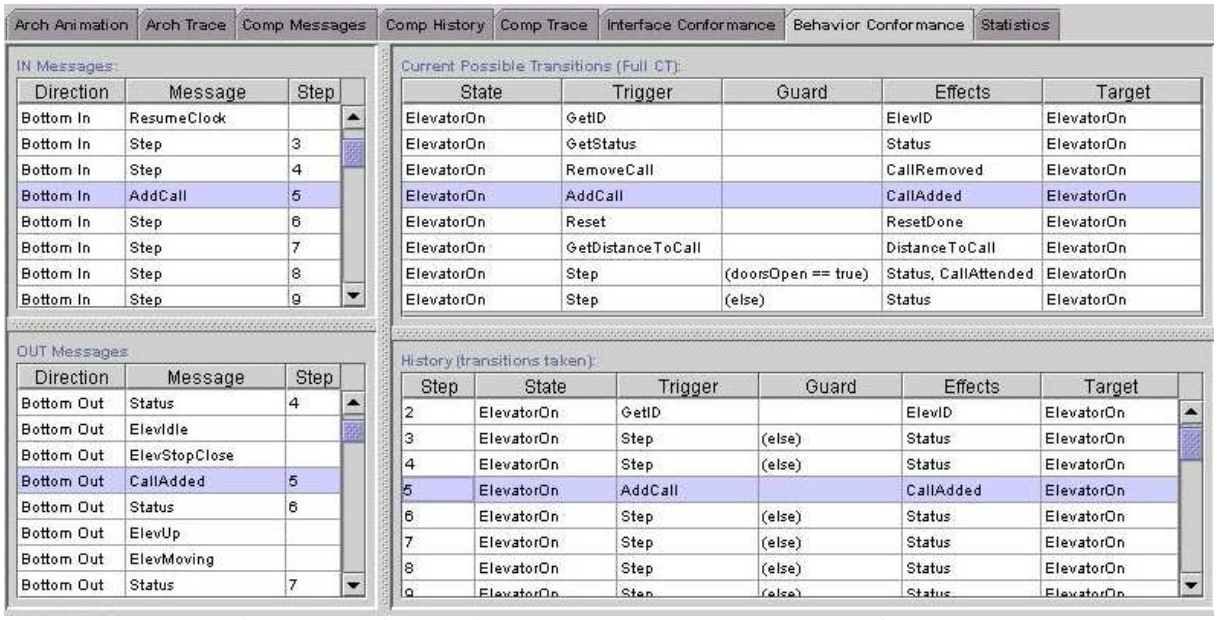

Figure 7. Conformance Verification of the *Elevator* component in the context of architectural execution

7. Conclusions

There is a noticeable gap between the *state-of-the-art* and the *state-of-the-practice* in architecture-based analysis techniques. While *art* (in the research community) is based on ADLs and their analysis tools, for the most part practice consists of specifying software architectures using widespread design models for which analysis capabilities are extremely limited.

There are basically two distinct approaches to reducing this gap: bringing *art-to-practice* (making software developers use ADLs for specification and their tools for analysis) or bringing *practice-to-art* (making architecture analysis techniques and tools more accessible to the developer, focusing them on more common and widespread design description methods).

Bringing *art-to-practice* would have the advantage that software architecture is better captured and specified with ADLs than with common design models. Moreover, ADLs and analysis tools are already available, although the analysis capabilities provided with ADLs are primarily syntactic in nature. The disadvantage of this approach, however, is that it requires time and major investment in training and incentive programs to become *practice*. A smooth technology transition (or introduction) is necessary.

Bringing *practice-to-art* would have the advantage of smoother transition, whereby developers could start using architecture-based analysis tools with common design models (e.g., UML) without too much effort. Clearly, the disadvantage here is that UML does not adequately represent software architecture concepts and some enhancement is required. Although some researches are addressing this issue [18], the problem remains that architecture-based analysis tools using UML descriptions are not yet available and must still be developed.

The work presented in this paper is a step in trying to reduce the gap by integrating *art* and *practice* in architecture-based analysis using architecture specifications based on both ADLs and UML statecharts. Additional work is still required in this integration to transition architecture-based analysis into a software development practice.

As future research directions, we are pursuing both approaches to reducing the *art-practice* gap. Following the first approach, we are considering the ACME interchange format [5] for specifying structural properties of the architecture, and integrating statecharts (and other UML models) to describe behavioral properties. This would then enable our architecutre-based analysis toolkit to support a variety of ADLs, each translated into ACME. Following the second approach, we are considering enhancements to UML that promote completely specifying the structural and behavioral properties of software architecture, and modifying our analysis toolkit for this purpose.

We continue also to consider supporting other types of analysis for architecture specification and implementation. For instance, we are investigating and developing further architecture dependence analysis capabilities, symbolic architectural simulation, and architectural integration test criteria and test generation. We believe that an integrated set of support capabilities for architecture and component specification and analysis will greatly enhance the quality of systems developed in the architecture-based and component-based software engineering paradigms.

8. Acknowledgments

This effort was sponsored by the National Science Foundation under grant number NSF ITR 0083099; and by the Brazilian Research Agency (CNPq) under scholarship grants number 200022/98-9 and 200258/98-2. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the U. S. or Brazilian Governments.

9. References

1. R. J. Allen, A Formal Approach to Software Architecture, Ph.D. Thesis. *Carnegie Mellon University, Technical Report Number: CMU-CS-97-144*, May 1997.
2. R. J. Allen, R. Douence, and D. Garlan, Specifying and Analyzing Dynamic Software Architectures. In Proceedings of Conference on Fundamental Approaches to Software Engineering, Lisbon, Portugal, March 1998
3. D. Giannakopoulou, Model Checking for Concurrent Software Architectures, PhD thesis, Department of Computing, Imperial College, 1998
4. Failures Divergence Refinement: User Manual and Tutorial. Formal Systems (Europe) Ltd., Oxford, England, 1.3 edition, August 1993.
5. D. Garlan, R. Monroe, D. Wile, Acme: An Architecture Description Interchange Language. In *Proceeding of CASCON'97*, November, 1997
6. D. Harel, A. Pnueli, J.P. Schmidt, and R. Sherman, On the formal semantics of statecharts. *In Proceedings of the 2nd IEEE Symposium on Logic in Computer Science, pages 54—64, Ithaca, New York*, June 1987.
7. G. J. Holzmann, The Model Checker Spin. IEEE Trans.on Software Engineering, 23(5): 279—295, May 1997.
8. D. C. Luckham, J. Vera, An Event-Based Architecture Definition Language. *IEEE Transactions on Software Engineering*, Vol 21, No 9, pp.717-734. Sep. 1995

9. D. C. Luckham, J. J. Kenney, L. M. Augustine, 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.
10. J. Magee, N. Dulay, S. Eisenbach, and J. Kramer, Specifying Distributed Software Architecture, in *Proc. Of the 5th European Software Engineering Conference (ESEC'95)*, Sitges, Spain, September 1995. *Lecture Notes in Computer Science 989*, pp. 137-153. W. Schäfer and P. Botella, Eds.
11. N. Medvidovic and R. N. Taylor, A Framework for Classifying and Comparing Architecture Description Languages. In Proceedings of the Sixth European Software Engineering Conference together with the Fifth ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE 97), pp. 60-76, Zurich, Switzerland, September 22-25, 1997.
12. N. Medvidovic, Architecture-Based Specification-Time Software Evolution, PhD dissertation, Information and Computer Science, University of California, Irvine, 1999
13. N. Medvidovic, D. Rosenblum, and R. Taylor, A Language and Environment for Architecture-Based Software Development and Evolution: *Proceedings of 21st International Conference on Software Engineering*, Los Angeles, CA, May 1999.
14. E. Mikk, Y. Lakhnech, M. Siegel, G. J. Holzmann, Implementing Statecharts in Promela/Spin, *In The Proceedings of Wift'98 Workshop* , 1998.
15. OMG Unified Modeling Language Specification (draft), Version1.3 alphaR2, January 1999 http://www.rational.com/uml/resources/documentation
16. D. E. Perry and A. L. Wolf, Foundations for the Study of Software Architecture. *ACM Software Engineering Notes*, 17(4):40-52, October 1992.
17. J. E. Robbins and D. F. Redmiles, Cognitive Support, UML Adherence, and XMI Interchange in Argo/UML In Proceedings of The First International Symposium on Constructing Software Engineering Tools (CoSET'99), Los Angeles, CA, USA, May 1999.
18. J. E. Robbins, D. F. Redmiles, Cognitive Support, UML Adherence, and XMI Interchange in Argo/UML, Information and Software Technology, to appear, 2000
19. M. Shaw and P. Clements, Toward Boxology: Preliminary classification of architectural styles. In Proceedings of Second International Workshop of Software Architecture (ISAW-2), San Fransisco (CA) USA, October, 1996
20. M. Shaw, R. DeLine, D. V. Klein, T. L. Ross, D. M. Young, and G. Zelesnik, Abstractions for Software Architecture and Tools to Support Them. *IEEE Transactions on Software Engineering*, vol. 21, no. 4, pp. 314-335, April 1995.
21. J. A. Stafford, D. J. Richardson, and A. L. Wolf, Aladdin: A Tool for Architecture-Level Dependence Analysis of Software Systems. *Technical Report CU-CS-858-98, Department of Computer Science, University of Colorado*, April 1998.
22. S. Vestal, MetaH Programmer's Manual, Version 1.09.Technical Report, Honeywell Technology Center, April 1996.
23. M. Vieira, M. Dias, and D. J. Richardson, DAS-BOOT: Design- and Specification-Based Object-Oriented Testing <http://www.ics.uci.edu/~rosatea/das-boot>, 1999
24. M. Weiser, Program Slicing, *IEEE Transactions on Software Engineering*, SE-10(4):352-357, July 1984
25. M. Young, and R. Taylor, Rethinking the Taxonomy of Fault Detection Techniques, in Proceedings of the 11th International Conference on Software Engineering, pp. 53-62, Pittsburgh, PA, May 1989
26. D. Harel, "On Statecharts with Overlapping", ACM Transactions on Software Engineering & Methodology, October 1992, pages 399-421.
27. D. Harel, "The STATEMATE Semantics of Statecharts", ACM Transactions on Software Engineering & Methodology, October 1996, pages 293-333.
28. M. von der Beeck, "A Comparison of Statecharts Variants", 3rd International Symposium of Formal Techniques in Real-time & Fault-tolerant systems, September 1994.
29. J. Lilius and I. Porres, The Semantics of UML State Machines, TUCS Technical Report No. 273, May 1999.