Suitability of the UML as an Architecture Description Language with Applications to Testing

by

Aynur Abdurazik

ISE-TR-00-01

February, 2000

Information and Software Engineering
George Mason University
Fairfax, Virginia 22030

Unlimited Distribution. *This report may be distributed without restriction. All partial or complete copies of this report must contain a copy of this page.*

Suitability of the UML as an Architecture Description Language with Applications to Testing

Aynur Abdurazik
Department of Information and Software Engineering
George Mason University
Fairfax, VA 22303

March 3, 2000

Abstract

Increasingly, very high level designs of large software systems are being described by software architectures. A software architecture expresses the overall structure of the system in an abstract, structured way. The Unified Modeling Language (UML) is widely used to express mid- and low-level designs of software, and recent proposals have been made to adapt the UML for use as an architecture design language (ADL). This research is looking into problems associated with creating system-level software tests that are based on architecture descriptions. This paper discusses issues with using the UML as an ADL, and general problems with then using the architecture descriptions as a basis for generating tests.

1 Introduction

Architecture Description Languages (ADLs) are common notations for representing *software architectures*. It has been claimed that ADLs lie at the conceptual intersection among requirements, programming, and modeling languages, and yet they are distinct from all three [BCK98]. ADLs are also said to enhance mutual communication among stake holders, and to support the analysis of early design decisions.

Some tool support exists for ADL based software development [AAM99]. However, despite their popularity among the research community, ADLs have yet to reach common practice. The reason is said to be that ADLs are not well integrated with common development methods [RMRR97].

The Unified Modeling Language (UML) is a language for specifying, constructing, visualizing, and documenting artifacts of software-intensive systems. UML, being a third-generation object-oriented modeling language, does so by providing a collection of views to capture different aspects of the systems to be developed. Some example views are *Use cases* to capture user requirements, *class diagrams* to capture static structure of objects, *collaboration* and *sequence diagrams* to capture dynamic interactions between objects and systems, and *package* and *deployment views*.

Use cases encode detailed system interactions of required system behavior, and are used in the understanding and analysis phases of development. Class diagrams encode the structure of objects that are proposed as a solution to provide required system behavior. Interaction diagrams model possible dynamic interaction between objects and can include both timing and message synchronization annotations. Package diagrams provide a way for designers to collect groups of objects for organizational purposes. Subsystem diagrams show system decomposition into functional units with clearly demarcated interfaces. Furthermore, UML is extensible through user defined stereotypes to cater to special purpose modeling needs. Many CASE tool vendors have already committed to supporting UML, and it has become a common notation for object-oriented design.

Bass et al. [BCK98] state that ADLs differ from modeling languages because the focus of modeling languages are the behavior of whole system rather than of their parts, whereas the primary mission of ADLs is to represent components and their connectivity. Consequently, Robbins et al. [RMRR97] extended UML for use with more than one ADLs, one for C2-style architectures and Wright. They did so by incorporating enough stereotypes in UML to encode the chosen ADL. Consequently, looking for generic extensions that are capable of incorporating any ADL has not been addressed. The objective of this proposal is to determine a well defined set of requirements for a language to be an ADL [BCK98] and an extension of UML that would incorporate them. We examine the software architectures and their description languages with this objective in mind.

1.1 Software Architecture

Software architectures have always existed, but explicitly stated to a lesser extent. They have been made explicit by researchers and practitioners due to the difficulties in cost estimation and process management, unique characteristics of programming in the large, and need for software reuse. Solutions to the above problems were the origins of software architecture concepts.

An architecture provides an early handle for achieving a system's quality attributes. Specific quality goals for a system are manifested in architectural decisions [BCK98]. Different architectures satisfy specific quality and behavioral requirements. Since an architecture is a key to quality, it follows that analysis of an architecture can (and should) be performed to evaluate it with respect to how well suited it is for its intended purpose. Analysis is only useful in the presence of clearly articulated goals for the artifact being analyzed. Software quality cannot be introduced late in a project, it must be inherent from the beginning, built in by design and requirements. An architecture-based assessment provides only one dimension of a system's quality characteristics and is a necessary, but not sufficient, component for evaluating the overall quality of a system.

Software architecture has been defined in various ways. We give a brief summary of five published definitions.

- **Perry and Wolf** [PW92]:
  Software architecture consists of three components: elements, form, and rationale. Elements are either processing, data, or connecting elements. Form is defined in terms of the properties of, and the relationships among, the elements – that is, the constraints on the elements. The rationale provides the underlying basis for the architecture in terms of the system constraints, which most often are derived from the system requirements.

  Some of the benefits that can be expected from software architecture as a major discipline are: (1) architecture as the framework for satisfying requirements, (2) architecture as the technical basis for design and as the managerial basis for cost estimation and process management, (3) architecture as an effective basis for reuse, and (4) architecture as the basis for dependency and consistency analysis.

- **Shaw and Garlan:**
  "The architecture of a software system defines that system in terms of computational components and interactions among those components" [SG96]. In addition to specifying the structure and topology of the system, the architecture shows the correspondence between the system requirements and elements of the constructed system, thereby providing some rationale for the design decisions. At the architectural level, relevant system-level issues typically

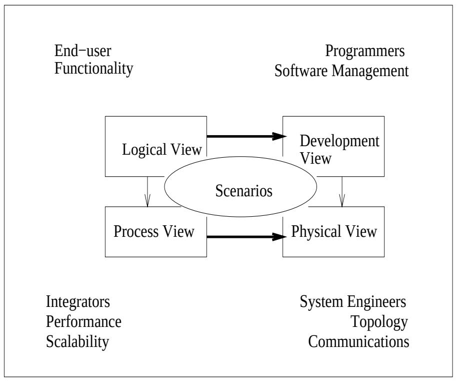


Figure 1: **The "4 + 1" View Model**

include properties such as capacity, throughput, consistency, and component compatibility. Architectural models clarify structural and semantic differences among components and interactions.

- **Bass, Clements, and Kazman** [BCK98]:
"The software architecture of a program or computing system is the structure or structures of the system, which comprise software components, the externally visible properties of those components, and the relationships among them."

- **Kruchten** [Kru95]:
Kruchten describes a software architecture by using a model composed of multiple *views* or perspectives. Figure 1 shows the model which is made up of five main views. The *logical view* is the object model of the design, the *process view* captures the concurrency and synchronization aspects of the design, the *physical view* describes the mapping(s) of the software onto the hardware and reflects its distributed aspect, and the *development view* describes the static organization of the software in its development environment.

  Kruchten used Perry & Wolf's definition of software architecture independently on each view. Each view has its elements (components, containers, and connectors), forms and patterns, and rationale and constraints that connect the architecture to some of the requirements. Different views can have different architectural styles, hence allowing the coexistence of multiple styles in one system.

- *Definition from UML Developers* [BRJ98]: Architecture is the set of significant decisions about
  - The organization of a software system
  - Selection of the structural elements and the interfaces by which the system is composed
  - Their behavior, as specified in the collaborations among those elements
  - The composition of these structural and behavioral elements into progressively larger subsystems
  - The architectural style that guides this organization: the static and dynamic elements and their interfaces, their collaborations, and their composition

Software architecture is not only concerned with structure and behavior, but also with usage, functionality, performance, resilience, reuse, comprehensibility, economic and technology constraints and trade-offs, and aesthetic concerns.

Architectural style defines a set of general rules that describe or constrain the structure of architectures and the way their components interact. Styles are a mechanism for categorizing architectures and for defining their common characteristics. New architectures can be defined as instances of specific styles. Garlan and Shaw [SG96] identified the following architectural styles: Pipes and Filters, Object-Oriented, Event-Based, Domain-Specific, Layered, Repositories, Rule-Based, Process Control (Feedback), Distributed, Main program/subroutines, State-Transition based, and Heterogeneous.

1.2 Architecture Description Languages

Architecture description languages (ADLs) are tools for formally representing the architectures of systems. We have a number of ADLs that vary widely in terms of the abstractions they support and analysis capabilities they provide. This section surveys the characteristics of ADLs in terms of the classes of systems they support, the inherent properties of the language themselves, and the process and technology support they provide to represent, refine, analyze, and build systems from an architecture. Then we discuss what constitutes an ADL.

Eleven existing ADLs are described:

1. **Rapide**: Rapide is an event-based, concurrent, object-oriented language specifically designed for prototyping system architectures [LKA+95]. Rapide allows architectural designs to be simulated, and has tools for analyzing the results of those simulations. The primary design criteria for Rapide are (1) to provide *architecture constraints* that permit system architectures to be expressed in an executable form for simulation before implementation decision are made, (2) to adopt an *execution model* that captures distributed behavior and timing as precisely as possible, (3) to provide *formal constraints* and *mappings* to support constraint-based definition of reference architectures and testing of systems for conformance to architecture standards, and (4) to address some of the issues of *scalability* involved in modeling large system architectures.

   In Rapide, a system component consists of two separate parts: an interface defining those features through which it interacts with other components, and a module that either encapsulates an executable prototype of the component, or hierarchically defines the component as an architecture of other components.

   Rapide consists of five sub-languages: (1) the *type language* describes the interfaces of components, (2) the *architecture language* describes the flow of events between components, (3) the *specification language* describes abstract constraints on the behavior of components, (4) the *executable language* specifies executable modules, and (5) the *pattern language* describes patterns of events.

2. **Darwin:** Darwin is a language for describing software structures that has been around, in various syntactic guises, since 1991. Darwin encourages a component- or object-based approach to program structuring in which the unit of structure (the component) hides its behavior behind a well-defined interface. Programs are constructed by creating instances of component types and binding their interfaces together. Darwin considers such compositions also to be types and hence encourages hierarchical composition. The general form of a Darwin program

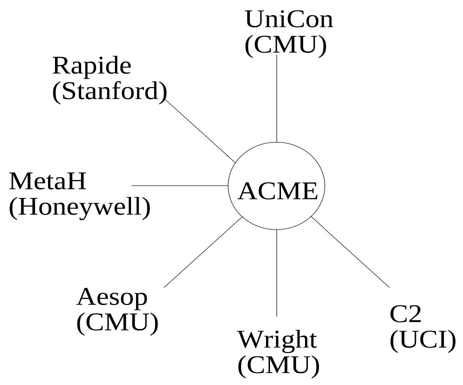


Figure 2: **ACME and Other ADLs**

is therefore a tree in which the root and all intermediate nodes are composite components; the leaves are primitive components encapsulating behavioral as opposed to structural aspects.

3. **Wright:** Wright supports the specification and analysis of interactions between architectural components [AG94]. The primary purpose of Wright is to analyze the interconnection behavior. Wright uses a subset of CSP [Hoa95] to provide a formal basis for specifying the behavior of components and connectors, as well as the protocols supported by their interface elements. In Wright, components are computation elements with multiple ports. A port is a logical point of interaction between component and its environment. A port defines the expectations of a component. The computation of a component describes the relationship between ports.

4. **Aesop:** Aesop [Gar95] supports the use of architectural styles.

5. **ACME:** ACME is developed as a joint effort of the software architecture research community as a common interchange format for architecture design tools. ACME provides a structural framework for characterizing architectures, together with annotation facilities for additional ADL-specific information. This scheme permits subsets of ADL tools to share architectural information that is jointly understood, while tolerating the presence of information that falls outside their common vocabulary. Figure 2 shows the ADLs that ACME [GMW97] can translate into each other.

6. **UniCon:** UniCon (language for UNIversal CONnector support) [Zel96] is an architecture description language (ADL) that describes software architectures in general. UniCon is organized around two symmetrical constructs: components and connectors. Components represent loci of computation and data in a software system. They are used to organize the computation and data into parts that have well-defined semantics and behaviors. Connectors represent classes of interactions among the components. They are used to mediate component interactions. Both components and connectors have a specification part and an implementation part. With proper set of primitive components, an architecture that is described in UniCon may be executable. UniCon supports external analysis tools. UniCon has a high-level compiler for architectural designs that support a mixture of heterogeneous component and connector types.

7. **MetaH:** MetaH [BEJV93] is intended to support analysis, verification, and production of real-time, fault-tolerant, secure, multi- processing, embedded software. MetaH allows no functional specification beyond a simple naming of inputs, shared objects, and outputs. Instead, MetaH captures connectivity information and behavioral information relevant to real-time scheduling, fault-tolerance, security, and scalable multiprocessing. MetaH is intended to be used in conjunction with other specialized tools, languages and library facilities that specify component functionality.

8. **C2**: C2 or C2SADL [MT97, EM99, AAM99] is an architecture description language designed for C2 style architectures. C2 supports the description of user interface systems using a message-based style. A C2 style architecture consists of three parts: components, connectors (buses), and their configuration. Connectors transmit messages between components. Each component has two connection points, a "top" and a "bottom". The top (bottom) of a component can only be attached to the bottom (top) of one connector. Components maintain state, perform operations, and exchange messages with other components via the top and bottom interface points. Inter-component messages are either *requests* for a component to perform an operation, or *notifications* that a given component has performed an operation or changed state.

   A C2 component consists of two main internal parts. An *internal object* stores state and implements the operations that the component provides. A *dialog specification* maps from messages received to operations on the internal object and from results of those operations to outgoing messages. Each component may be attached to at most one connector at the top and one at the bottom. A connector may be attached to any number of other components and connectors. request messages may only be sent "upward" through the architecture, and notification messages may only be sent "downward".

   C2SADL specifies architectures in three parts: component types, connector types, and configuration (topology). C2 specifies a component type with an invariant and sets of services a component provides and requires. A service consists of an interface and and operation. A single operation may export multiple interfaces. Invariants and operations (with their pre- and post- conditions) are specified as first-logic expressions. A component may be subtyped from another component, using heterogeneous subtyping that preserves the supertype component's naming, interface, behavior, implementation, or a combination of them.

9. **ROOM:** The Real-Time Object Oriented Modeling (ROOM) language combines a variant of component (actor) diagrams with another variant of state transition diagrams. It thereby fulfills the main requirements for an ADL given by Shaw and Garlan [SG96], but it offers no equivalent for UML's object constraint language or its package diagrams. ROOM offers two types of diagrams. *ROOM actor diagrams* describe the hierarchical decomposition of a software system into its components as well as all possible connections (communication channels) between these components. *ROOM charts* on the other hand are a variant of hierarchical state transition diagrams derived from StateCharts [Har87]. Any ROOM chart has an equivalent UML state transition diagram.

   A ROOM actor diagram defines the internal structure as well the external interfaces of a single component (class). The interfaces of different components, so-called *ports*, are bound to each other via binary *connectors*. Therefore, ROOM uses a variant of the "component-port-connector" model, allowing for the component architecture.

10. **SADL:** SADL is intended for the definition of software architecture hierarchies that are to

be analyzed formally. The SADL language can be used to specify both the structure and the semantics of an architecture. SADL support for explicit mappings between architectures, generic architectures, architectural styles (including well-formedness constraints), and architecture refinement patterns that provide routine solutions to common design problems. SADL is programming language independent, but can be tailored to model programs in most conventional programming languages. The advantage of SADL over ADLs is that SADL provides a formal basis for architectural refinement.

11. **ControlH:** ControlH is used to capture high-level specifications for real-time guidance, navigation and control systems [EJ93]. ControlH allows the modeling and analysis of continuous, time-varying signals. The predefined data types and operations, the syntax, and the semantics of the language have been tailored for that specific domain. ControlH relies heavily on tool support including Ada generator.

Bass et al. [BCK98] defines the following set of requirements for a language to be an ADL:

- The ADL must be suitable for communicating an architecture to all interested parties. All of the architecture's structures must be available through the ADL, including a range of dynamic and static structures. Components and connectors and their types must be identified in each of the structures, and the level of granularity of the information must be customizable for the renders of the architecture.

- An ADL must support the tasks of architecture creation, refinement, and validation. It must embody rules about what constitutes a complete or consistent architecture.

- An ADL must provide the ability to represent (even if indirectly) most of the common architectural styles.

- An ADL must have the ability to provide structures of the system that express architectural information but at the same time suppress implementation or non-architectural information.

- The ADL must provide a basis for further implementation. It must be possible to add information to the ADL specification to enable the final system specification to be derived from the ADL.

- If the language can express implementation-level information, it must contain capabilities for matching more than one implementation to the architecture-level structures of the system. That is, it must support specification of families of implementations that all satisfy a common architecture.

- An ADL must support either an analytical capability, based on architecture-level information, or a capability for quickly generating prototype implementations.

Problems with ADLs:

- ADLs introduce specific architectural assumptions, which can conflict with ones embodied in the existing middle-ware.

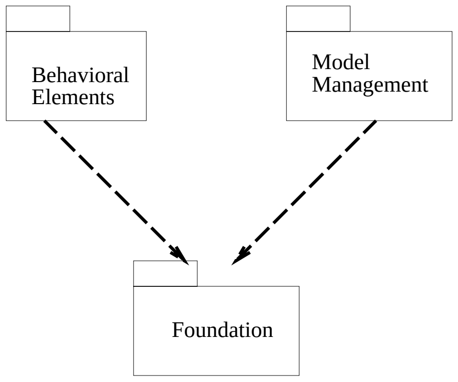


Figure 3: **UML Metamodel Top Level Packages**

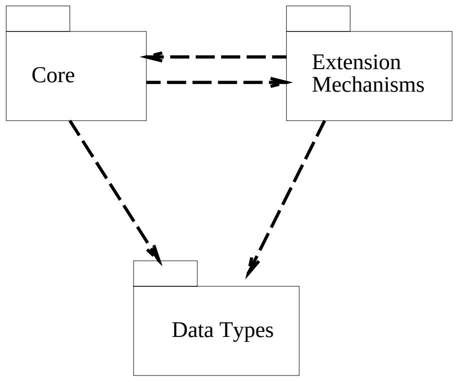


Figure 4: **Foundation Packages**

1.3 The Unified Modeling Language

The Unified Modeling Language (UML) is a language for specifying, visualizing, constructing, and documenting the artifacts of software systems, as well as for business modeling and other non-software systems. The UML represents a collection of the best engineering practices that have been found to be successful in the modeling of large and complex systems [Obj99].

The architecture of the UML is based on a four-layer meta-model structure: user objects, model, metamodel, and meta-metamodel.

The UML metamodel is a logical model, it is described in a semi-formal manner using abstract syntax, well-formedness rules, and semantics. The complexity of the UML metamodel is managed by organizing it into logical packages. Figure 3 shows the top level packages of UML metamodel. Figure 4 and 5 show the further decomposition of Foundation and Behavioral Elements packages. Model Management package is omitted because of space consideration.

In UML, the complex systems are approached through a small set of nearly independent views of a model. The UML defines the following graphical diagrams in terms of the views of a model:

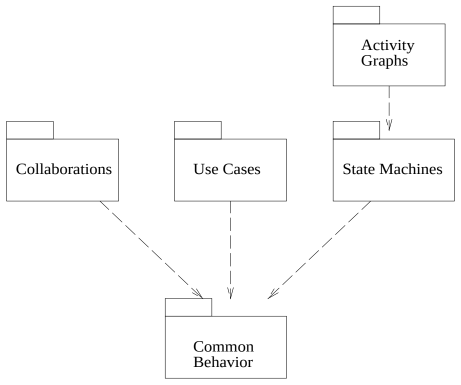


Figure 5: **Behavioral Elements Packages**

- use case diagram
- class diagram
- behavior diagram
  - statechart diagram
  - activity diagram
  - interaction diagrams:
    - sequence diagram
    - collaboration diagram
- implementation diagrams:
  - component diagram
  - deployment diagram

These diagrams provide multiple perspectives of the system under analysis or development. The underlying model integrates these perspectives so that a self-consistent system can be analyzed and built. These diagrams, along with supporting documentation, are the primary artifacts that a modeler sees [Obj99].

The following constructs of UML have possibility to be used in architecture description:

- **Class:** A class is a description of a set of objects that share the same attributes, operations, methods, relationships, and semantics [Obj99]. A class may use a set of interfaces to specify collections of operations it provides to its environment.
- **Classifier:** A classifier is an element that describes behavioral and structural features; it comes in several specific forms, including class, data type, interface, component, and others that are defined in other metamodel packages.
- **Package:** Defines a collection of related model elements. All modeling work is done in a package. Within a package we can reference elements (interfaces, specifications, architectures,

etc.) from another package if we *import* that package. In advanced usage, different aspects of a model element may be defined in different packages. Packages are used to separate (a) interfaces from implementations (b) different views of a subject, and (c) business process from specifications of designs.

- **Interface:** UML interface is a collection of operations that are used to specify services provided by a class or a component. All the most important component-based operating system facilities (such as COM+, CORBA, and Enterprise Java Beans) use interfaces as the glue that binds components together. Interfaces span logical and physical boundaries. The same interface you find used or realized by a component will be found used or realized by the classes that the component implements.

- **Component:** A *component* is a physical and replaceable part of a system that conforms to and provides the realization of a set of interfaces. Components are used to model the physical things that may reside on a node, such as executables, libraries, tables, files, and documents. A component typically represents the physical packaging of otherwise logical elements, such as classes, interfaces, and collaborations.

- **Subsystem:** A *subsystem* represents a behavioral unit in the physical system, and hence in the model. It is defined to be both a classifier and a package. A subsystem offers interfaces and has operations, and its contents may be partitioned into specification and realization elements. The specification of the subsystem consists of operations on the subsystem, together with specification elements such as use cases, state machines, etc. subsystem may or may not be instantiable.

- **Model:** A model is an abstraction of a physical system, with a certain purpose. It describes the physical system from a specific viewpoint and at a certain level of abstraction. A model contains all the model elements needed to represent a physical system completely by the criteria of this particular model. The model elements on a model are organized into a package/subsystem hierarchy, where the top-most package/subsystem represents the boundary of the physical system.

  Different models of the same physical system show different aspects of the system, from different viewpoints and/or levels of abstraction. The predefined stereotype <<systemModel>> can be applied to a model containing the entire set of models for the complete physical system.

  Relationships among elements in different models have no semantic impact on the contents of the models because of the self-containment of models. However, they are useful for tracing refinements and for keeping track of requirements between models.

UML can specify heterogeneous architecture. Subsystems can be used as components. Subsystems reflect different abstraction views. From the outside, a subsystem appears as a whole, collaborating with other parts of the system to fulfill its responsibilities. Its collaborators treat the subsystem as a black box. Subsystems are another encapsulation mechanism. The services provided by a subsystem are represented by interfaces and the corresponding operations. Other model elements (i.e., a state machine with action specification) further specify the behavior. These other model elements are called specification elements in the UML. The operations, interfaces, and the specification elements of the subsystem specify the system without reference to its parts [MWB99].

From the inside, a subsystem reveals itself to have a complex structure. It is a system of objects collaborating with each other to fulfill distinct responsibilities that contribute to the purpose of the

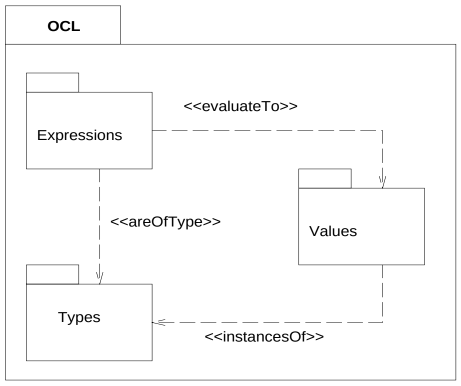


Figure 6: **Package Structure of the OCL Metamodel**

subsystem: the fulfillment of its responsibilities. These model elements specify the subsystem in terms of its parts; they are what UML calls the realization elements of the subsystem.

Complexity will vary by application domain and process phase. One of the key motivation in the minds of the UML developers was to create a set of semantics and notation that adequately addresses all scales of architectural complexity, across all domain [Obj99]. UML supports higher level development concepts such as components, collaborations, frameworks, and patterns. In object modeling, a component of a system is a subsystem or an object. A connector that consists of a hierarchy of parts is also a subsystem [MWB99].

1.4 Object Constraint Language

The UML uses mostly graphical notation to describe structural, dynamic, and functional aspects of a system. However, the graphical notation is inherently limited when specifying complex constraints. OCL is the standard for specifying invariants, preconditions, postconditions, and other kinds of constraints within UML [WK99]. The application of OCL is not limited to user models, it is also used by the UML standard itself for specifying well-formedness rules in context of UML semantics definition [Obj99]. OCL is a formal language but does not require strong mathematical background. Figure 6 shows the package structure of OCL metamodel. For details of OCL and further decomposition of packages see the papers by Warmer and Kleppe [WK99] and by Richters and Gogolla [RG99].

2 Related Work

This section gives a summary of work that related to the issues of using UML in architecture description.

2.1 Representing Architectures in the UML

Describing software architectures with UML works well for communicating the static structure of the architecture: the elements of the architecture, their relations, and the variability of a structure [HNS99]. These static properties are much more readily described with it than the dynamic properties. A particular sequence of activities can be easily described, but not general sequences. In addition, the ability to show peer-to-peer communication is missing from UML.

They separated the software architecture into four views: *conceptual, module, execution, and code*. The different views address different engineering concerns, and separation of such concerns helps the architect make sound decisions about design trade-offs. Each of the four views has particular elements that need to be described.

The conceptual view describes the architecture in terms of domain elements. The main concern of conceptual architecture is the functional features of the system. In conceptual view, UML Class Diagrams are used to show the static configuration, Real-Time Object-Oriented Modeling (ROOM) [SGW94] protocol declarations and UML Sequence Diagrams or State Diagrams for showing the protocols that ports adhere to, and UML Sequence diagrams for showing a particular sequence of interactions among a group of components.

The module view describes the decomposition of the software and its organization into layers. An important consideration here is limiting the impact of a change in external software or hardware. In module view, tables are used for describing the mapping between the conceptual and module views, UML Package Diagrams for showing subsystem decomposition dependencies, UML Class Diagrams for showing use-dependencies between modules, and UML Package Diagrams for showing use-dependencies among layers and the assignment of modules to layers.

The execution view is the run time view of the system: it is the mapping of modules to run time images, defining the communication among them, and assigning them to physical resources. Resource usage and performance are the key concerns in the execution view. In the execution view, the UML Class Diagrams are used to show the static configuration, UML Sequence Diagrams for showing the dynamic behavior of a configuration, or the transition between configurations, and UML State Diagrams or Sequence Diagrams for showing the protocol of a communication path.

The code view captures how modules and interfaces in the module view are mapped to source files, and run-time images in the execution view are mapped to executable files. In code view, tables are used to describe the mapping between elements in the module and execution views and elements in the code view, and UML component diagrams for showing the dependencies among source, intermediate, and executable files.

Hofmeister et al. [HNS99] concludes that the UML is deficient in describing the following elements:

- correspondences: A graphical notation is too cumbersome for straightforward mappings such as the correspondence between elements in different views. This information is more efficiently described in a table.

- protocols: The ability to show peer-to-peer communication is missing from UML. The communication part was described by adopting ROOM [SGW94] notation.

- ports on components: used nesting to show the relationships between ports and components, but this is visually somewhat misleading. A notation similar to the "lollipop" notation (lines with circles on one end) for the interfaces of a module is preferable.

- dynamic aspects of the structure: UML class diagrams describe the static structure of a system. Although UML sequence diagrams and state charts describe the dynamic behavior, but they don't support the dynamic configuration of a system.

- a general sequence of activities: UML sequence diagrams describe specific sequences of activities. Systems generally have defined modes, and the configuration of some of the modes may change over the time. A general sequence of activities is not supported by UML diagrams.

On the contrary, UML worked well for describing:

- the static structure of the architecture
- variability
- a particular sequence of activities

2.2 Integrating an Architecture Description with UML

Software architecture descriptions are high-level models of software systems. Some researchers have proposed special-purpose architectural notations that have a great deal of expressive power but are not well integrated with common development methods. Others have used mainstream development methods that are accessible to developers, but lack semantics needed for extensive analysis. Medvidovic et al. [RMRR97, MR99] described an approach to combining the advantages of these two ways of modeling architectures. They presented two examples of extending UML, an emerging standard design notation, for use with two architecture description languages, C2 and Wright. Their approach suggests a practical strategy for bringing architectural modeling into wider use, namely by incorporating substantial elements of architectural models into a standard design method.

2.3 UML and ROOM as an ADL

Rumpe et al. [RSRS99] explained some deficiencies of UML as an ADL, and made a proposal to eliminate these deficiencies by integrating the component-based OO modeling language ROOM with UML. UML combines a number of visual modeling sublanguages:

- Class diagrams and package diagrams offer all concepts of MILs (information hiding, import relationships, inheritance, genericity, etc.).
- The object constraint language OCL [WK99] allows for the definition of invariants as well as for pre- and postconditions and offers thereby the necessary means for "designing by contract".
- Various types of diagrams (state transition diagrams, collaboration diagrams etc.) may be used to model the dynamic behavior of networks of related objects.
- Finally, component and deployment diagrams may be used to define a mapping of logical software objects onto available hardware components.

One of UML's drawbacks is the UML component diagrams are not for representing *logical* decomposition of a software system into reusable and combinable subsystems. Also, UML does not provide the concept of connectors as first-class objects, which would be a hybrid of an association (association class) and a dependency between a class and an interface of another class. The authors suggested an extension of UML class diagrams with ROOM actor diagrams to allow the component-based description of software architectures.

2.4 Use recognized ADLs first, then map the architecture into UML design

There are three possible strategies in using UML in architecture design [AAM99]:

1. Use standard UML constructs to simulate modeling architectural concerns as would be done in an ADL [MR99]. However, in this approach, UML modeling capabilities do not fully satisfy architectural description requirements, such as explicit connector abstractions, compositional style etc.

2. Use UML's built in extension mechanisms (constraints, stereotypes and tagged values) [HNS99]. This approach was succesful in describing the elements of an architecture, their relations, and the variability of a structure, but failed in describing a general sequence of activities.

3. Augment the UML meta-model to directly support architectural concerns. Although this is a potentially effective approach, it would result in a notation that is incompatible with standard UML.

Egyed [EM99] developed a view integration framework in integrating C2 into UML. Abi-Antoun [AAM99] developed a semi-automated approach developed to assist in refining a high-level architecture specified in C2 into a design described with UML.

3 Architecture-based Testing

Testing is related to software quality. We generate test cases from programs and specifications. Specification-based testing has advantages over program-based testing in terms of providing test data early in development cycle. Formal requirement specifications have been used to generate test cases. The existing specification-based testing methods have limitations. For example, Offutt's method [OXL99, OL99] generates tests only from state-based specifications. Ammann and Black's specification-based mutation testing technique [AB99a, AB99b, ABM98] also relies on the type of specification – the specification has to be translated into a language that a model checker can recognize. However, in practice not many specifications are written using languages that can be model checked or languages that are state-based.

Software architecture descriptions are becoming an essential part of software development [BCK98, SG96]. Many important decisions about a software system are made at the architecture level, e.g., organization of a system as a composition of components; global control structures; the protocols for communication, synchronization, and data access; the assignment of functionality to design elements; the composition of design elements; physical distribution; scaling and performance; dimensions of evolution; and selection among design alternatives. Verification and validation of these decisions are essential in developing a software that is cost-effective, reliable, and maintainable.

Formalization of architecture description is a major concern among researchers and practitioners [BRJ98, BCK98, SG96]. Architecture descriptions that are written with architecture description languages allow us to take advantage of its formalism in generating test cases.

Jin and Offutt [Jin99] have defined six general properties to be tested at software architecture level. In the list of properties, a *conflict* occurs when rules, constraints or semantics cannot both be satisfied at the same time. In general, *deadlock* implies that a process does not participate in any events, but has not yet terminated successfully. A process is *deadlock free* if it can never go into a deadlock state.

1. **Component Consistency Requirements**
   Semantics, constraints and interfaces can be associated with components. They should be consistent with respect to each other and this consistency needs to be considered at the architecture level. Interfaces have types as well as data and control constraints.

- Component constraints and semantics should have no conflicts.
- Component constraints and semantics should be deadlock free.
- Component constraints and semantics should have no conflicts with the component interfaces constraints.

2. **Connector Consistency Requirements**
   A connector also contains interfaces, semantics, and constraints that need to be consistent. Interfaces have types as well as data and control constraints.

   - Connector constraints and semantics should have no conflicts.
   - Connector constraints and semantics should be deadlock free.
   - Connector constraints and semantics should have no conflicts with the associated connector interfaces constraints.

3. **Component-Connector Compatibility Requirements**
   Component interfaces are associated with connector interfaces to enable interactions. Informally, *compatibility* means that a component interface behaves in a manner that is consistent with assumptions made by the connector.

   - Component interfaces should be compatible with the associated connector interfaces.
   - For some compatibility requirements, it must be determined whether the component/connector relationship is deadlock free.

4. **Configuration Requirements**
   The configuration of a software architecture should be tested against several test requirements. An *initiation state* is the "start state", the state that the system is initially in. There are explicit *data flows* through the architecture of the system; a data element is given a value (*defined*) in its *source component* and the value is used in a *target component*. There are also explicit *control flows*; each architecture element has one or more designated *next element*. This transfer of execution could be between states in a component, through connectors, or across components.

   - Initiation Event: There should be at least one event declared to initiate the overall configuration.
   - Data Flow Reachability: A data element should be able to reach its designated target component from its source component through the connectors. The data element should reach the target component without having its value modified.
   - Control Flow Reachability: Every architecture should be able to reach its designated next element.
   - Connectivity: A component or connector interfaces with no next element or previous element is said to be "dangling". Dangling components and connector interfaces could indicate potential problems.
   - Interactions that in isolation are deadlock free can interact in such a way as to cause a deadlock situation. It should be the case that the system is deadlock free.

5. **Style Restriction Requirement**

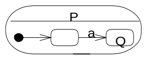
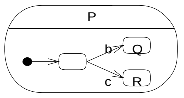
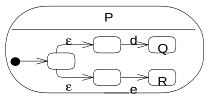
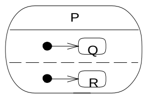
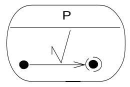
CSP Concept	CSP Notation	UML State Machine
Prefixing	$P = a \rightarrow Q$	
Alternative (deterministic choice)	$P = b \rightarrow Q \square c \rightarrow R$	
Desicion (non−deterministic choice)	$P = d \rightarrow Q \sqcap e \rightarrow R$	
Parellel Composition	$P = Q \parallel R$	
Success Event	$P = \surd$	

Figure 7: **UML State Machine templates for Wright's CSP constructs**

- The architecture style being used imposes some constraints on the software configuration. The system being used must satisfy those constraints.

In this section we attempt to suggest a way to generate test case from architecture descriptions. We generate test case from Wright specifications by translating them into UML statecharts, and then do a coverage analysis in terms of Jin's architecture level test requirements. Figure 7 shows the UML State Machine templates for Wright's CSP constructs [RMRR97].

We use the Wright specification example in Figure 8 from [AG94] to illustrate the translation process of Wright specification into UML statecharts. This example is for a system that transforms a stream of characters by capitalizing alternate characters while reducing the others to lowercase.

Wright's scoping of events is modeled in UML by prefixing every event's name with the name of the role to which the event belongs. The UML state machine of *Pipe* is shown in Figure 9.

Generating test cases from traditional flat state machines is straightforward and has well defined criteria and test data generation method [OA99]. In order to use the existing criteria and techniques, we may want to translate the hierarchical state chart into a flat state machine. However, this translation may cause a state explosion. Instead, we graphically distinguish the transitions only. A transition in a UML state machine consists of the following five elements: (1) *source* – the originating state vertex, (2) *target* – the target state vertex, (3) *trigger* – the event that triggers the transition, (4) *guard* – a boolean predicate that must be true for the transition to be fired, and (5) *effect* – an optional action to be performed when the transition fires. The UML standard allows trigger-less transitions (null state transitions) in the statechart. Such transitions are called *completion transitions*, and have an implicit trigger, the *completion event*, which is generated when all transition and entry actions and activities in the currently active state are completed. Completion events have priorities over all other events.

We use a set of transitions T_n to represent the statechart [LP99]. Figure 10 shows part of the transitions of statechart in Figure 9. Given this set of transitions, we can use Offutt's four level state-based test case generation criteria to do coverage analysis and generate actual test cases.

```
System Capitalize
    Component Split =
        port In = read?x → In □ read-eof → close → √
        port Left, Right = write!x → Out ⊓ close → √
        comp spec =
            let Close = In.close → Left.close → Right.close → √
            in Close □
                In.read?x → Left.write!x → ( Close □ In.read?x → Right.write!x → computation )
    Component Upper
        port In [input protocol]
        port Out [output protocol]
        comp spec [Upper specification]
    ...
    Connector Pipe =
        role Writer = write!x → Writer ⊓ close → √
        role Reader =
            let ExitOnly = close → √
            in let DoRead = ( read?x → Reader □ read-eof → ExitOnly )
            in DoRead √ ExitOnly
        glue = let ReadOnly = Reader.read!y → ReadOnly □ Reader.read-eof → Reader.close → √
                □ Reader.close → √
                in let WriteOnly = Writer.write?x → WriteOnly □ Writer.close → √
                in  Writer.write?x → glue
                    □ Reader.read!y → glue
                    □ Writer.close → ReadOnly
                    □ Reader.close → WriteOnly
        spec ∀ Reader.read_i!y • ∃ Writer.write_j?x • i = j ∧ x = y
                ∧ Reader.read-eof ⇒ ( Writer.close ∧ #Reader.read = #Writer.write )
Instances
    split:  Split; upper:  Upper; lower:  Lower; merge:  Merge;
    p1,p2,p3,p4:  Pipe
Attachments
    split.Left as p1.Writer;
    upper.In as p1.Reader;
    split.Right as p2.Writer;
    lower.In as p2.Reader;
    ...
end Capitalize.
```

Figure 8: A Wright Specification Example

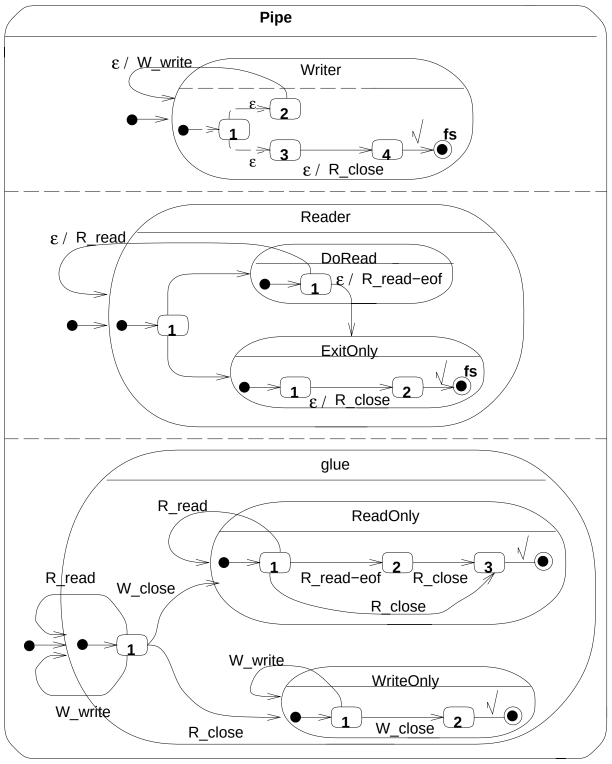


Figure 9: **UML State Machine Model of the *Pipe* Connector**

t1 :	Pipe (glue (1))	$\xrightarrow{\text{R\_read}}$	Pipe (glue)
t2 :	Pipe (glue (1))	$\xrightarrow{\text{W\_write}}$	Pipe (glue)
t3 :	Pipe (glue (1))	$\xrightarrow{\text{W\_close}}$	Pipe (glue (ReadOnly))
t4 :	Pipe (glue (1))	$\xrightarrow{\text{R\_close}}$	Pipe (glue (WriteOnly))
t5 :	Pipe (glue (ReadOnly(1)))	$\xrightarrow{\text{R\_read}}$	Pipe (glue (ReadOnly))
t6 :	Pipe (glue (ReadOnly(1)))	$\xrightarrow{\text{R\_read-eof}}$	Pipe (glue (ReadOnly(2)))
t7 :	Pipe (glue (ReadOnly(1)))	$\xrightarrow{\text{R\_close}}$	Pipe (glue (ReadOnly(3)))
t8 :	Pipe (glue (ReadOnly(2)))	$\xrightarrow{\text{R\_close}}$	Pipe (glue (ReadOnly(3)))
t9 :	Pipe (glue (ReadOnly(3)))	$\xrightarrow{\surd}$	Pipe (glue (ReadOnly (fs)))
t10 :	Pipe (glue (WriteOnly(1)))	$\xrightarrow{\text{W\_write}}$	Pipe (glue (WriteOnly))
t11 :	Pipe (glue (WriteOnly(1)))	$\xrightarrow{\text{W\_close}}$	Pipe (glue (WriteOnly(2)))
t12 :	Pipe (glue (WriteOnly(2)))	$\xrightarrow{\surd}$	Pipe (glue (WriteOnly (fs)))

Figure 10: **Transitions for the Pipe State Machine**

The criteria are *transition coverage, full predicate coverage, transition-pair coverage*, and *complete sequence coverage.* The details of the this method is given elsewhere [OXL99, OL99].

Figure 11 shows the test requirements at architecture level. Identifying which criterion satisfies which requirements needs more work. Remember that UML statecharts only describe the behavior of individual components or connectors. From statecharts we may be able to generate test cases to test component and connectors independently. However, other important issues of architecture level testing, e.g. the interaction or communication between components, compatibility of the data exchanged between components can not be tested. To test the conformance of components and connectors and the configuration of whole architecture, we need to specify an architecture configuration and component-connector connectivity explicitly and precisely in UML and extend the existing methods.

4 Future Work

The following issues have to be resolved in order to generate test cases from architecture based specifications in UML:

- the possible mapping between architecture level test requirements and state-based test generation criteria
- how to represent architecture configuration in UML
- how to deal with null transitions in UML state machines
- the possibility of dealing with interface compatibility and deadlock detection in UML

Test Requirements		Criterion
Component	Semantics and constraints have no conflict	
	Deadlock Free	
	Semantics/Constraints have no conflict with interface	
Connector	Semantics and constraints have no conflict	
	Deadlock Free	
	Semantics/Constraints have no conflict with interface	
Component − Connector	Compatibility	
	Connectivity	
Configuration	Type Usage	
	Data Flow	
	Control Flow	
	Style Rules	

Figure 11: **Test Requirements Coverage**

5 Conclusions

This paper gave an overview of existing ADLs, and surveyed the suitability of UML as an ADL with applications to testing. The extensibility of UML provides convenience in modeling to a degree that everything can be modeled in UML. However, tool support would be difficult.

We also made an attempt to generate test cases from Wright specification through existing testing criteria and techniques. The existing testing techniques may not satisfy the testing requirements at the architecture level.

6 Acknowledgements

This paper was originally written for a PhD seminar in Fall 1999, Software Architectures. Assistance from professors Liz White, Jeff Offutt, and Duminda Wijesekera is greatly appreciated.

References

[AAM99] Marwan Abi-Antoun and Nenad Medvidovic. Enabling the Refinement of a Software Architecture into a Design. In *Proceedings of the Second IEEE International Conference on the Unified Modeling Language (UML99)*, page 17, Fort Collins, CO, October 1999. IEEE Computer Society Press.

[AB99a] Paul Ammann and Paul E. Black. Abstracting formal specifications to generate software tests via model checking. In *Proceedings of the 18th Digital Avionics Systems Conference (DASC99)*, volume 2, pages 10.A.6.1–10, St. Louis, Missouri, October 1999. IEEE.

[AB99b] Paul E. Ammann and Paul E. Black. A specification-based coverage metric to evaluate test sets. In *Proceedings HASE99*, 1999.

[ABM98] Paul E. Ammann, Paul E. Black, and William Majurski. Using model checking to generate tests from specifications. In *Proceedings of the Second IEEE International Conference on Formal Engineering Methods (ICFEM'98)*, pages 46–54. IEEE Computer Society, December 1998.

[AG94] Robert Allen and David Garlan. Beyond Definition/Use: Architectural Interconnection. In *Proceedings of ACM Workshop on Interface Definition Languages*, pages 35–45, Portland, Oregon, January 1994.

[BCK98] Len Bass, Paul Clements, and Rick Kazman. *Software Architecture in Practice.* Addison Wesley, 1998.

[BEJV93] Pam Binns, Matt Englehart, Mike Jackson, and Steve Vestal. Domain-Specific Software Architectures for Guidance, Navigation, and Control. Technical report, Honeywell Technology Center, 1993. Available at http://www-ast.tds-gn.lmco.com/arch/arch-ref.html.

[BRJ98] Grady Booch, James Rumbaugh, and Ivar Jacobson. *The Unified Modeling Language User Guide.* Addison-Wesley, 1998. ISBN 0-201-57168-4.

[EJ93] Matt Englehart and Mike Jackson. ControlH: A Specification Language and Code Generator for Real-Time N&C Applications. Technical report, Honeywell Technology Center, 1993. Available at http://www-ast.tds-gn.lmco.com/arch/arch-ref.html.

[EM99] Alexander Egyed and Nenad Medvidovic. Extending Architectural Representation in UML with View Integration. In *Proceedings of the Second IEEE International Conference on the Unified Modeling Language (UML99)*, page 2, Fort Collins, CO, October 1999. IEEE Computer Society Press.

[Gar95] David Garlan. An Introduction to the Aesop System. Available at http://www.cs.cmu.edu/afs/cs/project/able/www/aesop/aesop_home.html, July 1995.

[GMW97] David Garlan, Robert T. Monroe, and David Wile. Acme: An Architecture Description Interchange Language. In *Proceedings of CASCON'97*, pages 169–183, Toronto, Ontario, November 1997.

[Har87] David Harel. Statecharts: A Visual Formalism for Complex Systems. *Science of Computer Programming*, (8):231–274, 1987.

[HNS99] C. Hofmeister, R. L. Nord, and D. Soni. Describing Software Architecture with UML. In *Proceedings of the First Working IFIP Conference on Software Architecture*, pages 145–160, San Antonio, TX, February 1999. IEEE Computer Society Press.

[Hoa95] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice Hall, 1995.

[Jin99] Zhenyi Jin. Software-Architecture Based Testing. PhD Dissertation Proposal, July 1999.

[Kru95] Philippe Kruchten. Architectural Blueprints - The "4 + 1" View Model of Software Architecture. *IEEE Software*, 12(6):42–50, November 1995.

[LKA$^+$95] David C. Luckham, John J. Kenney, Larry M. Augustin, James Vera, Doug Bryan, and Walter Mann. Specification and Analysis of System Architecture Using Rapide. *IEEE Transactions on Software Engineering*, 21(4):336–355, April 1995.

[LP99] Johan Lilius and Ivan Porres Paltor. Formalising UML State Machines for Model Checking. In *Proceedings of the Second IEEE International Conference on the Unified Modeling Language (UML99)*, pages 430–445, Fort Collins, CO, October 1999. IEEE Computer Society Press.

[MR99] Nenad Medvidovic and David S. Rosenblum. Assessing the Suitability of a Standard Design Method for Modeling Software Architectures. In *Proceedings of the First IFIP Working Conference on Software Architecture*, San Antonio, TX, February 1999. IEEE Computer Society Press.

[MT97] Nenad Medvidovic and Richard N. Taylor. A Framework for Classifying and Comparing Architecture Description Languages. Technical Report ICS-TR-97-02, University of California, Irvine, Department of Information and Computer Science, Irvine, CA, February 1997.

[MWB99] Joaquin Miller and Rebecca Wirfs-Brock. How Can a Subsystem Be Both a Package and a Classifier? In *Proceedings of the Second IEEE International Conference on the Unified Modeling Language (UML99)*, pages 584–597, Fort Collins, CO, October 1999. IEEE Computer Society Press.

[OA99] Jeff Offutt and Aynur Abdurazik. Generating tests from UML specifications. In *Proceedings of the Second IEEE International Conference on the Unified Modeling Language (UML99)*, pages 416–429, Fort Collins, CO, October 1999. IEEE Computer Society Press.

[Obj99] Object Management Group. *OMG UML Specification Version 1.3*, June 1999. Available at http://www.omg.org/uml/.

[OL99] Jeff Offutt and Shaoying Liu. Generating test data from SOFL specifications. *The Journal of Systems and Software*, 49(1):49–62, December 1999.

[OXL99] Jeff Offutt, Yiwei Xiong, and Shaoying Liu. Criteria for generating specification-based tests. In *Proceedings of the Fifth IEEE International Conference on Engineering of Complex Computer Systems (ICECCS '99)*, pages 119–131, Las Vegas, NV, October 1999. IEEE Computer Society Press.

[PW92] Dewayne E. Perry and Alexander L. Wolf. Foundations for the Study of Software Architecture. In *ACM SIGSOFT Software Engineering Notes*, volume 17, pages 40–52, October 1992.

[RG99] Mark Richters and Martin Gogolla. A Metamodel for OCL. In *Proceedings of the Second IEEE International Conference on the Unified Modeling Language (UML99)*, pages 156–171, Fort Collins, CO, October 1999. IEEE Computer Society Press.

[RMRR97] Jason E. Robbins, Nenad Medvidovic, David F. Redmiles, and David S. Rosenblum. Integrating Architecture Description Languages with a Standard Design Method. Technical Report ICS-TR-97-35, University of California, Irvine, Department of Information and Computer Science, August 1997.

[RSRS99] B. Rumpe, M. Schoenmakers, A. Radermacher, and A. Schurr. UML + ROOM as a Standard ADL? In *Proceedings of the Fifth IEEE International Conference on Engineering of Complex Computer Systems*, page 43, Las Vegas, Nevada, October 1999. IEEE Computer Society Press.

[SG96] Mary Shaw and David Garlan. *Software Architecture - Perspectives on an Emerging Discipline*. Prentice Hall, 1996.

[SGW94] B. Selic, G. Gullekson, and P.T. Ward. *Real-Time Object-Oriented Modeling*. John Wiley and Sons, New York, 1994.

[WK99] Jos Warmer and Anneke Kleppe. *The Object Constraint Language*. Addison-Wesley, 1999. ISBN 0-201-37940-6.

[Zel96] Gregory Zelesnik. *The UniCon Language Reference Manual*, May 1996. Available at http://www.cs.cmu.edu/ UniCon/reference-manual/Reference_Manual_1.html.