# Integrating Architecture Description Languages with a Standard Design Method

**Jason E. Robbins** **Nenad Medvidovic** **David F. Redmiles** **David S. Rosenblum**

Dept. of Information and Computer Science
University of California, Irvine
Irvine, CA 92697
{jrobbins,neno,redmiles,dsr}@ics.uci.edu

**ABSTRACT**

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. We describe an approach to combining the advantages of these two ways of modeling architectures. We present two examples of extending UML, an emerging standard design notation, for use with two architecture description languages, C2 and Wright. Our 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.

**Keywords**

Software architecture, object-oriented design, architecture description languages, constraint languages, incremental development

## 1 INTRODUCTION

Architecture-based software development is an approach to designing software in which developers focus on one or more high-level models of the software system rather than program source code. Architectural models include elements such as software components, communication mechanisms, states, processes, threads, hosts, events, external systems, and source code modules [6, 9, 10, 17, 23, 24]. Relationships between these elements address such issues as message passing, data flow, resource usage, dependencies, state transitions, causality, and temporal orderings. The basic promise of software architecture research is that better software systems can be achieved by modeling their important aspects during development. Choosing which aspects to model and how to evaluate them are two decisions that frame software architecture research [13].

Part of the software architecture research community, primarily academics, has focused on analytic evaluation of architecture descriptions. Answering difficult evaluation questions demands powerful modeling and analysis techniques that address specific aspects in depth. By paying the cost of making a detailed model, developers gain the benefit of knowing the answers to these questions. In this sense, software architecture descriptions serve primarily as input to analysis tools. For example, determining the possibility of deadlock requires specialized, formal models of the possible behavior and communication of each thread of control [3]. However, the emphasis on depth over breadth of the model can make it difficult to integrate these models with other development artifacts, because the rigor of formal methods draws the modeler's attention away from day-to-day development concerns. The use of special-purpose modeling languages has made this part of the architecture community fairly fragmented, as revealed by a recent survey of architecture description languages [14].

Another part of the community, primarily from industry, has focused on choosing which aspects to model. Modeling the wide range of issues that arise in software development demands a family of models that span and relate the issues of concern. By paying the cost of making such models, developers gain the benefit of clarifying and communicating their understanding of the system. In this sense software architectures serve primarily as the "big picture" of the system under development. For example, upgrading a database application requires an understanding of the various kinds of users and their respective tasks, the data schema, and the application's software components and their interfaces. However, emphasizing breadth over depth potentially allows many problems and errors to go undetected, because lack of rigor allows developers to ignore certain details. Several competing notations have been used in this part of the community, but they share central concepts, have been tempered by mainstream use, and have been formalized to some extent [4, 25]. There now exists a concerted effort to standardize methods for object-oriented analysis and design [16].

Standardization provides an economy of scale that results in more and better tools, better interoperability between tools, more available developers who are skilled in using that notation, and lower overall training costs. When special-purpose notations are needed, they can often be based on, or related to, standard notations. Doing so provides them with some of the benefits of the standard, and allows for more direct comparison and evaluation in terms of the value added by the special-purpose notation.

We use the Unified Modeling Language (UML) [18] as a starting point for bringing architectural modeling into wider use. UML is well suited for this because it provides a useful and extensible set of predefined constructs, it is semi-formally defined, it has substantial tool support, and it is based on experience with mainstream development methods. The next section describes UML and our strategy for adapting it to our needs. Sections 3 and 4 provide examples of adapting UML with semantics specific to two ADLs, C2 and Wright. Section 5 expands on our approach and contrasts it to related work. Section 6 discusses the contributions of our approach: specifically, it is a way to integrate the power of ADLs with the day-to-day usefulness of UML; and more generally, it suggests a practical strategy for achieving partial integration of architectural models as needed for specific development tasks.

## 2 UML AND ITS EXTENSION MECHANISMS

### 2.1 UML Background

A UML model of a software system consists of several partial models, each of which addresses a certain set of issues at a certain level of fidelity. There are eight issues addressed by UML models: (1) classes and their declared attributes, operations, and relationships; (2) the possible states and behavior of individual classes; (3) packages of classes and their dependencies; (4) example scenarios of system usage including kinds of users and relationships between user tasks; (5) the behavior of the overall system in the context of a usage scenario; (6) examples of object instances with actual attributes and relationships in the context of a scenario; (7) examples of the actual behavior of interacting instances in the context of a scenario; and (8) the deployment and communication of software components on distributed hosts. Fidelity refers to how close the model will be to the eventual implementation of the system: low-fidelity models tend to be used early in the life-cycle and be more problem-oriented and generic, whereas high-fidelity models tend to be used later and be more solution-oriented and specific. Increasing fidelity demands effort and knowledge to build more detailed models, but results in more properties of the model holding true in the system.

The UML is a graphical language with well-defined syntax and semantics. The syntax of the graphical presentation is specified by examples and a mapping from graphical elements to elements of the underlying semantic model [20]. The syntax and semantics of the underlying model are specified semi-formally via a meta-model, descriptive text, and constraints [19]. The meta-model is itself a UML model that specifies the abstract syntax of UML models. This is much like using a BNF grammar to specify the syntax of a programming language. For example, the UML meta-model states that a Class is one kind of model element with certain attributes, and that a Feature is another kind of model element with its own attributes, and that there is a one-to-many composition relationship between them. Semantic constraints are expressed in the Object Constraint Language (OCL) which is based on first-order predicate logic [21]. Each OCL expression is evaluated in the context of some model element (referred to as "self") and may use attributes and relationships of that element as terms. OCL also defines common operations on sets and bags, and constructs for traversing relationships so that attributes of other model elements may also be used as terms. Traversing a one-to-many or many-to-many relationship results in a set of instances. Several examples of OCL constraints are given below.

### 2.2 UML Extension Mechanisms

UML is an extensible language in that new constructs may be added to address new issues in software development. Three mechanisms are provided to allow limited extension to new issues without changing the existing syntax or semantics of the language. (1) *Constraints* place semantic restrictions on particular design elements. (2) *Tagged values* allow new attributes to be added to particular elements of the model. (3) *Stereotypes* allow groups of constraints and tagged values to be given descriptive names and applied to other model elements; the semantic effect is as if the constraints and tagged values were applied directly to those elements.

Figure 1 presents an example of using UML to model part of a human resources system. A company employs many workers, offers many training courses, and owns many robots. Robots and employees are workers. Labor union contracts constrain companies such that robots may not make up more than 10% of the work force. A training course contains many trainees, and each trainee may take from 1 to 4 courses. In this example, Trainee is an interface (a set of operations) rather than a full class. An employee is capable of performing all the operations of Trainee.

Suppose we wish to impose the design constraint that "a person may not be a composite element of another class," in other words, "a person must be the whole in any whole-part relationships." This does not prevent a person from participating in containment relationships, only composite relationships. In UML, containment (white diamond) indicates that one object is temporarily subordinate to one or more others, whereas composition (black diamond) indicates that an object is subordinate to exactly one other object throughout its life-time. In this example, composition would mean that employees could not participate in any other aggregates and never work for another company. Constraints may be applied directly to a class or, as we have done here, constraints may be applied to a stereotype (e.g., Person) and the stereotype applied to a class (e.g., Employee). The constraint may be stated formally in OCL as:

**Stereotype** Person for instances of meta-class Class
[1] If a person is in any composite relationship, it must be the composite.

```
self.oclType.assocEnd.forAll(myEnd |
  myEnd.association.assocEnd->exists(anyEnd |
    anyEnd.aggregation = composite) implies
      myEnd.aggregation = composite)
```

Note: The above constraint is sufficient because the UML already constrains associations to have at most one composite end.

The labor union rule uses terms from the model to constrain the state of the system at run-time. In contrast, the Person stereotype uses terms from the UML meta-model to constrain the model of the system. Traversing the "oclType" association allows us to refer to the meta-model, rather than the design at hand. Figure 2 shows the parts of the UML meta-model used in this paper. We have simplified the meta-model for purposes of illustration, but all the constraints we define can be easily rewritten for use with the complete meta-model.

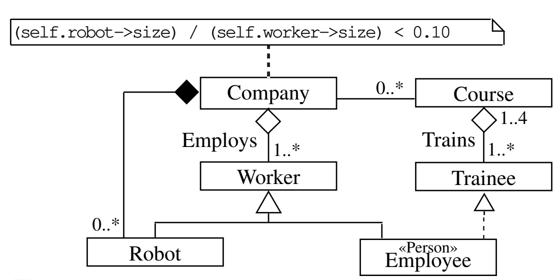
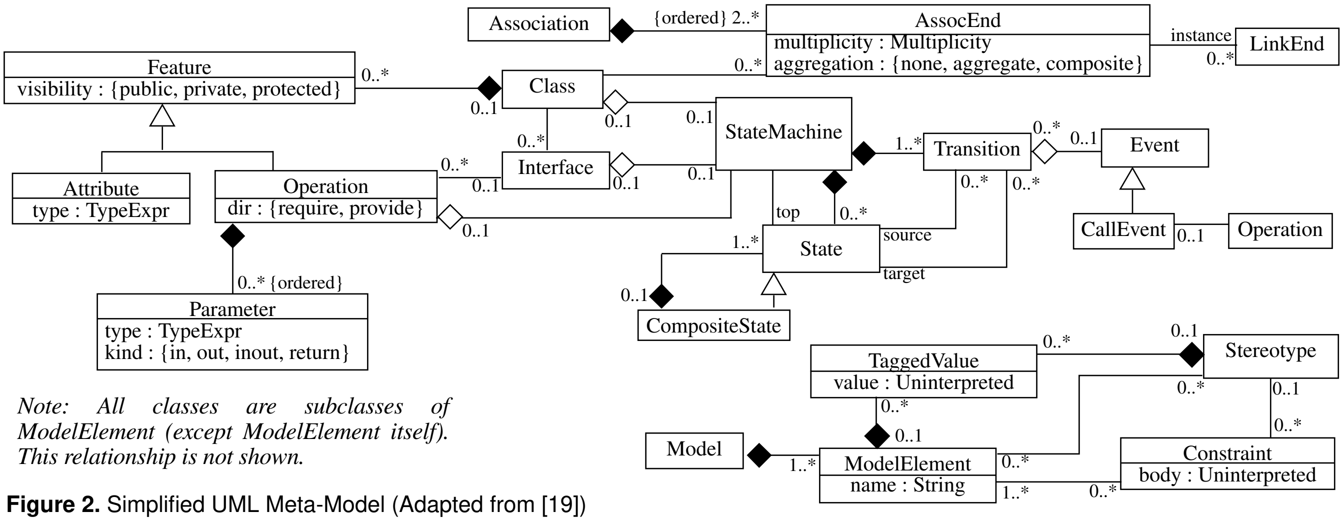



**Figure 1.** An example design expressed in UML




*Note: All classes are subclasses of ModelElement (except ModelElement itself). This relationship is not shown.*

**Figure 2.** Simplified UML Meta-Model (Adapted from [19])

### 2.3 Our Strategy for Adapting UML

One straightforward approach to using an ADL with UML is to define an ADL-specific meta-model. This approach has been used in more comprehensive formalization of architectural styles [1, 12]. Defining a new meta-model helps to formalize the ADL, but does not aid integration with standard design methods. By defining our new meta-classes as subclasses of existing meta-classes we would achieve some integration. For example, defining Component as a subclass of meta-class Class would give it the ability to participate in any relationship in which Class can participate. This is basically the integration that we desire. However, this integration approach requires *modifications* to the meta-model that would not *conform* to the UML standard, therefore we cannot expect UML-compliant tools to support it.

For the reason above, we restrict ourselves to using UML's built-in extension mechanisms on existing meta-classes. This allows the use of existing UML-compliant tools to represent the desired architectural models, and style conformance checking when OCL-compliant tools become available. Our basic strategy is to

- choose an existing meta-class from the UML meta-model that is semantically close to an ADL construct, then
- define a stereotype that can be applied to instances of that meta-class to constrain its semantics to that of the ADL.

In the next two sections, we demonstrate this strategy and illustrate the results with example specifications.

## 3 INTEGRATING UML AND C2

### 3.1 Overview of C2

C2 is a software architecture style for user interface intensive systems [24]. C2 SADL is an ADL for describing C2-style architectures [12, 14]; henceforth we use "C2" to refer to the combination C2 and C2 SADL. In a C2-style architecture, *connectors* transmit messages between components, while *components* maintain state, perform operations, and exchange messages with other components via two interfaces (named "top" and "bottom"). Each interface consists of a set of messages that may be sent and a set of messages that may be received. 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 four internal parts. An *internal object* stores state and implements the operations that the component provides. A wrapper on the *internal object* monitors all requested operations and sends notifications through the bottom interface. A *dialog specification* maps from messages received to operations on the internal object. Optionally, a *translator* may modify some messages so as to match those understood by other components, thus adapting a component to fit into a particular architecture.

In the C2 style, components may not directly exchange messages; they may only do so via connectors. Each component interface may be attached to at most one connector. 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."

The C2 style further demands that components communicate with each other only through message-passing, never through shared memory. Also, C2 requires that notifications sent from a component correspond to the operations of its internal object, rather than the needs of any components that receive those notifications. This constraint on notifications helps to ensure *substrate independence*, which is the ability to reuse a C2 component in architectures with differing substrate components (e.g., different window systems). The C2 style explicitly does not make any assumptions about the language in which the components or connectors are implemented, whether or not components have their own threads of control, the deployment of components to hosts, or the communication protocol used by connectors.

Figure 3 shows an example C2-style architecture. This system consists of four components and two connectors. One component is a database server, two are graphical user interfaces (GUI) to the database, and one is a window-system binding. One GUI is for posing queries, viewing result, and making updates. The other GUI is for configuring the database server. When either user interface is used to request a modification, a request message is sent upward to the connector, and then to the database. When the database

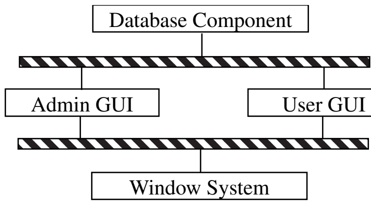



**Figure 3.** An example C2 architecture for a database application

performs an operation, a notification message is sent to the connector and is ultimately received by both GUI components. This style of component interaction is influenced by Model-View-Controller designs and supports multi-user systems and multi-view interfaces [8].

UML provides constructs for modeling software components, their interfaces, and their deployment on hosts. However, these built-in constructs are not suitable for describing C2-style software architectures because they assume both too much and too little. Components in UML are assumed to be concrete executable artifacts that take up machine resources such as memory. In contrast, C2 components are conceptual artifacts that decompose the system's state and behavior. C2 components may be implemented by concrete components, but they are not themselves concrete. Furthermore, components in UML may have any number of interfaces and any internal structure, whereas C2 components must follow the C2-style rules. Since "vanilla" UML does not fit our needs, we will adapt it to express several aspects of the C2 style.

### 3.2 C2 Operations in UML

The UML meta-class Operation matches the C2 concept of a message specification. UML Operations consist of a name and a parameter list (which may contain returned values). Operations indicate whether they will be provided or required (i.e., they may be received or sent). Operations may be public, private, or protected. To model C2 message specifications we add a tag to differentiate notifications from requests and constrain Operation to have no return values. C2 messages are all public, but that constraint is built into the UML meta-class Interface used below.

**Stereotype** C2Operation for instances of meta-class Operation
[1] C2Operations are tagged as either notifications or requests.

```
c2MsgType : enum { notification, request }
```

[2] C2 messages do not have return values.

```
self.parameter->forAll(p | p.kind <> return)
```

### 3.3 C2 Components in UML

The UML meta-class Class is closest to C2's notion of component. Classes may provide multiple interfaces with operations, may own internal parts, and may participate in associations with other classes. However, there are aspects of Class that are not appropriate, namely, they may have methods and attributes. In UML, an operation is a specification of a procedural abstraction (i.e., a procedure signature with optional pre- and post-conditions), while a method is a procedure body. Components in C2 provide only operations, not methods, and those operations must be part of interfaces provided by the component, not directly part of the component. Furthermore, a C2 conceptual component is assumed to have no state other than the state of its internal parts, and thus may have no direct attributes.

**Stereotype** C2Interface for instances of meta-class Interface
[1] A C2 interface has a tagged value identifying its position.

```
c2pos : enum { top, bottom }
```

[2] All C2Interface operations must have stereotype C2Operation.

```
self.oclType.operation->forAll(o | o.stereotype = C2Operation)
```

**Stereotype** C2Component for instances of meta-class Class
[1] C2Components may not directly contain features (i.e., methods, operations, or attributes).

```
self.oclType.feature->size = 0
```

[2] C2Components must implement exactly two interfaces, which must be C2Interfaces, one top, and the other bottom.

```
self.oclType.interface->size = 2 and
self.oclType.interface->forAll(i |
  i.stereotype = C2Interface) and
self.oclType.interface->exists(i | i.c2pos = top) and
self.oclType.interface->exists(i | i.c2pos = bottom)
```

[3] Requests travel "upward" only, i.e., they are sent through top interfaces and received through bottom interfaces.

```
Let topInt = self.oclType.interface->select(i |
    i.c2pos = top),
Let botInt = self.oclType.interface->select(i |
    i.c2pos = bottom),
topInt.operation->forAll(o |
  (o.c2MsgType = request) implies o.dir = require) and
botInt.operation->forAll(o |
  (o.c2MsgType = request) implies o.dir = provide)
```

[4] Notifications travel "downward" only. Similar to the constraint above.

[5] Each C2Component has one instance in the running system.

```
self.allInstances->size = 1
```

[6] C2Components participate in at most four whole-part relationships named internalObject, wrapper, dialog, and translator.

```
Let wholes = self.oclType.assocEnd->select(
    aggregation = composite),
(whole->size <= 4) and
((wholes.association.name->asSet) - Set {
    "internalObject", "wrapper", "dialog",
    "translator"})->size = 0
```

[7] Each operation on the internal object has a corresponding notification which is sent from the component's bottom interface.

```
Let ops = self.internalObject.feature->select(f |
    f->isKindOf(Operation)),
Let botInt = self.oclType.interface->select(i |
    i.c2pos = bottom),
ops->forAll(op |
  botInt->exists(note |
    (op.name = note.name and
     op.parameter = note.parameter) implies
      note.dir = required and note.c2MsgType = notification))
```

### 3.4 C2 Connectors in UML

C2 connectors share many of the constraints of C2 components. One difference is that they do not have any prescribed internal structure. Components and connectors are treated differently in the architecture composition rules discussed below. Another difference is that connectors may not define their own interfaces; instead their interfaces are determined by the components that they connect.

We can model C2 connectors using a stereotype C2Connector that is similar to C2Component. Below, we reuse some constraints and add two new ones. But first, we introduce three stereotypes for modeling the attachments of components to connectors. These attachments are needed to determine component interfaces.

**Stereotype** C2AttachOverComp for instances of meta-class Association
[1] C2 attachments are binary associations.

```
self.oclType.assocEnd->size = 2
```

[2] The first end of the association must be to a C2 component.

```
Let ends = self.oclType.assocEnd,
ends[1].multiplicity = "1..1" and
ends[1].class.stereotype = C2Component
```

[3] The second end of the association must be to a C2 connector.

```
Let ends = self.oclType.assocEnd,
ends[2].multiplicity = "1..1" and
ends[2].class.stereotype = C2Connector
```

**Stereotype** C2AttachUnderComp for instances of meta-class Association. Same as C2AttachOverComp, except that the first end must be to a connector, and the second end must be to a component.

**Stereotype** C2AttachConnConn for instances of meta-class Association
[1] C2 attachments are binary associations.

```
self.oclType.assocEnd->size = 2
```

[2] Each end of the association must be on a C2 connector.

```
self.oclType.assocEnd->forAll(ae |
  ae.multiplicity = "1..1" and
  ae.class.stereotype = C2Connector)
```

[3] The two ends are not both on the same C2 connector.

```
self.oclType.assocEnd[1].class <>
        self.oclType.assocEnd[2].class
```

**Stereotype** C2Connector for instances of meta-class Class
[1-5] Same as constraints 1-5 on C2Component.

[6] The top interface of a connector is determined by the components and connectors attached to its bottom.

```
Let topInt = self.oclType.interface->select(i | i.c2pos = top),
Let downAttach = self.oclType.assocEnd.associa-
    tion->select(a | a.assocEnd[2] = self.oclType),
Let topsIntsBelow = downAttach.assocEnd[1].inter-
    face->select(i | i.c2pos = top),
topsIntsBelow.operation->asSet = topInt.operation->asSet
```

[7] The bottom interface of a connector is determined by the components and connectors attached to its top. This is similar to the constraint above.

### 3.5 C2 Architectures in UML

Now we turn our attention to the overall composition of components and connectors in the architecture of a system. Recall that well-formed C2 architectures consist of components and connectors, components may be attached to one connector on the top and one on the bottom, and the top (bottom) of a connector may be attached to any number of other connectors' bottoms (tops). Below, we also add two new rules that guard against degenerate cases.

**Stereotype** C2Architecture for instances of meta-class Model
[1] A C2 architecture is made up of only C2 model elements.

```
self.oclType.modelElement->forAll(me |
  me.stereotype = C2Component or
  me.stereotype = C2Connector or
  me.stereotype = C2AttachOverComp or
  me.stereotype = C2AttachUnderComp or
  me.stereotype = C2AttachConnConn)
```

[2] Each C2Component has at most one C2AttachOverComp.

```
Let comps = self.oclType.modelElement->select(me |
    me.stereotype = C2Component),
comps->forAll(c |
    c.assocEnd.association->select(a |
        a.stereotype = C2AttachUnderComp)->size <= 1)
```

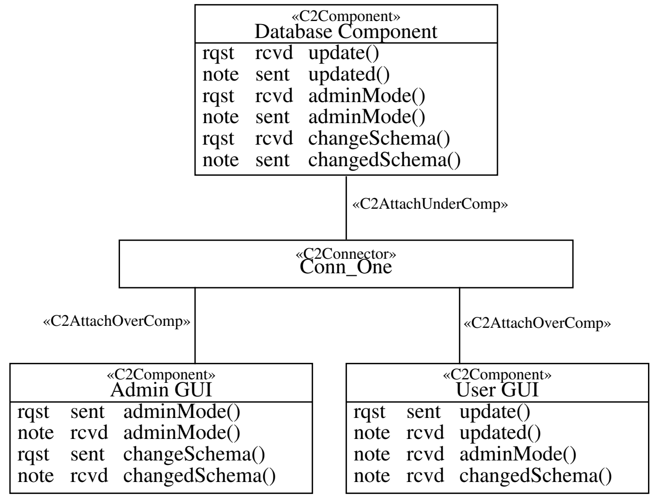



**Figure 4.** C2 architecture from Figure 3 expressed in UML

[3] Each C2Component has at most one C2AttachUnderComp. Similar to the constraint above.

[4] C2Components do not participate in any non-C2 associations.

```
Let comps = self.oclType.modelElement->select(me |
    me.stereotype = C2Component),
comps.assocEnd.association->forAll(a |
  a.stereotype = C2AttachOverComp or
  a.stereotype = C2AttachUnderComp)
```

[5] C2Connectors do not participate in any non-C2 associations.

```
Let conns = self.oclType.modelElement->select(me |
  me.stereotype = C2Connector),
conns.assocEnd.association->forAll(a |
  a.stereotype = C2AttachOverComp or
  a.stereotype = C2AttachUnderComp or
  a.stereotype = C2AttachConnConn)
```

[6] Each C2Component must be attached to some connector.

```
Let comps = self.oclType.modelElement->select(me |
    me.stereotype = C2Component),
comps->forAll(c |
    c.assocEnd.association->size > 0)
```

[7] Each C2Connector must be attached to some connector or component.

```
Let conns = self.oclType.elements->select(e |
    e.stereotype = C2Connector),
conns->forAll(c |
    c.assocEnd.association->size > 0)
```

### 3.6 Example C2 Architecture

Figure 4 shows the UML graphical notation for the same system shown in Figure 3 to illustrate the C2 style. We show some operations and omit others as needed to clarify the discussion below. Each element is marked with its stereotype in small double angle brackets. Alternatively, UML allows icons to be used to denote the stereotype.

Given a C2 architecture that is modeled in UML, it can be related to other standard UML model elements that are commonly used in software development. Figure 5 makes explicit our assumptions about the kinds of users who will use this system and their tasks. Figure 6 is a sequence diagram showing how the system behaves in the context of a particular use case. Explicitly modeling these aspects of the system enhances C2's support for component-based development of systems with complex user interfaces.

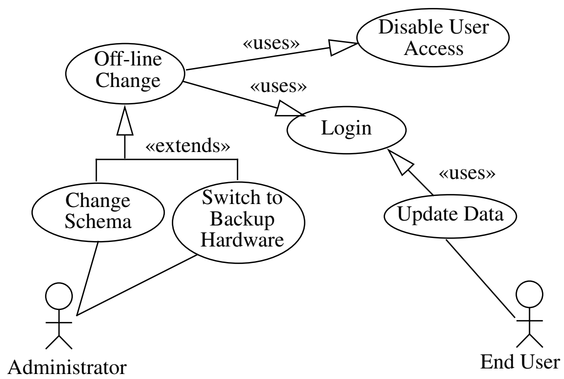



**Figure 5.** Some use cases for the example database system

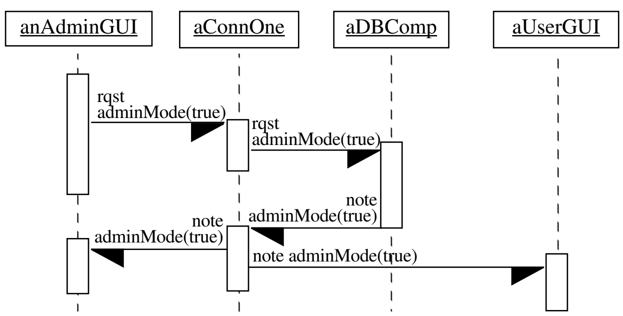



**Figure 6.** Sequence diagram for Disable User Access

### 3.7 Benefits of Integrating UML and C2

Adapting UML to enforce the C2-style rules has been fairly straightforward, because many C2 concepts are found in UML. Neither C2 nor UML constrain the choice of implementation language or require that any two components be implemented in the same language. Neither C2 nor UML constrain the choice of GUI toolkits or inter-process communication mechanisms. Neither C2 nor UML (as we have used it) assume that any two components run in the same thread of control or on the same host. Both C2 and UML limit communication to message passing and include specifications of messages that may be sent and received. Although we did not model details of the internal parts of a C2 component or the behavior of any C2 constructs, we feel those aspects of C2 could be modeled in UML. In fact, we provide an example of modeling behavior in the next section.

Some concepts of C2 are very different from those of UML and object-oriented design in general. For example, mainstream object-oriented design has a strict dichotomy between classes and instances. Since each class may have multiple instances, associations between classes may have multiplicity greater than one (e.g., there could be any number of employees in Figure 1). Furthermore, the features of an instance are declared in its class. In contrast, the interface of a C2 connector is determined by context rather than declared, and the addition of a new component instance at run-time is considered an architectural change. We addressed this difference by demanding that each C2 component and connector have exactly one instance. If a system uses two connectors, they must each have their own class in the design, although they may be implemented by the same concrete components. Another conceptual difference is that it is legal for C2 messages to be sent and not received by any component, whereas UML assumes that every message sent will be received. We have declined to address this last difference since it introduces more complexity than we feel it merits. As will be discussed more in Section 5, our approach allows aspects of an ADL to be expressed in UML or left to special purpose tools as desired.

## 4 INTEGRATING UML AND WRIGHT

The preceding section demonstrated that an ADL that supports a specific architectural style can be modeled in UML. This section shows the applicability of our approach to a general-purpose ADL, Wright [2, 3]. A more recent version of Wright also supports system families, architectural styles, and hierarchical composition. We do not address these newer features here, but believe that they could be incorporated into our model.

An architecture in Wright is described in three parts:
- component and connector types;
- component and connector instances; and
- configurations of component and connector instances.

Unlike C2, Wright does not enforce the rules of a particular style, but is applicable to multiple styles. However, it still places certain topological constraints on architectures. For example, as in C2, two components cannot be directly connected, but must communicate through a connector; on the other hand, unlike C2, Wright disallows two connectors from being directly attached to one another.

The remainder of the section describes an extension to UML for modeling Wright architectures. For brevity, stereotypes and constraints are elided whenever they are obvious from the discussion in this or the previous section.

### 4.1 Behavioral Specification in Wright

Wright uses a subset of CSP [7] to provide a formal basis for specifying the behavior of components and connectors, as well as the protocols supported by their interface elements. Given that this subset "defines processes that are essentially finite state" [2], it is possible to model Wright's behavioral specifications using UML's State Machines [20].

CSP processes are entities that engage in communication events. An event, e, can be primitive, or it can input or output a data item x (denoted in CSP with `e?x` or `e!x`, respectively). CSP events are modeled in State Machines as shown in Figure 7.

These two types of state transitions can be used in modeling more complex CSP expressions supported by Wright. Table 1 presents the mapping from CSP to State Machines using events with no actions (Figure 7a); the mapping for null events with actions (Figure 7b) is straightforward. It is possible for CSP events to have no associated data (see Figure 8 below). In such a case, the semantics of State Machines force us to make a choice as to which entities generate events and which observe them. We choose to model Wright ports and roles (described below) with event-generating actions, and computation and glue with transitions that observe those events.

The state machines in Table 1 can be used as templates from which equivalents of more complex CSP expressions can be


**Figure 7.** (a) A CSP event with input data, e?x, is modeled in UML State Machines as a state transition event with no action. (b) A CSP event, e, with output data, e!x, is modeled as a null state transition event that results in action e.

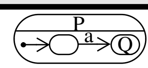
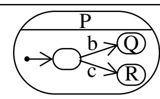
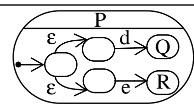
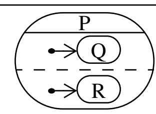
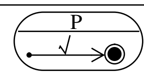
| CSP Concept | CSP Notation | UML State Machine |
|---|---|---|
| Prefixing | $P = a \rightarrow Q$ |  |
| Alternative (deterministic choice) | $P = b \rightarrow Q \,[]\, c \rightarrow R$ |  |
| Decision (non-deterministic choice) | $P = d \rightarrow Q \sqcap e \rightarrow R$ |  |
| Parallel Composition | $P = Q \parallel R$ |  |
| Success Event | $P = \surd$ |  |

**Table 1.** UML State Machine templates for Wright's CSP constructs

formed. Therefore, a "Wright" state machine is described by the following stereotypes.

**Stereotype** WSMTransition for instances of meta-class Transition
[1] A transition is tagged as one of the two cases shown in Figure 7.
```
WSMtransitionType : enum { event, action }
```
[2] An "event" transition consists of an event only (Figure 7a).
```
self.oclType.WSMtransitionType = event implies
  (self.oclType.event.oclIsTypeOf(CallEvent) and
  self.oclType.ActionSequence->size = 0)
```
[3] An "action" transition consists of a null event and an action (Figure 7b).
```
self.oclType.WSMtransitionType = action implies
  (self.oclType.event->size = 0 and
  self.oclType.ActionSequence.Action->size = 1)
```
**Stereotype** WrightStateMachine for instances of metaclass StateMachine
[1] A WrightStateMachine consists of one of the composite states discussed above, and partially depicted in Table 1. Each simple state may be refined as another WrightStateMachine. This constraint is elided in the interest of space.

[2] All WrightStateMachine transitions must be WSMTransitions.
```
self.oclType.transition->forAll(t | t = WSMTransition)
```

## 4.2 Wright Component and Connector Interfaces in UML

Each Wright interface (a *port* in a component or a *role* in a connector) has one or more operations. In Wright, these operations are modeled implicitly, as part of a port or role's CSP protocol. We choose to model the operations explicitly in UML. The CSP protocols associated with a port or role are modeled as WrightStateMachines.

**Stereotype** WrightOperation for instances of meta-class Operation
[1] WrightOperations do not have parameters; parameters are implicit in the CSP specification associated with each operation
```
self.parameter->size = 0
```

**Stereotype** WrightInterface for instances of meta-class Interface
[1] WrightInterfaces are tagged as either ports or roles.
```
WrightInterfaceType : enum { port, role }
```
[2] All operations in a WrightInterface are WrightOperations.
```
self.oclType.operation->forAll(o |
  o.stereotype = WrightOperation)
```
[3] Exactly one WrightStateMachine is associated with each WrightInterface.
```
self.oclType.stateMachine->size = 1 and
self.oclType.stateMachine->forAll(s |
  s.stereotype = WrightStateMachine)
```
[4] In a WrightInterface, a WrightStateMachine is expressed only in terms of that interface's operations; these are operations on the state machine's call events.
```
self.oclType.stateMachine.transition->forAll(t |
  (t.event.oclIsTypeOf(CallEvent)) implies
    self.oclType.operation->exists(o |
      o = t.event.operation))
```
A WrightInterface, as modeled above, specifies the alphabet of a port or role.

## 4.3 Wright Connectors in UML

A connector type in Wright is described as a set of *roles*, which describe the expected behavior of the interacting components, and a *glue*, which defines the connector's behavior, by specifying how its roles interact.

We will model Wright connectors with the UML meta-class Class. Wright connectors provide multiple interfaces (roles) and participate in associations with other classes (Wright components). Wright connector types are assumed to have no state other than the state of their internal parts, and thus may have no direct attributes.

**Stereotype** WrightGlue for instances of meta-class Operation
[1] WrightGlue is modeled as a WrightOperation.
```
self.oclType.operation->forAll(o |
  o.stereotype = WrightOperation)
```
[2] WrightGlue contains a single WrightStateMachine.
```
self.oclType.stateMachine->size = 1 and
self.oclType.stateMachine->forAll(s |
  s.stereotype = WrightStateMachine)
```
**Stereotype** WrightConnector for instances of meta-class Class
[1] WrightConnectors must implement at least one WrightInterfaceType, which must be a role.
```
self.oclType.interface->size >= 1 and
self.oclType.interface->forAll(i |
  i.stereotype = WrightInterface and
  i.WrightInterfaceType = role)
```
[2] A WrightConnector contains a single glue.
```
self.oclType.operation->size = 1 and
self.oclType.operation->forAll(o |
  o.stereotype = WrightGlue)
```
[3] Operations with no data and with input data that belong to the different interface elements of a connector are the trigger events in glue's state machine.
```
self.oclType.operation.stateMachine.transition->forAll(t |
  (t.event.oclIsTypeOf(CallEvent)) implies
    self.oclType.interface.operation->exists(o |
      o = t.event.operation))
```
[4] Operations with output data that belong to the different interface elements of a connector are the actions in glue's state machine. Similar to the above constraint.

[5] The semantics of a Wright connector can be described as the parallel interaction of its glue and roles [2].

```
self.oclType.stateMachine->size = 1 and
self.oclType.stateMachine->forAll(sm |
  sm.state->size = 1 and sm.state->forAll(s |
    s.oclType = CompositeState and s.isConcurrent = true and
    s.state->size = 1 + self.oclType.interface->size and
    s.state->exists (gs |
      gs = self.oclType.operation.stateMachine.top) and
    self.oclType.interface->forAll(i |
      s.state->exists (rs | rs = i.stateMachine.top))))
```

[6] A WrightConnector must have at least one instance in the running system.

```
self.allInstances->size >= 1
```

### 4.4 Wright Components in UML

A component type is modeled by a set of *ports*, which export the component's interface, and a *computation* specification, which defines the component's behavior. We model Wright components in UML with a stereotype WrightComponent. This stereotype has much in common with the WrightConnector stereotype, and is thus omitted.

### 4.5 Wright Architectures in UML

We introduce stereotypes for modeling the attachments of components to connectors and for Wright architectures. Unlike C2, which considers architectures to be networks of abstract placeholders, Wright architectures are composed of component and connector instances. One solution we considered was to define WrightConnectorInstance and WrightComponentInstance stereotypes and express architectural topology in terms of them. However, we believe that it is undesirable to introduce instances at this level, since we are still dealing with design issues. Additionally, we have found that most of the constraints on component and connector instances can be expressed in terms of their corresponding types. Therefore, we refer to component and connector types in the stereotypes below.[1]

**Stereotype** WrightAttachment for instances of meta-class Association

[1] Wright attachments are associations between two elements.

```
self.oclType.assocEnd->size = 2
```

[2] One end of the association must be to a Wright component.

```
Let ends = self.oclType.assocEnd,
ends[1].multiplicity = "1..1" and
ends[1].class.stereotype = WrightComponent
```

[3] The other end of the association must be to a Wright connector.

```
Let roles = self.oclType.assocEnd,
ends[2].multiplicity = "1..1" and
ends[2].class.stereotype = WrightConnector
```

**Stereotype** WrightArchitecture for instances of meta-class Model

[1] A WrightArchitecture is made up of only Wright model elements.

```
self.oclType.elements->forAll(e |
  e.stereotype = WrightComponent or
  e.stereotype = WrightConnector or
  e.stereotype = WrightAttachment)
```

[2] Each WrightComponent port participates in at most one WrightConnector role.

```
Let comps = self.oclType.elements->select(e |
  e.stereotype = WrightComponent),
comps.assocEnd->forAll(ae | ae.linkEnd->size = 1)
```

[3] Each WrightConnector role is fulfilled by at most one WrightComponent port. Similar to the constraint above.

[4] WrightComponents and WrightConnectors do not participate in any non-Wright associations. Similar to constraints [4-5] in Section 3.5.

1.The one exception is in constraints 2 and 3 of the WrightArchitecture stereotype: "linkEnd" refers to an instance of a class (type).

```
connector Pipe =
  role Writer = write → Writer ⊓ close → √
  role Reader =
    let ExitOnly = close → √
    in let DoRead = (read → Reader
                     [] read-eof → ExitOnly)
    in DoRead ⊓ ExitOnly
  glue = let ReadOnly = Reader.read → ReadOnly
                        [] Reader.read-eof
                           → Reader.close → √
                        [] Reader.close → √
      in let WriteOnly = Writer.write → WriteOnly
                         [] Writer.close → √
      in Writer.write → glue
       [] Reader.read → glue
       [] Writer.close → ReadOnly
       [] Reader.close → WriteOnly
```

**Figure 8.** A connector specified in Wright (adapted from [2])

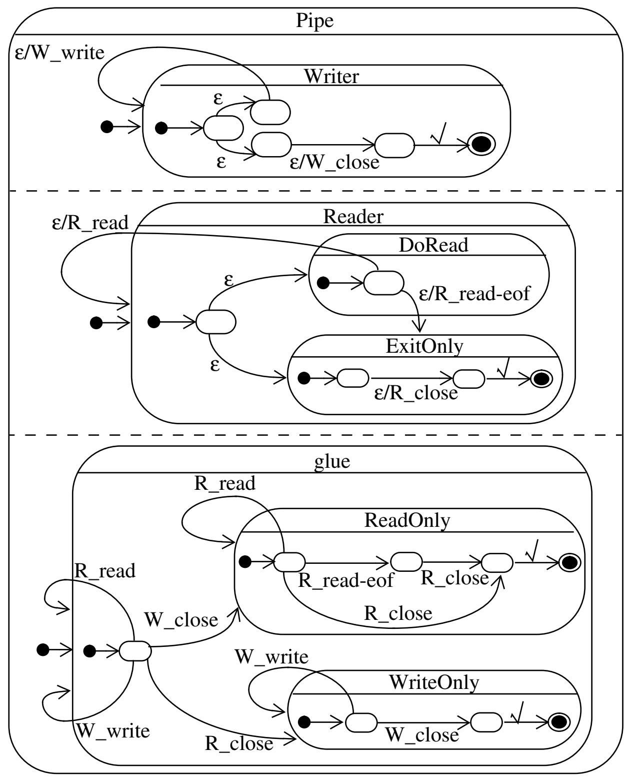



**Figure 9.** UML State Machine model of the *Pipe* connector

The semantics of port-role attachments in Wright are formally defined [3]. However, Wright places no language-level constraints on port-role pairs. Instead, establishing and enforcing these constraints is the task of external analysis tools. Hence, we provide no port-role compatibility constraints.

### 4.6 Example Partial Wright Architecture

Having provided an extension to UML for modeling Wright architectures, we now demonstrate how that extension is used to describe a Wright specification. Figure 8 shows the *Pipe* connector example from [2]. The UML State Machine model of the *Pipe* is shown in Figure 9. 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 class diagram for *Pipe* is analogous to the C2 diagram shown in Figure 4, and has been omitted for brevity

### 4.7 Benefits of Integrating UML and Wright

Modeling an ADL such as Wright in UML provides benefits both to practitioners who prefer Wright as a design notation and to those who are more familiar with UML. Mapping a Wright architecture to UML enables a Wright user to leverage a wide number of general-purpose UML tools (e.g., code generation, simulation, analysis, reverse engineering, and so forth). On the other hand, being able to map a UML design of a system to Wright (by adhering to the constraints specified in this section) would enable a UML designer to utilize Wright's powerful analysis capabilities, such as interface compatibility checking and deadlock detection.

## 5 CORE MODELS AND EXTENSIONS

Notational standardization has a wide range of benefits, as discussed in the introduction. The challenge of standardization is finding a language that is general enough to capture needed concepts without adding too much complexity. It is tempting to extend the UML meta-model to fully capture each feature of each ADL. However, such a notation would be overly complex and incompatible with standard UML tools. There has never been a single programming language that served the needs of all programmers, and there is no reason to expect a single ADL to meet the needs of all software architects. This has led the software architecture community to attempt interchange rather than standardization of ADLs.

ACME is an architecture interchange language that supports automatic transformation of a system modeled in one ADL to an equivalent model in another ADL [5]. This allows architects to model and analyze their system architecture in one ADL and then translate the model to another ADL for further analysis. Architects need not work directly with ACME; they may instead use the ADL and toolset that is most suited to the current issue of concern. ACME's approach is easier than providing direct mappings between pairs of ADLs because the ACME language serves as an intermediate step and provides additional tool support. ACME's *architectural ontology* plays a role analogous to UML's meta-model; however it is smaller and focuses on structural aspects of architectures.

Full realization of ACME's goals presents a number of challenges. Complete, automated translation among a set of ADLs requires a set of semantic mappings that involve every concept of every ADL in the set, which may not be possible given that different ADLs address different system aspects and have different semantics. The translation approach depends on exploiting constructs common to every ADL. At this point, the evident commonalities are syntactic rather than semantic [14]. For these reasons ACME emphasizes a partial and incremental approach.

ACME uses a seven element architectural ontology together with key-value pairs to represent arbitrary, uninterpreted architectural features and a template mechanism that leverages commonalities. Like ACME, our approach uses a fixed ontology (the UML meta-model), key-value pairs (tagged-values), and templates (stereotypes). However, UML provides much richer semantics due to its more comprehensive meta-model and its first-order predicate logic constraints.

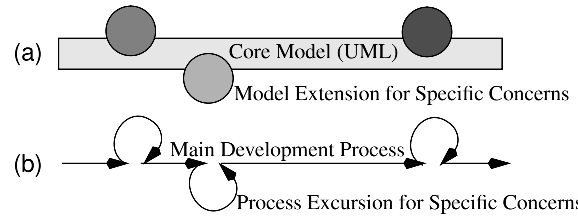



**Figure 10.** (a) A core model with extensions
(b) Sketch of an associated process

A fundamental difference is that our approach does not use translation between notations, but rather uses a core model with several independent extensions. We use UML as our core model and assume that developers are able to use UML constructs, such as classes and use cases, in day-to-day development activities. We extend this core model with specific attributes and constraints as needed for specific analyses. As new issues of concern arise in development, new attributes may be added to support analyses relevant to those concerns. The semantics of the core model are always enforced by UML-compliant tools. The semantics of each extension are enforced by the constraints of that extension and the constraints imposed by the desired analyses. Dependencies and conflicts may arise between the attributes in different extensions, and must be handled by developers just as they manage the other myriad dependencies and potential conflicts of software development. This situation is not ideal, but it is practical: it uses available methods and tools that are well integrated into day-to-day development, and it is incremental. We feel that these features are key to bringing the benefits of architectural modeling into mainstream use.

In using a core model and extensions, the question arises of what should be in the core and what should be left to extensions. Technical considerations play some role in this decision. For example, ACME's simple architectural ontology eases tool building, whereas UML's larger meta-model presents a higher barrier. Development processes also influence the core model. For example, object-oriented design and use cases are widely used by practitioners and directly relate to day-to-day development activities. We choose UML as our core model because it is grounded in mainstream development practices, already has substantial tool support, and provides explicit extension mechanisms.

Figure 10 sketches a process in which developers use the core model and some available extensions for day-to-day development concerns and take *process excursions* as needed to address specific architectural concerns identified during the main process. Information learned in excursions guides later decisions in the main process. Different concerns will arise as the main process progresses and model fidelity increases. For example, deadlock can only be addressed once system behavior is specified in detail. We envision developers using UML normally and ADL-specific tools as needed; an alternative process more suited to researchers might involve using an ADL normally and UML tools as needed (e.g., to generate code).

## 6 CONCLUSIONS

Further research into this approach will attempt to integrate UML with the semantics of other ADLs, apply object-oriented concepts such as polymorphism and inheritance to architectural elements [15], exploit more formal semantics [4, 25] and evaluate the effectiveness of the approach in practice. In addition to C2 and Wright, we have also investigated integrating UML with Darwin [11] and Rapide [10]. Each of these ADLs has certain aspects in common with UML, some of which can be expressed with UML's extension mechanisms, while others may be included in a UML specification but can only be interpreted by ADL-specific tools.

From our experience to date, adapting UML to address architectural concerns seems to require reasonable effort, be a useful complement to ADLs and their analysis tools, and be a practical step toward mainstream architectural modeling. Using UML has the benefits of leveraging mainstream tools, skills, and processes. It may also aid in comparison of ADLs because it forces some implicit assumptions to be explicitly stated in common terms.

Integrating architectural models into mainstream development methods is not simply a matter of convenience. Based on experience in complex system design, "mismatches between the systems models used by the R&D design team and those of the system engineer, manufacturer, and user have delayed delivery, raised costs, entailed product rework, and led to faulty failure diagnoses [22]." These problems arise when models become out of synch with the system and current design concerns, or when lessons learned in modeling are not communicated to developers. Integrating architectural models with standard design methods addresses both these issues.


## ACKNOWLEDGEMENTS

This material is based upon work supported by the National Science Foundation under Grant No. CCR-9624846 and Grant No. CCR-9701973. Effort also sponsored by the Defense Advanced Research Projects Agency, Rome Laboratory, Air Force Materiel Command, USAF under agreement numbers F30602-97-2-0021 and F30602-94-C-0218, and Air Force Office of Scientific Research under grant number F49620-98-1-0061. Additional support is provided by Rockwell International and Northrop Grumman Corp. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon. 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 Defense Advanced Research Projects Agency, Rome Laboratory or the U.S. Government.


## REFERENCES


1. Abowd, G., Allen, R., and Garlan, D. Formalizing style to understand descriptions of software architecture. Trans. Software Engineering and Methodology. October 1995. pp. 319-364.
2. Allen, R. and Garlan, D. Formalizing Architectural Connection. Proceedings of the 1994 International Conference on Software Engineering, Sorrento, Italy, May 1994. pp. 71-80.
3. Allen, R. and Garlan, D. A Formal Basis for Architectural Connection. ACM Transactions on Software Engineering and Methodology. vol. 6. no. 3. July 1997. pp. 213-249.
4. Bourdeau, R. and Cheng, B. A formal semantics for object model diagrams. IEEE Trans. on Software Engineering. vol. 21. no. 10. October 1995. pp. 799-821.
5. Garlan, D., Monroe, R. and Wile D. ACME: An architectural interconnection language. Proc. of CASCON'97. Toronto, Canada. November 1997.
6. Garlan, D. and Shaw M. An introduction to software architecture: Advances in software engineering and knowledge engineering, volume I. World Scientific Publishing, 1993.
7. Hoare, C. A. R. Communicating Sequential Processes. Prentice Hall, 1985.
8. Krasner, G. E. and Pope, S. T. A cookbook for using the model-view-controller user interface paradigm in smalltalk-80. J. Object-Oriented Programming. vol. 1. no. 3, Aug/Sept 1988. pp. 26-49.
9. Kruchten, P. B. The 4+1 view model of architecture. IEEE Software. Nov. 1995. pp. 42-50.
10. Luckham, D. C., and Vera, J. An event-based architecture definition language. IEEE Transactions on Software Engineering. vol. 21. no. 9. September 1995. pp. 717-734.
11. Magee, J. and Kramer, J. Dynamic structures in software architecture. Proc. of SIGSOFT'96. San Francisco, CA, October 1996.
12. Medvidovic N., Taylor, R. N., Whitehead, Jr. E. J. Formal modeling of software architectures at multiple levels of abstraction. Proc. California Software Symposium, Los Angeles, April 1996. pp. 29-40.
13. Medvidovic, N. and Rosenblum, D. S. Domains of concern in software architectures and architecture description languages. Proc. USENIX Conf. on Domain Specific Languages, Santa Barbara, CA, October. 1997. pp. 199-212.
14. Medvidovic, N. and Taylor, R. N. A framework for classifying and comparing architecture description languages. The Sixth European Software Engineering Conference together with SIGSOFT'97. Zurich, Switzerland, September 1997. pp. 60-76.
15. Medvidovic, N., Oreizy, P., Robbins, J. E., and Taylor, R. N. Using object-oriented typing to support architectural design in the C2 style. SIGSOFT'96. pp 24-32. San Francisco, CA, October 1996.
16. Object Management Group. Object analysis and design RFP-1. Object Management Group document ad/96-05-01. June 1996. Available from http://www.omg.org/docs/ad/96-05-01.pdf.
17. Perry, D. E. and Wolf, A. L. Foundations for the study of software architectures. Software Engineering Notes. October 1992.
18. Rational Partners (Rational, IBM, HP, Unisys, MCI, Microsoft, ObjecTime, Oracle, i-Logix, etc.). Proposal to the OMG in response to OA&D RFP-1. Object Management Group document ad/97-07-03. July 1997. Available from http://www.omg.org/docs/ad/.
19. Rational Partners. UML Semantics. Object Management Group document ad/97-08-04. Sept. 1997. Available from http://www.omg.org/docs/ad/97-08-04.pdf.
20. Rational Partners. UML Notation Guide. Object Management Group document ad/97-08-05. Sept. 1997. Available from http://www.omg.org/docs/ad/97-08-05.pdf.
21. Rational Software Corporation and IBM. Object constraint language specification. Object Management Group document ad/97-08-08. Sept. 1997. Available from http://www.omg.org/docs/ad/.
22. Rechtin, E. The synthesis of complex systems. IEEE Spectrum, July 1997. pp. 51-55.
23. Soni, D., Nord, R., and Hofmeister C. Software architecture in industrial applications. Proc. of the 17th International Conference on Software Engineering. Seattle, WA. 1995. pp. 196-207.
24. Taylor, R. N., Medvidovic, N., Anderson, K., Whitehead, Jr., E. J., Robbins, J. E., Nies, K. A., Oreizy, P., and Dubrow, D. L. A component and message-based architectural style for GUI software. IEEE Trans. Software Engineering, June 1996, vol.22, no.6, pp.390-406.
25. Wang, E., Richter, H., and Cheng, B. Formalizing and integrating the dynamic model within OMT. Proc. IEEE International Conference on Software Engineering (ICSE'97). Boston, MA. May 1997. pp. 45-55.