Integrating C2 with the Unified Modeling Language

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

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

ABSTRACT
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. Choosing which aspects to model and how to evaluate them are two decisions that frame software architecture research. Some software architecture researchers have proposed special-purpose notations that have a great deal of expressive power but are not well integrated with common development methods. Others have used general-purpose notations that are accessible to developers, but lack details needed for extensive analysis. In this paper we describe an approach to combining the advantages offered by these two different kinds of notation. In particular, we extend UML, an emerging standard notation for object-oriented design, with semantics specific to C2, an architectural style for user-interface intensive systems. Doing so suggests a practical strategy for bringing architectural modeling into the mainstream of software development and achieving partial integration of architectural models as needed.

Keywords
Software architecture, object-oriented design, formal methods, constraint languages, incremental development

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, libraries, classes, states, processes, threads, hosts, events, use cases, external systems, and source code modules [1, 6, 8, 9, 16, 22, 24]. Relationships between these elements address such issues as message passing, data flow, resource usage, dependencies, state transitions, causality, and temporal orderings. Some architectural models additionally support multiple levels of refinement with mappings between corresponding elements at successive levels [14, 9]. 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 [11].

Part of the software architecture research community, primarily academics, has focused on analytic evaluation. Answering difficult evaluation questions demands powerful modeling and analysis techniques that address specific aspects in depth. By paying the cost of developing a detailed model, software 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. 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 and to low-level modeling constructs (e.g., will this function result in a set of integers, or a set of sets of integers?). The use of special-purpose modeling languages has made this part of the architecture community fairly fragmented, as revealed by a recent survey describing nine different architecture description languages [12].

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 developing a model, software 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 application's user interfaces, the data schema, and the application's software components and their interfaces. 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 [e.g., 2, 3, 21], but they share central concepts and have been tempered by use. There now exists a concerted effort to standardize methods for object-oriented analysis and design [15].

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.

In the remainder of this paper we provide one example of combining these two emphases and outline an approach to software development that allows developers to gain some of the benefits of each. First, we describe an example of each kind of notation — the C2 architectural style which addresses a narrow set of issues in depth and UML which addresses a broad set of issues but leaves many details unspecified. Then, we describe how UML's extension mechanisms can be used to express several aspects of C2. The final section discusses the contributions of expressing C2 in terms of UML: specifically, it is a way to integrate the guidance provided by C2 with design notations more familiar to practitioners; and more generally, it suggests a practical strategy for achieving partial integration of architectural models as needed.

BACKGROUND

Overview of C2

C2 is a software architecture style for user interface intensive systems [24]. A C2-style architecture consists of software components and connectors. Connectors transmit messages between components. 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 those that may be received. Inter-component messages are classified into two types: requests for a component to perform an operation, and 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.

Components and connectors are composed according to several style rules. 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 1 shows an example C2-style architecture. This system consists of three components and one connector. One component is a database server, the other two are graphical user interfaces (GUI) to the database. 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 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 [7].

Overview of the UML and Its Extension Mechanism

The Unified Modeling Language (UML) is a third-generation object-oriented analysis and design notation. The first generation of these notations modeled the attributes and operations of classes and how classes were related to one another [4]. The second generation of these notations added constructs to model more development concerns, such as behavior of objects, higher-level grouping of classes, deployment and communications of these groups on distributed hosts, scheduling of operations, life-times of objects, and system usage scenarios [e.g., 3, 21, 2]. These notations and their associated methods have been widely used by practitioners and are supported by dozens of tools. For the third generation, the UML consolidates, formalizes, and standardizes many of these constructs [17].

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. Eight issues addressed by UML are: (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.

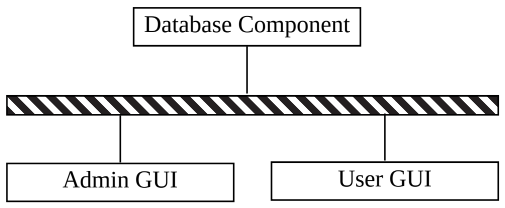


Figure 1. An example C2-style architecture

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 [19]. The syntax and semantics of the underlying model are specified semi-formally via a meta-model, descriptive text, and constraints [18]. 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 a textual expression language based on first-order predicate logic [20]. 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 bag of instances. Several examples of OCL constraints are given below.

The UML is an extensible language in that new constructs may be added to address new issues in software development. The most obvious way to extend the language is to add new elements to the meta-model. This would actually define a new language which is related to the UML but has new semantics. Within the UML itself, three constructs are provided to allow limited extension to new issues without violating 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. Using these extension mechanisms to define new constructs results in models that still comply with standard UML.

Figure 2 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 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."

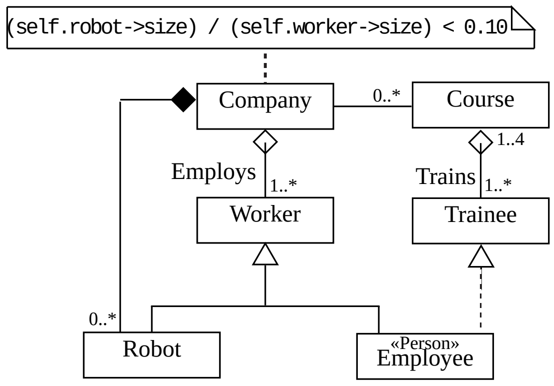


Figure 2. An example design expressed in UML

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

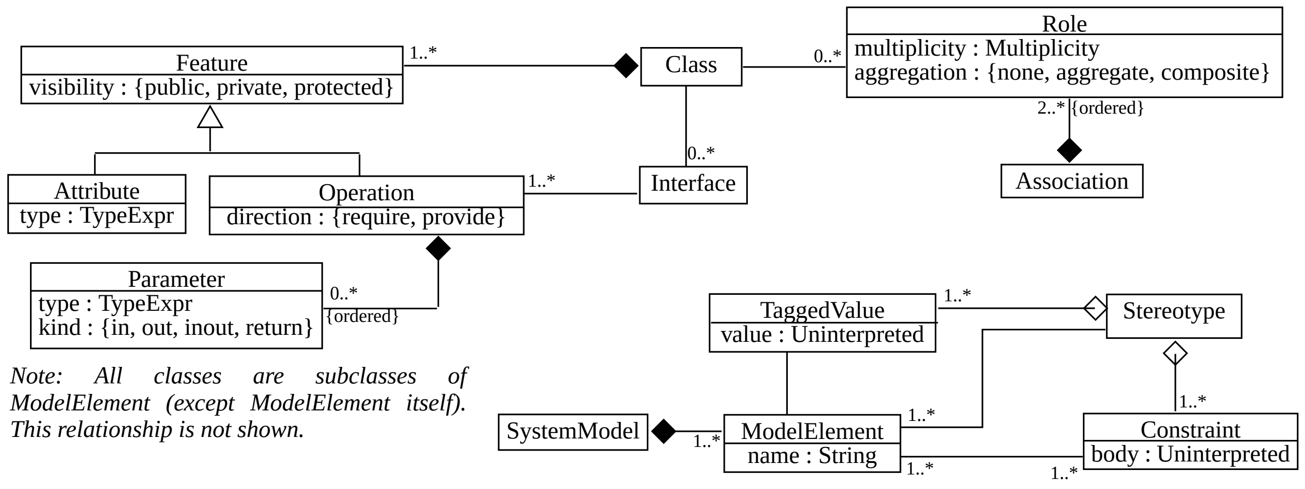
[1] If a person is in any composite relationship, it must be the composite.
```
self.oclType.role.forAll(myRole |
  myRole.association.role->exists(anyRole |
    anyRole.aggregation = composite) implies
      myRole.aggregation = composite)
```

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

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 3 shows the relevant parts of the UML meta-model. We have simplified the meta-model for purposes of illustration in this paper, but all the constraints we define can be easily rewritten for use with the complete meta-model.

EXTENDING THE UML TO MODEL C2-STYLE ARCHITECTURES

The 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 mapped to concrete components that implement them, but they are not themselves concrete. Furthermore, components in UML



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

Figure 3. Simplified UML Meta-Model (Adapted from [18])

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 extend it to express several aspects of the C2 style.

One straightforward approach to modeling C2 is to define a C2-specific meta-model which describes the syntax of individual C2 designs. This approach can be seen in part of a more comprehensive formalization of the C2 style [10]. Figure 4 shows a simple UML version of such a meta-model. Providing this meta-model by itself does help to formalize the style, but does not help integration with standard design methods. By defining our new meta-classes as subclasses of existing meta-classes we would achieve some integration. For example, making Component 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, but since it modifies the meta-

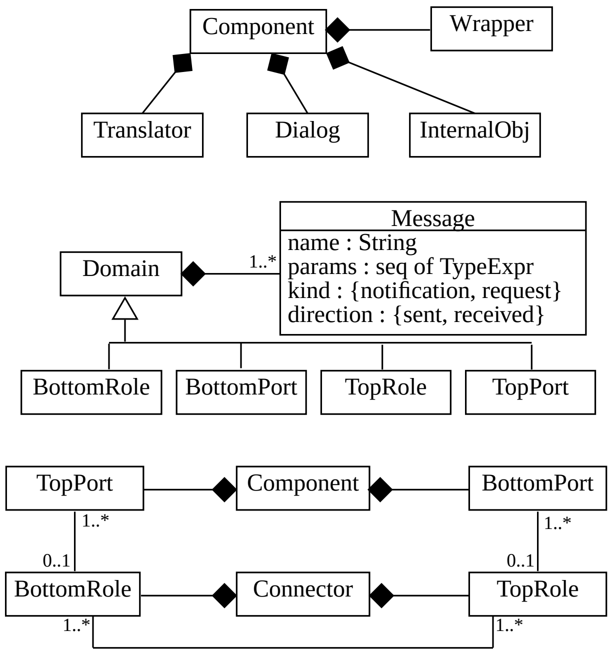


Figure 4. A meta-model for C2-style architectures

model, it does not conform to the UML standard and we cannot expect UML-compliant tools to support it. Instead we limit ourselves to using UML's built-in customization 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 first pick an existing meta-class from the UML meta-model that is semantically close to a C2 construct, and then define a stereotype that can be applied to instances of that meta-class to constrain its semantics to the C2 style. We use this strategy in each of the following subsections. Finally, we illustrate the new constructs by modeling the same example C2-style architecture used above.

C2 Operations

The UML meta-class Operation matches the C2 concept of a message specification. Operations consist of a name, parameter list (some of which may be 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 will simply 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.

```
not self.parameter->exists(p | p.kind = return)
```

C2 Components

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 operations in a C2Interface 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.direction = require) and
botInt.operation->forAll(o |
  (o.c2MsgType = request) implies
     o.direction = provide)
```

[4] Notifications travel "downward" only. This is similar to the constraint above.
[5] Each C2Component will have a single instance in the running system.
```
self.oclType.allInstances->size = 1
```

[6] C2Components participate in at most four whole-part relationships named internalObject, wrapper, dialog, and translator.
```
Let wholes = self.oclType.role->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 correspoding 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.direction = require and
        note.c2MsgType = notification))
```

C2 Connectors

C2 Connectors are in many ways similar to C2 components. One difference is that they do not have the 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.role->size = 2
```

[2] The first end of the association must be to a C2 component.
```
Let roles = self.oclType.role,
roles[1].multiplicity = "1..1" and
roles[1].class.stereotype = C2Component
```

[3] The second end of the association must be to a C2 connector.
```
Let roles = self.oclType.role,
roles[2].multiplicity = "1..1" and
roles[2].class.stereotype = C2Connector
```

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

Stereotype C2AttachConnConn for instances of meta-class Association

[1] C2 attachments are binary associations.
```
self.oclType.role->size = 2
```

[2] Each role of the association must be on a C2 connector.
```
self.oclType.role->forAll(r |
  r.multiplicity = "1..1" and
  r.class.stereotype = C2Connector)
```

[3] The two roles are not both on the same C2 connector.
```
self.oclType.role[1].class<>self.oclType.role[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.role.associa-
    tion->select(a | a.role[2] = self.oclType),
Let topsIntsBelow = downAttach.role[1].inter-
    face->select(i | i.c2pos = top),
topsIntsBelow.operation->asSet = topInt.operat-
ion->asSet
```

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

Constructs for C2 Architectures

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). As further examples, we add two new rules that guard against degenerate cases.

Stereotype C2Architecture for instances of meta-class SystemModel

[1] A C2 architecture is made up of only C2 model elements.
```
self.oclType.elements->forAll(e |
  e.stereotype = C2Component or
  e.stereotype = C2Connector or
  e.stereotype = C2AttachUnderComp or
  e.stereotype = C2AttachOverComp or
  e.stereotype = C2AttachConnConn)
```

[2] Each C2Component has at most one C2AttachOverComp.
```
Let comps = self.oclType.elements->select(e |
    e.stereotype = C2Component),
comps.role.association->select(a |
    a.stereotype = C2AttachUnderComp)->size <= 1
```

[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.elements->select(e |
    e.stereotype = C2Component),
comps.role.association->forAll(a |
  a.stereotype = C2AttachOverComp or
  a.stereotype = C2AttachUnderComp)
```

[5] C2Connectors do not participate in any non-C2 associations.
```
Let conns = self.oclType.elements->select(e |
  e.stereotype = C2Connector),
conns.role.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.elements->select(e |
    e.stereotype = C2Component),
comps.role.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.role.association->size > 0
```

Example Architecture

Figure 5 shows the UML graphical notation for the same system shown above to illustrate the C2 style (Figure 1). We show some operations and omit others as needed to clarify the discussion below. Each element is marked with its stereotype in *guillemets* (i.e., small double angle brackets). Alternatively, UML allows icons to be used to denote the stereotype.

We have defined precise modeling constructs, and there is a clear mapping between the visual elements and model elements. However, the overall visualization is in no way constrained to look the way a C2 diagram is normally intended to look. For example, a component that is above a certain connector in the model may be drawn beside it in the visualization. UML currently does not define a visualization meta-model, so there are no standard objects to constrain.

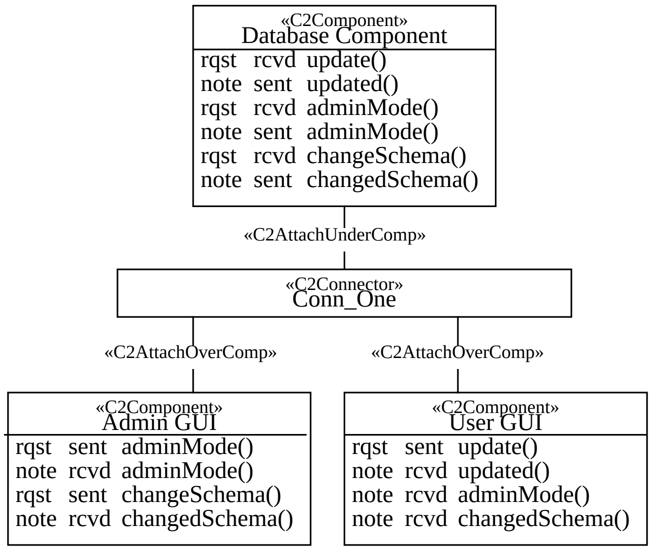


Figure 5. An example C2-style architecture for a dababase

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 6 makes explicit our assumptions about the kinds of users who will use this system and their tasks. Figure 7 is a sequence diagram showing how the system behaves in the context of a particular use case.

DISCUSSION

Modeling C2 in UML

Extending UML to enforce the C2-style rules has been fairly straightforward. 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. For example, the behavior of the dialog part of a C2 component may be modeled using a UML statechart.

Some concepts of C2 are already part of UML. Neither C2 nor UML constrain the choice of implementation language or require that any two components be implemented in the same language (although using UML with object-oriented languages is much easier than with non-object-oriented languages). 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.

Some concepts of C2 are very different from those of UML and those of object-oriented design in general. For example, mainstream object-oriented design has a strict dichotomy between classes and instances. Since each class may have

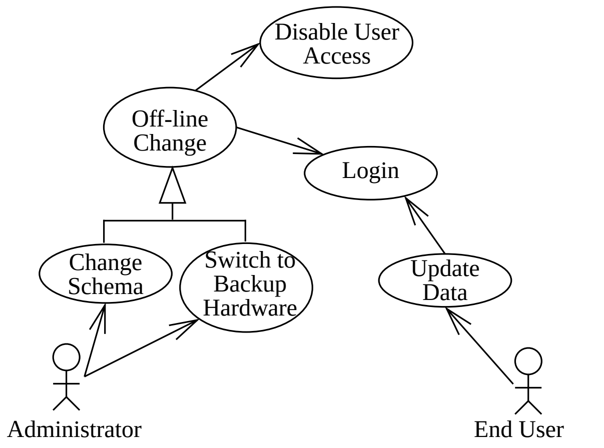


Figure 6. Some use cases for the example database system

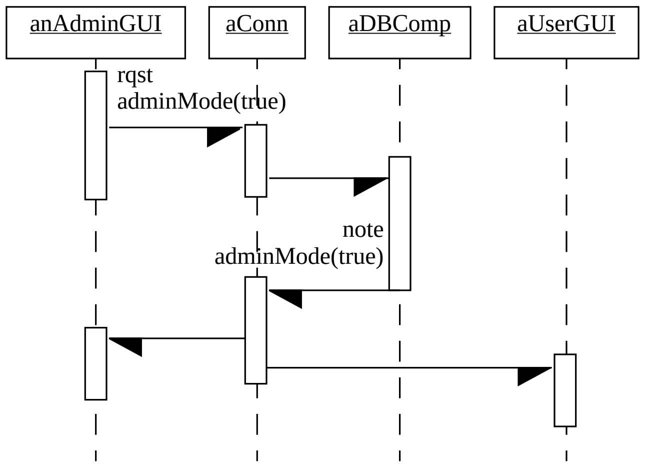


Figure 7. Sequence diagram for Disable User Access

multiple instances, associations between classes may have multiplicity greater than one (e.g., there could be any number of employees in our initial UML example). 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 considered various ways of formally addressing this difference, but each added significant complexity. Declining to state one semantic aspect of C2 does not undermine the basic advantages of our approach.

Core Models and Extensions

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 all the concepts needed. 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 integration of ADLs, rather than standardization.

ACME is an architecture interchange language that attempts to allow automatic transformation of a system modeled in one ADL to an equivalent model in another ADL [5]. This assumes that there is a meaningful equivalent, which may not be true given that different ADLs focus on different system aspects and have different semantics. Furthermore, it requires a set of semantic mappings that involve every concept of every ADL. The translation approach seems tractable only to the extent that common constructs occur in every ADL. At this point the only evident commonality is that ADLs provide syntactic elements for components and connectors, while semantic definitions of components and connectors vary across ADLs [12].

In contrast, our approach does not use translation between notations, but rather uses a core model with several independent extensions. The UML provides a starting point with more standard constructs than components and connectors. We call this our core model, and assume that developers are able to use it 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 tools and languages 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.

Future Work

Further research into this approach will attempt to extend UML to model the semantics of other ADLs, apply object-oriented concepts such as polymorphism and inheritance to architectural elements [13], and evaluate the effectiveness of the approach in practice.

Experience with modeling architectures using extended UML may lead to new and better models that relate architectural concerns. Taylor proposes that architectural research should stem from, and be evaluated in terms of, its impact on system development [23]. Grounding architectural models in a widely-used design method is one important step in bringing together the communities of theory and practice.

ACKNOWLEDGMENTS

The authors would like to acknowledge Richard Taylor, Peyman Oreizy, Nenad Medvidovic, Art Reyes, and Joseph Lee.

This material is based upon work supported by the National Science Foundation under Grant No. CCR-9624846 and Grant No. CCR-9701973. Effort sponsored by the Defense Advanced Research Projects Agency, and Rome Laboratory, Air Force Materiel Command, USAF, under agreement number F30602-97-2-0021 and F30602-94-C-0218. Additional support is provided by Rockwell International. 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. Approved for Public Release — Distribution Unlimited.

REFERENCES

1. Abowd, G., Allen, R., and Garlan, D. Using style to understand descriptions of software architecture. SIGSOFT Software Engineering
2. Awad, M., Kuusela, J. and Ziegler, J. Object-Oriented Technology for Real-Time Systems: A Practical Approach Using OMT and Fusion. Prentice-Hall, 1996.
3. Booch, G. Object-Oriented Analysis and Design with Applications. Second Edition. Addison-Wesley, 1994.
4. Coad, P. and Yourdon, E. Object Oriented Design. Yourdon Press, 1991.
5. Garlan, D., Monroe, R. and Wile D. ACME: An architectural interconnection language. Technical Report, CMU-CS-95-219, Carnegie Mellon University, November 1995.
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. Krasner, G. E. and Pope, S. T. A cookbook for using the model-view-controller user interface paradigm in smalltalk-80. Journal of Object-Oriented Programming. 1(3):26-49, August/September 1988.
8. Kruchten, P. B. The 4+1 view model of architecture. IEEE Software. Nov. 1995, 42-50.
9. Luckham, D. C., and Vera, J. An event-based architecture definition language. IEEE Transactions on Software Engineering, pages 717-734, September 1995.
10. Medvidovic N., Taylor, R. N., Whitehead, Jr. E. J. Formal modeling of software architectures at multiple levels of abstraction. In Proceedings of the California Software Symposium (CSS'96), Los Angeles, CA, April 1996.
11. Medvidovic, N. and Rosenblum, D. S. Architectural domains: A framework for characterizing architectural definition languages. Proc. USENIX Conf. on Domain Specific Languages, Santa Barbara, CA, October. 1997. To appear.
12. Medvidovic, N. and Taylor, R. N. A framework for classifying and comparing architecture description languages. To appear in The Sixth European Software Engineering Conference together with Fifth ACM SIGSOFT Symposium on the Foundations on Software Engineering. Zurich, Switzerland, September, 1997.
13. Medvidovic, N., Oreizy, P., Robbins, J. E., and Taylor, R. N. Using object-oriented typing to support architectural design in the C2 style. In Proceedings of ACM SIGSOFT'96. pp 24-32. San Fransico, CA, October 1996.
14. Moriconi M., Qian, X., and Riemenschneider, R. A. Correct architecture refinement. IEEE Transactions on Software Engineering, pages 356-372, April 1995.
15. 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.
16. Perry, D. E. and Wolf, A. L. Foundations for the study of software architectures. ACM SIGSOFT Software Engineering Notes, pages 40-52, October 1992.
17. 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/97-07-03.pdf.
18. Rational Partners. UML Semantics. Object Management Group document ad/97-07-07. July 1997. Available from http://www.omg.org/docs/ad/97-07-07.pdf.
19. Rational Partners. UML Notation Guide. Object Management Group document ad/97-07-08. July 1997. Available from http://www.omg.org/docs/ad/97-07-08.pdf.
20. Rational Software Corporation and IBM. Object constraint language specification. Object Management Group document ad/97-07-05. July 1997. Available from http://www.omg.org/docs/ad/97-07-05.pdf.
21. Rombaugh, J. et. al. Object-Oriented Modeling and Design. Prentice-Hall, 1991.
22. Soni, D., Nord, R., and Hofmeister C. Software architecture in industrial applications. International Conference on Software Engineering 17, 1995, 196-207.
23. Taylor, R. N. Generalizations from domain experience: The superior paradigm for software architecture research? In Proceedings of the Second International Software Architecture Workshop (ISAW-2), San Fransico, CA, October 1996.
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 Transactions on Software Engineering, June 1996, vol.22, no.6, 390-406.