# Assessing the Suitability of a Standard Design Method for Modeling Software Architectures

Nenad Medvidovic and David S. Rosenblum
Department of Information and Computer Science
University of California, Irvine
Irvine, CA 92697-3425, U.S.A.
{neno,dsr}@ics.uci.edu

***Abstract*** *— Software architecture descriptions are high-level models of software systems. Most existing special-purpose architectural notations have a great deal of expressive power but are not well integrated with common development methods. Conversely, mainstream development methods are accessible to developers, but lack the semantics needed for extensive analysis. In our previous work, we described an approach to combining the advantages of these two ways of modeling architectures. While this approach suggested a practical strategy for bringing architectural modeling into wider use, it introduced specialized extensions to a standard modeling notation, which could also hamper wide adoption of the approach. This paper attempts to assess the suitability of a standard design method "as is" for modeling software architectures.*

***Keywords*** *— Software architecture, architectural style, object-oriented design, architecture description languages, Unified Modeling Language*

## 1. INTRODUCTION

Software architecture is an aspect of software engineering directed at reducing the costs of developing applications and increasing the potential for commonality among different members of a closely related product family [6, 19]. Software development based on common architectural idioms has its focus shifted from lines-of-code to coarser-grained architectural elements and their overall interconnection structure. This enables developers to abstract away the unnecessary details and focus on the "big picture:" system structure, high level communication protocols, assignment of software components and connectors to hardware components, development process, and so on [6, 7, 9, 19, 28, 29]. The basic promise of software architecture research is that better software systems can result from modeling their important aspects during, and especially early in the 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 has focused on analytic evaluation of architecture descriptions. Many researchers have come to believe that, to obtain the benefits of an architectural focus, software architecture must be provided with its own body of specification languages and analysis techniques [3, 5, 32]. Such languages are needed to demonstrate properties of a system upstream, thus minimizing the costs of errors. They are also needed to provide abstractions that are adequate for modeling a large system, while ensuring sufficient detail for establishing properties of interest. A large number of architecture description languages (ADLs) has been proposed [2, 4, 9, 10, 12, 17, 25, 31].

Each ADL embodies a particular approach to the specification and evolution of an architecture. Answering specific evaluation questions demands powerful, specialized modeling and analysis techniques that address specific aspects in depth. 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 has focused on modeling a wide range of issues that arise in software development, perhaps with 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. 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 there now exists a concerted effort to standardize methods for object-oriented analysis and design [18].

In our previous work, we described an approach to combining the advantages of specialized, highly formal methods of modeling architectures with general, less formal design methods [24]. This approach suggested a practical strategy for bringing architectural modeling into wider use, namely by incorporating substantial elements of architectural models into a standard design method, the Unified Modeling Language (UML) [20]. However, our technique is not without drawbacks: for each architectural approach and ADL, we introduced a somewhat specialized extension to UML. In particular, we relied heavily on UML's Object Constraint Language (OCL) [23] to specify architecture- and ADL-specific concepts.

OCL constraints are highly formal. Their formality may hamper wide adoption of our technique, although end users of the enhanced UML meta-model typically will not need to write OCL constraints. Furthermore, OCL is a part of the standard UML definition and it is expected that standardized UML tools will be able to process it. However, OCL is considered an uninterpreted part of UML and UML tools may not support it to the extent needed for creating, manipulating, analyzing, and evolving designs. For this reason, in this paper we attempt to assess the suitability of UML "as is" for modeling software architectures. In particular, we focus on one of the architectural approaches we addressed previously [24], the C2 architectural style [29]. We use a simple meeting scheduler application to highlight the issues. In the process, we attempt to shed light on the relationship between architecture and design.

The paper is organized as follows. The next section briefly describes UML. Section 3 briefly describes the example application, a meeting scheduler, used to illustrate our arguments throughout the paper. In Section 4, we introduce the C2 style and discuss a possible C2 architecture for the meeting scheduler application. Section 5 provides a "C2 style" UML design of the meeting scheduler. We discuss the results and key lessons learned in Section 6. Our conclusions round out the paper.

## 2. OVERVIEW OF UML

### 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 are more problem-oriented and generic, whereas high-fidelity models tend to be used later and are 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.

UML is a graphical language with fairly 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 [22]. The syntax and semantics of the underlying model are specified semi-formally via a meta-model, descriptive text, and constraints [21]. 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.

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

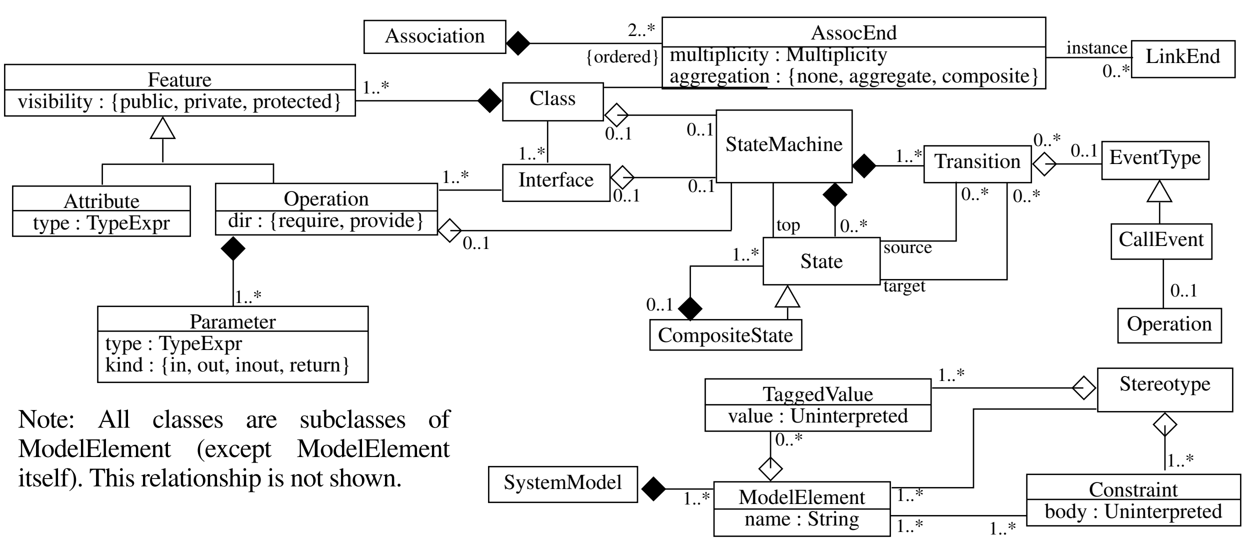


**Figure 1.** Simplified UML Meta-Model (Adapted from [21])

and applied to other model elements; the semantic effect is as if the constraints and tagged values were applied directly to those elements. Another possible extension mechanism is to modify the meta-model, but this approach results in a completely new notation to which standard UML tools cannot be applied. We discuss this approach in more detail in Section 2.2.

Figure 1 shows the parts of the UML meta-model used in this paper. We have simplified the meta-model for purposes of illustration.

### 2.2. Our Strategy for Adapting UML for Architecture Modeling

In [24] we studied two possible approaches to using UML to model architectures. One approach is to define an ADL-specific meta-model. This approach has been used in more comprehensive formalizations 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.

The approach for which we opted instead was to restrict ourselves to using UML's built-in extension mechanisms on existing meta-classes [24]. This allows the use of existing and future UML-compliant tools to represent the desired architectural models, and to support architectural style conformance checking when OCL-compliant tools become available. Our basic strategy was to first choose an existing meta-class from the UML meta-model that is semantically close to an ADL construct, and then define a stereotype that can be applied to instances of that meta-class to constrain its semantics to that of the ADL.

Neither of the two approaches answers the deeper question of UML's suitability for modeling software architectures "as is," i.e., without defining meta-models specific to a particular architectural approach or extending the existing UML meta-model. Such an exercise would highlight the respective advantages of special- and general-purpose design notations in modeling architectures. It also has the potential to further clarify the relationship between software architecture and design. Therefore, in this paper we study the characteristics of using the existing UML features to model architectures in a particular style, C2.

## 3. EXAMPLE APPLICATION

The example we selected to motivate the discussion in this paper is a simplified version of the meeting scheduler problem, initially proposed by van Lamsweerde and colleagues [8] and recently considered as a candidate model problem in software architectures [27]. We have chosen this problem partly because of our prior experience with designing and implementing a distributed meeting scheduler in the C2 architectural style, described in [29].

Meetings are typically arranged in the following way. A meeting initiator asks all potential meeting attendees for a set of dates on which they cannot attend the meeting (their "exclusion" set) and a set of dates on which they would prefer the meeting to take place (their "preference" set). The exclusion and preference sets are contained in some time interval prescribed by the meeting initiator (the "date range").

The initiator also asks active participants to provide any special equipment requirements on the meeting location (e.g., overhead-projector, workstation, network connection, telephones); the initiator may also ask important participants to state preferences for the meeting location.

The proposed meeting date should belong to the stated date range and to none of the exclusion sets. It should also ideally belong to as many preference sets as possible. A date conflict occurs when no such date can be found. A conflict is strong when no date can be found within the date range and outside all exclusion sets; it is weak when dates can be found within the date range and outside all exclusion sets, but no date can be found at the intersection of all preference sets. Conflicts can be resolved in several ways:

- the initiator extends the date range;
- some participants expand their preference set or narrow down their exclusion set; or
- some participants withdraw from the meeting.

## 4. MODELING THE EXAMPLE APPLICATION IN C2

### 4.1. Overview of C2

C2 is a software architectural style for user interface intensive systems [29]. C2SADEL is an ADL for describing C2-style architectures [12, 15]; henceforth, in the interest of clarity, we use "C2" to refer to the combination C2 and C2SADEL. 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.

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(s) in which the components or connectors are implemented, whether or not components execute in their own threads of control, the deployment of components to hosts, or the communication protocol(s) used by connectors.

### 4.2. Modeling the Meeting Scheduler in C2

Figure 2 shows a graphical depiction of a possible C2-style architecture for a simple meeting scheduler system. This system consists of components supporting the functionality of a *MeetingInitiator* and several potential meeting *Attendees* and *ImportantAttendees*. Three C2 connectors are used to route messages among the components. Certain messages from the *Initiator* are sent both to *Attendees* and *ImportantAttendees*, while others (e.g., to obtain meeting location preferences) are only routed to *ImportantAttendees*. Since a C2 component has only one communication port on its top and one on its bottom, and all message routing functionality is relegated to connectors, it is the responsibility of *MainConn* to ensure that *AttConn* and *ImportantAttConn* above it receive only those message relevant to their respective attached components.

The *Initiator* component sends requests for meeting information to *Attendees* and *ImportantAttendees*. The two sets of components notify the *Initiator* component, which attempts to schedule a meeting and either requests that each potential attendee mark it in his/her calendar (if the meeting can be scheduled), or it sends other requests to attendees to extend the date range, remove a set of excluded dates, add preferred dates, or withdraw from the meeting. Each *Attendee* and *ImportantAttendee* component, in turn, notifies the *Initiator* of its date, equipment, and location preferences, as well as excluded dates. *Attendee* and *ImportantAttendee* components cannot make requests of the *MeetingInitiator* component, since they are above it in the architecture.

Most of this information is implicit in the graphical view of the architecture shown in Figure 2. For this reason, we

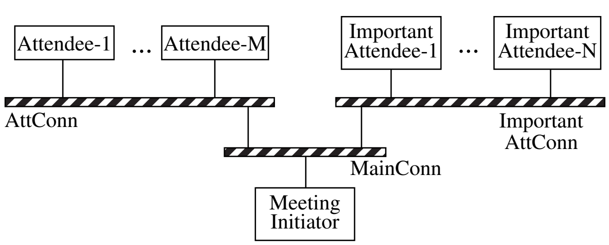


**Figure 2.** A C2-style architecture for a meeting scheduler system.

specify the architecture in C2SADEL, a textual language for modeling C2-style architectures [11, 12, 15]. For simplicity, we assume that all attendees' equipment needs will be met, and that a meeting location will be available on the given date and that it will be satisfactory for all (or most) of the important attendees.

The *MeetingInitiator* component is specified below. The component only communicates with other parts of the architecture through its top port.

```
component MeetingInitiator is
  interface
    top_domain is
      out
        GetPrefSet ();
        GetExclSet ();
        GetEquipReqts ();
        GetLocPrefs ();
        RemoveExclSet ();
        RequestWithdrawal (to Attendee);
        RequestWithdrawal (to ImportantAttendee);
        AddPrefDates ();
        MarkMtg (d : date; l : loc_type);
      in
        PrefSet (p : date_rng);
        ExclSet (e : date_rng);
        EquipReqts (eq : equip_type);
        LocPref (l : loc_type);
    bottom_domain is
      out null;
      in null;
  parameters null;
  methods
    procedure Start ();
    procedure Finish ();
    procedure SchedMtg (p : set date_rng;
                        e : set date_rng);
    procedure AddPrefSet (pref : date_rng);
    procedure AddExclSet (exc : date_rng);
    procedure AddEquipReqts (eq : equip_type);
    procedure AddLocPref (l : loc_type);
    function AttendInfoCompl () return boolean;
    procedure IncNumAttends (n : integer);
    function GetNumAttends () : return integer;
  behavior
    startup
      invoke_methods Start;
      always_generate GetPrefSet, GetExclSet,
        GetEquipReqts, GetLocPrefs;
    cleanup
      invoke_methods Finish;
      always_generate null;
    received_messages PrefSet;
      invoke_methods AddPrefSet, IncNumAttends,
        AttendInfoCompl, GetNumAttends, SchedMtg;
      may_generate RemoveExclSet xor
        RequestWithdrawal xor MarkMtg;
    received_messages ExclSet;
      invoke_methods AddExclSet, AttendInfoCompl,
        GetNumAttends, SchedMtg;
      may_generate AddPrefDates xor RemoveExclSet
        xor RequestWithdrawal xor MarkMtg;
    received_messages EquipReqts;
      invoke_methods AddEquipReqts,
        AttendInfoCompl, GetNumAttends, SchedMtg;
      may_generate AddPrefDates xor RemoveExclSet
        xor RequestWithdrawal xor MarkMtg;
    received_messages LocPref;
      invoke_methods AddLocPref;
      always_generate null;
  context
    bottom_most computational_unit;
end MeetingInitiator;
```

The *Attendee* and *ImportantAttendee* components receive meeting scheduling requests from the *Initiator* and notify it of the appropriate information. The two types of components only communicate with other parts of the architecture through their bottom ports.

```
component Attendee is
   interface
      top_domain is
         out null;
         in null;
      bottom_domain is
         out
            PrefSet (p : date_rng);
            ExclSet (e : date_rng);
            EquipReqts (eq : equip_type);
            Witdrawn ();
         in
            GetPrefSet ();
            GetExclSet ();
            GetEquipReqts ();
            RemoveExclSet ();
            RequestWithdrawal ();
            AddPrefDates ();
            MarkMtg (d : date; l : loc_type);
   parameters null;
   methods
      procedure Start ();
      procedure Finish ();
      procedure NoteMtg (d : date; l : loc_type);
      function DeterminePrefSet () return date_rng;
      function DetermineExclSet () return date_rng;
      function AddPrefDates () return date_rng;
      function RemoveExclSet () return date_rng;
      procedure DetermineEquipReqts (eq : equip_type);
   behavior
      startup
         invoke_methods Start;
         always_generate null;
      cleanup
         invoke_methods Finish;
         always_generate null;
      received_messages GetPrefSet;
         invoke_methods DeterminePrefSet;
         always_generate PrefSet;
      received_messages AddPrefDates;
         invoke_methods AddPrefDates;
         always_generate PrefSet;
      received_messages GetExclSet;
         invoke_methods DetermineExclSet;
         always_generate ExclSet;
      received_messages GetEquipReqts;
         invoke_methods DetermineEquipReqts;
         always_generate EquipReqts;
      received_messages RemoveExclSet;
         invoke_methods RemoveExclSet;
         always_generate ExclSet;
      received_messages RequestWithdrawal;
         invoke_methods Finish;
         always_generate Withdrawn;
      received_messages MarkMtg;
         invoke_methods NoteMtg;
         always_generate null;
   context
      top_most computational_unit;
end Attendee;
```

*ImportantAttendee* is a specialization of the *Attendee* component: it duplicates all of *Attendee*'s functionality and adds specification of meeting location preferences. *ImportantAttendee* is thus specified as a subtype of *Attendee* that preserves its interface and behavior, but can implement that behavior in a new manner.

```
component ImportantAttendee is
   subtype Attendee (int and beh)
   interface
      bottom_domain is
         out
            LocPrefs (l : loc_type);
         in
            GetLocPrefs ();
   methods
      function DetermineLocPrefs () return loc_type;
   behavior
      received_messages GetLocPrefs;
         invoke_methods DetermineLocPrefs;
         always_generate LocPrefs;
end ImportantAttendee;
```

The *MeetingScheduler* architecture depicted in Figure 2 is shown below. The architecture is specified with conceptual components (i.e., component types). Each conceptual component (e.g., *Attendee*) can be instantiated multiple times in a *system*.

```
architecture MeetingScheduler is
   conceptual_components
      top_most
         Attendee;
         ImportantAttendee;
      internal null;
      bottom_most
         MeetingInitiator;
   connectors
      connector MainConn is
         message_filter no_filtering;
      end MainConn;
      connector AttConn is
         message_filter no_filtering;
      end AttConn;
      connector ImportantAttConn is
         message_filter no_filtering;
      end ImportantAttConn;
   architectural_topology
      connector AttConn connections
         top_ports
            Attendee;
         bottom_ports
            MainConn;
      connector ImportantAttConn connections
         top_ports
            ImportantAttendee;
         bottom_ports
            MainConn;
      connector MainConn connections
         top_ports
            AttConn;
            ImportantAttConn;
         bottom_ports
            MeetingInitiator;
end MeetingScheduler;
```

An instance of the architecture (a system) is specified by instantiating the components. For example, an instance of the meeting scheduler application with three participants and two important participants is specified as follows.

```
system MeetingScheduler_1 is
   architecture MeetingScheduler with
         Attendee instance Att_1, Att_2, Att_3;
         ImportantAttendee instance ImpAtt_1, ImpAtt_2;
         MeetingInitiator instance MtgInit_1;
end MeetingScheduler_1;
```

## 5. MODELING THE C2-STYLE MEETING SCHEDULER APPLICATION IN UML

The process of designing a C2-style application in UML should be driven and constrained both by the rules of C2 and the modeling features available in UML. The two must be considered simultaneously. For this reason, the initial steps in this process are to develop a domain model for a given application in UML and an informal C2 architectural diagram, such as the one from Figure 2. Such an architectural diagram is key to making the appropriate mappings between classes in the domain and architectural components. Furthermore, it points to the need to explicitly model connectors in any C2-style architecture. Another important aspect of C2 architectures is the prominence of components' message interfaces. This is reflected in a UML design by modeling interfaces explicitly and independently of the classes that will implement those interfaces.

Our initial attempt at a UML class diagram for the meeting scheduler application is shown in Figure 3. The diagram shows the domain model for the meeting scheduler application consisting of the domain classes, their inheritance relationships, and their associations. The diagram abstracts away many architectural details, such as the mapping of classes in the domain to implementation components, the order of interactions among the different classes, and so forth. Furthermore, much of the semantics of class interaction is missing from the diagram. For example, the *Invites* association associates two *Meetings* with one or more *Attendees* and one *MeetingInitiator*. However, the association does not make clear the fact that the two *Meetings* are intended to represent a range of possible meeting dates, rather than a pair of related meetings.

Each class exports one or more interfaces, shown in Figure 4. The *ImportantMtgInit* and *ImportantMtgAttend* interfaces inherit from the *MtgInit* and *MtgAttend* interfaces, respectively. The only difference is the added operation to request and notify of location preferences.

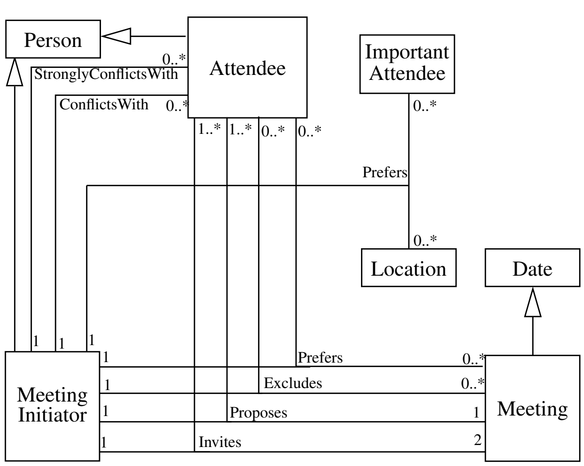


**Figure 3.** UML class diagram for the meeting scheduler application. Details (attributes and methods) of each individual class have been suppressed for clarity.

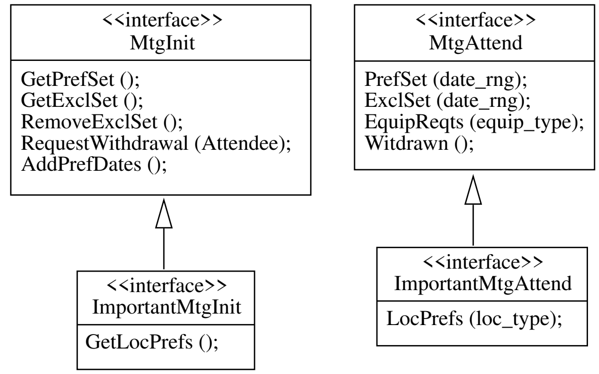


**Figure 4.** Meeting scheduler class interfaces.

Note that every interface element corresponds to a C2 message in the architecture specified in Section 4.2. All methods in the UML design will be implemented as asynchronous message passes, as they would in C2. Since C2 components communicate via implicit invocation, C2 messages do not have return values; this is also reflected in Figure 4.

In order to model a C2 architecture in UML, connectors must be defined. Although connectors fulfill a role different from components, they can also be modeled with UML classes. However a C2 connector is by definition generic and can accommodate any number ant type of C2 components; informally, the interface of a C2 connector is a union of the interfaces of its attached components. UML does not support this form of genericity, so that the connectors specified in UML have to be application-

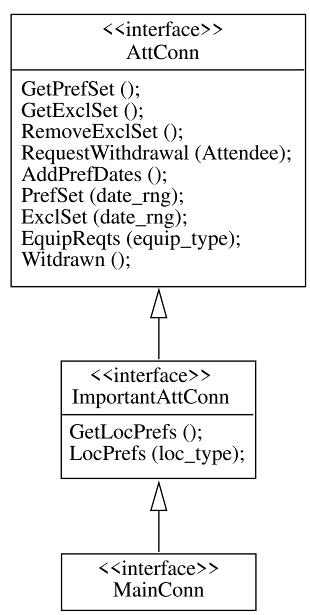


**Figure 5.** Application-specific UML classes representing C2 connectors.

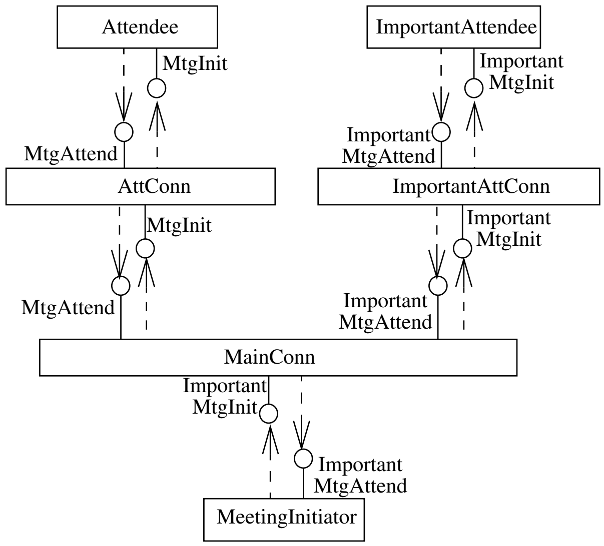


**Figure 6.** UML class diagram for the meeting scheduler application designed in the C2 architectural style.

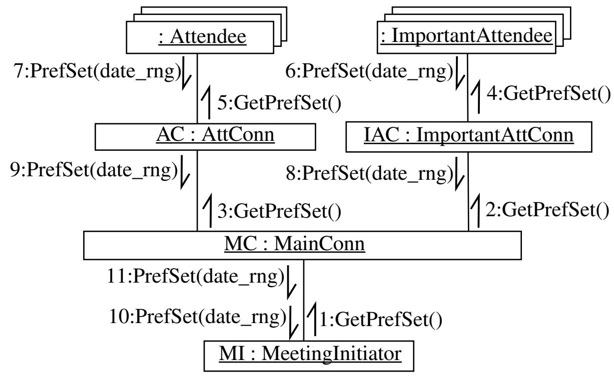


**Figure 7.** Collaboration diagram for the meeting scheduler application showing a response to a request issued by the *MeetingInitiator* to both *Attendees* and *ImportantAttendees*.

specific. For that purpose, the connectors for the meeting scheduler application share the components' interfaces. Each connector can be thought of as a simple class that forwards each message it receives to the appropriate components. Therefore, while the component class interface specifications, shown in Figure 4, correspond to the different C2 components' outgoing messages (i.e., their provided functionality), the connector interfaces are routers of both the incoming and outgoing messages, as depicted in Figure 5. Connectors do not add any functionality at the domain model level; we have thus chosen to omit them from the class diagram in Figure 3.

A refined class diagram for the meeting scheduler application is shown in Figure 6. The *Attendee* and *ImportantAttendee* classes are related by interface inheritance, which is depicted in Figure 4, but is only implicit in Figure 6 (and altogether omitted from Figure 3). We have omitted from Figure 6 the *Location*, *Meeting*, and *Date* classes shown in Figure 3, since they have not been impacted. We have also omitted the two superclasses for the components and connectors (*Person* and *Conn*, respectively).

Note that the class diagram in Figure 6 is similar in its structure to the C2 architecture depicted in Figure 2. The only difference is that the diagram in Figure 2 depicts instances of the different components and connectors, while a UML class diagram depicts classes and their associations. UML provides several types of diagrams that depict class instances (objects). One candidate is UML's object diagrams; however, we choose to depict a collaboration diagram to further draw the contrast between UML and C2.

Figure 7 shows the collaboration between an instance of the *MeetingInitiator* class (*MI*) and any instances of *Attendee* and *ImportantAttendee* classes: *MI* issues a request for a set of preferred meeting dates; *MC*, an instance of the *MainConn* class routes the request to instances of both connectors above it, *AC* and *IAC*, which, in turn, route the requests to all components attached on their top sides; each participant component chooses a preferred date and notifies any components below it of that choice; these notification messages will eventually be routed to *MI* via the connectors. Note that, if *MI* had sent the request to get meeting location preferences (*GetLocPrefs* in the *ImportantMtgInit* interface in Figure 4), *MC* would have routed them only to *IAC* and none of the instances of the *Attendee* class would have received that request.

The above diagrams, and particularly Figure 6, differ from a C2 architecture in that they explicitly specify only the messages a component receives (via interface attachments to a component rectangle). UML also allows specification of messages a component sends; we believe those messages to be obvious from the diagram and have thus chosen to omit them to simplify the diagrams.

# 6. DISCUSSION

The exercise of modeling a C2-style architecture in UML has been fairly successful. Part of the success can be attributed to the fact that many architectural concepts are found in UML (e.g., interfaces, component associations, behavioral modeling, and so forth). On the other hand, the modeling capabilities provided by UML do not always fully satisfy the needs of architectural description. We discuss several major similarities and differences in this section.

## 6.1. Software Modeling Philosophies

Neither C2 nor UML constrain the choice of implementation language or require that any two components be implemented in the same language or thread of control. C2 limits communication to asynchronous message passing and UML supports this restriction. Both C2 and UML 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 (components, connectors, communication ports, and so forth) in our UML specification, we believe that many of those aspects could be modeled with UML's sequence, collaboration, statechart, and activity diagrams. Existing ADLs, including C2SADEL, are often not able to support all of these kinds of semantic models [14].

### 6.2. Assumptions

Like any notation, UML embodies its developers' assumptions about its intended usage. "Architecting" a system was not an intended use of UML. While one can indeed focus on the different perspectives when modeling a system (discussed above), a software architect may find that the support for those perspectives found in UML only partially satisfies his/her needs.

For example, in modeling the collaboration among C2 components shown in Figure 7, we were forced to assign a relative ordering to messages in the architecture. In effect, since all C2 components and connectors can execute in their own thread(s) of control, such an ordering cannot always be determined. Indeed, it is possible that message 4 would be sent before message 3.

### 6.3. Problem Domain Modeling

UML supports modeling a problem domain, as we have briefly shown in this paper. A C2 architectural model, however, often hides some of the information present in a domain model. For example, meeting, equipment, and location information is present in Figure 3, but is missing from the C2 architecture specified in Section 4 and its corresponding UML diagram in Figure 6. Modeling all the relevant information early in the development lifecycle is crucial to the success of a software project. Therefore, a domain model should be considered a separate and useful architectural perspective [13, 30].

### 6.4. Architectural Abstractions

Some concepts of C2, and software architectures in general, are very different from those of UML and object-oriented design in general. Connectors are first-class entities in C2. While the functionality of a connector can typically be abstracted by a class/component [9, 10], C2 connectors have the added property that their interfaces are context reflective. This property is designed into C2SADEL and C2's implementation infrastructure [16] for all connectors, whereas the approach described in this paper requires specialized modeling of application-specific connector classes in UML.

The underlying problem is even deeper. Although UML may provide modeling power equivalent to or surpassing that of an ADL, the abstractions it provides may not match an architect's mental model of the system as faithfully as the architect's ADL of choice. If the primary purpose of a language is to provide a vehicle of expression that matches the intuitions and practices of users, then that language should aspire to reflect those intentions and practices [26]. We believe this to be a key issue and one that argues against considering a notation like UML to be a "mainstream" ADL: a given language (e.g., UML) offers a set of abstractions that an architect uses as design tools; if certain abstractions (e.g., components and connectors) are buried in others (e.g., classes), the architect's job is made more (and unnecessarily) difficult; separating components from connectors, raising them both to visibility as top-level abstractions, and endowing them with certain features and limitations also raises them in the consciousness of the designer.

### 6.5. Architectural Styles

Architecture is the appropriate level of abstraction at which rules of a compositional style (i.e., an architectural style) can be exploited and should be elaborated. Doing so results in a set of heuristics that, if followed, will guarantee a resulting system certain desirable properties.

Standard UML provides no support for architectural styles. The rules of different styles have to be built into UML by constraining its meta-model, as we have done previously [24]. Therefore, in choosing to use UML "as is", we have removed one shortcoming of our previous approach, only to introduce another. In particular, every C2 architecture designed in the manner we described in this paper adheres to the UML meta-model and, as such, can be understood by a typical UML user and manipulated with standardized UML tools. On the other hand, the process of modeling a C2 architecture in UML is heuristic- rather than constraint-driven. Therefore, there is no guarantee that the designer will always adhere to the rules of C2. For this reason, it may also be more difficult to provide support for automated translation of "C2-style" UML designs into C2SADEL for C2-specific manipulations.

## 7. CONCLUSIONS

We found this initial attempt at modeling a C2-style architecture in UML useful. It highlighted those UML characteristics that show potential for aiding architectural modeling, but also pointed out some of UML's shortcomings in this regard. This experience can also serve as a solid basis for further study, both with other C2 architectures, as well as with other ADLs (e.g., Wright [2]) and architectural styles (e.g., client-server).

Before we can draw definitive conclusions about the relative merits of this approach and the approach described in our previous work [24], further research into the techniques described in the two papers is needed. One necessary step to integrate UML with other ADLs discussed in [24]: Wright [2], Darwin [10], and Rapide [9]. Each of these ADLs has certain aspects in common with UML; these were expressed with UML's extension mechanisms. We intend to investigate whether they can also be expressed in UML without extensions.

Our experience to date indicates that adapting UML to address architectural concerns requires reasonable effort, has the potential to be a useful complement to ADLs and their analysis tools, and may 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 the comparison of ADLs because it forces some implicit assumptions to be explicitly stated in common terms.

## 8. ACKNOWLEDGEMENTS

We wish to thank J. Robbins and D. Redmiles for their insights into the issues in integrating UML with architectures and their collaboration on other aspects of this work.

Effort sponsored by the Defense Advanced Research Projects Agency, and Air Force Research Laboratory, Air Force Materiel Command, USAF, under agreement number F30602-97-2-0021 and by the Air Force Office of Scientific Research, Air Force Materiel Command, USAF, under grant number F49620-98-1-0061. This material is also partially based on work supported by the National Science Foundation under Grant No. CCR-9701973. 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, Air Force Research Laboratory, Air Force Office of Scientific Research or the U.S. Government.

## 9. REFERENCES

1. G. Abowd, R. Allen, and D. Garlan. Formalizing style to understand descriptions of software architecture. *ACM Transactions on Software Engineering and Methodolog*y, pp. 319-364, October 1995.
2. R. Allen and D. Garlan. A Formal Basis for Architectural Connection. *ACM Transactions on Software Engineering and Methodology*, pp. 213-249, July 1997.
3. D. Garlan, editor. *Proceedings of the First International Workshop on Architectures for Software Systems*, Seattle, WA, April 1995.
4. D. Garlan, R. Allen, and J. Ockerbloom. Exploiting Style in Architectural Design Environments. In *Proceedings of SIGSOFT'94: Foundations of Software Engineering*, pp. 175-188, New Orleans, Louisiana, USA, December 1994.
5. D. Garlan, F. N. Paulisch, and W. F. Tichy, editors. Summary of the Dagstuhl Workshop on Software Architecture, February 1995. Reprinted in *ACM Software Engineering Notes*, pp. 63-83, July 1995.
6. D. Garlan and M. Shaw. *An introduction to software architecture: Advances in software engineering and knowledge engineering*, volume I. World Scientific Publishing, 1993.
7. P. B. Kruchten. The 4+1 view model of architecture. *IEEE Software*, pp. 42-50, November 1995.
8. A. van Lamsweerde, R. Darimont and P. Massonet. The Meeting Scheduler System: Preliminary Definition. University of Louvain, Unité d'informatique, B-1348 Louvain-la-Neuve (Belgium), October 1992.
9. D. C. Luckham and J. Vera. An event-based architecture definition language. *IEEE Transactions on Software Engineering*, pp. 717-734, September 1995.
10. J. Magee and J. Kramer. Dynamic structures in software architecture. In *Proceedings of ACM SIGSOFT'96: Fourth Symposium on the Foundations of Software Engineering (FSE4)*, pp. 3-14, San Francisco, CA, October 1996.
11. N. Medvidovic. ADLs and Dynamic Architecture Changes. In A. L. Wolf, ed., *Proceedings of the Second International Software Architecture Workshop (ISAW-2)*, pp. 24-27, San Francisco, CA, October 1996.
12. N. Medvidovic, R. N. Taylor, and E. J. Whitehead, Jr. Formal Modeling of Software Architectures at Multiple Levels of Abstraction. In *Proceedings of the California Software Symposium 1996*, pp. 28-40, Los Angeles, CA, April 1996.
13. N. Medvidovic and D. S. Rosenblum. Domains of Concern in Software Architectures and Architecture Description Languages. In *Proceedings of the USENIX Conference on Domain Specific Languages*, pp. 199-212, Santa Barbara, CA, October. 1997.
14. N. Medvidovic and R. N. Taylor. A Framework for Classifying and Comparing Architecture Description Languages. In *Proceedings of the Sixth European Software Engineering Conference together with Fifth ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pp. 60-76, Zurich, Switzerland, September 22-25, 1997.
15. N. Medvidovic, P. Oreizy, J. E. Robbins, and R. N. Taylor. Using object-oriented typing to support architectural design in the C2 style. In *Proceedings of ACM SIGSOFT'96: Fourth Symposium on the Foundations of Software Engineering (FSE4)*, pp. 24-32, San Francisco, CA, October 1996.
16. N. Medvidovic, P. Oreizy, and R. N. Taylor. Reuse of Off-the-Shelf Components in C2-Style Architectures. In *Proceedings of the 1997 Symposium on Software Reusability (SSR'97)*, pages 190-198, Boston, MA, May 1997. Also in *Proceedings of the 1997 International Conference on Software Engineering (ICSE'97)*, pages 692-700, Boston, MA, May 1997.
17. M. Moriconi, X. Qian, and R. A. Riemenschneider. Correct Architecture Refinement. *IEEE Transactions on Software Engineering*, pp. 356-372, April 1995.
18. 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.

19. D. E. Perry and A. L. Wolf. Foundations for the Study of Software Architectures. *ACM SIGSOFT Software Engineering Notes*, pp. 40-52, October 1992.

20. 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/.

21. 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.

22. 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.

23. 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/.

24. J. E. Robbins, N. Medvidovic, D. F. Redmiles, and D. S. Rosenblum. Integrating Architecture Description Languages with a Standard Design Method. In *Proceedings of the 20th International Conference on Software Engineering (ICSE'98)*, pp. 209-218, Kyoto, Japan, April 19-25, 1998.

25. M. Shaw, R. DeLine, D. V. Klein, T. L. Ross, D. M. Young, and G. Zelesnik. Abstractions for Software Architecture and Tools to Support Them. *IEEE Transactions on Software Engineering*, pp. 314-335, April 1995.

26. M. Shaw and D. Garlan. Formulations and Formalisms in Software Architecture. Jan van Leeuwen, editor, *Computer Science Today: Recent Trends and Developments*, Springer-Verlag Lecture Notes in Computer Science, Volume 1000, 1995.

27. M. Shaw, D. Garlan, R. Allen, D. Klein, J. Ockerbloom, C. Scott, M. Schumacher. Candidate Model Problems in Software Architecture. Unpublished manuscript, November 1995. Available from http://www.cs.cmu.edu/afs/cs/project/compose/www/html/ModProb/.

28. D. Soni, R. Nord, and C. Hofmeister. Software Architecture in Industrial Applications. In Proceedings of the 17th International Conference on Software Engineering, pp. 196-207, Seattle, WA, April 1995.

29. R. N. Taylor, N. Medvidovic, K. M. Anderson, E. J. Whitehead, Jr., J. E. Robbins, K. A. Nies, P. Oreizy, and D. L. Dubrow. A Component- and Message-Based Architectural Style for GUI Software. IEEE Transactions on Software Engineering, pp. 390-406, June 1996.

30. W. Tracz. DSSA (Domain-Specific Software Architecture) Pedagogical Example. *ACM SIGSOFT Software Engineering Notes*, July 1995.

31. S. Vestal. MetaH Programmer's Manual, Version 1.09. Technical Report, Honeywell Technology Center, April 1996.

32. A. L. Wolf, editor. *Proceedings of the Second International Software Architecture Workshop (ISAW-2)*, San Francisco, CA, October 1996.