Modeling Software Architectures in the Unified Modeling Language

Nenad Medvidovic
Computer Science Department
University of Southern California
Los Angeles, CA 90089-0781
http://sunset.usc.edu/~neno/

David S. Rosenblum, **Jason E. Robbins**, **David F. Redmiles**
Department of Information and Computer Science
University of California, Irvine
Irvine, CA 92697-3425
http://www.ics.uci.edu/{~dsr,~jrobbins,~redmiles}/

Abstract. The Unified Modeling Language (UML) is a family of design notations that is rapidly becoming a de facto standard software design language. UML provides a variety of useful capabilities to the software designer, including multiple, inter-related design views, a semi-formal semantics expressed as a UML meta model, and an associated language for expressing formal logic constraints on design elements. However, UML currently lacks support for capturing and exploiting certain *architectural concerns* whose importance has been demonstrated through the research and practice of software architectures. In particular, UML lacks direct support for modeling and exploiting *architectural styles*, *explicit software connectors*, and local and global *architectural constraints*. This paper presents two strategies for supporting such architectural concerns within UML. One strategy involves using UML "as is," while the other incorporates useful features of existing architecture description languages (ADLs) as UML extensions. We discuss the applicability, strengths, and weaknesses of the two strategies. The strategies are applied on three ADLs that, as a whole, represent a broad cross-section of present-day ADL capabilities.

Categories and Subject Descriptors: D.2.2 [**Software Engineering**]: Design Tools and Techniques; D.2.11 [**Software Engineering**]: Software Architectures

General Terms: Design, Languages, Standardization

Additional Key Words and Phrases: C2, formal modeling, Object Constraint Language, object-oriented design, Rapide, software architecture, Unified Modeling Language, Wright

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 [17,43]. 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 unnecessary details and focus on the "big picture"—system structure, high level communication protocols, assignment of software

components and connectors to hosts, development process, and so on [17,23,27,43,55,56]. The basic promise of software architecture research is that better software systems can result from modeling their important architectural aspects throughout, and especially early in, the development lifecycle. Choosing which aspects to model and how to evaluate them are two decisions that frame software architecture research [32].

To date, the software architecture research community has focused predominantly on analytic evaluation of architectural descriptions. Many researchers have come to believe that, to obtain the benefits of an explicit architectural focus, software architecture must be provided with its own body of specification languages and analysis techniques [12,16,29,62]. Such languages are needed to define and analyze properties of a system upstream in its development, thus minimizing the costs of detecting and removing errors. The languages 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) have been proposed [4,13,27,28,34,37,39, 52,59].

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 system 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 study of ADLs [36].

Another community, primarily from industry, has focused on modeling a wide range of issues that arise in software development, perhaps with a family of (often less formal) 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 some important details.

TABLE 1. Software Architecture Community Fragmentation

	RESEARCH COMMUNITY	PRACTITIONER COMMUNITY
MAJOR FOCUS	analytic evaluation of architectural models	wide range of development issues
ARTIFACTS	individual models	families of models to span and relate issues
RIGOR	formal modeling notations	practicality over rigor
PRIMARY GOAL	powerful analysis techniques	architecture as "big picture" in development
SCOPE	depth over breadth	breadth over depth
OUTCOME	special-purpose solutions	general-purpose solutions

These two predominant perspectives on software architecture are summarized in Table 1, which compares them along a number of important dimensions. We acknowledge that the positions of the

two communities are significantly more complex than represented in the table. However, we believe that the table provides a useful, if simplified, overview of the relationship between the two communities and motivates the need to bridge the chasm between them.

As in the case of the numerous ADLs produced by the research community, several competing notations have been used in the practitioner community. However, there now exists a concerted effort to standardize on notations and methods for software analysis and design. Standardization provides an economy of scale that results in more and better tools, better interoperability between tools, a larger number of available developers skilled in using the standard notation, and lower overall training costs. One hypothesis of this paper is that the benefits of standardization need not be achieved at the expense of losing the power afforded by specialized notations. Instead, when special-purpose notations are needed, they can often be based on, or related to, standard notations.

Specifically, in this paper we investigate the possibility of using the Unified Modeling Language (UML) [5], an emerging standard software design language, as a starting point for bringing architectural modeling into wider, industrial use. At first glance, UML appears to be well suited for this because it provides a large, useful, and extensible set of predefined constructs, is semi-formally defined, has the potential for substantial tool support, and is based on experience with mainstream development methods. The primary goal of this work is an assessment of UML's expressive power for *modeling* software architectures in the manner in which existing ADLs model architectures. To this end, we have conducted an extensive examination of UML's ability to model the architectural concepts provided by several ADLs. Our study forms a necessary foundation for further investigating the possibility of providing a broadly applicable extension of UML for architecture modeling.

Representing in UML the architectural building blocks supported by an ADL (e.g., Wright's components, connectors, ports, roles, and styles [4]) offers potential benefits both to practitioners who prefer the ADL as a design notation and to those who are more familiar with UML. For example, if a mapping were enabled from an architecture modeled in Wright to one in UML, a Wright user might be able to leverage a wide number of general-purpose UML tools for later stages of development, such as tools for code generation, simulation, analysis, reverse engineering, and so forth. Conversely, if UML were extended to include Wright's modeling capabilities, it would potentially enable a UML user to exploit the powerful analyses for which Wright is suited, such as interface compatibility checking and deadlock detection.

In order to evaluate UML's suitability for modeling software architectures in the manner outlined above, we have placed no restrictions on the manner in which UML is used for this purpose, other than the requirement that the resulting approach still involve *standard* UML. The motivation for this requirement is clear: altering UML *in any way* to better support the needs of software architectures invalidates the argument for using a standard notation in the first place. We have identified three possible strategies for using UML to model architectures. Since a preliminary evaluation indicated that one of the strategies results in a notation that is *not* legal UML, we have pursued only two strategies in depth.

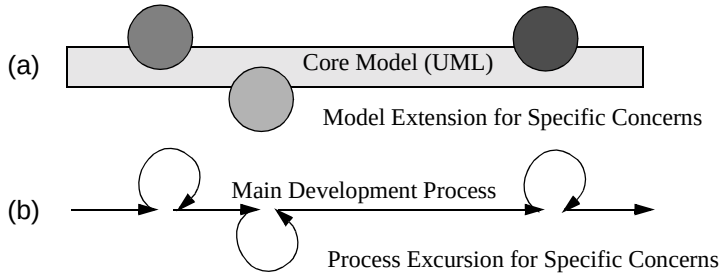


FIGURE 1. (a) A core model with extensions (b) Sketch of an associated process.

It is important to note that we envision the strategies discussed in this paper being used by practitioners in the context of their *existing* software processes and have thus tried to refrain from prescribing a particular process for relating ADLs and UML. But at least at a high level, our overall approach can be visualized in the context of a modeled software system as shown in Figure 1a: Standard, "core" UML is constrained (either implicitly in the way it is used, or explicitly via the mechanisms discussed below) to address specific architectural concerns identified by the architect. The conceptual view of the corresponding process is given in Figure 1b: occasional "excursions" from the main development process may be undertaken as needed to address the identified architectural concerns. These model extensions (Figure 1a) and process excursions (Figure 1b) may involve mapping between UML and ADLs that provide particular kinds of support, or they may involve using a particular UML extension and corresponding UML-compliant tools that have been developed to provide the necessary support.

We have defined a minimum set of requirements for objectively evaluating UML's ability to represent software architectures effectively. This set was derived from our extensive studies of ADLs [36] and software system development concerns that have significant architectural relevance [32,35]. While certainly not exhaustive, we have found these requirements to be sufficiently broad to highlight both the strengths and weaknesses of UML in this endeavor. The requirements are as follows:

- UML should be well suited to model the *structural* concerns (i.e., the configuration or topology [36] of a system). This requirement is also advocated in a study of modeling the structural aspects of architecture in UML by Garlan et al. [14].
- UML should be able to capture a variety of *stylistic* issues addressed both explicitly and implicitly by ADLs. These issues include a standard design vocabulary, recurring topologies, and, possibly, generic system behavior.
- UML should be able to model the different *behavioral* aspects of a system focused upon by different ADLs. While this may appear to be an unfair requirement, given the wide range of semantic models employed by existing ADLs (e.g., CSP [4], partially ordered event sets [27], π-calculus [28], first-order logic [34]), its primary goal is to highlight UML's limitations and suggest possible areas for improvement. Moreover, our study has shown UML to be surprisingly flexible in representing a wide range of semantic concerns.
- UML should be able to support modeling of a wide range of component *interaction* paradigms (whether specific to or independent of a particular style). This requirement stems from one of the key contributions of software architecture research—the focus on component interactions (i.e., software *connectors*) as first-class system modeling concerns [38,51].

- Finally, a requirement derived from the ones above is that UML should be able to capture any *constraints* arising from a system's structure, behavior, interactions, and style(s).

The remainder of the paper is organized as follows. Section 2 presents a brief overview of UML. Section 3 identifies and briefly evaluates the three possible strategies to modeling architectures in UML. Sections 4 and 5 then present in-depth evaluations of the two viable strategies based on modeling capabilities provided by three ADLs: C2 [30], Wright [3], and Rapide [27]. Section 6 discusses related work. Section 7 presents our conclusions, summarizing the strengths and weaknesses of the presented strategies and outlining plans for future research.

2 An Overview of UML

UML is a modeling language with a semi-formal syntax and semantics. It is defined within a general four-layer metamodeling architecture shown in Figure 2. The *meta-meta model* layer defines a language for specifying the meta model layer. The *meta model* layer, in turn, defines legal specifications in a given modeling language; for example, the UML meta model defines legal UML specifications. The *model* layer is used to define models of specific software systems. And the *user objects* layer is used to construct specific instances of a given model.

The model and meta model layers are most relevant for modeling software architectures in UML. They are summarized in the remainder of this section. The section also presents a brief overview of UML's associated constraint language, the Object Constraint Language (OCL). For more extensive details, the reader is referred to standard texts on UML and OCL [5,46,61] and to the draft specification developed by the Object Management Group (OMG) [42].

2.1 UML Design Models and Diagrams

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. UML models address a number of design issues through a variety of diagrams: (1) classes and their declared attributes, operations, and relationships;

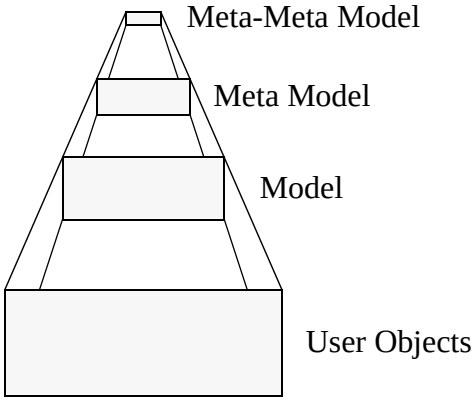


FIGURE 2. The four-layer metamodeling architecture of UML. The diagram qualitatively depicts the increase in the sizes of the modeling spaces at each level. For example, the single *meta-meta model* can be used to define a number of *meta models*, such as UML's, which can, in turn, be used to *model* countless *user objects*.

(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 closely the model will correspond 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.

Figure 3 presents an example of a UML model in which a UML class diagram is used 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* (i.e., they *inherit* from *Worker* as subclasses). Labor union contracts constrain *Companies* such that *Robots* may not make up more than 10% of the work force. This is stated in a constraint at the top of the class diagram; the details of this constraint will be explained shortly. A training *Course* contains many *Trainees*, and each *Trainee* may take from one to four *Courses*. In this example, *Trainee* is an interface (a set of exported operations) rather than a full class. An *Employee* is capable of performing all the operations of *Trainee*. In UML, *aggregation* (white diamond) is an association indicating that one object is temporarily subordinate to one or more others, whereas *composition* (black diamond), a stronger form of aggregation, is an association indicating that an object is subordinate to exactly one other object throughout its lifetime. The association between *Company* and *Course* involves no inheritance, aggregation or composition.

2.2 UML Extension Mechanisms and the Object Constraint Language (OCL)

Designers periodically may need to extend UML in well-defined ways in order to capture certain kinds of modeling concerns. UML provides a number of extension mechanisms that allow designers to customize and extend the semantics of model elements:

1. *Constraints* place added semantic restrictions on model elements. The range of possibilities for constraints are numerous and include type constraints on class attribute values, constraints on the construction of associations between classes, and so on.

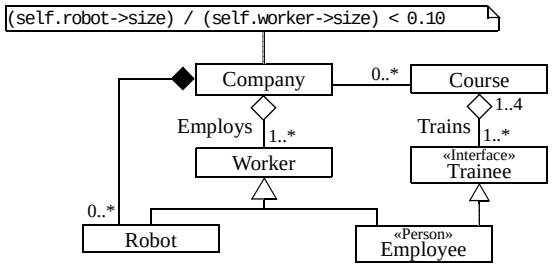


FIGURE 3. An example design expressed in UML.

2. *Tagged values* allow attributes to be associated with model elements. For instance, a project may wish to associate "version" and "author" tags or other such metadata with certain model elements.
3. *Stereotypes* allow groups of constraints and tagged values to be given descriptive names and applied (with the name specified in double angle brackets) to model elements, effectively creating a new yet restricted form of meta class for constructing models. The semantic effect is as if the constraints and tagged values were attached directly to those elements. For instance, interfaces are identified in class diagrams by attaching the stereotype name *«interface»* to class icons; among other things, the stereotype constrains an interface to declare only operations and no attributes.
4. *Profiles* are predefined sets of stereotypes, tagged values, constraints, and icons to support modeling in specific domains. The UML specification currently defines profiles for the Unified Process and for Business Modeling [42].

It is possible to express constraints on UML models using the Object Constraint Language (OCL), which combines first-order predicate logic with a diagram navigation language [42,61]. Each OCL expression is specified and evaluated in the context of (the instances of) some model element (referred to as *self*) and may use attributes and relationships of that element as terms. The *self* instance may be a UML classifier (such as a class or an interface), or an element used by a classifier (such as an attribute, an operation, or an end element of associations), or another type of model element. OCL also defines operations on sets, bags and sequences to support construction and manipulation of collections of model elements in OCL expressions. For instance, the operations defined within a class form a set that can be traversed in order to apply a constraint to each operation.

The top of Figure 3 illustrates a simple OCL constraint on the instances of class *Company* (the *self* model element) expressed in terms of the cardinalities of its associations with *Robot* and *Worker* classes. Each association is identified by the name of the role filled by the class at the other end of the association. By default, the role name is the name of the class itself with the first letter changed to lower case, and the role name evaluates to the set of all instances filling the role. Referring to associations and roles in this manner provides a means of navigating through the enclosing diagram and is therefore a key technique for constructing constraints in OCL. The predefined property *size* is used to obtain cardinalities of collections of elements (in this case the number of elements filling the *robot* role and the number of elements filling the *worker* role). Thus, the constraint says that the number of instances of *Robots* aggregated to an instance of *Company* divided by the number of instances of *Workers* aggregated to the instance of *Company* must be less than one-tenth.

We further describe and illustrate OCL with constraints on the UML meta model in the next section and in Section 5.

2.3 The UML Meta Model

As mentioned above, UML is a graphical language with semi-formal syntax and semantics, which are specified via a meta model, informal descriptive text, and constraints [42]. The meta model is itself a UML model that specifies the abstract syntax of UML models. 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. Thus, in terms of Figure 2, the meta class *Class* is defined at the *Meta*

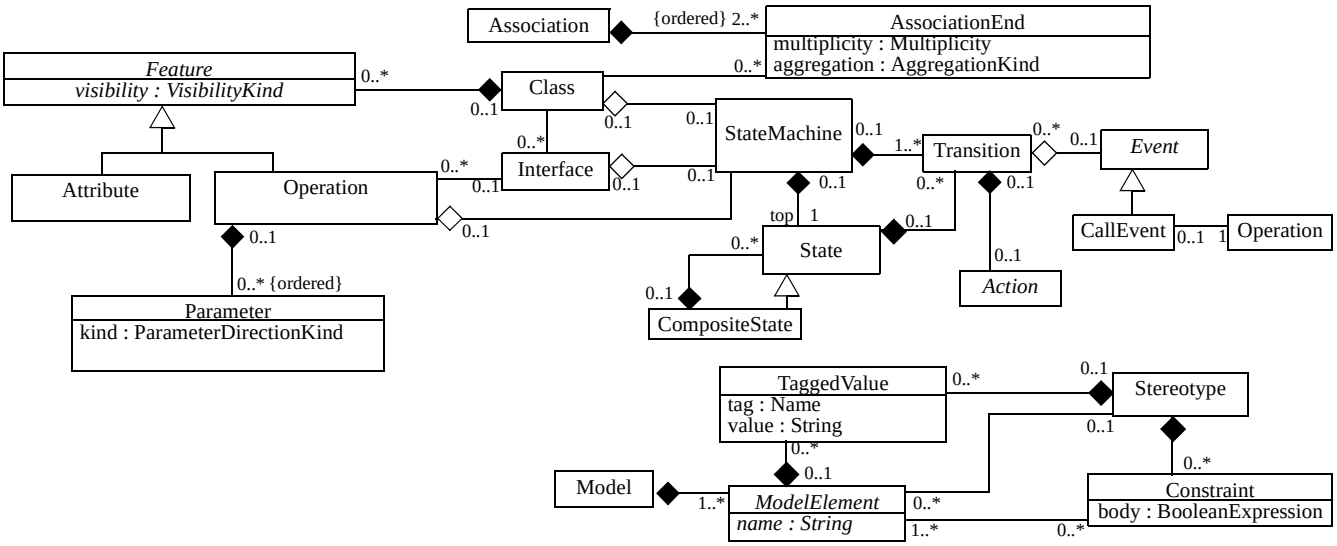


FIGURE 4. Simplified UML meta model (adapted from [42]). Italicized classes are abstract (i.e., non-instantiable) classes. All classes are subclasses of ModelElement (except ModelElement itself); this relationship is not shown.

Model level, and instances of *Class* are the classes defined in software system models at the *Model* level. Figure 4 depicts the parts of the UML meta model used in this paper.

A powerful application of the extension mechanisms described in Section 2.2 is to constrain the way the meta model is used in constructing system models. In particular, a stereotype can be defined for use with a particular meta model element and then applied to instances of that element in the *Model* level (thereby constraining all instances of the stereotyped element at the *User Objects* level of Figure 2). A stereotype thus essentially creates a new modeling construct, but one whose use still results in legal UML models. For example, suppose we wish to enhance the class diagram of Figure 3 to impose a 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 this example, composition would mean that employees could not participate in any other aggregates and never work for another company. 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, not the composed.

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

Note that the stereotype is defined for use with classes (i.e., instances of the meta model element *Class*) in system models, and thus we could apply this stereotype to class *Worker* in Figure 3; to do so, the stereotype name would be specified in double angle brackets (i.e., as <<*Person*>>) above the name *Worker* in *Worker*'s class icon. Associating the stereotype with a meta model element in this way allows the stereotype to be defined in terms of attributes, roles, and other elements at the Meta

Model level. The first line of the OCL constraint defined in the stereotype is a universal quantifier over all association ends of the stereotyped class. In particular, *self* is an instance of the meta model element *Class*; *Class* has associations with instances of the meta model element *AssociationEnd*, which by default fills a role called *associationEnd* in each such association (see Figure 4). For each such association end *myEnd*, the second line is a universal quantifier over all the association ends of the association to which *myEnd* is attached (and thus *myEnd* is included in the quantification). Again, note that *association* and *associationEnd* in this line refer to roles defined for associations in the meta model. For each such association end *anyEnd*, the third line checks to see if the *aggregation* attribute of *anyEnd* is *composite*, indicating that *anyEnd* is a composite of the association. If there is a composite association end, then the fourth line states the requirement that *myEnd* also must be a composite of the association. Because UML already constrains associations to have at most one composite end, this in effect constrains *myEnd* to be the only composite in the association.

The labor union constraint presented in Figure 3 and described in Section 2.2 uses terms from the model to constrain the state of the system at run-time. In contrast, the stereotype Person uses terms from the UML meta model to constrain the model of the system. In addition, although not depicted in Figure 4, models themselves are defined in the meta model through the meta class *Model*. This makes it is possible to apply constraints to whole diagrams, which for example allows one to constrain all the elements of a diagram to uniformly use a particular set of stereotypes. As described in the next section, we use these techniques of constraining the UML meta model in our second strategy for supporting architectural modeling in UML.

3 Modeling Software Architectures in UML

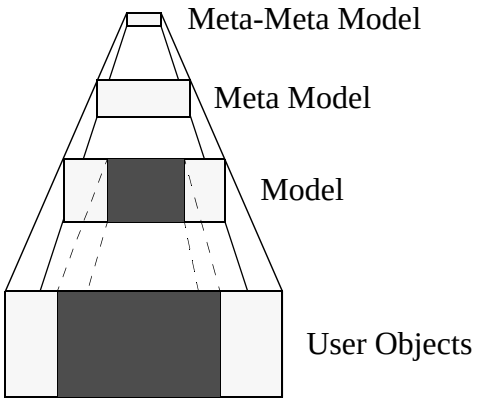
The four-layer metamodeling architecture of UML suggests three possible strategies for modeling software architectures using UML:

1. use UML “as is;”
2. constrain the UML meta model using UML’s built-in extension mechanisms; and
3. extend the UML meta model to directly support the needed architectural concepts.

Each approach has certain potential advantages and disadvantages. This section presents a brief discussion and preliminary evaluation of the approaches. Recall from the introduction that, in order to reap the benefits of standardization (e.g., understandability and manipulability by standard tools), we *require* that any resulting notation adhere to the syntax and semantics of UML.

3.1 Strategy 1: Using UML “As Is”

The simplest strategy is to use the existing UML notation to represent software architectures. Assessing the practicality of this approach requires an evaluation of the suitability of UML’s modeling features for representing specific architectural concepts. A major advantage of the approach is that it would result in architectural models that are immediately understandable by any UML user and manipulable by UML-compliant tools. However, the approach would provide no means for explicitly



[1]

FIGURE 5. The UML model is explicitly constrained to support software architecture modeling needs.

representing the relationship between existing UML constructs and architectural concepts for which there is no direct UML counterpart (such as software connectors and architectural style rules). Rather, this relationship would have to be maintained implicitly by the software architect.

3.2 Strategy 2: Constraining UML

The space of software development situations and concerns for which UML is intended exceeds that of ADLs (e.g., as reflected in UML's support for requirements analysis and specification, and low-level design). Therefore, one possible approach to modeling architectures in UML is to constrain UML. UML is an extensible language in that new constructs may be added to address new concerns in software development. It provides a means for incorporating new modeling capabilities and addressing new development concerns without changing the existing syntax or semantics of UML. This is accomplished via the extension mechanisms described in Section 2.2. Conceptually, this approach can be represented using UML's metamodeling architecture from Figure 2. As depicted in Figure 5, only a relevant portion of the UML modeling space is made available to the software architect.

The major advantage of this approach is that it explicitly represents and enforces architectural constraints. Furthermore, an architecture specified in this manner would still be manipulable by standard UML tools and would be understandable to UML users (with some added effort in studying the OCL constraints). A disadvantage of the approach is that it may be difficult to fully and correctly specify the boundaries of the modeling space in Figure 5. Additionally, as a practical concern, tools that enforce OCL constraints in UML specifications are only beginning to emerge [57].

3.3 Strategy 3: Augmenting UML

One obvious, and therefore tempting, approach to adapting UML to support the needs of software architectures is to augment UML's meta model, as shown in Figure 6. Augmenting the meta model helps to formally incorporate new modeling capabilities into UML. The potential benefit of such an extension is that it could fully capture every desired feature of every ADL and provide "native" support for software architectures in UML. However, the challenge of standardization is find-

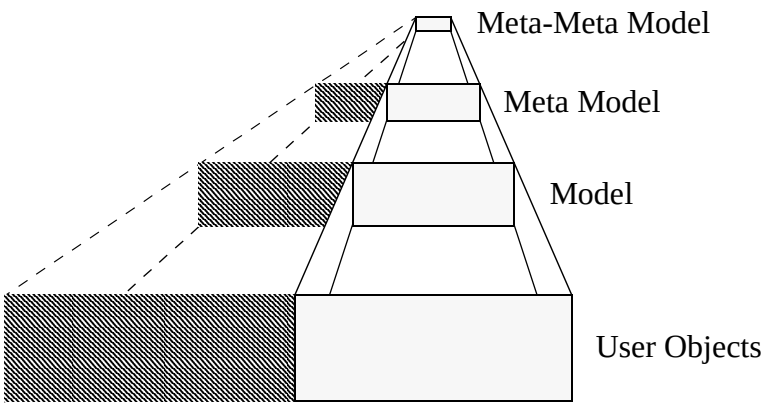


FIGURE 6. The UML meta model is extended to support software architecture modeling needs.

ing a language that is general enough to capture needed concepts without adding too much complexity, while such a modification would result in a notation that is overly complex. More importantly, the notation would not conform to the UML standard and could become incompatible with UML-compliant tools.

Given that it violates the key requirement that the resulting notation adhere to the syntax and semantics of UML, we do not pursue the third strategy further. We discuss the first two strategies, outlined in Sections 3.1 and 3.2, in more detail below.

4 Strategy 1: UML as an Architecture Description Language

At first blush, it appears that the rich set of notations and features provided by UML make it suitable “as is” for modeling software architectures. Indeed, many of the proponents of UML believe that its support for modeling the architecture of a system is entirely adequate. This viewpoint is perhaps best represented by the Unified Software Process, a process developed by the creators of UML for “architecture-centric” development of systems using UML [22]. However, we note that there is still widespread disagreement as to what a software architecture is, and hence we expect there to be even greater disagreement as to how to model an architecture in UML.

We evaluate the presumption of UML’s adequacy by using UML to model applications in the same manner as they would be modeled using an ADL. This strategy allows us to assess the support provided by UML for the needs of architectural modeling and to compare directly the modeling power provided by UML to that of an ADL.

To illustrate this strategy, we model an application in the C2 architectural style and its accompanying ADL [56]. While neither the chosen application nor the style are universally applicable, they are sufficient to highlight the important similarities and differences between UML and ADLs. This example is representative in that a number of issues we encountered are independent of C2 or the application’s characteristics, in particular representing architectural structure and individual elements (components and connectors) in UML, modeling component and connector interfaces in UML, identi-

fying different roles that the elements of a UML domain model play in the architecture, and the process of transforming a UML domain model into an architectural model.

4.1 Example Application

The selected example application is a simplified version of the meeting scheduler problem, initially described by van Lamsweerde and colleagues [10] and recently considered as a candidate model problem in software architectures [54]. In this application, 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 meeting initiator also asks active participants to provide any special equipment requirements on the meeting location (e.g., projector, workstation, network connection, telephones). The meeting 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 meeting initiator extends the date range;
- some participants expand their preference set or narrow down their exclusion set; or
- some participants withdraw from the meeting.

4.2 Overview of C2

Before proceeding with the architectural design of the application, we provide a high level overview of the C2 architectural style [56], needed to understand this example. Section 5 contains a more detailed discussion of the style's rules. C2 and its accompanying ADL [30,34,37] are used for highly distributed software systems. In a C2-style architecture, software *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. A component interface may be attached to at most one connector. A connector may be attached to any number of other components and connectors. 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. Request messages may only be sent "upward" through the architecture, and notification messages may only be sent "downward."

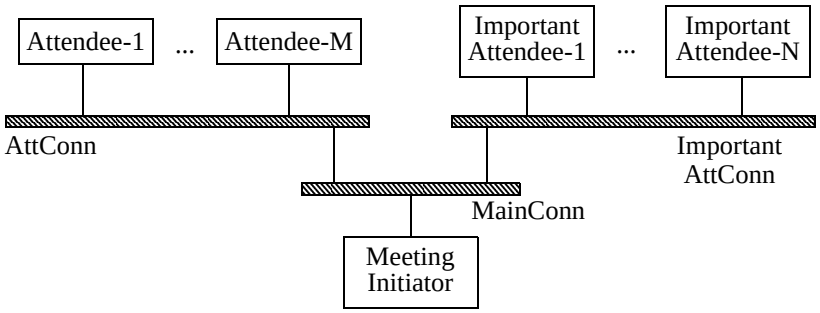


FIGURE 7. A C2-style architecture for a meeting scheduler system.

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 its operations, 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 GUI toolkits). 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.

4.3 Modeling the Meeting Scheduler in C2

This section presents a partial model of the meeting scheduler application in C2 and its ADL.[1] The purpose of this model is to introduce the reader to the nuances of architectural decomposition according to the rules of C2, as well as to serve as a basis of evaluating the corresponding UML model, given in Section 4.4. Figure 7 shows a graphical depiction of a C2-style architecture for the meeting scheduler system. The 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 *MeetingInitiator* 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 messages relevant to their respective attached components.

The *MeetingInitiator* component initiates the computation by sending requests for meeting information to *Attendees* and *ImportantAttendees*. The two sets of components notify the *MeetingInitiator* component, which attempts to schedule a meeting and either requests that each potential

1. A complete model of the application is given in [33].

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 *MeetingInitiator* 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 7. For this reason, we specify the architecture in C2's textual ADL [30,37]. 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. The requests it sends to initiate the computation in the system are specified in the *startup* segment of its *behavior*.[2]

```
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);
    behavior
        startup always_generate GetPrefSet, GetExclSet, GetEquipReqts, GetLocPrefs;
        received_messages PrefSet may_generate RemoveExclSet xor RequestWithdrawal xor MarkMtg;
        received_messages ExclSet may_generate AddPrefDates xor RemoveExclSet xor RequestWithdrawal xor MarkMtg;
        received_messages EquipReqts may_generate AddPrefDates xor RemoveExclSet xor RequestWithdrawal xor MarkMtg;
        received_messages LocPref always_generate null;
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
        bottom_domain is
            out
                PrefSet (p : date_rng);
                ExclSet (e : date_rng);
                EquipReqts (eq : equip_type);
            in
                GetPrefSet ();
                GetExclSet ();
                GetEquipReqts ();
```

2. *Startup* and *cleanup* are optional parts of a component's specification that indicate any special processing needed after the component is instantiated and before it is removed from a system, respectively (see [33]). In an OO language, *startup* functionality is typically provided as part of an object's *constructor*, while proper *cleanup* is ensured by the *destructor.*

```
            RemoveExclSet ();
            RequestWithdrawal ();
            AddPrefDates ();
            MarkMtg (d : date; l : loc_type);
    behavior
        received_messages GetPrefSet always_generate PrefSet;
        received_messages AddPrefDates always_generate PrefSet;
        received_messages GetExclSet always_generate ExclSet;
        received_messages GetEquipReqts always_generate EquipReqts;
        received_messages RemoveExclSet always_generate ExclSet;
        received_messages RequestWithdrawal always_generate null;
        received_messages MarkMtg always_generate null;
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 (though it 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 ();
    behavior
        received_messages GetLocPrefs always_generate LocPrefs;
end ImportantAttendee;
```

The *MeetingScheduler* architecture depicted in Figure 7 is shown below. The architecture is specified with the conceptual components (i.e., component types) defined above. Each conceptual component (e.g., *Attendee*) can be instantiated multiple times in a *system*.

```
architecture MeetingScheduler is
    conceptual_components
        Attendee;ImportantAttendee; MeetingInitiator;
    connectors
        connector MainConn is message_filter no_filtering;
        connector AttConn is message_filter no_filtering;
        connector ImportantAttConn is message_filter no_filtering;
    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;
```

4.4 Modeling the C2-Style Meeting Scheduler in UML

UML provides constructs for modeling software components, their interfaces, and their deployment on hosts.[3] However, these built-in constructs are not suitable for describing architecture-level

components because they assume both too much and too little. Components in UML are assumed to be concrete, executable artifacts that consume machine resources such as memory. In contrast, architectural components are conceptual artifacts that decompose the system's state and behavior. Although instances of architectural components in a given system may be implemented by concrete UML component instances, the architectural components are not themselves concrete. Furthermore, components in UML may have any number of interfaces and any internal structure, whereas architectural components must satisfy any rules or constraints imposed on them (e.g., by an architectural style such as C2). For these reasons, we have chosen instead to use UML classes to model architectural components.

The key to this strategy for relating UML and an ADL is ensuring that the design of an application in UML be driven and constrained both by the modeling features available in UML and the constraints imposed by the ADL (and possibly its underlying architectural style rules). The two must be considered simultaneously. For this reason, the initial steps in this process are to develop (1) a domain model for the application expressed in UML and (2) an informal architectural diagram, such as the C2 diagram from Figure 7. The architectural diagram is key to making the appropriate mappings between classes in the domain model and components in the architectural diagram. This step is similar to relating domain models and reference architectures in the domain-specific software architecture (DSSA) process [58]. One effect of the mapping is that it directly points to the need to explicitly model architectural constructs that commonly are not found in UML designs, such as the connectors and component message interfaces found in a C2-style architecture.

Our initial attempt at a UML class diagram for the meeting scheduler application is shown in Figure 8. The diagram depicts the domain model for the meeting scheduler application consisting of the domain classes, their inheritance relationships, and their associations. Apart from limiting *MeetingInitiator* to a single instance and specifying possible cardinalities of the other components, 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 association *Invites* 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.

Message interfaces are prominent elements of C2-style components (recall Section 4.3). This is reflected in a UML design by modeling interfaces (i.e., class icons stereotyped with <<*interface*>>) explicitly and independently of the classes that will implement those interfaces. Each class corresponding to a component exports one or more of the interfaces shown in Figure 9. The interfaces *ImportantMtgInit* and *ImportantMtgAttend* inherit from the interfaces *MtgInit* and *MtgAttend,* respectively. The only difference is the added operation to request and notify of location preferences. Note

3. Unless otherwise noted, "component" in this discussion refers to a component *type,* as opposed to a specific *instance* of that type. This is the case with both architectural components (modeled in an ADL) and UML components.

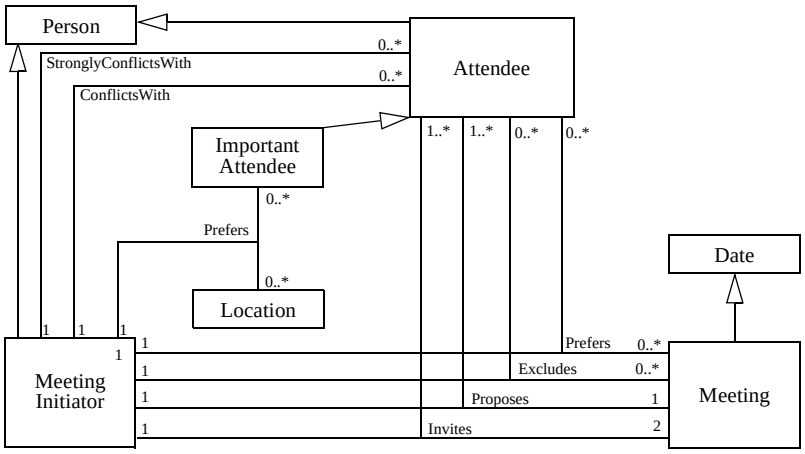


FIGURE 8. UML class diagram for the meeting scheduler application. Details (attributes and operations) of each individual class have been elided for clarity.

that every method signature (i.e., UML operation) in Figure 9 corresponds to a C2 message in the architecture specified in Section 4.3. All operations in the UML model will be implemented as asynchronous message passes, as they would in C2. For this reason, the method signatures in Figure 9 lack return types.

In order to model a C2 architecture in UML, connectors must be defined. Although connectors fulfill a role different from components, they can be modeled also with UML classes. However, a C2 connector is by definition generic and can accommodate connections to any number and 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; instead, the connectors specified in UML must be application-specific and must have fixed interfaces. To reflect the generic nature of C2 connectors, the connector classes for the meeting scheduler application realize the same interfaces as the components they connect. Each connector can be thought of as a simple class that (possibly filters and) forwards the messages it receives to the appropriate components. Therefore, while the component class interface specifications, shown in Figure 9, 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 10. Connectors do not add any functionality at the domain model level; they are thus absent from the class diagram in Figure 8.

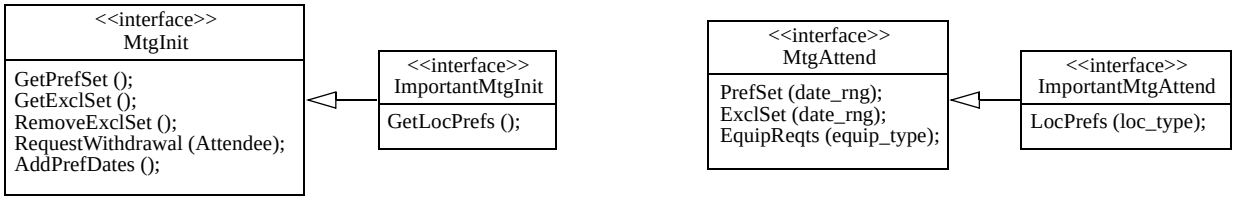


FIGURE 9. Meeting scheduler class interfaces.

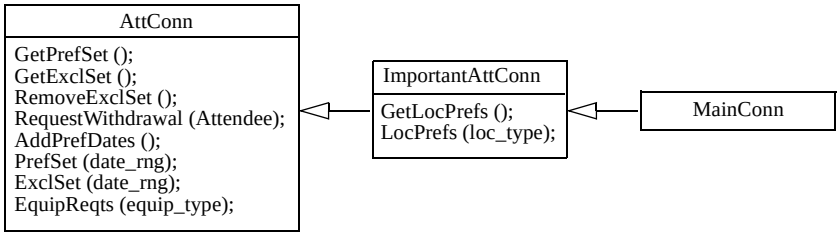


FIGURE 10. Application-specific UML classes representing C2 connectors.

A refined class diagram for the meeting scheduler application is shown in Figure 11, which depicts primarily the interface relationships between the classes. In particular, each solid arc from a class to a circle labeled with an interface name is a “lollipop” depicting the realization of the interface by the class, while each dashed arrow from a class to a lollipop depicts a dependency the class has on the interface. The classes *Attendee* and *ImportantAttendee* are related by interface inheritance, which is depicted in Figure 9, but is only implicit in Figure 11. We have omitted from Figure 11 the classes *Location*, *Meeting*, and *Date* shown in Figure 8, since they represent the data exchanged by the components in the system and have not been impacted. We have also omitted the two superclasses for the components and connectors (*Person* and *Conn*, respectively).

The class diagram in Figure 11 has been deliberately structured to highlight its similarity with the C2 architecture depicted in Figure 7. One difference is that the diagram in Figure 7 depicts instances of the different components and connectors, while a UML class diagram depicts classes (i.e., types) and their associations (with multiplicities used to convey information about the number of possible instances); in other words, the class diagram represents the *possible* relationships among instances of the depicted classes. Furthermore, being a class diagram, it does not formally capture the

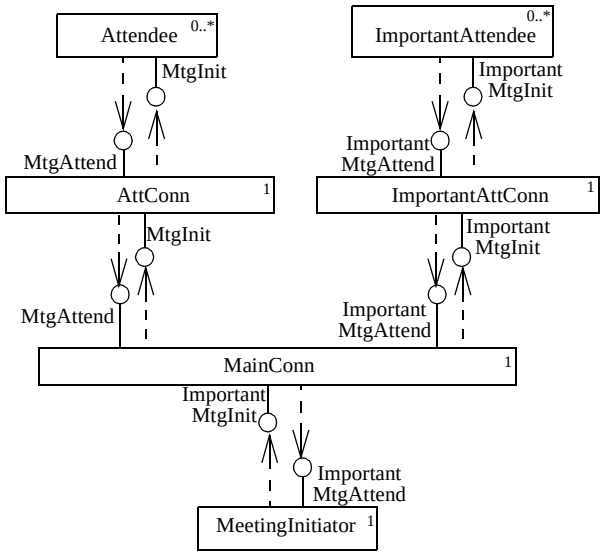


FIGURE 11. UML class diagram for the meeting scheduler application designed in the C2 architectural style.

topological constraints implied by its layout. To both depict class instances and more accurately convey topological intent, we use a collaboration diagram.

Figure 12 depicts a collaboration between an instance of the *MeetingInitiator* class (*MI*) and instances of *Attendee* and *ImportantAttendee* classes (with the collaboration represented by the numbered sequence of operation invocations). In particular, *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 9), *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 11, differ from a C2 architecture in that they explicitly specify only the messages a component receives (via interface attachments to the class icon for a component). On the other hand, a model of a C2-style architecture also specifies the messages sent by components, as well as structural and behavioral aspects of the architecture. The issue of architectural structure and behavior is further discussed below.

4.5 Discussion

We base our assessment of UML's suitability for modeling software architectures using this first strategy on the evaluation requirements introduced in Section 1. The exercise described above demonstrated that, to a large extent, we can successfully model a C2-style architecture in UML. Part of the success can be attributed to the fact that, as anticipated, many architectural concepts are found in UML (e.g., interfaces, components, component associations, and so forth). The same basic strategy can be used to model the *structure* of architectures that adhere to other styles and/or are modeled with other ADLs, e.g., ACME [15], Darwin [28], or UniCon [52].

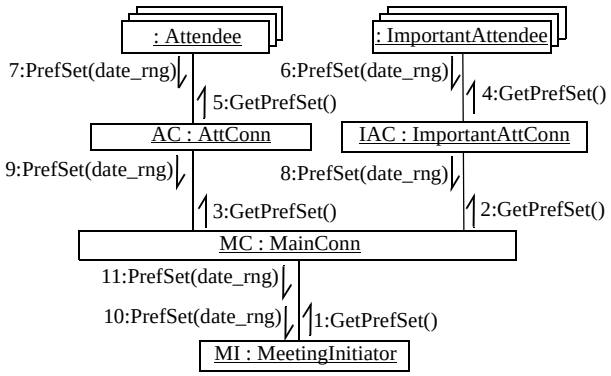


FIGURE 12. Collaboration diagram for the meeting scheduler application showing a response to a request issued by the *MeetingInitiator* to both *Attendees* and *ImportantAttendees*.

It must be noted, however, that the modeling capabilities provided by UML "as is" do not fully satisfy the structural needs of architectural description for two key reasons. First, UML does not provide specialized constructs for modeling architectural artifacts. For example, although they are different architectural entities with very different responsibilities, connectors and components must be modeled in UML using the same mechanism. Second, the rules of a given architectural *style* are directly reflected in its corresponding ADL and maintained by the accompanying toolset, whereas those rules must be applied mentally by the software architect who chooses to use UML. "Emulating" particular structural *constraints* in UML, as was done in the example in Section 4.4, is an error-prone approach. Furthermore, additional documentation must accompany such a UML model to ensure that no future modifications violate the desired constraints.

Note that the purpose of this work is a general assessment of the ability of UML "as is" to model software architectures. In order to more thoroughly evaluate UML in this regard and point out all of its strengths and shortcomings, one could extend the approach discussed above by representing the major structural ADL features using additional UML diagrams (e.g., package diagrams), as in [14]. While the specific details of such an evaluation are likely to vary from one chosen representation to another, the two major shortcomings of UML discussed above will remain.

In addition to structural aspects of an architecture, a number of ADLs (e.g., Rapide [27] and Wright [3,4]) also provide constructs for modeling the dynamic component *behavior* and *interactions* in the architecture. UML's features, such as sequence, collaboration, and statechart diagrams, can be used effectively to this end (see Section 5). As with the structural constructs, however, it may be difficult to ensure that the intended behaviors or interactions, as they would be specified in an ADL (e.g., in Wright's communicating sequential processes, or CSP [20]), are correctly modeled in UML (e.g., using statecharts). These potential difficulties motivate our exploration of the second strategy introduced in Section 3.2.

5 Strategy 2: Constraining UML to Model Software Architectures

The second strategy for modeling architectures in UML involves using OCL to specify additional constraints on existing meta classes of UML's meta model. In principle, this allows the use of existing UML-compliant tools to represent and analyze the desired architectural models, and ensure architectural constraints. This strategy involves

- selecting one or more existing meta classes from the UML meta model in which to situate a given ADL modeling construct or capability, and
- defining a stereotype that can be applied to instances of those meta classes in order to constrain their semantics to that of the associated ADL feature.

This strategy treats UML as a core notation that is extended in order to support specific architectural concerns. Note that this notion of extension is different from the one discussed in Section 3.3 and depicted in Figure 6: UML is *conceptually* extended to provide architects with additional modeling tools that originally did not exist in UML; however, the UML meta model remains intact and the OCL

facilities are actually used to *constrain* the notation to a specific UML-compliant subset. As new concerns arise in development, new extensions may be added to support those concerns. The semantics of the core notation is always enforced by UML-compliant tools. The semantics of each extension is enforced by the constraints of that extension. Dependencies and conflicts may arise between different extensions and must be handled by developers just as they manage other development dependencies and conflicts. 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.

We demonstrate this approach by providing examples of UML extensions for three ADLs: C2, Wright, and Rapide. We selected these languages because our extensive study of ADLs [36] indicates that they constitute a broadly representative set of capabilities found in current ADLs:

- C2 provides guidance for structural decomposition and event-based interaction according to a particular but fairly general architectural style;
- Wright enables behavioral and interaction modeling of individual architectural elements; and
- Rapide supports specification of local and global behavioral constraints.

The extensions based on these ADLs allow a broad assessment of UML's suitability for architecture modeling. Furthermore, they provide several insights that could inform the design of a UML profile for architectural modeling. Each of the three extensions is discussed in more detail below and evaluated with respect to the requirements established in Section 1.

5.1 Extensions Based on C2

The basic elements of the C2 style and its accompanying ADL were discussed in Section 4. The ADL is tightly tied to the C2 style; its syntax and semantics directly derive from the style. In this section we further elaborate on the C2 ADL's elements and model their semantics in UML via stereotypes.[4] The key elements of a C2 architectural description are *components*, *connectors*, and their *architectures*. Components and connectors interact by exchanging *messages* (also referred to as *events*); a message received by a component typically results in one or more outgoing messages. The style constraints that determine legal architectural topologies were informally discussed in Section 4.2 and will be formally specified below. Note that the ADL also allows simple causal relationships to be specified between incoming and outgoing messages in a component. However, we have chosen not to model this aspect of the ADL; instead, we point the reader to Section 5.3, where a much more expressive mechanism for modeling event causality is discussed and represented in UML.

4. In this section we present a representative sample of the stereotypes we have defined for C2. For a full specification, see Robbins et al. [44].

5.1.1 C2 Messages in UML

The UML meta class *Operation* matches the C2 concept of a message specification. A UML operation consists of a name, a parameter list and an optional return value. Operations may be public, private, or protected. To model C2 message specifications, we add a tag to differentiate notifications from requests and to constrain *Operation* to have no return values. C2 messages are all public, but that property is built into the UML meta class Interface, used in the definition of stereotype C2Interface below.

Stereotype C2Operation for instances of meta class Operation
--1-- C2Operations are tagged as either notifications or requests.
```
c2MsgType : enum { notification, request }
```
--2-- C2Operations are tagged as either incoming or outgoing.
```
c2MsgDir : enum { in, out }
```
--3-- C2 messages do not have return values.
```
self.parameter->forAll(p | p.kind <> return)
```

This stereotype is intended for application to operations (which are defined within classes and interfaces). The stereotype contains both tagged values (*c2MsgType* and *c2MsgDir*) and a universally quantified constraint on the parameters of the stereotyped operation (in particular, on the attribute *kind* defined for meta class *Parameter* in the meta model, as shown in Figure 4).

5.1.2 C2 Components in UML

The UML meta class *Class* is closest to C2's notion of component.[5] 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 that a class 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.

Stereotype C2Interface for instances of meta class Interface
--1-- A C2Interface has a tagged value identifying its position.
```
c2pos : enum { top, bottom }
```
--2-- All C2Interface operations must have stereotype C2Operation.
```
self.operation->forAll(o | o.stereotype = C2Operation)
```

5. Other researchers have explored using different UML constructs to model components. For example, Soni et al. use UML Packages to model composite components [21], while Garlan et al. explore the possibility of modeling components using several additional UML elements [14] (see Section 6). We believe that no single selection of UML constructs will be sufficient to fulfill everyone's architecture needs. Furthermore, multiple options may be pursued even in a single project. Although it may be possible to model C2 components with, say, UML packages instead of classes, such an exercise is out of the scope of this paper.

Stereotype C2Component for instances of meta class Class

--1-- C2Components must implement exactly two interfaces, which must be C2Interfaces,
-- one top, and the other bottom.

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

--2-- Requests travel “upward” only, i.e., they are sent through top interfaces and received
-- through bottom interfaces.

```
let topInt = self.interface->select(i | i.c2pos = top) in
let botInt = self.interface->select(i | i.c2pos = bottom) in
topInt.operation->forAll(o |
   (o.c2MsgType = request) implies (o.c2MsgDir = out)) and
botInt.operation->forAll(o |
   (o.c2MsgType = request) implies (o.c2MsgDir = in))
```

--3-- Notifications travel “downward” only. Similar to the constraint above.

--4-- Each C2Component has at least one instance in the running system.

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

The constraints in these stereotypes use many OCL features illustrated earlier in the paper, including quantification over meta model elements and cardinalities of collections. The second constraint of C2Component defines additional attributes (*topInt* and *botInt*) that are used to aid the definition of the constraint. The property *allInstances* returns all the instances of the associated model element (the *self* in the case of constraint 4 in stereotype C2Component) in existence at the time the expression is evaluated. The operation *select* selects a subset of an associated set for which the specified expression is true.

5.1.3 C2 Connectors in UML

C2 connectors share many of the constraints of C2 components. However, 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.associationEnd->size = 2
```

--2-- One end of the attachment must be a single C2Component.

```
let ends = self.associationEnd in
ends[1].multiplicity.min = 1 and ends[1].multiplicity.max = 1 and
ends[1].class.stereotype = C2Component
```

--3-- The other end of the attachment must be a single C2Connector.

```
let ends = self.associationEnd in
ends[2].multiplicity.min = 1 and ends[2].multiplicity.max = 1 and
ends[2].class.stereotype = C2Connector
```

Stereotype C2AttachUnderComp for instances of meta class Association. Same as C2AttachOverComp, but with the order reversed.

Stereotype C2AttachConnConn for instances of meta class Association

--1-- C2 attachments are binary associations.

```
self.associationEnd->size = 2
```

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

```
self.associationEnd->forAll(ae |
   ae.multiplicity.min = 1 and ae.multiplicity.max = 1 and
   ae.class.stereotype = C2Connector)
```

--3-- The two ends are not the same C2Connector.

```
self.associationEnd[1].class <> self.associationEnd[2].class
```

Stereotype C2Connector for instances of meta class Class

--1 through 3-- Same as constraints 1-3 on C2Component.

--4-- Each C2 connector has exactly one instance in the running system.

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

--5-- The top interface of a connector is determined by the components and connectors attached
-- to its bottom.

```
let topInt = self.interface->select(i | i.c2pos = top) in
let downAttach = self.associationEnd.association->select(a |
      a.associationEnd[2] = self) in
let topsIntsBelow = downAttach.associationEnd[1].interface->select(i |
      i.c2pos = top) in
   topsIntsBelow.operation->asSet = topInt.operation->asSet
```

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

The above stereotypes use the attribute *multiplicity* of association ends. Note that because the meta-level association between an *Association* and an *AssociationEnd* is ordered, the *associationEnd* role evaluates to a sequence (which is indexable) rather than to a set. While UML places no semantic significance on this ordering, and while modelers usually do not concern themselves with the underlying order of an association, we nevertheless found it necessary to exploit this ordering to encode topological information about the architecture. As will be seen later, this requires the architect to use a particular diagrammatic convention allowed by UML to ensure that the required order is maintained. This may seem somewhat inelegant, but the only alternative we could conceive is to encode such topological information in additional tagged values in the relevant stereotypes. We have opted against this second alternative (adding tagged values) because it would complicate the model, while, at the same time, still requiring the architect to explicitly select appropriate values for the tags.

Note also that it is possible to specify constraints in terms of the stereotypes associated with a model element. This is done above in the *C2Attach* stereotypes, as well as below in the

C2Architecture stereotype, to ensure that the C2 stereotypes are used consistently and completely when defining the topology of a C2 architecture.

5.1.4 C2 Architectures in UML

We now turn our attention to the overall composition of components and connectors in the architecture of a system. Recall from Section 4.2 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 (constraints 7 and 8).

Stereotype C2Architecture for instances of meta class Model

--1-- The classes in a C2Architecture must all be C2 model elements.

```
self.modelElement->select(me | me.oclIsKindOf(Class))->forAll(c |
   c.stereotype = C2Component or
   c.stereotype = C2Connector)
```

--2-- The associations in a C2Architecture must all be C2 model elements.

```
self.modelElement->select(me | me.oclIsKindOf(Association))->forAll(a |
   a.stereotype = C2AttachOverComp or
   a.stereotype = C2AttachUnderComp or
   a.stereotype = C2AttachConnConn)
```

--3-- Each C2Component has at most one C2AttachOverComp.

```
let comps = self.modelElement->select(me |
      me.stereotype = C2Component) in
comps->forAll(c |
      c.associationEnd.association->select(a |
            a.stereotype = C2AttachOverComp)->size <= 1)
```

--4-- Each C2Component has at most one C2AttachUnderComp. Similar to the constraint above.

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

```
let conns = self.modelElement->select(me |
      me.stereotype = C2Connector) in
conns.associationEnd.association->forAll(a |
   a.stereotype = C2AttachOverComp or
   a.stereotype = C2AttachUnderComp or
   a.stereotype = C2AttachConnConn)
```

--6-- C2Components do not participate in any non-C2 associations. Similar to the constraint
-- above, but without the third disjunct.

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

```
let conns = self.modelElement->select(e |
      e.stereotype = C2Connector) in
conns->forAll(c |
      c.associationEnd->size > 0)
```

--8-- Each C2Component must be attached to some connector. Similar to the constraint above.

The operation *oclIsKindOf* used in the above stereotypes is a predicate on the meta model class of the associated model instance. It evaluates to true if the instance belongs to the specified class or one of its subclasses. This operation is used in situations where a class of interest in the meta model is a subclass of some superclass that is directly accessible within the enclosing expression. This is the situation with *Class* and *Association*, which are two of the many subclasses of *ModelElement* (recall

Figure 4); in turn, *ModelElement* is directly associated with the class *Model* to which the stereotype applies.

5.1.5 Discussion of C2 Extensions

Constraining UML to enforce the rules of the C2 style has been fairly straightforward, because many (mostly structural) 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 UML (as we have used it in this section) nor C2 constrain the choice of inter-process communication mechanisms, nor do they assume that any two components run in the same thread of control or on the same host. Both UML and C2 support interactions via message passing. However, it should be noted that UML only allows specification of the messages received (corresponding to the operations provided) by a class, but not messages sent by the class; *call actions* can be used in a state diagram associated with the class to invoke required operations in another class, but the call actions are not explicitly declared as elements of the invoking class. We see this as a major shortcoming of UML, and we were forced to build the distinction between provided and required operations into our model indirectly by using the tagged values *c2MsgType* and *c2MsgDir*, which for required operations merely document the intent that an operation be required. Finally, although we did not model details of the internal parts of a C2 component [56] or the *behavior* of any C2 constructs, such aspects can be modeled in UML, as demonstrated in the following sections where we capture the internal behavioral aspects of model elements expressed in Wright and Rapide.

It is important to note that, while a majority of the concepts discussed above are implicit in the C2 ADL (recall the example in Section 4), each concept had to be carefully, explicitly specified in UML. The reason, of course, is that C2 ADL's sole purpose is to model C2-style architectures and most of its semantics is directly derived from the style, while UML has a much broader intended scope. This is also the major difference between our first strategy to modeling software architectures in UML, discussed in Section 4, and this strategy: unlike the first strategy, where the different architectural concepts (e.g., components, connectors, messages) were *implicit* in the UML design, this approach *explicitly* defines and constrains all relevant concepts. For example, the C2-style architecture represented in a UML class diagram in Figure 11 now appears as shown in Figure 13. This diagram clearly distinguishes between components, connectors, and their different kinds of associations. Although components and connectors are derived by constraining the same UML meta class, this has been abstracted away in the diagram. In addition, as mentioned in Section 5.1.3, the formalization of the semantics of attachments in C2 requires an explicit diagrammatic indication (using a notation defined by UML) of the underlying order of the attachment associations. This is the purpose of the upward-pointing triangles on all the attachment associations in Figure 13.

It is also worth pointing out that some concepts of C2 are very different from those of UML and object-oriented design in general. For example, mainstream object-oriented design maintains a strict dichotomy between classes and instances where all the major traits (i.e., the "blueprint") of an instance are specified in its class definition. In contrast, as already discussed, the interface of a C2

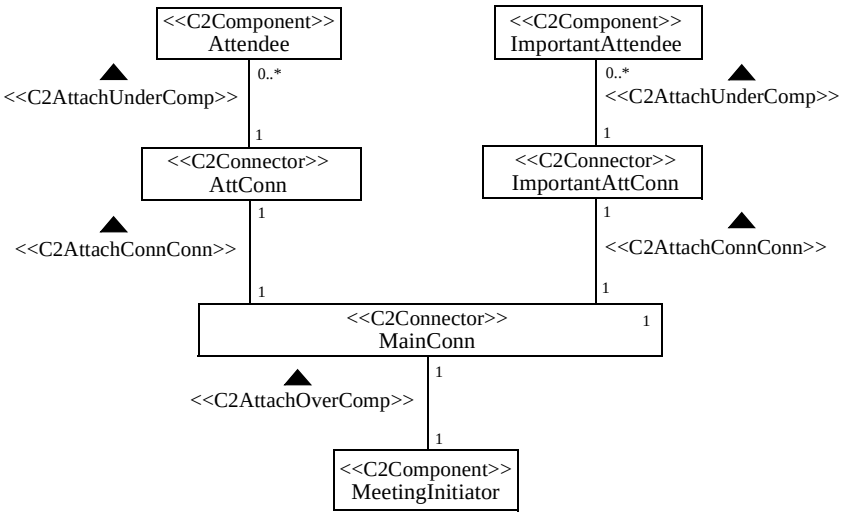


FIGURE 13. The C2-style architecture depicted in Figures 7 and 11 expressed in "constrained" UML. Component interfaces have been omitted to draw attention to the stereotypes defining the different architectural elements. For clarity, we show the resulting topology among classes (i.e., component types) rather than their intances.

connector is determined by context rather than declared; the addition of a new component instance at run-time is considered an architectural change. 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 module. Another conceptual difference is that, strictly speaking, 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 does not involve a key property of C2 and would introduce more complexity than we feel it merits.

5.2 Extensions Based on Wright

The preceding section demonstrates that an ADL that supports a specific architectural style can be modeled in UML. This section demonstrates the applicability of our second strategy to a general-purpose ADL, Wright [3,4]. In addition to the features modeled below, a more recent version of Wright also supports system families, architectural styles, and hierarchical composition [2]. We do not address these newer features here; as mentioned, the preceding section illustrates how support for architectural styles can be incorporated into UML using our second strategy, while Hofmeister et al. [21] have shown that a system family and hierarchical composition can be incorporated using an approach that is in essence an instance of our second strategy.

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,

FIGURE 14. (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.

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.

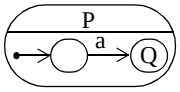
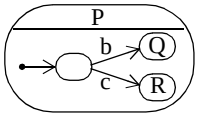
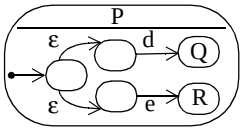
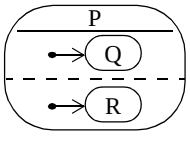
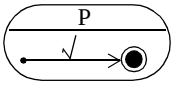
The remainder of the section describes an extension to UML for modeling Wright architectures. Stereotypes and constraints are elided whenever they are obvious from the discussion in this or the previous section.

5.2.1 Behavioral Specification in Wright

Wright uses a subset of CSP [20] 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" [3], it is possible to model Wright's behavioral specifications using UML state machine diagrams.

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 14.

TABLE 2. UML State Machine Templates for Wright's CSP Constructs

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

These two types of state transitions can be used in modeling more complex CSP expressions supported by Wright. Table 2 presents the mapping from CSP to state machines using events with no actions (Figure 14a); the mapping for null events with actions (Figure 14b) is straightforward. It is possible for CSP events to have no associated data. In such a case, the semantics of state machines forces 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 2 can be used as templates from which equivalents of more complex CSP expressions can be formed.[6] 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 14.
WSMtransitionType : enum { event, action }
--2-- An "event" transition consists of a call event only (Figure 14a).
self.WSMtransitionType = event implies
   (self.event->notEmpty and self.event.oclIsKindOf(CallEvent) and
    self.action->isEmpty)
--3-- An "action" transition consists of a null event and a single action (Figure 14b).
self.WSMtransitionType = action implies
   (self.event->isEmpty and self.action->size = 1)
```

Stereotype WrightState for instances of meta class State

```
--1-- All Transitions in a composite WrightState must be WSMTransitions
self.oclIsKindOf(CompositeState) implies
   self.transition->forAll(t | t.stereotype = WSMTransition)
--2-- WrightState applies recursively to its nested states
self.oclIsKindOf(CompositeState) implies
   (self.oclAsType(CompositeState).state->forAll(s |
      s.stereotype = WrightState))
```

Stereotype WrightStateMachine for instances of meta class StateMachine

```
--1-- A WrightStateMachine consists of one of the composite states discussed above, and
   -- partially depicted in Table 2. This constraint is elided in the interest of space.
--2-- All WrightStateMachine transitions must be WSMTransitions.
self.top.oclIsKindOf(CompositeState) implies
   self.top.transition->forAll(t | t.stereotype = WSMTransition)
--3-- The nested states of the top state of a WrightStateMachine must be WrightStates
self.top.oclIsKindOf(CompositeState) implies
   (self.top.oclAsType(CompositeState).state->forAll(s |
      s.stereotype = WrightState))
```

The stereotypes above use the operation *oclAsType* to coerce the associated model element to the specified class of the meta model, thereby providing access the other meta model elements accessible from the specified meta class. They also illustrate the basic pattern we use to constrain state

6. Note that operational models of CSP based on state machines and transition systems are well known; e.g., see Roscoe [45] and Scattergood [47].

machines, namely expressing constraints in terms of model elements reachable from the single, top-level state of a state machine (i.e., its attribute *top*).

5.2.2 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->isEmpty
```

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.operation->forAll(o | o.stereotype = WrightOperation)
```

```
--3-- Exactly one WrightStateMachine is associated with each WrightInterface.
self.stateMachine->size = 1 and
self.stateMachine->forAll(sm | sm.stereotype = WrightStateMachine)
```

```
--4-- The WrightStateMachine of a WrightInterface is expressed only in terms of that interface's
   -- operations, all of which must be operations associated with the call events on the transitions
   -- of the state machine.
self.stateMachine.transition->forAll(t |
   (t.event.oclIsKindOf(CallEvent)) implies
      self.operation->exists(o | o = t.event.operation))
```

5.2.3 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 contains a single WrightStateMachine.
self.stateMachine->size = 1 and
self.stateMachine->forAll(sm | sm.stereotype = WrightStateMachine)
```

Stereotype WrightConnector for instances of meta class Class

--1-- WrightConnectors must implement at least one WrightInterface, which must be a role.

```
self.interface->size >= 1 and
self.interface->forAll(i |
   i.stereotype = WrightInterface and
   i.WrightInterfaceType = role)
```

--2-- A WrightConnector contains a single glue.

```
self.operation->size = 1 and
self.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 the glue's state machine.

```
self.operation.stateMachine.transition->forAll(t |
   (t.event.oclIsKindOf(CallEvent)) implies
      self.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 [3].

```
let glueop = self.operation->select(o | o.stereotype = WrightGlue) in
self.stateMachine->size = 1 and
self.stateMachine->forAll(sm |
   sm.top.oclIsKindOf(CompositeState) implies
      (sm.top.isConcurrent = true and
         sm.top.state->size = 1 + self.interface->size and
         sm.top.state->exists (gs | gs = glueop.stateMachine.top) and
         self.interface->forAll(i |
            sm.top.state->exists (rs | rs = i.stateMachine.top)))
```

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

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

The fifth constraint of stereotype WrightConnector is rather complex, since it must specify a number of conditions arising from the semantics of connectors in Wright. In particular, for the state machine associated with a connector, its top-level state must have concurrent substates. One of these substates is the top state of the glue's state machine, and each of the remaining substates is the top state of the state machine of a (role) interface. Thus, the number of substates must be one plus the number of roles, and the glue and all roles must have their top states represented as substates of the state machine of the connector. Every role plus the glue is represented exactly once in the state machine of the connector, and the states thus represented are concurrently composed to form the top state of the connector.

5.2.4 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.

5.2.5 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 it is undesirable to introduce instances at this level, since we are dealing with design issues. Additionally, we have found that most of the rules for composition of component and connector instances hold for their corresponding types. Therefore, we refer to component and connector types in the stereotypes below.

Stereotype WrightAttachment for instances of meta class Association

```
--1-- Wright attachments are associations between two elements.
self.associationEnd->size = 2

--2-- One end of the association must be the port of a WrightComponent, and the other must be
  -- the role of a WrightConnector.
let ends = self.associationEnd in
ends[1].multiplicity.min = 1 and ends[1].multiplicity.max = 1 and
ends[2].multiplicity.min = 1 and ends[2].multiplicity.max = 1 and
((ends[1].interface.stereotype = WrightInterface and
      ends[1].interface.WrightInterfaceType = port and
   ends[2].interface.stereotype = WrightInterface and
      ends[2].interface.WrightInterfaceType = role) or
 (ends[2].interface.stereotype = WrightInterface and
      ends[2].interface.WrightInterfaceType = port and
   ends[1].interface.stereotype = WrightInterface and
      ends[1].interface.WrightInterfaceType = role) or
```

Stereotype WrightArchitecture for instances of meta class Model

```
--1-- The classes in a WrightArchitecture must all be Wright model elements.
self.modelElement->select(me | me.oclIsKindOf(Class))->forAll(c |
   (c.stereotype = WrightComponent or
    c.stereotype = WrightConnector)

--2-- The associations in a WrightArchitecture must all be Wright model elements.
self.modelElement->select(me | me.oclIsKindOf(Association))->forAll(a |
   a.stereotype = WrightAttachment)

--3-- Each WrightComponent port participates in at most one association with a
  -- WrightConnector role, and vice versa.
let comps = self.modelElement->select(e | e.stereotype = WrightInterface) in
comps.associationEnd->size <= 1

--4 and 5-- WrightComponents and WrightConnectors do not participate in any non-Wright
  -- associations. Similar to constraints 5 and 6, respectively, of stereotype C2Architecture
  -- in Section 5.1.4.
```

The semantics of port-role attachments in Wright are formally defined [4]. 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. Furthermore, unlike the situation described in Figure 5.1.3, where we found it necessary to exploit the underlying order of constrained associations, in the case of WrightAttachment we found it unneces-

```
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 15. A connector specified in Wright (adapted from [3]).

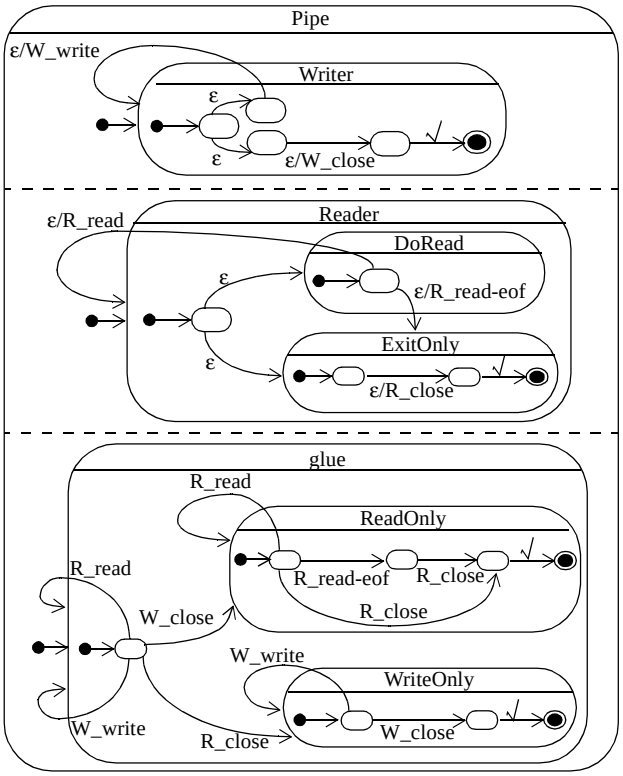


FIGURE 16. UML state machine model of the *Pipe* connector.

sary to account for the ordering, and so that stereotype allows the association to be specified in either order.

5.2.6 Discussion of Wright Extensions

We have defined the stereotypes for Wright in much the same way we did for C2 in Section 5.1. Similar aspects of the two ADLs were captured and *constrained* in similar ways. For example, components and connectors are modeled as stereotyped classes, and their valid compositions in an architecture (i.e., architectural *structure*) as stereotyped associations. At the same time, modeling Wright's component and connector *semantics* required a significant augmentation to what was done for C2.

As an example, consider a Wright specification of a *Pipe* connector, adapted from [3] and shown in Figure 15. The connector is represented in UML by using the stereotyped class WrightConnector; it

is analogous to one of the stereotyped C2Connector classes from Figure 13 and has been omitted for brevity. However, unlike the UML model of C2, which was entirely captured by stereotyping classes and their associations, we model the complex internal *behavior* and *interactions* of a Wright component or connector using the UML state machine diagrams. The state machine model of the *Pipe* connector is shown in Figure 16. 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.

Figure 16 demonstrates how the state machines from Table 2 become atomic building blocks of a CSP specification modeled in UML. In that sense, Table 2 serves the same purpose as OCL stereotypes: it places *constraints* on the allowed uses of a UML construct. Note, however, that while adherence to stereotypes must be ensured by UML-compliant tools, this aspect of our approach does not: a standard UML tool may treat a particular state machine as valid even if it violates the mapping given in Table 2. We do not consider this a problem. UML state machines are a powerful modeling formalism in their own right [18,19] and we have shown how they can model a useful subset of CSP. In doing so, we have given practitioners the choice of either mapping architectural models between UML and Wright in order to exploit Wright's tool support or building comparable support in UML. While neither of the two tasks is trivial, the benefits of either one may outweigh the difficulties.

5.3 Extensions Based on Rapide

This section describes the application of our second strategy to Rapide, an ADL that has a particularly rich semantic basis and supports the specification of architectural constraints [26,27]. With Rapide we also begin to encounter some of the limitations of the second strategy. As will be seen, these limitations stem from weaknesses and ambiguities in the semantics of UML itself.

The underlying behavioral model of Rapide is *partially ordered sets* (or *posets*) *of events*. In particular, the behavior of components is characterized in Rapide primarily in terms of events, which can be associated with typed parameters. Components observe events in the external environment of their execution, and they declare these events as *in actions*. Components generate events into their external environment, and they declare these events as *out actions*. Components can be multi-threaded; as threads within or across components synchronize with each other, they establish causal dependencies between their event streams. Hence, the behaviors of both individual components and a complete architecture can be represented by an event poset, in which event orderings represent causal dependencies introduced by thread synchronizations. The structural aspects of components and architectures in Rapide can be represented with UML class diagrams in a manner similar to the approaches described previously for C2 and Wright. In this section we focus our attention on representing the event-based behavioral modeling features of Rapide.

Rapide supports two kinds of event-based specifications. First, a component behavior can be specified in terms of *state transition rules*, in which a component observes a pattern of events and then generates an associated pattern of events in response. Second, event pattern *constraints* can be used to specify restrictions on the content of the poset generated by a component or an architecture. The

Rapide tools currently restrict the specification of constraints to *never constraints*, which specify event patterns that should never occur within the behavior of the enclosing component or architecture; our extensions for constraints thus adhere to this restriction.

Both kinds of event-based specification in Rapide are expressed in terms of *event patterns*, compound patterns of events expressed using a variety of compositional operators. These operators can be used to specify when events should happen in sequence in the causal order, when they should happen independently of each other in the causal order, when one of a set should happen, when all of a set should happen, and so on.

Analysis of Rapide specifications is carried out via runtime simulation of an architectural model. The Rapide runtime system executes state transition rules to drive the simulation, and it evaluates constraints against the generated poset. In particular, the runtime system *matches* event patterns against corresponding events in the poset of an execution; the matching of any portion of event pattern can be constrained by a Boolean-valued *guard* associated with the portion. *Placeholders* also can be used within an event pattern to achieve unification-style binding to corresponding values in the matched events.

We use a simple *Bank* component for a banking system to illustrate some of the features of Rapide and to illustrate our approach to representing Rapide's event modeling features in UML:

```
type Bank is interface
action in    Open_Account (Customer : Integer),
             Deposit (Acct : Natural; Amt : Float),
             Withdraw (Acct : Natural; Amt: Float);
       out   Assign_Account (Customer : Integer; Acct : Natural),
             New_Balance (Acct : Natural; Amt : Float);
behavior
   type Account_Array is array [Natural] of ref (Float);
   Accounts : Account_Array (1 .. 100, default is ref_to (Float, 0.0));
   Last : var Natural := 0;
begin
   (?C : Integer)
      Open_Account (?C) where $Last < 100  =>  Last := $Last + 1; Assign_Account (?C, $Last);;
   (?A : Natural; ?D : Float)
      Deposit (?A, ?D)  =>  Accounts[?A] := $(Accounts[?A]) + ?D; New_Balance (?A, $(Accounts[?A]);;
   (?A : Natural; ?D : Float)
      Withdraw (?A, ?D)  =>  Accounts[?A] := $(Accounts[?A]) – ?D; New_Balance (?A, $(Accounts[?A]);;
constraint
   never (?A : Natural; ?D : Float)
      New_Balance (?A, ?D) where ?D < 0.0;
   never (?C1, ?C2 : Integer; ?A1, ?A2 : Natural)
      (Assign_Account (?C1, ?A1) –> Assign_Account (?C2, ?A2)) where ?A2 == ?A1;
end Bank;
```

As shown in the specification, a Rapide component is defined by an *interface* type. The *Bank* component observes three kinds of events (opening an account, and depositing and withdrawing money in an account), as specified in the declaration of its in actions. Its out actions specify that it generates two kinds of events (assigning an account number and reporting a new balance). Each of these kinds of events is parameterized with appropriate information (integer customer numbers, natural account numbers, floating-point dollar amounts).

The *behavior* section of the component sets up an array of 100 accounts to hold account balances (with each account initialized to zero, and the variable *Last* used to remember the most recently assigned account number). The component's behavior is specified by three state transition rules, each

of which uses placeholders (the variables denoted with "?") to quantify over all possible occurrences of the events mentioned in the rules. The portion of a rule to the left of the "=>" symbol is the rule's *trigger*, which, if matched, causes the portion to the right of the "=>" symbol, the rule's *body*, to be executed. Note that the bodies specify both generated events and updates to local state variables. In general, the state transition rules of a Rapide component are unordered and are fired repeatedly; the choice of which rule to fire is nondeterministic, although the availability and unavailability of matches for the triggers helps narrow the choice.

The first rule of the *Bank* component says that whenever a customer ?C opens an account (as signified by the occurrence of an *Open_Account* event), then the value of the variable *Last* is incremented (using an assignment statement) and the same customer ?C is assigned the value of variable *Last* as the new account number (through the generation of an *Assign_Account* event). The reading of the variable's value is denoted with "$". The use of the guard **(**the *where* clause) on the event in the trigger ensures that the rule is triggered only if the array of accounts has not been exhausted. The rule would fire for all occurrences of an *Open_Account* event for which the guard is true. The second rule says that whenever an amount ?D is deposited to an account ?A (via a *Deposit* event), then the balance of the same account ?A is updated to reflect the deposit, and then the new balance is reported (via a *New_Balance* event). The third rule handles withdrawals in a manner similar to the rule for deposits.

The *constraints* section of the component describes two patterns of events that should never happen during the execution of the component. The first says that the component should never generate a *New_Balance* event for any account ?A and any amount ?D less than 0.0 (thereby disallowing the reporting of negative balances). The second says that there should never be a compound *sequence* (as indicated by the "–>" operator) of two *Assign_Account* events in which the first event assigns an account ?A1 to a customer ?C1 and the second assigns an account ?A2 to a customer ?C2, with ?A1 and ?A2 being equal (thereby disallowing the assignment of the same account to multiple customers).

Additional component interface types would be declared to complete the specification of the banking system's components, and then instances of the types would be declared in a Rapide *architecture* declaration to specify the configuration of the component instances.

For the remainder of this section we focus in detail on the representation of Rapide component behaviors in UML. We will then conclude with a brief discussion of other features of Rapide.

5.3.1 Representing Rapide Event Patterns in UML

There is a natural correspondence between events in Rapide and *signals* in UML state diagrams. Hence we can use UML signals to model events in our extensions for Rapide. Like a Rapide event, a UML signal corresponds to an atomic occurrence in time, is associated with a set of parameters of arbitrary type, is associated with a single component (i.e., UML class), is generated by a component (as a *send action*) as part of the execution of its state machine, and can be observed by a component (as a *signal event)* during the execution of its state machine. A signal is observed or sent by a component thread asynchronously with respect to the execution of other threads in the same or other compo-

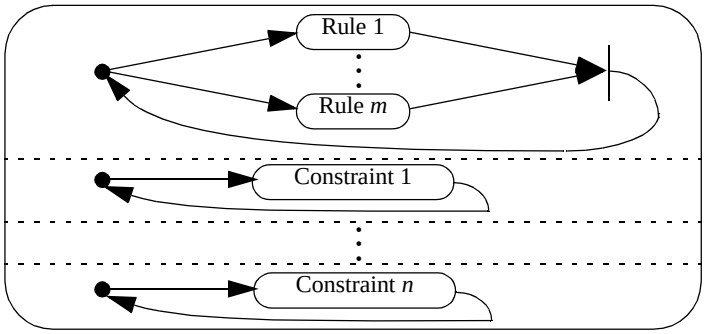


FIGURE 17. State machine template for modeling Rapide component behaviors and behavior constraints in UML

nents. A signal may be broadcast to multiple components and hence is observable by multiple components (and possibly at multiple places within a component state machine), although a particular instance of a signal is observed at most once by any one state machine thread.

A signal event typically triggers a transition in the state machine of a component, and a send action is typically executed as a response. Therefore, we can model Rapide component behaviors (i.e., the set of state transition rules associated with a Rapide component) as UML state machines whose transitions are associated with signal event triggers and send actions that correspond to individual Rapide events. The state machine notation in UML is rich enough to model all of the compound event patterns that can be specified in Rapide, and thus we represent compound event patterns with corresponding composite states in the UML state machines. Furthermore, the trigger of a state machine transition can be constrained by a Boolean *guard condition*, and in this way we can model the guards on Rapide event patterns.

Figure 17 sketches a state machine illustrating the basic approach to modeling Rapide component behaviors in UML. The figure shows a state machine for a component having *m* state transition rules and *n* constraints, with the set of rules forming one substate and each constraint forming an additional parallel substate. Each parallel substate is designed as a looping machine, to reflect the fact that rules are selected and executed repeatedly (until the component terminates), and the fact that constraints are checked repeatedly. Furthermore, each state representing a rule or constraint in Figure 17 in actuality comprises an appropriate composition of nested substates and transitions that models its associated pattern of events in a manner similar to the patterns defined for Wright in Table 2. Within the parallel substate for the rules, the machine is designed so that one rule at a time is fired, with the choice of rule being nondeterministic. Within the parallel substates for each constraint, a special signal is used to represent the violation of the constraint.

Figure 18 shows how the general form of Figure 17 would be instantiated for the *Bank* component described previously. The top parallel substate represents the three state transition rules of the *Bank* component. The middle and bottom parallel substates represent the first and second constraints, respectively, of the *Bank* component. Note that the updates to local state variables in the Rapide state

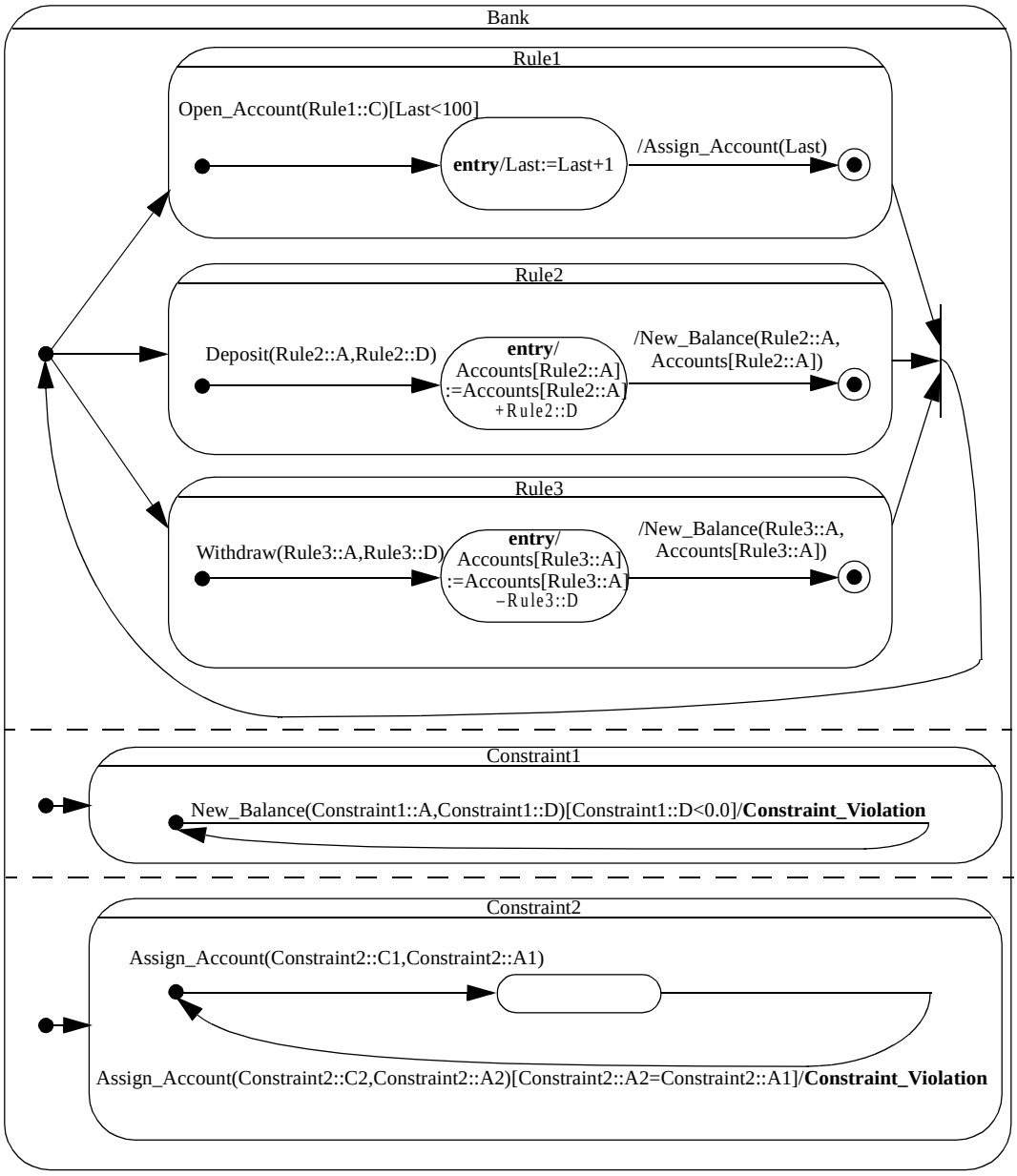


FIGURE 18. UML state machine representation of Rapide *Bank* component.

transition rules are represented by entry actions in intermediate states (i.e., the states between the transitions representing the rule triggers and the events of the rule bodies). As shown in the final transitions of the middle and bottom parallel substates, the signal *Constraint_Violation* is used to represent the violation of the associated constraint. The bottom parallel substate illustrates the representation of a Rapide composite event pattern, in this case, a sequence of two events. The sequencing of the constraint is reflected in the sequencing of the two UML transitions through an intermediate state.

Construction of a state machine such as the one shown in Figure 18 raises a number of additional problems that must be dealt with in the representation of Rapide event patterns in UML. We discuss these problems next.

5.3.2 Representing Rapide Variables and Placeholders in UML

The first serious problem we encountered is how to represent variables and placeholders declared in a Rapide component specification. Local variables declared inside a component (such as *Accounts* and *Last* in the *Bank* component) can be represented in UML as private attributes of the associated class. Thus, *Accounts* and *Last* would be declared as private attributes of the UML class representing the *Bank* component, reducing the problem to representation of placeholders declared within Rapide state transition rules and constraints.

In a UML state machine, a simple variable mentioned on a transition and not otherwise declared as an attribute of the associated class is visible only to that transition and its target state. However, the scope of the placeholders declared in the Rapide state transition rules and constraints extends throughout the associated rule or constraint. To represent this larger scope of the placeholders in the UML state machine, the placeholder names must be qualified to clarify their scope. Furthermore, in order to segregate the scopes of the variables for each state transition rule and constraint, the portion of the UML state machine representing each rule and constraint must be represented by its own named substate, as demonstrated in Figure 18.

For instance, the second state transition rule of the *Bank* component declares a placeholder ?A and uses it in the *Deposit* event in the rule trigger, in the *New_Balance* event in the rule body, and in the update to the *Accounts* array in the rule body. To achieve the desired effect in UML, the variable representing the placeholder ?A is named *Rule2::A* in all cases in substate *Rule2*. Using the unqualified name A everywhere within substate *Rule2* would cause one variable named A to be associated with the *Deposit* transition and a different variable named A to be associated with the *New_Balance* transition, thereby losing the binding of the A used in *New_Balance* to the A used in *Deposit*.

These subtleties involving variables in UML are a consequence of UML's semantics of *namespaces*, which are associated with many different kinds of model elements. However, we were unable to determine from the UML specification [42] whether our use of variable naming described here actually achieves the desired effect, including both the implicit declaration of variable A in Rule2's namespace and the binding behavior induced by the placeholder semantics of Rapide.

5.3.3 Representing Multiple Matches of Rapide Event Patterns in UML

The framework described above for representing Rapide events and event patterns in UML is complicated further by the fact that there may be multiple candidate sets of events that can form a partial match of a Rapide event pattern. All such sets must be maintained until a complete match is found. This applies both to the selection of a particular state transition rule to fire and to the detection of all possible constraint violations. Hence, the UML state machine representing an event pattern must be

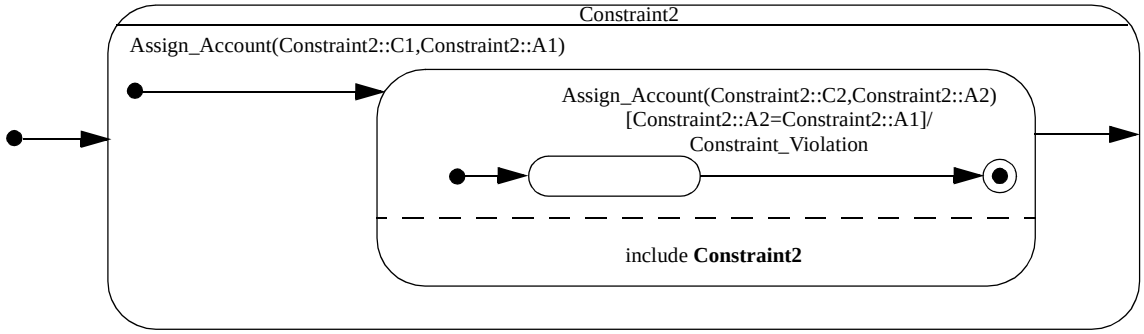


FIGURE 19. Use of submachine state reference in UML representation of Rapide constraints.

constructed in such a way that it is “re-entrant” in a manner corresponding to the matching semantics of Rapide.

We use recursive submachine state references in a UML state diagram to represent the re-entrant behavior of Rapide event patterns. A submachine state reference is a pseudo-state that references another state machine by name, with the semantics being that the named state machine is expanded in place of the reference.[7] Essentially, once there is a match for a single event of a larger event pattern, the state machine representing the pattern must be re-entered to allow subsequent matches of the full pattern.

Figure 19 illustrates how a submachine state reference is used in the representation of a Rapide constraint. The figure presents an improved version of substate *Constraint2* of Figure 18, which represents the second constraint of the *Bank* component. The improved version uses a recursive reference to *Constraint2* (the *include* clause), thereby allowing multiple parallel attempts to match the constraint, one for every occurrence of an *Assign_Account* event.

Rapide state transition rules can be treated in a similar manner, except that a complete match of one rule’s trigger terminates any partial matches for other rules. Although not shown here, the termination can be achieved with judicious use of signals.

5.3.4 Representing Rapide Event Causality in UML

Our representational framework for Rapide events is complicated still further by the need to maintain information about the causal ordering of events in Rapide. The problems in maintaining information about causality in distributed systems are well known [25]. The traditional solution for explicitly representing the partial order of a set of events is to associate a *vector timestamp* with each event [11]. In UML, many of the more subtle nuances of event ordering semantics are left unspecified, or are specified very loosely [42]. In particular, we interpret the semantics of UML (especially the semantics of event queues for state machines) as allowing one component to send signals in one order

7. It is not clear whether the UML semantics [42] actually allows submachine state references to be recursive, and we know of no UML tools that support them.

and another component to observe the signals in a completely different order, with no means for avoiding deadlock in the observing component. Hence, signals in UML do not respect typical notions of causality. In Rapide, however, an event pattern may specify that, say, a sequence of causally ordered events is to be matched, or that a set of events that are not causally ordered is to be matched; the Rapide runtime system will gather the necessary information about the generated events and their causal relationships in order to perform the match as specified.

There are a number of possible approaches to addressing causality in UML. One approach would be to ignore causality, forgo the possibility of constraining component behaviors with respect to causally related events, and simply accept UML's "looser" semantics for event ordering. A second approach would be to specify and use vector timestamps as additional model elements for encoding the partial order within the signals used in a UML model; OCL stereotypes would be needed to formalize the semantics of the vector timestamps and to ensure their consistent and correct use across the model. A third approach would be to specify an additional model element that is the equivalent of the Rapide runtime system; this would still require the declaration and use of additional information in the signals used in a UML model.

Each of these approaches has its strengths and weaknesses, but they all share the disadvantage of greatly complicating the resulting models. This suggests that causality among events is an architectural concern that simply cannot be represented in UML in a straightforward manner at the present time. While we intend to explore these possibilities in greater detail in the future, we are hopeful that future versions of UML will incorporate a more precise and comprehensive semantics for event ordering.

5.3.5 Discussion of Rapide Extensions

This section has focused on representing Rapide's features for event-based behavioral modeling of architectural components. Rapide is a large language that contains a number of additional features, including features for encapsulating specifications in reusable modules and for specifying component behaviors with traditional procedural statements. Representation of these features should be straightforward in UML, and hence we have ignored them here.

Rapide differs from other ADLs in a number of ways. First, Rapide does not support a notion of connectors as first class architectural elements. Instead, *connection rules* are used to specify *interactions* between components. In particular, a connection rule is defined as part of a Rapide architecture declaration, and it specifies how events generated by one set of components are connected to events observed by another set of components. Hence, the configuration of an architecture in Rapide is *implicit*, arising as a result of the firing of the architecture's connection rules. Many of the concepts underlying architectural connection rules are similar to those underlying component state transition rules. Hence, their representation in UML would be similar. The main difference is that, with Rapide components represented as UML classes, and with Rapide component behaviors represented as UML state machines, the representation of Rapide connection rules in UML requires the association of sig-

nals sent by one component's state machine with the signals received by another's state machine. However, while UML provides syntactic mechanisms for achieving such associations, the semantics of the resulting composite behavior is problematic, as described in Section 5.3.4.

As discussed above, Rapide's model of events and event patterns has many natural analogs in the features of UML state diagrams. This is not too surprising since UML state diagrams are based on statecharts [18]. Both Rapide and statecharts are suited to specifying the behavior of reactive systems, and both exploit the complementary functions provided by events and states in operational models of behavior. But in constraining UML to represent Rapide, we were confronted by a number of semantic ambiguities and limitations of UML, especially in its semantics for state diagrams. These problems suggest that, in general, the semantics of UML will need to be made more precise in order to support modeling of certain kinds of architectural concerns.

6 Related Work

UML represents a maturation in the development of object-oriented design notations. It offers a diverse collection of notations for capturing many aspects of the software development lifecycle, including not only traditional design concerns (such as functional decomposition), but also aspects of requirements analysis (particularly domain modeling), implementation, and testing (particularly scenario-based functional testing). This paper has described approaches to overcoming a key weakness of UML, its lack of adequate support for modeling software architectural concerns. In this section we discuss related work, including techniques for exploiting the architectural modeling capabilities of one language within another language, and work on improving the semantic basis and modeling features of UML and other object-oriented notations.

6.1 Architectural Interchange

This paper has described two strategies for providing software architects with a variety of architecture modeling capabilities in a single, widely-used notation. One common thread between the two strategies is their attempt at *standardization* — finding and exploiting a base notation (UML) that is general enough to capture needed capabilities without providing too many opportunities for incompatibilities or adding too much complexity. The architecture research community has attempted a different approach to supporting the diverse needs of architects, namely *architectural interchange*, as a way of tolerating the existence and use of multiple, incompatible notations. In particular, architectural interchange is intended to allow architects to move between different ADLs so that they need not all agree to use a single, standard ADL.

ACME is an architecture interchange language that is intended to support automatic transformation of a system modeled in one ADL to an equivalent model in another ADL [15]. This allows architects to model and analyze their system architecture in one ADL and then translate the model to another ADL for further analysis. 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, ACME's ontology is smaller than UML's meta model and focuses only 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 [36]. Furthermore, a study by Di Nitto and Rosenblum demonstrated wide variation and little overlap in the abilities of ADLs to support modeling of architectural styles induced by common middleware infrastructures [7], suggesting that there is little in the way of semantic commonality among ADLs to be interchanged. For these reasons, ACME emphasizes a partial and incremental approach.

The approach discussed in this paper does not use translation between notations, but is rather based on a core model (Strategies 1 and 2), possibly with several independent extensions (Strategy 2). 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 has the potential to ease 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 (and growing) tool support, and provides explicit extension mechanisms.

6.2 Architectures as Collections of Views

The work described in this paper focuses on a set of approaches to software architectures that has emerged from one part of the research community: specification of structural and behavioral aspects of a software system centered around a (formal) notation, an ADL. Another part of the community has tried to identify useful architectural perspectives, or *views*. A model expressed in an ADL essentially provides a single view of an architecture, which is typically formal. In contrast, the views constructed in multiple-view approaches are often informal or semi-formal. Two representative examples of work with multiple views are provided by Kruchten [23] and Soni et al. [41].

Kruchten presents the 4+1 view model of software architectures. The four main views are the *logical*, *process*, *development*, and *physical* views. Together, these views capture a software system's architecture. Kruchten provides system usage scenarios, similar to UML's use cases, as the fifth view to relate the other four.

Similarly, Soni et al. identify four structural categories of software architectures—*conceptual*, *module interconnection*, *execution* and *code*. The module interconnection view is a refinement of the

conceptual view; it provides a functional and layered decomposition of the system. The execution and code views closely correspond to Kruchten's dynamic and static views, respectively.

Although these approaches are in certain ways more comprehensive than ADL-centered approaches, the strategies for modeling architectures in UML described in this paper are applicable to the multiple-view approaches as well. Indeed, Soni et al. demonstrate how UML can be constrained to model their four architectural views [21]. Their approach is an instance of our Strategy 2, whereby UML constructs are stereotyped to model architectural constructs. The strategy and examples we presented in Section 5 go beyond their approach in that we also use OCL to formally specify the stereotypes.

6.3 Other Work with UML and Design Notations

In addition to the work of Soni et al. with UML mentioned above, four other related efforts with object-oriented design notations deserve mention.

Recently, Garlan and Kompanek conducted a study whose goal was to enumerate and evaluate different options an architect has in selecting UML modeling constructs to represent architectural structure (i.e., components, connectors, systems, and styles) [14]. Unlike our choice of (stereotyped) UML classes and class instances to represent architectural elements, Garlan and Kompanek investigate five possibilities: UML classes as architectural types and objects as their instances; UML stereotypes as types and classes as instances; UML classes as both types and instances; UML components; and UML subsystems (stereotyped UML packages). Their study shows that, while each of the five choices has its merits, none of them is an ideal fit for the needs of software architectures in terms of semantic match, understandability, or completeness. While our work has also focused on non-structural aspects of architectures (behaviors, interactions, and constraints), we view Garlan and Kompanek's study as a useful extension to the work presented in this paper.

The work that is perhaps most similar to our own is the tailoring of UML to support real-time systems development. Selic and Rumbaugh [48,50] describe the use of stereotypes to augment UML with architecture modeling constructs borrowed from the Real-Time Object-Oriented Method (ROOM) [49]. The main architectural building block in this approach is the *capsule* (i.e., a simple or compound component), which provides one or more *ports* to support interaction with other capsules (including nested sub-capsules). Ports of different capsules are connected via *connectors* (which capture interaction relationships between capsules and which resemble simple attachments between architectural elements found in ADLs). UML collaboration diagrams are used to model an architecture as a configuration of capsules. Behavior is modeled in this approach as *protocols*, which are specified by of stereotypes that identify a protocol and its *protocol roles*. UML state machines are used to specify behavioral models of capsule and protocol implementations, for which the UML state machine notation has been augmented with a stereotype for a *chain state* construct (a degenerate form of state used to chain transitions between internal states of different compound states). Instances of all these newly-introduced architectural constructs are specified as stereotyped instances of existing UML meta

model elements, so that the resulting models are still valid UML. Hence, this approach is a successful instance of our Strategy 2, and we view it as an independent confirmation of the utility of that strategy.

Cheng and colleagues have worked on strengthening the formal underpinnings of OMT (the Object Modeling Technique), a precursor to UML [6,60]. In particular, they describe an approach to deriving algebraic specifications from OMT models (in particular, Larch specifications from OMT object models [6] and LOTOS specifications from OMT dynamic models [60]). The derived specifications can then be subjected to rigorous, automated analyses for design errors, including inconsistencies between the associated object and dynamic views. While not specifically addressing architectural concerns, their research is very much in the spirit of research on ADLs, which has attempted to create languages with precise formal semantics to enable early analysis of architectural models. However, while Cheng's work involves deriving separate formal models from an object-oriented notation, our work involves enhancing such a notation with well-defined architecture modeling capabilities.

The creators of UML have developed a process model, the Unified Software Process, for applying UML in object-oriented analysis and design [22,24]. The Unified Software Process is oriented toward early development of an architecture from use cases. It is worth noting that the Unified Software Process is supported by a UML profile comprising a number of UML stereotypes and tagged values—much like the approach described in this paper—but also requires additional graphical notation beyond what UML provides (along the lines of our Strategy 3, described in Section 3.3). In the Unified Software Process, the *architecture description* of a system is a cross-section of the "architecturally-significant" elements of the models in the *architecture baseline*. The architecture baseline comprises early versions of the use-case model, analysis model, and design model developed as part of the *elaboration phase* [22].[8] The architecture description is developed iteratively and is considered to be fundamentally important for analyzing and understanding the structure, behavior, performance and other global characteristics of a software system under construction. Hence, the architecture description achieves the level of importance advocated by the software architecture research community. However, the fact that the Unified Software Process considers the architecture to be an implicit attribute of existing UML models rather than an explicit model in its own right is at odds with the accepted orthodoxy of software architecture research.

7 Discussion and Conclusions

From our experience to date, adapting UML to address architectural concerns seems to require reasonable effort, to be a useful complement to ADLs (and, potentially, their analysis tools), and to 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. Finally, the example ADL-specific

8. Roughly speaking, the elaboration phase is the phase in which requirements are elaborated into an initial design.

extensions performed as illustrations of our second strategy (Section 5) can be looked at as a basis of an evolvable, broadly applicable extension of UML for architectural modeling.

The two strategies we describe are not without drawbacks: For each architectural approach and ADL, we introduced a somewhat specialized usage convention (Strategy 1) or semantic extension (Strategy 2). Furthermore, our second strategy relied heavily on OCL, whose formality may hinder wide adoption of the strategy even though end users of the constrained UML model typically will not need to write OCL constraints. 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 architectural models. As serious as these drawbacks may be, we believe them to be eclipsed by the potential benefits that can accrue.

This effort has thus furthered our understanding of UML and its suitability for supporting architecture-based software development. While it may not represent a definitive study of the relationship between UML and ADLs, it has given us valuable insights on which we intend to base our future work. These insights and areas of future work are discussed below.

7.1 Key Insights in Relating UML and ADLs

7.1.1 Software Modeling Philosophies

Neither UML nor ADLs constrain the choice of implementation language or require that any two components be implemented in the same language or thread of control. ADLs or styles may assume particular communication protocols, such as C2's asynchronous message passing, and UML typically supports such restrictions. Unlike some ADLs' component specifications, UML classes only support specifications of events that may be received, but not those that may be sent.

The behavior of architectural constructs (components, connectors, communication ports, and so forth) to a large degree can be modeled with UML's sequence, collaboration, statechart, and activity diagrams. Existing ADLs are typically able to support only a subset of these kinds of semantic models [36].

7.1.2 Assumptions about Intended Usage

Like any notation, UML embodies its creators' assumptions about its intended usage. "Architecting" a system in the sense it is used in the software architecture community and in this paper—by employing conceptual components, connectors, and their configurations, exploiting rules of specific architectural styles, and modeling local and global architectural constraints—was not an intended use of UML. UML's genesis and its primary strength is modeling software from an object-oriented perspective, where the major system elements (components), their constituent building blocks (subcomponents), their interactions (connectors), and the data exchanged among them are all represented in the same way—as objects. Furthermore, UML embodies a philosophy of maximum flexibility in the use of the notation by designers and developers, which necessarily comes at a loss of some formality

and rigor. This would appear to conflict with the expectations ADL purveyors have about the level of formality desired or needed by practicing software designers. For these reasons, while one can indeed focus on the different architecturally-relevant perspectives when modeling a system in UML, a software architect may find that the support for those perspectives provided by UML only partially satisfies his/her needs.

7.1.3 Problem Domain Modeling

UML provides extensive support for modeling a problem domain. On the other hand, architectural models described in ADLs often hide much of the information present in a domain model, as seen in Section 4. This can be considered a shortcoming of ADLs given that a domain model is considered to be a centerpiece of a large category of software architecture models, namely domain-specific software architectures (DSSA) [58]. 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 useful architectural complement [32,58].

7.1.4 Architectural Abstractions

Some concepts of software architectures are very different from those of UML (and of object-oriented design in general). Connectors are first-class entities in many ADLs. As demonstrated in this paper, the functionality of a connector can typically be abstracted by a class or component. However, connectors may have properties that are not directly supported by a UML class. For example, the interfaces of C2 connectors are context-reflective; our attempts to model such connectors in UML required specialized modeling of application-specific connector classes.

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 intentions and practices of users, then that language should aspire to reflect those intentions and practices [53]. 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 other abstractions (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 constraints also raises them in the consciousness of the designer.

7.1.5 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 that a resulting system has certain desirable properties or lacks undesirable properties.

Standard UML provides no support for architectural styles; the rules of different styles somehow have to be "built into" UML (e.g., with a style-specific profile). We demonstrated in Section 5 how this can be done using stereotypes. On the other hand, choosing to use UML "as is" introduces a problem in this regard: while every architecture designed using the strategy outlined in Section 4 adheres to the UML meta model, and can be relatively easily understood by a typical UML user and manipulated with standardized UML tools, there is no guarantee that the designer will always adhere to the rules of a given style.

7.1.6 Implementation Support

An ADL is frequently accompanied by tools that generate (parts of) the infrastructure of systems modeled in the ADL. This infrastructure is often also referred to as "glue code" [52] and it enforces the desired topology, interfaces, and interactions among system components. At the same time, ADL specifications do not supply enough detail to generate the entire system (e.g., the internal functionality of individual components), leaving a sizable task to external tools or human developers. Using UML in the manner discussed in this paper has the potential to augment the ADL-specific generation tools with similar tools provided to generate implementations from UML models. For example, an implementation of a statechart model of a component, such as the one depicted in Figure 16, can be generated by using the STATEMATE tool [19]. Given that the consistency between the ADL and UML models of the architecture is ensured by the OCL constraints provided in our mapping, it is reasonable to expect that the "internal" part of the implementation generated from the UML model will fit with the "external" part generated from the ADL. We are currently validating this hypothesis in the context of our prototype environment for mapping ADL specifications to UML specifications [1].

7.2 Future Work

We intend to expand this work in several directions, including providing tool support for using UML in architecture modeling, maintaining traceability and consistency between architectural and design decisions, and combining the existing implementation generation capabilities for ADLs and UML. We also intend to draw upon our experience to date to suggest specific extensions needed in the UML meta model to better support software architectures.

We have already begun to address several of these issues. We have developed an initial integration of DRADEL [34], an environment for C2 style architecture-based development, with Rational Rose [40], an environment for software design and implementation with UML [1]. The integration enables automated mapping from an architecture described in C2's ADL into UML using both Strategies 1 and 2. Currently, this mapping is uni-directional, and the UML model is consistent with respect to the architecture only initially; any subsequent refinements of the UML model may violate architectural decisions. Also, as additional views are introduced into the design (e.g., activity and deployment dia-

grams), their consistency with the existing views (e.g., state and class diagrams) must be ensured. To this end, we are beginning to develop a set of techniques and associated tools to ensure full integration of views in UML [8,9]. The ultimate goal of this work is to apply and evaluate the two strategies described here in the context of large-scale case studies.

Another, long-term goal is to augment existing capabilities for generating implementations from architectures. Such capabilities have typically only dealt with a system's overall interconnection and interaction characteristics (recall Section 7.1.6). We intend to augment them with support for implementing individual architectural elements that is already available or is emerging in the context of different UML modeling diagrams. Ensuring that individual design elements are consistent with the overall architecture is a necessary first step in accomplishing this task in a meaningful way.

Finally, it is important to note that the relevant future work is not restricted to our research, but also includes UML itself. In the process of revising and completing this work over the past three years, UML has gone through several revisions (from UML 1.0 to the current 1.3). While advertised as "minor," these revisions forced us to change several details of our mappings from the three ADLs (e.g., the details concerning the meta model and OCL). The revisions also resulted in a shortcoming that was introduced into UML only with version 1.3: UML 1.1 was actually able to model both provided and required operations of a class, while UML 1.3 is not. We hope that the major revision to UML 2.0, which is currently in preparation, can remedy some of the shortcomings of UML as an architecture modeling notation identified in this paper.

Acknowledgments

We thank the anonymous referees for their extremely thorough and helpful comments on the manuscript.

This material is based upon work supported by the National Science Foundation under Grant No. CCR-9624846, Grant No. CCR-9701973, and Grant No. CCR-9985441. 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 the 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. M. Abi-Antoun and N. Medvidovic. Enabling the Refinement of a Software Architecture into a Design. In *Proceedings of the Second International Conference on the Unified Modeling Language (UML '99)*, pp. 17–31, Fort Collins, CO, October 1999.

2. R.J. Allen. *A Formal Approach to Software Architecture*. Ph.D. thesis, Carnegie Mellon University, 1997.
3. R. Allen and D. Garlan. Formalizing Architectural Connection. In *Proceedings of the 16th International Conference on Software Engineering*, pp. 71–80, Sorrento, Italy, May 1994.
4. R. Allen and D. Garlan. A Formal Basis for Architectural Connection. *ACM Transactions on Software Engineering and Methodology*, pp. 213–249, July 1997.
5. G. Booch, I. Jacobson and J. Rumbaugh. *The Unified Modeling Language User Guide.* Addison-Wesley, 1998.
6. R.H. Bourdeau and B.H.C. Cheng. A Formal Semantics of Object Models. *IEEE Transactions on Software Engineering* 21(10), pp. 799–821, October 1995.
7. E. Di Nitto and D.S. Rosenblum. Exploiting ADLs to Specify Architectural Styles Induced by Middleware Infrastructures. To appear in *Proceedings of the 21st International Conference on Software Engineering*, Los Angeles, CA, May 1999.
8. A. Egyed and N. Medvidovic. Extending Architectural Representation in UML with View Integration. In *Proceedings of the Second International Conference on the Unified Modeling Language (UML '99)*, pp. 2–16, Fort Collins, CO, October 1999.
9. A. Egyed and N. Medvidovic. A Formal Approach to Heterogeneous Software Modeling. In *Proceedings of the Third International Conference on the Fundamental Approaches to Software Engineering (FASE 2000)*, pp. 178–192, Berlin, Germany, March–April 2000.
10. M.S. Feather, S. Fickas and A. van Lamsweerde. Requirements and Specification Exemplars. *Automated Software Engineering* 4(4), pp. 419–438, 1997.
11. C.J. Fidge. Logical Time in Distributed Computing Systems. *IEEE Computer* 24(8), pp. 28–33, August 1991.
12. D. Garlan (ed.). *Proceedings of the First International Workshop on Architectures for Software Systems*, Seattle, WA, April 1995.
13. 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.
14. D. Garlan, A. Kompanek. Reconciling the Needs of Architectural Description with Object-Modeling Notations. To appear in *Proceedings of the Third International Conference on the Unified Modeling Language (UML 2000)*, York, UK, October 2000.
15. D. Garlan, R. Monroe and D. Wile. ACME: An Architectural Interconnection Language. In *Proceedings of CASCON'97*, Toronto, Canada, November 1997.
16. D. Garlan, F.N. Paulisch and W.F. Tichy (eds.). Summary of the Dagstuhl Workshop on Software Architecture, February 1995. Reprinted in *ACM Software Engineering Notes*, pp. 63–83, July 1995.
17. D. Garlan and M. Shaw. *An Introduction to Software Architecture: Advances in Software Engineering and Knowledge Engineering*, Volume I. World Scientific Publishing, 1993.
18. D. Harel. Statecharts: A Visual Formalism for Complex Systems. *Science of Computer Programming* 8, pp. 231–274, 1987.
19. D. Harel and A. Naamad. The STATEMATE Semantics of Statecharts. *ACM Transactions on Software Engineering and Methodology* 5(4), pp.293–333, October 1996.
20. C.A.R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.
21. C. Hofmeister, R.L. Nord and D. Soni. Describing Software Architecture with UML. In *Proceedings of The First Working IFIP Conference on Software Architecture (WICSA1)*, pp. 145-159, San Antonio, TX, February 1999.
22. I. Jacobson, G. Booch and J. Rumbaugh. *The Unified Software Development Process*. Addison-Wesley, 1999.
23. P.B. Kruchten. The 4+1 View Model of Architecture. *IEEE Software*, pp. 42–50, November 1995.

24. P.B. Kruchten. *The Rational Unified Process*. Addison-Wesley, 1998.

25. L. Lamport. Time, Clocks and the Ordering of Events in a Distributed System. *Communications of the ACM* 21(7), pp. 558–565, July 1978.

26. D.C. Luckham, J.J. Kenney, L.M. Augustin, J. Vera, D. Bryan and W. Mann. Specification and Analysis of System Architecture Using Rapide. *IEEE Transactions on Software Engineering* 21(4), pp. 336–355, April 1995.

27. D.C. Luckham and J. Vera. An Event-Based Architecture Definition Language. *IEEE Transactions on Software Engineering* 21(9), pp. 717–734, September 1995.

28. 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.

29. J. Magee and D.E. Perry (eds.). *Proceedings of the Third International Software Architecture Workshop (ISAW-3)*, Lake Buena Vista, FL, November 1998.

30. 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.

31. 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)*, pp. 190–198, Boston, MA, May 1997. Also in *Proceedings of the 1997 International Conference on Software Engineering (ICSE'97)*, pp. 692–700, Boston, MA, May 1997.

32. 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.

33. N. Medvidovic and D.S. Rosenblum. Assessing the Suitability of a Standard Design Method for Modeling Software Architectures. In *Proceedings of the First IFIP Working Conference on Software Architecture (WICSA1)*, pp. 161–182, San Antonio, TX, February 1999.

34. N. Medvidovic, D.S. Rosenblum and R.N. Taylor. A Language and Environment for Architecture-Based Software Development and Evolution. In *Proceedings of the 21st International Conference on Software Engineering*, pp. 44–53, Los Angeles, CA, May 1999.

35. N. Medvidovic and R.N. Taylor. Separating Fact from Fiction in Software Architecture. In *Proceedings of the Third International Software Architecture Workshop (ISAW-3)*, pp. 105–108, Lake Buena Vista, FL, November 1998.

36. N. Medvidovic and R.N. Taylor. A Classification and Comparison Framework for Software Architecture Description Languages. *IEEE Transactions on Software Engineering* 26(1), pp. 70–93, January 2000.

37. 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.

38. N. Mehta, N. Medvidovic and S. Phadke. Towards a Taxonomy of Software Connectors. In *Proceedings of the 22nd International Conference on Software Engineering (ICSE 2000)*, pp. 178–187, Limerick, Ireland, June 2000.

39. M. Moriconi, X. Qian, and R.A. Riemenschneider. Correct Architecture Refinement. *IEEE Transactions on Software Engineering*, pp. 356–372, April 1995.

40. Rational Software Corporation. *Rational Rose 98: Using Rational Rose*.

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

42. Object Management Group. *OMG UML Specification Version 1.3*. March 2000.

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

44. 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 1998.

45. A.W. Roscoe. *Two Papers on CSP.* Technical Monograph PRG-67, Oxford University Computing Laboratory, 1988.

46. J. Rumbaugh, I. Jacobson and G. Booch. *The Unified Modeling Language Reference Manual.* Addison-Wesley, 1998.

47. B. Scattergood. *The Semantics and Implementation of Machine-Readable CSP.* D. Phil. dissertation, Oxford University, 1998.

48. B. Selic. Turning Clockwise: Using UML in the Real-Time Domain. *Communications of the ACM* 42(10), pp. 46–54, October 1999.

49. B. Selic, G. Gullekson and P. Ward. *Real-Time Object-Oriented Modeling*. Wiley, 1994.

50. B. Selic and J. Rumbaugh. *Using UML for Modeling Complex Real-Time Systems*. ObjectTime white paper, March 11, 1998. Accessed June 2000 at Web site http://www.objectime.com/otl/technical/umlrt.pdf.

51. M. Shaw. Procedure Calls are the Assembly Language of Software Interconnection: Connectors Deserve First-Class Status. In *Studies of Software Design, Proceedings of an ICSE '93 Workshop*, Lecture Notes in Computer Science No 1078, Springer-Verlag, pp. 17–32, 1996.

52. 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.

53. M. Shaw and D. Garlan. Formulations and Formalisms in Software Architecture. In J. van Leeuwen (ed.), *Computer Science Today: Recent Trends and Developments*, Springer-Verlag Lecture Notes in Computer Science, Volume 1000, 1995.

54. 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/.

55. 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.

56. 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.

57. Tigris. Design your UML models with ArgoUML. http://argouml.tigris.org/v08/press-release.html.

58. W. Tracz. DSSA (Domain-Specific Software Architecture): Pedagogical Example. *ACM SIGSOFT Software Engineering Notes*, pp. 49–62, July 1995.

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

60. E.Y. Wang, H.A. Richter, B.H.C. Cheng. Formalizing and Integrating the Dynamic Model within OMT. In *Proceedings of the 1997 International Conference on Software Engineering*, pp. 45–55, Boston, MA, May 1997.

61. J.B. Warmer and A.G. Kleppe. *The Object Constraint Language: Precise Modeling with UML.* Addison-Wesley, 1998.

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