Enabling the Refinement of a Software Architecture into a Design

Marwan Abi-Antoun and Nenad Medvidovic

Computer Science Department
University of Southern California
Los Angeles, CA 90089-0781, USA
{marwan,neno}@sunset.usc.edu

Abstract. Software architecture research has thus far mainly addressed formal specification and analysis of coarse-grained software models. The formality of architectural descriptions, their lack of support for downstream development activities, and their poor integration with mainstream approaches have made them unattractive to a large segment of the development community. This paper demonstrates how a mainstream design notation, the Unified Modeling Language (UML), can help address these concerns. We describe a semi-automated approach developed to assist in refining a high-level architecture specified in an architecture description language (ADL) into a design described with UML. To this end, we have integrated DRADEL, an environment for architecture modeling and analysis, with Rational Rose®, a commercial off-the-shelf (COTS) UML modeling tool. We have defined a set of rules to transform an architectural representation into an initial UML model that can then be further refined. We believe this approach to be easily adaptable to different ADLs, to the changes in our understanding of UML, and to the changes in UML itself.

1 Introduction

Architecture-based development is currently receiving a lot of attention, from both the academic and industrial communities. This is evidenced by numerous architecture-based approaches to software development that have emerged and an increasing number of conferences and symposia that focus on software architectures [1, 2, 3, 4, 5]. However, to a large degree, the claimed potential of software architectures remains unfulfilled. One reason is that most existing research is centered around highly specialized architecture description languages (ADLs), only addressing *modeling* and *analysis* of specific aspects of architectures (e.g., their structure), and rarely focusing on the broader *development* picture. A detailed survey of the area [6] shows limited overall tool support that repeatedly focuses only on well-understood problems: editing, parsing, syntactic and structural analysis, and so forth. The narrow focus and limited tool support partially account for the lack of transitioning of architecture research into the software development mainstream.

On the other hand, the industrial segment of the software development community has focused on comprehensive approaches to software development, through integrated methodologies and tools that address the entire software lifecycle. One important recent development is the Unified Modeling Language (UML) [7, 8], a

© Springer-Verlag Berlin Heidelberg 1999

general purpose modeling language that provides an expressive and extensible graphical notation. UML is emerging as a *de facto* software design standard with the potential for industry wide adoption and extensive, sophisticated tool support. However, existing work [10,11] has identified areas in architectural design where the expressiveness of standard UML alone is not entirely adequate.

We hypothesize that software architecture researchers can benefit from integrating their approaches with UML. The benefits of doing so would be manifold: easier adoption of their work, much more extensive software modeling and tool support, the ability to exploit UML in refining coarse-grained architectural elements and implementing them, and so on. This would, in turn, benefit mainstream UML users by providing them with powerful notations and analysis tools that have an explicit architectural focus and are specialized for certain development situations.

In order to validate our hypothesis, we have begun studying the relationship between UML and ADLs. In our previous work, we discussed and demonstrated the possible strategies for marrying architecture-based approaches with UML [9, 10, 11]. This paper further refines our ideas. In particular, we define a set of rules for transforming ADL-based models into their UML counterparts and provide tool support for automatically generating an initial UML model corresponding to the ADL specification. The UML model becomes a starting point for refining the architecture into a design and eventually an implementation. To this end, we have integrated two environments: DRADEL [12], which focuses on ADL-based modeling, analysis, and simulation of architectures, and Rational Rose® [13], which supports software development using UML. Although the resulting environment supports transformation from a specific ADL (C2SADEL [12]) to UML, we believe that the underlying methodology is general and flexible enough to be applicable to other ADLs.

The remainder of the paper is organized as follows. Section 2 discusses the issues in refining an architecture into a design while ensuring consistency and traceability. In Section 3, we propose our approach for refining C2SADEL into UML, while Section 4 discusses our implementation of this approach. An overview of related work and conclusions round out the paper.

2 Motivation

The problem of refining a coarse-grained architecture specified in an ADL into a design described in a modeling language comprises two separate tasks:

1. Refining the architecture into the design (forward engineering)
2. Abstracting the architecture from the design (reverse engineering)

Although the focus of our work to date has been on refinement, we briefly discuss both aspects of the problem below. In an iterative design process, the designer may need to apply both refinement and abstraction repeatedly.

2.1 Refinement

For our purposes, the starting point of refinement is an architecture composed of coarse-grained components and connectors, and their configurations. The architecture adheres to some architectural style (e.g., client-server, pipe-and-filter, layered) and its representation may be formalized using an ADL. That architecture is refined into a design, and eventually, an implementation.

Given that architectures are intended to describe systems at a high-level of abstraction, directly refining an architectural model into a design or implementation may not be possible. One reason is that the design space rapidly expands with the decrease in abstraction levels, as shown in Fig. 1.

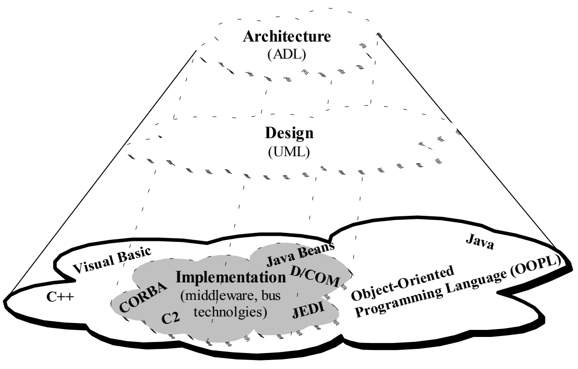

Fig. 1. From Architecture to Implementation

A solution to this problem is to bound the target (implementation) space by employing specific middleware technologies. A more general refinement approach includes design as an intermediate step. During design, constructs such as classes with attributes, operations, and associations, instances of objects collaborating in a scenario, and so forth, are identified. These are more effectively expressed in a notation like UML than in an ADL. However, various problems might arise. First, the design may no longer be faithful to the rules of the selected architectural style. Second, maintaining traceability is inherently difficult, because of the possible many-to-many mappings from the elements in the problem domain to the elements in the solution space (Fig. 2): a given element from one space can map to zero, one, or more elements in the "lower level" space. Third, some architectural elements, such as connectors, are given first class status in architectures [14], but may not have direct design or implementation counterparts. Typically, connectors are "designed away" into various class associations and object interactions or are "coded away" into programming language statements distributed across different components.

2.2 Abstraction

Since any design process is inherently iterative, and software designers often have to deal with legacy systems, it is also desirable to be able to perform the reverse step, i.e., to abstract the architecture from the design. This involves obtaining from the UML model a description similar to the one represented in an ADL, using either formal transformations [9] or various reverse engineering heuristics (e.g., [15]).

However, converting an architectural representation in an ADL to a representation in UML and then back might lead to either of the following cases:

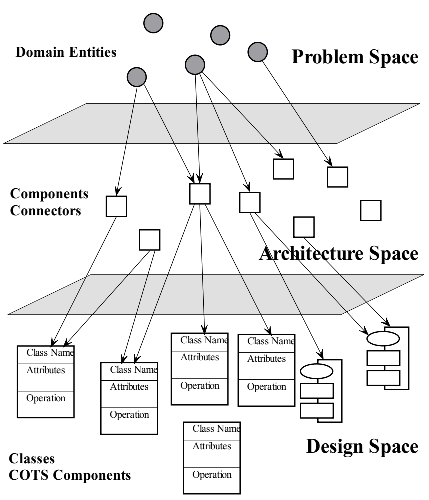

Fig. 2. Traceability during Refinement

- The resulting representations are not “equivalent”, because one representation can capture some information that the other cannot (i.e., the transformations may lead to loss of information).
- The resulting representations are equivalent, but not identical, and proving the equivalence is non-trivial.

The problem is complicated further by the fact that systems are not usually developed according to a single, consistent idiom and variations may occur at different levels of refinement/abstraction [16].

In the remainder of the paper, we will discuss the specifics of our approach for mapping from a candidate ADL to UML in a manner that maintains traceability and preserves correctness with respect to the original (architectural) model.

3 Mapping from an ADL to the UML

There are several possible strategies for refining ADL models into UML [11]. We briefly summarize them here, and describe how we used specific instances of those strategies.

- Strategy #1 consists of using standard UML constructs to simulate modeling architectural concerns as would be done in an ADL [10].
- Strategy #2 consists of using UML’s built-in extension mechanisms (stereotypes and tagged values) [17] and the Object Constraint Language (OCL) [18] to constrain the semantics of meta-classes to those of ADL constructs.
- Strategy #3 consists of augmenting the UML meta-model to directly support architectural concerns. Although this is a potentially effective approach, it would result in a notation that is incompatible with standard UML. Since one of our goals for this work is conformance with standard UML and corresponding tools, we do not currently pursue this strategy.

We illustrate how we combined strategies #1 and #2 using C2SADEL, an ADL for C2-style architectures [19]. In our previous work [9, 10], we manually mapped several ADLs into UML using the two strategies. We use those mappings as a basis for providing automated support for our current task. Before proceeding with the details of our approach, we briefly summarize the relevant rules of the C2 style and C2SADEL constructs. We also give a brief example of a C2-style architecture used for illustration in the remainder of the paper.

3.1 Overview of C2

We have selected the C2 architectural style as a vehicle for exploring our ideas because it provides a number of useful rules for high-level system composition, demonstrated in numerous applications across several domains [19]; at the same time, the rules of the C2 style are broad enough to render it widely applicable [20,21].

An architecture in the C2 style consists of components, connectors (buses), and their configurations. Each component has two connection points, a “top” and a “bottom.” The top (bottom) of a component can only be attached to the bottom (top) of one bus. It is not possible for components to be attached directly to each other: buses always have to act as intermediaries between them. Furthermore, a component cannot be attached to itself. However, buses can be attached together: in such a case, each bus considers the other as a component with regard to the publication and forwarding of events. Components communicate by exchanging two types of events: service requests to components above and notifications of completed services to components below.

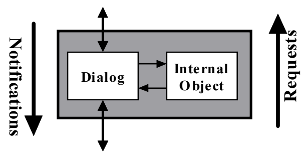

Fig. 3. Internal Architecture of a C2 Component

A C2-style component has a canonical internal architecture, consisting of an object with a defined interface and a dialog, as shown in Fig. 3. The internal object of a component can be arbitrarily complex. It is fully encapsulated inside the component, so that only the dialog can directly invoke the access routines of the object. The dialog, in turn, is in charge of interacting with the rest of the architecture via events.

A unique aspect of C2 buses is their context-reflective property [19]: a C2 bus is not defined to have a specific interface; instead, the “interface” it exports is a function of the interfaces of the components and connectors attached to it. This property of buses is a direct enabler of C2’s support for dynamic adaptation [22].

C2’s accompanying ADL, C2SADEL [12], specifies architectures in three parts: component types, connector types, and topology (or configuration). The topology, in turn, defines component and connector instances for a given system and their interconnections. C2SADEL specifies a component type with an invariant and sets of services a component provides and requires. A service consists of an interface and an operation. A single operation may export multiple interfaces (see the map at the bottom of Fig. 6). Invariants and operations (with their pre- and post-conditions) are specified as first-order logic expressions. A component may be subtyped from another component, using heterogeneous subtyping that preserves the supertype component’s naming, interface, behavior, implementation, or a combination of them [12, 23].

We illustrate these concepts and subsequent discussion with an example. The example architecture is a variant of the logistics system for routing incoming cargo to a set of warehouses, first introduced in [22] and shown in Fig 4. The *DeliveryPort*, *Vehicle,* and *Warehouse* components keep track of the state of a port, a transportation vehicle, and a warehouse, respectively; each of them may be instantiated multiple times in a system. The *DeliveryPortArtist*, *VehicleArtist*, and *WarehouseArtist* components are responsible for graphically depicting the state of their respective components to the end-user. The *Layout Manager* organizes the display based on the actual number of port, vehicle, and warehouse instances. *SystemClock* provides consistent time measurement to interested components, while the *Map* component informs vehicles of routes and distances. The *Router* component determines when cargo arrives at a port, keeps track of available transport vehicles at each port, and tracks the cargo during its delivery to a

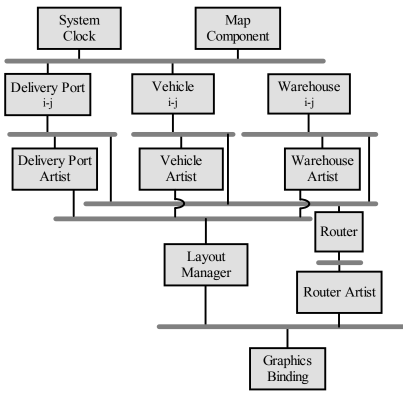

Fig. 4. C2 Architecture of the Cargo Routing System

warehouse. *RouterArtist* allows entry of new cargo as it arrives at a port and informs the *Router* component when the end-user decides to route cargo. The *GraphicsBinding* component renders the drawing information sent from the artists using a graphics toolkit, such as Java's AWT.

An extract of a C2SADEL specification of this architecture is given in Fig. 5[1]. A partial specification of the *DeliveryPort* component is shown in Fig. 6[2]. Note that *DeliveryPort* is a subtype of the more general *CargoRouteEntity* component, which is evolved by preserving both its interface and behavior.

```
architecture CargoRouteSystem is {
 component_types {
    component DeliveryPort is extern {Port.c2;}
    component GraphicsBinding is virtual {}
    ...
 }
 connector_types {
    connector FiltConn is {filter msg_filter;}
    connector RegConn is {filter no_filter;}
 }
 architectural_topology {
    component_instances {
       Runway : DeliveryPort;
       Binding : GraphicsBinding;
       ...
    }
    connector_instances {
       UtilityConn : FiltConn;
       BindingConn : RegConn;
       ...
    }
    connections {
       connector UtilityConn {
          top SimClock, DistanceCalc;
          bottom Runway, Truck;
       }
       connector BindingConn {
          top LayoutArtist, RouteArt;
          bottom Binding;
       }
       ...
    }
 }
}
```

Fig. 5. Cargo Routing System architecture specified in C2SADEL

```
component DeliveryPort is
 subtype CargoRouteEntity (int \and beh) {
    state {
       cargo          : \set Shipment;
       selected          : Integer;
       ...
    }
    invariant {
       (cap >= 0) \and (cap <= max_cap);
    }
    interface {
       prov ip_selshp: Select(sel : Integer);
       req  ir_clktck: ClockTick();
       ...
    }
    operations {
       prov op_selshp: {
          let  num : Integer;
          pre  num <= #cargo;
          post ~selected = num;
       }
       req or_clktck: {
          let  time : STATE_VARIABLE;
          post ~time = time + 1;
       }
       ...
    }
    map {
       ip_selshp -> op_selshp (sel -> num);
       ir_clktck -> or_clktck ();
       ...
    }
}
```

Fig.6. DeliveryPort component type specified in C2SADEL

3.2 Mapping C2SADEL to UML Using Strategy #1

We initially map the internal object of each C2 component to a set of UML classes determined by the state maintained in the C2 component. The designer can then further refine the internal object by adding native UML constructs, such as classes with various associations (e.g., aggregation) and relationships (e.g., generalization), as shown in Fig. 7. State variables of a C2 component's internal object become private attributes of the UML classes representing the internal object; provided operations become public class operations; provided operation pre/post conditions and signatures become pre/post conditions and signatures on the corresponding class operations. Fig. 7 illustrates the internal objects for the *DeliveryPort* and the *DeliveryPortArtist* components.

As our previous work demonstrated [9,10], there are no direct UML counterparts to architectural connectors. For that reason, we map C2 buses to UML interfaces, where

[1]The "extern" and "virtual" keywords are not relevant to this discussion. We omit their explanations in the interest of brevity.
[2]"~" denotes the value of a variable after the operation has been performed, while "#" denotes set cardinality.

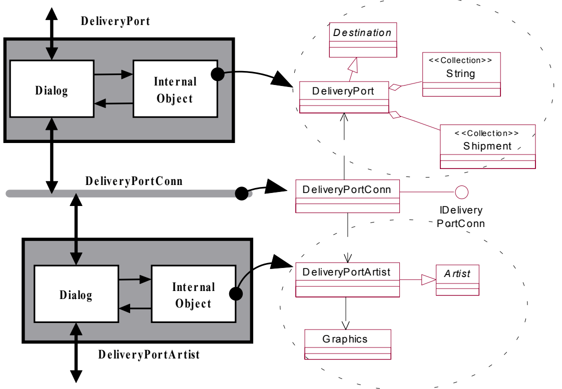

Fig.7. Designing the Internal Object. *Artist*, *Destination*, *ShipmentCollection*, *StringCollection* are not a part of the transformation, but were added manually by the designer.

a UML interface is a collection of operations that are used to specify services provided by a class. To satisfy the context-reflective property of the buses, the operations provided by the interface (shown in Fig. 8) are roughly the union of the provided operations of all components attached to the bus, as discussed in [10]. In Fig. 7, the interface is graphically rendered as a circle, whereas in Fig. 8, it is graphically rendered as a stereotyped <<interface>> class in order to expose its operations.

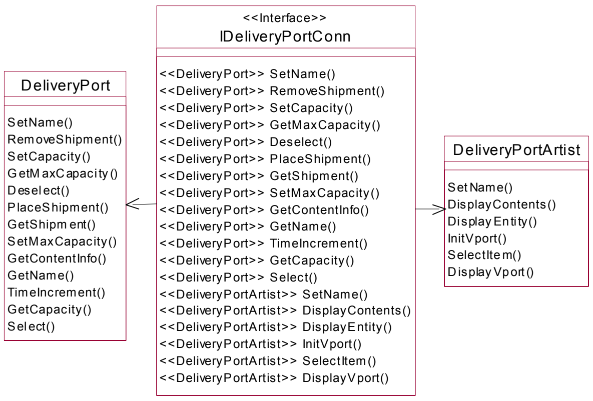
<<Interface>>
IDeliveryPortConn
<<DeliveryPort>> SetName()
<<DeliveryPort>> RemoveShipment()
<<DeliveryPort>> SetCapacity()
<<DeliveryPort>> GetMaxCapacity()
<<DeliveryPort>> Deselect()
<<DeliveryPort>> PlaceShipment()
<<DeliveryPort>> GetShipment()
<<DeliveryPort>> SetMaxCapacity()
<<DeliveryPort>> GetContentInfo()
<<DeliveryPort>> GetName()
<<DeliveryPort>> TimeIncrement()
<<DeliveryPort>> GetCapacity()
<<DeliveryPort>> Select()
<<DeliveryPortArtist>> SetName()
<<DeliveryPortArtist>> DisplayContents()
<<DeliveryPortArtist>> DisplayEntity()
<<DeliveryPortArtist>> InitVport()
<<DeliveryPortArtist>> SelectItem()
<<DeliveryPortArtist>> DisplayVport()
DeliveryPort
SetName()
RemoveShipment()
SetCapacity()
GetMaxCapacity()
Deselect()
PlaceShipment()
GetShipment()
SetMaxCapacity()
GetContentInfo()
GetName()
TimeIncrement()
GetCapacity()
Select()
DeliveryPortArtist
SetName()
DisplayContents()
DisplayEntity()
InitVport()
SelectItem()
DisplayVport()

Fig. 8. Interface for the ***DeliveryPortConn*** connector

Finally, in Fig. 9, we use a UML object diagram (i.e., instance form of a class diagram) to represent the architectural configuration: in this case, object links[3] represent instances of associations. Note that the architecture is currently based on explicit method invocations and completely bypasses the dialogs: *aDeliveryPort* is an instance of the *DeliveryPort* class (in Fig. 7); *DeliveryPortConn* is an instance of a class that realizes the interface *IDeliveryPortConn*, such that each method of *DeliveryPortConn* delegates the received messages to methods of the attached objects (*aDeliveryPort* and *theDeliveryPortArtist*). The object *DeliveryPortConn* performs the combined functionality of the dialogs of the C2 components *DeliveryPort* and *DeliveryPortArtist* and the C2 connector *DeliveryPortConn*, namely routing the requests with only (explicit) method invocations, instead of (implicit) event notifications coupled with dialogs invoking access routines of the internal objects. In summary, this transformation does not include certain style-specific concerns, such as event notification or implicit invocation.

[3]The importance of OCL constraints for preserving correctness becomes apparent in this case: although a C2 component cannot be attached to itself, links to self are legal in UML object interaction diagrams.

3.3 Mapping C2SADEL to UML Using Strategy #2

In this transformation, we generate UML constructs equivalent to architectural constructs using stereotypes. Our approach currently does not use OCL for reasons discussed in Section 4, but we do not foresee difficulties in including OCL constraints as tagged values (or "invisible" properties), as discussed in [9, 11]. Fig. 10 illustrates a fragment of the Cargo Routing System architecture, modeled using stereotypes. For instance, the *DeliveryPortComponent* <<C2-Component>> class has top and bottom interfaces, *ITop-DeliveryPort-Component* and *IBottom-DeliveryPort-Component*, respectively. The intermediate connector is mapped to a <<C2-Connector>> class, *Delivery-PortConn-Connector*. In addition, we generate UML Components, which correspond to modules in UML and can realize a number of classes and interfaces (in that case, the interface of the component is represented by the interfaces it realizes). The UML Component corresponding to a C2 component realizes the <<C2-Component>> class, its top and bottom interfaces, and the classes representing the

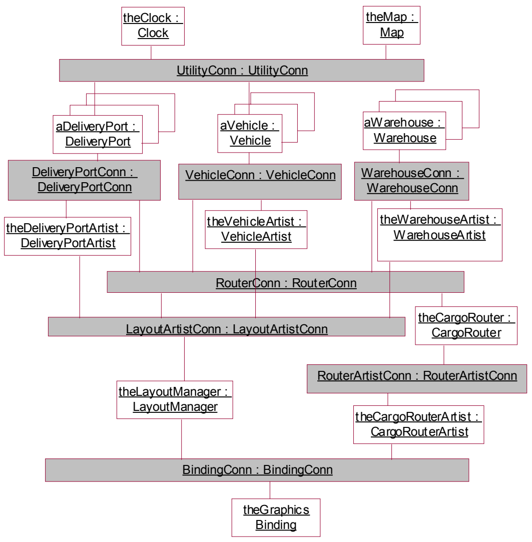
theClock : Clock
theMap : Map
UtilityConn : UtilityConn
aDeliveryPort : DeliveryPort
aVehicle : Vehicle
aWarehouse : Warehouse
DeliveryPortConn : DeliveryPortConn
VehicleConn : VehicleConn
WarehouseConn : WarehouseConn
theDeliveryPortArtist : DeliveryPortArtist
theVehicleArtist : VehicleArtist
theWarehouseArtist : WarehouseArtist
RouterConn : RouterConn
LayoutArtistConn : LayoutArtistConn
theCargoRouter : CargoRouter
RouterArtistConn : RouterArtistConn
theLayoutManager : LayoutManager
theCargoRouterArtist : CargoRouterArtist
BindingConn : BindingConn
theGraphics Binding

Fig. 9. Object diagram representing architecture

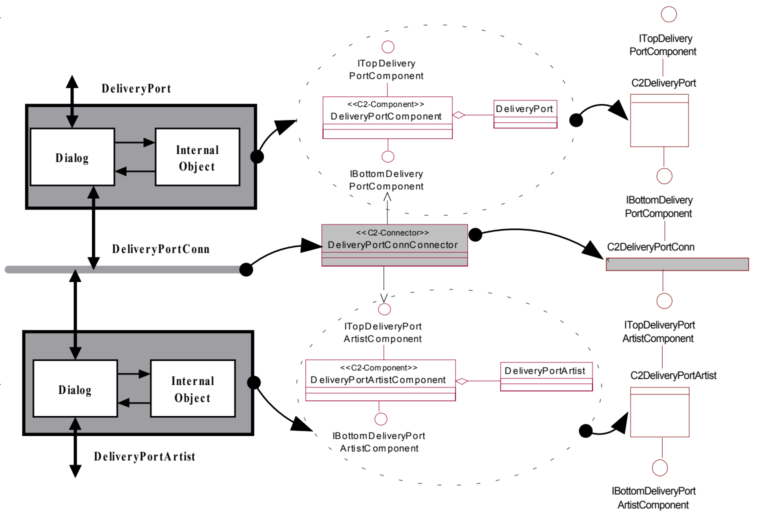
DeliveryPort
Dialog
Internal Object
DeliveryPortConn
Dialog
Internal Object
DeliveryPortArtist
ITopDelivery PortComponent
<<C2-Component>> DeliveryPortComponent
DeliveryPort
IBottomDelivery PortComponent
<<C2-Connector>> DeliveryPortConnConnector
ITopDeliveryPort ArtistComponent
<<C2-Component>> DeliveryPortArtistComponent
DeliveryPortArtist
IBottomDeliveryPort ArtistComponent
ITopDelivery PortComponent
C2DeliveryPort
IBottomDelivery PortComponent
C2DeliveryPortConn
ITopDeliveryPort ArtistComponent
C2DeliveryPortArtist
IBottomDeliveryPort ArtistComponent

Fig. 10. Representing Architectural Constructs in UML. Although not shown in the figure, each C2 component will also have to realize the classes added by the designer to represent the internal object

internal object discussed in Section 3.2. For example, *C2DeliveryPort* realizes *DeliveryPortComponent,ITopDeliveryPortComponent,IBottomDeliveryPortComponent* and *DeliveryPort*. In order to satisfy the context-reflective property, the UML Component for a C2 connector realizes the <<C2-Connector>> class, as well as the bottom interfaces of Components/Connectors above, and the top interfaces of all Components/Connectors below. We also use a UML component diagram to represent the architecture: a UML dependency relationship between two UML components represents an attachment of a C2 component (or connector) with another connector.

3.4 Transformation Rules

As described above in the context of an example, the transformation from C2SADEL to UML is defined by a set of rules. The internal object is transformed using the rules shown in Table 1, while the architectural style concerns are addressed by the rules shown in Table 2. Note that, in the case of C2, it is not possible to represent in UML the architectural concerns without representing the internal objects first. The most important requirement of the transformation rules is that the generated UML design model be initially correct by construction with respect to the rules of the architectural style. However, as the design is modified, the problems discussed in Section 2 may arise. Detecting and resolving those inconsistencies is critical, but has been outside the scope of our work to date.

Table 1. Transformation Rules for the Internal Object

- Internal Object ✐ Class
  - State Variable ✐ Class Private Attribute
  - Component Invariant ✐ Tagged Value + Class Documentation
  - Provided Operation ✐ Class Operation
  - Required Operation ✐ Class Documentation
  - Operation Pre/Post Condition ✐ Pre/Post Condition on Class Operation
  - Message Return Type ✐ Return Type on Class Operation
  - Message Parameter ✐ Parameter (Name + Type) on Class Operation
- Connector ✐ Interface (<<Interface>> Class)
  - Connector Interface ✐ Union of Operations of attached Objects/Interfaces
    - Message Originator ✐ Operation <<Stereotype>>
- Architecture Configuration (explicit invocation) ✐ (Object) Collaboration Diagram
  - Component Instance ✐ Internal Object Class Instance
    - Connector Instance ✐ <<Interface>> Class Instance
    - Component/Connector Binding ✐ Object Link (instance of an association)

Table 2. Transformation Rules for Architectural Constructs

- Component ✐ <<C2-Component>> Class
  - Internal Object ✐ <<C2-Component>> Class Attribute
  - Component Top Interface ✐ <<Interface>> Class
  - Component Bottom Interface ✐ <<Interface>> Class
  - Outgoing Request ✐ <<Interface>> Class <<out>> Operation
  - Incoming Notification ✐ <<Interface>> Class <<in>> Operation
- Connector ✐ <<C2-Connector>> Class
  - Connector Top Interface ✐ Union of Bottom Interfaces of attached Components/Connectors
  - Connector Bottom Interface ✐ Union of Top Interfaces of attached Components/Connectors
- Architecture Configuration (implicit invocation + event notification) ✐ Component Diagram
    - Component Instance ✐ Component realizing…
    - Connector Instance ✐ Component realizing…

3.5 Limitations of UML to Represent Software Architectures

Standard UML constructs and its built-in extensions are well suited for strategies #1 and #2. However, as mentioned above, UML may not be able to express all the information represented in an ADL [11]. In this particular case, UML's support for subtyping may not be able to adequately express the heterogeneous component subtyping mechanisms provided in C2SADEL, where different aspects of a component are preserved (e.g., interface, behavior, implementation, or a combination of them) [12, 23]. UML has been strongly influenced by object-oriented programming languages that typically support only a subset of possible subtyping/subclassing relationships: for example, in C++, inheritance means both interface and implementation inheritance [24]; this also appears to be the case in UML.

While using standard UML constructs for representing architectural concerns, we tried to maintain a parallel between type (instance) forms in the ADL and the type (instance) forms in UML. Thus, a class instance (i.e., an object) should correspond to a component instance. Similarly, an object type (i.e., a class) should correspond to a component type. However, the mapping is not obvious when dealing with connectors. C2 connectors defined as interfaces (equivalent to abstract classes, i.e., type form) cannot be inherently defined because they do not have interfaces of their own. Instead, we need to know the architectural configuration (instance), and specifically the instances of components and connectors attached to a given connector, to be able to define connector interfaces.

4 Integrating Rational Rose® with DRADEL

To integrate architecture-based development with mainstream approaches, one will need to bridge the gap between the tool support for architecture-based approaches and the support for modeling and design. To that end, we integrated DRADEL, a tool for modeling, analysis, and evolution of architectures described in C2SADEL, with Rational Rose®, a commercial tool that supports UML-based software modeling and development. The integrated environment[4] enables the designer to generate a UML representation of an architecture. The designer can then use the expressive power of UML to iteratively refine the resulting UML model and eventually produce an implementation using the code generation capabilities of Rose.

4.1 Architecture of the Rational Rose® Environment

Rose provides support for UML and its built-in extension mechanisms, such as stereotypes and tagged values (available as properties in Rose). Rose also provides code generation and reverse engineering capabilities for several programming languages [25, 26]. Rose itself is an extensible tool [27]: the Rose Extensibility Interface (REI) provides read/write access to the model elements (packages, classes, attributes, operations, stereotypes, etc.), their properties, the diagrams (graphical properties of model elements), and to the application itself (to execute scripts, set the visibility of the main application window, etc.).

[4]Note that the UML diagrams shown above as examples were generated using the environment. In some cases, Rational Rose is not fully compliant with the UML standard.

In addition, Rose also makes the REI Automation Objects (model elements, properties, diagrams, and application) available to external applications using the Microsoft Component Object Model (COM) [28]. For our purposes, we are using Rose as an Automation Server and an external application (developed in Microsoft Visual J++) as the Automation Controller.

4.2 Architecture of the DRADEL Environment

DRADEL [12] is a prototype environment for architecture modeling in C2SADEL, analysis of internal architectural consistency, topological constraint checking, type conformance checking among interacting components, and generation of application skeletons using the C2 implementation infrastructure. DRADEL itself is designed in the C2 style (see the left side of Fig. 11) and implemented using the C2 implementation infrastructure [29].

4.3 Integration Approach

Integrating Rose with DRADEL presented a challenge: DRADEL is implemented in Java, whereas Rose is COTS software, with no available source code. We solved this problem by porting DRADEL to Microsoft Visual J++ 6.0 and by generating Java wrappers for the Rose Automation objects [30]. Porting DRADEL to Visual J++ did not require modifications to the DRADEL code, since it uses standard Java syntax. The integration itself required a new C2 component in the DRADEL architecture, *UMLGenerator*, and some modifications for the user interface component, *UserPalette* (Fig. 11). The *UMLGenerator* component traverses the architectural representation maintained by DRADEL, initializes the Rose application, and creates Rose model elements and diagrams, based on the transformation rules described in Section 3.

The advantage of the approach is that DRADEL is still independently extensible: one can exploit the C2 style to add new components to DRADEL without any effect on its interaction with Rose. Similarly, Rose is still extensible independently of this application: in particular, it can still act as an Automation Controller for an external Automation Server (such as Microsoft Word), e.g., to generate a formatted report of the model.

There are several potential disadvantages of the approach we adopted. First, the *UMLGenerator* component communicates with Rose using synchronous remote procedure calls, mandated by COM, unlike the rest of the C2-style architecture, where communication is based on asynchronous events and implicit invocation. In our future work, we plan to address this problem using the technique described in [31]. Second, the integration mechanism is platform dependent, because the Java wrappers to the Rose COM objects require extensions specific to the Microsoft Java Virtual Machine. Finally, the most important limitation of our current integration of Rose and DRADEL is that Rose cannot generate notifications reflecting changes in the internal state (e.g., *ClassAdded*) in response to requests (e.g., *AddClass*). The reason is that Rose is not a Connectable Object [32]. Had that been the case, we could have packaged Rose as an internal object of the *UMLGenerator* C2 component, and enabled full two-way communication with DRADEL.

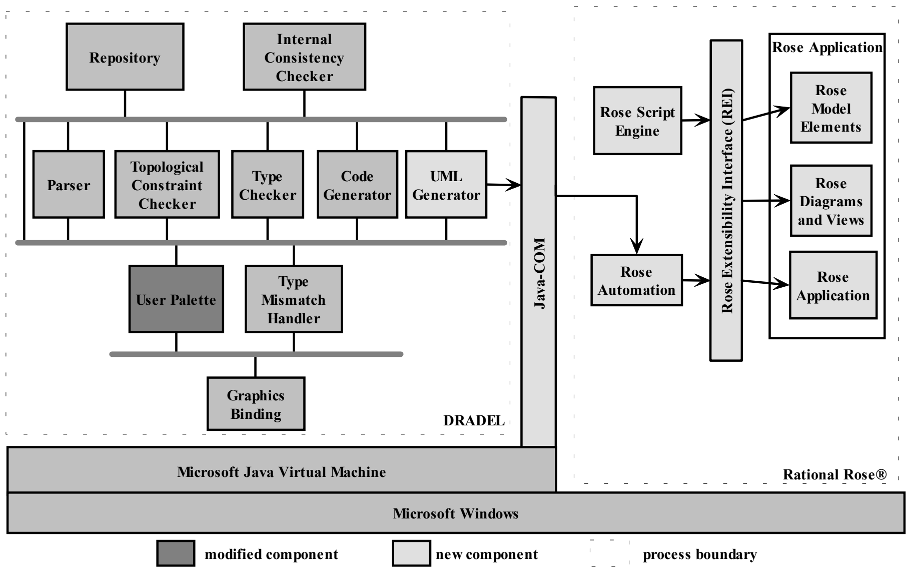

Fig. 11. Architecture of the integrated environment**.** *DRADEL and Rose are running in separate processes and communicating through synchronous COM remote procedure calls*

5 Related Work

Our work has been influenced by a large body of research and practical experience. In the interest of brevity, we only compare it to the most relevant approaches. Since this project builds on our previously published results, we do not focus on its relation to other work (e.g., [33, 34]), described elsewhere [9, 11].

Our approach to software architectures has emerged from the part of the research community that focuses on specifying structural and possibly behavioral aspects of a software system centered around a (formal) ADL. Another part of the community has tried to identify useful architectural perspectives, or *views*. Two representative examples are provided by Kruchten [35] and Hofmeister et al. [17, 36]. Although these approaches are in certain ways more comprehensive than ADL-centered approaches, our technique for refining architectures into UML models is applicable to them as well. Indeed, Hofmeister et al. demonstrate how UML can be constrained to model their four architectural views (conceptual, module, execution, and code). However, their technique is currently entirely manual.

As already discussed, architecture researchers have largely ignored the problem of refining architectures into designs and/or implementations. One exception is the approach proposed by Moriconi et al. [37], which incrementally transforms an architecture across levels of abstraction using a series of refinement maps. The maps must satisfy a correctness-preserving criterion, which mandates that all decisions made at a given level be maintained at all subsequent levels and that no new decisions be introduced. We believe this to be overly stringent. It sacrifices design flexibility to a notion of (absolute) correctness. The role of the human designer is virtually eliminated. Finally, formally proving the relative correctness of architectures at different refinement levels may prove impractical for large architectures and large numbers of levels. Such an approach can be of value, however, if only applied to the most critical parts of a system, and complemented by a more pragmatic technique, such as the one proposed in this paper.

6 Conclusions and Future Research Directions

Software architectures provide a promising basis for improving the state-of-the-art in software development. However, no improvement can be achieved simply by focusing solely on architectures, just like a new programming language cannot by itself solve the problems of software engineering. A programming language is only a tool that allows (but does not force) developers to put sound software engineering techniques into practice. Similarly, one can think of software architectures and ADLs as tools that also must be supported with specific techniques to achieve desired properties. Additionally, ensuring system properties at the level of architecture is of little value unless it can also be ensured that those properties will be preserved in the resulting implementation.

In this paper, we have presented a practical approach for transferring architecture-level decisions to the design and, subsequently, the implementation. The specific details of the approach, outlined in the transformation rules in Tables 1 and 2, are likely to change as our understanding of the relationship between UML and software architectures evolves, as support for refining additional ADLs into UML is added, and as UML itself evolves. However, we believe that the approach can be easily adapted to accommodate any such changes.

Our future work will exploit the integrated environment we have produced to gain additional insights along several dimensions, including:

- transformation of architectural elements that do not have direct design counterparts (e.g., connectors) into UML;
- analysis of a design represented in UML for conformance to a given architectural style. To this end, we plan to abstract architectural information from Rose and analyze it in DRADEL. We are currently planning to use Rose Architect [38] to abstract the architecture from the design, and Rose's reverse engineering capabilities to abstract the design from the implementation;
- enforcement of style rules, by attaching OCL constraints to the relevant UML model elements and integrating tool support to enforce the constraints [39]; and
- expansion of the current support for modeling architectural behavior to include dynamic behavior, e.g., by using UML statechart diagrams.

References

1 Garlan, D., Paulisch, F.N., Tichy, W.F., editors: *Summary of the Dagstuhl Workshop on Software Architecture*, February 1995

2 Garlan, D., editor: *Proceedings of the First International Workshop on Architectures for Software Systems (ISAW-1)*, Seattle, WA, April 1995

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

4 Magee, J., and Perry, D.E., editors: *Proceedings of the Third International Software Architecture Workshop (ISAW-3)*, Orlando, FL, November 1998

5 Donohoe, P., editor: *Proceedings of the First Working IFIP Conference on Software Architecture (WICSA1)*, San Antonio, TX, February 1999

6 Medvidovic, N., Taylor, R.N.: A Classification and Comparison Framework for Software Architecture Description Languages. In *IEEE Transactions on Software Engineering*, to appear

7 Booch, G., Jacobson, I., Rumbaugh, J.: *The Unified Modeling Language User Guide*, Addison-Wesley, 1998

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

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

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

11 Medvidovic, N., Rosenblum, D.S., Robbins, J.E., Redmiles, D.F.: Modeling Software Architectures in the Unified Modeling Language. In submission

12 Medvidovic, N., Rosenblum, D.S., Taylor, R.N.: A Language and Environment for Architecture-Based Software Development and Evolution. In *Proceedings of the 21st International Conference on Software Engineering (ICSE'99)*, Los Angeles, CA, May 16-22, 1999

13 Rational Software Corporation, *Rational Rose 98: Using Rational Rose*

14 Shaw, M.: Procedure Calls are the Assembly Language of Software Interconnections: Connectors Deserve First-Class Status. *Workshop on Studies of Software Design*, 1993

15 Harris, D.R., Reubenstein, H.B., Yeh, A.S.: Reverse Engineering to the Architectural Level, In *Proceedings of the 17th International Conference on Software Engineering (ICSE'95)*, Seattle, Washington, 1995

16 Garlan, D., Shaw, M.: An Introduction to Software Architecture, *Advances in Software Engineering*, vol. 1, World Scientific Publishing Company, 1993

17 Hofmeister, C., Nord, R.L., and Soni, D.: Describing Software Architecture with UML. In *Proceedings of the First IFIP Working Conference on Software Architecture (WICSA1)*, San Antonio, TX, February 1999

18 Warmer, J.B., Kleppe, A.G.: *The Object Constraint Language: Precise Modeling With UML*, Addison-Wesley, 1999

19 Taylor, R.N., Medvidovic, N., Anderson, K.M., Whitehead, E.J. Jr., Robbins, J.E., Nies, K.A., Oreizy, P., Dubrow, D.L.: A Component- and Message-Based Architectural Style for GUI Software. *IEEE Transactions on Software Engineering*, vol. 22, no. 6, pp. 390-406, June 1996

20 Di Nitto, E., and Rosenblum, D.S.: Exploiting ADLs to Specify Architectural Styles Induced by Middleware Infrastructures. In *Proceedings of the 21st International Conference on Software Engineering*, pp. 13-22, Los Angeles, CA, May 1999

21 Yakimovich, D., Bieman, J.M., and Basili, V.R.: Software Architecture Classification for Estimating the Cost of COTS Integration. In *Proceedings of the 21st International Conference on Software Engineering*, pp. 296-302, Los Angeles, CA, May 1999

22 Oreizy, P., Medvidovic, N., Taylor, R.N.: Architecture-Based Runtime Software Evolution. In *Proceedings of the 20th International Conference on Software Engineering (ICSE'98*), pp. 177-186, Kyoto, Japan, April 1998

23 Medvidovic, N., Oreizy, P., Robbins, J.E., Taylor, R.N.: Using Object-Oriented Typing to Support Architectural Design in the C2 Style. In *Proceedings of the Fourth ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE4)*, pp. 24-32, San Francisco, CA, October 16-18, 1996

24 Gamma, E., Helm, R., Johnson,R., Vlissides, J.: *Design Patterns: Elements of Reusable Object-Oriented Software,* Addison-Wesley, 1994

25 Rational Software Corporation: *Rational Rose 98: Roundtrip Engineering with Java*, 1998

26 Rational Software Corporation: *Rational Rose 98: Roundtrip Engineering with C++*, 1998

27 Rational Software Corporation: *Rational Rose 98 Extensibility User Guide*, 1998

28 Williams, S., Kindel, C.: The Component Object Model, *Dr. Dobb's Journal*, December 1994

29 Medvidovic, N., Oreizy, P., Taylor, R.N.: Reuse of Off-the-Shelf Components in C2-Style Architectures. In *Proceedings of the 1997 International Conference on Software Engineering (ICSE'97)*, Boston, MA, May 1997
30 Verbowski, C.: Integrating Java and COM, Microsoft Corporation, January 1999. http://www.microsoft.com/java/resource/java_com.htm
31 Dashofy, E.M., Medvidovic, N., Taylor, R.N.: Using Off-The-Shelf Middleware to Implement Connectors in Distributed Software Architectures, In *Proceedings of the 21st International Conference on Software Engineering (ICSE'99)*, Los Angeles, CA, May 1999
32 Brockschmidt, K.: *Inside OLE*, Second Edition, Microsoft Press, 1995
33 Garlan, D., Monroe, R., Wile, D.: ACME: An Architecture Description Interchange Language. *CASCON'97*, November 1997
34 Wang, E.Y., Richter, H.A., Cheng, B.H.C.: Formalizing and Integrating the Dynamic Model within OMT. In *Proceedings of the 19th International Conference on Software Engineering*, Boston, MA, May 1997
35 Kruchten, P.B.: The 4+1 view model of architecture, *IEEE Software,* Nov. 1995. pp. 42-50
36 Soni, D., Nord, R.L., and Hofmeister, C.: Software Architecture in Industrial Applications. In *Proceedings of the 17th International Conference on Software Engineering (ICSE'95)*, Seattle, WA, 1995
37 Moriconi, M., Qian, X., Riemenschneider, R.A.: Correct Architecture Refinement. In *IEEE Transactions on Software Engineering*, April 1995
38 Egyed, A., Kruchten, P.: Rose/Architect: A Tool to Visualize Architecture. In *Proceedings of the 32nd Hawaii International Conference on System Sciences (HICSS-32)*, January 1999
39 IBM Corporation, The Object Constraint Language: (OCL): the expression language for the UML http://www.software.ibm.com/ad/standards/ocl.html