Published in Proceedings of the 2nd International Conference on the Unified Modeling Language (UML), Fort Collins, CO, October 1999, pp. 2-16

Extending Architectural Representation in UML with View Integration

Alexander Egyed and Nenad Medvidovic

Center for Software Engineering
University of Southern California
Los Angeles, CA 90089-0781, USA
{aegyed,neno}@sunset.usc.edu

Abstract. UML has established itself as the leading OO analysis and design methodology. Recently, it has also been increasingly used as a foundation for representing numerous (diagrammatic) views that are outside the standardized set of UML views. An example are architecture description languages. The main advantages of representing other types of views in UML are 1) a common data model and 2) a common set of tools that can be used to manipulate that model. However, attempts at representing additional views in UML usually fall short of their full integration with existing views. Integration extends representation by also describing interactions among multiple views, thus capturing the inter-view relationships. Those inter-view relationships are essential to enable automated identification of consistency and conformance mismatches. This work describes a view integration framework and demonstrates how an architecture description language, which was previously only represented in UML, can now be fully integrated into UML.

1 Introduction

Software systems are characterized by unprecedented complexity. One effective means of dealing with that complexity is to consider a system from a particular perspective, or view. Views enable software developers to reduce the amount of information they have to deal with at any given time. It has been recognized that "it is not the number of details, as such, that contributes to complexity, but the number of details of which we have to be aware at the same time." [1].

A major drawback of describing systems as collections of views is that the software development process tends to become rather view centric (seeing views instead of the big picture). Such a view centric approach exhibits a fair amount of redundancy across different views as a side effect. That redundancy is the cause for inter-view mismatches, such as inconsistencies or incompletenesses. On top of that, views are used independently, concurrently, are subjected to different audiences (interpretations) and the manner in which model information is shared is extremely inconsistent. All this implies that information about a system must be captured multiple times and must be kept consistent.

To deal with this problem, a major emphasis needs to be placed on mismatch identification and reconciliation within and among views (view integration). We design

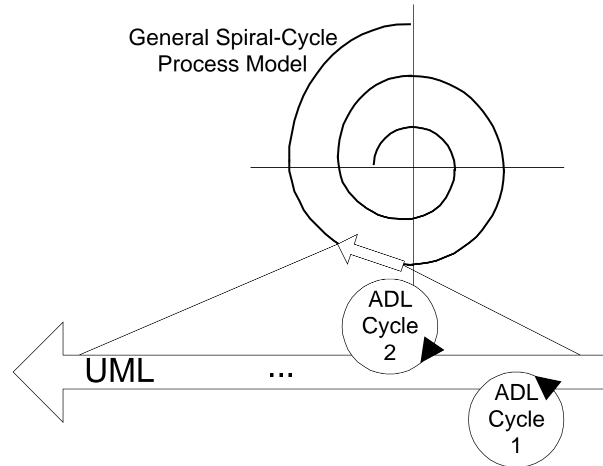

Figure 1. Substituting UML (general-purpose) with ADLs (specific)

not only because we want to build (compose) but also because we want to understand. Thus, a major focus of software development is to analyze and verify the conceptual integrity, consistency, and completeness of the model of a system.

There are numerous reasons for the lack of automated assistance in identifying view mismatches. We believe that one of the major reasons is improper integration of views on a meta-level: although both the notation and semantics of individual views may be very well defined, meta-models that integrate these different views are often inadequate or missing. The Unified Modeling Language (UML) [2] is a good example of this. UML defines a set of views (such as class diagrams, sequence diagrams, and state diagrams) and defines a meta-model for those views. However, the UML meta-model was primarily designed to deal with the issue of capturing and representing modeling elements of views in a common data model (repository).

Another problem with UML is that its views had to be designed to be generally understandable and they are therefore rather simple. The result is that UML view semantics are not well defined so as not to over-constrain the language or limit its usability. Therefore, UML becomes less suitable in domains were more precision (performance, reliability, etc.) is required. One way of addressing this deficiency is the use of additional views. Figure 1 shows an example on how a general purpose development methodology such as UML may be used together with more (domain) specific description languages such as architecture description languages (ADLs). What we mean by architecture is a course grain description of a system with high-level components, connectors, and their configuration. A design further refines the architecture by elaborating on the details of individual components/connectors as well as their interactions. In Figure 1, UML serves as a general development model, whereas more specific views can be generated by taking excursions off the main process to investigate specific concerns, e.g., deadlock detection among modeling elements. Although some UML views support behavioral design, these are inadequate

in automatically detecting potential deadlocks. More comprehensive behavior models and views can therefore be used to augment UML.

The challenge of *integrating* UML becomes more than just integrating existing UML views but also integrating additional views that may be used during the development life-cycle. The ultimate goal of view integration is to provide automatic assistance in identifying view mismatches. Although ensuring the conceptual integrity of models/views may not be fully automatable, there are various types of mismatches that can be identified and even resolved in an automated or semi-automated fashion.

2 Motivation for View Integration in UML

A number of options have recently been proposed to represent certain architectural concerns in UML [3,4,5]. However to date none of them have explored the possibility of ensuring the consistency of these new views with existing ones. We refer to this issue of ensuring consistency among different views as view integration (as opposed to mere view representation). A contribution of this paper is a framework and a set of techniques for integrating existing views with newly introduced architectural views.

This section explains in a bit more detail the existing support in UML for view representation and its current limitations with regard to view integration.

2.1 Architectural Representation in UML

In UML, a number of views are captured and sufficiently represented through the use of the UML meta-model. UML views capture both structural and behavioral aspects of software development. Structural views make use of classes, packages, use cases, and so forth. Behavioral views are represented through scenarios, states, and activities.

Furthermore, UML views may be general or instantiated. Generalized views capture model information and configuration that are true during the entire life time of a model. Instantiated views, on the other hand, depict examples or possible scenarios which usually only describe a subset of the interactions a model element goes through during its life.

Although the UML meta-model could be extended to capture additional non-UML views (such as ADLs), this approach would result in a model that would become UML incompatible. UML, however, supports the need of refinement and augmentation of its specifications through three built-in extension mechanisms:

- *Constraints* place semantic restrictions on particular design elements. UML uses the Object Constraint Language (OCL) to define constraints [2].
- *Tagged values* allow new attributes to be added to particular elements of the model.
- *Stereotypes* allow groups of constraints and tagged values to be given descriptive names and applied to other model elements; the semantic effect is as if the constraints and tagged values were applied directly to those elements.

Using above mechanisms enables us to represent new concepts in UML. For instance, we could choose to incorporate Entity-Relationship models expressed via stereotyped and constrained class diagrams. We could also start representing architectural description languages (e.g. C2 [6], Wright [7], Rapide [8]) within the UML framework. The advantages of doing so are several:

- *Common Model Representation*: Modeling information of different types of views (UML and non-UML) can be physically stored in the same repository. This eliminates problems associated with distributed development information (e.g., access, loss) and their interpretation (e.g., exotic data format).
- *Reduced Toolset for Model Manipulation*: Being able to use UML elements to represent non-UML artifacts enables us to use existing UML toolsets to create those views. For instance, one diagram drawing tool can be used on different types of views.
- *Unified Way of Cross-Referencing Model Information*: Having modeling information stored at one physical location further enables us to cross-reference that information. Cross-referencing is useful for maintaining the traceability among development elements.

2.2 Deficiencies of pure View Representation

The UML extension mechanism discussed above is adequate for representing numerous diagrammatic views. The only major limitation is that existing UML modeling elements (such as classes, objects, activities, states, or packages) must be used (stereotyped and constrained) to represent new concepts. This becomes a particular problem when external concepts cannot be represented by what UML provides.

A more severe shortfall of view representation is that it comes up short in fully integrating views with each other: Although UML and its meta-model define notational and semantic aspects of individual views in detail, inter-view relationships are not captured in sufficient detail. Without these information, the (UML) model is nothing more than a collection of loosely coupled (or completely unrelated) views. Figure 2 illustrates this by showing UML views as being separate entities within a common environment (UML meta-model). Although, some views are weakly integrated (e.g. class and sequence diagrams), in general, UML views are independent.

Figure 2 shows, the lack of view integration extends beyond existing UML views to non-UML views represented in UML (e.g. ADLs). For instance, if a system is specified in some architectural fashion then its realization (in the design and implementation) must adhere to the constraints imposed by that architecture. UML view representation only limits how information can be described in UML, but does not concern itself whether that information is consistent with other parts of the model. View representation alone would allow creation of multiple views, each of which would correctly conform to its specifications; however, their combination would not build a coherent unit. We therefore speak of view integration as an extension to view representation to ensure the conceptual integrity (consistency and completeness) of the entire model *across* the boundaries of individual views.

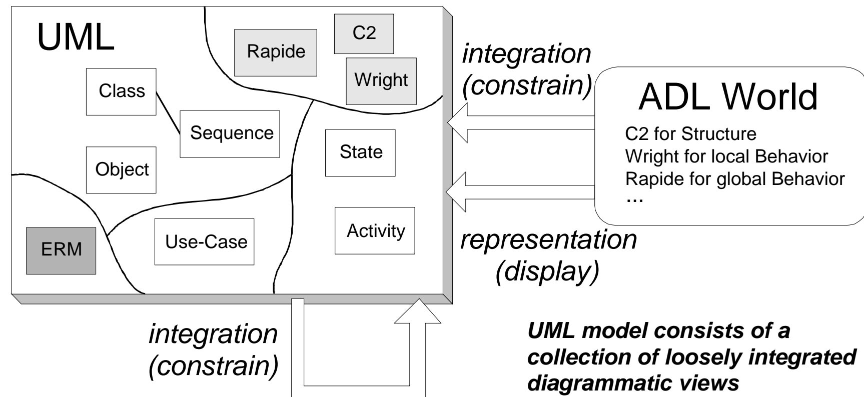

Figure 2: Views and ADLs represented in UML

2.3 Outline

The remainder of this paper will be organized as follows. First we present a simple example of the difference between view representation and view integration. We then introduce a view integration framework and its corresponding activities to describe the problem of integration. To illustrate this framework, we will describe the representation and subsequent integration of an architecture style, C2, in more detail. We conclude the paper by summarizing the key issues and solutions.

3 Example: Layered Architecture constrains UML Design

Before we demonstrate how to represent and integrate a more complex architectural style, C2, we highlight the differences between representation and integration using a simpler, well-understood, layered architectural style.

"*The layered architectural* [style] *helps to structure applications that can be decomposed into groups of subtasks in which each group of subtasks is at a particular level of abstraction*." [9] The layered style defines which part of a system are allowed to interact and which are not. For instance, assume that we have a trivial layered system with four layers: (1) User Interface, (2) Application Framework, (3) Network and (4) Database. The layered style defines that components within layer 1 (User Interface) may talk to other components in layer 1 as well as to components that are part of layer 2. Similarly, layer 2 components may interact among themselves, as well as with layer 1 and layer 3 components. The layered architecture, however, disallows a user interface component (layer 1) to talk directly to, say, the database in layer 4 without going through the intermediate layers 2 and 3.

In order to make use of the layered architectural style in UML, we need to represent layers in UML. An easy way of doing this is by using stereotyped UML packages. For instance, we may create a *User Interface* package with the stereotype *layer*

1, an *Application Framework* package with the stereotype *layer 2* and so forth. Next, we may use OCL to constrain the ways in which the layers (stereotyped packages) may interact. Thus, with OCL we could specify that layer 1 may depend on layer 2, layer 2 may depend on layer 3 and so forth. Note that we need not specify that layer 1 is allowed to talk to itself because that *knowledge* is already implicit in packages. Having specified how to represent layers, a UML design tool may now be used to create layered architectural diagrams.

Thus, we now have the means of *representing* an layered architectural view in UML but nothing more. Fully integrating the layered style into UML also requires that the realization of a system (i.e., its design and subsequent implementation) still conforms to the architectural style rules. Both design and implementation will make use of different types of views and, thus, we need to ensure that both are still consistent with the constraints imposed by the architecture.

For instance, if we design our system using UML class or sequence diagram(s) then the way those classes may interact is limited by both the notation of UML class diagrams and the layered style. The former constraint is usually supported by UML design tools (e.g. Rational Rose). However, the latter cannot yet be supported since we did not yet specify that relationship. To do this we need to ensure that classes are always associated with layers and that calling dependencies between those classes correspond to calling dependencies of associated layers. There are basically two types of mismatches that may happen at this stage:

- If the architecture defines some layers for which there are no associated classes in the design, then this may indicate a potential conformance mismatch.
- If the design contains a class dependency that contradicts the layer dependency in the architecture, then this may indicate a potential consistency mismatch.

4 UML Integration

To address the view mismatch problem, we have investigated ways of describing and identifying the causes of architectural mismatches across UML views. To this end, we have devised and applied a view integration framework, accompanied by a set of activities and techniques for identifying mismatches in an automated fashion; this framework is depicted in Figure 3 and described below.

The system model represents the model repository (e.g. UML model) of the designed software system. Software developers use views to add new information to the system model and to modify existing information (view synthesis). The view analysis activity interacts with the system model and the view synthesis so that new information can be validated against the existing information in the model to ensure their conceptual integrity.

This approach exploits redundancy between views. For instance, view A contains information about view B; this information can be seen as a constraint on B. The view integration framework is used to verify those constraints and, thereby, the consistency across views. Since there is more to view integration than constraints and consistency rules, our view integration framework also provides an environment where we can apply those rules in a meaningful way. Therefore, as already discussed, we see view

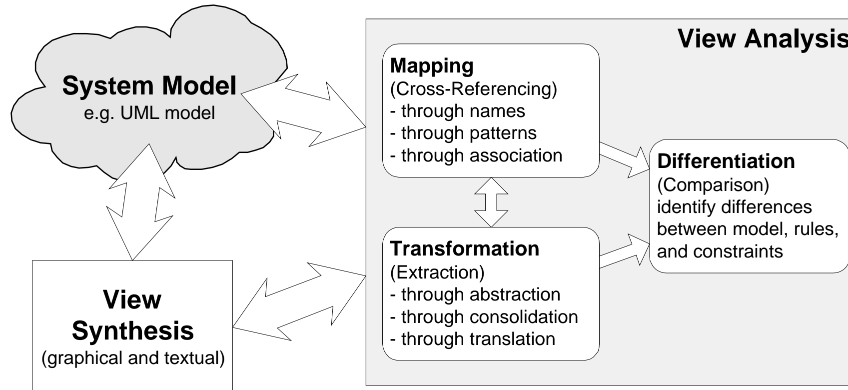

Figure 3. View Integration Framework and Activities

integration as an extension to view representation. The former extends the latter not only by rules and constraints but also by defining *what* information can be exchanged and *how* it can be exchanged. Only after the *what* and *how* have been established, can inconsistencies be identified and resolved automatically.

- Mapping: Identifies related pieces of information and thereby describes *what* information is overlapping.
- Transformation: Extracts and manipulates model elements of views in such a manner that they can be interpreted and used by other views (*how* to address information exchange).
- Differentiation: Traverses the model to identify (potential) mismatches within its elements. Mismatch identification rules can frequently be complemented by mismatch resolution rules.

It is out of the scope of this paper to deal with automated mapping and transformation in detail. Automation techniques for both are described in [10] and [11]. We will primarily focus on *Differentiation* in this paper. To illustrate the behavioral integration of an ADL later on, we will also demonstrate one *Transformation* technique.

5 C2 and UML

Section 3 highlighted the difference between view representation and integration in the case of the layered architectural style. However, this example fell short of conveying in detail how the two are actually accomplished. This section will complement that discussion by showing how the C2 architectural style [6] can be represented and then integrated into UML using the approach we propose.

5.1 Overview of C2

C2 is an architectural style intended for highly distributed software systems [6]. In a C2-style architecture, *connectors* (buses) 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 sets of messages that may be sent and received. Inter-component messages are either *requests* for a component to perform an operation, or *notifications* that a given component has performed an operation or changed state.

A C2 component consists of two main internal parts. An *internal object* stores state and implements the operations that the component provides, while a *dialog specification* maps from messages received to operations on the internal object and from results of those operations to outgoing messages. Two components’ dialogs may not directly exchange messages; they may only do so via connectors. Each component may be attached to at most one connector at the top and one at the bottom. A connector may be attached to any number of other components and connectors. Request messages may only be sent “upward” through the architecture, and notification messages may only be sent “downward.”

The C2 style further demands that components communicate with each other only through message-passing, never through shared memory. Also, C2 requires that notifications sent from a component correspond to the operations of its internal object, rather than the needs of any components that receive those notifications. This constraint on notifications helps to ensure *substrate independence*, which is the ability to reuse a C2 component in architectures with differing substrate components (e.g., different window systems).

Left side of Figure 4 shows an example C2-style architecture. This system consists of four components and two connectors. One component is a *database manager*. Interacting with the *database manager* are the *database administrator*, who has direct access to the database, and the *transaction manager*, who uses the database either via

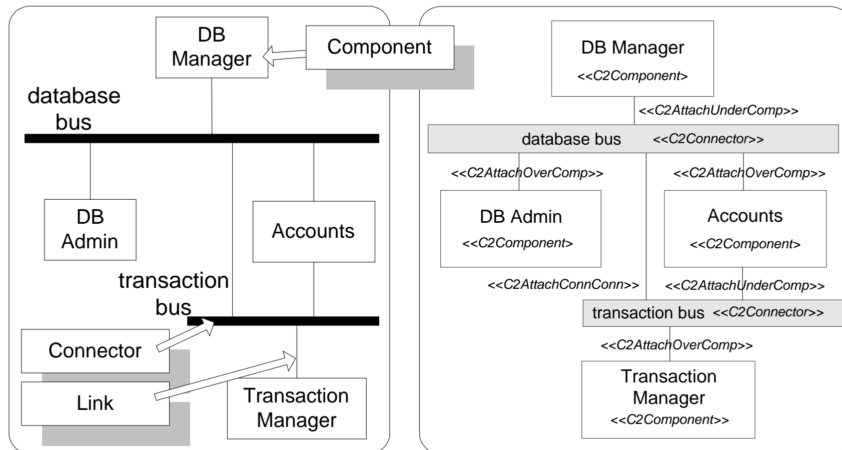

Figure 4. Simple C2 Architecture Example and its UML representation

the *Account* component (handles the transaction on a given account) or directly via the connector to connector link. This C2 diagram roughly corresponds to a layered system where the top component is the data source, the bottom components constitute the user interface, and the middle component shows the mitigating application layer. The right side of Figure 4 shows how this same C2 system can be represented in UML using a UML class diagram as a template. We chose not to modify the UML meta-model in order to stay consistent with the UML definition. Instead we adapted existing UML model elements to represent new concepts and used the UML extensibility mechanism to be able to distinguish between them. The following section will elaborate on that.

5.2 C2 Representation

In our previous work, we have begun exploring the issues in representing ADLs in UML, both using existing UML diagrams [12] and extending them via stereotypes [3]. In this section, we represent the key aspects of C2 in UML. This section is also intended to sensitize the reader to the issues inherent in representing certain (external) architectural concerns in UML.

The UML meta class *Operation* matches the C2 concept of *Message Specification*. UML Operations consist of a name and a parameter list and indicate whether they will be provided or required. To model C2 message specifications we add a tag to differentiate notifications from requests and constrain operation to have no return values. Unlike UML operations, C2 messages are all public, but that constraint is built into the UML meta-class *Interface* used below.

Stereotype C2Operation for instances of meta-class Operation

[1] C2Operations are tagged as either notifications or requests.

```
C2MSGTYPE : ENUM { NOTIFICATION, REQUEST }
```

[2] C2 messages do not have return values.

```
SELF.PARAMETER->FORALL(P | P.KIND <> RETURN)
```

We represent C2 components in UML using the meta class *Class*. Classes may provide multiple interfaces with operations, may own internal parts, and may participate in associations with other classes. We chose to model components as classes as an illustration only. There are other possibilities, including UML Component or Package meta classes (for example, see [4]). In the next section, we indeed show an example of a C2 component represented as a collection of classes.

Stereotype C2Interface for instances of meta-class Interface

A C2 interface has a tagged value identifying its position.

```
C2POS : ENUM { TOP, BOTTOM }
```

All C2Interface operations must have stereotype C2Operation.

```
SELF.OCLTYPE.OPERATION->FORALL(O | O.STEREOTYPE = C2OPERATION)
```

Stereotype C2Component for instances of meta-class Class

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

```
SELF.OCLTYPE.INTERFACE->SIZE = 2 AND
SELF.OCLTYPE.INTERFACE->FORALL(I|I.STEREOTYPE=C2INTERFACE) AND
SELF.OCLTYPE.INTERFACE->EXISTS(I | I.C2POS = TOP) AND
SELF.OCLTYPE.INTERFACE->EXISTS(I | I.C2POS = BOTTOM)
```

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

```
LET TOPINT = SELF.OCLTYPE.INTERFACE->SELECT(I|I.C2POS = TOP),
LET BOTINT = SELF.OCLTYPE.INTERFACE->SELECT(I|I.C2POS = BOTTOM),
TOPINT.OPERATION->FORALL(O|(O.C2MSGTYPE=REQUEST) IMPLIES O.DIR=REQUIRE) AND
BOTINT.OPERATION->FORALL(O|(O.C2MSGTYPE=REQUEST) IMPLIES O.DIR=PROVIDE)
```

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

[4] C2Components participate in at most two whole-part relationships named internalObject, and dialog.

```
LET WHOLES = SELF.OCLTYPE.ASSOCEND->SELECT(AGGREGATION = COMPOSITE),
(WHOLE->SIZE <= 4) AND
((WHOLES.ASSOCIATION.NAME->ASSET)-SET{“INTERNALOBJECT”,“DIALOG”})->SIZE=0
```

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

```
LET OPS = SELF.INTERNALOBJECT.FEATURE->SELECT(F | F-
>ISKINDOF(OPERATION)),
LET BOTINT = SELF.OCLTYPE.INTERFACE->SELECT(I | I.C2POS = BOTTOM),
OPS->FORALL(OP | BOTINT->EXISTS(NOTE | (OP.NAME = NOTE.NAME AND
                OP.PARAMETER = NOTE.PARAMETER) IMPLIES
                NOT.DIR = REQUIRED AND NOTE.C2MSGTYPE=NOTIFICATION))
```

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

Stereotype C2AttachOverComp for instances of meta-class Association

Stereotype C2AttachUnderComp for instances of meta-class Association.

Stereotype C2AttachConnConn for instances of meta-class Association

Stereotype C2Connector for instances of meta-class Class

[1-5] Same as constraints 1-5 on C2Component.

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

```
LET TOPINT = SELF.OCLTYPE.INTERFACE->SELECT(I | I.C2POS = TOP),
LET DOWNATTACH = SELF.OCLTYPE.ASSOCEND.ASSOCIATION->SELECT(A |
     A.ASSOCEND[2] = SELF.OCLTYPE),
LET TOPSINTSBELOW=DOWNATTACH.ASSOCEND[1].INTERFACE->SELECT(I|I.C2POS=
     TOP), TOPSINTSBELOW.OPERATION->ASSET = TOPINT.OPERATION->ASSET
```

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

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

Stereotype C2Architecture for instances of meta-class Model

[8] A C2 architecture is made up of only C2 model elements.

```
SELF.OCLTYPE.MODELELEMENT->FORALL(ME|ME.STEREOTYPE= C2COMPONENT OR
    ME.STEREOTYPE = C2CONNECTOR OR ME.STEREOTYPE = C2ATTACHOVERCOMP OR
ME.STEREOTYPE = C2ATTACHUNDERCOMP OR ME.STEREOTYPE = C2ATTACHCONNCONN)
```

[9] Each C2Component has at most one C2AttachOverComp.

```
LET COMPS=SELF.OCLTYPE.MODELELEMENT->SELECT(ME|ME.STEREOTYPE=C2COMPONENT),
```

```
COMPS->FORALL(C | C.ASSOCEND.ASSOCIATION->SELECT(A |
           A.STEREOTYPE = C2ATTACHUNDERCOMP)->SIZE <= 1)
```

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

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

```
LET COMPS=SELF.OCLTYPE.MODELELEMENT->SELECT(ME|ME.STEREOTYPE=C2COMPONENT),
COMPS->FORALL(C | C.ASSOCEND.ASSOCIATION->SIZE > 0)
```

[12] Each C2Connector must be attached to some connector or component. Similar to the constraint above.

5.3 C2 Integration

To demonstrate the C2/UML integration we need to consider an architecture defined in the C2 style and a corresponding design in UML. Figure 5 (right side) shows a UML class diagram that realizes or refines the C2 architecture presented in Figure 4. Mapping from UML classes to C2 components/connectors is shown with dotted lines. Basically, C2 components and connectors may be seen as the interfaces for compact, self-sustaining sections of the implementation. Since C2 elements (components and connectors) are often coarse grain, it is reasonable to assume that a collection of classes is needed to implement a single C2 element.

Because of the fact that a C2 element is a black box, nothing can be said about how classes that are part of a single element are supposed to interact. However, the interaction of classes belonging to different C2 elements are constrained by the C2 style. For

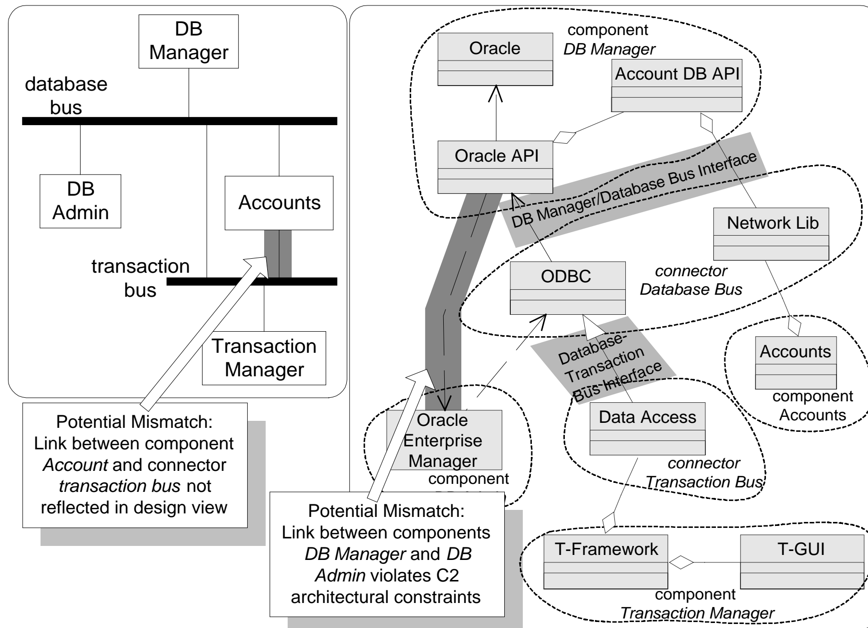

Figure 5. C2 and corresponding Class View plus Mismatches

instance, if design classes *Oracle*, *Oracle API*, and *Account DB API* correspond to the C2 component *DB Manager*, and if the design classes *ODBC* and *Network Lib* correspond to the C2 connector *Database Bus*, then the interaction between these two groups of classes needs to be consistent with the corresponding C2 architecture constraints. In the right half of Figure 5, the class links corresponding to the C2 component/connector link from *DB Manager* to *Database Bus* can be seen in the shaded area in the upper half of the diagram. The shaded area in the lower half is the class link corresponding to the C2 connector to connector link from *Database Bus* to *Transaction Bus*.

5.3.1 C2 Structural Integration

Having (manually) identified the *Mapping* from C2 to UML, we can now automatically identify mismatches between the C2 view and the UML class view in Figure 5. In this example, no transformation is needed since the structures of both views are similar (the need for transformation for the C2/UML integration will be illustrated later). Figure 6 shows a simplified algorithm that can be used to identify mismatches.

The first step in Figure 6 corresponds to the *Mapping* activity outlined above. In the second step we need to traverse both the C2 diagram and the UML class diagram and mark all links with corresponding counterpart. For instance, in case of the link from *DB Manager* to *Database Bus* in Figure 4, the equivalent design-level class dependencies corresponding to that link are the dependency arrows from *ODBC* to *Oracle API* and from *Network Lib* to *Account DB API* (see Figure 5). Thus, we mark all those links and repeat that process for the remaining ones.

Step 3 further marks all links between classes that are part of a single C2 component or connector. This is necessary because the C2 architecture in Figure 4 does not specify what happens within a component/connector and thus we must assume that class configurations corresponding to single C2 elements are consistent with the architecture by default.

Once steps 2 and 3 are concluded, we should have ideally marked all links in both the C2 view and the class view. If this is not the case, then we have identified potential mismatches. Figure 5 already shows this for both the C2 and UML class diagrams with the unmarked links highlighted and pointed to by arrows:

- a C2 link between *Accounts* and *Transaction Bus* has no corresponding class relationship. This indicates a potential nonconformance since the design does not reflect everything the architecture demands.

```
1. FOR EACH C2 COMPONENT AND C2 CONNECTOR FIND CORRESPONDING UML CLASSES
2. FOR EACH C2 LINK
     FIND AND MARK C2 LINK AS WELL AS CORRESPONDING CLASS LINKS (INTERFACE)
3. FOR EACH UML CLASS LINK
     FIND AND MARK LINKS BETWEEN TWO CLASSES, WHERE BOTH CLASSES CORRESPOND TO
     ONLY ONE C2 COMPONENT OR CONNECTOR
4. FOR EACH UNMARKED C2 LINK RAISE NONCONFORMANCE MISMATCH
5. FOR EACH UNMARKED CLASS LINK RAISE INCONSISTENCY MISMATCH
6. FOR EACH C2 COMPONENT FIND AT LEAST ONE CLASS CALL DEPENDENCY BETWEEN CLASSES
     CORRESPONDING TO THAT C2 COMPONENT AND OTHER C2 COMPONENTS CONNECTED VIA
     THE SAME CONNECTOR
```

Figure 6. Differentiation Algorithm to identify Mismatches between Views

- a class dependency link from *Oracle API* to *Oracle Enterprise Manager*. These two classes belong to different C2 components and in C2 a direct link from one component to another is illegal. Thus, this unmarked link indicates a potential inconsistency since the design seems to contradict the architecture.

5.3.2 C2 Behavioral Integration

The algorithm in Figure 6 has thus far ensured the structural integration of the C2 and class diagrams. Having ensured that the proper model elements interact and none of the interaction is missing or inconsistent does not ensure that those modeling elements interact the proper way. However, the final integration step (step 6) of our algorithm addresses behavioral integration.

A C2 architecture is more than just a structure with interfaces. It also describes how components and connectors are supposed to interact. For instance, in Figure 4 the *Transaction Manager* may interact with *DB Manager* and *Accounts* and vice versa; however, the *Transaction Manager* is not allowed to interact with *DB Admin* (although they both link to the same connector). Thus, a full integration of C2 and class diagrams must also ensure that the class view dependencies adhere to C2 behavioral constraints. In order to verify that the behavior of classes corresponding to *Transaction Manager* follows the C2 architectural guidelines we need to verify the following: there must be at least one calling dependency[1] between a class corresponding to *Transaction Manager* and one corresponding to *DB Manager* and *Accounts*, and, there must be no calling dependencies between classes corresponding to *Transaction Manager* and any other classes (e.g., *DB Admin*).

In order to do that, we need a technique that allows us to abstract away intermediate helper classes. For instance, in above example of *Transaction Manager* to *DB Manager* we need to know the relationship of T-Framework (point of external access to *Transaction Manage*r) and *Oracle API* or *Account DB API* (two points of external access to *DB Manag*er). However, the relationship from, *T-Framework* to say *Oracle API* is obscured by the intermediate classes *Data Access* and *ODBC*. Thus, we need a technique that can eliminate both intermediate classes and leave only the pure (transitive) relationship from *T-Framework* to *Oracle API*. To this end, we use the transformation technique Rose/Architect [13].

Rose/Architect (RA) (described in detail in [13]) identifies patterns of groups of three classes and replaces them with simpler patterns using transitive relationships. Currently, the RA model consists of roughly 80 rules of abstraction. Rule 2, for instance, describes the case of a class which is generalized by a second class (opposite of inheritance), which in turn is dependent on a third class (see Figure 7). This three-class pattern can now be simplified by eliminating the middle class and creating a transitive relationship (a dependency in this case) which spans from the first class to the third one. The underlying RA model describes these rules and how they must be applied to yield an effective result.

Figure 7 shows the RA refinement steps for the case of the *Transaction Manager* to *DB Manager* relationship of our design view (Figure 5). After applying two rules

---

[1] Note that C2 prohibts the use of shared variables. Since classes do not make use of events or other triggering mechanism this leave only procedure calls as an option.

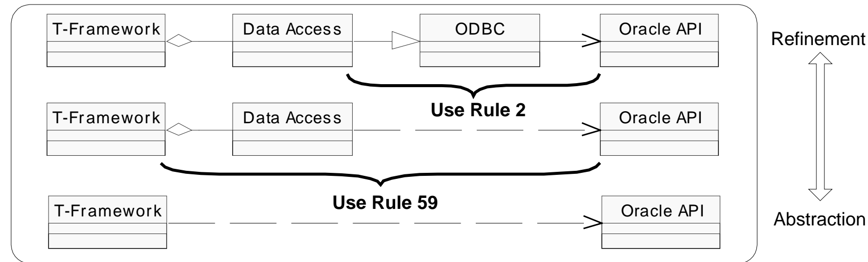

Figure 7. Using Rose/Architect to derive Behavior Dependencies

(rules 2 and 59 respectively, see [13]) we get a simplified pattern of two classes and a dependency relationship between them. If this is also done for the other classes discussed above, we can automatically verify whether the behavior of a class diagram conforms to the behavioral constraints opposed by a C2 architecture. Through this process, we find a potential mismatch between the classes corresponding to *Accounts* and *DB Manager:* no dependency relationship (mandated by the C2 architecture) could be found after abstracting away the *Network Lib* helper class.

6 Conclusion

This work outlined the differences between view representation and view integration. The former is satisfied by merely using some predefined modeling elements and adapting them so that they may be used to represent new modeling elements. The works of Robbins et al. [3] on C2 and Wright, Hofmeister et al. [4] on various conceptual architectural views, and Lyons [5] on ROOM are examples of representing architectural notations in UML. These approaches fall short of fully integrating the notations with UML, however, their view representations provide a valuable starting point for view integration. Automated integration is only possible once views are represented within a single model that supports both a common way of accessing modeling elements and cross-referencing them.

In order to integrate C2 into UML, we presented a view integration framework and demonstrated its use. For view integration, we needed to accomplish the three activities discussed in Figure 3: *Mapping*, *Transformation*, and *Differentiation*. *Differentiation* was demonstrated using the algorithm in Figure 6; Rose/Architect was presented as a *Transformation* technique to support behavioral analysis; we did not discuss *Mapping* in detail, as it was discussed elsewhere.

As mentioned previously, views are nothing more than an abstraction of relevant information from its model. Views are necessary to present that information in some meaningful way to a stakeholder (developer, architect, customer, etc.). When we talk about the need to integrate views, we are really talking about the need of having a system model integrated with its views. Although a full integration effort may seem improbable at this point, our initial experience indicates that it can still be attempted and (semi) automated for significant parts of a system. To date we have provided

initial tool support to semi-automate both C2 to UML representation as well as C2 to UML integration [14].

7 Acknowledgements

This research is sponsored by DARPA through Rome Laboratory under contract F30602-94-C-0195 and by the Affiliates of the USC Center for Software Engineering: http://sunset.usc.edu/CSE/Affiliates.html.

8 References

1. Siegfried, S.: *Understanding Object-Oriented Software Engineering*. IEEE Press (1996)
2. Rumbaugh, J., Jacobson, I., and Booch, G.: *The Unified Modeling Language Reference Manual*. Addison-Wesley (1998)
3. Robbins, J. E., Medvidovic, N., Redmiles, D. F., Rosenblum, D. S.: Integrating Architecture Description Languages with a Standard Design Method. *Proceedings of the 20th International Conference on Software Engineering*, Kyoto, Japan (1998)
4. Hofmeister, C., Nord, R. L., and Soni, D.: Describing Software Architecture in UML. *Proceedings of the First Working IFIP Conference on Software Architecture (WICSA1)*, Kluwer Academic Publishers, Boston, Dordrecht, London (1999) 145-159
5. Lyons, A.: UML for Real-Time Overview. White Paper, ObjectTime (1998)
6. Taylor, R. N., Medvidovic, N., Anderson, K., Whitehead, Jr., E. J., Robbins, J. E., Nies, K. A., Oreizy, P., and Dubrow, D. L. A component and message-based architectural style for GUI software, *IEEE Trans. Software Engineering*, June, Vol.22, No.6 (1996) 390-406
7. Allen, R. and Garlan, D. A Formal Basis for Architectural Connection, *ACM Transactions on Software Engineering and Methodology*, Vol. 6, No. 3, July (1997) 213-249
8. Luckham, D. C. and Vera J. An Event-Based Architecture Definition Language, *IEEE Transactions on Software Engineering*, September (1995)
9. Buschman, F., Meunier, R., Rohnert, H., Sommerlad, P., Stal, M.: *A System of Patterns: Pattern-Oriented Software Architecture*. Wiley (1996)
10. Egyed, A.: Automating Architectural View Integration in UML. *Technical Report USC-CSE-99-511*, Center for Software Engineering, University of Southern California, Los Angeles, CA 90089-0781 (1999)
11. Egyed, A.: Using Patterns to Integrate UML Views. *Technical Report USCCSE-99-515*, Center for Software Engineering, University of Southern California, Los Angeles, CA 90089-0781 (1999)
12. N., Medvidovic, D.S. Rosenblum: Assessing the Suitability of a Standard Design Method for Modeling Software Architectures. *Proceedings of the TC2 First Working IFIP Conference on Software Architecture (WICSA1)*, Kluwer Academic Publishers (1999) 161-182
13. Egyed, A. and Kruchten, P.: Rose/Architect: a tool to visualize software architecture. *Proceedings of the 32nd Annual Hawaii Conference on Systems Sciences* (1999)
14. Center for Software Engineering, University of Southern California: Software Architecture, *http://sunset.usc.edu/software_architecture/*