Published in Proceedings of the 3rd International Conference on Foundational Aspects of Software Engineering (FASE), Berlin, Germany, 2000.

A Formal Approach to Heterogeneous Software Modeling

Alexander Egyed and Nenad Medvidovic

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

Abstract. The problem of consistently engineering large, complex software systems of today is often addressed by introducing new, "improved" models. Examples of such models are architectural, design, structural, behavioral, and so forth. Each software model is intended to highlight a *particular view* of a desired system. A combination of multiple models is needed to represent and understand the *entire system*. Ensuring that the various models used in development are consistent relative to each other thus becomes a critical concern. This paper presents an approach that integrates and ensures the consistency across an architectural and a number of design models. The goal of this work is to combine the respective strengths of a powerful, specialized (architecture-based) modeling approach with a widely used, general (design-based) approach. We have formally addressed the various details of our approach, which has allowed us to construct a large set of supporting tools to automate the related development activities. We use an example application throughout the paper to illustrate the concepts.

1 Introduction

The software community places great hopes on software modeling notations and techniques to ease a variety of development challenges. The intellectual toolset available to software developers has been steadily enriched with more powerful and more comprehensive models. At the same time, the growing number of heterogeneous models has resulted in an observable split within this community: one part of the community is working on and with special-purpose development models; another part focuses on general-purpose models. Special-purpose models tend to focus on individual software development issues and are typically accompanied by powerful analytical evaluation tools. General-purpose models, on the other hand, address a wider range of issues that arise during development, typically resulting in a family of models that span and relate those issues.

This split has impacted the two communities' visions of how software development challenges are best addressed. A special-purpose approach is typically centered around a single design notation with a narrow modeling and analysis focus (e.g., an architecture description language, or ADL [14]). A general-purpose approach embraces a *family* of design notations with a much broader, system-wide focus (e.g., the

Unified Modeling Language, or UML [1]). Thus, for instance, UML emphasizes modeling practicality and breadth, while ADLs tend to emphasize rigor and depth. Both these perspectives are needed to address the broad spectrum of rapidly changing situations that arise in software development. Our previous work has demonstrated that the two perspectives can indeed play complementary roles in *modeling and analyzing software architectures* [10,12,16]. We have recently begun using combinations of the general- and special-purpose approaches to aid us in the task of *sound software refinement*: refining high-level software models (i.e., architectures) into lower-level models (i.e., designs and implementations) in a manner that preserves the desired system properties and relationships [7].

This paper discusses the issues we have encountered in attempting to bridge the two perspectives and the solutions we have developed to address those issues. In particular, we have augmented our ADL-based approach to modeling, analysis, simulation, and evolution of architecturally relevant aspects of a system (e.g., system-level structure and interactions, performance, reliability), with the strengths of UML: supporting a broad range of both high- and low-level development concerns with a widely adopted, standard notation.

Integrating an ADL-based approach with UML is a non-trivial task. One difficulty arises from the difference in the modeling foci, language constructs, and assumptions between an ADL and UML. Therefore, the first critical challenge is to ensure that the model, as specified in the ADL, is transferred into UML as faithfully as possible, and vice versa. Another difficulty is a by-product of the numerous modeling notations within UML (component, class, object, state chart, use case, activity, etc. diagrams): one view of a system model, as depicted in one notation (e.g., class diagram), may be inconsistent with another view, as depicted in another notation (e.g., activity diagram). Thus, the second critical challenge is to ensure that changes made in one view are reflected as faithfully as possible in another.

We have developed and exploited two techniques to deal with these two challenges. They are illustrated in Figure 1. The figure depicts the relationship between the UML modeling constructs ("Core UML") and the constructs of an ADL, such as Rapide [6] or C2 [18]. As indicated in the figure, only a certain, small number of ADL constructs can be represented in "Core UML". This should come as no surprise since, as discussed above, ADLs are special-purpose notations that tend to favor rigor and formalism - concepts that are less present in UML due to its practitioner-oriented, general-purpose nature. However, UML provides a mechanism that allows "Core UML" to be extended to address new modeling needs (depicted by the "UML Extensibility Mechanism" ellipse in Figure 1). We exploit this feature of UML to address the first challenge: supporting the transformation of an ADL model into a UML model and vice

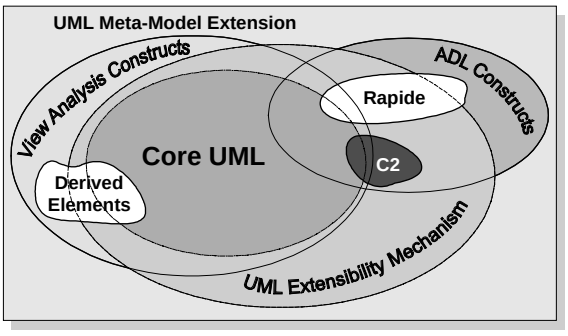

Fig. 1. Extending and Augmenting UML; challenges to represent ADL and analysis Constructs in UML.

versa.[1] The second challenge deals with ensuring the consistency across UML's modeling notations. Since this area has not been addressed by the designers of UML, the strategy we pursue is to augment UML with constructs that maintain the relationships among the modeling elements *across* different notations (depicted by the "View Analysis Constructs" ellipse in Figure 1). In tandem, the two techniques allow us to specify, refine, and integrate the heterogeneous models supported by an ADL and by the different UML notations, alleviating the shortcomings of both languages in the process.

The remainder of the paper is organized as follows. Section 2 presents our approach to software architecture modeling, analysis, and simulation. Section 3 outlines our technique for transferring architectural decisions into UML. Section 4 focuses on ensuring the consistency among UML models, both across different UML notations and across levels of abstraction (e.g., high-level vs. low-level design). Our conclusions round out the paper.

2 Our Approach to Architecture-Based Development

The concepts discussed in this paper are independent of the application domain, the specifics of the chosen architectural style, and the modeling language(s). For illustration purposes, we have selected the C2 architectural style [18]. C2 serves solely as a vehicle for exploring our ideas. It allows us to discuss the development issues relevant at the architectural level and motivates the discussion of transforming an architectural model into UML. This section also highlights certain approaches to software modeling and analysis that are not available in UML.

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." Components communicate solely by exchanging messages. 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. However, two buses can be attached together.

We illustrate the above concepts with an example. The architecture we use is a logistics system for routing incoming cargo to a set of warehouses, shown in Figure 2. The *DelPort*, *Vehicle*, and *Warehouse* components are objects that keep track of the states of a delivery port, a transportation vehicle, and a warehouse, respectively. Each may be instantiated multiple times in a system. The *DelPortArt*, *VehicleArt*, and *WarehouseArt* components are responsible for graphically depicting the states of their respective objects to the end-user. The *CargoRouter* organizes the display based on the actual number of port, vehicle, and warehouse instances in the system. The *Clock* provides consistent time measurement to interested components, while the *NextShipment* 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 warehouse. The *GraphicsBinding* component renders the drawing requests sent from the *CargoRouter* using a graphics toolkit, such as Java's AWT. The five connectors

[1] Note that a portion of Rapide falls outside this "extended UML" in Figure 1. This is a reflection of the fact that UML is not always able to support all features of a given ADL [12].

receive, filter, and route the messages sent by components to their appropriate recepients.

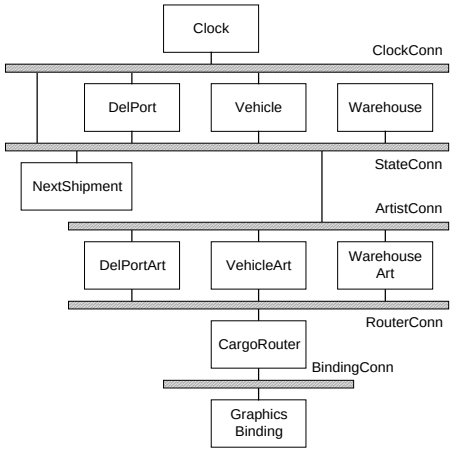

Fig. 2. Architecture of the cargo routing system

2.1 Architectural Analysis

To support architecture modeling, analysis, implementation, and evolution, we have developed a formal *type system* for software architectures [8,11,13]. We treat every component specification at the architectural level as an *architectural type*. We distinguish architectural types from *basic types* (e.g., integer, boolean, string, array, record, etc.). A component has a name, a set of internal state variables, a set of interface elements, an associated behavior, and (possibly) an implementation. Each interface element has a direction indicator (*prov*ided or *req*uired), a name, a set of parameters, and (possibly) a result. Component behavior consists of an invariant and a set of operations. The invariant is used to specify properties that must be true of all component states. Each operation has preconditions, postconditions, and (possibly) a result. Like interface elements, operations can be *prov*ided or *req*uired. The preconditions and postconditions of required operations express the *expected* semantics for those operations. We separate the interface from the behavior, defining a mapping function from interface elements to operations. This function is a total surjection: each interface element is mapped to exactly one operation, while each operation implements at least one interface. An interface element can be mapped to an operation only if the types of its parameters are subtypes of the corresponding variable types in the operation, while the type of its result is a supertype of operation's result type. This property directly enables a single operation to export multiple interfaces.

This set of definitions allows us to formally treat two distinct development activities: evolution of individual components via subtyping [8] and analysis of an architecture for conformance among interacting components. In this paper we focus on the latter. The left side of Figure 3 shows the relevant definitions of interface parameter conformance and operation conformance.[2] The right side of Figure 3 is a reflection of our experience that components need not always be able to fully interoperate in an architecture, but that mismatches should be allowed under certain situations (e.g., COTS reuse). The two extreme points on the spectrum of type conformance are: *minimal type conformance*, where at least one service (interface and

[2] The definitions are specified in Z, a language for modeling mathematical objects based on first order logic and set theory [17]. Z uses standard logical connectives ($\wedge$, $\vee$, $\Rightarrow$, etc.) and set-theoretic operations($\in$, $\cup$, $\subseteq$, etc.). The complete formal defintion of the type system is given in [11].

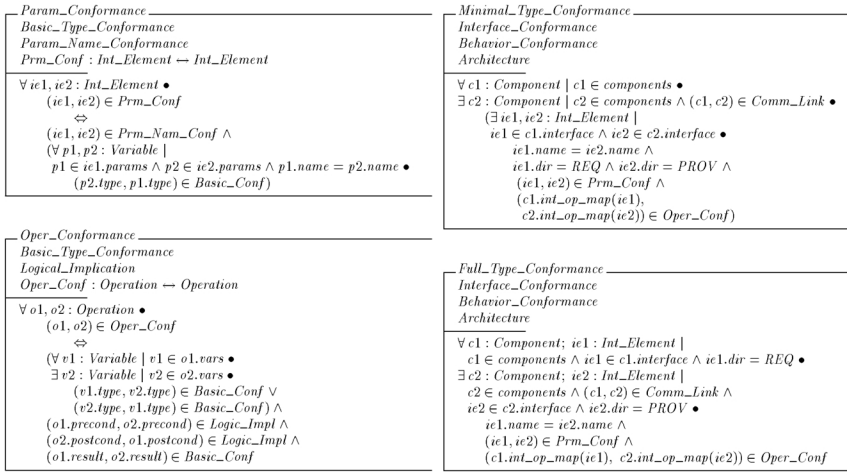

Fig. 3. Formal specification of architectural type conformance predicates

corresponding operation) required by each component is provided by some other component along its communication links; and *full type conformance*, where every service required by every component is provided by some component along its communication links.

2.2 Example of Architectural Analysis

We illustrate architectural type conformance with a simple example drawn from the cargo routing application (recall Figure 2). Figure 4 shows partial models of the *DelPort* and *DelPortArt* components, specified in the C2SADEL ADL [13]. The two components are *intended* to interact in the cargo routing system (i.e., there is a communication path between them in the architecture via *StateConn* and *ArtistConn*). In the example from Figure 4, *DelPortArt* requires one service: *unloadShipment*, which is mapped to its operation *or1*. *DelPort* provides an operation, *op1*, with a matching interface (as required by the interface parameter conformance predicate in Figure 3). Thus, to establish type conformance, we must make sure that the pre- and postconditions of the two operations are properly related as specified in the operation conformance and minimal type conformance predicates in Figure 3. To do so, we must establish that the following relationships hold:

- *DelPortArt or1 pre* ⇒ *DelPort op1 pre*

  `(s ∉ contents) ⇒ true`

  Since *DelPort*'s *op1* does not have any preconditions, the right-hand side (RHS) of the implication becomes `true`. Therefore the entire implication evaluates to true regardless of the truth value of its left-hand side (LHS).
- *DelPort op1 post* ⇒ *DelPortArt or1 post*

  `((~cap = cap + ShpSize(s)) ∧ (s ∈ ~cargo)) ⇒ (s ∈ ~contents)`

```
component DelPortComponent is {
state { cap : Int; max_cap : Int;
        cargo : \set Shipment;
        ShpSize : Shipment -> Int }
invariant { cap \eqgreater 0 \and
            cap \eqless max_cap; }
interface {
   prov ip1: unloadShipment(s : Shipment);
   req ir1: Tick(); }
operations {
   prov op1: {
      let s : Shipment;
      post (~cap = cap + ShpSize(s))
           \and (s \in ~cargo); }
   req or1: {
      let time : STATE_VARIABLE;
      post ~time = 1 + time; } }
map {
   ip1 -> op1 (s -> s);
   ir1 -> or1 (); }
}
```

```
component DelPortArtComponent is {
state { selects : \set Int;
        UniqueID : Int \x Int -> Int; }
invariant { #selects \eqgreater 0; }
interface {
   prov ip1: selectShipment(port : Int;
                            shp : Int);
   req ir1: unloadShipment(s : Shipment); }
operations {
   prov op1: {
      let pid : Int; sid : Int;
      post (#~selects = #selects + 1) \and
           (UniqueID(pid,sid) \in selects); }
   req or1: {
      let s : Shipment;
          contents : STATE_VARIABLE;
      pre (s \not_in contents);
      post (s \in ~contents); } }
map {
   ip1 -> op1 (port -> pid, shp -> sid);
   ir1 -> or1 (s -> s); }
}
```

Fig. 4. Partial specification of the DelPort and DelPortArt components. # denotes set cardinality; ~ denotes the value of a variable after the operation executes; STATE_VARIABLE is a placeholder for any basic type

The two matching interface elements (*unloadShipment*) of the *DelPort* and *DelPortArt* components have matching parameters (`s`), which are mapped to the respective components' operation variables (also `s`). *DelPortArt*'s variable `contents` is of type STATE_VARIABLE, which is intended to generically describe the internal state of another component in a required operation.[3] `contents` can be unified with the *DelPort* internal state variable `cargo`, so that the implication becomes

$((\sim cap = cap + ShpSize(s)) \wedge (s \in \sim cargo)) \Rightarrow (s \in \sim cargo)$

This implication is of the form $(A \wedge B) \Rightarrow B$ and is also true: an implication can be false only if its RHS (`B`) evaluates to false; however, in this case that would result in the LHS $(A \wedge B)$ also evaluating to false, making the implication true.

We have thus established that, at the least, minimal type conformance holds in the architectural interaction between *DelPort* and *DelPortArt*.

2.3 Architectural Simulation

The example above demonstrate how an architecture can be analyzed statically for desired properties. Another way in which we have been able to analyze C2-style architectures has been through early simulations of applications, built based on the applications' architectural models. To this end, we have exploited C2's event-based nature: one can rapidly construct a partial implementation of an architecture that mainly focuses on the components' external (asynchronous message) interfaces. For example, a prototype of the cargo routing application discussed above was initially implemented and later augmented to include a foreign-language user interface by a single developer in a matter of hours [2]. Our support tools also allow insertion of event monitors and filters to explicitly observe message traffic and assess dynamic properties of the architecture. Any inconsistencies in the architecture not detected by

[3] For a more formal and in-depth treatment of STATE_VARIABLE types, see [11]

type conformance checking, such as inconsistencies in component interaction protocols, are likely to manifest themselves during simulations.

2.4 Tool Support

The activities described in this section (architecture modeling, analysis, simulation, implementation, and evolution) are supported by DRADEL, a component-based development environment [13], and a light-weigt simulation and implementation infrastructure [9]. Three of DRADEL's components are particularly relevant to this discussion (a fourth component will be discussed in Section 3):

- The *TopologicalConstraintChecker* component analyzes an architecture for adherence to design heuristics and architectural style rules. Currently, this component ensures the rules of the C2 style, but can be easily replaced with components enforcing other kinds of design rules.
- The *TypeChecker* component implements the rules of our architectural type system, briefly discussed in Section 2.1. The *TypeChecker* automatically performs the conformance checks such as those shown in Section 2.2.
- The *CodeGenerator* component automatically generates architecture implementation skeletons discussed in Section 2.3. The skeletons are generated based on a component's architectural model specified in C2SADEL: all message generation, marshalling, and unmarshalling code is produced, as is a stub for the component's internal (application-specific) behavior. Component stubs are then completed manually. The amount of effort needed to complete a stub depends on the desired faithfulness of the prototype implementation to the final system and it can range from a few to several hundred lines of code.

3 Refining an Architectural Model into a UML Model

Once an architectural model is constructured and its analysis and simulation demonstrate the presence (or absence) of properties of interest, the model can be used as a basis for system design and implementation. This requires the transfer of information from the architectural model to a (high-level) design and subsequent refinement of that design. One way to effectively accomplish this task is by transferring the application model from an ADL to a notation better suited for addressing lower-level design issues. We employ UML [1] to that end.

UML is a semi-formally defined design language with a large, extensible set of modeling features that span a number of modeling diagrams (see Section 1). Our previous work has studied in depth the relationsip between ADLs and UML [10,12,16]. One outcome of this research has been a set of well-defined strategies one could employ to meaningfully transfer an ADL model into UML. Based on this work, we have recently augmented the DRADEL environment discussed in Section 2.4 to include *UMLGenerator*, a component that transforms a C2SADEL specification into a UML model. *UMLGenerator* uses a set of formally defined rules that specify how each C2SADEL construct is transformed into a corresponding (set of) UML construct(s). The rules make use of predefined UML constructs, as well as

stereotypes, UML's built-in extensibility mechanisms. Stereotypes allow the addition of attributes to existing modeling elements (via *tagged values*) and the restriction of modeling element semantics (via *constraints*). The constraint portion of a stereotype is formally specified in UML's Object Constraint Language (OCL) [19]. For example, the compositional rules of the C2 architectural style, discussed in Section 2, can be specified using a UML stereotype as follows.

Stereotype C2Architecture for instances of meta class Model

[1] 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)
```

[2] Each C2Component has at most one C2AttachConnAbove.

```
LET COMPS=SELF.OCLTYPE.MODELELEMENT->SELECT(ME|ME.STEREOTYPE=C2COMPONENT),
COMPS->FORALL(C | C.ASSOCEND.ASSOCIATION->SELECT(A |
            A.STEREOTYPE = C2ATTACHCONNABOVE)->SIZE <= 1)
```

[3] Each C2Component has at most one C2AttachConnBelow.
Similar to the constraint above.

[4] 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)
```

[5] Each C2Connector must be attached to some connector or component.

```
LET CONNS=SELF.OCLTYPE.ELEMENTS->SELECT(E|E.STEREOTYPE=C2CONNECTOR),
CONNS->FORALL(C | C.ASSOCEND.ASSOCIATION->SIZE > 0)
```

The above stereotype describes only one portion of C2 and is accompanied by other stereotypes defining C2 components and connectors, their interactions, and the internal makeup of individual components and connectors as specified in C2SADEL [13]. This complete specification of C2 concepts in terms of UML stereotypes was used as the basis for implementing the *UMLGenerator* component. Figure 5 shows a partial and somewhat simplified set of C2SADEL-to-UML transformation rules as encoded in the *UMLGenerator*.

Internal Component 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
Architecture Configuration (explicit invocation) → (Object) Collaboration Diagram
    Component Instance → Internal Component Object Class Instance
    Connector Instance → «Interface» Class Instance
    Component/Connector Binding → Object Link (instance of an association)
Component → «C2-Component» Class
    Internal Component Object → «C2-Component» Class Attribute
    Component Top Interface → «Interface» Class
    Component Bottom Interface → «Interface» Class
    Outgoing Message → «Interface» Class «out» Operation
Incoming Message → «Interface» Class «in» Operation

Fig. 5. Excerpt from the rule set for transforming C2SADEL into UML. «» denotes stereotypes

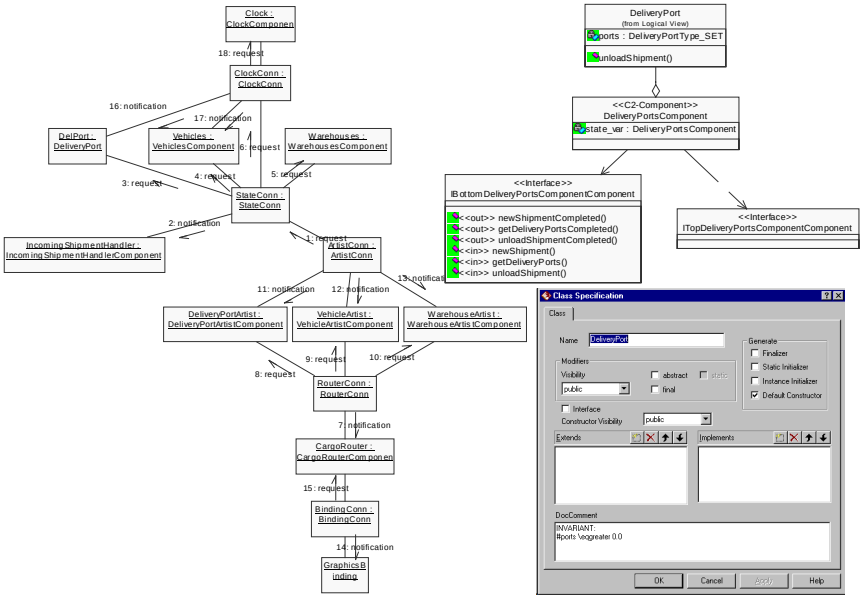

Fig. 6. Partial UML model of CargoRouter architecture (using Rational Rose™)

The UML specification resulting from this transformation is stored as a Rational Rose™ model [15]. A portion of the Rose model for the cargo router architecture from Figure 2 is shown in Figure 6: it depicts the entire architecture as a UML collaboration diagram (left) and the attributes of and class diagram corresponding to the *DeliveryPort* component (right). This automatically generated Rose model is consistent with the architectural model and is used as the basis for further, possibly manual refinement of the architecture as discussed below.

4 Design Refinement and Analysis

Architectural refinement enables us to transform our C2 architecture into a (high-level) UML design. Since that design will likely be further refined into lower-level designs (and an implementation), those subsequent refinements may become inconsistent with the original architecture. This is particularly likely if the refinements are done manually. This section will discuss how a refinement can be automated by employing a technique that augments UML (recall Figure 1).

4.1 View Integration Framework

To enable automated design analysis we have devised and applied a view integration framework, accompanied with a set of activities and techniques for identifying

mismatches in an automatable fashion [3]. This approach exploits redundancy between views: for instance, if view A contains information about view B, this information can be seen as a constraint on B. The view integration framework is used to enforce such constraints and, thereby, the consistency across the views. In addition to constraints and consistency rules, our framework also defines *what* information can be exchanged and *how* it can be exchanged. This is critical for automating the process of identifying and resolving inconsistencies since direct comparison between views is usually infeasible. Our framework has three major activities:

- **Mapping** identifies related pieces of information and thereby describes what information is overlapping. Mapping is often done manually, e.g., via naming dictionaries and traceability matrices.
- **Transformation** simplifies (i.e., abstracted) detailed views or generalizes specific views. This activity describes how information can be exchanged and results in derived modeling information.
- **Differentiation** traverses the model to identify mismatches. *Mapping* indicates what information should be compared; *Transformation* indicates how that information should be compared.

We will illustrate these concepts using the cargo router example. Figure 7 depicts an excerpt of its design in UML in the form of a class diagram. This design varies considerably from the high-level design we generated in Section 3. As the design is further modified, it will become increasingly difficult to see whether the changes are consistent with the original architecture, depicted in Figure 2. Again, note that the architectural and design views show distinct but overlapping information; this redundancy between the views is used in the form of constraints in helping to verify consistency. It has been our observation that the major challenge of view integration is not the actual comparison of views (*Differentiation*) but instead the *Transformation* and *Mapping* of modeling information [3]. Since *Mapping* tends to be predominantly manual, *Transformation* becomes the key enabling technology for automated view integration. We, therefore, discuss it on more detail.

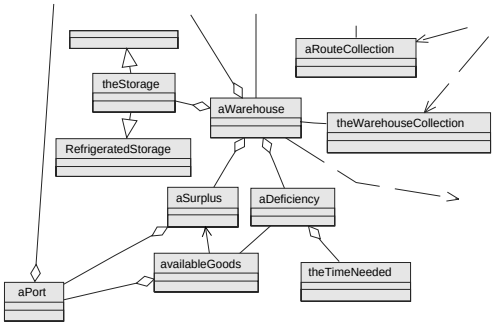

Fig. 7. Design Refinement of CargoRouter

4.2 Transformation

Figure 8 captures the three major dimensions of view transformation [3]. Software modeling views can be seen as abstract or concrete on the one hand, and generic or specific on the other. The abstract-concrete dimension was foreshadowed in Section 3 where the C2 architecture was the abstract view and the generated UML model was the concrete view. Note that a view's level of abstraction is relative. Thus, for in-

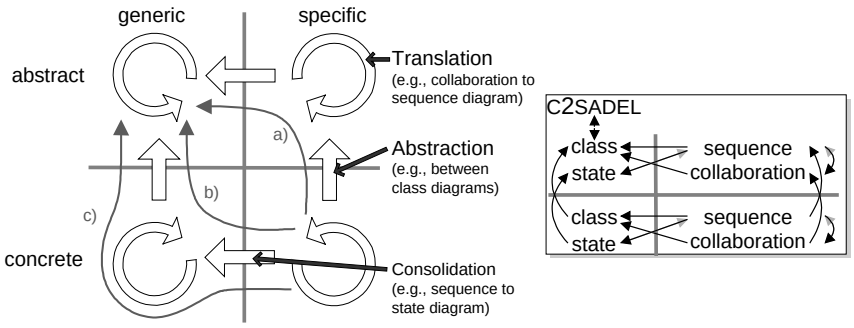

Fig. 8. Two dimensions of views with three dimensions of view transformations. Paths a-d denote possible transformation paths from concrete-specific to abstract.

stance, the derived UML model depicted in Figure 6 is concrete relative to the C2 view but abstract relative to the class diagram in Figure 7.[4]

The generic-specific dimension denotes the generality of modeling information. For instance, a class diagram naturally describes a general relationship between classes, whereas a sequence diagram describes a specific scenario. Another way of saying this is that constructs in general views must capture the union of all specific views, and that a specific view must fit within the boundaries of its respective generic one. The level of generality/specificity is, again, in the eye of the beholder (e.g., a view is more generic than some views and more specific than others).

Having two dimensions of views implies three types of transformation axes: *Abstraction* to capture vertical transformation, *Consolidation* to capture horizontal transformation, and *Translation* to capture transformation within a single quadrant (both input and output views are of the same category). We have also observed that often only uni-dimensions transformations can be fully automated (e.g., [4,5]) - usually going from concrete to abstract or from specific to generic.

The framework depicted in Figure 8 allows us to combine simple transformation techniques to enable more complex transformations in support of the integration of C2SADEL and UML. Figure 8 depicts one such complex transformation going from the lower right quadrant (e.g., a concrete sequence diagram) to the upper left quadrant (e.g., an abstract class diagram). The three paths ("a" to "c") indicate the three alternative transformation scenarios on how to get there. Path "a" first abstracts the sequence diagram and then consolidates it to yield a class diagram view. Path "b" first consolidates and then abstracts. Path "c" consolidates, translates, and abstracts. There are of course other variations using additional translations. The translation step may seem redundant since it does not seem to put us any closer to the target quadrant. Nevertheless, translation can help us in circumventing a non-existing transformation by translating to another view that can be abstracted/consolidated or by switching to a more reliable abstraction/consolidation method not available within the current view.

[4] Note that the implementation is the most concrete view of a system.

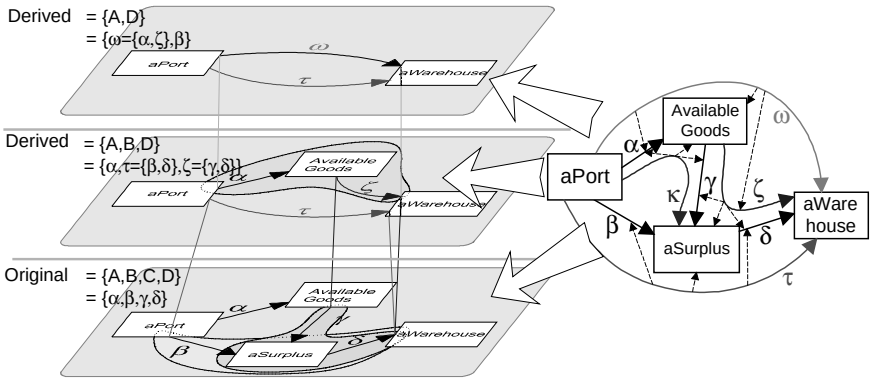

Fig. 9. Representation of Abstraction Transformation (multiple views and their storage)

The right hand-side of Figure 8 depicts existing transformations our approach currently supports (e.g., concrete class to abstract class, sequence to state, etc.). Note that a few arrows are double-headed, with the second head drawn in a light gray shade. These cases denote examples where the reverse transformation is semi-automated.

4.3 Implication of Transformation for Consistency Checking

Complex transformations are not just a serial execution of individual transformations. To enable complex transformations we also need to address the issue of how to capture and maintain modeling information – in particular, how to handle derived modeling information [3]. Thus, transformation methods need to be integrated in order to avoid the two fundamental problems that may arise:

1. Transformation Redundancy: a modeling element should be transformed at most once using the same transformation method, thus avoiding transformation redundancy.
2. Multiple Derived Results: multiple transformations of a modeling element using different transformation methods may result in multiple results. The model must therefore support multiple interpretations.

Figure 9 demonstrates both issues using *relation abstraction* (a method that collapses relations among multiple modeling elements into simpler relations). The bottom-most layer on the left hand-side shows an excerpt of the class diagram from Figure 7 with four classifiers (boxes) and a number of relations (links) between them. The middle layer is more abstract in that the classifier *aSurplus* has been eliminated and a more abstract relation (τ) has been created to replace it. The top-most layer further eliminates the classifier *availableGoods*. We now have three views. If we were to store the views separately, we might eventually introduce inconsistencies between them. For instance, any changes to the bottom view would require updates to all its higher-level abstractions. This would likely become unmanageable in a large-scale system with a large number of user-created views plus all their transformations. Furthermore, if we again take the bottom view and abstract *aSurplus* away, we are

duplicating both effort and storage since we are not aware of the previous abstraction. The right hand-side of Figure 9 depicts a possible solution for these two problems. It suggests using an n-ary relationship among modeling elements. The figures on the left are now simply projections of the model on the right. The underlying model on the right minimizes information duplication. Note that UML does not support n-ary relationships among many of its modeling elements, partially motivating our decision to augment it. Revisiting Figure 1, the problem of how to deal with derived modeling elements can only be solved by augmenting the model.

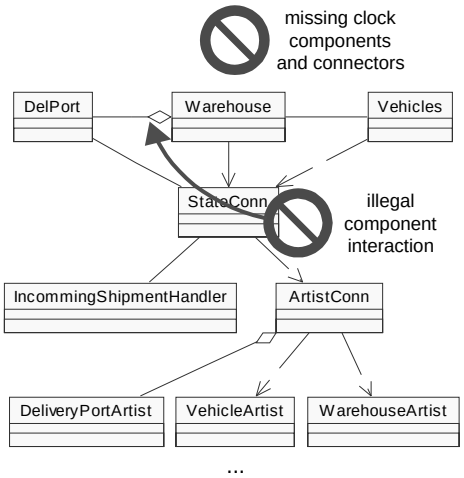

Fig. 10. Except of Abstracted Class Diagram

4.4 Consistency Checking

The issues discussed in sections 4.1 and 4.2 form the foundation that allows us to check the relative consistency of two or more UML models. Thus, in order to ensure the consistency of the UML model in Figure 7 with respect to the original C2 architecture in Figure 2 (or Figure 6) we can automatically derive the transformation paths, perform the actual transformations (the refinement of the C2 model and the abstraction of the class diagram), and then compare their respective results (two class diagrams at similar levels of abstraction). Figure 10 shows an excerpt of the abstraction from Figure 7 using a set of abstraction rules discussed in Section 4.2 and completely specified in [4]. We can now observe that the architecture differs from the design in several ways: the design does not include the *Clock* component or its related connector and depicts some component interactions (e.g., *DelPort* to *Warehouse*) that are not defined in the architecture.

Although it is possible that both these cases are results of deliberate design decisions, transformation has enabled us to easily compare the two views and quickly highlight their inconsistencies. The human designer can then either address the inconsistencies or ignore them. To date, we have automated a part of our view integration framework in a tool called UML/Analyzer [3]. The abstracted view depicted in Figure 10 can be derived automatically using our tool.

5 Conclusion

This paper addressed architectural modeling using a specialized approach, C2, and showed how it can be used to complement a general-purpose modeling language,

UML. We discussed the benefits of C2 and UML as well as the need for integrating their respective strengths. The integration of C2 and UML is not a trivial undertaking and we introduced two techniques to enable it: *constraining UML* and *augmenting it*. We showed how to represent C2 in UML (Section 3) in order to illustrate *constraining UML* and how to perform complex transformations and view integration (Section 4) in order to illustrate *augmenting UML*.

This paper also presented formal approaches to software development to support a variety of development activities (specification, analysis, and modeling). Formalism plays a major role during all of those activities. Formalism is also the foundation of automation. The benefits of automation were illustrated throughout this paper. In Section 2, we discussed the analytic powers of special-purpose models like C2. Using C2 we were able to automatically analyze static and dynamic properties of our system. Automation then helped us in further refining our special-purpose C2 model into a general-purpose UML model (Section 3). This automated refinement capability removed a major obstacle to model integration since no manual labor was required in making tools and models work together. Finally, automation supported us during model validation. We illustrated automated consistency checking between views and demonstrated this using a C2 architecture and its corresponding UML design (Sec. 4).

To date, we have automated many of the concepts discussed in this paper and created two tool suites: SAAGE (Dradel+Rose) to enable architectural modeling and refinement; and UML/Analyzer (also integrated with Rose) to enable automated transformations and consistency checking. It is our long-term vision to further extend the automation support to other aspects of the software life cycle.

6 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.

7 References

1. Booch, G., Jacobson, I., Rumbaugh, J.: The Unified Modeling Language User Guide, Addison-Wesley, 1998
2. DARPA. Evolutionary Design of Complex Software (EDCS): Demonstration Days 1999. http://www.if.afrl.af.mil/programs/edcs/demo-days-99/
3. Egyed, A.: "Integrating Architectural Views in UML," Qualifying Report, Technical Report, Center for Software Engineering, University of Southern California, USC-CSE-99-514, http://sunset.usc.edu/TechRpts/Papers/usccse99-514/usccse99-514.pdf, 1999
4. Egyed, A. and Kruchten, P.: Rose/Architect: a tool to visualize software architecture. *Proceedings of the 32nd Annual Hawaii Conference on Systems Sciences* (1999)

5. Koskimies, K., Systä, T., Tuomi, J., and Männistö, T. (1998) “Automated Support for Modelling OO Software,” IEEE Software, January, pp. 87-94.
6. Luckham, D.C. and Vera, J.: “An Event-Based Architecture Definition Language.” IEEE Transactions on Software Engineering, vol. 21, no. 9, pages 717-734, September 1995.
7. Medvidovic, N., Egyed, A., and Rosenblum, D.S.: Round-Trip Software Engineering Using UML: From Architecture to Design and Back. In Proceedings of the Second International Workshop on Object-Oriented Reengineering (WOOR’99), Toulouse, France, September 6, 1999.
8. Medvidovic, N., Oreizy, P., Robbins, J.E., and Taylor, R.N.: Using Object-Oriented Typing to Support Architectural Design in the C2 Style. In Proceedings of the Fourth ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE4), pp. 24-32, San Francisco, CA, October 16-18, 1996.
9. Medvidovic, N., Oreizy, P., and Taylor, R.N.: 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 17-19, 1997. Also in Proceedings of the 1997 International Conference on Software Engineering (ICSE'97), pp. 692-700, Boston, MA, May 17-23, 1997.
10. Medvidovic, N. and Rosenblum, D.S.: Assessing the Suitability of a Standard Design Method for Modeling Software Architectures. In Proceedings of the First Working IFIP Conference on Software Architecture (WICSA1), pp. 161-182, San Antonio, TX, February 22-24, 1999.
11. Medvidovic, N., Rosenblum, D.S., and Taylor, R.N.: A Type Theory for Software Architectures. Technical Report, UCI-ICS-98-14, Department of Information and Computer Science, University of California, Irvine, April 1998.
12. Medvidovic, N., Rosenblum, D.S., Robbins, J.E., Redmiles, D.F.: Modeling Software Architectures in the Unified Modeling Language. In submission.
13. Medvidovic, N., Rosenblum, D.S., and 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), pp. 44-53, Los Angeles, CA, May 16-22, 1999.
14. Medvidovic, N., and Taylor, R.N.: A Classification and Comparison Framework for Software Architecture Description Languages. IEEE Transactions on Software Engineering, to appear, 2000.
15. Rational Software Corporation, Rational Rose 98: Using Rational Rose
16. Robbins, J.E., Medvidovic, N., Redmiles, D.F., and Rosenblum, D.S.: Integrating Architecture Description Languages with a Standard Design Method. In Proceedings of the 20th International Conference on Software Engineering (ICSE'98), pp. 209-218, Kyoto, Japan, April 19-25, 1998.
17. Spivey, J.M.: The Z notation: a reference manual. Prentice Hall, New York, 1989.
18. Taylor, R.N., Medvidovic, N., Anderson, K.N., Whitehead, E.J., Jr., Robbins, J.E., Nies, K.A., Oreizy, P., and Dubrow, D.L.: A Component- and Message-Based Architectural Style for GUI Software. IEEE Transactions on Software Engineering, vol. 22, no. 6, pp. 390-406, 1996.
19. Warmer, J.B., Kleppe, A.G.: The Object Constraint Language: Precise Modeling With UML, Addison-Wesley, 1999