Modeling Multiple Aspects of Software Components

Roshanak Roshandel Nenad Medvidovic

Computer Science Department
University of Southern California
Los Angeles, CA 90089-0781 U.S.A.
{roshande,neno}@usc.edu

ABSTRACT

A software component is typically modeled from one or more of four functional aspects: interface, static behavior, dynamic behavior, and interaction protocol. Each of these aspects helps to ensure different levels of component compatibility and interoperability. Existing approaches to component modeling have either focused on only one of the aspects (e.g., interfaces in various IDLs) or on well-understood combinations of two of the aspects (e.g., interfaces and their associated pre- and post-conditions in static behavioral modeling approaches). This paper argues that, in order to accrue the true benefits of component-based software development, one may need to model all four aspects of components. However, this requires that consistency among the multiple aspects be maintained. We offer an approach to modeling components using the four-view perspective (called *the Quartet*) and identify the points at which the consistency among the models must be maintained.

1. INTRODUCTION

Component-based software engineering has emerged as an important discipline for developing large and complex software systems. Software components have become the primary abstraction level at which software development and evolution are carried out. We consider a software component to be any self-contained unit of functionality in a software system that exports its services via an interface, encapsulates the realization of those services, and possibly maintains transient internal state. In the context of this paper, we further focus on components for which information on their interface and behavior may be obtained. In order to ensure the desired properties of component-based systems (e.g., correctness, compatibility, interchangeability), both individual components and the resulting systems' architectural configurations must be modeled and analyzed.

The role of components as software systems' building blocks has been studied extensively in the area of software architectures [11,15]. In this paper, we focus on the components themselves. While there are many aspects of a software component worthy of careful study (e.g., modeling notations [2], implementation platforms [1], evolution mechanisms [9]), we restrict our study in this paper to an aspect only partially considered in existing literature: internal consistency among different models of a component. The direct motivation for this paper is our observation that there are four primary *functional* aspects of a software component: (1) *interface*, (2) *static behavior*, (3) *dynamic behavior*, and (4) *interaction protocol*. Each of these four aspects represents and helps to ensure different characteristics of a component. Moreover, the four aspects have complementary strengths and weaknesses. Existing approaches to component-based development typically select different subsets of these four aspects (e.g., interface and static behavior [9], or interface and interaction protocol [16]). At the same time, different approaches treat each individual aspect in very similar ways (e.g., modeling static behaviors via pre- and post-conditions, or modeling interaction protocols via finite state machines, or FSM).

The four aspects' complementary strengths and weaknesses, as well as their consistent treatment in literature suggest the possibility of using the four modeling aspects in concert. However, what is missing from this picture is an understanding of the different relationships among these different models in a single component. Figure 1 depicts the space of possible intra-component model relationship clusters. Each cluster represents a range of possible relationships, including not only "exact" matches, but also "relaxed" matches [17] between the models in question. Of these six clusters, only the pair-wise relationships between a component's interface and its other modeling aspects have been studied extensively (relationships *1*, *2*, and *3* in Figure 1).

This paper suggests an approach to completing the modeling space depicted in Figure 1. We discuss the extensions required to commonly used modeling approaches for each aspect in order to enable us to relate them and ensure their compatibility. We also discuss the advantages and drawbacks inherent in modeling all four aspects (referred to as the Quartet in the remainder of the paper) and six relationships shown in Figure 1. This paper represents a starting point in a much larger study. By addressing all the relationships shown in Figure 1 we eventually hope to accomplish several important long-term goals:

- enrich, and in some respects complete, the existing body of knowledge in component modeling and analysis,
- suggest constraints on and provide guidelines to practical modeling techniques, which typically select only a subset of the quartet,
- provide a basis for additional operations on components, such as retrieval, reuse, and interchange [17],
- suggest ways of creating one (possibly partial) model from another automatically, and
- provide better implementation generation capabilities from thus enriched system models.

The rest of the paper is organized as follows. Section 2 summarizes existing approaches to component modeling techniques and introduces the Quartet in more detail. Section 3 demonstrates our specific approach to component modeling using the Quartet and provides details of each modeling perspective. Section 4 discusses the relationships among the four modeling aspects shown in Figure 1 by identifying their interdependencies. Finally, Section 5 discusses our on-going research and future directions.

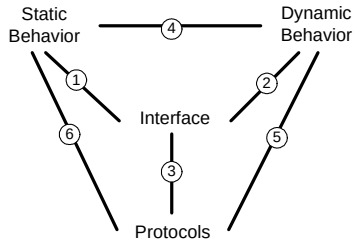

Figure 1. Model relationships within a software component.

2. COMPONENT MODELING

Modeling from multiple perspectives has been identified as an effective way to capture a variety of important properties of component-based software systems [2,3,6,8,10]. A well known example is UML, which employs nine diagrams (also called views) to model requirements, structural and behavioral design, deployment, and other aspects of a system. When several system aspects are modeled using different modeling views, inconsistencies may arise.

Ensuring consistency among heterogeneous models of a software system is a major software engineering challenge that has been studied in multiple approaches, with different foci. Due to space constraints, we discuss a small number of representative approaches here. [4] offers a model reconciliation technique particularly suited to requirements engineering. The assumption made by the technique is that the requirements specifications are captured formally. [5] also provide a formal solution to maintaining inter-model consistency, though more directly applicable at the software architectural level. One criticism that could be levied at these approaches is that their formality lessens the likelihood of their adoption. On the other hand, [7] provide more specialized approaches for maintaining consistency among UML diagrams. While their potential for wide adoption is aided by their focus on UML, these approaches may be ultimately harmed by UML's lack of formal semantics.

In this paper, we address similar problems to those cited above, but with a specific focus on multiple *functional* modeling aspects of a single software component. We advocate a four-level modeling technique (called *the Quartet*). Using the Quartet, a component's structural, behavioral (both static and dynamic), and interaction properties may be described and used in the analysis of a software system that encompasses the component:

1. *Interface* models specify the points by which a component interacts with other components in a system.
2. *Static behavior* models describe the functionality of a component discretely, i.e., at particular "snapshots" during the system's execution.
3. *Dynamic behavior* models provide a continuous view of *how* a component arrives at different states in its execution.
4. *Interaction protocol* models provide an *external* view of the component and its legal interactions with other components.

Typically, the static and dynamic component behaviors and interaction protocols are expressed in terms of a component's interface model (hence their positioning in Figure 1).

3. OUR APPROACH

The approach to component modeling we advocate is based on the concept of the Quartet discussed in the previous section: a complete functional model of a software component can be achieved only by focusing on all four aspects of the Quartet. At the same time, focusing on all four aspects has the potential to introduce certain problems (e.g., large number of modeling notations that developers have to master, model inconsistencies) that must be carefully addressed. While we use a particular notation in the discussion below, the approach is generic such that it can be easily adapted to other modeling notations. In this section, we focus on the conceptual elements of our approach, with limited focus on our specific notation used. Component models are specified from the following four modeling perspectives:[1]

```
Component_Model:
    (Interface, Static_Behavior,
     Dynamic_Behavior, Interaction_Protocol);
```

3.1. Interface

Interface modeling serves as the core of our component modeling approach and is extensively leveraged by other three modeling levels. An interface is specified in terms of several *interface elements*. Each interface element has a direction (+ or -), name (method signature), a set of input parameters, and possibly a return type (output parameter). The direction indicates whether the component *requires* (+) the service (i.e., operation) associated with the interface element or *provides* (-) it to the rest of the system. In other words:

```
Interface_Model: {Interface_Element};
Interface_Element:
    (Direction, Method_signature,
     {Input_parameter}, Output_parameter);
```

3.2. Static Behavior

We adopt a widely used approach to static modeling [9], which relies on first-order predicate logic to specify static behavioral properties of a component in terms of the component's *state variables*, the constraints associated with them (*invariants*), *interfaces* (as modeled in the interface model), *operations* (accessed via interfaces) and their corresponding *pre-* and *post-conditions*. In other words:

```
Static_Behaviors:
    ({State_variable}, Invariant, {Operation});
State_variable: (name, type);
Invariant: (logical_expression);
Operation:
    ({Interface_Element}, Pre_cond, Post_cond);
Pre/Post_cond: (logical_expression);
```

3.3. Dynamic Behavior

A dynamic behavior model provides a continuous view of the component's internal execution details. Variations of state-based modeling techniques have been typically used to model a component's internal behavior (e.g., in UML). Such approaches describe the component's dynamic behavior using a set of *sequencing constraints* that define legal ordering of the operations performed by the component. These operations may belong to one of two categories: (1) they may be directly related to the interfaces of the component as described in both interface and static behavioral models; or (2) they may be internal operations of the component (i.e., invisible to the rest of the system such as private methods in a UML class). To simplify our discussion, we only focus on the first case: publicly accessible operations. The second case may be reduced to the first one using the concept of *hierarchy* in StateCharts: internal operations may be abstracted away by building a higher-level state-machine that describes the dynamic behavior only in terms of the component's interfaces.

A dynamic model serves as a conceptual bridge between the component's protocol and static behavioral models. On the one hand, a dynamic model serves as a refinement of the static model as it further details a component's internal behavior. On the other hand, by leveraging a state-based notation, a dynamic model may be used to specify the sequence by which a component's operations get executed. Fully describing a component's dynamic behavior is essential in achieving two key objectives. First, it provides a rich model that can be used to perform sophisticated analysis and simulation of component behavior. Second, it can serve as an important intermediate level model to generate implementation level artifacts from architectural specification.

Existing approaches to dynamic behavior modeling employ an *abstract* notion of component state. These approaches treat states as entities of secondary importance, with the transitions between states playing the main role in behavioral modeling. Component states are often only specified by their name and set of incoming and outgoing transitions. We offer an extended notion of dynamic modeling that defines a state in terms of a set of variables maintained by the component and their associated invariants. These invariants constrain the values of and dependencies among the variables [14].

To summarize, our dynamic behavior model consists of a set of initial states and a sequence of guarded transitions from an origin to a destination state. Furthermore, a state is specified in terms of constraints it imposes over a set of a component's state

1. Concise formulations are used in this section to clarify our definitions and are not meant to serve as a formal specification of our model.

variables. In other words,

```
Dynamic_Behavior: (InitState,
   {State:(Direction)Transition->State});
State: (Name, Variables, Invariant);
Transition: (Label, {Parameter}, Guard);
Guard: (logical_expression);
```

3.4. Interaction Protocols

Finally, we adopt the widely used notation for specifying component interaction protocols, originally proposed in [11]. Finite state semantics are used to define valid sequences of invocations of component operations. Since interaction protocols are concerned with an "external" view of a component, valid sequences of invocations are specified irrespective of the component's internal state or the pre-conditions required for an operation's invocation. Our notation specifies protocols in terms of a set of initial states, and a sequence of transitions from an origin state to a destination.

```
Interaction_Protocol: (InitState,
   {State:(Direction)Transition->State});
State: (Name);
Transition: (Label, {Parameter});
```

4. RELATING COMPONENT MODELS

As previously discussed, modeling complex software systems from multiple perspectives is essential in capturing a multitude of structural, behavioral, and interaction properties of the system under development. The key issue however, is maintaining the consistency among these models [4,5,7]. We address the issue of consistency in the context of functional component modeling based on the Quartet technique.

In order to ensure the consistency among the models, their inter-relationships must be understood. Figure 1 depicts the conceptual relationships among these models. We categorize these relationships into two groups: syntactic and semantic. A *syntactic* relationship is one in which a model (re)uses the elements of another model directly and without the need for interpretation. For instance, interfaces and their input/output parameters (as specified in the interface model) are directly reused in the static behavior model of a component (relationship *1* in Figure 1). The same is true for relationships *2* and *3*, where the dynamic behavior and protocol models (re)use the names of the interface elements as transition labels in their respective FSMs.

Alternatively, a *semantic* relationship is one in which modeling elements are interpreted using the "meaning" of other elements. That is, specification of elements in one model *indirectly* affects the specification of elements in a different model. For instance, an operation's pre-condition in the static behavior model specifies the condition that must be satisfied in order for the operation to be executed. Similarly, in the dynamic behavior model, a transition's guard ensures that the transition will only be taken when the guard condition is satisfied. The relationship between a transition's guard in the dynamic behavior model and the corresponding operation's pre-condition in the static behavior model is semantic in nature: one must be interpreted in terms of the other (e.g., by establishing logical equivalence or implication) before their (in)consistency can be established. Examples of this type of relationship are relationships *4* and *5* in Figure 1.

In the remainder of this section we focus in more detail on the six relationships among the component model quartet depicted in Figure 1.

4.1. Interface vs. Other Models (Relationships *1, 2, 3*)

The interface model plays a central role in the design of other component models. Regardless of whether the goal of modeling is to design a component's interaction with the rest of the system or to model details of the component's internal behavior, interface models will be extensively leveraged.

When modeling a component's behaviors from a static perspective, the component's operations are specified in terms of interfaces through which they are accessed. As discussed in Section 3, an interface element specified in the interface model is mapped to an operation, which is further specified in terms of its pre- and post-conditions that must be satisfied, respectively, prior to and after the operation's invocation.

In the dynamic behavior and interaction protocol models, activations of transitions result in changes to the component's state. Activation of these transitions is caused by internal or external stimuli. Since invocation of component operations results in changes to the component's state, there is a relationship between these operations' invocations (accessed via interfaces) and the transitions' activations. The labels on these transitions (as defined in Section 3) directly relate to the interfaces captured in the interface model.

The relationship between the interface model and other models is syntactic in nature. The relationship is also unidirectional: all interface elements in an interface model may be leveraged in the dynamic and protocol models as transition labels; however, not all transition labels will necessarily relate to an interface element. Our (informal) discussion provides a conceptual view of this relationship and can be used as a framework to build automated analysis support to ensure the consistency among the interface and remaining three models within a component.

4.2. Static vs. Dynamic Behavior (Relationship *4*)

An important concept in relating static and dynamic behavior models is the notion of *state* in the dynamic model and its connection to the static specification of component's *state variables* and their associated *invariant*. Additionally, operation *pre- and post-conditions* in the static behavior model and *transition guard*s in the dynamic behavior model are semantically related. We have extensively studied these relationships in [13,14] and identified the ranges of all such possible relationships. The corresponding concepts in the two models may be equivalent, or they may be related by logical implication. Although their equivalence ensures their inter-consistency, in some cases equivalence may be too restrictive. A discussion of such cases is given below.

Transition Guard vs. Operation Pre-Condition. At any state in a component's dynamic behavior model, multiple outgoing transitions may share the same label, but with different guards on the label. In order to relate an operation's pre-condition in the static model to the guards on the corresponding transitions in the dynamic model, we define the *union guard (UG)* of a transition label at a given state:

$$UG = \bigvee_{i=1}^{n} G_i$$

where n is the number of outgoing transitions with the same label at a given state and G_i is the guard associated with the i^{th} transition.

Clearly, if UG is equivalent to its corresponding operation's pre-condition, the consistency at this level is achieved. However, if we consider the static behavior model to be an abstract specification of the component's semantics, the dynamic behavioral model becomes the concrete realization of those semantics. In that case, if UG is stronger than the corresponding operation's pre-condition, the operation may still be invoked safely: UG places bounds on the operation's (i.e., transition's) invocation, ensuring that the operation may never be invoked under circumstances that violate its pre-condition; in other words, UG should imply the corresponding operation's pre-condition.

State Invariant vs. Component Invariant. The state of a component in the static behavior specification is modeled using a set of state variables. The possible values of these variables are constrained by the *component's invariant*. Furthermore, a component's operations may modify the state variables' values, thus modifying the state of the component as a whole. The dynamic behavior model, in turn, specifies internal details of the component's states when the component's services are invoked. As described in Section 2, these states are defined using a name, a set of variables, and an invariant associated with these variables (called *state invariant*). It is crucial to define the states in the

dynamic behavior state machine in a manner consistent with the static specification of component's state and invariant.

Once again, an equivalence relation among these two elements may be too restrictive. In particular, if a state's invariant in the dynamic model is stronger than the component's invariant in the static model (i.e., state's invariant implies component's invariant), then the state is simply bounding the component's invariant, and does not permit for circumstances under which the component's invariant is violated. This relationship preserves the properties of the abstract specification (i.e., static model) in its concrete realization (i.e., dynamic model) and thus may be considered less restrictive than the equivalence. For more discussion on these, and a study of other possible relationships, see [14].

State Invariants vs. Operation Post-Condition. The final important relationship between a component's static and dynamic behavior models is that of an operation's post-condition and the invariant associated with the corresponding transition's destination state.

In the static behavior model, each operation's post-condition must hold true following the operation's invocation. In the dynamic behavior model, once a transition is taken, the state of the component changes from the transition's origin state to its destination state. Consequently, the state invariant constraining the destination state and the operation's post-condition are related. Again, the equivalence relationship may be unnecessarily restrictive. Analogously to the previous cases, if the invariant associated with a transition's destination state is stronger than the corresponding operation's post-condition (i.e., destination state's invariant implies the corresponding operation's post-condition), then the operation may still be invoked safely.

4.3. Dynamic Behavior vs. Protocol (Relationship *5*)

As previously mentioned, the relationship between the dynamic behavior and interaction protocol models of a component is semantic in nature: the concepts of the two models relate to each other in an indirect way.

As discussed in Section 3 we model a component's dynamic behavior by enhancing traditional FSMs with state invariants. Our approach to modeling interaction protocols also leverages FSMs to specify acceptable traces of execution of component services. The relationship between the dynamic behavior model and the interaction protocol model thus may be characterized in terms of the relationship between the two state machines. These two state machines are at different granularity levels however: the dynamic behavior model details the internal behavior of the component based on both internally- and externally-visible transitions, guards, and state invariants; on the other hand, the protocol model simply specifies the externally-visible behavior of the component, with an exclusive focus on transitions.

Our goal here is not to define a formal technique to ensure equivalence of two arbitrary state machines. This would first require some calibration on the models to even make them comparable. Additionally, several approaches have studied the equivalence of StateCharts [2,16]. Instead, we provide a more pragmatic approach to ensure the consistency of the two models. We consider the dynamic behavior model to be the concrete realization of the system under development, while the protocol of interaction provides a guideline for the correct execution sequence of the component's interfaces. Assuming that the interaction protocol model demonstrates *all* valid sequences of operations invocations of the component, the dynamic behavioral model should be designed such that any legal sequence of invocations of the component would also result in a legal execution of the component's dynamic behavior FSM. In other words, the dynamic behavioral model may be more general than the protocol of interactions; any execution trace obtained by the protocol model, must result in a legal execution of component's dynamic behavioral model.

4.4. Static Behavior vs. Protocol (Relationship *6*)

As discussed in Section 3.3, we consider the dynamic behavior model to be a bridge between a component's interaction protocol and static behavior specification models. The interaction protocol model specifies the valid sequence by which the component's interfaces may be accessed. In doing so, it fails to take into account the component's internal behavior (e.g., the pre-conditions that must be satisfied prior to an operation's invocation). Consequently, we believe that there is no direct conceptual relationship between the static behavior and interaction protocol models. Note, however, that the two models are related indirectly via a component's interface and dynamic behavior models.

5. CONCLUSION

In this paper, we argued for a four-level modeling approach, referred to as *the Quartet*, that can be used to model structural, static and dynamic behavioral, and interaction properties of a software component. We also discussed the conceptual dependencies among these models and highlighted specific points at which consistency among them must be established. While it may be argued that practitioners will be reluctant to use our approach in "real" development situations because it requires too much rigor and familiarity with too many notations, we believe such a criticism to be misplaced: the experience of UML has shown that practitioners will be all too happy to adopt multiple notations if those notations solve important problems. It should also be noted that our approach allows developers to select whatever subset of the Quartet they wish, but gives them an understanding of how incorporating additional component aspects is likely to impact their existing models.

6. REFERENCES

[1] Allen R., Garlan D., "A formal basis for architectural connection", *ACM TOSEM*, 6(3):213–249, 1997.

[2] Booch G., Jacobson I., Rumbaugh J. "*The Unified Modeling Language User Guide*", Addison-Wesley, Reading, MA.

[3] Farías A., Südholt M., "On Components with Explicit Protocols Satisfying a Notion of Correctness by Construction", in *Confederated Int'l Conf. CoopIS/DOA/ODBASE* 2002.

[4] Finkelstein A., et al., "Inconsistency Handling in Multi-Perspective Specifications", *IEEE TSE*, August 1994.

[5] Fradet P., et al., "Consistency checking for multiple view software architectures", in *ESEC/FSE 1999*.

[6] Hofmeister C., et al., "Describing Software Architecture with UML," in *WICSA1*, San Antonio, TX, February 1999.

[7] Hnatkowska B., et al., "Consistency Checking in UML Models", in *4th Int'l Conf. on Information System Modeling (ISM01)*, Czech Republic, 2001.

[8] Krutchen, P.B. "The 4+1 View Model of Architecture", *IEEE Software 12*, pp. 42 - 50, 1995.

[9] Liskov B. H., Wing J. M., "A Behavioral Notion of Subtyping", *ACM TOSEM*, November 1994.

[10] Nuseibeh B., et al., "Expressing the Relationships Between Multiple Views in Requirements Specification", in *ICSE-15*, Baltimore, Maryland, 1993.

[11] Perry D.E., and Wolf A.L., "Foundations for the Study of Software Architectures", ACM SIGSOFT Software Engineering Notes, 17(4): 40-52, October 1992.

[12] Plasil F., Visnovsky S., "Behavior Protocols for Software Components", *IEEE TSE*, November 2002.

[13] Roshandel R., Medvidovic N., "Coupling Static and Dynamic Semantics in an Architecture Description Language", in *Working Conf. on Complex and Dynamic Systems Architectures*, Brisbane, Australia, December 2001.

[14] Roshandel R., Medvidovic N., "Relating Software Component Models", *Tech Rep't USC-CSE-2003-504*, March 2003.

[15] Shaw M., Garlan D., "Software Architecture: Perspectives on an Emerging Discipline", Prentice-Hall, 1996.

[16] Yellin D.M., Strom R.E., "Protocol Specifications and Component Adaptors," *ACM TOPLAS*, vol. 19, no. 2, 1997.

[17] Zaremski A.M., Wing J.M., "Specification Matching of Software Components", *ACM TOSEM*, vol. 6, no. 4, 1997.