

Multi-View Software Component Modeling for Dependability

Roshanak Roshandel and Nenad Medvidovic

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

Abstract. Modeling software components from multiple perspectives provides complementary views of a software system and enables sophisticated analyses of its functionality. A software component is traditionally 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 pairs of aspects (e.g., interfaces and their associated pre- and post-conditions in static behavior models). We advocate that, in order to accrue the true benefits of component-based software development, one needs to model all four aspects of components. In such a case, ensuring the consistency among the multiple views becomes critical. We offer an approach to modeling components using a four-view perspective (called the Quartet) and identify the points at which the consistency among the models must be maintained. We outline the range of possible intra-component, inter-model relationships, discuss their respective advantages and drawbacks, and motivate the specific choices we have made in our on-going research on ensuring the dependability of software systems from an architectural perspective.

Keywords: Software architecture, software component, dependability, reliability, the Quartet

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., dependability attributes such as

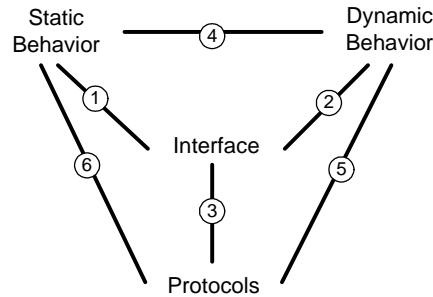


Figure 1. Software Model Relationships within a Component.

correctness, compatibility, interchangeability, and functional reliability), 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 [18,23,30]. 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 [6,18], implementation platforms [1,3], evolution mechanisms [15,17]), we restrict our study to an aspect of dependability only partially considered in existing literature: internal consistency among different models of a component. We consider this aspect from an architectural modeling perspective, as opposed to an implementation or runtime perspective.

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 the four modeling aspects represents and helps to ensure different characteristics of a component. Moreover, the four aspects have complementary strengths and weaknesses. As detailed in Section 3, existing approaches to component-based development typically select different subsets of these four aspects (e.g., interface and static behavior [15], or interface and interaction protocol [31]). 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 [32] 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 comprehensive functional basis for quantifying software dependability attributes such as architectural reliability,
- provide a basis for additional operations on components, such as retrieval, reuse, and interchange [32],
- 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. In Section 2 we introduce a simple example that will be used throughout the paper to clarify concepts. Section 3 provides an overview of existing approaches to component modeling techniques and introduces the Quartet in more detail. Section 4 demonstrates our specific approach to component modeling using the Quartet and provides details of each modeling perspective. Section 5 discusses the relationships among the four modeling aspects shown in Figure 1 by identifying their interdependencies. In Section 6 we discuss the implications of our approach on dependability of software systems. Finally, Section 7 discusses our on-going research and future directions.

2 Example

Throughout this paper, we use a simple example of a robotic rover to illustrate the introduced concepts. The robotic rover, called SCROver, is designed and developed in collaboration with NASA’s Jet Propulsion Laboratory, and in accordance with their Mission Data System (MDS) methodology. To avoid unnecessary complexity, we discuss a simplified version of the application. In this paper, we focus on SCROver’s “wall following” behavior. In this mode, the rover uses a laser rangefinder to determine the distance to the wall, drives forward while maintaining a fixed distance from that wall, and turns both inside and outside corners when it encounters them. This scenario also involves sensing and controlled locomotion, including reducing speed when approaching obstacles.

The system contains five main components: *controller*, *estimator*, *sensor*, *actuator*, and a *database*. The sensor component gathers physical data (e.g., distance from the wall) from the environment. The estimator component accesses the data and passes them to the controller for control decisions. The controller component issues

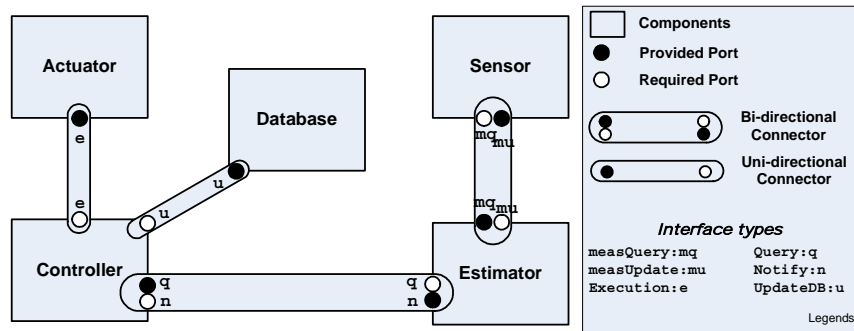


Figure 2. SCROver's Software Architecture.

commands to the actuator to change the direction or speed of the rover. The database component stores the “state” of the rover at certain intervals, as well as when a change in the values happens. Figure 2 shows a high-level architecture of the system in terms of the constituent components, connectors, and their associated interfaces (ports): the rectangular boxes represent components in the system; the ovals are connectors; the dark circles on a component correspond to interfaces of services *provided* by the component/connector, while the light circles represent interfaces of services *required* by the component. To illustrate our approach, we will specifically define the static and dynamic behavioral models and protocols of interactions for the Controller component in Section 4.

3 Component Modeling

Building good models of complex software systems in terms of their constituent components is an important step in realizing the goals of architecture-based software development [18]. Effective architectural modeling should provide a good view of the structural and compositional aspects of a system; it should also detail the system's behavior. Modeling from multiple perspectives has been identified as an effective way to capture a variety of important properties of component-based software systems [6,7,12,14,20]. 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. In [10] a model reconciliation technique particularly suited to requirements engineering is offered. The assumption made by the technique is that the requirements specifications are captured formally. In [5,11] a formal solution to maintaining inter-model consistency is presented that is 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, in [8,13] more specialized approaches for maintaining consistency among UML diagrams are offered. 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-view 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. The Quartet also has the potential to provide a basis for generating implementation-level artifacts from component models. In the rest of this section, we will first motivate the four component aspects. We will then discuss existing techniques for modeling each aspect. We will use this discussion as the basis for studying the dependencies among these models and implications of maintaining their interconsistency in Sections 4 and 5.

3.1 Introducing the “Quartet”

Traditionally, functional characteristics of software components have been modeled predominantly from the following four perspectives:

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 throughout its execution.
4. *Interaction protocol* models provide an *external* view of the component and how it may legally interact with other components in the system.

3.2 Existing Approaches to the Quartet

Interface modeling. Component modeling has been most frequently performed at the level of interfaces. This has included matching interface names and associated input/output parameter types. Component interface modeling has become routine, spanning modern programming languages, interface definition languages (IDLs) [19,21], architecture description languages (ADLs) [18], and general-purpose modeling notations such as UML. However, software modeling solely at this level does not guarantee many important properties, such as interoperability or substitutability of components: two components may associate vastly different meanings with identical interfaces.

Static Behavior Modeling. Several approaches have extended interface modeling with static behavioral semantics [1,15,22,32]. Such approaches describe the behavioral

properties of a system at specific snapshots in the system's execution. This is done primarily using invariants on the component states and pre- and post-conditions associated with the components' operations. Static behavioral specification techniques are successful at describing what the state of a component should be at specific points of time. However, they are not expressive enough to represent *how* the component arrives at a given state.

Dynamic Behavior Modeling. The deficiencies associated with static behavior modeling have led to a third group of component modeling techniques and notations. Modeling dynamic component behavior results in a more detailed view of the component and *how* it arrives at certain states during its execution. It provides a continuous view of the component's internal execution details. While this view of component modeling has not been practiced as widely as interface or static behavior modeling, there are several notable examples of it. For instance, UML has adopted a StateChart-based technique to model the dynamic behaviors of its conceptual components (i.e., Classes). Other variations of state-based techniques (e.g., FSM) have been used for similar purposes (e.g., [9]). Finally, Wright [3] uses CSP to model dynamic behaviors of its components and connectors.

Interaction Protocol Modeling. The last category of component modeling approaches focuses on legal protocols of interaction among components. This view of modeling provides a continuous *external* view of a component's execution by specifying the allowed execution traces of its operations (accessed via interfaces). Several techniques for specifying interaction protocols have been developed. These techniques are based on CSP [3], FSM [31], temporal logic [1], and regular languages [24]. They often focus on detailed formal models of the interaction protocols and enable proofs of protocol properties. However, some may not scale very well, while others may be too formal and complex for routine use by practitioners.

Typically, the static and dynamic component behaviors and interaction protocols are expressed in terms of a component's interface model. For instance, at the level of static behavior modeling, the pre- and post-conditions of an operation are tied to the specific interface through which the operation is accessed. Similarly, the widely adopted protocol modeling approach [31] uses finite-state machines in which component interfaces serve as labels on the transitions. The same is also true of UML's use of interfaces specified in class diagrams for modeling event/action pairs in the corresponding StateCharts model. This is why we chose to place *Interface* at the center of the diagram shown in Figure 1.

4 Our Approach

The approach to component modeling we advocate is based on the concept of the Quartet introduced in Section 3. We argue that 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);
```

4.1 Interface

Interface modeling serves as the core of our component modeling approach and is extensively leveraged by the other three modeling views. A component's interface has a type and 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:  
(Type,  
  {Interface_Element});  
  
Interface_Element:  
(Direction,  
  Method_signature,  
  {Input_parameter},  
  Output_parameter);
```

In the context of the SCRover example discussed in Section 2, the *Controller* component exposes four interfaces through its four ports: *e*, *u*, *q*, and *n*, correspond to *Execution*, *UpdateDB*, *Query*, and *Notification* interface types, respectively (recall Figure 2). Each of these interfaces may have several interface elements associated with it. These interface elements are enumerated in Figure 3, for instance

```
Controller:  
u: -updateSpeed(val: SpeedType):Boolean;  
q: +getWallDist():DistanceType;
```

where *updateSpeed()* is an interface element required by the *Controller* component. Its input parameter *val* is of user-defined type *SpeedType* and returns a *Boolean* indicating a successful change of speed operation. On the other hand, *getWallDist()* is provided by the controller component, has no parameters, and returns a value of type *DistanceType*.

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

4.2 Static Behavior

We adopt a widely used approach for static behavior modeling [15], which relies on first-order predicate logic to specify functional properties of a component in terms of the component's *state variables*, *invariants* (the constraints associated with the state variables), *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);
```

A partial static behavior specification of SCROver example's *Controller* component is depicted in Figure 3.

4.3 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 model of interaction protocols and its static behavioral model. On the one hand, a dynamic model serves as a refinement of the static model as it further details the internal behavior of the component. 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

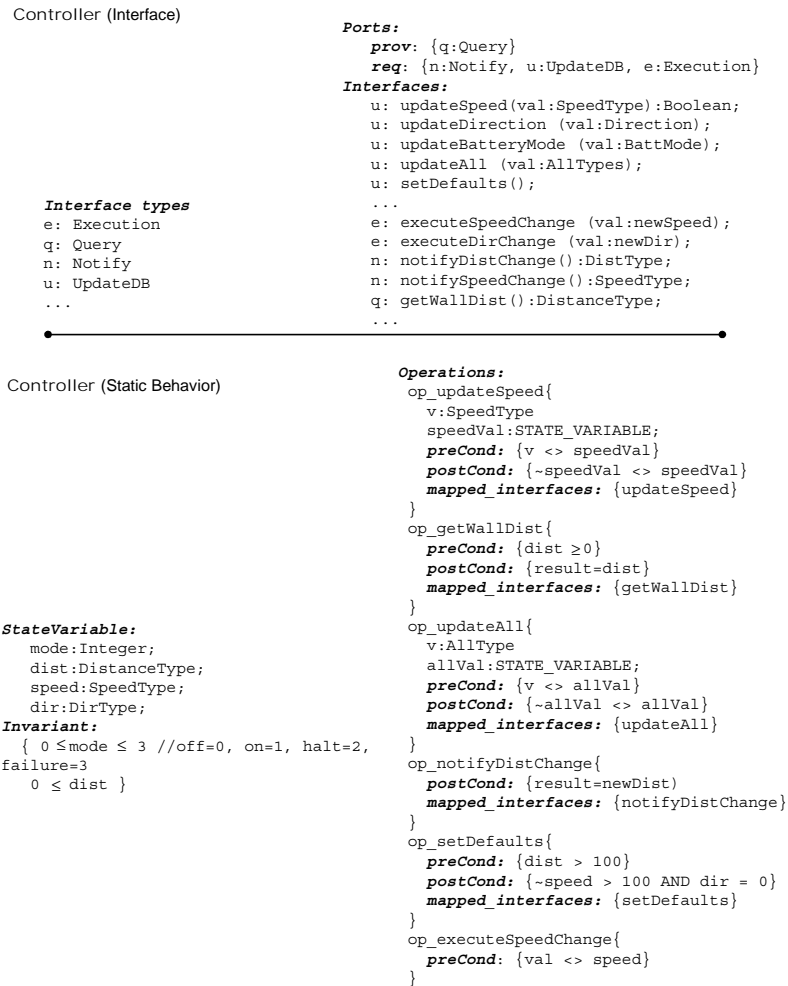
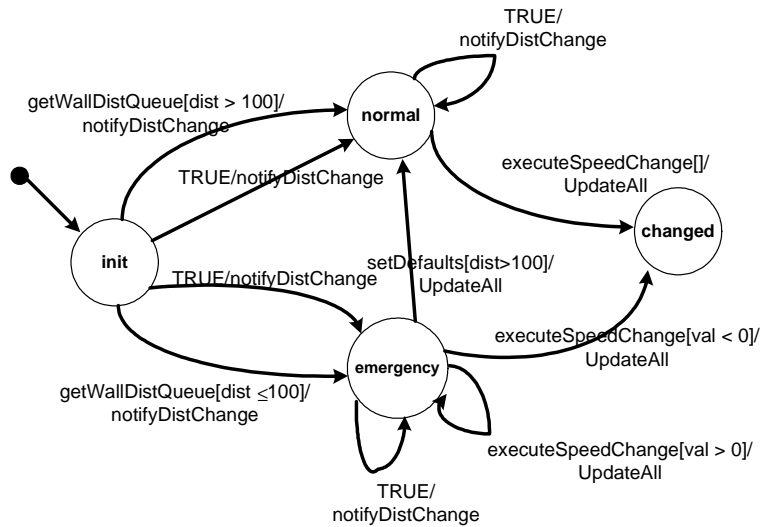


Figure 3. Partial View of the *Controller* component's Interface and Static Behavior.

in achieving two key objectives. First, it provides a rich model that can be used to perform sophisticated analysis and simulation of the component's behavior. Second, it can serve as an important intermediate level model to generate implementation level artifacts from the 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



Controller (Interaction Protocols)

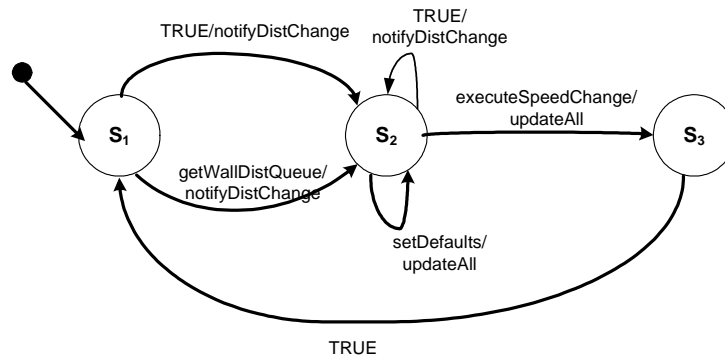


Figure 4. Partial View of Controller Component's Dynamic Behavior (top) and Interaction Protocol (bottom)

invariants. These invariants constrain the values of and dependencies among the variables [28]. As examples of this extension to the definition of states consider the invariants associated with *Normal* and *Emergency* states in the top diagram of Figure 4 (the details of state invariants are actually omitted from the diagram itself for clarity).

$Normal_{inv}: (0 \leq dir < 360) \text{ AND } (0 < speed < 500)$
 $Emergency_{inv}: (100 < speed < 200) \text{ AND } (dir = 0) \text{ AND } (dist \leq 100)$

In summary, 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 variables. In other words,

```

Dynamic_Behavior:
  (InitState,
   {State: (Direction)Transition->State});

State:
  (Name,
   Variables,
   Invariant);

Transition:
  (Label,
   {Parameter},
   Guard);

Guard:
  (Logical_Expression);

```

A model of the *Controller* component's dynamic behavior is depicted in the middle portion of Figure 4. Note that the transitions are decorated with the StateCharts' event/action pairs. We omit further discussion of actions in this paper since their usage is not critical to the paper's argument.

4.4 Interaction Protocols

Finally, we adopt the widely used notation for specifying component interaction protocols, originally proposed in [23]. 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. A simple interaction protocol for the *Controller* component is shown in the bottom portion of Figure 4. 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});

```

5 Relating Component Models

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 requirement for ensuring that dependable systems will result from this process is to maintain the consistency among the different system models [5,8,10,11,13]. 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 designed using the “meaning” and interpretation of the 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.

5.1 Relationships 1, 2, and 3 — Interface vs. Other Models

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 4, 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 4) 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.

5.2 Relationship 4 — Static Behavior vs. Dynamic Behavior

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 guards* in the dynamic behavior model are semantically related. We have extensively studied these relationships in [28] 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 would ensure 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 given 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 first define the *union guard (UG)* of a transition label at a given state. UG is the disjunction of all guards G associated with outgoing transitions that carry the same label:

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

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.

As an example in Figure 4, the dynamic model is designed such that different states (*Normal* and *Emergency*) are going to be reachable as destinations of the *getWallDist()* transition depending on the distance of the encountered obstacle (*dist* variable in the transition guards). In this case at state *Normal* we have:

$$UG_{\text{getWallDist}} = (\text{dist} > 100) \text{ OR } (0 < \text{dist} \leq 100)$$

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 functionality, the dynamic behavioral model becomes the concrete realization of that functionality. In that case, if UG is stronger than the corresponding operation’s pre-condition, the operation may still be invoked safely. The reason for this is that UG places bounds on the operation’s (i.e., transition’s) invocation, ensuring that the operation will never be invoked under circumstances that violate its pre-condition; in other words, UG should imply the

corresponding operation's pre-condition. This is the case for the *getWallDist()* operation in the rover's *Controller* component.

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 3, these states are defined using a name, a set of variables, and an invariant associated with these variables (called *state's invariant*). It is crucial to define the states in the dynamic behavior state machine in a manner consistent with the static specification of component 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 equivalence. A simple case is that of state *Normal* and its invariant in the *Controller* component. Relating the component invariant and the invariant of state *Normal*, we have:

$$\begin{aligned} \text{Normal}_{\text{inv}} &: (0 \leq \text{dir} < 360) \text{ AND } (0 < \text{speed} < 500) \text{ AND } (\text{dist} > 100) \\ \text{Controller}_{\text{inv}} &: (0 \leq \text{dir} < 360) \text{ AND } (0 < \text{speed} < 1000) \text{ AND } (\text{dist} \geq 0); \\ \text{Normal}_{\text{inv}} &\Rightarrow \text{Controller}_{\text{inv}} \end{aligned}$$

For more discussion on these, and a study of other possible relationships, see [28].

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. For example, in Figure 3, the post-condition of the *setDefault()* operation is specified as:

$$\text{setDefault}_{\text{post}}: (\text{speed} > 100) \text{ AND } (\text{dir} = 0)$$

while state *Normal* is a destination state for *setDefault()* and we have:

$$\text{Normal}_{\text{inv}}: (0 \leq \text{dir} < 360) \text{ AND } (0 < \text{speed} < 500) \text{ AND } (\text{dist} > 100)$$

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. As an example consider the specification of state *Normal* and operation *setDefault()* shown above. Clearly, the appropriate implication relationship does not exist. The *setDefault()* operation may assign the value of the variable *speed* to be greater than 500, which would, in turn, negatively affect the component's dependability.

5.3 Relationship 5 — Dynamic Behavior vs. Interaction Protocols

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. Examples of the two models are shown in Figure 4.²

Our goal here is not to define a formal technique to ensure the equivalence of two arbitrary state machines. This would first require some calibration of the models to even make them comparable. Additionally, several approaches have studied the equivalence of StateCharts [4,16,31]. 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. For example, consider again models of the *Controller* component specified in Figure 4. Assuming that the interaction protocol model demonstrates *all* the valid sequences of operation invocations of the component, it can be deduced that the multiple consecutive invocation of *setDefault()* are permitted. However, based on the dynamic model, only one such operation is possible. Consequently, the dynamic and protocol models are not equivalent. Since, the *Controller* component's dynamic behavior FSM is less general than its protocol FSM, some legal sequences of invocations of the component are not permitted by the component's dynamic behavior FSM. This, in turn, directly impacts the component's dependability.

² Recall that in our example, the dynamic model is only specified at the level of external operations. Adding internal operations to the model increases the granularity gap between the dynamic and protocol models.

5.4 Relationship 6 — Static Behavior vs. Interaction Protocol

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.

6 Implications of the Quartet on system-level reliability

The goal of our work is to support modeling architectural aspects of complex software systems from multiple perspectives and to ensure the inter- and intra-consistency among these models. Such consistency is critical in building dependable software systems, where complex components interact to achieve a desired functionality. Dependability attributes must therefore be “built into” the software system throughout the development process, including during the architectural design phase. This paper has presented a first step in this direction. We have restricted our focus in the paper to software components alone. However, the Quartet enables a natural progression toward system-wide modeling and analysis: a configuration of software components in a system's architecture is “tied together” via the components' interface and interaction protocol models; the remaining two models of each component are abstracted away from other components.

Several existing approaches (e.g., [3,31]) have demonstrated how protocols can be leveraged to assess a system's architecture-level (functional) consistency. This paper is part of our ongoing research in quantifying, measuring, and consequently ensuring architecture-level dependability of software systems. Specifically, our ongoing work on architecture-level reliability estimation of software systems directly leverages the analyses results obtained from the Quartet modeling, and applies those results to a stochastic model to estimate component-level reliability [25,26]. Our stochastic reliability model is applicable to early stages of development when the implementation artifacts are unavailable and the exact execution profile of the system is unknown. Furthermore, we use these results to estimate the overall system-level reliability in a compositional manner. A series of concurrent state machines, each representing an individual component's interaction protocol, are used to build an augmented Markov model. We believe that our approach could be directly applied to other dependability attributes that are probabilistic in nature (e.g., availability).

7 Conclusion and future work

In this paper, we argued for a four-view modeling approach, referred to as *the Quartet*, that can be used to comprehensively 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 and maintained in the interest of ensuring the component's dependability. While the discussion provided in this paper has been semi-formal, we believe that it provides a promising starting point for formalization and development of related modeling and analysis tools. Such tools are likely to alleviate a common criticism levied at an approach such as ours: practitioners will be reluctant to use it in "real" development situations because it requires too much rigor and familiarity with too many notations. We believe such a criticism to be misplaced for several reasons. First, this approach does not require formality and rigor beyond what an average computer science undergraduate must become familiar with: FSM and first-order predicate logic. Second, the experience of UML has shown that practitioners will be all too happy to adopt multiple notations if those notations solve important problems. Third, the potential for automated system specification, model inter-consistency analysis, and implementation in our view should outweigh any perceptions of difficulty in adopting the Quartet. Finally, this approach still allows developers to select whatever subset of the Quartet they wish, but should give them an understanding of how incorporating additional component aspects is likely to impact their existing models. In fact, we have recently begun using three of the four Quartet models (interface, static behavior, and protocol) to calculate system reliability at the architectural level [25,26].

The ideas discussed in this paper were inspired by our previous work, in which we integrated support for modeling static and dynamic system behaviors at the architectural level [27]. We are currently reengineering and extending this tool support using an extensible XML-based ADL [29]. Our long-term goal is to use this tool support in formalizing the component modeling framework introduced in this paper, and to provide an extensible basis for modeling and ensuring the consistency among interacting and interchangeable components in large-scale software systems.

8 References

1. Aguirre N., Maibaum T.S.E., "A Temporal Logic Approach to Component Based System Specification and Reasoning", in *Proceedings of the 5th ICSE Workshop on Component-Based Software Engineering*, Orlando, FL, 2002.
2. America P. "Designing an Object-Oriented Programming Language with Behavioral Subtyping", *Lecture Notes in Computer Science*, vol. 489, Springer-Verlag, 1991.
3. Allen R., Garlan D., "A Formal Basis for Architectural Connection", *ACM Transactions on Software Engineering and Methodology*, 6(3):213-249, 1997.
4. Ashar P., Gupta A., Malik S., "Using complete-1-distinguishability for FSM equivalence checking", *ACM Transactions on Design Automation of Electronic Systems*, Vol. 6, No. 4, pp 569-590, October 2001.
5. Balzer R., "Tolerating Inconsistency", in *Proceedings of 13th International Conference on Software Engineering (ICSE-13)*, Austin, Texas, 1991.
6. Booch G., Jacobson I., Rumbaugh J. "*The Unified Modeling Language User Guide*", Addison-Wesley, Reading, MA.

7. Dias M., Vieira M., "Software Architecture Analysis based on Statechart Semantics", in *Proceedings of the 10th International Workshop on Software Specification and Design, FSE-8*, San Diego, USA, November 2000.
8. Egyed, A. "Scalable Consistency Checking between Diagrams - The ViewIntegra Approach," in *Proceedings of the 16th IEEE International Conference on Automated Software Engineering*, San Diego, CA, 2001.
9. Farias A., Südholt M., "On Components with Explicit Protocols Satisfying a Notion of Correctness by Construction". in *Proceedings of Confederated International Conferences CoopIS/DOA/ODBASE 2002*.
10. Finkelstein A., Gabbay D., Hunter A., Kramer J., and Nuseibeh B., "Inconsistency Handling in Multi-Perspective Specifications", *IEEE Transactions on Software Engineering*, 20(8): 569-578, August 1994.
11. Fradet P., Le Métayer D., Périn M., "Consistency checking for multiple view software architectures", in *Proceeding of ESEC / SIGSOFT FSE 1999*.
12. Hofmeister C., Nord R.L., and Soni D., "Describing Software Architecture with UML" In *Proceedings of the TC2 First Working IFIP Conference on Software Architecture (WICSA1)*, San Antonio, TX, February 22-24, 1999.
13. Hnatkowska B., Huzar Z., Magott J., "Consistency Checking in UML Models", in *Proceedings of Fourth International Conference on Information System Modeling (ISM01)*, Czech Republic, 2001.
14. Krutchen, P.B. "The 4+1 View Model of Architecture", *IEEE Software* 12, pp. 42 - 50, 1995.
15. Liskov B. H., Wing J. M., "A Behavioral Notion of Subtyping", *ACM Transactions on Programming Languages and Systems*, November 1994.
16. Maggiolo-Schettini A., Peron A., and Tini S., "Equivalence of Statecharts", In *Proceedings of CONCUR '96*, Springer, Berlin, 1996
17. 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*, Los Angeles, CA, May 16-22, 1999.
18. Medvidovic N., Taylor R.N., "A Classification and Comparison Framework for Software Architecture Description Languages", *IEEE Transactions on Software Engineering*, January 2000.
19. Microsoft Developer Network Library, *Common Object Model Specification*, Microsoft Corporation, 1996.
20. Nuseibeh B., Kramer J., and Finkelstein A., "Expressing the Relationships Between Multiple Views in Requirements Specification", in *Proceedings of the 15th International Conference on Software Engineering (ICSE-15)*, Baltimore, Maryland, USA, 1993.
21. Object Management Group, *The Common Object Request Broker: Architecture and Specification*, Document Number 91.12.1, OMG, December 1991.
22. *The Object Constraint Language (OCL)*, <http://www-3.ibm.com/software/ad/library/standards/ocl.html>.
23. 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.
24. Plasil F., Visnovsky S., "Behavior Protocols for Software Components", *IEEE Transactions on Software Engineering* 28(11), pp. 1056-1076, November 2002.
25. Roshandel R., "Calculating Architectural Reliability via Modeling and Analysis" (Qualifying Exam Report), *USC Technical Report Number USC-CSE-2003-516*, December 2003.
26. Roshandel R., "Calculating Architectural Reliability via Modeling and Analysis", In the *proceedings of the Doctoral Symposium of the 26th International Conference on Software Engineering*, (to appear), Scotland, UK, 2004.

27. Roshandel R., Medvidovic N., "Coupling Static and Dynamic Semantics in an Architecture Description Language", in *Proceeding of Working Conference on Complex and Dynamic Systems Architectures*, Brisbane, Australia, December 2001.
28. Roshandel R., Medvidovic N., "Relating Software Component Models", *USC Technical Report Number USC-CSE-2003-504*, March 2003.
29. Roshandel R., van der Hoek A., Mikic-Rakic M., Medvidovic N., "Mae - A System Model and Environment for Managing Architectural Evolution", Submitted to *ACM Transactions on Software Engineering and Methodology (In review)*, October 2002.
30. Shaw M., Garlan D., "Software Architecture: Perspectives on an Emerging Discipline". Prentice-Hall, 1996.
31. Yellin D.M., Strom R.E., "Protocol Specifications and Component Adaptors," *ACM Transactions on Programming Languages and Systems*, Vol. 19, No. 2, 1997.
32. Zaremski A.M., Wing J.M., "Specification Matching of Software Components", *ACM Transactions on Software Engineering and Methodology*, 6(4):333-369, 1997.