

# Relating Software Component Models

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 at one or more of four levels: interface, static behavior, dynamic behavior, and interaction protocol. Each of these levels helps to ensure different aspects of component compatibility and interoperability. Existing approaches to component modeling have either focused on only one of the levels (e.g., interfaces in various IDLs) or on well-understood combinations of two of the levels (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 components at all four levels. Before that can be possible, one needs to understand the relationships among the different models. We detail one such pair-wise relationship—between static and dynamic component models—and draw parallels between it and the remaining models.

## 1. INTRODUCTION AND MOTIVATION

In recent years, component-based software engineering has emerged as an important discipline for developing large and complex software systems. In order to reduce software development costs, researchers and practitioners have proposed various solutions including reuse of off-the-shelf components and component evolution. Employing such techniques requires architectural modeling and analysis, to ensure compatibility between interacting or interchangeable components. There are four primary levels at which software components are modeled: (1) interfaces, (2) static behaviors, (3) dynamic behaviors, and (4) interaction protocols. Each of these levels helps to ensure different aspects of component compatibility and interoperability.

Traditionally, component modeling has been 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) [10,11], architecture description languages (ADLs) [9], and general-purpose modeling notations such as UML. However, software modeling solely at this level does not guarantee interoperability or substitutability of components; two components may associate vastly different meanings with identical interfaces.

Several approaches have extended interface modeling with *static behavioral* semantics [4,8,13,15]. 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 in 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.

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 level of component modeling has not been practiced as widely as interface or static behavioral modeling, there are several notable examples of it. 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) has been used for a similar purpose (e.g., [5]). Finally, Wright [2] uses CSP to model dynamic behaviors of its components and connectors.

The last category of component modeling approaches focuses on *interaction protocols* among components. This level 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 [2], FSM [14], temporal logic [1], and regular languages [12]. 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 the 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 [14] uses finite-state machines in which component interfaces serve as labels on the transitions. The same is also true of UML's StateCharts and its use of interfaces specified in the class diagrams for modeling events/actions on the state diagram model.

The range of possible relationships between the interface and static behaviors of components is well understood and discussed extensively in [15]. However, no analogous pair-wise connections have been made for static behavior, dynamic behavior, and protocol models of a component. This lack of proper understanding of the relationships *between* the different levels of component specification may result in subtle errors that may be introduced between component models, while each *individual* model remains syntactically and semantically correct.

In this paper, we provide a set of criteria for establishing the relationship between the static and dynamic behavioral models of a component. We enrich an existing approach to modeling dynamic component behaviors, such that a relationship between the two models may be discussed and established. We present an extensive study of the possible relationships and illustrate those relationships with a simple spacecraft controller example. We also analyze the possible impact of each identified relationship on the inter-consistency

of the two models. In order to complete this picture, an analogous study would need to be completed for the relationships between the static behavioral models and interaction protocols, as well as the dynamic behavioral models and protocols. That study is the subject of our on-going work, but is beyond the scope of this paper. Even though without these relationships the full consistency among the four modeling levels cannot be established, the work presented in this paper is a step toward this goal. Our work also provides a basis for developing rich behavioral models of the system under development, which, in turn, results in better implementation-generation capabilities from system models.

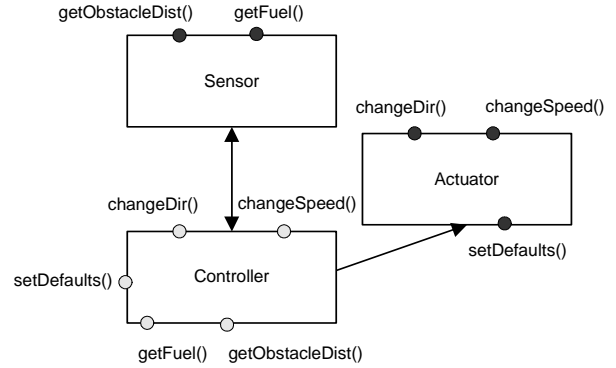
The rest of the paper is organized as follows: Section 2 introduces an example that we use throughout this paper. Our adopted notation for describing static and dynamic behavioral models is described in Section . The relationship between a component's static and dynamic behavioral model is discussed in Section 4. We discuss our on-going and future work in Section 5.

## 2. EXAMPLE

We use a simple example throughout this paper, to illustrate the concepts. The example is a space craft controller system that contains three components: *sensor*, *controller*, and *actuator*. The sensor component gathers physical data (e.g., remaining fuel, distance to obstacles) from the environment. The controller component accesses the data and, based on its value, issues commands to the actuator to change the direction or speed of the spacecraft. Figure 1 briefly shows three views of this system. First, a high-level architecture is depicted in terms of the constituent components and their associated interfaces; the rectangular boxes represent components in the system; the dark circles on the component correspond to *provided* interface elements while the light circles represent *required* interface elements. We then show a static behavioral specification of the *controller* component in terms of its state variables, the associated invariants and operation pre- and post-conditions. As an example, consider the *changeSpeed* interface element. The pre-condition of its corresponding operation requires the distance of the spacecraft from an obstacle to be less than 100 unit. ( $obstacleDist < 100$ ) before it may be invoked, and its post-condition indicates that the value of *speed* has changed ( $speed'$  denotes the new value of *speed*) and that 20 units of fuel is required for this operation; thus the amount of fuel is reduced by 20. Based on this static behavioral model, it is also clear that a *changeDir* operation is invoked when the spacecraft's distance from an obstacle is less than 300, but more than 100 units. Additionally, a *setDefault* operation may be invoked when there is no obstacle within the threshold of 300 units. Finally, a state-based model of the dynamic behavior of the system is presented. The labels on state transitions correspond to the component's interface elements. Throughout the figure, details of interface's input/output parameters are omitted for brevity.

## 3. ENRICHING COMPONENT MODELS

Modeling components at the levels of interfaces, static and dynamic behaviors, and protocols provides four distinct, complementary views on the system under development. Establishing the relationship among the four models is key for the system modeling to be effective and provide a useful basis for architectural analysis. The static behaviors specified using a component's invariants and operation pre- and post-conditions typically directly leverage the interface model. The range of possible relationships between the interface and behavior models is extensively studied by Zaremski and Wing [15]. This range is addressed in the context of substitutability of a



Spacecraft Controller System

### Controller (Static Behaviors)

```

StateVar:
{speed:Double;
 obstacleDist: Double;
 fuelLeft: Double;
 dir: Double}
Invariants:
(0 < obstacleDist) AND (0 ≤ dir < 360) AND
(0 < speed < 500) AND (0 < fuelLeft < 4000);
Operations:
getObstacleDist.Pre=()
getObstacleDist.Post=
(obstacleDist' <> obstacleDist)
changeDir.Pre= (100 ≤ obstacleDist < 300)
changeDir.Post=(dir' <> dir AND
fuelLeft' = fuelLeft - 10)
changeSpeed.Pre=
(0 < obstacleDist < 100)
changeSpeed.Post=(speed' <> speed AND
fuelLeft' = fuelLeft - 20)
setDefault.Pre=(obstacleDist ≥ 300)
setDefault.Post=(speed' > 100 AND dir'= 0)

```

### Controller (Dynamic Behaviors)

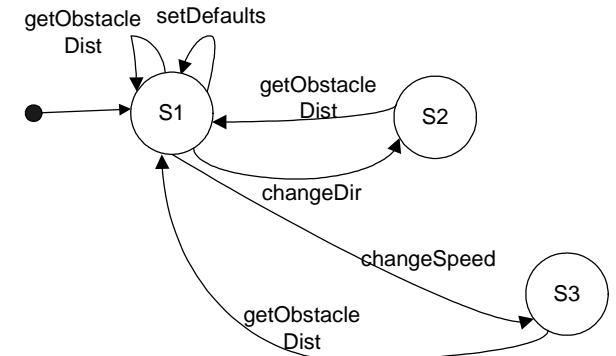


Figure 1. Spacecraft Controller System

component (supertype) by its evolved version (subtype).

Dynamic behaviors of components have been modeled using notations such as CSP, temporal logic, and variants of state-based modeling. These techniques typically leverage the component's interfaces to model dynamic behaviors of the component. For instance [1] takes a temporal logic based

approach to specifying a component's behavior, using the component's interfaces and defines *axioms* that describe the dynamic behavior of the component. Similarly, state-based notations such as UML's state diagrams leverage a component's interface (specified in the class diagrams) for modeling events/actions on the state diagram model. In other state-based dynamic modeling techniques, component interfaces are used to define the transitions on the state machine representing the dynamic behavior of the component. One exception to directly leveraging the component's interfaces is [2], which instead focuses on connectors and protocol specifications at connection points, or ports, using CSP.

Despite the wide ranges of approaches, all of the above techniques fall short of exactly defining what constitutes the states in the dynamic model's state machines, or what effects the static behavioral specification and the dynamic model have on one another.

We have adopted an existing approach to dynamic modeling and slightly enhanced it to provide a basis for precisely identifying the meaning of state. In turn, this allows us to study and establish the relationships between the different levels of component models. While our ongoing work is addressing all four levels of component models, due to space constraints in this paper we only focus on the static and dynamic behavioral models. In this section we first describe our adopted notations for static and dynamic behavior specification. We then describe the necessary enhancement to the dynamic modeling notation. Finally, we show how this model can be used in conjunction with the static behavioral model to describe a component more completely and accurately.

### 3.1. Static Behavior Modeling

The static specification of a component describes its behavior at specific snapshots in the system's execution. Static behavioral specification techniques are successful in 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. We adopt a widely used approach to static modeling [8] consisting of the following set of concepts:

- *State variables* – describe the state of a component. A state variable has a name and an associated type.
- *Invariants* – constrain the state variables to specify a valid range of values for each variable. Invariants are typically specified using first-order logic.
- *Interface* – describes a component's services. An interface may compose a number of *interface elements*. Each interface element has a name, may have a set of *input* parameters, and possibly a *result* type.
- *Operations* – further describe the static behavior of a component in terms of pre- and post-conditions that must be satisfied prior to and after an operation is invoked, respectively. Each operation may be mapped to one or more interface elements. An operation is accessed via its corresponding interface element.

An example of the above concepts is showed in Figure 1, where the static behavior of a spacecraft's controller component is demonstrated.

### 3.2. Dynamic Behavior Modeling

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.

Variations of state-based modeling techniques (e.g., FSM,

StateCharts) have been frequently used to model a component's internal behavior. Such approaches describe the component's dynamic behaviors using a set of *sequencing constraints* that define legal orderings 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). As an example of the latter case, consider a component that has a *multiply (num1, num2):Int* interface. This component may perform the multiplication operation by successive additions. However, the addition operation may be internal to the component and thus invisible to the rest of the system.

To simplify our discussion throughout this paper, we only focus on the first case: publicly accessible operation. The second case may be reduced to the first one using the concept of *hierarchy* in the StateCharts semantics; 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.

We adopt a widely used approach to dynamic modeling consisting of the following set of concepts:

- *State* – is an abstraction representing a single state of a component during the system's execution.
- *Activation* – is a condition caused by an external stimulus that may result in changes in the state of the component. The stimuli may originate from outside the component, resulting in changes to the given component's state, or they may be initiated from within the component, in turn causing changes in the states of other components.
- *Transition* – marks a change in the component's state from an *origin* to a *destination* state once activated. At a given state, a transition may be outgoing or incoming.
- *Transition Label* – is referred to the label appearing on a transition. The transition labels directly correspond to the interface elements' names as described in the static behavioral model. In a given state, multiple transitions carrying the same label may be specified. However, each such transition must have a different guard (see below).
- *Guard* – is a condition on a transition. A transition will only be activated if its guard value is evaluated to be *true* at the time when the external stimulus is received. We model a transition's guards using first-order logic.
- *Union Guard* – at a given state, the union guard (UG) of a transition label refers to the disjunction of all guards associated with the outgoing transitions that carry the same transition label, that is:

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

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

Henceforth, we use the StateChart depicted in Figure 2 as the dynamic behavioral model for the controller component of the spacecraft example.

### 3.3. An Extension to the Dynamic Model

A component's dynamic behavioral model as discussed above is capable of describing the details of the component's execution in terms of the order in which its operations may be invoked (via the corresponding interfaces). Such description is based on an *abstract* notion of component states and there are no specific guidelines that identify what the precise meaning

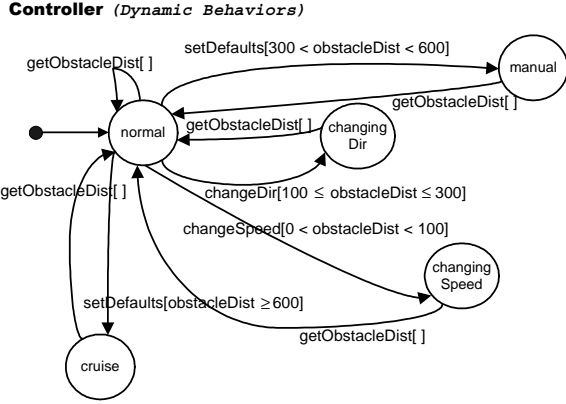


Figure 2. State-based model of Controller component's dynamic behavior

of a state is in the dynamic model. Such guidelines would be helpful when designing the dynamic model in a manner that preserve required system's properties as specified in the static model. Concretely defining the states in the dynamic specification also results in a richer model that may be used for more complex analysis of the system's runtime properties, as well as generation of implementation level artifacts.

As previously discussed, the states of a component in the static behavioral specification of a system are modeled using a set of state variables. The possible values of these state variables may be constrained by component's invariants specification. Furthermore, a component's operations may modify the state variables' values, thus modifying the state of the component as a whole. Consequently, we extend the dynamic behavioral model to specify the state of a component in terms of one or more invariants that govern the component's state variables. The invariant associated with a state  $S$  is denoted by  $S_{inv}$ . As an example, in Figure 2, the states *normal*, *changingDir*, and *changingSpeed* in the StateChart may be specified as:

$$\begin{aligned}
 normal_{inv}: & (0 < obstacleDist, 0 \leq dir < 360, \\
 & 0 < speed < 500, 0 < fuelLeft < 4000) \\
 changingDir_{inv}: & (100 \leq obstacleDist < 300) \\
 changingSpeed_{inv}: & (0 < obstacleDist < 100)
 \end{aligned}$$

Sometimes a state's invariant may only constrain a subset of the component's invariants. Essentially, this implies that the values of the other state variables are unimportant to this state and that the state does not modify those variable. For instance, the state *changingDir* as defined above only constrains the value of the *obstacleDist* variable and poses no limitations on *dir*, *fuelLeft*, and *speed*. In these cases the values for the variables not constrained by the state are considered to be "don't care" values and thus will not be specified in the state specification. This observation directly impacts the relationship between the static and dynamic models, as detailed in the next section. We now use this slightly extended model to analyze the range of the possible relationship between the static and dynamic behaviors of the components.

## 4. MODEL RELATIONSHIPS

Understanding the relationship between different levels of component models provides the means for maintaining the consistency across the models. In turn, this can result in the ability to develop implementation-level artifacts in a manner consistent with the initial architecture. In this section, we analyze in detail the range of possible relationships between the

static and dynamic behavioral models.

### 4.1. Transition Guard vs. Operation Pre-Condition

An operation's pre-condition as described in the static behavioral model is defined as a set of constraints on the invocation of the corresponding interface element. We previously discussed that a transition in the state-machine of the dynamic behavioral model corresponds to an interface element in the static behavioral model. Moreover, as mentioned in the previous section, our state-based semantics leverage guards as conditions associated with a transition's execution. Intuitively, in a given state, there is a relationship between the union guard of a transition label, and the corresponding operation's pre-condition. The following cases analyze the range of such relationships:

**Case 1.** The strongest relationship between the union guard of a transition label originating from a given state and the pre-condition associated with the corresponding operation is that of equivalence. In other words,  $P \Leftrightarrow UG$

This relationship ensures that the transitions in the dynamic model will be activated only in situations that satisfy the pre-conditions specified in the static model. It also ensures that all of the circumstances under which the operation may be invoked in the static model are accounted for in the dynamic behavioral model.

As an example in Figure 2, the dynamic model is designed such that different states (*manual* and *cruise*) are going to be reachable as destinations of the *setDefaults* transition depending on the distance of the encountered obstacle (*obstacleDist* variable in the transition guards). However, the UG of *setDefaults* is not equivalent to the pre-condition of the *setDefaults* operation given in the static behavior model in Figure 1 (the *setDefaults* transitions do not account for the value of *obstacleDist = 300*). Therefore, the UG of the transition label and the operation's pre-condition are not equivalent. This does not necessarily mean that the two models are violating one another, as further discussed below.

**Case 2.** If the UG of a transition label is stronger than the corresponding operation's pre-condition, then the operation may still be invoked safely. The reason for this is that the UG simply places bounds on the operation's invocation, ensuring that the operation can never be invoked under circumstances that violate its pre-condition. In other words,  $UG \Rightarrow P$

Let us consider the static behavioral model to be an abstract specification of the component's semantics. The dynamic behavioral model then becomes the concrete realization of those semantics. As a result, the above relationship preserves the properties of the abstract specification in its concrete realization. In that sense, this relationship is less restrictive, but just as effective as *Case 1*. Note that the *setDefaults* example described in *Case 1* represents an instance of this relationship.

**Case 3.** If the UG of a transition label is weaker than the corresponding operation's pre-condition, then the operation may not be invoked safely under certain circumstances. The reason for this is that the UG would allow the operation to be invoked under circumstances that violate its pre-condition. In other words,  $P \Rightarrow UG$

This relationship may have some undesirable effects on the running system since it enables the concrete realization (i.e., the dynamic model) to operate under a wider range of conditions than those specified in the abstract specification (i.e., the static model). An example of this case is the *changeDir* transition originating from the state *normal* in Figure 2: the guard on this transition allows *obstacleDist* to equal 300, which is

not allowed by the corresponding operation’s pre-condition in Figure 1 (see below).

$$UG_{changeDir} = \{100 \leq obstacleDist \leq 300\}$$

$$P_{changeDir} = \{100 \leq obstacleDist < 300\}$$

**Case 4.** Sometimes the UG of a transition label and the pre-condition of the corresponding operation may share no logical relationship, i.e., neither implies the other.

No such relationships exist in our controller component example. However, they are easy to envision. For instance, the UG of a transition may specify that  $speed > 100$ , while the corresponding operation’s pre-condition specifies that  $speed < 110$ . This situation may or may not result in an invalid invocation of the operation in question. Runtime analysis would be required to determine whether the operation’s pre-condition is violated. Since no guarantees can be made at component specification time, this situation is best to be avoided.

## 4.2. State Invariants vs. Component Invariants

In the previous section we discussed the relation between the guards on a transition and the corresponding operation’s pre-condition. Once a transition is invoked, it may result in a change to the component’s state. In Section 3, we discussed how states in the dynamic model can be specified in terms of the static specification’s state variables and their constraints. We now take a closer look at the possible range of such relationships.

**Case 1.** The strongest relationship between the invariant that constrains a given state in the dynamic model and the component’s invariant specified in the static model is that of equivalence. In other words,  $State_{inv} \Leftrightarrow Comp_{inv}$

This relationship ensures that a given state in the dynamic model constrains the component’s state variables such that the state can accept all and only those values specified by the static specification. As an example in Figure 2, consider the state *normal*:

$$normal_{inv}: \{0 < obstacleDist, 0 \leq dir < 360, \\ 0 < speed < 500, 0 < fuelLeft < 4000\}$$

This state constrains all of the component’s state variables and does so in a manner that is equivalent with the static model depicted in Figure 1. However, as discussed in Section 3, an individual state in the dynamic model may not constrain *all* of the component’s state variables. For instance, the *changingSpeed* state (as defined in Section 3.3) only constrains the value of *obstacleDist* and “*does not care*” about the values of the other state variables. In such cases, when analyzing the relationship, we only do so with respect to the *common* set of state variables.

**Case 2.** If a state’s invariant is stronger than component-level invariant, then the state is simply bounding the component’s invariant further, but does not permit for circumstances under which the component’s invariant is violated. That is,

$$State_{inv} \Rightarrow Comp_{inv}$$

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 *Case 1*. Note that the  $changingSpeed_{inv}$  example discussed in *Case 1* above, represents an instance of this relationship.

**Case 3.** If a state’s invariant is weaker than the component-level invariant, then the component’s invariants specification may be violated. That is, the dynamic behavioral model may

allow situations that may not be permitted by the static model.

In other words,  $Comp_{inv} \Rightarrow State_{inv}$

This relationship may have undesirable effects on the system, since the concrete realization (i.e., the dynamic model) does not adhere to the abstract specification (i.e., the static model). No such relationship exists in our controller component model.

**Case 4.** Sometimes no logical relationship between the component’s invariant and the invariant associated with a given state may be established, i.e., neither implies the other. This situation may or may not result in an invalid invocation and requires runtime analysis of the system implemented based on the static and dynamic models.

## 4.3. State Invariants vs. Operation Post-Condition

The final important relationship between a component’s models is that of an operation’s post-condition and the invariant associated with the corresponding transition’s destination state. In the static behavioral model, each operation’s post-condition must be *true* after operation’s completion. In the dynamic behavioral model, upon activating a transition, the state of the component will change from the transition’s origin state to its destination state. Consequently, the destination state’s invariant and the operation’s post-condition are related. The space of their possible relationships is discussed below.

**Case 1.** The strongest relationship between an operation’s post-condition (P) and the invariant associated with the destination state of the corresponding transition is that of equivalence. In other words,  $State_{inv} \Leftrightarrow P$

This relationship ensures that invocation of a given transition will not result in circumstances that the corresponding operation’s post-condition does not permit. It also ensures that the operation’s post-condition does not violate the invariant of the appropriate destination state. No instance of this relationship exist in our example for Figure 1 and 2.

**Case 2.** If the invariant associated with a transition’s destination state is stronger than the corresponding operation’s post-condition (P), then the operation may still be invoked safely, for the reasons analogous to those discussed cases in Section 4.1 and 4.2. In other words,  $State_{inv} \Rightarrow P$

An example of this case is the *setDefault* transition originating from the state *normal* and destined at *manual* in Figure 2: the  $manual_{inv}$  may be specified as

$$manual_{inv} = \{100 < speed < 200, dir = 0\}$$

while

$$setDefault.Post = \{speed > 100, dir = 0\}$$

**Case 3.** If the invariant associated with a transition’s destination state is weaker than the corresponding operation’s post-condition (P), i.e.,  $P \Rightarrow State_{inv}$ , then the desired outcome of the operation may be violated. The argument is analogous to *Cases 3* in the preceding two sections.

An example of this case is the *setDefault* transition originating from *normal* and destined at *cruise* in Figure 2:  $manual_{inv}$  may be specified as

$$cruise_{inv}: \{obstacleDist \geq 600, speed > 100, dir \geq 0\}$$

while

$$setDefault.Post = \{speed > 100, dir = 0\}$$

The destination state *cruise* in this case can accept transitions that would set *dir* to a positive value. However, such values for *dir* are not permitted based on the abstract specification in Figure 1.

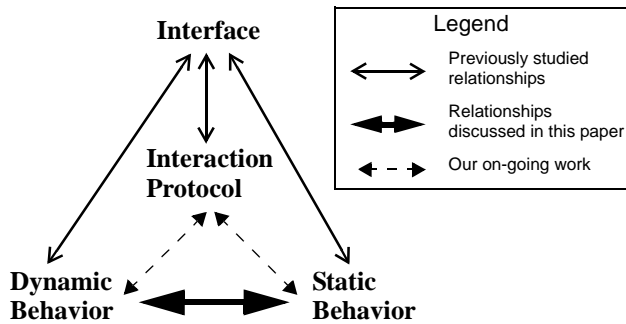


Figure 3. Possible relationships among the four levels of component specifications.

**Case 4.** Sometimes no logical relationship between the operation's post-condition and the destination state of the corresponding transition in the dynamic model can be established. As before, runtime analysis would be required to determine whether the transition may be safely invoked.

## 5. DISCUSSION

Architecture-based software development seeks to improve the design and development of large, complex, heterogeneous, and long-running systems. In support of this goal, a large number of techniques have been provided to rigorously model a software system's components. These techniques result in models of component *interfaces*, *static* and *dynamic* behaviors, and *interaction protocols*. However, in order for the modeling to be effective and the analysis of the models meaningful, certain relationships among the four levels of models and their elements must exist. Researchers have previously considered such relationships between interfaces and the other three modeling levels, but they have stopped short of establishing the remaining connections (see Figure 3).

In this paper, we have presented a preliminary study of the space of possible relationships between the pairs of the remaining three modeling levels. In particular, we have focused on the relationships between static and dynamic behavioral models. We are currently developing a similar understanding of the relationships between static and dynamic behaviors on the one hand and interaction protocols on the other. We believe that understanding those relationships is a pre-requisite to building comprehensive component specifications and keeping their various aspects consistent with one another.

The next step in this study will formally address component evolution and substitutability by establishing the subtyping relationships between the different component modeling levels. Similar work has been done for specific levels in existing modeling approaches (e.g., interface-level subtyping in programming languages or static behavior-level subtyping in the work of Liskov and Wing [8] and Zaremski and Wing [15]). However, exploring the range of such relationships *across* the modeling levels (e.g., does the supertype's static behavior impact the subtype's interaction protocol, and how) is an open research question.

Eventually, relationships such as those discussed above will need to be established for collections of *interacting* components in a software system. Our ultimate goal in this work is to extend our existing software architecture-based development tool support to encompass flexible multi-level component modeling and automated analysis.

## 6. REFERENCES

- [1] Aguirre N., Maibaum T.S.E., "A Temporal Logic Approach to ComponentBased System Specification and Reasoning", in *Proceedings of the 5th ICSE Workshop on Component-Based Software Engineering*, Orlando, FL, 2002.
- [2] Allen, R., Garlan, D., "A formal basis for architectural connection" *ACM Transactions on Software Engineering and Methodology*, 6(3):213–249, 1997.
- [3] Allen R. and Garlan G., "Formalizing Architectural Connection", In *Proceedings of the sixteenth International Conference on Software Engineering*, pages 71-80, Sorrento, Italy, May 1994.
- [4] America P. "Designing an Object-Oriented Programming Language with Behavioral Subtyping", *Lecture Notes in Computer Science*, vol. 489, Springer-Verlag, 1991.
- [5] Dias, M., Richardson, D., "Identifying Cause & Effect Relations between Events in Concurrent Event-Based Components", In *Proceedings of International Conference on Automated Software Engineering (ASE 2002)*, Edinburgh, September 2002.
- [6] Fariás 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*.
- [7] Harel, D., "Statecharts: A Visual Formalism for Complex Systems", *Science of Computer Programming*, 1987.
- [8] Liskov B. H., Wing J. M., "A Behavioral Notion of Subtyping", *ACM Transactions on Programming Languages and Systems*, November 1994.
- [9] Medvidovic N., Taylor R.N., "A Classification and Comparison Framework for Software Architecture Description Languages", *IEEE Transactions on Software Engineering*, January 2000.
- [10] Microsoft Developer Network Library, *Common Object Model Specification*, Microsoft Corporation, 1996.
- [11] Object Management Group, *The Common Object Request Broker: Architecture and Specification*, Document Number 91.12.1, Revision 1.1, OMG, December 1991.
- [12] Plasil F., Visnovsky S., "Behavior Protocols for Software Components", *IEEE Transactions on Software Engineering* 28(11), pp. 1056–1076, November 2002.
- [13] *The Object Constraint Language (OCL)*, <http://www-3.ibm.com/software/ad/library/standards/ocl.html>.
- [14] Yellin D.M., Strom R.E., "Protocol Specifications and Component Adaptors," *ACM Transactions on Programming Languages and Systems*, Vol 19, No. 2, pp 292-333, 1997.
- [15] Zaremski A.M., Wing J.M., "Specification Matching of Software Components", *ACM Transactions on Software Engineering and Methodology*, 6(4):333–369, 1997.