Verifying Behavioral Component Interoperability Using Positive/Negative Model Checking

Weimin Ma
Dept. of Computer Science
The Univ. of Texas at Dallas
weiminma@utdallas.edu

Lawrence Chung
Dept. of Computer Science
The Univ. of Texas at Dallas
chung@utdallas.edu

Kendra Cooper
Dept. of Computer Science
The Univ. of Texas at Dallas
kcooper@utdallas.edu

Abstract - Component-based development needs to establish structural interoperability as well as behavioral interoperability among components. To solve a structural or behavioral mismatch, adapters are generally constructed using the basic scenarios, while ignoring the exceptional scenarios. This paper proposes an approach for extending or refining the integrated components, glued together using adapters, after basic scenarios have been considered. The approach examines the integrated components using exceptional scenarios. Furthermore, formal models for the integrated components are built and verified against safety and liveness properties derived from basic and exceptional scenarios. A traffic signal control system example is given to illustrate the approach.

Keywords: Component, behavioral interoperability, positive/negative model, model checking

1. Introduction

Component-based development (CBD) has the potential of reducing effort and time, and taking advantage of the built-in qualities of constituent components. Usually the starting point is to assess the component’s structural interoperability in the interface attributes (name, type match) and interface operations (name, parameter list). A structural match cannot guarantee the sequences of communications among components are consistent. This leads to the assessment of behavioral component interoperability. Adaptors [1][2] are necessary when components are either structurally or behaviorally non-interoperable.

Recent work in CBD has focused on the architecture of the components [3][4][5]. However, it is crucial to verify that the integrated components glued together using adapters realize the stakeholders’ requirements in terms of the communications among constituent components.

Based on the proposal by Jackson and Zave [6][7], domain knowledge and specification together satisfy the requirements, i.e. D, S |= R, this paper proposes a methodology to address behavioral component interoperability. The behavioral specification of the integrated components consolidates individual component’s behavior, specified using UML statechart, through either basic scenarios (i.e. positive scenarios) or exceptional scenarios (i.e. negative scenarios). The specification of integrated components (i.e. specification model) built following the positive scenarios is a positive model, and the specification model built following the negative scenarios is a negative model. Our approach uses the notion of negative models in gaining insights about the behavior of a positive (i.e. desirably correct) or a negative (i.e. undesirable) model of integrated components. There are infinite number of causes that the positive model fails, and the search space can be reduced by examining the negative model. As shown in Figure 1, exceptional scenarios are depicted using the negative frame (i.e. neg sd) in UML sequence diagram. Scenarios weave the specification components into the specification of the integrated system (i.e. specification model). Properties extracted from sequence diagram, e.g., in the same direction traffic light and pedestrian signal shall not conflict, are checked against the specification model. Domain knowledge is used to specify external events.

This paper concerns behavioral component interoperability on the requirements level. Therefore, component is defined as the requirements-level specification in this paper. The definition of behavioral interoperability is defined as the ability of each individually workable component to be integrated into a system which can work properly with respect to both basic scenarios and negative scenarios. A specification is different from the requirements, as it specifies the general solution to the goals identified in the requirements. Requirements define the full functionalities of the system, which includes hardware and software.

This paper benefits from the following related research. The proposal from Zave and Jackson [7] and Grosu and Smolka [8] lays the groundwork of the approach proposed in our paper. Zave and Jackson explain the precious nature of relationships among
domain, specification and requirements, and Grosu and Smolka help us to define negative adapters from negative traces in scenarios. CHARMY [5] uses a formal approach for property validation and verification to model-check software architecture compliance to certain functional temporal properties. In CHARMY, UML component model and state transition are used to represent software architecture, and message sequence chart is utilized to represent properties. Bracciali et al. [2] focus on adapting mismatched behaviors that component may exhibit with an adapter. Component interface is IDL (Interface Definition Language) interface extended with the behavioral description of the component using π-calculus. Mouakher et al. [1] address component adaptation issue and propose an approach to use B formal method to represent component model translated from UML component diagram and state transition, where a state transition is associated with each interface.

![Figure 1: Methodology for verifying behavioral interoperability using positive/negative model](image)

The rest of the paper is organized as follows. Section 2 introduces the background and notations used in this paper. In section 3, a traffic signal control system example illustrates the approach of using the notion of negative adapters in gaining insights about the behavior of a positive or a negative model. Finally, Section 4 gives the conclusion and future work.

2. Background

This section starts with the introduction of the traffic signal control system (TSCS) example in which illustration of our approach is explained. Then the notations and techniques used in this paper to address the approach is introduced and explained.

2.1. Traffic Signal Control System Example

A local city administration wants to build a traffic signal control system in main intersections, such that a traffic controller controls a traffic light and a pedestrian signal in the same direction. A candidate controller starts at red light and transitions through various states, while a candidate traffic light starts at green light followed by yellow light and then red light. A candidate pedestrian signal receives two non-interrupted messages together in order to proceed.

The controller and the traffic light are non-interoperable. The reason is the initial color setting between the controller and the traffic light, i.e. red light vs. green light. This problem can be resolved with an adapter when the local city administration wants to take CBD approach. Adapters can be constructed with respect to basic interactions represented in sequence diagram. For example, pedestrian signal’s press button is pressed before the traffic light is in yellow. The adapter built in this manner is called positive adapter. A specification model integrated with components and positive adapters is a positive model. Adapter can also be constructed following the exceptional interactions represented in sequence diagram, e.g., pedestrian signal’s press button being pressed after the traffic light is in yellow, which leads the adapter into an undesirable state. The adapter built in this manner is called negative adapter. A specification model integrated with components and negative adapters is a negative model. The basic scenarios show the ordinary flow of interactions among constituent components, therefore represent liveness properties. The exceptional scenarios show the flow of interactions that should not occur, therefore represent safety properties.

Figure 2 depicts the decision tree for our approach. This process assumes that the existing components are already structurally interoperable. These components are either behaviorally non-interoperable or behaviorally interoperable. In either case, both positive model and negative model are checked against selected safety properties and liveness properties. Model checking technique is used to perform the property verification of these models.

![Figure 2: Decision tree for our approach](image)
2.2. UML Diagrams

In this paper, each specification component is modeled using UML class and statechart. Each UML class depicts an entity on requirements level, which has behavioral specification using UML statechart. During component integration, components are interdependent and communicated through interleaving interactions. For the traffic signal control system, the controller, the traffic light and the pedestrian signal all have their own sequence of actions and states to go through, as shown in Figure 3. It is critical to determine whether every component’s sequence of actions can contribute to a working system. Figure 3 shows adaptation non-interoperable components, i.e., components that are non-interoperable even using adapters, and adaptation interoperable components, i.e., components that are interoperable using adapters.

![Controller Spec Component](image)

**Figure 4: Controller specification component**

### 2.2.2. Statechart

UML statechart is used to model behavioral aspect of a specification component. A UML statechart is one particular kind of state machine proposed by Harel [9]. A statechart models an event-ordered behavior that specifies the sequences of states an object (more specifically, specification component) goes through during its lifetime. An event triggers transition and causes response. The similar definition of statechart and automaton adopted by the model checker makes the model checking technique a convenient tool to verify the state-related properties of a specification component. Figure 4 shows a statechart Controller_ST for specification component Controller. It contains six states, i.e., “Red Light On”, “Green Light On”, “Interim”, “Pedestrian Signal Blinking”, “No Walk”, and “Yellow On”. Transition from “Red Light On” state to “Green Light On” state contains event “After (some time)” and the send signal “Turn on”.

### 2.3. Model Checking and Branching Temporal Logic

Model checking is a procedure to decide whether a given structure is a model of a logical formula [10]. The procedure to perform model checking is to traverse the product of a system structure automaton and the Büchi automaton, converted from a property formula, and check whether certain property is violated or not. When the properties hold, the model checker says “yes”; when the properties do not hold, the model checker says “no” and gives a counter example. Intuition tells us that verifying state transition model of a system against certain properties can lend insights about its corresponding positive model.

Temporal logic expresses properties of event orderings in time [11]. There are linear time logic and branching time logic. Computation tree logic belongs to the branching time logic, which is an infinite tree and every moment each node has several successors. Safety and liveness queries in UPPAAL [12] are expressed using the timed computation tree logic.
(TCTL). There are four quantifiers in TCTL, which are 
E (exists a path), A (for all paths), [] (all states in a 
path) and <> (some state in a path). The following 
combination are supported, i.e., A[], A<>, E] and 
E<> . For example, the property “traffic light in yellow 
and pedestrian signal in walk cannot happen together” 
can be expressed in TCTL as “[A[] not (TL_sys.Yellow_On and PS_sys.Walking)”. 

There are many model checkers available, 
including SPIN [10], SMV, and UPPAAL. In order to 
conform to the specification model of this paper SCR 
(Software Cost Reduction) [13] is used, whose 
underlying verification tools can be SPIN and SMV. 
SCR uses four-variable model which contains 
monitored variables, controlled variables and mode 
classes. Since the table notation of SCR is not intuitive 
and user friendly, UPPAAL is then used to model 
integrated components with adapters. The safety and 
liveness properties are checked against these models.

3. Illustration

Following the traffic signal control system example 
described in Section 2.1, the local city administration 
is willing to take the CBD approach. Two suites of 
specification components are chosen. In suite one the 
components are adaptation non-interoperable, i.e. non-
interoperable even with adapters, and in suite two the 
components are adaptation interoperable, i.e. 
interoperable with the help of adapters. The local city 
administration has a particular requirement, i.e., 
pedestrian signal should not allow pedestrian to walk 
while the traffic light is not green. The environment 
tells that the pedestrian can press the press button at 
any time.

3.1. Building Positive and Negative Models 
Using UML

For the adaptation non-interoperable case, i.e., 
\¬ \exists A. such that I_d(controller, A, pedestrian signal) 
(there does not exist a positive adapter to bridge the 
behavioral gap between a controller and a pedestrian 
signal), the pedestrian signal receives messages “Turn 
on green” and “Press to walk” atomically, which 
means that these two messages cannot be interrupted. 
A controller sends and receives messages in the 
interruptive manner. For the adaptation interoperable 
case, i.e. \exists A. such that I_d(controller, A, traffic light) 
(there exists a positive adapter to bridge the behavioral 
gap between the controller and the traffic light), the 
controller starts at the state “Red light on”, while the 
traffic light starts at the state “Green light on”. The 
pedestrian signal for the adaptation non-interoperable 
case is depicted in Figure 5. The traffic signal for the 
adaptation interoperable case is depicted in Figure 6. 

The corresponding message interactions for the 
adaptation non-interoperable and adaptation 
interoperable cases are shown in Figure 3.

**Figure 5: Pedestrian signal specification component**

**Figure 6: Traffic light specification component**

For both adaptation non-interoperable case and 
adaptation interoperable case, a positive adapter and a 
negative adapter are constructed to glue the candidate 
specification components. The positive adapter is 
constructed upon positive scenarios, shown as 
scenarios other than the scenarios framed using “neg” 
depicted in Figure 7, and a negative adapter is built 
upon the scenarios including the negative scenarios in 
frame “neg”. The negative scenarios within frame 
“neg” are explained as follows: when the traffic light is 
in yellow, if the pedestrian presses the press button, 
then the controller receives an “Apply to walk” 
message and is ready to issue a “Walk” message to the 
pedestrian signal. These are undesirable scenarios, 
since the pedestrian signal could be in the “Walking” 
state while the traffic light is in yellow or red.

As enlightened by [8], the negative adapter (Figure 
8) transitions to an “Undesirable” state on receiving 
the “Press to walk” message when in the “Yellow on” 
state. After entering the undesirable state with respect 
to the negative scenarios, the adapter is designed in 
such a way that it then transitions into the initial state 
after receiving a “Terminate” message from the 
controller. Similarly, the negative adapter between the 
controller and traffic light for the adaptation 
interoperable case is also constructed, but is not shown 
here due to the page limit.
Figure 7: Scenarios described in sequence diagram with negative traces in the frame “neg”

Figure 8: Specification component of a negative adapter for non-interoperable components

3.2. Translating Positive and Negative Models into Formalism

After the construction of the positive and negative models, the examination of whether the negative model can help to lend insights of the positive model is performed. Formal approach is considered due to its ability to verify properties against the state-transition type of specification components. UPPAAL and SCR are used to model the integrated components with positive and negative adapters. This is because SCR is suitable for modeling requirements-level specification and UPPAAL is user friendly.

State transitions of a negative adapter which bridges the specification component “Controller” and “Pedestrian Signal” are translated into UPPAAL templates, as shown in Figure 9. Two templates for one adapter show the concurrency of the controller controlling both traffic light and pedestrian signal. When the negative adapter receives an asynchronous message “Turn on” from the specification component “Controller”, it forks into two concurrent branches. One branch depicts the concurrent behavior of the controlling pedestrian signal known as “Adapter_PS” in Figure 9. It transitions from “Initial” state to an intermediate “Dummy_1” state. Since for the pedestrian signal component, the message “Turn on” and “Press to walk” are atomic messages, then the UPPAAL template “Adapter_PS” halts at “Dummy_1” location (i.e. committed location) until it receives the message “Press” (i.e. “Press_monitor” in UPPAAL template) from the press button. Then the adapter issues the “Apply to walk” message to the controller and waits for the message “Walk” from the controller. After the adapter receives the “Walk” message, it sends out a “Proceed” message to the pedestrian signal. After that, when the adapter receives the “Blink pedestrian” message from the controller, the adapter sends a “Blink pedestrian signal” message to the pedestrian signal. Eventually the adapter sends a “Dismiss blink” message to the pedestrian signal after receiving the “Terminate blink” message from the controller.

Figure 9: Snapshot of UPPAAL templates of a negative adapter

Another concurrent branch of the adapter with respect to the adaptation non-interoperable components communicates the controller to the traffic light. Besides the sequential translation of messages between the controller and the traffic light, the adapter enters an “Undesirable” state when receiving “Press to walk” message while in “Yellow on” state. This is to
depict the negative traces in the sequence diagram with respect to the local city administration’s requirements. When in the “Undesirable” state and received “Terminate” message, the adapter then transitions into a state that continues with the positive scenario.

The adaptation non-interoperability lies in the fact that unless the pedestrian signal receives messages “Turn on” from the controller and “Press to walk” from the press button at the same time, the pedestrian signal cannot proceed. In real life phenomenon, this means that the controller can only control the traffic light, not the pedestrian signal even when the pedestrian presses the press button. The atomic nature of the messages “Turn on” and “Press to walk” of pedestrian signal is realized using “committed” location in UPPAAL. As shown in the template Adapter_PS in Figure 9, locations “Dummy_1” and “Dummy_2” are committed locations.

After the translation of the positive and negative models from UML notation to formalism, the models in formal notation are checked against safety and liveness properties to explore the possibility of extending or refining the positive model using negative model.

3.3. Refining Positive Model Using Negative Model

The translation between UML notation and a state transition formalism of positive and negative models is performed manually in this paper. After the translation of these models, next step is to perform the simulation using the built-in simulator of UPPAAL and SCR. The simulator of UPPAAL has better graphical interface hence is easier to perform the simulation of the positive and negative models with respect to a random or interactive traces. After checking that the interactions with respect to the positive and negative scenarios do occur during the simulation of the models, the models are then verified against safety and liveness properties shown in Table 1. The verification results for the positive and negative models with respect to an adaptation non-interoperable and an adaptation interoperable component are shown in Table 2.

For both adaptation non-interoperable case and adaptation interoperable case, the intuition tells that the verification result for the negative model is different from the positive model when verified against the same properties. The reason is that the negative model intentionally includes the scenarios that are undesirable from the requirements’ standpoint. It is shown in Table 2 that the verification results are the same for both positive and negative models, with respect to both an adaptation non-interoperable case and an adaptation interoperable case. This shows that these particular positive models and negative models are equivalent in addressing the same requirements when the domain knowledge is the same. After careful examination, we find that these negative models are constructed in such way that they complement their corresponding positive models with respect to negative scenarios.

Table 1: Safety and liveness properties in English and temporal logic

<table>
<thead>
<tr>
<th>Safety</th>
<th>Liveness</th>
</tr>
</thead>
</table>
| 1. Components can work together: \[ A \] not deadlock. | 0. When controller is in “red light on”, eventually pedestrian signal is in “do not walk”:
\[ A \] (Controller_sys.Red_Light_On imply PS_sys.Do_Not_Walk) |
| 2. Traffic light is in yellow and pedestrian signal is in walking cannot happen at the same time:
\[ A \] (not (TL_sys.Yellow_On and PS_sys.Walking)) | 1. When push button is pressed, eventually pedestrian signal is in walking:
\[ A <> (PB_sys.Press_issued imply PS_sys.Walking) ] |
| | 2. When controller is in “red light on”, eventually pedestrian signal is in “do not walk” and traffic light is in initial:
| | 3. When controller is in “green light on”, eventually traffic light is in “light on”:
\[ A <> (Controller_sys.Green_Light_On imply TL_sys.Light_On) ] |
| | 4. When controller is in “red light on”, eventually pedestrian signal is in “do not walk” and traffic light is in initial:
| | 5. When controller is in “green light on”, eventually traffic light is in “light on”:
\[ A <> (Controller_sys.Green_Light_On imply TL_sys.Light_On) ] |
| | 6. When controller is in “green light on”, eventually traffic light is in “light on”:
\[ A <> (Controller_sys.Green_Light_On imply TL_sys.Light_On) ] |
| | 7. When controller is in “walk ready”, eventually pedestrian signal is in “walk ready”:
\[ A <> (Controller_sys.Walk_Ready imply PS_sys.Walk_Ready) ] |
| | 8. When controller is in walking, eventually pedestrian signal is in walking:
\[ A <> (Controller_sys.Walking imply PS/sys.Walking) ] |
| | 9. When controller is in “pedestrian signal blinking”, eventually pedestrian signal is in blinking:
\[ A <> (Controller_sys.pedestrian_signal_blinking imply PS/sys.Blinking) ] |
| | 10. When controller is in “no walk”, eventually pedestrian signal is in “do not walk”:
\[ A <> (Controller_sys.no_walk imply PS_sys.Do_Not_Walk) ] |
| | 11. When controller is in “yellow on”, eventually traffic light is in “yellow on”:
\[ A <> (Controller_sys.yellow_on imply TL_sys.Yellow_On) ] |

Table 2: Verification results for positive and negative models using model checker

<table>
<thead>
<tr>
<th>Safety</th>
<th>Liveness</th>
</tr>
</thead>
<tbody>
<tr>
<td>Satisfied</td>
<td>Unsatisfied</td>
</tr>
<tr>
<td>¬Ia(C1, C2) ∧ ¬A, s.t. Ia(C1, C2) ∧ 1 (Safety 2)</td>
<td>1 (deadlock)</td>
</tr>
<tr>
<td>¬Ia(C1, C2) ∧ ¬A, s.t. Ia(C1, C2) ∧ 1 (Safety 2)</td>
<td>1 (deadlock)</td>
</tr>
<tr>
<td>Ia(C1, C2) ∧ Ia(C1, C2) ∧ 1 (Safety 2)</td>
<td>2</td>
</tr>
<tr>
<td>Ia(C1, C2) ∧ Ia(C1, C2) ∧ 1 (Safety 2)</td>
<td>2</td>
</tr>
</tbody>
</table>

The following summarizes our observation through the example as to how the negative model helps to extend or refine the positive model:

1. A gap exists between the specification model and the requirements with or without the domain knowledge;
2. The gap between requirements and specification makes the specification inadequate to address the requirements, and is also difficult to be detected by solely checking the positive model;
3. When both positive model and negative model satisfy the same properties described in scenarios, the positive model is refined by adding the transitions and states that the negative model possesses.

4. Conclusions and Future Work

A critically challenging issue with component-based software development is how to compose behaviorally mismatched components with, or without, adapters. In determining the interoperability between components, or lack thereof, identifying such important properties as safety and liveness of components is a difficult task if they are examined solely based on their behavior as represented using a state transition formalism. It is often times more natural to understand real life phenomena through interactions before such examinations can take place. In this paper, we have presented an approach to using the notion of “negative” models in gaining insights about the behavior of a positive (i.e., desirably correct) or a negative (i.e., undesirable) model of integrated components, with or without an adapter, when verifying them using a model checker. The benefits of this approach include our ability to extend or refine a model, if it is correct, and to detect and correct a model, if it is incorrect.

Some novel distinctives of the proposed approach are:
1. Components are represented as a requirements-level specification;
2. Both positive- and negative- scenarios are considered in developing/choosing adapters for achieving component interoperability;
3. Both safety and liveness properties are verified against positive and negative models using the model checking technique;
4. A negative model is developed from a negative scenario and verified, which lends insights about its corresponding positive model, when verification of the positive model itself is not enough to reveal if the (specification) model, together with the domain model, satisfies the requirements (i.e., S, D |- R).

Future work includes addressing the limitations of the proposed approach, concerning the detailed extensions/refinements of a positive model that its corresponding negative model can guide. This is important since the space of potential causes of a component mismatch is often times huge, if not infinite. Another line of future research concerns how to use model checking techniques in verifying a specification model against non-functional requirements. It is also desirable that the translation between UML notation and a state transition formalism is automatically performed. Finally, a variety of case studies are needed to assess the utility of the proposed approach, concerning both functional and non-functional requirements.

References