Compatibility-Guaranteeing Software Component Evolution 
based on Composition Rules

Sunae Lee, Joon-Sang Lee, and Doo-Hwan Bae 
Dept. of Electrical Engineering and Computer Science, KAIST

Abstract
Since building large-scale software is usually big burden to most developers, it has been an important issue for many researchers. In this paper, we suggest a mechanism that can be used to support such large-scale development. Through composition rules via subtyping within Statecharts, incremental construction of software can be achieved. Among the composition rules (i.e. delegation rule and mixin rule), we mainly focus on the delegation rule in our work. Not only we can check the subtype property, but also can verify the behavior compatibility of composite results that are available by composition rules. This new mechanism is helpful for analysts as well as designers, and it can be used as a guideline for incremental and compatible construction of component based software.

1. Introduction
Software systems have become larger and more complex than before. In such trend, the inheritance mechanisms have been suggested as a good solution for building complex and large systems. However, the flexible management of subclassing sometimes causes unexpected behaviors. Past experience has shown that unrestricted use of inheritance mechanisms leads to system architectures that are hard to understand and maintain, since there are usually arbitrary differences between a supertype and its subtype.

Prior work[11] is not enough to formulate behavior compatibility, especially when it comes to the liveness properties[4]. In this paper, we suggest a novel mechanism for component evolution based on two composition rules. With these composition rules, incremental construction of software is possible.

Generally, state machines can be used to specify a software system. Like the state machine, object-oriented design methodologies typically use notations based on Finite State Automata, Petri-nets, and Statecharts[5]. Among these notations, we use Statecharts. Through the composition rules, simple Statecharts can be extended to the complex ones.

This paper is organized as follows. Section 2 discusses the behavioral compatibility guaranteeing method. The notion of subtyping provides a basis for checking the behavior compatibility between components. In Section 3, we provide an intuitive explanation about two composition rules with examples. Section 4 describes a formal definition of the delegation rule and Section 5 shows the behavior compatibility of the composition result. In section 6, we conclude our approach with the description of future work.

2. Related Work (Compatibility guaranteeing utensil: subtyping)
The reuse of existing components to the utmost makes the building of software easy. Many concepts and theories such as inheritance, aggregation and delegation have been introduced to support reuse. Using these, we propose a set of generic composition operators for software design and implementation. It can provide a mechanism for incremental software construction.

Taivalsaari[6] mentioned that inheritance can be used in two utensils. One is for implementation reuse, and another is for specialization. When focusing on specialization, inheritance can be regarded as incremental modification mechanism rather than for conceptual modeling. Because of flexible management, many object-oriented languages have given up strong constraints for inheritance[7]. However, it causes too faint relationships between a supertype and its subtype thus makes software hard to understand and fuzzy. On the other hand, subtype property checking ensures that unexpected behavior does not occur. So it introduces an alternative view.

Behavior Diagram, similar with Petri-net and adjust three types of consistencies (observation, weak invocation, strong invocation consistency) as liveness property. In this paper, we use these three types of consistencies for checking behavioral consistency as subtype requirements.

3. Informal description of two composition rules

There are two kinds of composition rules. One is the delegation rule that can be used as a kind of black-box composition. A designer is interested only in the functionality of a component and composition of functionalities. Another is the mixin rule that can be used as a kind of white-box composition. With the mixin rule, designer can make many variations of one component easily.

3.1 Delegation rule

Delegation[2] is an implementation mechanism where an object forwards an operation to another object for execution. To simplify the problem, we assume that the delegation rule is a binary operation between two components. Fig.1 represents two components for a virtual memory paging scheme. Component A allows read only access and Component B allows write only access. A and B have similar behaviors on the different target. The functionalities of A and B are realized in D composition result. Fig.2 shows a Statechart for D.

![Fig.1] Two components for virtual memory paging scheme

![Fig.2] Composition result D

3.2 Mixin rule

A mixin is an abstract subclass[3], definition that may be applied to different superclasses to create a related family of modified classes. For example, a mixin might be defined as adding a border to a window class: the mixin could be applied to any kind of window to create a bordered-

window class. We assumed that the mixin rule is also a binary operation between two components. Fig.3 shows the idea of the mixin rule.

![Fig.3] Mixin rule

4. Formal definition of delegation rule (A * B = D)

As we mention before, the delegation rule is a binary operation. To get rid of ambiguity, we define A, B and D formally as Statechart.

To describe this rule, we regard a Statechart as a set of tuples (s_a, s_b, S, E, C, T) and a function f.

- s_a: initial state
- s_b: final state
- S: set of states
- E: set of events
- C: set of conditions
- T: set of transition \( t \in T \subseteq (s \in S \times e \in E \times s' \in S) \)
- f: a function that maps \( e \in E \) to \( c \in C \)

Then, A and B can be represented as follows.

- A: \((s_0, s_A, \Sigma_A, E_A, C_A, T_A, f_A)\)
- B: \((s_0, s_B, \Sigma_B, \Sigma_B, E_B, C_B, T_B, f_B)\)

To make explanation simple, we assumed that A and B have similar behavior and D want to have both behavior of A and B. For example, A represents rent video and B represents rent DVD. D wants to have two behaviors at the same time. This assumption is represented using transition mapping between A and B.

- For every transition of \( t_a \in T_A \), there exists a corresponding transition \( t_b \in T_B \)
- For every transition of \( t_b \in T_B \), there exists a corresponding transition \( t_a \in T_A \)

This assumption ensures that there exist corresponding states and events between Statecharts A and B.

On the above basic model and assumption, the delegation rule is defined as follows:

- Every state of A and B is included in the states of D.
- \( D' \)'s initial state can be \( A' \)'s initial state or \( B' \)'s initial state
- \( D' \)'s final state can be \( A' \)'s final state or \( B' \)'s final state
- Every event of A and B is included in the events of D.
- Every transition of A and B is included in the transitions of D.
- Make a new event that provides transition from A to B with condition \( c_1. (e_{A\text{old}}[c_1] \in E_D) \)
- Make a new event that provides transition from B to A with condition \( c_2. (e_{B\text{old}}[c_2] \in E_D) \)
- Make new transitions from all states of A to \( B' \)'s initial state. \( (\forall s \in A \{ s \in A\text{old}[c_1], s_{A0} \in T_D \}) \)
6. Conclusion and future work

In this paper, we suggested composition rules and checked compatibility of composition result using subtype requirement. Although the mixin rule is not defined in detail yet, the delegation rule is formalized and checked behavioral consistency. Through these two rules, a construction mechanism with efficient and incremental reuse is provided. Moreover, this mechanism guarantees compatibility of software component and becomes a safe and easy software evolution guideline.

For the future work, more concrete proof of subtype requirements has to be investigated. In addition, extension of two rules is considerable. Our research goal is to propose a set of generic composition algebra that makes it possible to reason out compatibility at various levels and to generate automated synthesis of code.

7. Reference