안정적 확장이 보장되는 소프트웨어를 위한 정형적 조합 법칙

이선애[O] 이준상 배두환
한국과학기술원 전자전산학과
{salee,joon,bae}@se.kaist.ac.kr

Compatibility-Guaranteeing Software Component Evolution based on Composition Rules

Sunae Lee[0], 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[1] 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 about 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.

Wing[8] suggested a clear understanding of how subtypes and supertypes are related especially in the criteria of safety property. *Subtype requirement*[8] provides a stronger constraint for the behavior of subtypes. Michael Schrefl and Markus Stumptner[4] propose a set of necessary and sufficient rules for checking behavior consistency between object life cycles of object types in specialization hierarchies with multiple inheritances. They define the behavior checking rules in the realm of Object

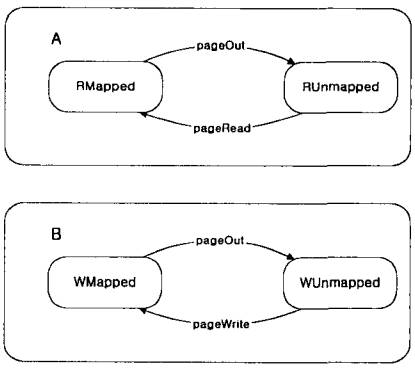
Behavior Diagram, similar with Petri-net and adjust three types of consistencies (observation, weak invocation, strong invocation consistency) as liveness property. In this paper, we use these three types of consistencies for checking behavioral consistency as subtype requirements.

3. Informal description of two composition rules

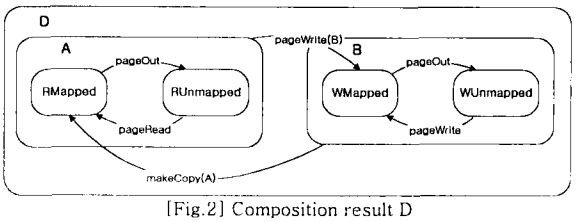
There are two kinds of composition rules. One is the delegation rule that can be used as a kind of black-box composition. A designer is interested only in the functionality of a component and composition of functionalities. Another is the mixin rule that can be used as a kind of white-box composition.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



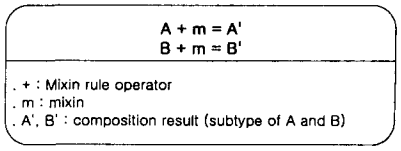
[Fig.1] Two components for virtual memory paging scheme



[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_o, s_f, S, E, C, T) and a function f.

- s_o : initial state
- s_f : 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 : (sa_0, sa_f, S_A, E_A, C_A, T_A) , f_A
- B : (sb_0, sb_f, S_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 represent *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 c1. ($e_{AtoB}[c1] \in E_D$)
- Make a new event that provides transition from B to A with condition c2. ($e_{BtoA}[c2] \in E_D$)
- Make new transitions from all states of A to B' s initial state.($\forall s \in S_A$ [(s, $e_{AtoB}[c1]$, sa_0) $\in T_D$])

- Make new transitions from all states of B to A' s initial state. ($\forall s \in S_B[\ (s, e_{BtoA}[c2], sb_0) \in T_D\ ]$)

According to the delegation rule, the Statechart of composition result D can be described.

- D : (sd_0, sd_f, S_D, E_D, C_D, T_D) , f_D

Each column of D is defined as follows:

- D : (sd_0, sd_f, S_D, E_D, C_D, T_D) , f_D
- $sd_0 = sa_0 \cup sb_0$
- $sd_f = sa_f \cup sb_f$
- $S_D = S_A \cup S_B$
- $E_D = E_A \cup E_B \cup \{new\ e_{AtoB}[c1]\} \cup \{new\ e_{BtoA}[c2]\}$
- $C_D = C_A \cup C_B \cup \{new\ c1\} \cup \{new\ c2\}$
- $T_D = T_A \cup T_B \cup \{new\ t \in T_D(\ \forall s \in S_A[\ (s, e_{AtoB}[c1], sa_0)]\ )\} \cup \{new\ t \in T_D(\ \forall s \in S_A[\ (s, e_{BtoA}[c2], sb_0)]\ )\}$
- $f_D : e \in E_D \rightarrow c \in C_D$

By the delegation rule, A and B of Fig.1 is composed to D as represented in Fig.2.

5. Compatibility check of delegation rule

Michael Schrefl[4] suggests three types of properties: observation consistency, weak invocation consistency, and strong invocation consistency. They provide necessary and sufficient rules for checking behavior consistency and compatibility. Moreover, they are useful for checking behavior consistency as subtype requirement. In the delegation rule, A and B are supertypes of D. In this Section, we will show that D satisfies a subtype requirement of A and B.

First, observation consistency[4] means that each instance of a supertype must also be observable in an instance of its subtype. It requires that every possible trace (i.e. sequence of transitions) of a supertype must be observable at its subtype. Since D has all traces of A and B, D satisfies observation consistency.

Second, weak invocation consistency[4] is satisfied if one can use instances of a subtype in the same way as instances of its supertype: any sequence of transitions that can be performed on instance of a supertype can also be performed on instances of its subtype. Because D does not have any new functionality which does not appear at A and B, every sequence of transitions that can be performed on instances of A and B can also be performed on instances of D. So D satisfies weak invocation consistency.

Third, strong invocation consistency[4] requires additionally that transitions added at a subtype do not interfere with transitions inherited from the supertype. However, in the middle of trace A, (i.e. from any state of A) new event may interfere sequence of transitions and a state of any new event may move to initial state of B. Therefore, D does not satisfy strong invocation consistency.

In conclusion, D satisfies observation consistency and weak invocation consistency among these three criteria. These ensure behavior consistency of D and guarantee subtype requirements at weak invocation level.

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

[1] B. Meyer, Object-Oriented Software Construction. Prentice Hall, 1988.

[2] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen, Object-Oriented Modeling and Design, Prentice-Hall International, 1991

[3] Gilad Bracha and William Cook, " Mixin-based Inhritance," In Proceedings of ECOOP/OOPSLA' 90,1990. pp.303-311

[4] M. Schrefl and M. Stumptner, " Behavior-consistent specialization of object life cycles," ACM Transactions on Software Engineering and Methodology, vol. 11, pp. 92-148, January 2002

[5] David Harel, " Statecharts: A visual formalism for complex systems," Science of Computer Programming, p.p. 231-274, August 1987

[6] A. Taivalsaari, " On the notion of inheritance," ACM Computing Surveys, vol. 28, No.3,pp.438-479, 1996

[7] Peter Wegner, " OOPS MESSENGER," ACM PRESS, August 1990

[8] B. H. Liskov and J. M. Wing, " A Behavioral notion of subtyping," ACM Transactions on Programming Languages and Systems, vol. 15, pp. 1911-1841, November 1994