A component composition model providing dynamic, flexible, and hierarchical composition of components for supporting software evolution

In-Gyu Kim a,*, Doo-Hwan Bae a,1, Jang-Eui Hong b,2

a Department of Electrical Engineering and Computer Science, KAIST, 373-1, Guseong-dong, Yuseong-gu, Daejon 305-701, Republic of Korea
b School of Electrical and Computer Engineering, CBNU, 12, Gaeshin-dong, Heungdok-gu, Cheongju 361-763, Republic of Korea

Received 29 December 2005; received in revised form 11 February 2007; accepted 16 February 2007
Available online 4 March 2007

Abstract

Component composition is one of the practical and effective approaches for supporting software evolution. However, existing component composition techniques need to be complemented by advanced features which address various sophisticated composition issues. In this paper, we introduce a set of features that supports and manages dynamic as well as flexible composition of components in a controlled way. We also propose a component composition model that supports these features. The proposed model enables dynamic, flexible, and hierarchical composition of components by providing and manipulating dedicated composition information, which in turn increases reusability of components and capabilities for supporting software evolution. To show the benefits of our model concretely, we provide a Hotel Reservation System case study. The experimental results show that our model supports software evolution effectively and provides efficient and modular structures, refactoring, and collaboration-level extensions as well.

© 2007 Elsevier Inc. All rights reserved.

Keywords: Software evolution; Software composition technique; Component composition model; Component reuse

1. Introduction

Software evolution is inevitable in software systems used ubiquitously in many business areas with continuous market and business requirements changes (Lehman, 1998; Nierstrasz, 2002). Thus, system developers and maintainers strive to construct evolvable software systems that can address requirements changes effectively. As competition among enterprises increases, there are stronger motives for demanding adaptation techniques which enable a software system to adapt to new or changing requirements as quickly as possible.

Composition techniques (Abadi and Cardelli, 1996; Abmann, 2003; Ben-Shaul et al., 2001; Gamma et al., 1995; Kniesel, 1999; Ostermann and Mezini, 2001; Seiter et al., 1998; Truyen et al., 2001) provide great flexibility and efficiency supporting software evolution. Composition techniques also play a crucial role in component based software development (CBSD). By composing existing or customized components, a software system can be assembled as rapidly and cost-effectively as an automobile is assembled by composing machine parts (Szyperski, 1998).

* Corresponding author. Tel.: +82 42 869 3579; fax: +82 42 869 8488.
E-mail addresses: igkim@se.kaist.ac.kr (I.-G. Kim), bae@se.kaist.ac.kr (D.-H. Bae), jehong@chungbuk.ac.kr (J.-E. Hong).
1 Tel.: +82 42 869 3539; fax: +82 42 869 3510.
2 Tel.: +82 43 261 2261; fax: +82 43 273 2265.

0164-1212/$ - see front matter © 2007 Elsevier Inc. All rights reserved.
doi:10.1016/j.jss.2007.02.047
However, existing composition techniques are limited. Specifically, they only provide simple composition features such as adding, deleting, and changing components but fall short of managing dynamic, flexible, and disciplined composition of components. This limitation keeps composition techniques from being used for software evolution. It also makes it hard to develop new components or to reuse existing components by combining components or configuring composition information according to changed contexts. In this paper, we present a set of features that supports and manages dynamic and flexible composition of components systematically. We also propose a component composition model that supports these features.

This paper is organized as follows: Section 2 explains existing composition techniques and software composition research projects supporting software evolution. Section 3 proposes a set of features to support controlled management of sophisticated composition. Then, we describe a component composition model that incorporates these features in Section 4. Section 5 presents a case study using a simplified Hotel Reservation System (HRS) and discusses the pros and cons of our model. Finally, we make concluding remarks in Section 6.

2. Related work

We classify adaptation techniques and research projects that support software evolution into nine categories as shown in Table 1. The pros and cons of each category are explained briefly.\(^3\)

Among the categories, composition-based techniques are considered a powerful and cost-effective approach because of their flexibility and increased productivity. Some of the composition techniques closely related to our approach are explained below.

Ostermann and Mezini (2001) propose compound references, a new abstraction for object references, that provides explicit linguistic means for expressing and combining individual composition properties on-demand. They provide five composition properties to describe the relationship between two modules to be composed. Although these composition properties are useful for describing various relationships between two composition units, they do not address controlled configuration of composition among groups of objects.

Context relation between classes directly models dynamic evolution (Seiter et al., 1998). It introduces a new form of dynamic binding. Attaching a context object to a base object alters the base object’s method table based on class updates.

---

\(^3\) For further information, please refer to Kim et al. (2006).
defined by the context class. Context relation supports method-level updating, but not component-level updating. In addition, it does not provide sufficient reference primitives for referring to other new or existing components.

HADAS (Ben-Shaul et al., 2001) supports dynamic adaptation by modifying the structure and behavior of components. Each component is split into two sections: Fixed and Extensible. Data items and methods defined in the Fixed section are not changed during the component’s lifetime. The Extensible section is the mutable portion of the component through which its structure and behavior can be changed, and in which new methods can be dynamically added or removed. Although HADAS supports dynamic adaptation and attempts to get benefits from both class-based and instance-based changes, it can be improved further. First, it will be useful to provide more classified component types supporting services specific to various dynamic contexts. Second, in HADAS, components cannot be used as deltas (i.e., adapting roles) to adapt other components.

DC-AOP (Kim et al., 2001) is a platform, which supports dynamic composition of functions using code mobility for scalable mobile agents. However, DC-AOP does not address how added functions can access other added or existing functions. In addition, even though DC-AOP provides a simple composition model, it does not provide controlled management of sophisticated composition.

Lasagne (Truyen et al., 2001) defines a platform-independent architecture for dynamic customization of component-based systems using wrappers. In Lasagne, the composition logic is externalized from the code of clients, core system, and extensions by encapsulating it in a composition policy. Besides, an application consists of minimal functional core components and a set of potential extensions that can be selectively integrated within these core components. Each extension is implemented as a layer of mixin-like wrappers, simultaneously tailoring multiple components of the application and their interactions. Although it supports advanced concepts such as collaboration-based extension and explicit composition logic, Lasagne falls short of supporting flexible recomposition among original components and wrappers.

3. Features supporting software evolution in component composition techniques

Component composition techniques usually make use of the following primitive composition features: adding, deleting, and changing. However, such primitive features are not enough to support and manage a dynamic and flexible composition of components. They must be complemented by advanced composition capabilities addressing complex composition issues such as how to express various composition semantics. Kim et al. (2006) surveyed related literature and composition techniques, and proposed dimensions of component composition models. The dimensions, as shown in Fig. 1, can be used to classify composition techniques based on their capabilities. From the above survey of composition techniques and the comparison of results from the composition dimensions, features to manage complicated component composition in a disciplined way and to increase composition capabilities are identified and presented below.

- **Component-level variability (CLV):** To maximize reusability of software systems using components, component-level variability should be supported as well as component composition level variability. When a software system is developed by composing components, especially those made by third parties, components often do not provide the exact functionality required by the system. Thus, for effective reconfiguration and reuse of components, component-level variability should be supported.

- **Providing various configurable composition information (PVCCI):** In most composition techniques, only interface information is used for composition. However, interface information (e.g., signature information) lacks capabilities such as expressing composition semantics. For advanced or flexible composition, configurable composition information should be provided explicitly and be used and manipulated by component customers. Explicit composition information enables the customization of components and provides different behaviors by changing the composition information. That is, it enables components to be reused in different environments.

- **Managing internal component groups differently according to a specific goal (MICGD):** According to the particular characteristics of each application domain, composition properties of components in the software system should be managed differently. However, in most of previous component composition approaches, manipulating internal components differently according to goals is not sufficiently considered. For example, even if HADAS (Ben-Shaul et al., 2001) classifies components into two types: Fixed and Extensible, it is necessary to refine the classification to provide services specific to various dynamic situations such as dynamic loading or unloading of secure communication protocols. Therefore, a classification of internal components into groups according to goals and separate management of each group is recommended.

- **Supporting components as both roles – delta and original (SCBR):** In most composition techniques it is not easy to reuse delta roles from existing components to make new components. For example, in wrapper-based techniques such as

---

4 Mixin is an abstract subclass which can be applied to different superclasses to build various modified classes (Bracha and Cook, 1990). In this paper, mixin-like wrappers mean that wrappers can be composed with different (not fixed) components to provide various functions.
Adapter Pattern (Gamma et al., 1995), it is not easy for wrappers (delta roles) to adapt components other than the original components targeted by the wrappers since flexible recomposition among original components and wrappers is not supported. If components can play both roles, more reusability can be achieved by simple and uniform composition of (original) components and (delta) components.

- **Supporting expressive and changeable composition semantics (SECCS):** According to the characteristics of the domain or the goal, various composition semantics might be required or need to be changed dynamically. For example, in some mobile agent applications, certain components are composed with association semantics, other components with delegation semantics, and so on. In addition, these composition semantics might be changed at run-time. However, existing composition techniques have limited support for (re)combination of composition semantics. Most support only one or two pre-defined composition semantics such as inheritance. They do not support combination of primitive composition semantics. To address flexible composition of components, there is a need for various composition operators which provide expressive and changeable composition semantics among components.

- **Addressing the dynamic reference problem (ADRP):** When composing components, new and existing components might not know which components they can refer to. For example, when components A, B, and C are composed, component A can refer to either only component B or both components B and C. We name this reference scope decision the **dynamic reference problem**. For disciplined composition, this dynamic reference problem should be addressed.

The above features are related to various aspects of composition. The relationship between the features and the composition dimensions in Fig. 1 is shown in Table 2. The ‘□’ represents a feature with a strong relationship to the dimension while ‘□’ corresponds to a combination of composition primitives (composition capability dimension). In addition, the feature is affected by providing reference primitives (reference primitive dimension) and corresponding configuration information (required composition information dimension). The feature is affected indirectly by composition time since run-time composition or software evolution may require changing composition semantics (composition time dimension).

While these features may not be complete, they are essential elements for supporting software system evolution effectively.

### 4. A component composition model supporting dynamic, flexible, and hierarchical composition of components

In this section, we introduce our component composition model which consists of the component model (Section 4.1) and the composition model (Section 4.2) and supports the composition features presented in Section 3. Section 4.3 shows how our model addresses the features.
4.1. Component model

4.1.1. Hierarchical structure of components

It is important to raise the level of abstraction in such a way that the evolution can be expressed, reasoned about, and implemented. One way of raising the level of abstraction is a hierarchical composition of components. Hierarchical composition enables component composition to be applied uniformly to both component adaptation and application assembly in CBSD. It also allows components to be adapted by other components and allows the adapted components to be used in adapting other components as well. It increases reusability by enabling components to play both roles: original components and delta components. It is powerful and useful to be able to manage different composition levels uniformly.

For hierarchical composition our component model conforms to the composite structure in the Composite Pattern (Gamma et al., 1995). That is, components are classified into LeafComponents and FrameComponents. LeafComponent is a base component for composition and has no internal components. FrameComponent has LeafComponents or other FrameComponents as internal components. To distinguish a component from its internal components, the parent component is called as the owner component. Fig. 2 shows the relationship between LeafComponent and FrameComponent in UML (Booch et al., 1998).

“Frame” and “Leaf” are roles that any component can play. If a component (called Car_Module) is declared as a LeafComponent, it does not include any other components as internal components. However, if it is declared as a FrameComponent, it can have other components (e.g. Engine_Module) as internal components.

4.1.2. Component definition

We use the following syntactic notations:

- $id$ is a unique identifier in string format.
- $method$ is method declaration.
- `<a>^list` ::= `<a>` | `<a>`, `<a>`^list  // `<a>` is any nonterminal.

![Fig. 2. Relationship between LeafComponent and FrameComponent.](image)
LeafComponent is defined as follows:

```
<LeafComponent> ::= name_of id, type_of id, <interface>\^list
<interface> ::= method // In the future version of our model,
// <interface> will be expanded with a set of methods.
```

FrameComponent is defined as follows:

```
<FrameComponent> ::= name_of id, type_of id, <configuration>
<configuration> ::= <internal.structure.info> <interface.propagation.rule> <reference.rule>
<internal.structure.info> ::= structure <internal.component.groups> <links>
<internal.component.groups> ::= groups (<group.type> has <component>\^list)\^list
<group.type> ::= Fixed | Changeable | Optional | Additional
<component> ::= <LeafComponent> | <FrameComponent>
<links> ::= links (<link.type> has <component_pair>\^list)\^list
<link.type> ::= Association | Composition | Override
<component_pair> ::= (component, component)
<interface.propagation.rule> ::= propagations (<component>\^list has <propagation.type>\^list)\^list
<propagation.type> ::= Transparent | Blocked | cancelled
<cancelled> ::= Cancelled <interface>\^list
<reference.rule> ::= reference_rule (<component>\^list has <reference.primitive>\^list)\^list
<reference.primitive> ::= Own | AssociationLink | CompositionLink | OverrideLink |
| <visibleList> | <permittedList> | <group.primitive> | <groups>
<group.primitive> ::= Fixed | Changeable | Optional | Additional
<groups> ::= Mandatory | Basic | Frame
<visibleList> ::= VisibleList <component>\^list end
<permittedList> ::= PermittedList <component>\^list end |
| PermittedList <group.primitive>\^list end |
| PermittedList <component>\^list, <group.primitive>\^list end
```

Fig. 3 shows the constituent information of FrameComponent\(^5\) in UML format. Each component has configuration information as well as name and type information. In the remaining parts of this section, the configuration information is explained in detail.

4.1.3. Internal component group

Components should provide an ability to adapt their internal structures according to changed contexts. Our model components achieve this objective by allowing internal components to be added to a component, changed with new ones, or deleted.

However, if internal components can be changed in unrestricted ways, the owner component may lose control. For disciplined control of composition, internal components are categorized and managed differently according to the purpose and scope of each category. In our model, internal components are classified into four categories:

- A Fixed component is the most fundamental component in that it will not change. Since it provides indispensable functions, a fixed component cannot be deleted or changed into another component.
- A Changeable component exists within the owner component as does a fixed component but can be changed to another component of the same type. It can change existing components with differently implemented components.

---

\(^5\) From now on, we use the term component to mean FrameComponent, unless specified otherwise.
An Optional component can be dynamically added to (or deleted from) the owner component. It is expected to be included in the owner component at design-time although it might not be available all the time. For example, a language-specific function such as English is not necessary for an editor component all the time. When the language-specific function is required, the editor component obtains the corresponding language component and uses it. Additionally, to reduce the memory space and network traffic especially in mobile agent applications, optional components can be added and deleted using code mobility when necessary.

An Additional component satisfies “unanticipated” requirements. An owner component can add, at run-time, new components to its internal components even though such addition is not anticipated at design-time. For example, when upgrading a component with a monitoring function like saving log information, a new internal component, called Monitor, can be added into the owner component. The additional component is different from the optional component in that it is not anticipated by the original designer.

These four groups of internal components may not be a complete classification for all application domains. If, in a particular domain, specific component groups other than the above four groups are required, new groups can be accommodated by defining, adding, and using them accordingly. The point we want to emphasize is that for disciplined control of composition, internal components should be categorized and managed according to specific group characteristics.

4.1.4. Internal link type

In our model, internal components have various relationships. The following internal link types, each of which has different composition semantics, are proposed:

- Association link: If there is an association link between two components, one component can invoke services (or interfaces) of the other component and vice versa.
- Composition link: If there is a composition link between two components, the added component can communicate with the adding component and vice versa. This link is used between dynamically added internal components and adding internal components.
- Override link: If there is an override link between a super-component and a sub-component, the sub-component can use interfaces of the super-component as if the sub-component inherits the super-component. That is, the sub-component can use interfaces of the super-component if the sub-component does not already have them. The sub-component can also override the interfaces of the super-component if both have the same interfaces.

4.1.5. Interface type

In our model, interfaces are separate from the implementation and are divided into provided and frame interfaces as shown in Fig. 4.

- Provided interface: Provided interfaces are the interfaces for (external) components to access services.
• Frame interface: Frame interfaces are composed of the provided interfaces of all internal components within an owner component (FrameComponent) and can be accessed by any internal component of the owner component. When an owner component is created or changed by (re)composing internal components, all interfaces need not be provided to external client software systems. Some of them can be provided to external client software systems as provided interfaces of its owner component. However, internal components need to access all internal interfaces to perform their jobs. For an internal component to access all internal components freely, frame interfaces are proposed by our model. They enable an internal component to access services of other internal components. Since our model enables an owner component to change internal components at run-time, frame interfaces need to be changed accordingly. Specifically, when a new internal component is added, its interfaces are added into frame interfaces. Similarly, when the added internal component is deleted, its interfaces are deleted from frame interfaces.

FrameComponent has both types of interfaces: provided and frame interfaces. On the other hand, LeafComponent has only provided interfaces because it has no internal components. The following relation exists between the two interface types of a FrameComponent: \{provided interfaces\} ⊆ \{frame interfaces\}. Thus, provided interfaces are used for external clients and frame interfaces are used for internal clients.

4.2. Composition model

In this subsection, a composition model using our scheme is explained. The composition model, together with the component model, enables dynamic composition and increases reusability of components through hierarchical composition and through reconfigurable information such as reference rules and interface propagation rules.

4.2.1. Composition construct

For dynamic composition of internal components, the following constructs are provided:

• Add: A component is added to an owner component.
• Delete: An existing component is deleted from its owner component.
• Change: An existing component is changed to a new component of the same type.

4.2.2. Interface propagation rule

Frame interfaces of an owner component include all provided interfaces of its internal components. However, all frame interfaces of the owner component need not be provided interfaces of the owner component. Usually, only some of the frame interfaces are mapped into the provided interfaces. Interface propagation rules specify the mapping between frame interfaces and provided interfaces of the owner component. Interface propagation rules are provided as follows:

• Transparent: All provided interfaces of an internal component become provided interfaces of the owner component. Full propagation of provided interfaces from the internal component to the owner component is performed.
• Blocked: No provided interfaces of an internal component are permitted to become provided interfaces of the owner component. All propagations are blocked.

Section 4.2.2 shows the mapping rules between frame interfaces and provided interfaces.
• Cancelled (selectively blocked): Provided interfaces of an internal component in Cancelled clauses are not allowed to become provided interfaces of the owner component. For example, if an internal component has a rule “Cancelled:interface1,” all interfaces except interface1 become provided interfaces of the owner component. Thus, cancelled allows partial propagation.

4.2.3. Reference rule

Each internal component in the owner component can refer to other internal components in the owner component.7 Reference rules specify which internal components can be referred to by an internal component. The reference rule is defined as follows:

```
<reference_rule> ::= reference_rule (<component>list has <reference_primitive>list)list
<reference_primitive> ::= Own | AssociationLink | CompositionLink | OverrideLink |
                       <visibleList> | <permittedList> | <group_primitive> |
<group_primitive> ::= Fixed | Changeable | Optional | Additional
<groups> ::= Mandatory | Basic | Frame
<visibleList> ::= VisibleList <component>list end
<permittedList> ::= PermittedList <component>list end |
                   PermittedList <group_primitive>list end |
                   PermittedList <component>list, <group_primitive>list end
```

Reference primitives are explained in the following:

• Own: A component can use its own interfaces. If a component does not have the Own primitive, it cannot refer to its own interfaces.
• AssociationLink: If a component has the AssociationLink primitive, it can use the interfaces of the associated components (i.e., components having an internal link of Association with it). Similarly, there are CompositionLinks which are applied to optional or additional components.
• OverrideLink: If two components have OverrideLink, a later added (or changed) component overrides the early added (or changed) existing component. Assume a component (called Child) has an OverrideLink relation with a component (called Parent). Suppose Child is requested to provide a service by clients but Child has no interfaces for the service while Parent does have interfaces for the service. Child can forward the request to Parent and Parent will provide the service instead of Child. In addition, if both Child and Parent have the interface for the service, the interface of Child is executed instead of that of Parent (i.e., the interface of Child overrides the interface of Parent).
• VisibleList: VisibleList specifies other internal components that can be referred to. That is, an internal component can refer to other internal components listed in its VisibleList.
• PermittedList: PermittedList of an internal component (called A) specifies internal components which can refer to it (i.e., A). Only those internal components included in PermittedList can refer to the internal components of A. In other words, if an internal component has PermittedList, it allows only internal components in PermittedList to refer to it. Any internal component that is not in PermittedList cannot refer to the component. This primitive has a different characteristic than other primitives in that it does not specify which other internal components an internal component can refer to, but rather specifies which other internal components can refer to it.
• Fixed: Internal component groups can be used as useful reference primitives. For example, some internal components like a potentially harmful newly added component need to be restricted to refer internal components within a specific component group such as Fixed. The Fixed reference primitive enables components to refer to internal components in Fixed category (i.e., fixed components). Similarly, there are Changeable, Optional, and Additional primitives.
• Mandatory: To provide a way to access internal components which usually provide fundamental services and always exist within the owner component, the Mandatory reference primitive is proposed. That is, if an internal component has a Mandatory reference primitive, it can access fixed and changeable components.
• Basic: Basic reference primitive enables an internal component to refer to fixed, changeable, and optional components, which are modelled at design-time and thus can be considered as basic functions.

---

7 In this paper, “refer” means that components can use the interfaces of the referred components.
• Frame: To provide a way of accessing all internal components, Frame reference primitive is proposed. If an internal component has a Frame reference primitive, it can refer to all internal components including additional components.

Mandatory, Basic, and Frame are syntactic sugars. For example, Mandatory can be expressed by Fixed and Changeable. Relationships among Mandatory, Basic, Frame, and other primitives (Fixed, Changeable, Optional, Additional) are shown graphically in Fig. 5.

Reference rules enable components to provide different behaviors in diverse contexts. Fig. 6 shows a component which conforms to our model. Using this component, we explain how reference primitives can produce the various behaviors of the component. Two sample reference rules of the example component can be expressed as follows:

![Diagram](image-url)

**Fig. 5.** Relationship among Mandatory, Basic, Frame, and other primitives.

![Diagram](image-url)

**Fig. 6.** An example component conforming to our component composition model.
The internal components in Fig. 6 could have various reference rules like the above two reference rules. Parts of possible sample reference rules are introduced and explained in the following:

- **C1 has Own**: In this case, C1 can use its own interfaces such as I_1A and I_1B.
- **C1 has AssociationLink**: C1 can refer to associated components C2 and C3.
- **C1 has PermittedList C2 end**: Only C2 can use the interfaces of C1. Other components cannot refer to C1.
- **C4 has VisibleList C2, C3 end**: C4 can refer to components in VisibleList (i.e. C2 and C3).
- **C4 has Mandatory**: C4 can refer to mandatory components C1–C3.
- **C6 has Frame**: C4 can use frame interfaces. That is, C4 can refer to all components C1–C6.
- **C6 has CompositionLink**: C6 can refer to parent or ancestor components in the composition link (i.e. C1 and C4).

If a component has a reference rule composed of various reference primitives, the component can refer to all components that can be referred to by the primitives. More complicated reference rules can be expressed by composing various reference primitives. However, for the soundness of the composed reference primitives, conflicts should be checked.

A programming infrastructure (API) for our proposed model has been implemented by Java language.\(^8\)

### Table 3
Composition features in Section 3 and our component composition model

<table>
<thead>
<tr>
<th>Composition feature</th>
<th>Features of our model</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hierarchical structure of components</td>
<td>Internal component group</td>
</tr>
<tr>
<td>CLVa ^a^</td>
<td>O</td>
</tr>
<tr>
<td>PVCCb ^b^</td>
<td>O</td>
</tr>
<tr>
<td>MICGDc ^c^</td>
<td>O</td>
</tr>
<tr>
<td>SCBRd ^d^</td>
<td>O</td>
</tr>
<tr>
<td>SECCSe ^e^</td>
<td>O</td>
</tr>
<tr>
<td>ADRPf ^f^</td>
<td>O</td>
</tr>
</tbody>
</table>

^a^ Component-level variability.
^b^ Providing various configurable composition information.
^c^ Managing internal component groups differently according to a specific goal.
^d^ Supporting components as both roles – delta and original roles.
^e^ Supporting expressive and changeable composition semantics.
^f^ Addressing the dynamic reference problem.

```javascript
// Sample Reference Rule #1
C1, C2, C3 has Own, AssociationLink,
C4 has Own, Mandatory,
C5 has Frame,
C6 has Own, Mandatory, CompositionLink

// Sample Reference Rule #2
C1 has Own, AssociationLink,
PermittedList C2, C4, C5, C6 end,
C2, C3 has Own, VisibleList C1 end,
C4, C5, C6 has Mandatory
```

The internal components in Fig. 6 could have various reference rules like the above two reference rules. Parts of possible sample reference rules are introduced and explained in the following:

- **C1 has Own**: In this case, C1 can use its own interfaces such as I_1A and I_1B.
- **C1 has AssociationLink**: C1 can refer to associated components C2 and C3.
- **C1 has PermittedList C2 end**: Only C2 can use the interfaces of C1. Other components cannot refer to C1.
- **C4 has VisibleList C2, C3 end**: C4 can refer to components in VisibleList (i.e. C2 and C3).
- **C4 has Mandatory**: C4 can refer to mandatory components C1–C3.
- **C6 has Frame**: C4 can use frame interfaces. That is, C4 can refer to all components C1–C6.
- **C6 has CompositionLink**: C6 can refer to parent or ancestor components in the composition link (i.e. C1 and C4).

If a component has a reference rule composed of various reference primitives, the component can refer to all components that can be referred to by the primitives. More complicated reference rules can be expressed by composing various reference primitives. However, for the soundness of the composed reference primitives, conflicts should be checked.

A programming infrastructure (API) for our proposed model has been implemented by Java language.\(^8\)

### 4.3. Our model and composition features

Table 3 shows how our model addresses the composition features presented in Section 3. “O” means that the composition feature is mainly supported by the model feature. For example, the composition feature of supporting expressive and changeable composition semantics is mainly addressed by reference rules in our model.

### 5. A case study: Hotel Reservation System (HRS)

In this section, we introduce one of the case studies, Hotel Reservation System (HRS) that provides customers with basic hotel services such as reservation and check-in. A sequence of requirements changes is applied one by one to this HRS to demonstrate how the proposed component composition model can support requirements changes effectively.

\(^8\) For more information, please refer to Kim and Bae (2005b).
5.1. Baseline implementation of HRS

The baseline version of HRS is implemented using our model, JavaServer Pages (JSP), Java, and MySql. The use case diagram of the baseline HRS is shown in Fig. 7. The corresponding structural model including Java classes and JSP files of the HRS is shown in Fig. 8.9

5.2. Applying requirements changes and evaluations

To demonstrate the usefulness of our model, various requirements changes are applied to the baseline HRS. After applying the requirements changes one by one, the resulting model is analyzed. Some of the results, which show the benefits of our model more effectively, are presented as follows:

5.2.1. Requirement change 1 (RC1)

Two room services (video ($10/a video), dinner ($100/person)) are added to the baseline HRS.

5.2.2. Applying RC1

We add this requirement to change the baseline HRS in two ways. In one way (RC1_version1), HotelMgt LeafComponent in the baseline HRS (Fig. 8) is changed into HotelMgt FrameComponent with HotelInfoMgt and RoomMgt as internal components (Fig. 9). RoomMgt provides room-service related functions and HotelInfoMgt is explicitly refactored out (modeled) for future extensions and plays the same role as HotelMgt in the baseline HRS. In addition, methods related to room services are added (CheckInMgt) or modified (CheckOutMgt). Fig. 9 shows the changed HRS system in UML class diagram.10 In Fig. 9, ovals over the class diagram focus the related parts. Stereotype <<contain>> in associations indicates fixed relationship. For example, HotelMgt has HotelInfoMgt and RoomMgt as fixed internal components.

In the other way (RC1_version2), the HRS of RC1_version1 is refactored as shown in Fig. 10 to enhance extensibility. CheckInMgt LeafComponent in Fig. 9 is changed into CheckInMgt FrameComponent in Fig. 10 to make it more flexible. It can have other components (e.g., CheckInMgtCore and RoomInfoOfReservation) as internal components, which allows systems to be extended flexibly by changing internal components or (re)configuring the composition information. Also, methods related to room services are factored into an explicit component (RoomInfoOfReservation).

5.2.3. Evaluation of RC1

The first result (RC1_version1) shows that our model allows easy addition of a new function (i.e., room services) through a new component (i.e., RoomMgt) and including it into an existing component (HotelMgt).

The second result (RC1_version2) shows our model also supports refactoring of existing components (e.g., CheckInMgt). In addition, it enables components to be customized easily by changing internal components or configuring com-

---

9 Since JSP files play application-level roles as well as presentation-level roles, both Java classes and JSP files are necessary to present structural aspect of HRS.
10 JSP files are not shown for the brief and clear explanation.
Fig. 8. Structural model of the baseline HRS.

Fig. 9. RCI_version1: HRS after applying RC1.
position information. For instance, CheckInMgt FrameComponent can be used differently: allowing access to RoomInfoOfReservation services or prohibiting access. The former can be used in the new HRS to provide room services and the latter can be used in the baseline HRS.

5.2.4. Requirement change 5 (RC5) ¹¹

The existing check-out service is upgraded to provide a currency exchange service (e.g., Dollars into Euros).

5.2.5. Applying RC5

To change the existing check-out service to provide a currency exchange service, CheckOutMgt LeafComponent shown in Fig. 8 is replaced by the components in Fig. 11. The currency exchange service is provided by CurrencyConverterDCM component, which can be changed into concrete components such as CurrencyConverterDollarsIntoEurosDCM. ¹²

5.2.6. Evaluation of RC5

We observe that our composition approach has the following benefits over the traditional approaches of changing functions of components directly:

- Our model enables developers to model the currency exchange service by explicit components and reuse them in other software systems.

---

¹¹ RC5 means 5th requirement change applied in the HRS case study. Information of requirements changes such as RC2, which are not explained here can be found in Kim and Bae (2005a).

¹² DCM components means that components are implemented using our component composition model. DCM stands for dynamic component composition model.
Our model can provide as many benefits as the Strategy pattern (Gamma et al., 1995) due to the dynamic composition capability of DCM components. That is, our model provides the specific currency exchange service at run-time by dynamic composition of components.

Our model enables HRS to control the currency exchange service in a disciplined way by configuring composition information such as reference rules. For example, CheckOutMgtCore can access CurrencyConverterDCM only if its reference rule allows (e.g., Frame). Therefore, our model provides developer with a flexible composition mechanism.

5.2.7. Requirement change 9 (RC9)
This requirement adds a notification service. When there is a request for deletion of VIP customer information, HRS should notify staff of the deletion request immediately.

5.2.8. Applying RC9
We implement this change in two ways: inheritance and our model. With inheritance, CustomerMgt is inherited by CustomerMgtWithNotification which overrides deleteCustomer method as shown in Fig. 12 (left part). Then, existing JSP files using the notification service are changed so they refer to a new subclass CustomerMgtWithNotification instead of CustomerMgt.

With our approach, shown in Fig. 12 (right part), CustomerMgt has two components in its internal components: CustomerMgtCore and DeleteCustomerNotificationDCM. Our approach enables a client-side software system (e.g., DeleteCustomerWC.jsp) to maintain the same reference as before since CustomerMgt can provide the notification service by having DeleteCustomerNotificationDCM as an internal component. That is, by having DeleteCustomerNotificationDCM as an internal component, CustomerMgt incorporates the notification service and the client-side software system can use the service without changing the reference to CustomerMgt in contrast to the inheritance approach.

5.2.9. Evaluation of RC9
When a new service is added, using traditional OO inheritance, existing clients often use a new subclass for the new service. However, it is inconvenient and error-prone to change the related client-side software system whenever a new service is added or deleted. In our approach, if a new service is added, the client-side software system can use the new service without changing the reference to CustomerMgt by adding (or changing) the corresponding new internal components into CustomerMgt. Our approach enables the client-side software system to be maintained easily. Furthermore, to accommodate a new service, inheritance makes a fixed hierarchy structure of classes such as deep inheritance hierarchy or widen inheritance hierarchy. In contrast to inheritance, our approach manages the hierarchy structure flexibly by dynamically composing DCM components and configuring composition information.

5.2.10. Requirement change 12 (RC12)
This requirement change provides the currency exchange service in the reservation subsystem. In RC5, the currency exchange service is added to the check-out service. In RC12, the same currency exchange service is applied to the reservation service.
5.2.11. Applying RC12

We implement this requirement change in two ways: reusing existing components and increasing reusability by refactoring. In the first way, since currency exchange components are already used in the check-out subsystem (RC5), we reuse the currency exchange components such as CurrencyConverterDCM. To reuse them, we transform ReservationMgt into FrameComponent which has ReservationMgtCore and CurrencyConverterDCM as internal components. In addition, we add cost-calculating methods using currency exchange components into ReservationMgtCore. Fig. 13 shows how the existing adapters (e.g., CurrencyConverterDCM) are reused in other adaptees (e.g., ReservationMgt).

In the second way, some refactorings in the first HRS are performed to find and use reusable components. First, functions used to calculate expected cost in both the reservation and check-out subsystems are modelled as a reusable component (CostCalculationDCM). Second, the reservation and check-out subsystems are refactored to use CostCalculationDCM. Fig. 14 shows a part of the HRS after refactorings.

5.2.12. Evaluation of RC12

One of the benefits of our approach is it makes refactoring easy. For example, methods of ReservationMgtCore in Fig. 13 could be reorganized into two components: ReservationMgtCore and CostCalculationDCM in Fig. 14. Calculation-related methods are easily (with little modification) moved into CostCalculationDCM because interfaces of internal components can be incorporated into those of the owner component by propagation rules. Another benefit is our approach increases reusability. In the first version, the reservation subsystem reuses the existing currency exchange components easily. In addition, after refactoring calculation-related methods into CostCalculationDCM, both the reservation and check-out subsystems are refactored to use CostCalculationDCM. Fig. 14 shows a part of the HRS after refactorings.

5.2.13. Final class diagram after applying all RCs

The final class diagram of HRS after incorporating all changes is shown in Fig. 15. A detailed evaluation of our model through a case study can be found in Kim and Bae (2005a).

5.3. Discussion

This subsection discusses the pros and cons of the proposed component composition model through the HRS case study. The baseline HRS was implemented based on our model and various requirements changes were applied to the baseline HRS. Through the HRS case study, the following benefits of our model are identified:

- easy addition of new functionality (RC1),
- supporting modular structure effectively (RC5, RC9, RC10),

13 Some of them are not explained in this paper due to lack of space. For further information, please refer to Kim and Bae (2005a).
supporting flexible customization by changing configuration information (RC1),
unburdening client software system adaptation jobs (RC9),
supporting dynamic composition of unexpected functionality (RC11),
ingcreasing reusability (RC10, RC12),
expressive composition semantics (RC10),
making refactoring easy (RC1, RC12),
supporting collaboration-based extensions easily (RC13).

Our approach has some drawbacks. First, when composing components, it is required to perform conflict checks (e.g. reference rule conflict check). Second, performance issues can be raised because services in the leaf internal components could be provided through several indirect invocations to the end users. Third, while (re)composing lots of components in various ways increases flexibility of systems, it might make inherent complexity of system (re)composition high. Our model needs to reduce this complexity further. Some of the above drawbacks will be explained more specifically in the further research issues of the following section.
6. Conclusion and further work

To support evolution in a software system, we believe component composition is a practical and effective way. This composition approach is useful especially for component-based software systems which are mainly based on the composition of existing components. However, existing component composition techniques have limitations in effectively supporting the evolution. In this paper, we suggest features which help support and manage dynamic, flexible, and disciplined composition of components. These include component-level variability, providing various configurable composition information, managing internal component groups, supporting components as both roles, supporting expressive and changeable composition semantics, and addressing the dynamic reference problem. In addition, we propose a new component composition model satisfying these features. In our model, components explicitly include reconfiguration information such as interface propagation rules and reference rules so components can be reused in various software systems by changing the information. Our model also supports hierarchical composition of components, which enables components to assume adapter and adaptee roles uniformly, which in turn supports various composition levels from component-level variability to application-level variability. Hierarchical composition also increases modularity. That is, highly cohesive and lowly coupled component relations can be maintained by including related components as internal components. After performing case studies like HRS, in addition to the above benefits, the following benefits of our model are also identified: dynamic composition of components, easy addition of new functionality, supporting modular structure effectively, unburdening client-side software systems, supporting dynamic composition of unexpected functionality, making refactoring easy, and supporting collaboration-level extensions.

There are several research issues worth considering further. First, when reference primitives are composed, conflict checking should be done. There could be conflicts between reference primitives because each has its own characteristics.
For example, AssociationLink and CompositionLink cannot be assigned to a relation between two components at the same time. Some reference primitives have a priority ordering. That is, if two reference primitives are applied at the same time, one primitive will have a higher priority than the other. Through automatic conflict checking, run-time reference errors could be prevented and reliable composition is possible. Second, through more case studies, some general problems which occur often in our model could be identified and collected and corresponding composition patterns could be proposed. Various composition patterns would enable developers to understand complex composition more easily and exchange their ideas using a common vocabulary. Third, domain specific classification of internal components other than Optional and Additional components could be considered. For example, an internal group corresponding to a collaboration-based extension “secure-context” can be considered. Such group could be useful. Consider, for instance, internal components in an owner component belonging to secure-context group. Components in this group have a security role whose interfaces are login(), logout(). When “Component_Group = {secure-context}” is enabled in a reference rule for another internal component of the owner component, the internal component can access all internal components in “secure-context” group. That is, by using the group, all internal components supporting the security role can be accessed and managed collectively. Fourth, it is necessary to decrease complexity in managing many explicit components. Our model models the functions of a system as explicit (fine or coarse grained) components and combines them into larger components. Explicitly modelled components give the system a more modular structure and make refactoring of the system easier. As more functions are required or changed, more explicit components might be necessary. In our model, such a complexity is managed by modular structuring of systems especially by the hierarchical composition of components. However, managing many components and increasing the flexibility of systems composed from them make the inherent complexity of systems high. To decrease the complexity, our model could be complemented by methods such as domain driven design strategies and patterns (Evans, 2004), software architecture and reusable assets (Bosch, 2000), and product line engineering (Clements and Northrop, 2002).

Acknowledgements

This work was supported by the Ministry of Information and Communication, Korea, under the Information Technology Research Center (ITRC) Support Program. We thank the anonymous reviewers of the earlier version of this paper for highly valuable suggestions and comments.

References

CORBA Components, 2002. v3.0 full specification. OMG.
Gamma, E., Helm, R., Johnson, R., Vlissides, J., 1995. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley.

