COMPOSITION OF ASPECTS BASED ON A RELATION MODEL: SYNERGY OF MULTIPLE PARADIGMS

IKJOO HAN* and DOO-HWAN BAE†

Division of Computer Science,
Department of Electrical Engineering and Computer Science,
Korea Advanced Institute of Science and Technology (KAIST),
373-1, Guseong-dong, Yuseong-gu, Daejeon, 305-701, Korea
*ijhan@salmosa.kaist.ac.kr
†bae@salmosa.kaist.ac.kr

Software composition for timely and affordable software development and evolution is one of the oldest pursuits of software engineering. In current software composition techniques, Component-Based Software Development (CBSD) and Aspect-Oriented Software Development (AOSD) have attracted academic and industrial attention. Black box composition used in CBSD provides simple and safe modularization for its strong information hiding, which is, however, the main obstacle for a black box composite to evolve later. This implies that an application developed through black box composition cannot take advantage of Aspect-Oriented Programming (AOP) used in AOSD. On the contrary, AOP enhances maintainability and comprehensibility by modularizing concerns crosscutting multiple components but lacks the support for the hierarchical and external composition of aspects themselves and compromises the important software engineering principles such as encapsulation, which is almost perfectly supported in black box composition. The role and role model have been recognized to have many similarities with CBSD and AOP but have significant differences with those composition techniques as well. Although each composition paradigm has its own advantages and disadvantages, there is no substantial support to realize the synergy of these composition paradigms; the black box composition, AOP, and role model. In this paper, a new composition technique based on representational abstraction of the relationship between component instances is introduced. The model supports the simple, elegant, and dynamic composition of components with its declarative form and provides the hooks through which an aspect can evolve and a parallel developed aspect can be merged at the instance level.

Keywords: Software composition; aspect-oriented programming; black box composition; component-based software development; role; relation model; logic.

1. Introduction

Component Based Software Development (CBSD) is one of the most recognizable current development techniques for building large software in a timely and affordable manner. In this technique, a software application is composed of prefabricated, reusable and independently developed pieces of software called components. Since components have usually been taken as a black box developed by third parties,
the black box composition attracts much focus and emphasis in CBSD described in [1]. In order to support safe modularization, the black box composition limits the knowledge of the components available to software composers to a set of their provided and required interfaces and the composer of components need not know their detailed implementations and components. The safeness of the black box composition is gained through this strong information hiding. The information hiding provided by the black box composition, however, limits the evolution of the composition and the deployments of new aspects to the composition, especially when the composite is considered as a component for later composition (the composer for this later composition might be also a third party). At best, the boundary where a new aspect can be applied is limited only to the interfaces of the composite.

Aspect oriented programming (AOP) is one of the most promising programming paradigms for post object-oriented programming. Aspect in AOP is to isolate and encapsulate the concern crosscutting software components which are modularized with a conventional software composition technique such as object-orientation. The common basic elements required in AOP techniques are the join point and advice. The join point is a hook of a program structure to which the advices of a crosscutting concern are attached and the advice is a specified behavior pertinent to the concern. Current AOPs like AspectJ and Hyper/J introduce aspects consisting of those elements to a base application at compile-time or run-time depending on their weaving environments. To achieve this flexibility, they, however, lose valuable modular mechanisms such as encapsulation, and hierarchical and external composition and compromise the ability to compile or reason about a modular unit separately [4]. The other shortcoming of current AOP techniques is that they cannot selectively apply aspects at the instance level.

The role and role model have many similarities with CBSD and AOP. In this sense, a role model has been implemented in AspectJ as in [14] but lost some of its important characteristics such as dynamicity. The role model with some enhanced features has been considered as suitable to support the main abstraction concepts of software architecture of CBSD [8]. However, most of role and role model have been considered to have many differences with other composition paradigms as well.

The black box composition, AOP, and role model have many similarities but differences as well. They have their own advantages and disadvantages. In this paper, we introduce a new approach to black box composition based on the relationship between the component instances, a kind of knowledge relevant to composition in order to achieve the synergic benefits of three composition paradigms. The relation model which we propose abstracts and encapsulates the relationship between component instances in a black box composite. The consequent model is expressed in the declarative forms based on the extension of the logic predicate which are accompanied with their behaviors. The abstraction of the relationship between component instances of a composition becomes the hook where the behavior of the composition can be attached to or replaced at the instance level for new concerns crosscutting component instances without breaking the encapsulations of the composition. Thus,
our relation model provides the modularization of black box composition, the flexibility of AOP, and the instance level dynamicity of the role model. Besides, it is robust to the change of relationship between components to some degree for the declarativeness of our relation model.

In the following section, the background of our work is introduced and current composition techniques are compared concisely. Then, our relation model and its elements are presented with an extension of an object-oriented language and accompanied by examples showing how the composition based on our relation model synergies the strengths of the different composition techniques in the subsequent section. Brief discussions including implementation issues and related works follow.

2. Background and Composition Techniques

One of the essential entities required for software composition is the manifestation of the relationship and the interaction mechanism among components while their representation and usages are different in composition techniques. The relationship defines which components should participate in the composition and which component interacts with which component. In the classical procedural paradigm, components are modules like functions and procedures. Their relationship based on procedure call mechanism is expressed in a call graph such as the structure chart of SA/SD (Structured Analysis/Structured Design). To change the relationship between the components of a composite, the invasive and direct modification of code is necessary in this paradigm.

The object-orientation uses various kinds of relationship between classes including association and generalization/specialization. With these relationships, classes can be variously assembled to make software applications, which can be said composites. The inheritance of object-orientation is, however, not a sufficient solution for the safe software composition and cannot guarantee the encapsulation of its components. Conventional object-orientation has the inherent problems in its inheritance mechanism such as the diamond inheritance and the fragile base class problems [1] and is criticized for the lack of global control abstraction as its first class entity in the literature [9, 13].

Many researches about collaboration based development try to support the global structure and control abstractions of an object-oriented system by separating the relationship and interaction between objects from their intrinsic behavior [4, 8, 9, 18]. They take those abstractions as their important encapsulation entities. The role and role model directly support these separation and encapsulation. A role of an object is defined as a set of extrinsic properties enabling the object to behave in a certain way expected by a set of other objects in collaboration [5]. The extrinsic properties of a role involve the extrinsic attributes and behavior, and the relationship between its intrinsic object and other objects. A set of roles are designed and/or implemented to support a collaboration as a role
model [5–8, 10, 17]. The main characteristics of the role and role model can be listed as follows:

- **Identity**: A core object and its roles have the same identity.
- **Dependency**: A role cannot exist without its core object.
- **Dynamicity**: Roles can be added to and removed from their core objects at run-time.
- **Multiplicity**: An object can have multiple roles at a point in time, no matter whether they are of different kind or of the same kind.
- **Role of role**: A role can become a core object of the other role.
- **Abstractivity**: Roles can be classified and generalized/specialized in a role hierarchy.

In CBSD, a component is taken as a set of interfaces, no matter whether it implements the interfaces directly or indirectly. A composite hides its implementation and the components from which it is composed behind its interfaces. By allowing the participation of any component satisfying the interfaces, a composite can be composed in a black box manner. The relationship between interfaces of the constituent components is defined as the connections of their required and provided interfaces within a composite. Thus, as long as a composite is treated as a black box and does not reveal the internal relationship of its components, the composite in CBSD does not provide sufficient information to define join points where a new crosscutting concern about its constituent components is introduced or an existing crosscutting concern evolves.

AOP separates and encapsulates the extrinsic properties for a concern crosscutting components from the intrinsic properties of components. AspectJ and Hyper/J are recognizable AOPs. Aspects are attached through join points over the components by a weaver at compile-time or run-time depending on their external environments as in AspectJ [2, 15] or composed into an application through generic rules in the configuration files called hypermodules at load-time in Hyper/J [3]. The abstractivity of aspects is supported by the aspect hierarchy in AspectJ which is similar to the class hierarchy of object-orientation or the definition of the dependencies of configuration files in Hyper/J. With these features, current AOP can support the introduction of new crosscutting concerns over components and the evolution of existing crosscutting concern. Current AOP does not, however, support the hierarchical and external composition of aspects and the encapsulation between aspects and their base application satisfactorily. AspectJ has no means to directly support the hierarchical and external composition of aspects which can be used at the design or implementation phases though it provides the aspect hierarchy. While a hyperslice encapsulating a crosscutting concern can be composed from hypermodules in Hyper/J, Hyper/J offers no support for encapsulation in that the composition dependency is defined only in separate configuration files and the local maintenance of a hyperslice in a composition dependency is difficult to
handle without the help of the Hyper/J tool. The problem of breaking encapsulation in AspectJ is explained and addressed in [16]. The refactoring of a base application influences the related aspects relying on the pointcuts such as those of AspectJ which are extracted from the structure of the base application. Although the external environment can manage the generation of pointcuts for the refactored application as in [16], aspects written in AspectJ cannot be applied by themselves without changed base classes. In other words, aspects in AspectJ cannot be taken as self-contained components. While AspectJ and Hyper/J have their own weakness as in encapsulation and external composition, aspectual collaboration which is usually taken as an AOP technique addresses this problem by combining modular programming (MP) and AOP. An aspectual collaboration can be composed from other constituent aspectual collaborations and supports hierarchical and external composition [4].

A role model and AOP have many similarities since these techniques are intended to supplement object-orientation with the support of the localized manipulation of concerns crosscutting multiple components. A role model in role based development can be taken as a collaborative crosscutting concern in AOP. There are, however, many differences between those two techniques as investigated in [12]. Identity is the most essential one. Aspect in AOP is not instantiated and is not related to its or its components identities. Therefore, the dynamic bindings of aspects at the instance level are impossible even though the dynamic weaver is provided as in [15]. Aspect binding at the instance level would make it possible that a component instance selectively participates in a set of instances of aspects and an instance of an aspect can be composed from other instances of the same or different aspects. On the contrary, the role model has the feature making the dynamic aspect composition at the instance level possible. Role of role feature in the role based system can be extended for role models to be composed at the instance level.

In some role based systems, roles in a role model can be separately instantiated [5, 6, 10, 17]. This leads to role binding anomaly found in [8]. Role binding anomaly is a problematic phenomenon such that the structural and behavioral constraints defined in a role system can be violated during the role-binding phase. If a role model has global architecture for a role system as in [8], role binding anomaly and the lack of aspect proclamation described in [12] can be alleviated.

Role models with the architecture for the collaboration of roles are suitable to support the main abstraction concepts of CBSD: gross structure abstraction and global control flow abstraction [8]. The core object of a role corresponds to the component in the black box composition for CBSD and a set of roles for a collaboration corresponds to the sub-architecture. Additionally, the dynamic binding of role can support the dynamicity of black box composition in CBSD.

AOP and black box composition in CBSD have conflicting characteristics. Black box composition hides its inside, whether it is a simple component or a composite. In AOP, it is necessary to use the inside structure of a component in order to define
Table 1. Comparison of component relationship in programming paradigms.

<table>
<thead>
<tr>
<th>Programming paradigm</th>
<th>Component</th>
<th>Relationship between components</th>
<th>Binding time of relationship</th>
<th>Evolution of the inside of a composite</th>
</tr>
</thead>
<tbody>
<tr>
<td>Procedural</td>
<td>Function</td>
<td>Function call</td>
<td>Static</td>
<td>Direct modification</td>
</tr>
<tr>
<td>Object-oriented</td>
<td>Class</td>
<td>Association and specialization/ generalization</td>
<td>Static</td>
<td>Inheritance</td>
</tr>
<tr>
<td>Role-based</td>
<td>Object</td>
<td>Role model</td>
<td>Dynamic</td>
<td>Inheritance in role hierarchy</td>
</tr>
<tr>
<td>Aspect-oriented</td>
<td>Class</td>
<td>Aspect</td>
<td>Depending on weaving technique</td>
<td>Inheritance in aspect hierarchy or modification of configuration dependencies</td>
</tr>
<tr>
<td>Component-based</td>
<td>Set of interfaces</td>
<td>Provided and required interfaces</td>
<td>Dynamic</td>
<td>White box manner depending on the implementation of composite</td>
</tr>
</tbody>
</table>

an aspect. These conflicts must be mitigated in order to apply AOP in black box composition.

Table 1 summarizes the differences and the similarities among current software composition techniques in terms of the relationship among components and the evolution of the composition. We can summarize the advantages and disadvantages of three paradigms which can be extracted from Table 1 and the discussions in this section as follows:

- AOP is good for declaring and handling concerns cutting across components but has its weaknesses in the encapsulation and the external and dynamic composition at the instance level.
- Black box composition is an incarnation of encapsulation and supports the hierarchical and external composition. It has, however, too strong information hiding to support the evolutions of the insides of compositions and the introductions of new crosscutting concerns.
- Role and role model support the dynamic composition at the instance level best but lack the crosscutting abstraction and its implementation.

No matter what they are called in those paradigms, a crosscutting concern must involve the relationship among components and the specified behavior related to the concern. In the next section, we will introduce a new composition paradigm based on our relation model, which supports the dynamic aspect composition at the instance level in a black box manner.

3. Aspect Composition Based on Relation Model

In this section, we present the relation model we have devised and how our relation model can overcome the weaknesses of current composition techniques and synergize their strengths with a simple running example.
3.1. Running example: university information system

Our running example is an integrated information processing system for a university system which has multiple campuses scattered over a country. The system must handle a number of concerns and they may be developed and evolved in parallel by an organization or by different organizations. The major concerns of the system are:

- **The personnel concern:** Basic information about related people such as professors, registrars, students, etc. should be handled. The information involves basic personnel information such as name, age, sex, etc.

- **The enrolment concern:** A student must be enrolled in a campus of the university for a semester. The system should manage basic information such as student identification number and the grade point average of each student. A student can register courses provided by the university system. A professor is responsible for teaching certain courses.

- **The payroll concern:** This concern is about managing salary and tax information of people employed by the university system such as professor, registrar, etc. with two policies; full-time or part-time. Each employee is paid by a campus of the university system and pays tax according to the tax regulation of the province the campus belongs to.

We can consider other minor concerns such as resource allocation which manages information such as office allocation for professors and computing resource allocation for students but we do not mention them in this article. We can also find that a major concern can have its own sub-concerns such as concerns about payroll and tax in the payroll concern.

Each major concern has its own business rules. For example, a student cannot take courses provided by campuses in which the student is not enrolled in the enrolment concern. These business rules may be changed occasionally and the system should be adapted to reflect those changes.

Three major concerns should be composed to construct the integrated information processing system for the university system whether they are developed in parallel by an organization or by different organizations. Even if they are developed by an organization, they should be composed at the final stage of development. In this article, we assume the parallel development of each concern by different organizations. During the final composition, some business rules may be added to the integrated system. For example, if a student is advised by a professor, the professor should be paid by the campus in which the student enrolled. This business rule needs information from the enrolment and the payroll concern. The merger of independently developed concerns like the case of our example is a typical usage of AOP as shown in [3].
3.2. Relation abstraction as pointcut for aspect

In this subsection, we will present essential properties for representing structural relationship between components in a composition. Relational properties of a composition are constructed as a relation. Relational properties are the analogies to pointcuts, into which behavior like advice of AOP is attached in a black box manner. In a relation, every participating component has its own responsibility. A role is taken as a part of a composition which a component is supposed to realize. In our relation model, we treat a role as a typeless structural slot that will be populated by a certain component later in order to hide the type and the detailed implementation of its responsibility. The rationales of typeless roles are that the extrinsic responsibility of a role could be taken as a part of a composition and the intrinsic responsibility of a role is not our concern if it is properly used and encapsulated with the extra entities in the composition. There are one or more roles in a composition. Each role has its multiplicity. The multiplicity of a role represents the possible number of the occurrences of a component instance as the role with respect to other component instances playing the other roles in a composition. We list the properties representing the relationship among components of a composition as follows and call them relational properties.

- A set of roles that components are supposed to play
- Multiplicities of roles
- Constraint over components

The constraint over components is necessary when defining the invariant condition which component instances should satisfy to participate in a composition. The above relational properties are sufficient to represent the relationship of components in a composition as a relation. Our relation model, however, has an additional relational property to represent the relationship between compositions which are used to compose a larger composition.

- Derivation rule between relations

The derivation rule between relations is almost the same as in the derivation of logics and a derived relation becomes another relation.

Relations for major concerns of the university information system are illustrated in Fig. 1. Sub-concerns of major concerns are identified by dotted ellipses. In Fig. 1, multiplicities of roles and some relations are omitted and relations can be taken as logics. Some relations are derived from other relations with derivation rules such as Personnel(campus, p), where p means a person and campus means a campus. Derivation rule means that there are compositions of small concerns to compose larger concerns. Personnel concern in Fig. 1 should be derived from Registrar, Professor, or Student concern as defined in derivation rules. We can also see that Enrolment concern has two sub-concerns; Enroll(campus, s) and RegisterCourse(s, course), where s means a student. RegisterCourse concern is composed of Enroll...
concern and OpenCourse concern. The derivation rule for RegisterCourse concern represents the business rule in Enrolment concern. Similarly, Payroll concern can be explained. In this way, small concerns defined as relations without derivation rule can be derived into larger concerns defined as relations with derivation rule in our relation model. Compared with object-oriented programming, relations without derivation rule are based on association between classes and are the smallest crosscutting concerns over classes participating in association. The rationale for the smallest crosscutting concern based on association is that class is started to possess extrinsic properties when it is associated with other classes. From a relational properties based on simple associations, we can merge new concerns over the same set of relational properties and compose larger concerns by deriving new relational properties from other sets of relational properties.

All relation properties for a relation representing a crosscutting concern like those illustrated in Fig. 1 are accessible from the outside of the composition. For example, relational properties of Personnel, Registrar, Professor, and Student relations are accessible from the outside of the personnel concern and they can be changed or merged. Relational properties of the composition are hooks where the evolution of the composition and the introduction of a new crosscutting concern take place. Compared to pointcut expressions in AspectJ, which points out the inside of classes, relational properties of our relation model express different kind of points of program, the composed structures of compositions.

On the other side of composition evolution, the relational properties of a composition can evolve in such a case that the set of components changes to adapt a technical evolution or the multiplicities of roles may change to adapt the change of the business requirement. The changes of relational properties can be categorized
as follows:

- Replacement of set of components for roles
- Change of multiplicity of role
- Change of constraint over components
- Change or addition of derivation rule

The replacement of the type for a role does not mean the change of the functionality of a composition. It is desirable that the functionality of a composition keeps unchanged regardless of the change of its relational properties. Since roles in our relational properties does not have type. We can change the set of types of roles as long as the name and number of roles would not change. In other words, the set of base classes can change more flexibly than pointcuts in AOP.

### 3.3. Relation elements for aspect and RJava

We refine the relational abstractions representing the relational properties as elements expressed in extensions of simple logic predicate. The elements for our relation model are listed as follows:

- Relation interface
- Relation class
- Relation query

The relation interface and relation class represents abstract and concrete compositions. They are called relations in our relation model. A relation interface proclaims a concern cutting across components without disclosing the types of participating components and the detailed implementation of the composition for the concern. Thus, the information hiding and the genericity of the relation interface is provided in a similar way to interfaces of object-oriented language such as Java. Relation interface, however, provides more flexibility than interface because a relation interface reveals its constituent components. When using the relation model for software development, relations with derivation rule should become relation interfaces during the development of software.

A relation class composes the detailed implementation of a crosscutting concern with a set of constituents which can include simple classes and other compositions. Relation class can use the functionalities of its constituents in a black box manner. A set of relation classes can implement a set of relation interfaces. When using our relation model, a relation class can be derived from other relations to merge concerns represented by them or the set of relation classes implementing a relation interface can be changed to adapt to the evolution of the concern represented by the relation interface. The relation query supports accessing aspect at the instance level in a declarative way instead of the references in conventional object-oriented paradigm.
Composition of Aspects Based on a Relation Model

Figure 2 shows the relationships between these relation elements. Relations in Fig. 2 represent the payroll concern of our university software. A relation interface, \( \text{IPayroll} \) is implemented by a set of relation classes, \( \text{FullTimePayroll} \) and \( \text{PartTimePayroll} \) and retrieved by relation queries. The implementation relationships are explicitly expressed by the edges from relation classes to relation interfaces. In the case of \( \text{Tax} \) relation, \( \text{ITax} \) is implemented by an intermediate relation class, \( \text{PayrollTax} \) relation because \( \text{PayrollTax} \) is a conjugation of two relations; \( \text{IPayroll} \) and \( \text{Province} \).

The relation elements can be used in modeling and design methodologies or as language constructs for an enhancement of an object-oriented language. Though the relation interface and relation class can be expressed as simple stereotypes of UML, we currently use a language which is enhanced from Java with the relation elements and name it RJava (Relation Java). The most important reason we choose the language enhancement for the first prototype of our relation model is that a design of crosscutting concern based on our relation model can be directly transformed into its implementation and vice versa. The code fragments listed hereafter are written in RJava. The syntax for the relation elements in RJava is listed in Appendix A.
3.4. Aspect based on relation model

The relation interface is a unit representing a crosscutting concern for a set of components and hides the detailed implementation of its behavior, types of components, and constraints over components. On the other hand, the relation class has all the details including composition structures of relations based on relationships of components.

The grammatical function of predicates as carriers of action has proponents in linguistics [7]. We have advanced this idea by extending our relations to have abstract or concrete behavior and become encapsulation entities representing collaborative crosscutting concerns.

A relation interface representing Payroll sub-concern between a campus and an employee, and its abstract behavior is expressed in RJava as follows:

```java
RInterface IPayroll(campus:1, employee:*)
{
    abstract float getPayroll();
    ...
}
```

IPayroll has two roles, campus and employee. The multiplicity of campus is one and that of employer is many. This means that a campus can pay to multiple employees but an employee can be paid by only one campus.

Relation classes are classified into basic relation class and derived relation class. Basic relation class has no derivation rule. Thus, a basic relation class is a concrete implementation of a simple crosscutting concern over a set of classes. The code of a basic relation class, FullTimePayroll implementing IPayroll is listed as follows:

```java
RClass FullTimePayroll(Campus campus: 1, Person employee: *)
    implements IPayroll(campus, employee)
    when employee.getAge() > 18
{
    float salary;
    float getPayroll() { return salary; }
    void changeSalary(float salary) { this.salary=salary; }
    ...
    FullTimePayroll(float salary) // a constructor
    { this.salary=salary; }
}
```

A relation class is composed of the roles with types and its concrete behavior. A relation class can also have its own attributes such as salary shown in FullTimePayroll, which are extrinsic properties for the concern. A role in a relation class plays the role with the same name of the relation interfaces it implements. An instance of a relation class is instantiated through the call of a constructor with first objects as its roles and subsequent values, which are used for the extra initialization in its argument list. The invariant constraint which the constituents of relation class should satisfy is expressed in when clause and is optional. If it is not satisfied, the
instance is not created. The instantiations of different relation classes implementing the $IPayroll$ relation interface can be coded as follows:

$$\text{new FullTimePayroll(kpuDaejeon, professorRoberts, 90000.0)}$$
$$\text{new PartTimePayroll(kpuSeoul, john, 40.0)}$$
$$\text{new PhysicianPayroll(kpuSeoul, drKim, 300.0)}$$

The last two instantiations suppose that the payroll concern is extended to allow a campus to employ students as part-timers or physicians in hospitals with policies other than $FullTimePayroll$ or $PartTimePayroll$. The first two arguments of instantiations are $campus$ and $employee$ roles and the third argument of each instantiation is used to initialize data related to their payrolls.

The typelessness of the role in the relation interface implies the composition based on our relation model can be made of arbitrary components regardless of their types. For example, types of $employee$ roles may be different in these instantiations; $professorRoberts$ is of $Professor$ type, $john$ is of $Student$, and $drKim$ is of $Physician$.

In this way, we can evolve a crosscutting concern by replacing the set of components without affecting relational properties of the relation interface.

An instance of a relation class is treated as a tuple of objects playing roles, which becomes members of the tuple spaces of the relation class and the relation interfaces implemented by it after the creation of the instance. The replacements of component instances playing roles are not allowed. Instead, an instance of a relation must be created and deleted in whole with proper operations; $new$ operation for creation and $not$ operation for deletion. A deleted instance of a relation is removed from tuple spaces of the relation and the relation interfaces it implements. In this way, the instance level consistency of relations is not broken.

A relation class composed from other relations is called a derived relation class. In other words, the relation class with derivation rule is a derived relation class. The derivation rule of a derived relation class defines how it is composed from other relations at the instance level. The relationship between roles in the derivation rule is expressed with the name correspondence. The $PayrollTax$ relation class from the payroll concern is a derived relation class. It represents concern about tax incurred by payroll from the university and implements $ITax$ relation interface. The code of $PayrollTax$ relation class is listed as follows:

$$\text{RClass PayrollTax(campus: 1, employee: *) implements ITax(campus, employee) :-IPayroll(campus,employee)payroll, Province(campus,authority)pr // derivation rule}$$

$$\{ \text{float getPayroll() equates payroll.getPayroll();}$$
$$\text{float getPayrollTax() \{}$$
$$\text{return pr.getTaxRate() * getPayroll();}$$
$$\}$$
$$\text{float getTax() // to implement ITax}$$
$$\{ \text{return getPayrollTax();} \}$$

...
Roles of a derived relation class cannot have types if they have same names as roles of relations in the derivation rule of the derived relation class. Otherwise, a role should have a type because it is a new role declared in the derivation relation class. In the case of PayrollTax, two roles, campus and employee cannot have types because campus and employee are roles of IPayroll. In a derivation rule, there can be a conjurator to conjugate its relations. The campus role is the conjurator that joins two relations, IPayroll and Province. The conjugation of relations means that an instance of a derived relation can be instantiated only if the conjurator role simultaneously participates in the instances of the relations in the derivation rule. If there is no conjurator in the derivation rule, any arbitrary combination of constituent relation instances can be derived into a new instance of the derived relation class. Identifiers, payroll and pr, respectively following IPayroll and Province relations, are references to constituent relation instances and their behaviors are used in implementing the behavior of PayrollTax as shown in getPayroll and getPayrollTax operations. When two operations in the derived relation and one of its constituent relations are identical, equates clause is used as a syntactic sugar easing the implementation of the operation of the derived relation class.

A derived relation class makes it possible to merge one or more existing concerns into a new concern and represents the external and hierarchical composition of concerns based on their causal relationship between concerns. In other words, a derived relation means that an aspect instance cutting across a set of component instances requires other aspect instances cutting across different sets of component instances which might include some of the former component instance set. In this way, our relation model support hierarchical and external composition of aspects.

A derivation rule can have a set of relation interfaces such as IPayroll in PayrollTax. Therefore, it is possible that a derived relation class is composed from its derivation rule including relation interfaces it implements directly or indirectly. This means that instances of a derived relation class can be implicitly created because instances of a relation interface in its derivation rule can be implicitly created if the relation interface is implemented by the derived relation class directly or indirectly. This recursive composition of relation is natural because our relation model is based on logics. In terms of aspect, the implicit instantiation of a recursively derived relation means that the creation of an instance of an aspect implies the creation of another instance of the same aspect. In this case, the explicit constructors of the derived relation classes are useless because they are never called. To supplement this problem of the instantiation of derived relation class, a derived relation class can have an initializer for the initial setting of its data which is invoked when they are implicitly instantiated. RInitializer (Recursive Initializer) in Appendix A is the syntax for the initializer of derived relation class and is a constructor without parameter list.

A relation interface can extend other relation interfaces. Whenever an instance of a relation instance is inserted into its tuple space, the instance is inserted into tuple spaces of relation interfaces it extends. A relation class can be changed as long as the
consequent change still satisfies the functionalities of interfaces it implements. These situations imply there are subtype relationships between relations. The subtyping rule between relations will be discussed in the next subsection.

A relation instance can be traced by keeping references to it but keeping these references are intricate and an obstacle to the evolvability of software composition. For example, it is very difficult to change the navigational direction of objects in conventional object-oriented programming. Another problem is the change of multiplicity. If the multiplicity of \textit{campus} changes from one to many in \textit{IPayroll}, it affects the clients of the relation. The relation query is devised to remedy this loss of evolvability incurred by handling references and changing multiplicities of roles. The relation query is not a major element in our relation model but complements our model when it is used as language constructs and nurture evolovability. In RJava, the relation query is the first order logic predicate returning a relation instance or a set of relation instances as a result of a retrieval of a tuple space using objects as keys. The number of relation instances found by a relation query is zero, one or plural according to its search result. If every key in the search condition is definite (specified as an object), only one relation instance may be found; call it singular relation query. If some of keys are indefinite (specified as "."), an ordered set of relation instances can be found; call it plural relation query. If there is no instance of a relation satisfying the search condition, the answer is \textit{null}, which means none or an empty set. For example, we can calculate total payroll of the \textit{kpuSeoul} campus as follows:

```java
double totalPayroll=0.0;
IPayroll seoulPayroll[] = IPayroll(kpuSeoul, \_); // query
if (seoulPayroll!=null)
    for(int i; seoulPayroll [i]!=null; i++)
        totalPayroll+= seoulPayroll[i].getPayroll();
...
```

The query in the example, \textit{IPayroll(kpuSeoul, \_)}, has an indefinite key in the position of \textit{employee}. Thus, the program fragment using relation query is coded to handle an ordered set of \textit{IPayroll} relation interface. With relation queries, we can selectively access the result of a crosscutting concern at the instance level and the code for references to relation instances is evolved without invasive modification.

### 3.5. Subtyping rule of relations

When determining the type of an object in object-oriented programming, the match of operations and their signatures are a necessary condition. But the subtyping rule of our relation model is different from that of object-oriented programming because relations have roles and their multiplicities in addition to its operations and their signatures. If the type of a relation is determined by only operations and their signatures, the relation query over the relation interface may not work correctly.
Therefore, we need a careful definition of the subtyping rule between relations in terms of the number of roles and their multiplicities.

Consequently, a relation needs to at least satisfy the following rules in order to become a subtype of another relation.

- The operations and their signatures of two relations must satisfy the subtyping rule of object-oriented programming.
- If a relation has a role which does not correspond to any role of its supertype relation interface, the multiplicity of the role must be one.

With these rules, we can safely subtype a relation interface to the other relation interfaces and also declare a relation class to implement a relation interface because the pluralities of the results of relation queries for the subtype relation are consistent with those for its supertype relation interface.

Another concern about the relation query is the change of the multiplicities of roles. The changes of the multiplicities of roles do not affect the result of the relation query syntactically because the pluralities of relation queries are not affected by the changes of multiplicities. But the change of multiplicities can affect the clients semantically. The multiplicity of campus in IPayroll is one. Thus, a client of IPayroll expects the ordered set with single relation instance as the result of IPayroll(john). If the client handles only the first relation instance of the result, it makes the client to work incorrectly if the multiplicity of campus changes from one to many.

3.6. Composition and evolution of aspects based on relation model

In this subsection, we show how our relation model can elegantly and effectively merge aspects which developed in parallel and evolve some aspects with the university information processing system. The university information system introduced in Sec. 3.1 has three major concerns; personnel, enrolment, and payroll concerns, which are developed by different organizations. For simplicity, we assume that they are given the same base classes such as Person, Campus, Course, etc. These given base classes have no more than intrinsic properties. For example, Person has intrinsic properties such as name, age, sex, address, marriage status, etc. and has no information about extrinsic properties such as reference to the department it belongs, courses it takes, payroll, etc. Each concern is developed as relations sketched in Fig. 1 with our relation model and integrated at the final stage of development.

At the final integration stage, there are a number of alternatives on how to compose relations from three concerns. The simplest is integrating them without any more composition or evolution of relations. In other words, the integrated information system provides no more services provided by relations in three major concerns and have no more business rules found in three major concerns.

It is, however, more realistic that more services and more business rules are necessary for the integrated system. When an additional service or a business rule is required, relations from different concerns are composed into a derived relation class. Figure 3 shows some relations are added to the integrated university system.
through derivations of relations of major concerns. In Fig. 3, three new relations are added to represent three additional concerns and one additional business rule as follows:

- **StudentEmploy**: A student enrolled into a campus as a full-time student can be employed by the campus. The student is paid by the campus according to the part-time policy. This relation class implements `IPayroll`.
- **ConfidentialPersonnel**: `IPersonnel` has services showing general information. The information about the payrolls of employees is confidential. This relation class provides services for these confidential and general informations.
- **PartTimeLecturer**: The original personnel concern did not consider part-time lecturers who come from the outside of the university. The personnel information about the part-time lecturer is handled by this relation class derived from `TeachCourse` relation because the lecturer should teach at least a course. This relation class should implement `IPayroll` because lecturers are paid due to teaching courses.
- **Advise**: If a student is advised by a professor, the professor should be paid by the campus in which the student enrolled. This is a new added business rule during integration.
The behaviors of composed relation classes are composed of the relations they are derived from. For example, the code for StudentEmploy relation is listed as follows:

\[
\text{RClass StudentEmploy(Campus campus: 1, Person student: *) implements IPayroll} \\
\text{::= FullTimeEnroll(campus, student) enroll, PartTimePayroll(campus, student) pay} \\
\text{}} \\
\text{float getPayroll() equates pay.getPayroll()} \\
\text{...}
\]

Advise is implemented as a relation class, which can be coded as follows:

\[
\text{RClass Advise(Person adviser: *, Person student: *)} \\
\text{when Professor(campus, adviser), IPayroll(campus, adviser), IEnroll(campus, student)} \\
\text{}} \\
\text{float regularMeetingTime() {...}} \\
\text{...}
\]

The invariant condition of Advise means an existential condition of relation instances satisfying the business rule. There are tens of compositions of relations representing new concerns and new business rules when we integrate major concerns but mentioning them is beyond the scope of this article.

Some relations in major concerns should be evolved after integration because new business rules are applied. For example, some campus may decide to give the additional salary for the married full-time employee. The FullTimePayroll relation class does not reflect this new salary policy. We can evolve IPayroll concern to adapt this change by adding a new relation class, MarriedPayroll. Its code can be listed as follows:

\[
\text{RClass MarriedPayroll(campus: 1, employee: *) implements IPayroll(campus, employee)} \\
\text{::= FullTimePayroll(campus, employee) pay, IPersonnel(campus, employee) personnel} \\
\text{}} \\
\text{float addition; float getSalary() { } \\
\text{if(personnel.getMarriage()) return pay.getSalary()+addition; else return pay.getSalary();} \\
\text{...} \\
\text{MarriedPayroll(float addition) // Constructor} \\
\text{}} \\
\text{...}
\]

In the above code, the IPersonnel relation class is used to check marriage status of employees. We can use this new relation classes to adapt the integrated university system to this new concern at the instance level. If kpu at Seoul takes this new salary policy, the necessary code fragment is listed as follows:

\[
\text{... not IPayroll(kpuSeoul, __);} \\
\text{new MarriedPayroll(kpuSeoul, __, 100.0);} \\
\text{...}
\]
With the \textit{not} operation in the code, all relation instances of \texttt{IPayroll} related to \texttt{kpuSeoul} are removed from tuple spaces of \texttt{IPayroll} but relation instances at \texttt{Full-TimePayroll} remain at its tuple space. The \textit{new} operation inserts all instances of \texttt{MarriedPayroll} into tuple spaces of \texttt{IPayroll}. In this way, the relation class representing a new business rule can sneak into existing relation interfaces at the instance level.

The composition like the above merger of relation models is possible at the class level with AOP techniques such as AspectJ and Hyper/J. In Hyper/J, two concerns can be defined as two hyperslices and they are composed through generic rules of a hyperspace. The solution using AspectJ for the composition of equivalent concerns like our university information system is awkward because the emphasis of AspectJ is placed on the additions of relatively minor aspects to a software composition based on a main aspect. If we compose two concerns by taking a concern as the main concern and taking the others as the minor concerns, the resulting composition is too structure sensitive to the main concern, which may be taken as a minor concern in an alternative design. Compared to compositions with Hyper/J, an advantage of the aspect composition based on our relation model is that the existing clients using the constituent relation models such as \texttt{IPayroll} and \texttt{IPersonnel} are not affected by their composition and persistent objects produced by the existing concerns can be used in the new merged application without the losses of their identities. This loss of identity hampers the reuses of persistent object in Hyper/J. On the contrary, the role and role model always make an object to maintain its identity. In many researches, the role model, however, is not properly abstracted and encapsulated, and the composition of role models is complicated as mentioned in [8]. Compared to black box composition, our relation model provides relational properties into which a new relation can be added or relations can be replaced more flexibly. In the example of \texttt{MarriedPayroll}, we cannot evolve a black box representing \texttt{FullTimePayroll} if the black box does not provide interfaces enabling composers to get objects playing the \textit{campus} and \textit{employee} roles.

3.7. \textit{Example of recursively derived relations}

One characteristic of the logic predicate is the recursive derivation of logics. Since our relation model is based on the logic predicate, it can provide recursive derivation of relations at the instance level. This characteristic of relation model makes it possible to design and develop various concerns based on recursive associations. This is one superiority of our relation model to current AOP, which does not even support external composition of aspects.

The example introduced in this subsection is a simple relation model representing a recursive derived association, ancestor association. The parent association between \texttt{Person} is declared as follows:

\begin{verbatim}
IParent(Person kid, Person parent) // relation interface
Father(Person kid, Person father) implements IParent(kid, father).
Mother(Person kid, Person mother) implements IParent(kid, mother).
\end{verbatim}
The relation model representing ancestor association is defined as follows:

\[
\text{IAncestor(descendant, ancestor).} \quad // \text{relation interface}
\]

\[
\text{Parent(Person kid, Person parent) implements IAncestor(kid, parent)}
\]

\[
:- \text{IParent(kid, parent)} \ p.
\]

\[
\text{Ancestor(Person kid, Person ancestor) implements IAncestor(kid, ancestor)}
\]

\[
:- \text{Parent(kid, descendant)} \ p, \text{IAncestor(descendant, ancestor)} \ a.
\]

Most instances of \text{IAncestor} are created implicitly through its recursive derivation rules of relation classes implementing it. A snapshot resulted from the execution of an application is depicted in Fig. 4. We can see that instances of IAncestor are composed of instances of IAncestor and IParents recursively. In the above code, the behavior is omitted. The detailed behavior can be added to the above relations to develop nontrivial concerns.

4. Discussion

Our relation model provides crosscutting concern over component instances, which can be dynamically handled at the instance level. In the relation model, the strength of black box composition, the guarantee of encapsulation is achieved because our model uses only black box classes and relations for the composition of crosscutting concerns and do not reveal the details of their implementation. Our relation model achieves more flexibility for aspect composition and evolution than black box composition because of the relational properties. Compared to AOP, aspect composition and evolution using our relation model has tradeoffs. Relational properties of our relation model cannot support aspect composition using the details of class structures which is possible in some AOP but the whole aspect composition is not affected by the change of base classes. The derivation rule in our relation model represents the hierarchical and external composition of aspects which is not supported in AOPs. The advantage of AOP is achieved through the definition of relation classes and relation interfaces crosscutting roles. Additionally, the crosscutting concerns over roles are handled dynamically at the instance level to the same degree as role and role model. Compared to role and role model, our relations provide gross structural
Table 2. Comparison of composition techniques.

<table>
<thead>
<tr>
<th></th>
<th>Black box composition</th>
<th>AOP</th>
<th>Role model</th>
<th>Relation model</th>
</tr>
</thead>
<tbody>
<tr>
<td>Aspect proclamation</td>
<td>Unrelated</td>
<td>Yes</td>
<td>Occasional</td>
<td>Yes</td>
</tr>
<tr>
<td>Dynamic composition</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>at the instance level</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Information hiding</td>
<td>Yes</td>
<td>Affected by</td>
<td>Occasional</td>
<td>Yes</td>
</tr>
<tr>
<td>of aspects (but not</td>
<td></td>
<td>class structures</td>
<td></td>
<td></td>
</tr>
<tr>
<td>aspects)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Hierarchical and</td>
<td>Yes</td>
<td>Weak</td>
<td>Weak</td>
<td>Yes</td>
</tr>
<tr>
<td>external composition</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Evolution of the inside</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>More flexible</td>
</tr>
<tr>
<td>of a composition</td>
<td>(through aspect</td>
<td>(through role</td>
<td></td>
<td>than black box</td>
</tr>
<tr>
<td></td>
<td>heirarchy)</td>
<td>hierarchy)</td>
<td></td>
<td>composition)</td>
</tr>
</tbody>
</table>

abstraction. Table 2 compares the black box composition, AOP, role model, and our relation model with respect to the characteristics of composition techniques.

The language constructs of our experimental RJava is not yet sufficient to express generic pointcut and advice in AspectJ terminology. We think that it is possible to add lexical expressions defining pointcuts in RJava. Programs written in RJava are translated into those in plain Java. The most essential technique implementing RJava is translating relation interfaces and classes into the hash maps of its instances for their tuple spaces, which are indexed by the identities of their roles. One of the limitations of RJava is that the type of a role cannot be an interface of Java. This is because interfaces of Java cannot be type-casted which is necessary in the implementation of the derived relation class. Operations expressed in `equates` clause are translated in simple forward calls. The overhead in performance is linear to object-oriented programming because a relation class doubles references to its roles and object traversals necessary to answer a relation query are anyway needed in object-oriented programming as well. One of the improvements we desire is the adoption of mechanism like the aggregation in COM in order to reduce the number of forwarding calls caused by `equates` clauses.

5. Related Works

Aspects in the relation model can be applied to only external properties of components because component classes and relations in a crosscutting concern are treated as simple black boxes and the necessary extra properties are assembled with them through the compositions of relation classes. However, this is a natural approach, as long as the intrinsic properties of a component cannot be changed without breaking encapsulation. The standpoint relying on the definition of the external protocol which components should satisfy to be composed into a composite can be found in the researches concerned with component frameworks. A variant of state chart is used to declare the external protocols which components called actor should follow.
in realtime object-oriented modeling (ROOM) [19] and the finite state automata are used in [20]. Tuple space similar to that of our relation model is used in Linda [21]. Linda has three simple operations over the tuple space: add, match and read, and match and remove. Compared to our relation model, Linda does not support hierarchical composition of our relation model. Linda has only one tuple space causing a central bottleneck but our relation model has a number of self-containing tuple spaces representing relation classes and interfaces.

In [7], Steinman classifies role concepts of various role based systems into three categories: roles as named places of a relationship, roles as a form of generalization/specialization, and roles as separate instances joined to an object. The most similar one to our relation model is the roles as named places of a relationship. In object-role model (ORM), an association equivalent to a predicate involves a number of roles corresponding to the places of a predicate and the roles are populated by objects. Object types and association types are derived from the roles. Associations can be nested or derived for information modeling [22, 23].

Pun and Kahn’s work in [24] extracts logical perspectives of an object-oriented program to logical objects and the relationships between objects are defined as simple logics in logical objects. Logical objects can be found through simple logics or derivation rules consisting of logics. However, their work is just an integration of object-oriented programming and logic programming and the relationship between objects is determined at the time of the definition of a logical class. In their work, the composition of aspects is not a concern at all.

6. Conclusion

It is said in [3] that present software is like clay; it is soft and malleable early in its lifetime, but eventually it hardens and becomes brittle. The hardness of software composition which increases as the development phase progresses is caused by contradictory information hiding. The more we hide, the freer we are from implementation details but the less we can reuse. The relation model we introduced in this paper is a result of the efforts to find the proper degree of information hiding of black box composition in order to make it possible to handle crosscutting concerns. The relation model supports black box composition in such a flexible way that crosscutting concerns can be evolved or introduced and with such instance-level dynamicity as in role based systems. Aspectual collaboration in [4] has the most similar consequence to our relation model in that it combines MP and AOP to achieve their synergy. Our relation model, however, supports the selective application of an aspect at the instance level and support role and role model without the necessity of separately handling role-binding anomalies mentioned in [8].

One of the future research directions of the relation model is the integration with de facto standard object-oriented modeling, UML and the design of software analysis and design tools based on the relation model. The other interesting research
is the self adaptable software composition based on the relation model. It is possible when the relation model is integrated with the learning system based on logics.

Appendix A. Added Syntax for Rjava

RelationInterface ::= ‘RInterface’ RelationInterfaceName ‘(InterfaceRoleList)’
    [‘extends’ SuperInterfaceList ] InterfaceBody
InterfaceRoleList ::= RoleName ‘:’ (Integer|‘*’) [‘,’ RoleName ‘:’ (Integer|‘*’) ]*
SuperInterfaceList ::= RelationInterfaceName ‘(’ AbstractRoleList ‘)’
    [‘,’ RelationInterfaceName ‘(’ AbstractRoleList ‘)’ ]*
AbstractRoleList ::= RoleName ‘,’ RoleName)*
RelationClass ::= ‘RClass’ RelationClassName ‘(ConcreteRoleList)’
    [‘implements’ SuperInterfaceList ] [‘:-’ DerivationRule ] [‘when’ ConditionalExpression ]
    [RInitializer ] ClassBody
ConcreteRoleList ::= ClassName RoleName [‘:’ (Integer | ‘*’ )]
    [‘,’ ClassName RoleName [‘:’ (Integer | ‘*’ ) ]]*
DerivationRule ::= RelationName ‘(’ AbstractRoleList ‘)’ RelationInstanceName
    [‘,’ RelationName ‘(’ AbstractRoleList ‘)’ RelationInstanceName ]*
RInitializer ::= ‘RInitializer’ ‘{’ statements ‘}’
RelationName ::= (RelationClassName | RelationInterfaceName)
RelationClassAllocationExpression ::= ‘new’ RelationClassName
    ‘(’ RelationParameterList ‘)’
RelationParameterList ::= RelationParameter (, RelationParameter )
RelationParameter ::= ( Expression | ‘_’ )

References


