IIETA International Information and Engineering Technology Association
Advancing the World of Information and Engineering

Ingénierie des Systèmes d'Information
Vol. 24, No. 3, June, 2019, pp. 321-329
Journal homepage: http://iieta.org/journals/isi

Collaborative Component Interaction

Saeid Masoumi, Ali Mahjur*

Faculty of Electrical & Computer Engineering, Malek Ashtar University of Technology, Tehran 1774-15875, Iran

Corresponding Author Email: mahjur@mut.ac.ir

https://doi.org/10.18280/isi.240312

Received: 18 January 2019
Accepted: 5 April 2019

Keywords:
programming language, reusability, collaboration, event, SOP

ABSTRACT

There are many collaboration-based languages, in which a collaboration has multiple roles inside and a collection of roles from different collaborations forms an object layout. Most of them use a form of single inheritance to build collaboration. This means every role in a sub collaboration inherits from a role in the super collaboration. In such a model, the way of interacting roles affects the reusability of roles and consequently collaborations.
To address this problem, this paper presents components that interact with each other in a collaborative manner. In our model, components as collaborations, instead of being inherited, are composed with each other. Moreover, the interaction of components is based on events, which are soft dependencies and do not affect the reusability of roles and components.

1. INTRODUCTION

There are two ways to specify an object. In the first way, the layout of the object (class) is specified. This is the approach used by the object oriented paradigm (Like C# and Java). In the second way, the collaborations that the object is involved in are specified. In this way, compiler gathers the roles of the object which are assigned by the collaborations to form its layout. This is the approach of the collaboration-based languages. Figure 1 illustrates both approaches in a simple form.

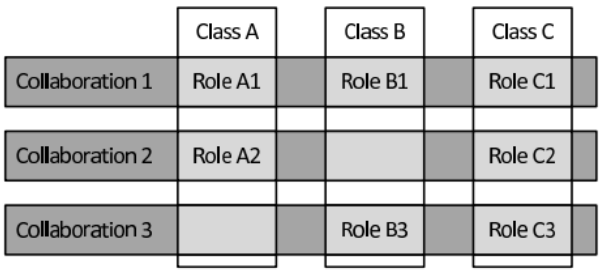

Figure 1. Class vs Collaboration

It is well known that the collaboration-based approach is superior to the class-based approach [1-6]. It improves reusability and supports separation of concerns. Contracts [1], Role-based designs with C++ templates [5, 7, 8], Mixin layers [9, 10], Set Oriented Programming (SOP) [11], ObjectTeam/Java [12], and J& [4] are examples of collaboration-based design.

In most role based paradigms, a collaboration may inherit from another collaboration. In such paradigms, roles in a derived collaboration inherit from the corresponding roles of the parent collaboration. Therefore, they do not offer independent roles.

For example, in Figure 1, Collaboration2 inherits from Collaboration1 and consequently A2 and C2 roles inherit from A1 and C1, respectively.

Having dependent roles makes it nearly impossible to crosscut a role by other roles in the same object. Role crosscutting happens, when a role injects some codes into another role of different collaboration. Currently, the only way of crosscutting roles is method calling, which is a hard dependency and lowers independency and reusability of collaborations.

For example, supposing that both C1 and C2 roles have m method, calling m method of C2 is hard-coded in m method of C3. This forces Collaboration2 (or any of its parents) to implement m method. Otherwise, compiler throws an error.

Set Oriented Programming (SOP) is one of the recent collaborative paradigms. It offers independent collaborations and roles- that is, a role in a collaboration does not depend on any other roles of other collaborations. Against inheritance based role paradigms, roles of an object in different collaborations may have different names.

Instead of inheriting collaborations, SOP composes them. This means, there are composite collaborations that are composed of other collaborations. Inside the composite collaboration, the roles (of similar objects) from composed collaborations are gathered and placed in a set, which is a new SOP concept for keeping the object layout by which various objects are created.

```
CollabComposer {
  A[];
  A{
    // Constructors
    // Fields
    // Methods
  }
  Collaboration1[A];
  Collaboration2[A];
  … m1(…){
    A obj=new A();
  }
}
```

```
Collaboration1[A1]{
  A1{
    // Constructors
    // Fields
    // Methods
  }
}
Collaboration2[A2]{
  A2{
    // Constructors
    // Fields
    // Methods
  }
}
```

Figure 2. An example of collaboraion composition in SOP

For example, in Figure 2, CollabComposer is a composite collaboration, which is composed of two other collaborations (Collaboration1 and Collaboration2) and has a role named A. A is a set for collecting A1 and A2 roles of an object. Also, CollabComposer itself can add a role to A. The name of this role is the same as set name. Therefore, there are three roles for this object, A1 from Collaboration1, A2 from Collaboration2 and a role from CollabComposer.

Similar to inheritance based role paradigms, in SOP, method calling is used to crosscut roles. Inside an object, the roles of the composer can crosscut other roles in the composed collaborations and vice versa. However, roles of composed collaborations cannot crosscut each other. For example, in Figure 2, A1 and A2 roles in A set cannot crosscut each other.

To overcome this problem, this paper introduces an event-based role crosscutting mechanism. Crosscutting roles is done by raising events, instead of calling methods. The outline of the proposed method is that events are defined and raised in the roles of composed collaborations. According to circumstances, in the composing collaboration, they are delegated to methods of other roles. In fact, the composer will decide which role event of a composed collaboration is delegated to which role method of a different composed collaboration. This way, the roles of an object from different collaborations can crosscut each other.

In our previous work [13] we applied events on object features and made OOP class interactive by a new composition method. It has led us to bring events into the collaboration-based design since events did not affect the reusability of classes.

The remainder of this paper is organized as follows. In the next section, we present a brief overview of SOP. Section 3 proposes our event-based role crosscutting mechanism. In section 4, a formal representation of our model is introduced. Section 5 discusses some related works and section 6 conclude the paper.

2. SET ORIENTED PROGRAMMING

The SOP paradigm is based on two abstractions: component and set. A component is a stateless compile time entity which represents a collaboration. It defines the roles of the objects involved in a collaboration. The objects having the same role in a component (collaboration) form a set.

In SOP, an application is a collection of components. One of the components is the main component that defines the complete behavior of the application. The others should be instantiated in order to be used.

A typical SOP program is shown in Figure 3. It is made of four components: *main*, *C1*, *C2* and *C3*. main represents the whole program. It is made of three components: two instances of *C1* and one instance of *C2*. Moreover, *C1* itself is made of *C2* and *C3*.

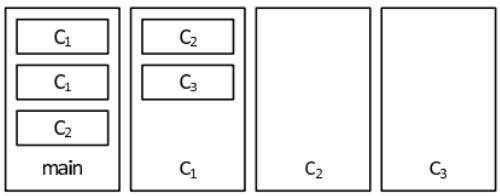

Figure 3. A class is the union of the roles defined by collaborations that the class is involved in

To define the behavior of a component, SOP has a collection of declarations. They are divided into two groups: the first group is the usual attribute declarations: attribute, method, constructor. The second group provides the reusability mechanisms of SOP: component instantiation.

2.1 Component and set

Let us begin the presentation of SOP by an example. The example is a doubly linked list. A doubly linked list has two roles: root and node. The root role is the role of the linked list itself and the node role is the role of the objects that can be nodes of the linked list. The collection of the objects that can have the root role form the Root set and the collection of the objects that can have the node role form the Node set.

In SOP, a linked list is represented by a component: LinkedList (Figure 4). It has two interface sets: Root and Node. The body of a component has the definitions of its internal sets and declarations. In this example, the LinkedList component has a constructor and two methods. The constructor and methods belong to the Root set. The methods are: insert and remove. insert inserts a node into the linked list and remove removes a node from it.

A set is the collection of objects having the same role in a component. Attributes, methods and constructors of a set are enclosed in a block identified by the set name. An interface set is a set that the behavior of its members is partially defined by its component. An internal set is a set that the behavior of its members is completely defined by its component. Objects can be created only from internal sets.

```
LinkedList[Root, Node] {
  …
  Root{
    Root(){
      …
    }
    void insert(Node n){
      …
    }
    void remove(Node n){
      …
    }
  }
}
```

Figure 4. LinkedList component

```
Family[Marriage, Husband, Wife, Child]{
  …
  LinkedList[Marriage, Child];
  Marriage{
    int date;
    void marry(int d, Husband h, Wife w){
      …
    }
    void addChild(Child c){
      insert(c);
      …
    }
  }
}
```

Figure 5. Family component uses a LinkedList insance to store the children of a marriage

As another example, consider a university. A university has various roles such as student, professor, department and course. The student and professor roles should be assigned to persons. Clearly a person is independent from university and university only assigns some roles such as professor or student to it. Therefore, student and professor are interface sets of the University component. However, department and course are only meaningful in a university and the university completely defines the behavior of them. Therefore, they are internal sets

of the University component. University has the following declaration.

```
University[This, Professor, Student] {
  Department[];
  Course[];
}
```

As components are stateless, this set is used to store the specific data of the University component. Of course, This is not a keyword and any other identifier can be used instead of it. In the LinkedList example, Root was used for this purpose.

2.2 Instantiation

When a component needs the service of another component, it declares an instance of that component. For example, consider a component that stores family information. A family is formed by the marriage of a man and a woman. In SOP, a family is implemented as a component: Family component (Figure 5). It has four interface sets: Marriage, Husband, Wife and Child. As a marriage may have more than one child, an instance of LinkedList is used to store them. The Root and Node sets of LinkedList are assigned to the Marriage and Child sets of Family respectively.

The code has two methods for the Marriage set: marry and addChild. marry gets a date, a husband and a wife and marries them. addChild gets a child and adds him to the linked list of children. To do so, it calls the insert method of the Root set of the LinkedList component.

A role assignment defined by an instance declaration may be named or unnamed. The LinkedList instance of the Family component defines the following role assignments.

```
Root → Marriage
Node → Child
```

In the declaration of the LinkedList instance, they are unnamed. However, it is possible to assign a name to each of them. Therefore, the following declarations are possible too.

```
LinkedList[Marriage children, Child];
LinkedList[Marriage, Child node];
LinkedList[Marriage children, Child node];
```

When a role is named, it resembles an attribute declaration. In the first sample, the Root role of the LinkedList instance is named children. In this case, the properties of the Root role of the LinkedList instance are accessible under the children name. Therefore, to add a child to a marriage, the following statement is used.

children.insert (child);

Every instance of a component is distinct from other instances of the same component. It means that if two instances of a component assign the same role to a set, two copies of the role are added to the members of the set.

To clarify, assume that one needs to store the children of a marriage in two ways. In the first way, they are sorted using their name and in the second way, they are sorted using their age. It is done using two instances of LinkedList.

```
LinkedList[Marriage ageSorted, Child];
LinkedList[Marriage nameSorted, Child];
```

In the above example, Marriage has two copies of the Root role of LinkedList. The first one is named ageSorted and the second one is named nameSorted. In addition, Child objects have two copies of the Node role. Both of them are unnamed. So, they are not accessible directly. Of course, they are not required to be directly accessible too. In fact, the LinkedList operations are defined on its Root set and the Node roles are accessible through them.

2.3 A complete application

In SOP, an application itself is a component called the main component. The name of the main component is arbitrary. It is the only component that cannot be instantiated. The main component has just one interface set whose name is the same as the component name. Unlike other components, the interface set of the main component is not mentioned explicitly. The interface set of the main component has a method named main. The execution of the application begins from this method and finishes when it finishes.

```
People {
  Person[];
  Person {
    int ID;
    Person (int _ID) {ID = _ID;}
    void print (){...}
  }
  LinkedList[People, Person];
  int main (int argc, char[][] argv) {
    People people;
    Person p;
    int i;

    people = new People ();
    for (i = 0; i < 5; ++i) {
      p = new Person (i);
      people.insert (p);
    }
    for (p = people.head; p; p = p.next)
      p.print ();
  }
}
```

Figure 6. A complete SOP application

As an example, Figure 6 shows an application named People. The application has an internal set named Person. It has a LinkedList of Person objects. The main method of the application creates 5 Person objects and inserts them into the linked list. Then, it iterates over the linked list and prints the Person objects.

2.4 Dynamic dispatch

Dynamic dispatch is an important capability of the object oriented paradigm. Programming languages have different approaches to specify methods which are dispatched dynamically. In SOP, when a component needs to call a method dynamically, it marks the method as extern. When an instance of such a component is created, the container component can provide a new implementation for the extern method.

```
Tree[Root, Node]{
  Node{
    extern int compare(Node n);
  }
  Root{
    void insert(Node n){
      Node ptr;
      …
      if (ptr.compare(n)<=0)
      {…}
    }
  }
}
```

Figure 7. Tree component has one extern method

```
People{
  Person[];
  Person{
    int compare(Person p){
      …
    }
  }
  Tree[People, Person];
}
```

Figure 8. People uses an instance of Tree to store Person objects

As an example, parts of the Tree component are shown in Figure 7. As the code snippet shows, its Node set has an extern method (compare). In Figure 8, Tree component is used to store persons in the People component. Therefore, it provides an implementation of compare method in Person set.

3. CROSSCUTTING ROLES

This section proposes a new way of crosscutting roles in SOP. As stated in section 1, roles of composed collaborations cannot crosscut each other. This means, a role cannot inject some codes to other roles and vice versa. To overcome this problem and make interoperability of roles, we use *event* as interaction mechanism.

We believe that every role method reaches some specific states from the beginning to the end point of its code. The number of states a method has is limited to the size of the method (the type of work it does). Reaching a method to a state, in a role composition, must be notified to other roles by raising an event.

On designing a role, programmer has to define the events of methods. The definition of an event begins with the *event* keyword. Similar to a method definition, an event has a return type that can be any data type (*void* is valid too). If the return type is not *void*, it must have a default value. Essentially, an event does not have any body at all. The following code snippet shows the general form of event definition and the way of raising it.

```
CollaborationName {
  SetName{
    event returnType eventName(paramType paramName)=defaultValue;
    … Method(…){
      …
      eventName(values);
      …
    }
  }
}
```

An event as a part of a role code refers to a state of a role method. Therefore, it should have a meaningful name since it is important for crosscutting roles and also helps to the understandability of the role code. For instance, BeforeAdd and AfterAdd are mostly reached events (states) in an element addition method (e.g. enqu, push, insert, etc.) of some data structures (like Queue, Stack, Tree, etc.).

Event raisings are not limited to the before and after points of methods. In fact, an event can be raised at any desired point of code. For example, in Figure 9, Stack at the beginning state of adding an element raises an event name evBeforeAdd and after successful adding of an element raises evAfterAdd. Also, two other events (i.e. evBeforeRemove and evAfterRemove) are defined and raised in pop method. These events are adequate for the most of compositions in which Stack participates.

An event can be taken either by a role or not at all by any role. In one hand, by accepting an event, the receiving role executes a method (in response to the event) and returns a result if needed. The result depends on the event definition. On the other hand, when an event is raised and not taken by any role, its default value is replaced in the raise locations.

In our model, a collaboration should not be aware of the future collaborations it will be composed with. It is a task of the composer to compose collaborations and roles and delegate role events to appropriate methods (like wiring of hardware components). For example, Stack has no information about who will catch evAfterAdd event (Counter or Log or any other role). The important thing is that push method has reached a state named evAfterAdd. This means, it successfully added an element to the stack and this stage is the best place for another role to crosscut Stack and do an action.

```
Stack [Root, Node] {
  Root {
    event bool evBeforeAdd()=true;
    event void evAfterAdd();
    event bool evBeforeRemove()=true;
    event void evAfterRemove();
    void push(Node n) {
      if(evBeforeAdd()) {
        ...
        evAfterAdd();
      }
    }
    Node pop() {
      if(evBeforeRemove()) {
        ...
        evAfterRemove();
        return ...
      }
    }
  }
}
```

Figure 9. Stack collaboration in SOP

The fate of events will be determined at object instantiation inside the composer. This means, when collaborations are composed and objects are instantiated, it becomes clear that which event of a role is delegated to which method of another role in a set. For example, Figure 10 composes Stack with Counter and just delegates two events of Stack to the Counter methods.

An event may have some arguments depending on the state it reflexes. When a role method raises an event, it gives state values to the event arguments. For example, in Figure 11, inc method of Counter informs other roles that it is going to increase the counter by sending its value over raising of evBeforeAdd.

```
StackCounter{
  Root[]; Node[];
  Stack[Root, Node] {
    Root{
      evAfterAdd=Counter.inc;
      evAfterRemove=Counter.dec;
    }
  };
  Counter[Root];

  void main(){
    Root r=new Root();
    Node p=new Node();
    r.push(p);
  }
}
```

Figure 10. Composing stack with counter

```
Counter[Root] {
  Root{
    event bool evBeforeAdd(int c)=true;
    event void evAfterAdd();
    event bool evBeforeRemove()=true;
    event void evAfterRemove();
    int count;
    List(){count=0;}
    int size(){return count;}
    void inc(){
      if (evBeforeAdd(cnt)){
        cnt++;
        evAfterAdd();
      }
    }
    void dec(){
      if (evBeforeRemove()){
        cnt--;
        evAfterRemove();
      }
    }
  }
}
```

Figure 11. Interactive counter

Figure 13 is a composite collaboration, which limits Counter by composing it with Limit and delegating its evBeforeAdd event to the check method of Limit.

Our model allows programmers to freely define and raise events at any point of code. Also, there is no restriction for the arguments of events. Although our type system automatically checks for any mismatch, it is a duty of programmer to check the signature of events and methods before any delegation.

```
Limit[Root] {
  Root{
    int size;
    Limit(int _size){
      size=_size;
    }
    bool check(int v){
      return v<size;
    }
  }
}
```

Figure 12. Limit collaboration

```
LimitedCounter[Root]{
  Root{
    Root(int max):Counter.Root(),Limit.Root(max){
    }
  }
  Counter[Root]{
    Root{
      evBeforeAdd=Limit.check;
    }
  };
  Limit[Root];
}
```

Figure 13. Composing counter with limit

It is obvious that hard-coding the role interactions inside the role definition makes it un-reusable. But, our events as interactions are soft dependencies. This means, when a collaboration is instantiated alone (not participated in a composition), its events become neutral operations, and wherever they are raised their default value is replaced. As a result, not only does our mechanism lets roles crosscut each other, but it also keeps reusability.

4. FORMAL REPRESENTATION

4.1 Grammar

A modified grammar of SOP is provided in Figure 14. It shows that a program (*Prog*) in SOP is a collection of component declarations. A component (*Comp*) is made of three collections: a collection of interface sets (*If*), a collection of internal sets (*In*) and a collection of declarations (*Decl*).

The union of the interface and internal sets of a component forms its sets (*Set*). The scope of a set is the component instance containing it. *Base* denotes the primitive types of the language. The union of the primitive types and set types of a component forms the types of the component (*Type*).

SOP has five categories of declarations: field declaration (*Field*), method declaration (*Meth*), event declaration (*Evt*), constructor declaration (*Cons*), component instance declaration (*Inst*).

A field declaration declares a field for set *Set* whose type is a primitive type or an internal set. Likewise, a method declaration defines a method for set *Set*. The type of the arguments and return value of a method can be any type including interface sets. An event declaration is the same as method declaration, just it does not have any implementation and needs a default value if its return type is non-void. Finally, a constructor declaration declares a constructor for *Set*.

$$
\begin{array}{rcl}
Prog & ::= & \overline{Comp} \\
Comp & ::= & \overline{If}, \overline{In}, \overline{Decl} \\
If & ::= & id \\
In & ::= & id \\
Set & ::= & If \mid In \\
Base & ::= & integer \\
Type & ::= & Base \mid Set \\
Decl & ::= & Field \mid Meth \mid Evt \mid Cons \mid Inst \\
Field & ::= & Set\{Base\ l\} \mid Set\{In\ l\} \\
Meth & ::= & Set\{Type\ id\ (\overline{Type\ id})\ t\} \\
Evt & ::= & T\ id\ (\overline{T\ id}) = l \\
Cons & ::= & Set\ (\overline{Type\ id})\ t \\
Inst & ::= & Comp'[\overline{M'}] \mid Comp'[\overline{M'}]\ Delg \\
M' & ::= & If' \rightarrow Set \mid If' \rightarrow Set\ l \\
Delg & ::= & \overline{id = id} \\
l & ::= & id \\
t & ::= & this \mid Type\ l \mid new\ In \mid t.l \mid t.f(\bar{t}) \mid t.e(\bar{t}) \mid t := t
\end{array}
$$

Figure 14. Modified SOP grammar

A component instance declaration declares an instance of a component ($Comp'$). It defines a mapping (M') from the interface sets of the instantiated component (If') into the sets of the container component. A mapping can be named or unnamed. Also, the fate of events in the instantiated components is determined here by delegations. A delegation (*Delg*) links an event from an instantiated component to a method from another instantiated component.

In the formal definition of the language only seven terms are defined: *this* (self object), variable declaration, object creation, field selection, method call, event raising and assignment. Some explanations for the terms are followed. *this* returns a pointer to the role of the object containing the method. Objects can be created only from internal sets and primitive types.

4.2 Semantics rules

In SOP, every declaration adds a role to an object. Some of them add a primitive role while others add a collection of roles. Therefore, an object is a tree of roles.

Figure 15 has the rules that form the layout of an object. They correspond to the declaration rules of the grammar. *L−FIELDB* adds a primitive value to a set. *L−FIELDI* adds a role to a set.

As it was mentioned, instance declaration defines a mapping from the interface sets of a component ($Comp'$) to the sets of the instantiating component (*Comp*). *L−INSTU* and *L−INSTN* add the role of an interface set of $Comp'$ to the corresponding set of *Comp*. They differ in whether the role is named or not. Note that each instance declaration adds its own role.

$$\frac{Set\{Base\ l\}}{< l = v_0 > \in Set.layout}\ [L-FIELDB]$$

$$\frac{Set\{In\ l\}}{< l = In.layout > \in Set.layout}\ [L-FIELDI]$$

$$\frac{If' \mapsto Set}{< If'.layout > \in Set.layout}\ [L-INSTU]$$

Figure 15. Layout construction rules

$$\frac{t_1 \to t_1'}{t_1.l \to t_1'.l}\ [E-SEL]$$

$$\frac{r_1 = \{l = b\}}{\hat{r_1}.l \to b}\ [E-FIELDB]$$

$$\frac{r_1 = \{l = r_2\}}{\hat{r_1}.l \to \hat{r_2}}\ [E-FIELDI, E-INSTN]$$

$$\frac{r_1 = \{l = \hat{r_2}\}}{\hat{r_1}.l \to cast(comp^t, \hat{r_2})}\ [E-ASSOC]$$

$$\frac{t_0 \to t_0'}{t_0.f(\bar{t}) \to t_0'.f(\bar{t})}\ [E-METH-THIS]$$

$$\frac{t_i \to t_i'}{\hat{r_0}.f(\bar{v}, t_i, \bar{t}) \to \hat{r_0}.f(\bar{v}, t_i', \bar{t})}\ [E-METH-ARG]$$

$$\frac{[this \to \hat{r}_0, \bar{x} \to cast(comp^d, \bar{v})]t_1 \to v_1}{\hat{r}_0.f(\bar{v}) \to cast(comp^t, v_1)}\ [E-METH]$$

$$\frac{t_i \to t_i'}{e(\bar{v}, t_i, \bar{t}) \to e(\bar{v}, t_i', \bar{t})}\ [E-EVENT-ARG]$$

$$\frac{\begin{array}{c}Set \in comp^t, Set_1 \in comp^{d1}, Set_2 \in comp^{d2}, Set_1 \mapsto Set, Set_2 \mapsto Set,\\ Set_1\{e(\bar{x}) = v_0\}, Set_2\{f(\bar{x})\ t_1\}, Set\{e = f\},\\ e(\bar{v}) \in Set_1\end{array}}{\begin{array}{c}cast(comp^{d2}, cast(comp^t, v)) \to v'\\ e(\bar{v}) \to f(\bar{v}) \to cast(comp^{d1}, cast(comp^t, v_1')) \to v_1\end{array}}\ [E-EVENT-DELG]$$

$$\frac{\begin{array}{c}Set \in comp^t, Set_1 \in comp^{d1}, Set_1 \mapsto Set,\\ Set_1\{e(\bar{x}) = v_0\}, Set\{e = \emptyset\},\\ e(\bar{v}) \in Set_1\end{array}}{e(\bar{v}) \to v_0}\ [E-EVENT-DVAL]$$

$$\frac{t_0 \to t_0'}{t_0 := t_1 \to t_0' := t_1}\ [E-ASSIGN1]$$

$$\frac{t_1 \to t_1'}{l := t_1 \to l := t_1'}\ [E-ASSIGN2]$$

$$\frac{l := v}{l = cast(comp^d, v)}\ [E-ASSIGN3]$$

Figure 16. Semantics rules

In the evaluation rules, sometime it is necessary to change the role of the object. For this reason, function cast (Figure 17) is provided. It gets a role of an object and returns the role of the object which is added by the given instantiation. *C−BASE* states that a primitive value does not depend on the component instance. *C−SELF* states that if the given role corresponds to the given instance, return the role itself. *C−INSTU* goes down or up from a role added by an unnamed role assignment. *C−INSTN* does the same for a named role assignment.

A term is evaluated in the context of the component instance containing it ($comp^t$). $comp^d$ is the component instance that has the declaration that the rule is for it.

The result of the evaluation of a term (*v*) is a primitive value (*b*) or a pointer to a role ($\hat{r}$). When the result of a term is a role two cases happen. Some terms get a role of an object and return another role of it. Some others select a different object. In such cases, the role of the object that corresponds to $comp^t$ is returned.

In the grammar, six terms are defined for the language. The first three ones are trivial. Therefore, no evaluation rules are provided for them. The evaluation rules for other terms (field selection, method call, event raising, assignment) are provided in Figure 16.

There are four rules to evaluate a field selection. The first one (*E−SEL*) is the congruence rule. The others are computation rules. They correspond to layout rules. If an evaluation rule corresponds to more than one layout rule it is assigned two names. *E−FIELDI* (*E−INSTN*) selects a role which is added by *L−FIELDI* or *L−INSTN*. It returns a pointer to the role. Note that as a new object may be selected, the role of the object which is returned is the role added by $comp^t$. Finally, *E− INSTU* returns an unnamed role which is added by *L−INSTU*.

There are three rules to evaluate a method call. The called method belongs to $comp^d$. *E −METH−THIS* evaluates the object that the method is called on. *E−METH–ARG* evaluates the arguments of the method. *E−METH* does the actual call. It assigns the roles that correspond to $comp^d$ to the arguments. Finally, the role of the object which is returned is the role added by $comp^t$.

$$\frac{}{cast(Comp, b) = b}\ [C-BASE]$$

$$\frac{}{cast(comp, \hat{r}) = \hat{r}}\ [C-SELF]$$

$$\frac{cast(Comp', \hat{r}) = \hat{r'}, r'' =< If'' \mapsto S' >}{\begin{array}{c}cast(Comp'', \hat{r}) = \hat{r''}\\ cast(Comp, \hat{r''}) = \hat{r}\end{array}}\ [C-INSTU]$$

$$\frac{cast(Comp', \hat{r}) = \hat{r'}, r'' =< If'' \mapsto S'\ l >}{\begin{array}{c}cast(Comp'', \hat{r}) = \hat{r''}\\ cast(Comp, \hat{r''}) = \hat{r}\end{array}}\ [C-INSTN]$$

Figure 17. Function cast

There are three rules to evaluate an event raising. E−EVENT–ARG evaluates the arguments of the event. The raised event belongs to $comp^{d1}$ and its delegated method belongs to $comp^{d2}$. In E−EVENT- DELG, $comp^{d1}$ and $comp^{d2}$ are instances of $comp^t$. Set_1 and Set_2 are respectively sets of $comp^{d1}$ and $comp^{d2}$ and mapped to *Set* in $comp^t$. Set_1 defines event *e* and *Set* delegates it to method *f*. Therefore, raising the event *e* causes calling the method *f*. The return value (v_1') must be cast since the event and the method are in different components. Finally, E−EVENT-DVAL evaluates the event to its default value.

The last group of evaluation rules evaluates an assignment. *E−ASSIGN1* and *E−ASSIGN2* evaluate the left and right sides of the assignment and *E−ASSIGN3* does the assignment. Assuming that the location *l* belongs to $comp^d$, the role of the value which corresponds to $comp^d$ is assigned to *l*.

4.3 Typing relation

The type of an expression is evaluated according to the component that contains it ($comp^t$).

Before discussing typing rules, it is necessary to introduce a function: *rtype*. *rtype* gets a set and the component instance containing it and returns a set of $comp^t$ that corresponds to it.

Figure 18 shows this function. The first rule states that every set of $comp^t$ is mapped onto itself. The second rule is the closure rule. It states that if an interface set (If''_1) of $comp''$ is mapped onto S'_1 and *rtype* of S'_1 is S_1 then *rtype* of If''_1 is S_1 too.

$$\frac{S_1 \in comp^t}{rtype(comp^t, S_1) = S_1}$$

$$\frac{If''_1 \mapsto S'_1, rtype(comp', S'_1) = S_1}{rtype(comp'', If''_1) = S_1}$$

Figure 18. Function rtype

$$\frac{t_1 : S_1,\ S_1\{T\ l\}}{t_1.l : T}[T-FIELDB, T-FIELDI]$$

$$\frac{t_1 : S_1,\ F'_1 \mapsto S_1}{t_1 : F'_1}[T-INSTU]$$

$$\frac{t_1 : S_1,\ F'_1 \mapsto S_1\ l}{t_1.l : F'_1}[T-INSTN]$$

$$\frac{t_1 : S_1, S_1\{T_1 f(\overline{T_2})\}, \overline{t_2} : \overline{T'_2}, rtype(\overline{T_2}) = rtype(\overline{T'_2})}{t_1.f(\overline{t_2}) : rtype(T_1)}[T-METH]$$

$$\frac{S_1\{T_1\ e(\overline{T_2}) = l\}, l: T_1, \overline{t_2}: \overline{T_2}}{e(\overline{t_2}): T_1}[T-EVENT]$$

$$\frac{S_1\{T_1\ e(\overline{T_2})\}, S_2\{T'_1\ f(\overline{T'_2})\}, S\{e = f\}, S_1 \mapsto S, S_2 \mapsto S,\ rtype(T1) = rtype(T'_1), rtype(\overline{T_2}) = rtype(\overline{T'_2})}{e(\overline{t_2}): rtype(\overline{T'_1})}[T-DELG]$$

$$\frac{t_1 : S_1, t_2 : S'_1, rtype(S_1) = rtype(S'_1)}{t_1 := t_2 : S_1}[T-ASSIGN]$$

Figure 19. Typing relation

Having *rtype*, Figure 19 shows the typing rules. Like semantics rules, typing rules are provided only for four terms: field selection, method call, event raising and assignment. There are three typing rules for field selection: *T −FIELDB* (*T−FIELDI*), *T−INSTU*, *T−INSTN*. They are almost trivial.

The next rule, *T−METH*, is the typing rule for method call. It has three parts. The first part specifies that the method should belong to the current role of the object. The second part specifies that the argument and the actual argument should have the same *rtype*. Finally, it specifies that the type of the return value is *rtype* of the return type of the method.

The typing rule of event raising, *T-EVENT*, specifies that in an event definition the type of default value and return type must have the same. Also, it shows that event arguments and the actual argument values should have the same *rtype*.

For type checking of a delegation the rule *T-DELG* is used. It shows that when an event is delegated to a method their return type and arguments should have the same *rtype*.

The last rule, *T−ASSIGN*, specifies two things: the conditions that should be true for assignment and the result type of assignment. Again, the *rtypes* of the left and the right sides of the assignment should be the same in order to do an assignment. The result type of the assignment is the type of the left side of the assignment.

4.4 Soundness

When a component is instantiated a mapping is defined from the interface sets of the instantiated component (*comp′*) into the sets of the instantiating component (*comp*).

$$If' \mapsto S$$

Using the type mapping function, it is possible to define a function that gets a component instance (*comp′*) and a set of it (*if ′*) and returns a set (*S*) of the component instance (*comp*) containing it. It is called extended type mapping (*ETM*).

$$ETM(comp', If') \mapsto S$$

It is clear that ETM is actually a function.

Having ETM, it is possible to show that *rtype* is actually a function that gets a type space and a set and returns a set of $comp^t$.

Now, it is time to prove a soundness theorem for *rtype* and cast functions.

Theorem. If r: S and rtype(S) = rtype(S′) then cast(comp′, r) is defined. Moreover, cast(Comp′, r) : S′.

An important conclusion of the theorem is as follows.

Corollary. If r : S and rtype(S) is defined then cast($comp^t$, r) is defined too.

Theorem Progress. A term t is either a value or there is t′ such that t → t′.

Theorem Preservation. If t: T and t → t′ then one of the following cases are true.

- t′: T.
- t′: T′ where $T' \mapsto T$.
- *t′: T′* where *T′* is a superset of *T*.

5. RELATED WORK

Due to the benefits of collaboration-based languages, a dozen of them were emerged in the past twenty years [1-6, 10, 12, 14-18]. They differ in the definition of collaboration abstraction, composition model and role attributes.

Some of the collaboration-based languages [5] do not have an abstraction to denote a collaboration. In such languages, the roles forming a collaboration are defined independently. In other words, in such languages the base abstraction is role.

However, most of them have an abstraction to represent a collaboration. Some of them use a state full abstraction [14, 16] for this purpose, while others use a stateless one [19].

The next feature is the way that a collaboration is defined. A few languages [15] use a traditional OOP language for this

purpose. Therefore, it is not possible that a collaboration uses another collaboration. However, in most languages a collaboration can be made using other collaborations. Often they use single inheritance [16] [3, 14] for this purpose. In such languages, when a collaboration inherits from another one every role in the sub collaboration inherits from a role in the super collaboration which has the same name (name matching).

Of course, single inheritance is not the only approach used for this purpose. J& [4] uses intersection types which is a form of multiple-inheritance. Object Team/Java [12] uses *playedby* clauses to add a role to a class. In Mixin layers [10], the super collaboration of a collaboration is specified when it is instantiated.

The next feature is the scope of the collaboration roles. In almost all collaboration-based paradigms roles are global. Therefore, it is possible to create an object whose type is a specific role everywhere in a program. An exception is Object Team/Java [12] where an object with a specific role can be created only within the container collaboration. However, the role is still accessible everywhere in the program.

6. CONCLUSION

This paper addresses the problem of crosscutting roles in collaboration-based design. Despite traditional OOP approaches, we use SOP (a new component-based programming). To crosscut object roles placed in different collaborations, events are used. In fact, an event is raised when a role method wants to inform its state to other roles. Against method calling, an event raising is a soft dependency. This means, when a collaboration does not want to participate in a composition, its events will be neutralized by compiler without changing its code. This way, events do not affect the reusability of roles and collaborations. As a result, our mechanism provides role interactivity while maintains reusability.

Future research should consider the potential effects of events more carefully in the design patterns problems. Real-world examples should be taken into account in order to examine the advantages, and disadvantages of the approach.

In the next work, we will discuss design pattern problems and compare our approach with the current solutions in terms of understandability, flexibility, and reusability.

Furthermore, role, event, and delegation are also needed to be visualized. Roles can be modeled by extending the UML class diagram notation and delegations during component composition can be modeled by extending the UML sequence diagram notation. Raise points also need to be marked in both or one of those diagram types.

REFERENCES

[1] Helm, R., Holland, I.M., Gangopadhyay, D. (1990). Contracts: Specifying behavioral compositions in object-oriented systems. SIGPLAN Not., 25(10): 169-180. https://doi.org/10.1145/97945.97967

[2] Mezini, M., Lieberherr, K. (1998). Adaptive plug-and-play components for evolutionary software development. SIGPLAN Not., 33(10): 97-116. https://doi.org/10.1145/286942.286950

[3] Nystrom, N., Chong, S., Myers, A.C. (2004). Scalable extensibility via nested inheritance. SIGPLAN Not., 39(10): 99-115. https://doi.org/10.1145/1035292.1028986

[4] Nystrom, N., Qi, X., Myers, A.C. (2006). J&: nested intersection for scalable software composition. SIGPLAN Not., 41(10): 21-36. https://doi.org/10.1145/1167473.1167476

[5] VanHilst, M., Notkin, D. (1996). Using role components in implement collaboration-based designs. SIGPLAN Not., 31(10): 359-369. 10.1145/236338.236375

[6] Ostermann, K. (2002). Dynamically composable collaborations with delegation layers. in Proceedings of the 16th European Conference on Object-Oriented Programming. Springer-Verlag, pp. 89-110. https://doi.org/10.1007/3-540-47993-7_4

[7] VanHilst, M., Notkin, D. (1996). Decoupling change from design. in Proceedings of the 4th ACM SIGSOFT symposium on Foundations of software engineering. ACM: San Francisco, California, USA. pp. 58-69. https://doi.org/10.1145/239098.239109

[8] VanHilst, M., Notkin, D. (1996). Using C++ templates to implement role-based designs. in Object Technologies for Advanced Software: Second JSSST International Symposium, ISOTAS '96 Kanazawa, Japan, March 11–15, 1996 Proceedings, K. Futatsugi and S. Matsuoka, Editors. Springer Berlin Heidelberg: Berlin, Heidelberg. pp. 22-37. https://doi.org/10.1007/3-540-60954-7_41

[9] Smaragdakis, Y., Batory, D. (1998). Implementing layered designs with mixin layers. in ECOOP'98 — Object-Oriented Programming: 12th European Conference Brussels, Belgium, July 20–24, 1998 Proceedings, E. Jul, Editor. Springer Berlin Heidelberg: Berlin, Heidelberg, pp. 550-570. https://doi.org/10.1007/BFb0054107

[10] Smaragdakis, Y., Batory, D. (2002). Mixin layers: an object-oriented implementation technique for refinements and collaboration-based designs. ACM Trans. Softw. Eng. Methodol., 11(2): 215-255.

[11] Mahjur, A. (2019). Set Oriented Programming. Submitted to IEEE Transaction on Software Engineering.

[12] Herrmann, S. (2003). Object teams: Improving modularity for crosscutting collaborations. in Revised Papers from the International Conference NetObjectDays on Objects, Components, Architectures, Services, and Applications for a Networked World. Springer-Verlag, pp. 248-264. https://doi.org/10.1007/3-540-36557-5_19

[13] Masoumi, S., Mahjur, A. (2019). Reusable and interactive classes: A new way of object composition. Turkish Journal of Electrical Engineering & Computer Sciences.

[14] Madsen, O.L., Moller-Pedersen, B. (1989). Virtual classes: A powerful mechanism in object-oriented programming. SIGPLAN Not., 24(10): 397-406. https://doi.org/10.1145/74877.74919

[15] Harrison, W., Ossher, H. (1993). Subject-oriented programming: a critique of pure objects. SIGPLAN Not., 28(10): 411-428. 10.1145/165854.165932

[16] Ernst, E. (2001). Family polymorphism. in Proceedings of the 15th European Conference on Object-Oriented Programming. Springer-Verlag, pp. 303-326. https://doi.org/10.1007/3-540-45337-7_17

[17] Baldoni, M., Boella, G., van der Torre, L. (2006). Roles as a Coordination Construct: Introducing powerJava. Electronic Notes in Theoretical Computer Science,

150(1): 9-29. https://doi.org/10.1016/j.entcs.2005.12.021
[18] Leuthäuser, M., Assmann, U. (2015). Enabling view-based Programming with SCROLL: Using roles and dynamic dispatch for etablishing view-based programming, in Proceedings of the 2015 Joint MORSE/VAO Workshop on Model-Driven Robot Software Engineering and View-based Software-Engineering. ACM: L'Aquila, Italy, pp. 25-33. https://doi.org/10.1145/2802059.2802062
[19] Jolly, P., Drossopoulou, S., Anderson, C., Ostermann, K. (2004). Simple Dependent Types: Concord. in ECOOP Workshop on Formal Techniques for Java Programs, ser. FTfJP.
[20] Bracha, G., Cook, W. (1990). Mixin-based inheritance. in Proceedings of the European conference on object-oriented programming on Object-oriented programming systems, languages, and applications. ACM: Ottawa, Canada, pp. 303-311. https://doi.org/10.1145/97946.97982
[21] Batory, D., Höfner, P., Kim, J. (2011). Feature interactions, products, and composition. in Proceedings of the 10th ACM international conference on Generative programming and component engineering. ACM: Portland, Oregon, USA. pp. 13-22. https://doi.org/10.1145/2189751.2047867