Area Ingeniería de Software.
Departamento de Informática y Estadística
FaEA. Universidad Nacional del Comahue
Buenos Aires 1.400. (8300) Neuquén. Argentina

Martinez Carod N., Flores A., Reynoso L.

A Formal Specification of a Pattern based on a Recursive Structure: the Composite Pattern

---

10 Diciembre 1999

UNC/AIS Reporte Técnico No. 1

Area Ingeniería de Software.
Departamento de Informática y Estadística
FaEA. Universidad Nacional del Comahue
Buenos Aires 1.400. (8300) Neuquén. Argentina

A Formal Specification of a Pattern based on a Recursive Structure: the Composite Pattern

Abstract

Design Patterns

1. Introduction

Because in the design environment is usual that recurrent situations can be found, it would be of great benefices to specify those situations in an adequate and proved way. Generally a designer is able to discover a pattern in the set of situations.

Object-oriented design patterns are a good way to identify and abstract the essential aspects of commonly recurring design structures, and thus provide a basis for reusable object-oriented design. However, patterns are invariably described informally which makes it difficult to give any meaningful certification of software developed using them.

The basis of this technical report was presented in [6], which introduces the concepts of GoF object-oriented design pattern and formalises their essential elements, as a way for checking the internal consistency of patterns structures.

We extend the preliminary model of the technical report already mentioned in order to formally describe a pattern based on a recursive structure and we show how this can be done using the Composite pattern. Section 2 present the essential elements of the Composite pattern. Section 3 introduces several concepts related to this pattern. Section 4 describes the formal specification of the composite pattern, and some modifications over the preliminary model.

2. Pattern Composite

The following example is not complete description as presented in the GoF catalogue. We include the essential components of this pattern: intent, structure, participants and collaborations, which will be formalised [1].

Intent

Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.

Structure

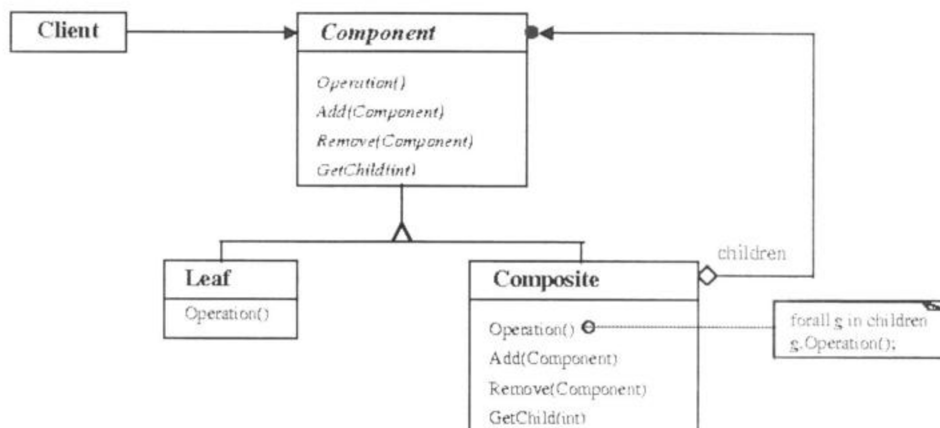

Figure 2.1

The following diagram (figure 2.2) shows a typical composite object structure of recursively composed Component objects:

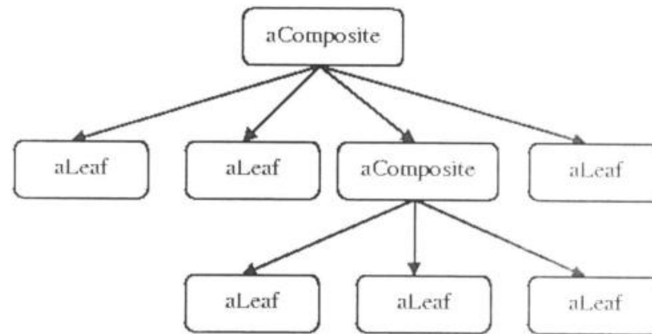

Figure 2.2

The following explain how a composite object structure can be built:

The composite pattern describes how to use recursive composition so those clients do not have to make this distinction.

The key to the Composite Pattern is an abstract class that represents both primitives and their containers. The abstract class declares operations that represent the main task to objects belonging to it. It also declares operations that all composite objects share, such as operations for accessing and managing its children.

The Composite class defines an aggregate of Component objects. Composite implements Operation to call Operation on its children, and it implements child-related operations accordingly. Because the Composite interface conforms to the Component interface, Composite objects can compose other Composite recursively.

Participants

- **Component**
  - declares the interface for objects in the composition,
  - implements default behaviour for the interface common to all classes, as appropriate

- **Leaf**
  - represents leaf objects in the composition. A leaf has no children
  - defines behaviour for primitive objects in the composition.

- **Composite**
  - defines behaviour for components having children
  - stores child components.
  - Implements child-related operations in the Component interface.

- **Client**
  - manipulates objects in the composition through the Component interface.

Collaborations

Clients uses the component class interface to interact with objects in the composite structure. If a recipient is a leaf, then the request is handled directly. If the recipient is a composite, then it usually forwards requests to its child components, possibly performing additional operations before and/or after forwarding.

3. Concepts used in composite pattern structure.

3.1 Inheritance

The Inheritance is an abstraction mechanism that is supported by the object-oriented programming languages. Inheritance is a relationship between one class which define a behaviour and data structure and other class or classes which are subclasses of it and present the same behaviour and/or data structure, but these subclasses can extend this behaviour and/or data structures [2].

Inheritance allows conceiving a new class of objects as refinement of another, to abstract out the similarities between classes and to design and specify only the differences for the new class. Thereby, the inheritance can be used as a reuse technique and to provide a powerful way to produce code that can be reused over and over.

In the composite pattern the inheritance relationship is used to allow treat uniformly both primitive objects as well as composite objects. In the component class there are operations, which perform the objective task in any component, and child management operations, that allow this class will be able to include objects in its state and in this way building an object hierarchy. These operations should be common to every subclasses of the component class, but that is not true because the leaf class inherit child management operations, which must be not allowed for a class without children as the leaf class. The real objective of the inheritance is present a common interface for subclasses, to allows treat both primitive objects and composite objects uniformly.

3.2. Recursive Structure.

There are several GoF's patterns based on recursive structures. One of the goals of this report is describe and formalise properties and constraints of the composite pattern, which is known as a pattern based on a recursive structure. This pattern belongs to a group of patterns, which present similar characteristics. The group of patterns based on recursive structures was identified in [3] as a way of classifies the GoF's design pattern.

As a way to understand the concept of recursive structure, we present a definition of it, which was extract of [5].

Definition:

> "A structure recursive can be defined by writing the name of the type being defined actually inside its own definition; or in the case of mutually recursive definition in the def*inition of some preceding type*."

UNIVERSIDAD NACIONAL DEL COMAHUE FOLIO N° 169 F. A. E.

3.2. Recursive Aggregation.

A common relationship in the structure of some patterns based on a recursive structure is a recursive aggregation.

An aggregation relationship is a special form of association describing a "part-whole" or "a part of" relationships. Aggregation implies that an aggregate object and its owner have identical lifetimes.

Rambaugh *et al.* [4] describe three kind of aggregation: fixed, variable and recursive. Each of them, allow to create a fixed aggregate structure (one level tree structure), shown in figure 3.a, a finite number of levels (two or more levels), figure 3.b., or unlimited levels, shown in figure 3.c.

Figure 3.a. Fixed aggregate.

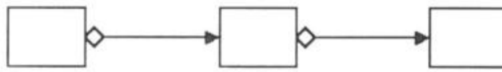

Figure 3.b. Variable aggregate.

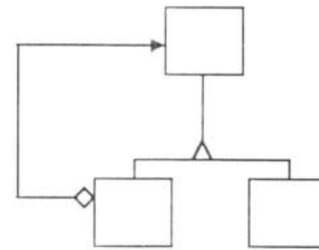

Figure 3.c. Recursive aggregate.

We can say the Composite pattern implements the "consists of" association, which permit establish an aggregation relationship. And it defines a recursive aggregation, which is easy to understand it, comparing the figure 3.c with the pattern structure (figure 2.1), which was shown in Section 2, when the essential elements of the Composite Pattern were presented.

3.3. Recursive Composition.

A common way to represent hierarchically structured information is through a technique called recursive composition, which entails building increasingly complex element out of simpler ones. We can use the recursive composition to represent any potentially complex, hierarchical structure. The Composite pattern captures the essence of recursive composition in object-oriented terms [1].

In order to allow recursive building of hierarchy in the Composite pattern, the subclass Composite has a reference to its Component superclass. In particular, this reference is a set of polimorphic variables (this set is called "children") that represent an aggregation relationship.

The recursive structure is built by the way of:

- inheritance
- a polimorphic variable[1] that implement the dynamic binding.

[1] A variable is considered a polimorphic variable if it accepts not just objects of one types but many types[1].

4. Formalising the Pattern Structure

The constraints mentioned in [6] are extended in order to support the Composite structure:

Participants

The common features described in [6] are reused to define the roles and responsibilities of the Composite Pattern participants.

In order to specify the composite scheme, have been established the roles, responsibilities and a variable, which are part of definition of the Composite Pattern. These and other constraints are applied to the composite participant instantiation defined by the object Cp.

```
scheme
  OB_COMPOSITE =
     class
       type
          Role_Type == component | composite | leaf | client,
          Sig_Type == operation | add | remove | getchild,
          Vble_Type == children | _ ,
          Param_Type = Role_Type | int | _ ,
     end
object
     Cp : OB_COMPOSITE
```

According to the Composite structure, there is exactly one class playing the component role, and another one class playing the composite role. The *exist_one_component* and the *exist_one_composite* definitions introduce these properties.

```
scheme
  COMPOSITE_PART =
    extend GEN_PART(Cp) with class
     value
      exist_one_component: PS.Wf_Pattern_Structure → Bool
      exist_one_component (ps) ≡ exist_one (Cp.component ) (ps),

      exist_one_composite: PS.Wf_Pattern_Structure → Bool
      exist_one_composite (ps) ≡ exist_one (Cp.composite ) (ps),
     end
```

A composite structure can only be invoked by a class whose role type is client. And a client sends its requirements to the class whose role type is component.

```
composite_client: PS.Wf_Pattern_Structure → Bool
composite_client (ps) ≡ can_only_invoke (Cp.client, Cp.component) (ps),
```

The *exist_leaf* definition verifies there must be at least one leaf role.

```
exist_leaf: PS.Wf_Pattern_Structure  →  Bool
exist_leaf (ps) ≡ exist_role (Cp.leaf ) (ps),
```

The class playing the component role is the parent of the subclasses whose role is either leaf or composite.

```
has_parent_comp: PS.Wf_Pattern_Structure  →  Bool
has_parent_comp (ps) ≡
 (
  has_parent (Cp.leaf, Cp.component) (ps) ∨
  has_parent (Cp.composite, Cp.component) (ps)
 ),
```

The following definition "knows its objects" means that composite and leaf objects are in the composite's state. This definition was used in [6] to specify a similar property in the observer pattern.

```
knows_its_children : PS.WF_Pattern_Structure → Bool
knows_its_children (ps) ≡
 (
  knows_its_objects(Cp.composite, Cp.composite) (ps) ∧
  knows_its_objects(Cp.composite, Cp.leaf) (ps)
 ),
```

There is a problem that involves safety and transparency. Defining the child management at the Component class gives transparency because all its components (Leaf and Composite) can be treated uniformly. This is not safety enough because the operations *add* and *remove* can be done from a Leaf. The decision we have taken was emphasise transparency over safety.

There are four ways to implement transparency. We assume that the interface's operations are always defined in the Component class (accordingly to implement transparency).

- First. Implement child management operations as default operations in the Component class. And redefine these operations only in Composite class.
- Second. Define child management operations in Component class and to implement them in both subclasses. There will be meaningless operations defined in a Leaf role. In this cases the meaningful of these operations would be changed to have sense.
- Third. Implement child management operations only in the Composite class leading this operations in both Component and Leaf classes, defined but not implemented. But in this case any intent to do a child-related operation in a Leaf, will result in a system error.
- Fourth. Define child management operations in the Component class, implemented as error in Leaf class and implemented as real in composite class.

Any intend to do a meaningless operation in a Leaf object will be detected as error.

We choose the last alternative to formalise the Composite Pattern.

By previous comment, some abstract declarations of the preliminary model in [6] has been changed in this extended work, to express that a signature may be real or error implemented. Because that, we had modified the global definition *Interface_Type* as follow:

```
scheme
 TYPES =
  class
   type
    ...
    Interface_Type == defined | implemented_real | implemented_error,
    ...
  end
```

Also the abstract declarations *has_impl_type_interface* and *has_impl_interface* must indicate either a subclass has all its signatures 'real' implemented or it has at least one signature 'real' implemented, and the others implemented as an 'error'. Then we substituted these two, by four new abstract declarations:

```
has_impl_type_real_interface:
PR.Role_Type × PR.Sig_Type → PS.WF_Pattern_Structure → Bool
has_impl_type_real_interface(r, s)(psc, psr) ≡
 (
  ∀ c : C.Wf_Class •
   c ∈ psc ∧ role_type(C.p_role(c)) = r ⇒
   ( ∃ sg : P.Signature •
      sg ∈ C.class_interface(c) ∧
      P.sig_name(P.sig_head(sg))) = s ∧
      {} = P.sig_parameters(P.sig_head(sg)) ∧
      P.interface_type(sg) = G.implemented_real )
 ),

has_impl_type_error_interface: PR.Role_Type × PR.Sig_Type →
PS.WF_Pattern_Structure → Bool
has_impl_type_error_interface(r, s)(psc, psr) ≡
 (
  ∀ c : C.Wf_Class •
   c ∈ psc ∧ role_type(C.p_role(c)) = r ⇒
   ( ∃ sg : P.Signature •
      sg ∈ C.class_interface(c) ∧
      P.sig_name(P.sig_head(sg))) = s ∧
      {} = P.sig_parameters(P.sig_head(sg)) ∧
      P.interface_type(sg) = G.implemeted_error )
 ),
```

```
has_impl_error_interface: PR.Role_Type × PR.Sig_Type ×
PR.Param_Type → PS.WF_Pattern_Structure → Bool
has_impl_error_interface(r, s, p)(psc, psr) ≡
 (
   ∀ c : C.Wf_Class •
     c ∈ psc ∧ role_type(C.p_role(c)) = r ⇒
     ( ∃ sg : P.Signature, pm : G.Parameter •
        sg ∈ C.class_interface(c) ∧
        P.sig_name(P.sig_head(sg))) = s ∧
        pm ∈ P.sig_parameters(P.sig_head(sg)) ∧
        parameter_type(pm) = p ∧
        P.interface_type(sg) = G.implemented_error )
 ),

has_impl_real_interface: PR.Role_Type × PR.Sig_Type × PR.Param_Type
→ PS.WF_Pattern_Structure → Bool
has_impl_real_interface(r, s, p)(psc, psr) ≡
 (
   ∀ c : C.Wf_Class •
     c ∈ psc ∧ role_type(C.p_role(c)) = r ⇒
     ( ∃ sg : P.Signature, pm : G.Parameter •
        sg ∈ C.class_interface(c) ∧
        P.sig_name(P.sig_head(sg))) = s ∧
        pm ∈ P.sig_parameters(P.sig_head(sg)) ∧
        parameter_type(pm) = p ∧
        P.interface_type(sg) = G.implemented_real )
 ),
```

The class whose role is component is an abstract class, then its subclasses should be concrete class, because in the general properties declared in [6] the abstract declaration *is_implemented_signature* specifies that a defined signature in an inheritance relation must be implemented by subclasses.

In the composite pattern, the two subclasses of the class playing the component role, are concrete classes. The interface of the class whose role type is composite, has all signatures implemented as real.

```
is_concrete_composite: PS.WF_Pattern_Structure → Bool
is_concrete_composite (ps) ≡
              is_concrete(Cp.component, Cp.composite)(ps)
```

Instead, in the interface of the class whose role type is leaf, there is only one signature, *operation()*, that it is 'real' implemented. The others signatures are implemented as error.

```
is_concrete_ leaf : PS.WF_Pattern_Structure → Bool
is_concrete_leaf (ps) ≡
              is_concrete_special (Cp.component, Cp.leaf)(ps),
```

And *is_concrete_special* is the abstract declaration that specifies the interface of a concrete class as the class already mentioned.

```
is_concrete_special : PR.Role_Type × PR.Role_Type →
PS.WF_Pattern_Structure → Bool
is_concrete_special(r1, r2) (psc, psr) ≡
 (
   ∀ c1, c2 : C.Wf_Class, s1 : P.Signature •
     c1 ∈ psc ∧
     c2 ∈ psc ∧
     role_type(C.p_role(c1)) = r1 ∧
     role_type(C.p_role(c2)) = r2 ∧
     s1 ∈ C.class_interface(c1) ∧ P.interface_type(s1) = G.defined ⇒
      ( ∃ s2 : P.Signature •
        P.sig_name(P.sig_head(s2)) = P.sig_name(P.sig_head(s1)) ∧
        s2 ∈ C.class_interface(c1) ∧ P.interface_type(s2) =
        (G.implemented_real ∨ G.implemented_error) )
 ),
```

The class playing a composite role has in its state a variable that contains all its child objects.

```
store_component : PS.WF_Pattern_Structure → Bool
store_component (ps) ≡ store_vble(Cp.composite, Cp.children)
```

In the structure of the Composite Pattern the composite and component class participants are connected by an aggregation relationship with many-many cardinality.

```
composite_relation: PS.WF_Pattern_Structure → Bool
composite_relation(ps) ≡
      like_bridge_relation(Cp.composite, Cp.component, G.aggregation,
      G.many)(ps)
```

A new abstract declaration needs to be defined in order to specify signatures with parameters.

```
has_def_interface : PR.Role_Type × PR.Sig_Type × PR.Param_Type →
PS.Wf_Pattern_Structure → Bool
has_def_interface (r, s, p) (psc, psr) ≡
 (
   ∀ c : C.Wf_Class•
     c1 ∈ psc  ∧ role_type(C.p_role(c) ) = r ⇒
     (
       ∃ sg : P.Signature, pm : G.Parameter •
        sg ∈ C.class_interface(c) ∧
        signature_type (P.sig_name(P.sig_head(sg))) = s  ∧
        parameter_type(pm) = p  ∧ P.interface_type(sg) = G.defined
     )
 ),
```

The four following constraints define the interface of the composite pattern in the class playing the role component, to lets clients treat individual and composite objects uniformly. The first signature defined below has not parameters, but the others three have parameters. For the last three signatures *has_def_interface will be* used.

```
operation_defined : PS.WF_Pattern_Structure → Bool
operation_defined (ps) ≡
     has_def_type_interface (Cp.component, Cp.operation) (ps)

add_defined: PS.WF_Pattern_Structure → Bool
add_defined (ps) ≡
     has_def_interface (Cp.component, Cp.add, Cp.component),

remove_defined : PS.WF_Pattern_Structure → Bool
remove_defined (ps) ≡
     has_def_interface (Cp.component, Cp.remove, Cp.component),

get_child_defined : PS.WF_Pattern_Structure → Bool
get_child_defined (ps) ≡
     has_def_interface (Cp.component, Cp.remove, Cp.int),
```

We had said the operation, *operation()*, is defined in the class whose role type is component, and it is 'real' implemented in its subclasses. This is specified by the way of follow abstract declaration:

```
operation_implemented : PS.WF_Pattern_Structure → Bool
operation_implemented (ps) ≡
     has_impl_type_real_interface (Cp.composite, Cp.operation) (ps) ∧
     has_impl_type_real_interface (Cp.leaf, Cp.operation) (ps)
```

The class playing the role composite implements the child management operations (add, remove, get_child), and any intent to misuse these operations (to add/remove objects from leaves) in an object of the class playing the leaf role are implemented as error.

```
add_impl : PS.WF_Pattern_Structure → Bool
add_impl(ps) ≡
 (
  has_impl_type_real_interface(Cp.composite, Cp.add, Cp.component) ∧
  has_impl_type_error_interface(Cp.leaf, Cp.add, Cp.component)
 ),

remove_impl : PS.WF_Pattern_Structure → Bool
remove_impl(ps) ≡
 (
  has_impl_type_real_interface(Cp.composite, Cp.remove, Cp.component) ∧
  has_impl_type_error_interface(Cp.leaf, Cp.remove, Cp.component)
 ),
```

```
get_child_impl : PS.WF_Pattern_Structure → Bool
get_child_impl(ps) ≡
 (
  has_impl_type_real_interface(Cp.composite, Cp.get_child, Cp.int) ∧
  has_impl_type_error_interface(Cp.leaf, Cp.get_child, Cp.int)
 ),
```

Composite Collaborations

In order to specify the Collaborations of the Composite Pattern, we had distinguished two important parts. In a first part we define the valid behaviour of collaborations in the Composite Pattern, using the abstract declaration *valid_behaviour_composite*. And the other is directed to establish the constraints about the pattern's collaborations.

```
valid_behavior_composite : CO.WF_Colls → Bool
valid_behaviour_composite (m) ≡
 (
  recipient_is_composite (Cp.client, Cp.composite, Cp.operation)(m) ∨
  recipient_is_leaf (Cp.client, Cp.leaf, Cp.operation)(m))
 ),
```

In the Section 2 the collaborations for the Composite Pattern were presented. Follow the preliminary model in [6] the collaborations are ordered by identifiers in the well-formed collaboration mapping. When a client interacts with objects in the composite structure is possible to note two situations. In both of them a client sends a request to an object in the structure. Depending on which is the receiver object, the request should be forwarded or not.

1. The recipient is a Leaf, formally described by:

```
recipient_is_leaf: PR.Role_Type × PR.Role_Type × PR.Sig_Type →
CO.WF_Colls → Bool
recipient_is_leaf (r1,r2,s)(m) ≡
 (
  ∃! id1:G.Coll_Id • { id1 } ⊆ dom m ∧
   coll_constraints(s,r1,r2,m(id1),{})
 ),
```

2. The recipient is a composite, and then it forwards requests to its child components. That is formally described by:

```
recipient_is_composite: PR.Role_Type × PR.Role_Type × PR.Sig_Type →
CO.WF_Colls → Bool
recipient_is_composite (r1, r2, s)(m) ≡
 (
  ∃! id1 : G.Coll_Id • { id1 } ⊆ dom m) ∧
    coll_constraints(s, r1, r2, m(id1),{}) ∧
```

forward_components(r2, s, id1)(m)

),

*id*1 identifies the unique collaboration between the client and the composite object. After that, the composite object forwards request to their child objects. Each of this request are indentify by different and unique id's and they have the same prerequisite, *id*1. We illustrate this in the figure 4.1.

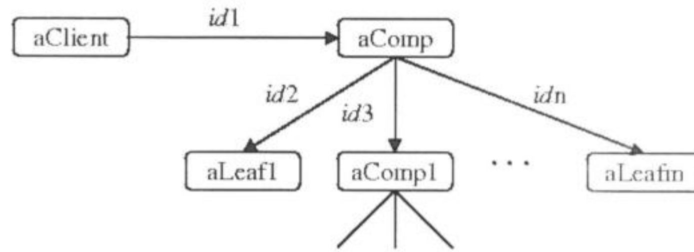

Figure 4.1

The following diagram (figure 4.2) generalize the previous:

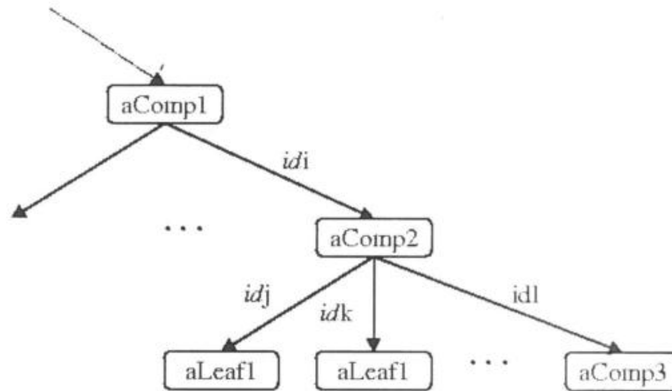

Figure 4.2

The *idi* collaboration between the *acomp1* and *acomp2* objects is the prerequisite for the collaborations between a *comp2* and its children. If that situation is valid, it should be valid for its composite child-objects, which is represented using the name of the abstract declaration *forward_components* being defined inside its own definition.

forward_components: PR.Role_Type × PR.Sig_Type × G.Coll_Id →
CO.WF_Colls → **Bool**
forward_components (r1, s, id1)(m) ≡
(
(∀ cp : G.concrete_Object •
role_type(C.p_role(C.object_class(cp))) = leaf ∧
object_in_receiver(cp, m(id1)) ⇒
(∃ ! id2: G.Coll_Id • { id2 } ⊆ **dom** m ∧
CO.receiver(Co.col(m(id2))) = C.object_class(cp) ∧

```
        coll_constraints(s, r1, leaf, m(id2), { id1 }) )
      ) ∧
      ( ∀ cp: G.concrete_Object •
        role_type(C.p_role(C.object_class(cp))) = composite ∧
        object_in_receiver(cp, m(id1)) ⇒
        ( ∃ ! id2: G.Coll_Id • { id2 } ⊆ dom m ∧
          CO.receiver(Co.col(m(id2))) = C.object_class(cp) ∧
          coll_constraints(s, r1, composite, m(id2), { id1 })
          ∧ forward_components(composite, s, id2)(m) )
      )
    ),
```

Is indispensable to establish some constraints to avoid any intent of describe collaborations not representing the valid behaviour of collaborations in Composite Pattern.

```
scheme
  COMPOSITE_COL =
    extend GEN_COL (Cp) with
        class
            value
            composite_coll : CO.WF_Colls → Bool
            composite_coll(m) ≡
             (
              ~ is_receiver(Cp.cliente)(m) ∧ ~ is_sender(Cp.leaf)(m)  ∧
              ~ loop_component (m) ∧ object_in_composite(m) ∧
              valid_behaviour_composite(m)
             )
            end
```

We had determined the following constraints, which have been included in the collaboration scheme of Composite Pattern:

- A client must not act as a receiver of collaboration.

```
is_receiver : PR.Role_Type → CO.WF_Colls → Bool
is_receiver (r ) (m) ≡
  ( ∃ id1 : G.Coll_Id  • role_type_receiver (m( id1) ) ),
```

- A leaf must not act as a sender of collaboration.

```
is_sender : PR.Role_Type → CO.WF_Colls → Bool
is_sender (r ) (m) ≡
  ( ∃ id1 : G.Coll_Id  • role_type_sender (m( id1) ) ),
```

- All component objects receiver of a collaboration whose sender is a composite must be in the composite's state.

```
object_in_composite: CO.WF_Colls → Bool
object_in_composite (m) ≡
 (
  ∀ id1 : G.Coll_Id • role_type_sender( m(id1) ) = Cp.composite
    ⇒ receiver_in_sender( m(id1) , Cp.children )
 ),
```

The abstract declaration *is_successor* finds directly or indirectly the successor of an object in an object structure. That is used in *loop_component*, which try to find a loop in set of collaborations of Composite Pattern. The figure 4.3 shows a case of an object trying to send a request to an object that is its successor.

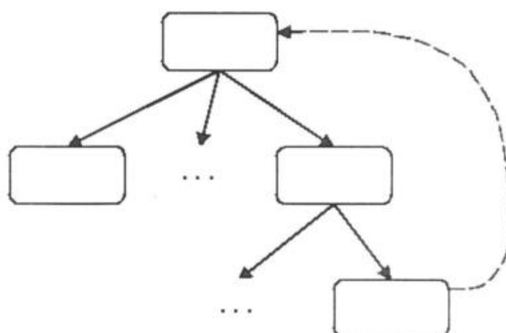

Figure 4.3

```
loop_component : CO.WF_Colls → Bool
loop_component (m) ≡
 ( ∃ id1, id2 : G.Coll_Id • isSuccessor ( id1, id2) (m) ∧
    isSuccessor (id2, id1) (m) ),

is_successor: G.Coll_Id × G.Coll_Id → CO.WF_Colls → Bool
is_successor ( id2, id1) (m) ≡
  ( role_type_receiver ( m(id1) ) = role_type_sender ( m(id2) ) ∨
   ( ∃ id3 : G.Coll_Id •
      isSuccessor (id3, id1) (m) ∧ isSuccessor (id2, id3) (m) )
  ),
```

5. Conclusions

In this report, we have presented a formal specification of a pattern belonging to the group of GoF patterns based on recursive structure, the Composite Pattern. Our work was based on the preliminary model [6] built over five patterns in the GoF catalogue. The specification was performed in RSL. This contains all the participants with their responsibilities and collaborations, which we consider they can be formalised.

References

[1] Gamma E., Helm R., Johnson R., and Vlissides J., *Design Pattern-Element of Reusable Object Oriented Software*. Addison Wesley, 1995.

[2] Wirfs-Brock R., Wilkerson B., and Wiener L., *Designing Object Oriented Software*. Englewood Cliffs, NJ: Prentice Hall, 1990.

[3] Pree W., *Design Patterns for Object Oriented Software Development*. Eddison Wesley, 1995.

[4] Rumbaugh J., Blaha M., Premerlani W., Eddy F., Lorensen W., *Object-Oriented Modelling and Design*, Prentice-Hall International.

[5] Dahl J. O., Dijkstra E.W., Hoare C.A.R., *Structured Programming*, Computer Science Classics.

[6] Cechich, A., Moore, R., *A Formal Specification of GoF Design Patterns*, UNU/IIST Report No. 151, 1999.