Verifying Meta-Patterns by extending a Formal Model of OO-Design.

Luis Reynoso and Alejandra Cechich

Departmento de Informática y Estadística - Universidad Nacional del Comahue.
Buenos Aires 1400 (8300) - Neuquen - Argentina
lreynoso, acechich@uncoma.edu.ar

Abstract. Meta-patterns are useful to document and to understand frameworks because they contribute to highlight their hot spots (aspects of an application domain that have to be kept flexible and can be adapted by applications built on frameworks). In this paper we describe an RSL -RAISE [1] Specification Languague- formal model to verify the use of meta-patterns in a design. Since each meta-pattern shows structural mechanisms for implementing variable aspects of a framework, our model was built by using a previous model of an object-oriented design as a way of representing a generic design. The properties of seven of Pree's meta-patterns ([16]) have been specified in a meta-level defined in terms of elements of a design, i.e. those provided by the formal model basis.

1 Introduction

As software engineering has evolved, new concepts and terminology were defined, such as software architectures, design patterns, or frameworks. Formal methods communities have contributed to describe them precisely; for instance, there are formalisations to describe software architectures [17],[18],[6]; analysis and design patterns [15], [7], [13], [3], [5]; aspects of UML [14], [8], [2], etc.

The aim behind these formalizations is to contribute to defining the semantics of the new concepts referred to above, adding precision for applying them. For instance, using formal methods allows to illustrate and describe how families of applications can be classified in terms of different software architectures; how families of application designs could implement different variable aspects represented by a design pattern; or how a family of domain-dependent applications could reuse a software architecture, designed in such a way that its components could be replaceable at variation points or hot-spots [10]. The three concepts mentioned – software architectures, design patterns and frameworks – deal with families of application designs which must exhibit general constraints inherent to the particular concept being used.

[1] Acronym of 'Rigorous Approach to Industrial Software Engineering'

UML, OMT, and other similar notations are developed to express a single application design at a time, and because of that many authors have observed their lack of mechanisms for modelling general concepts of families of designs [6],[11],[10].

In [9] we presented a formal model of an object-oriented design, which could be used for verifying the existence of a design pattern. In this paper, we use that formal model to represent two complementary designs: a framework design and an application design built by using the framework. Our purpose is to verify how the framework's points of variability and extensibility (hot spots, also called variable points) are implemented in the application. The verification can also be applied to a general design, not only to a framework and its instantiation. We use the theory of Pree's meta-patterns [16] for expressing different kinds of framework variation points.

The rest of the paper is divided as follows. Section 2 presents the basic concepts behind the formal model of an object-oriented design written in RSL [4],[19]. Section 3 briefly introduces Pree's Meta-patterns [16]. Section 4 describes the new formal model for matching designs against meta-patterns. Finally Section 5 describes the formal properties of seven of Pree's meta-patterns in the new formal model. We present in the last part, Section 6, our conclusions and future work.

2 Modelling an Object Oriented Design

A Formal Model of an Object-Oriented Design (OOD) was presented in [9] and [1]. In this model, it is possible to represent the structural aspects of a design by using an RSL [2] type representing a well-formed design structure, which is essentially a combination of two components: a set of Classes and a set of Relations.

```
Design_Structure = C.Classes × R.Wf_Relations,
Wf_Design_Structure =
                {| ds : Design_Structure • is_wf_design_structure(ds) |}
```

The main constituents of an object-oriented design are defined in the Model using RSL types such as maps, lists, sets, and variant and record types[3]. The RSL types were chosen according to the concept being represented.

The RSL specification of the Model is divided into five RSL modules or schemes which also correspond to the sections of the technical report [1] where the whole formalisation can be found. The five modules or schemes are:

1. TYPES-Module consists of general definitions of the model.

[2] A basic knowledge of RSL or some similar formal specification language is assumed.
[3] basics types of RSL can be found in [4].

2. METHODS-Module defines a set of methods for a class. Methods have a signature, and if the method is implemented its functionality can also be represented as a list of invocations and a map showing the variable changes produced by the method.
3. DESIGN_CLASS-Module uses the METHODS-module in order to define a class along with the state class. The definition of a class is used to formally define a set of classes.
4. DESIGN_RELATION-Module defines a set of possible relations that can be depicted in a design. The classes connected by a relation are identified by means of their names.
5. DESIGN_STRUCTURE-Module defines the correspondence between the last three modules.

G, M, C, R, and DS are object names that represents a scheme or module described in the previous list from 1 to 5 respectively. These names are used to refer to elements defined in one scheme from definitions in another scheme. Also, the way in which the new specification presented in this paper refers to a property defined in the general model is by using the object name of a scheme followed by a dot and the name of the property referred to. The following table 1 shows some examples of different properties of the OOD Model we reuse in this paper; their complete definition is included in [1].

Table 1. Some functions of the OOD which are reused in the Meta-pattern specification.

Property Name	Short Description
R.relation_type(r)	obtains the relation type of a relation r
R.source_class(r)	obtains the source class of a relation r
R.sink_class(r)	obtains the sink class of a relation r
R.relation_name(r)	obtains the name of a relation r
R.ag_ref(tr)	obtains the relation name, its sink and source cardinality if tr is an aggregation relation type
R.is_superclass(c1,c2,rs)	checks whether a class $c1$ is a superclass of the class $c2$ with respect to some set of relations rs
M.is_implemented(m)	verifies whether a method m is implemented
DS.has_state_var(v,c,ds)	checks whether a given state variable v is defined in a given class c
DS.class_of_method(m,c,ds)	returns the class c which contains the definition of the method as seen by the class c
DS.has_method(m,c,ds)	determines whether a given method m is defined in a given class c, either declared locally or inherited.

It is possible to instantiate the O-O design formal model with a particular design. To do that, a designer should only express the graphical notation of a

class diagram extended with the equivalent RSL type's structures defined in the model. No additional effort is required than mapping the constituent components of a class or relation to the same element in the formal model and composing those components.

The instantiation of a design is then verified for consistency properties. For example, checking the correspondence between the invocations received by a method and the appropriate methods defined or modelled in the invoking class; checking that a class receiving a request should be connected by a structural relation to the invoking class, etc. A complete example including a formal specification and its verification is shown in [9].

The whole design should exhibit all the well-formedness conditions defined for methods, class state, relations, parameters and so on, written in the value definition *is_wf_design_structure*. We refer readers to [9] for a detailed overview of the formal model. The specification of the Model and the new specification shown in this paper were syntax-checked and type-checked using the RAISE Tools [4].

3 Frameworks and Meta-patterns.

A framework implements the software architecture for a family of applications with similar characteristics [10]. A framework design shows the structure or skeleton of a software architecture, and applications built on that framework show a possible instantiation by implementing specific needs of an instance of the family of applications. Generally, frameworks contain a set of spots, hot or frozen. A frozen spot describes the basic functional structure of an application domain, while hot spots are the locations in a framework where parameterisation takes place.

Both spots express the knowledge of the framework's designers about the specific domain the framework represents. In other words, designers can indicate two kinds of responsibilities in terms of framework functionalities: those parts which are the exclusive responsibility of the framework designers, which are implemented by means of frozen spots; and parts where the responsibilities are shared between framework designers and framework application designers, which are implemented by means of hot spots.

There should be a great effort to understand what the framework does and how, and what it requires. In this area, Pree's meta-patterns can be used to document and to represent part of a framework's variability. Each meta-pattern is a hot spot.

In [16], Pree describes hot spots as composed of two kinds of methods: template and hook methods. These two kinds of methods are essentially similar to

[4] RAISE Tools (both source code and executables) used to process the meta-pattern specifications are available at: http://wwww.iist.unu.edu/raise

the TemplateMethod and PrimitiveOperation methods of the Template Method Pattern in [12]. In both cases a Template Method must invoke at least one Hook Method (or PrimitiveOperation).

In a Template Method Pattern, hook methods are always included in a subclass of the class that contains the Template Method. However, Pree describes different configurations of the classes, which could contain both kinds of methods (template or hook), and determines seven meta-patterns (defined in [16]) as depicted in Figure 1.

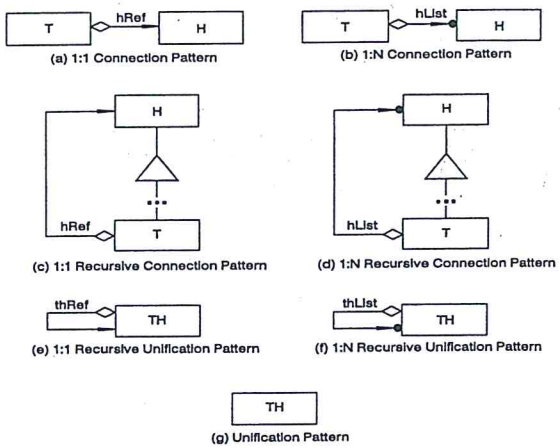

Fig. 1. Pree's Meta-Patterns

A T-box represents a Template class whereas an H-box represents a Hook class. The difference between some of the configurations of these two classes depends only on the cardinalities of the relations that connect them, with the name hRef describing a *reference* explicitling name an arrowhead aggregation while the name hList represents a *list of references* which is depicted in the diagram using an arrowhead aggregation with a black dot at the end (which is the notation used for relations whose cardinality is "many").

Frameworks designs represent an unfinished design, i.e. they only model the invariant part of a family of designs of a specific domain, leaving some incomplete parts (the variant parts) to be appropriately customised by application

designers. Both designs, the framework design and the framework instantiation design, must constitute a complete design.

Example 1. Suppose a framework for the domain of reservation systems is developed in which a rate calculation has to become one of the hot spots. The framework could include two classes RentalItem and RateCalculator, and an application framework could customise these two classes, defining subclasses of them in order to adapt the Item (subject) of a rental (in a rental car reservation system the Item would be a class Vehicle; in a hotel reservation system it would be a Room) and the rate calculation behaviour. Figure 2 [5] shows a subclass StdRC inherent to such an application framework which customises the rate calculation behaviour.

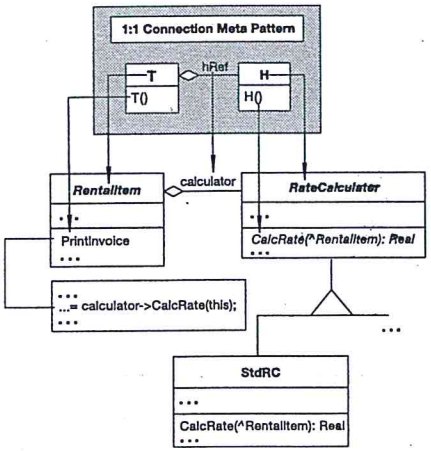

Fig. 2. Example of a design of an application framework and a hot spots.

Figure 2 also show how a 1:1 Connection meta-pattern could be chosen in the framework design to represent the hot spot of how rental rates are calculated.

We can use the type Design_Structure ([9],[1]), a simple combination of classes and relations, to represent the two complementary component designs of a framework. Both of them can be unified in a new design for checking consistency

[5] The design is also a motivation sample of [16].

conditions of the whole design (using the property *is_ wf_ design_ structure* of [1]).

```
scheme
  FRAMEWORKS =
class
 type
    Framework= DS.Design_Structure,
    Application= DS.Design_Structure,
    ApplicationFramework = DS.Design_Structure,
    Wf_ApplicationFramework=
                {| af : ApplicationFramework • is_correct_design(af) |}

 value
    application_Framework:
                Framework × Application → ApplicationFramework
    application_Framework((fd, fr),(ad, ar)) ≡ ((fd ∪ ad), (fr ∪ ar)),

    is_correct_design: ApplicationFramework → Bool
    is_correct_design(ds) ≡ DS.is_wf_design_structure(ds)
 end
```

4 Matching Designs to Meta-patterns

Now, we go on to explain how the model of a generic design (possibly a Wf_ ApplicationFramework design) could be linked to a meta-pattern in such a way that it is possible to determine whether or not the two match.

The link is done using a *meta-pattern renaming map*, a similar approach to that applied in the specification of design patterns (found in [9]) but with different type structures and properties.

The meta-pattern renaming map allows us to associate names of classes, methods and variables in a design with constant names appearing in a meta-pattern.

We define five RSL constant names of a meta-pattern: hook_class, hook_method, template_class, template_method and vRef, the last representing either an hRef or an hList variable name of a meta-pattern (i.e. there is no distinction in the model in terms of these names). These constants will represent roles played by design entities in the meta-pattern level.

Example 2. In figure 2 the correspondence or renaming is represented by the arrows which link RentalItem to a TemplateClass (represented as **T** in the figure), PrintInvoice to Template_method (represented as T() in the figure), and so on.

The renaming of a class, which is represented by the RSL type MetaClassRenaming, consists of the constant name of the class in the meta-pattern together with two renaming map, one for methods in the class (it simply maps a method in the design to one of the possible meta-pattern names: template_method or hook_method) and the other for each variable of the class.

```
scheme
  METAPATTERN_RENAMING =
class

type
MetaClassRenaming ::
        classname : G.Class_Name
        methodRenaming : G.Method_Name ~m~> G.Method_Name
        varRenaming : G.Variable_Name ~m~> G.Variable_Name,
value
        Template_class, Hook_class: G.Class_Name,
        Hook_method, Template_method : G.Method_Name,
        vRef : G.Variable_Name,
```

The well-formedness condition on the type MetaClassRenaming expresses the condition that if the class name is the constant Hook_class then the renaming of variables must be empty and all the methods should rename to Hook_method. In a similar way, if the class name is the constant Template_class all its method renaming maps should be Template_Method and the number of elements of the variable renaming must be one.

```
Wf_MetaClassRenaming =
    {| r : MetaClassRenaming • is_wf_MetaClassRenaming(r) |},

is_wf_MetaClassRenaming: MetaClassRenaming → Bool
is_wf_MetaClassRenaming(cr) ≡
    case classname(cr) of
        Hook_class → varRenaming(cr) = [ ] ∧
                (∀ m: G.Method_Name •
                        methodRenaming(cr)(m) = Hook_method),
        Template_class →
                card dom varRenaming(cr) = 1 ∧
                (∀ m: G.Method_Name •
                        methodRenaming(cr)(m) = Template_method)
    end,
```

However, it is possible for a single class in the design to play two different roles in the meta-pattern (but only in a unification pattern; see 1). We therefore map each design class to a set of class renamings in the meta-pattern renaming map.

```
MetaPatternRenaming = G.Class_Name ~m~> Wf_MetaClassRenaming-set,
```


The well-formedness condition requires that no design class can have either an empty set of renamings or a set containing more than two elements. In addition, because in a recursive connection and connection meta-patterns there should be two different design classes playing respectively Template_Class and Hook_class, the well-formedness condition also requires that the maps must have two elements for a meta-pattern.

When the maps have only one element, in a unification meta-pattern, the design class must have two class renamings in the meta-pattern-renaming map.

```
Wf_MetaPatternRenaming =
        {| r : MetaPatternRenaming • is_wf_MetaPatternRenaming(r) |},

value

is_unification: MetaPatternRenaming → Bool
is_unification(r) ≡
    ((card dom r) = 1 ) ⇒
        (∃ c: G.Class_Name •
                (card r(c)) = 2 ∧
                (∃ c1, c2 : MetaClassRenaming•
                        c1 ≠ c2 ∧ c1 ∈ r(c) ∧
                        classname(c1) = Template_class ∧
                        c2 ∈ r(c) ∧ classname(c2) = Hook_class))


is_connection: MetaPatternRenaming → Bool
is_connection(r) ≡
    (card dom r = 2) ⇒
        (∃ c1, c2: G.Class_Name •
                c1 ≠ c2 ∧
                (card r(c1)) = 1 ∧
                (∃ c11: MetaClassRenaming •
                        c11 ∈ r(c1) ∧
                        classname(c11) = Template_class) ∧
                        (card r(c2)) = 1 ∧
                (∃ c21: MetaClassRenaming•
                        c21 ∈ r(c2) ∧
                        classname(c21) = Template_class)),

is_wf_MetaPatternRenaming: MetaPatternRenaming → Bool
is_wf_MetaPatternRenaming(r) ≡
        (is_unification(r) ∨ is_connection(r)),
```

Since a design could include many meta-patterns, we define the type Design_MetaPatternRenaming as a well-formed design and a set of well-formed MetaPatternRenaming (the set does not allow two equal meta-patterns).

```
Design_MetaPatternRenaming =
        DS.Wf_ApplicationFramework × Wf_MetaPatternRenaming-set,

Wf_Design_MetaRenaming =
        {| pr : Design_MetaPatternRenaming •
        is_wf_design_metapatternrenaming(pr) |}
```

5 Specification of Meta-patterns

Since the meta-pattern specification is an extension of the OOD Model we should connect the two models and define the properties of Pree's meta-patterns.

5.1 Connecting the two formal models

There are four essential properties which the seven Pree's meta-patterns should verify individually as listed in this section. Although all of these properties have been formalised, we include here only two formal expressions for brevity reasons.

- The class playing the Template_Class role contains one implemented method (renamed to Template_Method).

```
Template_method_implemented:  Design_MetaPatternRenaming → Bool
Template_method_implemented ((dsc, dsr), mr) ≡
      (∀  mp: Wf_MetaPatternRenaming,
          cr: Wf_MetaClassRenaming,
          c: G.Class_Name, me: G.Method_Name •
                mp ∈ mr ∧
                c ∈ dom mp ∧ cr ∈ mp(c) ∧
                classname(cr) = Template_class ∧
                me ∈ dom methodRenaming(cr)
       ⇒
         DS.has_method(me, c, (dsc, dsr)) ∧
         let c =
                DS.class_of_method(me, c, (dsc, dsr)) in
                M.is_implemented(C.class_methods(dsc(c))(me))
       end),
```

- All the class, method and variable names used in the domain maps of wf_MetaPatternRenaming-set are elements of the Wf_Design_Structure.


```
is_correct_domain :  Design_MetaPatternRenaming → Bool
is_correct_domain(dmr) ≡
            domain_class_name(dmr) ∧
            domain_method_name(dmr) ∧
            domain_variable_name(dmr) ∧
            Template_method_implemented(dmr),


domain_class_name :  Design_MetaPatternRenaming → Bool
domain_class_name((dc, dr), mr) ≡
      (∀ m: Wf_MetaPatternRenaming •
            m ∈ mr ⇒ dom m ⊆ dom dc ),


domain_method_name : Design_MetaPatternRenaming → Bool
domain_method_name((dsc, dsr), mr) ≡
      (∀ mp: Wf_MetaPatternRenaming,
            cr: Wf_MetaClassRenaming,
            c: G.Class_Name,  me: G.Method_Name •
                  mp ∈ mr ∧
                  c ∈ dom mp ∧ cr ∈ mp(c) ∧
                  me ∈ dom methodRenaming(cr)
                  ⇒
                  DS.has_method(me, c, (dsc, dsr))),

domain_variable_name : Design_MetaPatternRenaming → Bool
domain_variable_name(d, mr) ≡
    (∀ mp: Wf_MetaPatternRenaming,
     cr: Wf_MetaClassRenaming,
     c: G.Class_Name, vd: G.Variable_Name •
           mp ∈ mr ∧ c ∈ dom mp  ∧ cr ∈ mp(c) ∧
           vd ∈ dom varRenaming(cr)
           ⇒
           DS.has_state_var(vd, c, d)),
```

- The body of a Template_method contains at least one invocation to a method that plays the Hook_method.

- 'The Hook_method can be an abstract method, a regular method that calls no other method, or again a template method' [16] (this property is supported by the RSL Model of OOD.).

5.2 Connection Patterns

There are two connection meta-patterns: 1:1 Connection pattern (11CP) and 1:N Connection pattern (1NCP) – see figure 1 (a) and (b). They represent an

abstract coupling between two different classes in the design playing the roles Template_class and Hook_class respectively.

The connection represented by a Ref relation has multiplicity one in the receiving class for the 1:1 CP and multiplicity many for the 1:N CP. Every invocation from a Template_Class to a Hook_Class uses a variable that plays the vRef role.

```
scheme
METAPATTERNS =
extend METAPATTERN_RENAMING

with class

value

design_class: G.Class_Name × Wf_MetaPatternRenaming → G.Class_Name
design_class(role, r) as c
        post c ∈ dom r ∧
             (∃ mc: MetaClassRenaming •
                        mc ∈ r(c) ∧ classname(mc) = role)
        pre card dom r = 2,

Connection:
F.Wf_ApplicationFramework × Wf_MetaPatternRenaming × G.Card→ Bool
Connection((fd,fr),r, t) ≡
         let  cd1 = design_class(Template_class,r),
              cd2 = design_class(Hook_class, r)
         in
             (∃  r: R.Wf_Relation, a: R.Ref •
                        r ∈ fr ∧
                        R.relation_type(r) = R.aggregation(a) ∧
                        R.source_class(r) = cd1 ∧
                        R.sink_class(r) = cd2 ∧
                        R.source_card(R.ag_ref(R.relation_type(r))) = t ∧
                        (∃ mr: MetaClassRenaming •
                                   mr ∈ r(cd1) ∧
                                   R.relation_name(R.ag_ref(R.relation_type(r)))
                                   ∈ dom varRenaming(mr)))
         end,

is_a_11CP:
        F.Wf_ApplicationFramework × Wf_MetaPatternRenaming → Bool
is_a_11CP(dr, r) ≡
           is_wf_MetaPatternRenaming(r) ∧
           is_connection(r) ∧
```



```
           Connection(dr, r, G.one),

is_a_1NCP:
        F.Wf_ApplicationFramework × Wf_MetaPatternRenaming → Bool
is_a_1NCP(dr, r) ≡
           is_wf_MetaPatternRenaming(r) ∧
           is_connection(r) ∧
           Connection(dr, r, G.many),
```

5.3 Recursive Connection Patterns

There are two recursive connection patterns: 1:1 Recursive Connection Pattern (11RC) and 1:N Recursive Connection Pattern (1NRC) - see figure 1 (c) and (d).

Example 3. The structure of the Composite and the MacroCommand patterns [12] shows a 1NRC meta-pattern, and the motivation of the Interpreter pattern [12] shows a possible use of 11RC in a definition of a grammar language.

Recursive connection patterns can be formally specified using the appropriate connection pattern with an additional restriction which expresses that the Template_class is a subclass of a Hook_class. An inheritance relation can connect these classes in a direct or an indirect way. To model it, we reuse the property *is_superclass* of [1] as follows:

```
hierarchical_prop:
      F.Wf_ApplicationFramework × Wf_MetaPatternRenaming → Bool
hierarchical_prop((dc, dr), r) ≡
            let  cd1 = design_class(Template_class,r),
                 cd2 = design_class(Hook_class, r)
            in   R.is_superclass(cd2, cd1, dr)
            end,

is_a_11RC:
      F.Wf_ApplicationFramework × Wf_MetaPatternRenaming → Bool
is_a_11RC(dr, r) ≡ is_a_11CP(dr, r) ∧
           hierarchical_prop(dr,r),

is_a_1NRC:
      F.Wf_ApplicationFramework × Wf_MetaPatternRenaming → Bool
is_a_1NRC(dr, r) ≡
           is_a_1NCP(dr, r) ∧ hierarchical_prop(dr,r)
```

5.4 Unification Patterns

There are three unification patterns as shown in Figure 1 (e), (f) and (g): 1:1 Recursive Unification Pattern (1:1RUP), 1:N Recursive Unification Pattern (1:NRUP), and Unification Pattern (UP).

UP, 1:1RUP and 1:NRUP verify general properties of meta-patterns including the property *is_unification*. 1:1RUP and 1:NRUP also include the property *Connection* with its appropriate parameter.

```
is_a_11UP:
    F.Wf_ApplicationFramework × Wf_MetaPatternRenaming → Bool
is_a_11UP(dr, r) ≡
        is_wf_MetaPatternRenaming(r) ∧ is_unification(r) ∧
        Connection(dr, r, G.one),

is_a_1NUP:
    F.Wf_ApplicationFramework × Wf_MetaPatternRenaming → Bool
is_a_1NUP(dr, r) ≡
        is_wf_MetaPatternRenaming(r) ∧
        is_unification(r) ∧
        Connection(dr, r, G.many),

is_a_UP :
    F.Wf_ApplicationFramework × Wf_MetaPatternRenaming → Bool
is_a_UP(dr, r) ≡
        is_wf_MetaPatternRenaming(r) ∧
        is_unification(r),
```

5.5 Verifying multiple meta-patterns in a design

After individually presenting the properties of Pree's meta-patterns, we can build a subtype of the Design_MetaPatternRenaming for modelling a well formed design along with a set of well formed meta-patterns in which each meta-pattern should verify one of the seven functions previously defined.

```
type

WF_Design_MetaPatternRenaming =
      {| wf_dm: Design_MetaPatternRenaming •
      is_wf_design_metapattern_ren_set(wf_dm) |}

value

is_wf_design_metapattern_ren_set: Design_MetaPatternRenaming → Bool
is_wf_design_metapattern_ren_set(d,mset) ≡
```



```
(
∀ e: Wf_MetaPatternRenaming• e ∈ mset ⇒
    (
      is_a_11RC(d,e) ∨ is_a_1NRC(d,e)  ∨
      is_a_11CP(d,e) ∨ is_a_1NCP(d,e) ∨
      is_a_11UP(d,e) ∨ is_a_1NUP(d,e) ∨
      is_a_UP(d,e))),
```

6 Conclusion

We have extended the formal model of object-oriented design presented in [9],[1] for modelling two complementary designs – a framework design and a framework instantiation – and we have explained how it is possible to combine both designs in a formal design structure and to check the semantics properties previously defined.

The well-formed structure introduced in this paper is used to verify the correct use of the seven Pree meta-patterns in the context of designing application frameworks. The precise and correct use of meta-patterns can improve the knowledge, understanding, and documentation of frameworks by letting framework designers and framework application designers precisely represent points of variability or extensibility of a family of designs.

In the next stage of our work, we are planning to implement the verification of Pree's meta-patterns in a Java tool, which is being developed for verifying the existence of GoF patterns from generic designs. Since the formal architecture of both kinds of verification are quite similar, extending the software tool can supply an enhanced supporting tool where patterns and meta-patterns will be synergically related.

7 Acknowledgments

Special thanks to Richard Moore (of IFAD, Denmark) for his invaluable comments and feedback on this work.

References

1. Flores A., Reynoso L., and Moore R. A Formal Model of Object-Oriented Design and GoF Design Patterns. Technical Report 200, UNU/IIST, P.O.Box 3058, Macau, August 2000.
2. Ruth Breu, Ursula Hinkel, Christoph Hofmann, Cornel Klein, Barbara Paech, Bernhard Rumpe, and Veronika Thurner. Towards a formalization of the unified modeling language. *Lecture Notes in Computer Science*, 1241, 1997.
3. A. Bucella and A. S. Cechich. A Formal Model for Some Behavioural Features of Analysis Patterns. In *IV Congreso Argentino de Ciencias de la Computacion*, Usuhaia, Argentina, 2001.

4. George C., Haff P., Havelund K., Haxthausen A., Milne R., Nielsen C., Prehn S., and Wagner K. *The RAISE Specification Language.* Prentice-Hall, 1992.
5. A. S. Cechich. A Formal Model for Semantics Statements of Analysis Patterns. In *IV Worshop Iberoamericano de Ingenieria de Requisitos y Ambientes de Software*, San Jose de Costa Rica, 2001.
6. Amnon H. Eden. LePUS: A Visual Formalism for Object-Oriented Architectures. In *Proceedings of the 6th World Conference on Integrated Design and Process Technology*, pages –, Pasadena, California, June 2002.
7. Amnon H. Eden, Y. Hirshfeld, and K. Lundqvist. LePUS : Symbolic Logic Modeling of Object Oriented Architectures: A Case Study. University of Karlskrona Ronneby, Ronneby, Sweden, 1999. Second Nordic Workshop on Software Architecture - NOSA'99.
8. A. Evans and A. Clark. Foundations of the unified modeling language. In Springer-Verlag, editor, *In 2nd Northern Formal Methods Workshop, Ilkley, Electronic Workshops in Computing*, 1998.
9. Andres Flores, Richard Moore, and Luis Reynoso. A Formal Model of Object-Oriented Design and GoF Design Patterns. In José Nuno Oliveira and Pamela Zave, editors, *FME2001: Formal Methods for Increasing Software Productivity*, volume 2021 of *Lecture Notes in Computer Science*, pages 223–241. Springer Verlag, 2001.
10. M. Fontoura, W. Pree, and B. Rumpe. UML-F: A Modeling Language for Object-Oriented Frameworks. In *BCS-FACS Northern Formal Methods Workshop.* Springer Verlag, 1997.
11. Marcus Fontoura, Carlos J. Lucena, Alexandre Andreatta, Sérgio E. Carvalho, and Celso C. Ribeiro. Using UML-F to enhance framework development: a case study in the local search heuristics domain. *The Journal of Systems and Software*, 57(3):201–206, 2001.
12. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison Wesley, 1995.
13. A. Lauder and S. Kent. Precise Visual Specification of Design Patterns. In *Ed. Jul, ECOOP'98*, pages 114–134. Springer-Verlag, 1998.
14. Zhiming Liu, Jifeng He, and Xiaoshan Li. Formalizing the Use of UML in Requirement Analysis. Technical Report 228, UNU/IIST, P.O.Box 3058, Macau, March 2001.
15. Tommi Mikkonen. Formalizing Design Patterns. In *Proceedings of the International Conference on Software Engineering ICSE'98*, pages 115–124. IEEE Computer Society Press, 1998.
16. W. Pree. *Design Patterns for Object-Oriented Software Development.* Addison-Wesley, 1995.
17. P.S.C. Alencar, D.D. Cowan, and C.J.P. Lucena. A Formal Approach to Architectural Design Patterns. In J. Woodcock M.-C. Gaudel, editor, *FME '96: Industrial Benefit and Advances in Formal Methods*, volume 1051, pages 576–594, Oxford, 1996. Springer Verlag.
18. Mary Shaw and David Garlan. Formulations and formalisms in software architecture. In *Computer Science Today*, pages 307–323. 1995.
19. Hung Dang Van, Chris George, Tomasz Janowski, and Richard Moore, editors. *Specification Case Studies in RAISE.* FACIT. Springer, 2002. ISBN 1-85233-359-6.