A Formal Model of Object-Oriented Design and GoF Design Patterns

Andres Flores[1], Richard Moore[2], and Luis Reynoso[1]

[1] Department of Informatics and Statistics - University of Comahue
Buenos Aires 1400, 8300 Neuquen, Argentina
E-mail: {aflores, lreynoso}@uncoma.edu.ar
[2] United Nations University - International Institute for Software Technology
P.O. Box 3058, Macau
E-mail: rm@iist.unu.edu

Abstract. Particularly in object-oriented design methods, design patterns are becoming increasingly popular as a way of identifying and abstracting the key aspects of commonly occurring design structures. The abstractness of the patterns means that they can be applied in many different domains, which makes them a valuable basis for reusable object-oriented design and hence for helping designers achieve more effective results. However, the standard literature on patterns invariably describes them informally, generally using natural language together with some sort of graphical notation, which makes it very difficult to give any meaningful certification that the patterns have been applied consistently and correctly in a design. In this paper, we describe a formal model of object-oriented design and design patterns which can be used to demonstrate that a particular design conforms to a given pattern, and we illustrate using an example how this can be done.

1 Introduction

Design patterns offer designers a way of reusing proven solutions to particular aspects of design rather than having to start each new design from scratch. Patterns are generic and abstract and embody "best practice" solutions to design problems which recur in a range of different contexts [1], although these solutions are not necessarily the simplest or most efficient for any given problem [10]. Patterns are also useful because they provide designers with an effective "shorthand" for communicating with each other about complex concepts [2]: the name of the pattern serves as a precise and concise way of referring to a design technique which is well-documented and which is known to work well.

One specific and popular set of software design patterns, which are independent of any specific application domain, are the so-called "GoF"[1] patterns which are described in the catalogue of Gamma et al. [9]. The GoF catalogue is thus a description of the know-how of expert designers in problems appearing in various different domains.

[1] "Gang of Four"

Although there is nothing in design patterns that makes them inherently object-oriented [2], the GoF catalogue uses object-oriented concepts to describe twenty three patterns which capture and compact the essential parts of corresponding design solutions. Each GoF pattern thus identifies a group of classes, together with the key aspects of their functionality and interactions, which commonly occur in a range of different object-oriented design problems.

The descriptions of the GoF patterns in [9] are largely informal, consisting of a combination of a graphical notation based on an extension of OMT (Object Modelling Technique [15]) together with natural language and sample code. This gives a very good intuitive picture of the patterns, but is not sufficiently precise to allow a designer to demonstrate conclusively that a particular problem matches a particular pattern or that a proposed solution is consistent with a particular pattern. The notation also makes it difficult to be certain that the patterns used are meaningful and contain no inconsistencies.

A formal model of patterns can help to alleviate these problems. One existing approach to this [5] represents patterns as formulae in LePUS, a language defined as a fragment of higher order monadic logic [6]. A second [12] formalises the temporal behaviour of patterns using the DisCo specification method, which is based on the Temporal Logic of Actions [11]. Another [4, 3] specifies the essential elements of GoF patterns using RSL (the RAISE Specification Language; [13]).

Our approach is based on the last of these, though it significantly extends the scope of the model used therein in several ways. First, we generalise the model so that it describes an arbitrary object-oriented design and not just the patterns. Second, we formally specify how to match a design against a pattern. And third we include in our model specifications of the behavioural properties of the design, specifically the actions that are to be performed by the methods, which was omitted from the model described in [4, 3]. In this way, we can formally model all the components of an generic object-oriented design and also formally check that a given subset of that design matches a given pattern.

We begin by giving an overview of the extended OMT notation and of our formal model of a generic object-oriented design based upon it in Section 2. Then we discuss how to formally link a design with a pattern in Section 3. Section 4 then shows how the properties of individual patterns are specified in our model, using the State pattern from the GoF catalogue [9] as an example, and Section 5 gives an example of the whole process of specifying a design and verifying that it matches a pattern, again using an example of a design based on the State pattern. We conclude with a summary of our work and an indication of future work we plan in this field.

2 A Formal Model of Object-Oriented Design

We base our formal model on the extended OMT [15] notation which is used in [9] to describe the structure of GoF patterns. An example of this is shown in Figure 1, which represents the structure of the State pattern.

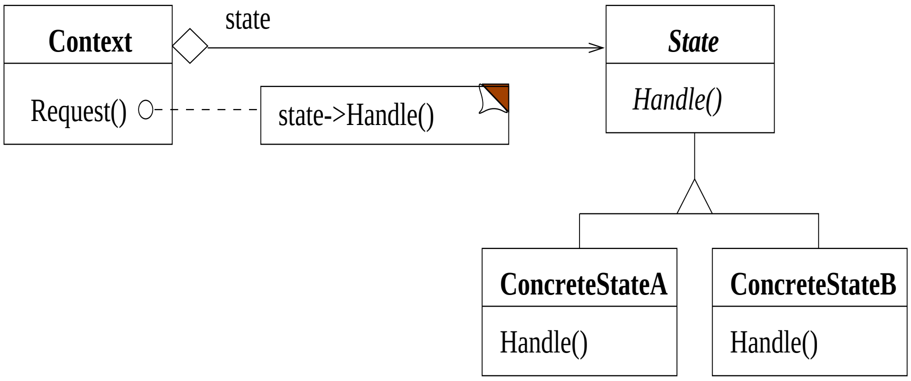

Fig. 1. State Pattern Structure

We begin by giving an informal overview of this notation in Section 2.1, then introduce our formal model in Section 2.2.

2.1 An Overview of Extended OMT Notation

In the extended OMT notation, a design consists essentially of a collection of classes and a collection of relations linking the classes. Each class is depicted as a rectangle containing the name of the class, the signatures (i.e. names and parameters) of the operations or methods which objects of the class can perform, and the state variables or instance variables which represent the internal data stored by instances of the class. Every class in a design has a unique name.

Classes and methods are designated as *abstract* or *concrete* by writing their name in italic or upright script respectively in the OMT diagram. No instances (objects) may be created from an abstract class, and an abstract method cannot be executed (often because the method is only completely defined in subclasses).

Concrete methods, which can be executed, may additionally have *annotations* in the OMT diagram which indicate what actions the method should perform. These annotations appear within rectangles with a “folded” corner which are attached to a method within the class desription rectangle by a dashed line ending in a small circle.

Thus, for example, the structure in Figure 1 indicates that the class Context in the State pattern is a concrete class which contains a concrete method called Request, while the class State is an abstract class which contains an abstract method called Handle. In addition, the annotation attached to the Request method in the Context class indicates that the action of this method is to invoke the Handle method on the variable called state.

Relations specify connections or communications between classes and are represented as lines linking classes in the OMT diagram. Four different types of relations are used – *inheritance*, *aggregation*, *association* and *instantiation* – and these are distinguished in the diagram using different types of lines. Inheritance relations have a triangle in the middle of the line whose point and base indicate respectively the superclass and subclasses. Thus, in the State pattern the State class is a superclass of the ConcreteState classes. Aggregation relations, which signify that one object is a constituent part or a sub-object of another, are drawn as solid lines with a diamond on one end and an arrowhead on the other, the arrowhead pointing towards the class of the sub-object. The relation between the Context class and the State class in Figure 1 is thus an aggregation relation which indicates that the Context class consists of a sub-object of the State class. Association relations are also shown as solid lines with an arrowhead on one end but they are unmarked at the other end, while instantiation relations are shown as dashed lines with an arrowhead on one end. An association relation indicates that one class communicates with another, and an instantiation relation indicates that a class creates objects belonging to another class. The arrowhead indicates the direction of the communication or the class being instantiated respectively.

Association and aggregation relations also have an associated *arity*, which may be one or many according to whether each object of one class communicates with or is composed of a single object or a collection of objects of the other class. Relations of arity many are indicated by adding a solid black circle to the front of the arrowhead.

2.2 The Formal Model

From the patterns and examples in [9] it can be seen that the various actions that can appear as annotations to methods in a design basically correspond to the different types of relations that can link the classes, except that at the level of the actions there is no distinction between aggregation and association relations. Thus, in our formal model we define three types of actions – invocation, instantiation, and self or super invocation – which we use to model these actions.

An invocation represents an interaction that corresponds to an association or aggregation relation: objects of one class request objects of another class to perform some action by executing some method. Some variable (generally the "name" of the relation) in the first class represents the object that receives the request, while the request itself consists of the name of the method which should be executed together with appropriate parameters for that method. In the RSL specification variables are represented by the type 'Variable_Name' and the parameters of a request by the type 'Actual_Parameters', which is basically just a list of variables which does not include the reserved variable name **super** which is used exclusively in super invocations (described below). Then the request as a whole is modelled using the type 'Actual_Signature' and the whole invocation by the type 'Invocation'.

Invocation ::
  call_vble : Variable_Name call_sig : Actual_Signature,
Actual_Signature ::
  meth_name : Method_Name a_params : Actual_Parameters,
Actual_Parameters = Wf_Variable_Name*

An instantiation of course represents an interaction which corresponds to an instantiation relation: one class requests another class to create a new object. In most object-oriented programming languages there are essentially two ways in which this sort of object creation can be performed. First, the class might create a "default" instance of itself (for example in Smalltalk by using the basic creation method *new* which is available in every class) and then set the state variables of this instance appropriately using other methods. Or second, the class may have other local creation methods which create customised instances directly using parameters to the methods (as, for example, in the parameterised method *new:* in Smalltalk which uses its parameter to additionally set the state of the instance it creates). We cover both of these situations in our model by defining an instantiation (the type 'Instantiation') to consist of the name of the class to be instantiated together with a possibly empty list of parameters.

$$\begin{array}{l}\mathrm{Instantiation} :: \\ \quad \mathrm{class\_name} : \mathrm{Class\_Name} \quad \mathrm{a\_params} : \mathrm{Actual\_Parameters}\end{array}$$

Self and super invocations are analogous to invocations except that the invocation is to the same class or to a superclass respectively. Super invocations therefore correspond in some sense to inheritance relations, but there is no such correspondence with relations in the case of self invocations because relations between a class and itself are generally not shown explicitly in an OMT diagram. In our formal model we use the type 'Invocation' to model both self and super invocations, except that in these cases the call variable of the invocation is the specific variable name 'self' or 'super' respectively.

$$\mathrm{self}, \mathrm{super} : \mathrm{Variable\_Name}$$

In general, an annotation can contain one or more instantiations or invocations, the order of which is generally important. We model this as a list of *requests*, where a request can be either an invocation or an instantiation. Annotations can also indicate assignments to variables, including both state variables and local (dummy) variables. Two forms of assignment are used: one where the results of some request are assigned to variables, generally for use in a later request, and the second where the parameters of the method are assigned to variables, generally state variables. These assignments are modelled using the type 'Variable_Change', which maps sets of variables to either requests or sets of variables. These correspond to the two forms of assignment described above. The predicate defining the subtype here ensures that the empty set is not in the domain of the map and that no two sets in the domain have variables in common[2].

$$\mathrm{Variables} = \mathrm{Wf\_Variable\_Name}\text{-}\mathbf{set},$$

[2] Although this latter condition might at first sight seem to be too restrictive, our model is an abstract one in which we only model the final value of each particular variable.

Request = Invocation | Instantiation | _,
Request_or_Var = Request | Variables,
Variable_Change =
{| m : Wf_Vble_Name-**set** $\overrightarrow{m}$ Request_or_Var •
is_wf_vchange(m)
|}

The requests and the variable assignments constitute the *body* of a concrete or *implemented* method. The actions performed by abstract or *defined* methods are unspecified, however, so these methods basically have no body. And some patterns (e.g. the Composite pattern in [9]), and hence of course designs, can include methods which are defined in a superclass but which should not be implemented in all subclasses and indeed do not make sense in some subclasses. Such methods, which we call *error* methods, also have no body. Our formal definition of the body of a method then takes the form of a variant type which defines each of the three different types of method, the bodies of defined and error methods being represented simply as the constants of the same names.

Method_Body ==
defined |
error |
implemented
(variable_change : Variable_Change, request_list : Request*)

The other important components of a method are its result, which we model as a set of variables, its formal parameters, which is a list of parameters, each of which is a variable (which cannot be 'self' or 'super'; the type 'Wf_Vble_Name') with optionally the name of a class indicating the type of that variable, and its name. The well-formedness condition 'is_wf_formal_parameters' on the formal parameters ensures that all the variables representing the formal parameters are distinct (so that they can be distinguished in the body of the method). There are also consistency conditions on the components of the method, for example that every set of variables to which the result of an instantiation is assigned can contain only one variable, and these are similarly embodied in the function 'is_wf_method' and hence in the definition of a *well-formed* method.

The method names are included in the form of a map from method names to well-formed methods since the names of the methods in a particular class must all be different. The constraint 'is_wf_class_method' simply states that certain reserved method names cannot be used.

Method ::
f_params : Wf_Formal_Parameters
meth_res : Result
body : Method_Body,
Result = Variables,
Parameter ==

$$
\begin{array}{l}
\quad \mathrm{var(Wf\_Vble\_Name)} \mid \\
\quad \mathrm{paramTyped(paramName : Wf\_Vble\_Name, className : Class\_Name)}, \\
\mathrm{Wf\_Formal\_Parameters} = \\
\quad \{|\ \mathrm{p : Parameter^{*}} \bullet \mathrm{is\_wf\_formal\_parameters(p)}\ |\} \\
\mathrm{Wf\_Method} = \{|\ \mathrm{m : Method} \bullet \mathrm{is\_wf\_method(m)}\ |\}, \\
\mathrm{Map\_Methods} = \mathrm{Method\_Name} \underset{m}{\overset{}{\rightarrow}} \mathrm{Wf\_Method}, \\
\mathrm{Class\_Method} = \{|\ \mathrm{m : Map\_Methods} \bullet \mathrm{is\_wf\_class\_method(m)}\ |\}
\end{array}
$$

The state of a class is similarly defined as a set of variables, which also may not include the reserved variables 'self' and 'super. This, together with the methods and the class type, which is simply either of the two values 'abstract' or 'concrete', then forms all the important components of a class definition except its name. We again include the class names using a map from class names to *well-formed* classes because the names of all classes in a design must be distinct. The well-formedness condition on a class requires that state variables cannot be used as formal parameters to methods.

$$
\begin{array}{l}
\mathrm{State} = \mathrm{Wf\_Vble\_Name}\text{-}\mathbf{set}, \\
\mathrm{Class\_Type} == \mathrm{abstract} \mid \mathrm{concrete}, \\
\mathrm{Design\_Class} :: \\
\quad \mathrm{class\_state : State} \\
\quad \mathrm{class\_methods : Class\_Method} \\
\quad \mathrm{class\_type : Class\_Type}, \\
\mathrm{Wf\_Class} = \{|\ \mathrm{c : Design\_Class} \bullet \mathrm{is\_wf\_class(c)}\ |\}, \\
\mathrm{Classes} = \mathrm{Class\_Name} \underset{m}{\overset{}{\rightarrow}} \mathrm{Wf\_Class}
\end{array}
$$

A relation is basically determined by the classes it links and its type, which may be inheritance, association, aggregation, or instantiation. All relations except inheritance relations are binary, linking a single *source* class to a single *sink* class. We in fact also model inheritance relations as binary relations by considering the case in which a class has several subclasses as many inheritance relations, one linking the superclass to each individual subclass. Thus, our basic definition of a relation is embodied in the record type 'Design_Relation'.

In the case of instantiation and inheritance relations, there can be at most one such relation between any pair of classes. The type together with the source and sink classes is thus sufficient to identify the relation uniquely. However, it is possible to have more than one association or aggregation relation between the same two classes, and furthermore the arity of these relations can be indicated, at least to the extent that it is either one or many. We therefore introduce the type 'Card' to represent the arity and use the names of the relations to distinguish between them. In this way, for example, the aggregation relation betwen the Context and State classes in the State pattern (see Figure 1) has arity one-one and is identified uniquely by its name state.

The well-formedness condition 'wf_relation' states that instantiation relations are not explicitly shown between a class and itself and that there cannot be inheritance relations between a class and itself.

Card == one | many,
Ref ::
relation_name : Wf_Vble_Name
sink_card : Card
source_card : Card,
Relation_Type ==
inheritance |
association(as_ref : Ref) |
aggregation(ag_ref : Ref) |
instantiation,
Design_Relation ::
relation_type : Relation_Type
source_class : Class_Name
sink_class : Class_Name,
Wf_Relation = {| r : Design_Relation • wf_relation(r) |}

An object-oriented design, which is represented in our model by the type 'Design_Structure', then simply consists of a collection of classes and a collection of relations, together with appropriate consistency conditions (for example that there are no circularities in inheritance relations, that an abstract class cannot be the sink of an instantiation relation because creating instances of abstract classes is not allowed, that an abstract class must have subclasses, etc. Full details of these consistency conditions can be found in [8]).

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

3 Matching Designs to Patterns

We now go on to explain how to link our model of a design to the design patterns in such a way that it is possible to determine whether or not the two match.

We make this link using a *renaming map*, which associates the names of entities (classes, methods, state variables and parameters) in the design with the names of corresponding entities in the pattern. Thus, the correspondences between state variables and between parameters are modelled using the type 'VariableRenaming', which simply maps variables in the design to variables in the pattern. The type 'Method_and_Parameter_Renaming' relates methods in the design to methods in the pattern. It consists of two parts: the first simply defines the correspondence between the names of the methods and the second relates their parameters. This nested structure is necessary because two different methods may have parameters with the same name.

The renaming of a class has a similarly nested structure, the type 'Class-Renaming' consisting of the name of the class in the pattern together with one renaming map for the methods in the class and another for the state variables. However, in this case it is possible for a single class in the design to play several *roles* in the pattern (for instance, in the example illustrating the Command pattern in [9] the class Application in the design plays both the Client and the Receiver roles in the pattern). We therefore map each design class to a set of class renamings in the renaming map, and the full renaming map is represented by the type 'Renaming'. The well-formedness condition requires that no design class can have an empty set of renamings and that the renamings of any one design class must all refer to different pattern classes.

$$
\begin{array}{l}
\text{VariableRenaming} = \text{Variable\_Name} \underset{m}{\rightarrow} \text{Variable\_Name},\\
\text{Method\_and\_Parameter\_Renaming} = \text{Method\_Name} \underset{m}{\rightarrow} \text{Method\_Renaming},\\
\text{Method\_Renaming} ::\\
\quad \text{method\_name} : \text{Method\_Name} \;\; \text{parameterRenaming} : \text{VariableRenaming},\\
\text{ClassRenaming} ::\\
\quad \text{classname} : \text{Class\_Name}\\
\quad \text{methodRenaming} : \text{Method\_and\_Parameter\_Renaming}\\
\quad \text{varRenaming} : \text{VariableRenaming},\\
\text{Renaming} = \text{Class\_Name} \underset{m}{\rightarrow} \text{ClassRenaming-}\mathbf{set},\\
\text{Wf\_Renaming} = \{|\ \text{r} : \text{Renaming} \bullet \text{is\_wf\_Renaming(r)}\ |\}
\end{array}
$$

Finally, we link the design with the renaming map through the type 'Design_Renaming'. Its well-formedness condition is quite complicated so we refer the reader to [8] for the details.

$$
\begin{array}{l}
\text{Design\_Renaming} = \text{Wf\_Design\_Structure} \times \text{Wf\_Renaming},\\
\text{Wf\_Design\_Renaming} =\\
\quad \{|\ \text{pr} : \text{Design\_Renaming} \bullet \text{is\_wf\_design\_renaming(pr)}\ |\}
\end{array}
$$

4 Specifying the Properties of the Patterns

In order to check whether a particular (subset of a) design matches a particular pattern, we formally specify functions which embody all the properties that the entities in the pattern must exhibit, then we require that every entity in the design which has a renaming under the renaming map to an entity in the pattern satisfies the properties of that entity in the pattern. We illustrate how the properties of the patterns are specified by considering the specification of the State pattern in [9] (see Figure 1).

The structure of the State pattern comprises a single hierarchy of classes rooted at the State class, together with a single Context class. The Context class basically defines a common interface which clients can use to interact with the various ConcreteState subclasses, and essentially it simply forwards requests appropriately via its state variable. This is represented by the single aggregation relation between these classes in the pattern structure.

We define the function 'hierarchy' to specify the first of these properties. This is a generic function which checks that a hierarchy of classes in the design has as its root a class which plays a given role in the pattern and which is unique in the design, has leaf classes which play any of a given set of roles in the pattern[3], and has no classes which play roles from a given set of roles (in this case Context and Client). The specific property of the State pattern that we require is then embodied in the function 'State_hierarchy' which simply instantiates the function 'hierarchy' with the required roles. We omit the specification of the function 'hierarchy', which is rather long, for brevity and refer the reader to [8] for the details.

$$
\begin{array}{l}
\text{State\_hierarchy} : \text{Wf\_Design\_Renaming} \to \textbf{Bool} \\
\text{State\_hierarchy(dr)} \equiv \\
\quad \text{hierarchy(State}, \{\text{ConcreteState}\}, \{\text{Context, Client}\}, \text{dr)}
\end{array}
$$

Another property of the State pattern is that there is a single class which plays the Context role, and this is a concrete class. This is specified using the functions 'exists_one' and 'is_concrete_class' from [8]. The function 'exists_one' checks that a single class in the design plays a given role in the pattern, and the function 'is_concrete_class' checks that all classes that play a given role are concrete. Again, the specifications of the required property of the State pattern, which are represented by the functions 'exists_one_Context' and 'is_concrete_Context', are obtained by instantiating these functions with the appropriate roles from the State pattern.

$$
\begin{array}{l}
\text{exists\_one} : \text{Class\_Name} \times \text{Wf\_Design\_Renaming} \to \textbf{Bool} \\
\text{exists\_one(cp, (ds, r))} \equiv \\
\quad (\exists!\ \text{cd} : \text{Class\_Name} \bullet \text{renaming\_class\_name(cd, cp, r)}), \\
\\
\text{is\_concrete\_class} : \text{Class\_Name} \times \text{Wf\_Design\_Renaming} \to \textbf{Bool} \\
\text{is\_concrete\_class(cp, ((dsc, dsr), r))} \equiv \\
\quad (\\
\qquad \forall\ \text{cd} : \text{Class\_Name} \bullet \\
\qquad\quad \text{renaming\_class\_name(cd, cp, r)} \Rightarrow \text{is\_concrete\_class(dsc(cd))} \\
\quad), \\
\\
\text{exists\_one\_Context} : \text{Wf\_Design\_Renaming} \to \textbf{Bool} \\
\text{exists\_one\_Context(dr)} \equiv \text{exists\_one(Context, dr)}, \\
\\
\text{is\_concrete\_Context} : \text{Wf\_Design\_Renaming} \to \textbf{Bool} \\
\text{is\_concrete\_Context(dr)} \equiv \text{is\_concrete\_class(Context, dr)}
\end{array}
$$

Other properties of the State pattern are specified similarly. These include, for example, that the class which plays the Context role contains a single state

[3] In this case there is only one class in this set, namely ConcreteState, but it is possible to have more than one class as, for example, in the Command pattern in [9].

variable which plays the state role and that it also contains at least one method which plays the Request role, all such methods being implemented and containing an invocation to the state variable of a method which plays the Handle role. Together with the properties specified above, these lead us to the definition of the function 'is_state_pattern' which embodies all the essential properties of the elements of the State pattern. Again, full details can be found in [8].

$$\begin{array}{l} \text{is\_state\_pattern} : \text{Wf\_Design\_Renaming} \to \mathbf{Bool} \\ \text{is\_state\_pattern}(\text{dr}) \equiv \\ \quad \text{State\_hierarchy}(\text{dr}) \wedge \\ \quad \text{exists\_one\_Context}(\text{dr}) \wedge \\ \quad \text{is\_concrete\_Context}(\text{dr}) \wedge \ldots \end{array}$$

In fact we have developed specifications of this form for all of the GoF structural patterns and nine of the eleven GoF behavioural patterns and we are currently working on the creational patterns and the remaining behavioural patterns. Full details of the specifications of the completed patterns can be found in [7] and [14] respectively.

5 An Example: Checking an Instantiation of the State Pattern

In this section we give an example of how an object-oriented design is represented in our model and how we relate this to a pattern using the renaming map. As the basis for this, we use the example which is used in [9] to illustrate the motivation and sample code of the State pattern.

This example is a model of a TCP network connection. This connection can be in one of several states – closed, established, listening, etc. – and different operations can be applied to these states to manipulate the connection.

The OMT-extended diagram of this design, where we only include classes representing the three states mentioned above, is shown in Figure 2.

We give only a representative sample of the specification of the design here, defining only the class TCPConnection and the relations in detail. The complete specification can be found in [8].

We begin by defining RSL constants which represent the names of the classes, methods, state variables and parameters which are used in the design. Those used in the class TCPConnection and the relations are:

TCPConnection : Class_Name,
TCPState : Class_Name,
TCPEstablished : Class_Name,
Client : Class_Name,
ActiveOpen : Method_Name,
PassiveOpen : Method_Name,
Close : Method_Name,

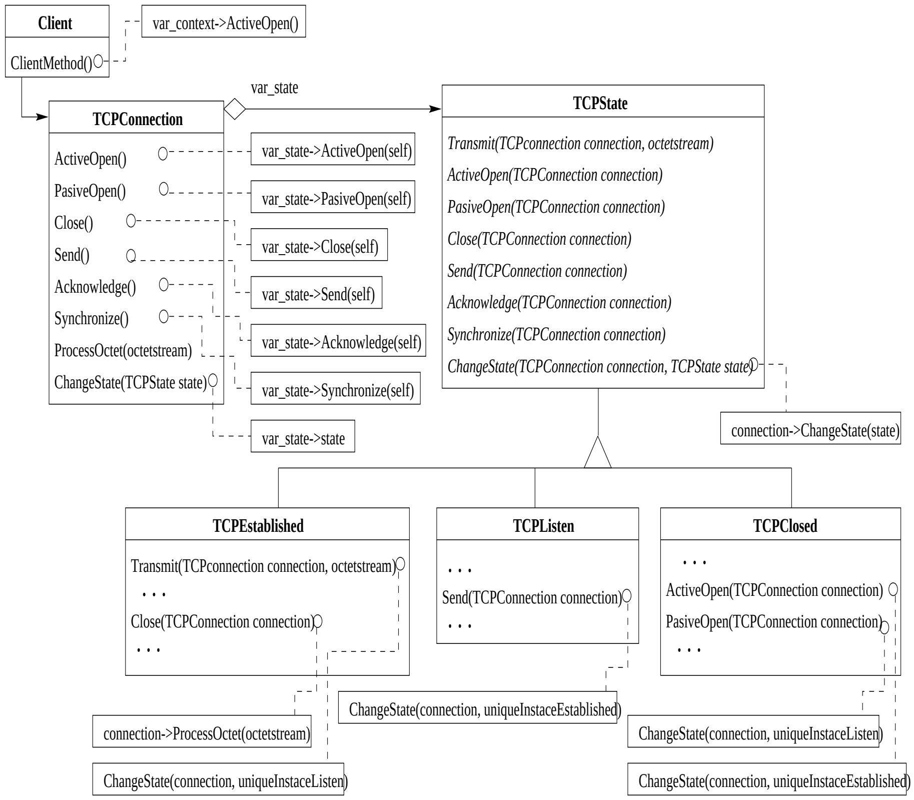

Fig. 2. Design for a TCP Network Connection

Send : Method_Name,
Acknowledge : Method_Name,
Synchronize : Method_Name,
ProcessOctet : Method_Name,
ChangeState : Method_Name,
var_state : Variable_Name,
var_context : Variable_Name,
state : Variable_Name,
octetstream : Variable_Name

Next we define the other parts of the methods – their bodies, results and parameters – and the collection of all methods in the class.

Although there are eight methods in the class TCPConnection, the forms of ActiveOpen, PassiveOpen, Close, Send, Acknowledge and Synchronize are essentially the same: each has no parameters, returns no result, causes no variable changes, and has a body which consists of a single invocation to the var_state variable of the corresponding method (i.e. the method with the same name) in the

TCPState class, the parameter of each invocation being self. The specifications of these six methods are therefore all identical up to the names involved. Therefore we only show the specification of one of them, ActiveOpen, here together with the specifications of ProcessOctet and ChangeState.

We first define constants representing the bodies of the methods.

Because there are many cases in which different methods in the design have essentially the same structure as, for example, with the six methods described above, we introduce generic parameterised functions to represent these common structures and then define the individual methods in terms of these. An additional advantage of this approach is that the generic functions are likely to be reusable across many different designs.

We therefore begin by defining the function 'one_inv_meth_body' which descibes in parameterised form the bodies of the first six methods in TCPConnection. This function then basically describes the body of any method in the design which consists of a single invocation to a given variable of a given method, the invocation having a single given parameter and the method involving no variable changes. Note that the invoked method and its parameter form an actual signature (see Section 2.2) in the specification. Then the body of the ActiveOpen method is represented by a constant, 'meth_body_AOctn', which is constructed by instantiating the function 'one_inv_meth_body' appropriately, in this case with the values var_state, ActiveOpen and self.

```
one_inv_meth_body :
   Variable_Name × Method_Name × Wf_Variable_Name →
      Method_Body
one_inv_meth_body(v, m, p) ≡
   implemented
      ([ ], ⟨mk_Invocation(v, mk_Actual_Signature(m, ⟨p⟩))⟩),

meth_body_AOctn : Method_Body =
   one_inv_meth_body(var_state, ActiveOpen, self)
```

The ProcessOctet and ChangeState methods are treated similarly. The first of these, like several other methods in the design, has no explicit body, so we introduce a generic constant 'empty_method_body' to represent the body of all such methods. The second simply assigns its parameter to a particular state variable, so its body is empty apart from a single variable change which represents this assignment. This type of body is modelled generically using the function 'assign_param_meth_body' and the body of the ChangeState method is again obtained by instantiating this function appropriately, in this case with the variables var_state and state.

```
empty_method_body : Method_Body = implemented([ ], ⟨⟩),

assign_param_meth_body :
   Variable_Name × Wf_Variable_Name → Method_Body
```

$\text{assign\_param\_meth\_body}(v, p) \equiv$
$\quad \text{implemented}([\,\{v\} \mapsto \text{Request\_or\_Var\_from\_Variable}(p)\,], \langle\rangle),$

$\text{meth\_body\_ChgSt} : \text{Method\_Body} =$
$\quad \text{assign\_param\_meth\_body}(\text{var\_state}, \text{state})$

Having defined the bodies of the methods, we now proceed to define the methods as a whole.

Again there are similarities in the structure of the methods: the first six methods in the TCPConnection class all have no parameters and no result, though they have different bodies; the method ProcessOctet has a single untyped parameter and no result; and the method ChangeState, in common with the majority of the methods in the other classes, has a single typed parameter and no result. We therefore introduce the two generic functions 'method_with_body' and 'method_with_body_param' to describe each of these forms in an appropriately parameterised way.

$\text{method\_with\_body} : \text{Method\_Body} \to \text{Method}$
$\text{method\_with\_body}(b) \equiv \text{mk\_Method}(\langle\rangle, \{\}, b),$

$\text{method\_with\_body\_param} :$
$\quad \text{Method\_Body} \times \text{Wf\_Formal\_Parameters} \to \text{Method}$
$\text{method\_with\_body\_param}(b, p) \equiv \text{mk\_Method}(p, \{\}, b)$

Then the specifications of the individual methods in the class TCPConnection are obtained by appropriately instantiating these generic functions, and the collection of all methods in the class, which is represented by the RSL constant 'Ctn_Class_Methods', is formed by constructing a map from each method name to the appropriate method. However, the methods constructed by these generic functions do not necessarily satisfy the well-formedness condition 'is_wf_method' (the result type of the functions is 'Method' not 'Wf_Method'). Similarly, the collection of methods must satisfy the well-formedness condition 'is_wf_class_method'. We must therefore check that these conditions are satisfied in order to be certain that the design is well-formed and the definition below is correctly typed.

$\text{Ctn\_Class\_Methods} : \text{Class\_Method} =$
$\quad [$
$\qquad \text{ActiveOpen} \mapsto \text{method\_with\_body}(\text{meth\_body\_AOctn}),$
$\qquad \text{PassiveOpen} \mapsto \text{method\_with\_body}(\text{meth\_body\_POctn}),$
$\qquad \text{Close} \mapsto \text{method\_with\_body}(\text{meth\_body\_Cctn}),$
$\qquad \text{Send} \mapsto \text{method\_with\_body}(\text{meth\_body\_Sctn}),$
$\qquad \text{Acknowledge} \mapsto \text{method\_with\_body}(\text{meth\_body\_Akctn}),$
$\qquad \text{Synchronize} \mapsto \text{method\_with\_body}(\text{meth\_body\_Syctn}),$
$\qquad \text{ProcessOctet} \mapsto$
$\qquad\quad \text{method\_with\_body\_param}$
$\qquad\qquad (\text{empty\_method\_body}, \langle \text{var}(\text{octetstream})\rangle),$

ChangeState $\mapsto$
method_with_body_param
(meth_body_ChgSt, $\langle$paramTyped(state, TCPState)$\rangle$)
]

The sets of methods for the other classes are defined similarly.

The next step is to incorporate the definitions of the methods in the class into a definition of the class as a whole. For this we need to additionally define the class state and its type.

In the design, the class TCPConnection has a single state variable var_state and is a concrete class. The specification of this class, which we must again check for well-formedness (the function 'is_wf_class') is therefore:

Ctn_Class : Wf_Class =
mk_Design_Class({var_state}, Ctn_Class_Methods, concrete)

Next we turn to the relations in the design. There are in fact one aggregation relation, one association relation and three inheritance relations (one between TCPState and each of its subclasses) included. Here we only show the specification of one of the inheritance relations because the others are entirely analogous up to the names of the classes involved.

The aggregation and association relations are both one-one, so their specifications (the constants 'agg_rel' and 'ass_rel' respectively) are similar apart from their types and the names of the classes and variables involved. The inheritance relations are simply specified as inheritance relations between the appropriate pair of classes. Each must of course be shown to satisfy the well-formedness condition 'wf_relation'.

agg_rel : Wf_Relation =
mk_Design_Relation
(
aggregation(mk_Ref(var_state, one, one)),
TCPConnection,
TCPState
),

ass_rel : Wf_Relation =
mk_Design_Relation
(
association(mk_Ref(var_context, one, one)),
Client,
TCPConnection
),

inh1_rel : Wf_Relation =
mk_Design_Relation(inheritance, TCPState, TCPEstablished)

The other classes and relations in the design are specified in a similar way, then the specification of the design as a whole is obtained by combining them together. To do this, we construct a map which associates each class name in the design with its definition and a set containing all the relations in the design. The design as a whole is then represented by the pair constructed from these two components. Checking the remaining well-formedness conditions then ensures that the design as a whole is well-formed.

Class_Map : Classes =
  [
    Client ↦ Cli_Class,
    TCPConnection ↦ Ctn_Class,
    TCPState ↦ Sta_Class,
    TCPEstablished ↦ Est_Class,
    TCPListen ↦ Lis_Class,
    TCPClosed ↦ Clo_Class
  ],

Rel_set : Wf_Relation-**set** =
  {agg_rel, inh1_rel, inh2_rel, inh3_rel, ass_rel},

State_DS : Wf_Design_Structure = (Class_Map, Rel_set)

This completes the specification of the design and we must now link the design to the pattern by defining a renaming mapping from the names of the classes, methods, state variables and parameters in the design to the corresponding entities which represent their roles in the pattern (see Figure 1). Again we concentrate on the class TCPConnection here.

The class TCPConnection corresponds to the Context class in the pattern, and the first six methods (ActiveOpen, PassiveOpen, Close, Send, Acknowledge, and Synchronize) in TCPConnection all correspond to the Request operation in the pattern. Thus, in this example there are many elements of the design which play a single role in the pattern.

Since all the above methods play the same role in the pattern and have no explicit parameters, they all have the same renaming. We therefore simplify our specification by introducing a constant 'Ctn_req_mtd' which represents this renaming. Then we construct a renaming map 'Ctn_mtd' for the methods (and their parameters) by mapping each of the methods at the design level to this constant.

Note that the methods ProcessOctet and ChangeState have no counterparts in the pattern so are simply omitted from the method renaming map.

Ctn_req_mtd : Method_Renaming = mk_Method_Renaming(S.Request, [ ]),

Ctn_mtd : Method_and_Parameter_Renaming =
  [

$$
\begin{array}{l}
\quad \text{ActiveOpen} \mapsto \text{Ctn\_req\_mtd}, \\
\quad \text{PassiveOpen} \mapsto \text{Ctn\_req\_mtd}, \\
\quad \text{Close} \mapsto \text{Ctn\_req\_mtd}, \\
\quad \text{Send} \mapsto \text{Ctn\_req\_mtd}, \\
\quad \text{Acknowledge} \mapsto \text{Ctn\_req\_mtd}, \\
\quad \text{Synchronize} \mapsto \text{Ctn\_req\_mtd} \\
]
\end{array}
$$

We similarly build a variable renaming map to associate the state variables in the TCPConnection class with those in the Context class. This is then combined with the method renaming to yield the renaming for the whole class.

$$
\begin{array}{l}
\text{Ctn\_vbles} : \text{VariableRenaming} = [\,\text{var\_state} \mapsto \text{S.state}\,], \\
\\
\text{Ctn\_Class\_Renaming} : \text{ClassRenaming} = \\
\quad \text{mk\_ClassRenaming(S.Context, Ctn\_mtd, Ctn\_vbles)}
\end{array}
$$

We follow the same procedure for the other classes in the design to obtain the renaming for the whole design, which simply associates the names of the classes in the design with the appropriate class renaming. Note that each design class plays a single role in the pattern so there is only a single class renaming for each design class. Again, we must check that the well-formedness condition ‘is_wf_Renaming’ is satisfied.

$$
\begin{array}{l}
\text{State\_Renaming} : \text{Wf\_Renaming} = \\
\quad [\\
\quad\quad \text{TCPConnection} \mapsto \{\text{Ctn\_Class\_Renaming}\}, \\
\quad\quad \text{TCPState} \mapsto \{\text{Sta\_Class\_Renaming}\}, \\
\quad\quad \text{TCPEstablished} \mapsto \{\text{Con\_Class\_Renaming}\}, \\
\quad\quad \text{TCPListen} \mapsto \{\text{Con\_Class\_Renaming}\}, \\
\quad\quad \text{TCPClosed} \mapsto \{\text{Con\_Class\_Renaming}\}, \\
\quad\quad \text{Client} \mapsto \{\text{Cli\_Class\_Renaming}\} \\
\quad]
\end{array}
$$

The final step is to combine the specifications of the design and the renaming and to check that these together satisfy the well-formedness condition ‘is_wf_design_renaming’.

$$
\text{State\_Pat\_Ren} : \text{Wf\_Design\_Renaming} = (\text{State\_DS}, \text{State\_Renaming})
$$

This value is then used as input to the function ‘is_state_pattern’ defined in Section 4 to check whether or not the TCP network connection design is an instance of the State pattern.

6 Conclusions

We have described a formal model of a generic object-oriented design based on the extended OMT notation and we have shown how a design in this model can be linked to a GoF pattern using the renaming map. We have furthermore shown how the specific properties of individual GoF patterns can be specified in this model, and we have illustrated using an example design how the specifications can be used to determine whether or not a given design matches a given pattern. This allows designers to be sure, as well as to demonstrate to others that they are using the patterns correctly and consistently.

Although we have limited our attention to GoF patterns in our current work, we believe that our basic model is in fact sufficiently general that it could be applied in a similar way to give formal descriptions of other design patterns. We have also limited ourselves so far to matching a single pattern at a time. However, we have considered the possibility of extending the model to deal with multiple patterns and we believe that this could easily be done by simply redefining the renaming map slightly and indeed we plan to investigate this extension in future work. We also plan to continue our work by investigating the provision of tool support based on our general model and pattern specifications.

References

1. Brad Appleton. Patterns and Software: Essential Concepts and Terminology. http://www.enteract.com/~bradapp, November 1997.
2. Kent Beck, James O. Coplien, Ron Crocker, Lutz Dominick, Gerard Meszaros, Frances Paulisch, and John Vlissides. Industrial Experience with Design Patterns. Technical report, First Class Software, AT&T, Motorola Inc, Siemens AG, Bell Northern Research, Siemens AG, and IBM Research. http://www1.bell-labs.com/user/cope/Patterns/ICSE96/icse.html.
3. Alejandra Cechich and Richard Moore. A Formal Specification of GoF Design Patterns. Technical Report 151, UNU/IIST, P.O.Box 3058, Macau, January 1999.
4. S. Alejandra Cechich and Richard Moore. A Formal Specification of GoF Design Patterns. In *Proceedings of the Asia Pacific Software Engineering Conference: APSEC'99*, Takamatsu, Japan, December 1999.
5. A. Eden, J. Gil, Y. Hirshfeld, and A. Yehudai. Towards a Mathematical Foundation for Design Patterns. http://www.math.tau.ac.il/~eden/bibliography.html.
6. A. Eden, Y. Hirshfeld, and A. Yehudai. LePUS - A Declarative Pattern Specification Language. http://www.math.tau.ac.il/~eden/bibliography.html.
7. Andres Flores and Richard Moore. GoF Structural Patterns: A Formal Specification. Technical Report 207, UNU/IIST, P.O. Box 3058, Macau, August 2000.
8. Andres Flores, Luis Reynoso, and Richard Moore. A Formal Model of Object-Oriented Design and GoF Design Patterns. Technical Report 200, UNU/IIST, P.O. Box 3058, Macau, July 2000.
9. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, 1995.
10. Ralph Johnson. Design Patterns in the Standard Java Libraries. In *Proceedings of the Asia Pacific Software Engineering Conference: Keynote Materials, Tutorial Notes*, pages 66–101, 1999.

11. L. Lamport. The Temporal Logic of Actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, May 1994.
12. Tommi Mikkonen. Formalizing Design Patterns. In *Proceedings of the International Conference on Software Engineering ICSE'98*, pages 115–124. IEEE Computer Society Press, 1998.
13. The RAISE Language Group. *The RAISE Specification Language*. BCS Practitioner Series. Prentice Hall, 1992.
14. Luis Reynoso and Richard Moore. GoF Behavioural Patterns: A Formal Specification. Technical Report 201, UNU/IIST, P.O. Box 3058, Macau, May 2000.
15. J. Rumbaugh. *Object-Oriented Modeling and Design*. Prentice Hall, 1991.