The United Nations University

UNU/IIST

International Institute for Software Technology

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

Andres Flores, Luis Reynoso and Richard Moore

July 2000

UNU/IIST Report No. 200

UNU/IIST and UNU/IIST Reports

UNU/IIST (United Nations University International Institute for Software Technology) is a Research and Training Centre of the United Nations University (UNU). It is based in Macau, and was founded in 1991. It started operations in July 1992. UNU/IIST is jointly funded by the Governor of Macau and the governments of the People's Republic of China and Portugal through a contribution to the UNU Endownment Fund. As well as providing two-thirds of the endownment fund, the Macau authorities also supply UNU/IIST with its office premises and furniture and subsidise fellow accommodation.

The mission of UNU/IIST is to assist developing countries in the application and development of software technology.

UNU/IIST contributes through its programmatic activities:

1. Advanced development projects, in which software techniques supported by tools are applied,
2. Research projects, in which new techniques for software development are investigated,
3. Curriculum development projects, in which courses of software technology for universities in developing countries are developed,
4. University development projects, which complement the curriculum development projects by aiming to strengthen all aspects of computer science teaching in universities in developing countries,
5. Courses, which typically teach advanced software development techniques,
6. Events, in which conferences and workshops are organised or supported by UNU/IIST, and
7. Dissemination, in which UNU/IIST regularly distributes to developing countries information on international progress of software technology.

Fellows, who are young scientists and engineers from developing countries, are invited to actively participate in all these projects. By doing the projects they are trained.

At present, the technical focus of UNU/IIST is on formal methods for software development. UNU/IIST is an internationally recognised center in the area of formal methods. However, no software technique is universally applicable. We are prepared to choose complementary techniques for our projects, if necessary.

UNU/IIST produces a report series. Reports are either Research R, Technical T, Compendia C or Administrative A. They are records of UNU/IIST activities and research and development achievements. Many of the reports are also published in conference proceedings and journals.

Please write to UNU/IIST at P.O. Box 3058, Macau or visit UNU/IIST home page: http://www.iist.unu.edu, if you would like to know more about UNU/IIST and its report series.

Zhou Chaochen, Director — 01.8.1997 – 31.7.2003

The United Nations University

UNU/IIST

International Institute for Software Technology

P.O. Box 3058
Macau

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

Andres Flores, Luis Reynoso and Richard Moore

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. Specifications of actual design patterns based on this model and examples of using these to verify that a design matches a pattern can be found in related reports [18, 11, 4].

Andres Flores is a Fellow of UNU/IIST (November 1999 to August 2000), on leave from Comahue University, Neuquén, Argentina, where he is an Assistant Teacher. His research interests are focused on the software engineering disciplines, mainly on those related with software analysis and design. He is currently working on the combination of formal and informal methods and its application in a real domain.

Luis Reynoso is a Fellow of UNU/IIST (November 1999 to May 2000), on leave from Comahue University, Neuquén, Argentina, where he is teaching assistant. His research interests are focused on the combination of formal and informal methods and software engineering. He also works in the Cadastral Provincial Direction of the Public Administration of the government of the province of Neuquén.

Richard Moore is a Research Fellow on the staff of UNU/IIST, a position he took up on October 1st 1995. He has an M.A. in mathematics from the University of Cambridge and a Ph.D. in physics from the University of Manchester. He has been engaged in computing science research in the field of formal methods since 1985, a large part of which was carried out in the formal methods group at Manchester University. He has written several papers on formal methods and is co-author of two books on formal methods – mural: a Formal Development Support System; and Proof in VDM: A Practitioner's Guide. He has also worked for the Defence Research Agency in Malvern, UK, on various formal methods projects, both as a consultant and as a full-time member of staff.

Contents

1 Introduction

The term *pattern* was originally used by architect Christopher Alexander [1] to describe and capture recurring themes that he found in architecture, though the same concept is common to many different fields: music, literature, psychology, archaeology, architecture, and so on [8]. More recently, the term was adopted by software engineers to represent key aspects of commonly occurring structures in software systems.

One area of software engineering in which the use of patterns is popular is software design. Software *design patterns* are generic and abstract and embody "best practice" solutions to design problems which recur in a range of different contexts [2], although the solutions represented by the patterns are not necessarily the simplest or most efficient solutions for any given problem [14]. In this way, 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 also useful because they provide designers with an effective "shorthand" for communicating with each other about complex concepts [5]: 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. [12]. The GoF catalogue is thus a description of the know-how of expert designers in problems appearing in various different domains.

The patterns in the GoF catalogue are described using a consistent format which has in fact effectively been adopted as the standard way of presenting software design patterns. This uses a graphical notation based on an extension of OMT (Object Modelling Technique [19]) to represent the main constituents of the pattern – classes, methods, and relationships between classes – and supplements this with natural language descriptions of the intent and motivation of the pattern and the roles and responsibilities of its constituents. In addition, examples of the use of the patterns, in the form of both designs and sample code, are included.

This form of presentation gives a very good intuitive picture of the patterns, but it 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. Moreover, it also makes it difficult to be certain that the patterns themselves are meaningful and contain no inconsistencies. Indeed, in some cases the descriptions of the patterns are intentionally left loose and incomplete to ensure that they are applicable in as wide a range of applications as possible, which can make it difficult for designers to be sure that they have interpreted and understood the patterns correctly.

A more precise notation can help to alleviate these problems and can also improve understanding of the patterns in general. One approach to this [9] represents patterns as formulae in LePUS, a language defined as a fragment of higher order monadic logic [10]. A second [16] formalises

[1] Gang of Four.

the temporal behaviour of patterns using the DisCo specification method, which is based on the Temporal Logic of Actions [15].

Our approach is based on that of [7, 6], which uses the RAISE Specification Language (RSL; [17]) to formally specify properties of the patterns, in particular the responsibilities and collaborations of the pattern participants. However, we significantly extend the scope of the model used therein. First, 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 could not be specified in the model used in [7, 6]. Second, we generalise the model so that it can describe an arbitrary object-oriented design and not just the patterns. And third, we formally specify how to match the design against a pattern. This allows us to formally specify the patterns in such a way that their consistency and completeness can be checked, and we are also able to formally verify that a given subset of a design corresponds to a given pattern.

There is also an important difference between the two models in the way the dynamic aspects of the patterns are specified. In the model of [7, 6] the structural aspects are specified statically while the collaborations are specified in terms of sequences of interactions. In our model, both the structural properties and the collaborations are represented statically, the collaborations being modelled partly by the relations between the classes and partly by the requests the operations make to other classes. This latter is incorporated by specifically modelling the bodies of the methods.

We begin by describing the essential elements that constitute a general object-oriented design in Section 2, then we go on to describe our formal model of this, which we have also written in RSL, in Section 3. Section 4 then explains how a design can be formally linked to a pattern, and Section 5 identifies and specifies a number of generic functions which represent properties of specific patterns. We conclude with a summary of our work and an indication of how it has already been applied and how we plan to extend it in the future.

2 Object-Oriented Design and GoF Design Patterns

There are many different techniques and notations for describing object-oriented designs, though they generally share the same basic elements and ideas. However, since we are primarily interested in using our formal model to specify properties of object-oriented design patterns, in particular the GoF patterns [12], we base it on the extended OMT [19] notation which is used in [12] to describe the structure of GoF patterns and which has to a large extent been used as a standard notation for describing patterns. An example of this notation is shown in Figure 1, which represents the structure of the Command pattern.

A design consists essentially of a collection of classes and a collection of relations linking the classes, where the classes represent the kinds of objects that make up the system and the relationships represent connections or communications between those objects. In the extended OMT notation, each class is depicted as a rectangle containing the name of the class in bold

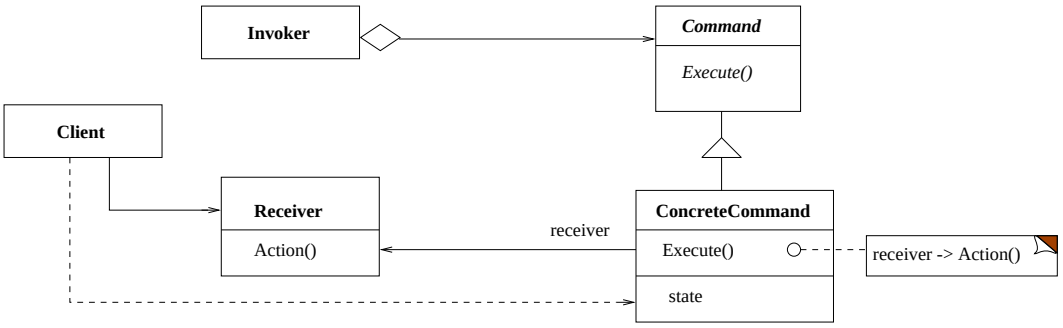

Figure 1: Command Pattern Structure

face type at the top. Below this, the signatures (i.e. names and appropriate parameters) of the operations or methods which objects of the class can perform appear in normal type, and finally the state variables or instance variables which represent the internal data stored by instances of the class appear. 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 description rectangle by a dashed line ending in a small circle.

Thus, for example, the structure in Figure 1 indicates that the class ConcreteCommand in the Command pattern is a concrete class which contains a concrete method called Execute and a state variable called state, while the class Command is an abstract class in which the method called Execute is abstract. In addition, the annotation attached to the Execute method in the ConcreteCommand class indicates that the action of this method is to invoke the Action method on the variable called receiver.

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*, *instantiation*, *aggregation* and *association* – and these are distinguished in the diagram using different types of lines.

Inheritance relations are drawn as solid lines with a triangle in the middle, the point and base of the triangle indicating respectively the superclass and subclasses in the relation. Thus, in the Command pattern the Command class is a superclass of the ConcreteCommand class.

Instantiation relations, which indicate that one class creates objects belonging to another class, are shown as dashed lines with an arrowhead on one end, the arrowhead pointing to the class

being instantiated. The instantiation relation between the Client class and the ConcreteCommand class in the Command pattern thus indicates that the Client class creates ConcreteCommand objects.

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 Invoker class and the Command class in Figure 1 is thus an aggregation relation which indicates that the Invoker class consists of a sub-object of the Command class.

Association relations are also shown as solid lines with an arrowhead on one end but they are unmarked at the other end. An association relation indicates that one class communicates with another, the arrowhead indicating the direction of the communication. The Command pattern has association relations between the Client and the Receiver classes and between the ConcreteCommand and the Receiver classes.

Association and aggregation relations also have an associated *arity* and may additionally have an associated name. The arity indicates the number of objects participating in the relation, and may be either 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, as shown in the aggregation relation between the MacroCommand and the Command classes in the extension to the Command pattern depicted in Figure 2. Names of relations, which generally correspond to state variables (although the state variables are not explicitly shown as such), are written above the line representing the relation as in the name commands of the same aggregation relation.

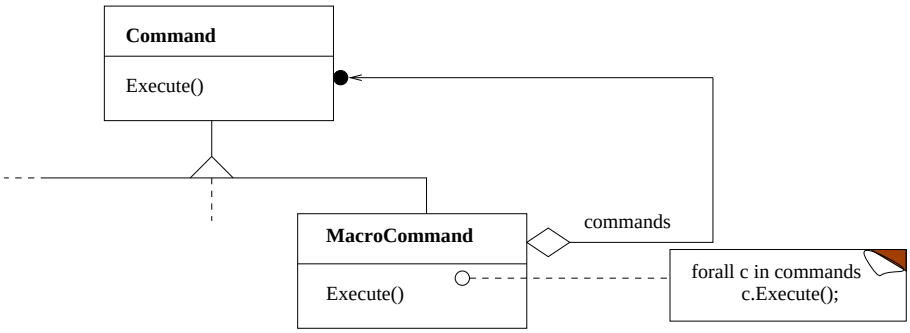

Figure 2: The Structure of the Macro Command

3 A Formal Model of Object-Oriented Design

The formal specification of the model is quite long so we divide the presentation into five sections. Each section deals with a particular element of the model and introduces the RSL type definitions which are used to describe that particular element as well as functions representing the well-formedness conditions that these types must satisfy and more general auxiliary functions which

are used in later sections. In fact each section also corresponds to an RSL module in the formal specification, and the name of the RSL object representing that module is given in parentheses after the title of the section. This name is used in later sections (modules) to refer to definitions in earlier sections (modules) in the standard RAISE fashion.

3.1 Some general definitions (G)

We begin by introducing a few basic types and constants which correspond to the most fundamental entities in an object-oriented design and which are used throughout the other modules. We also define some simple functions which describe their properties.

First, as explained in Section 2 above, a design essentially consists of a set of classes and a set of relations, with each class consisting in turn of a set of methods and a set of state variables. These four elements therefore constitute the basic building blocks of a design and they are uniquely distinguished by their names, though the names of (association and aggregation) relations are in fact the names of the variables with which they are associated. We therefore introduce three sort types to represent the three basic kinds of names appearing in the design: class names, method names, and variable names:

```
type
  Class_Name, Method_Name, Variable_Name
```

Each class has an associated type, which may be abstract or concrete, and each aggregation and association relation has an associated cardinality, which may be one or many. We model these possibilities using the variant types 'Class_Type' and 'Card' respectively, each of which simply consists of the appropriate two possible values.

```
type
  Class_Type == abstract | concrete,
  Card == one | many
```

Annotations to methods can be used to indicate the actions performed by the method, including invocations of other methods as in the annotation to the Execute method in the ConcreteCommand class in the Command pattern illustrated in Figure 1. Such invocations in general involve a variable which represents the receiver of the invocation and which is usually also the name of a corresponding association or aggregation relation, together with the name of the method the receiver should execute (in the Execute method the receiver is represented by the variable receiver, which is also the name of the association relation between the ConcreteCommand class and the Receiver class, and Action is the method it should execute). However, in some cases a method needs to invoke another method in the same class or in a superclass of its own class, and

in general relations between a class and itself are not shown in the extended OMT diagrams, which means that in these cases there is no association or aggregation relation, and hence no corresponding variable name to use in the invocation. To model these situations, we introduce two reserved variable names 'self' and 'super', which respectively represent the receivers of invocations to the same class and to a superclass of the current class (cf. the variables of the same name in Smalltalk). The fact that 'self' and 'super' are different values is enforced by an axiom.

value
  self, super : Variable_Name

axiom
  self $\neq$ super

The state (instance) variables of a class can simply be described as a set of variables, except that the set cannot include the two reserved variables 'self' and 'super'. We therefore introduce a subtype 'Wf_Vble_Name' of 'Variable_Name' whose defining predicate 'not_self_super' excludes these two values, and we use this type to define the type 'State' which represents the state variables of a class.

type
  Wf_Vble_Name = {| v : Variable_Name • not_self_super(v) |},
  State = Wf_Vble_Name-**set**

value
  not_self_super : Variable_Name $\rightarrow$ **Bool**
  not_self_super(v) $\equiv$ v $\neq$ self $\wedge$ v $\neq$ super

Similarly, the parameters passed as arguments to an invocation of a method are basically a list of variables which cannot include the variable 'super', though 'self' is allowed since an object can legitimately pass itself as an argument to a method call to another object. The defining predicate 'wf_vble_name' of the subtype 'Wf_Variable_Name' thus only excludes the value 'super', and this subtype is used to define the type 'Actual_Parameters' which represents the parameters of an invocation.

type
  Wf_Variable_Name = {| v : Variable_Name • wf_vble_name(v) |},
  Actual_Parameters = Wf_Variable_Name*

value
  wf_vble_name : Variable_Name $\rightarrow$ **Bool**
  wf_vble_name(v) $\equiv$ v $\neq$ super

A method definition can also include parameters, which we refer to as its formal parameters to distinguish them from the actual parameters used in invocations. The extended OMT notation in fact allows two kinds of formal parameters: ones in which only the name of the variable is given, and ones in which the variable name is accompanied by a class name which indicates the class of the object the variable represents. Formal parameters may not include the values 'self' or 'super' since they represent generic "placeholders" for actual variables, and all variable names used in the formal parameters must be different so that the variables can be distinguished in the body of the method.

We use the variant type 'Parameter' to represent the two different kinds of formal parameters, the variable names in both variants being represented by the type 'Wf_Vble_Name' which automatically excludes the unwanted values 'self' and 'super'. Then the formal parameters of a method are described by the subtype 'Wf_Formal_Parameters', which is a list of parameters in which no two parameters have the same variable name.

The defining predicate 'is_wf_formal_parameters' of the subtype, which represents the above consistency condition, is written in terms of the auxiliary function 'type_parameter' which simply extracts the variable name from a parameter.

```
type
  Parameter ==
    var(Wf_Vble_Name) |
    paramTyped(paramName : Wf_Vble_Name, className : Class_Name),
  Wf_Formal_Parameters =
    {| p : Parameter* • is_wf_formal_parameters(p) |}

value
  is_wf_formal_parameters : Parameter* → Bool
  is_wf_formal_parameters(p) ≡
    (
      ∀ i, j : Nat •
        {i, j} ⊆ inds p ∧
        type_parameter(p(i)) = type_parameter(p(j)) ⇒
        i = j
    ),

  type_parameter : Parameter → Variable_Name
  type_parameter(p) ≡
    case p of var(v) → v, paramTyped(v, c) → v end
```

We conclude this section by introducing some constants and auxiliary functions which will be used in later sections.

First, several GoF patterns involve interactions with collections of objects, though these in-

teractions are always implicit in the names of the methods (for example the methods Attach and Detach in the Observer pattern) and are not otherwise specified. We choose to make these interactions at least partially explicit in our model by introducing reserved method names 'collectionadd', 'collectionremove' and 'collectionelement' to represent general operations on some form of collection, though the exact form of the collection (set, list, or whatever) is left unspecified. Thus, the methods 'collectionadd' and 'collectionremove' represent methods for adding an element to a collection or removing an element from a collection respectively, while the method 'collectionelement' returns one element from a collection. Each of the methods requires a single parameter, which in the case of 'collectionadd' and 'collectionremove' represents the object to be added to/removed from the collection and in the case of 'collectionelement' indicates somehow the element required (for example if the collection is a list of objects the parameter to 'collectionelement' might indicate the position in the list).

At this level we only define the names of the methods, together with the auxiliary function 'is_collection_method' which checks whether a given method name is one of these reserved names. Other properties of these methods are defined in later sections.

```
value
  collectionadd, collectionremove, collectionelement : Method_Name,

  is_collection_method : Method_Name → Bool
  is_collection_method(m) ≡
    m = collectionadd ∨ m = collectionremove ∨ m = collectionelement
```

Finally, we define five auxiliary functions relating to parameters of methods. The first, 'parameter_in_set', checks whether a given variable name corresponds to one of the formal parameters of a method. This is used as the precondition to the second function, 'position_of', which returns the position of a given variable in the list of formal parameters – this is uniquely defined because of the property that no two formal parameters can have the same variable name. The third function 'match_params' checks that the actual parameters of a method are consistent with its formal parameters – there must simply be the same number of each. The fourth function 'set_f_params' gives the set of variables in the formal parameters of a method (i.e. ignoring any types). And the fifth function 'rol_of_parameter' returns the declared type of a given parameter, the precondition of the function ensuring that the parameter actually has a type.

```
value
  parameter_in_set : Variable_Name × Wf_Formal_Parameters → Bool
  parameter_in_set(v, f) ≡
    (∃ p : Parameter • p ∈ elems f ∧ type_parameter(p) = v),

  position_of : Variable_Name × Wf_Formal_Parameters ~→ Nat
  position_of(v, l) as i
    post i ∈ inds l ∧ type_parameter(l(i)) = v
```

pre parameter_in_set(v, l),

match_params : Actual_Parameters × Wf_Formal_Parameters → **Bool**
match_params(a, f) ≡ **len** a = **len** f,

set_f_params : Wf_Formal_Parameters → Variable_Name-**set**
set_f_params(fp) ≡
{ type_parameter(p) | p : Parameter • p ∈ **elems** fp },

rol_of_parameter : Variable_Name × Wf_Formal_Parameters $\xrightarrow{\sim}$ Class_Name
rol_of_parameter(v, f) **as** c
post paramTyped(v, c) ∈ **elems** f
pre var(v) ∉ **elems** f ∧ parameter_in_set(v, f)

3.2 Methods (M)

As explained in Section 2, annotations to methods indicate the actions that the methods perform. In fact there are essentially only two different kinds of interactions that appear in these annotations: invocations and instantiations.

An invocation represents an interaction which 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.

We model a request as the record type 'Actual_Signature', which comprises the name of the method together with its actual parameters (see Section 3.1), and the type 'Invocation' then models the whole of this kind of interaction, consisting of the name of the variable representing the receiver of the request plus the appropriate signature.

type
Actual_Signature ::
meth_name : G.Method_Name a_params : G.Actual_Parameters,
Invocation ::
call_vble : G.Variable_Name call_sig : Actual_Signature

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 representing an instantiation as the type 'Instantiation' which consists of the name of the class to be instantiated together with a possibly empty list of actual parameters to be used by the instantiation method.

type
  Instantiation ::
    class_name : G.Class_Name a_params : G.Actual_Parameters

In some cases, however, an annotation can indicate that a particular method should carry out different actions under different conditions, for example like if-then-else expressions in standard programming languages. We model these annotations using the type 'Alternative', which consists of two alternative "blocks" of actions. Each block, which is modelled by the type 'Block', consists of a list of requests, and each request is either an invocation, an instantiation, an alternative, or is just treated as plain text (the type 'Dots').

type
  Alternative = Block × Block,
  Block = Request*,
  Request = Invocation | Instantiation | Alternative | Dots,
  Dots = **Text**

Annotations may also indicate the results returned by methods and assignments to variables within the bodies of the methods, including assignments to both state variables and local (dummy) variables.

In actual code, the result of a method is likely to be a tuple or a list of variables, but we model this abstractly and use simply a set of variables since this is sufficient to capture the properties of a high-level design. However, the result set cannot include the variable 'super', so we use the type 'Wf_Variable_Name' to define the type 'Result'.

type
  Variables = G.Wf_Variable_Name-**set**,
  Result = Variables

For the assignments, two forms are used: one where the results of some request (instantiation, invocation or alternative) are assigned to variables, generally for use as parameters or receivers

in later requests, and the second where the parameters of the method are assigned to variables, generally state variables. We model the assignment of the results of a request as a mapping from sets of variables to the requests themselves, and the assignment to other variables, including parameters, as a mapping from sets of variables to sets of variables. In both cases the domain value of the mapping should not include the variables 'self' or 'super', and in the second case the range value of the mapping should not include the variable 'super' but it may include the variable 'self'. These two possibilities are combined in the type 'Variable_Change', which maps sets of variables to either requests or sets of variables and which additionally has a well-formedness constraint 'is_wf_vchange' which requires that we do not make an assignment to the empty set of variables. Additional constraints on the variable change map are defined below as part of the well-formedness condition on a method as a whole.

type
  Request_or_Var = Request | Variables,
  Variable_Change =
    {| m : G.Wf_Vble_Name-**set** $\overrightarrow{m}$ Request_or_Var • is_wf_vchange(m) |}

value
  is_wf_vchange : (G.Wf_Vble_Name-**set** $\overrightarrow{m}$ Request_or_Var) → **Bool**
  is_wf_vchange(m) ≡ {} ∉ **dom** m

The extended OMT notation distinguishes between abstract methods and concrete methods. Abstract methods correspond in general to methods in which only the signature is defined and are unexecutable, while concrete methods are defined completely and can be executed. However, in some cases it is useful to define some methods abstractly in a superclass even if they should not be executable or even do not make sense in all subclasses. (See for example the Composite pattern in [13] and the corresponding discussion thereof in [11]: the child management operations Add, Remove, and GetChild do not make sense at all in the Leaf classes although they are in principle available because they are inherited from the Component class.) In this case, the method should be concrete in all concrete subclasses but it is not executable. We therefore subdivide the classification of a concrete method into *error* and *implemented* methods, and in our model we therefore use a classification which distinguishes these three kinds of method.

Neither an abstract method nor an error method can be executed, so performs no actions and therefore does not have a *body* (i.e. it is defined entirely by its signature and possibly its result). We therefore represent the bodies of these methods simply by the constants 'defined' and 'error' respectively. An implemented method, on the other hand, must actually perform some action and so does have a body, though this could be empty since the actions of the method might not be specified in the design. In this case, the body consists of the requests and assignments noted in the annotation to the method, and we therefore model it using the variable change mapping defined above together with a list of requests which represents the actions performed by the method and the order in which they are performed. The body of the three kinds of methods is thus modelled using the variant type 'Method_Body'.

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

Then a method consists of a body, a result, and some formal parameters.

type
Method ::
f_params : G.Wf_Formal_Parameters
meth_res : Result
body : Method_Body

Of course there are a number of consistency conditions that the various elements in the above record type must satisfy in order for the definition to be reasonable.

First, any request which appears in the range of the variable change mapping of an implemented method must appear in the list of requests comprising the body of that method. This is represented by the function 'receiver_in_call_vble', which is defined in terms of three auxiliary functions 'variable_change_body', 'request_list_body' and 'requests'. The first two of these have a precondition which requires that the method is implemented (the function 'is_implemented') and return respectively the variable change mapping and the list of requests in the body of an implemented method. The third calculates the set of requests appearing in the range of the variable change mapping.

value
variable_change_body : Method $\xrightarrow{\sim}$ Variable_Change
variable_change_body(b) ≡
let implemented(s, _) = body(b) **in** s **end**
pre is_implemented(b),

request_list_body : Method $\xrightarrow{\sim}$ Request*
request_list_body(b) ≡
let implemented(_, i) = body(b) **in** i **end**
pre is_implemented(b),

is_implemented : Method → **Bool**
is_implemented(m) ≡ body(m) ≠ defined ∧ body(m) ≠ error,

requests : Request_or_Var-**set** → Request-**set**

```
requests(s) ≡
  { m | m : Request • Request_or_Var_from_Request(m) ∈ s },

receiver_in_call_vble : Method → Bool
receiver_in_call_vble(m) ≡
  is_implemented(m) ⇒
    requests(rng variable_change_body(m)) ⊆ elems request_list_body(m)
```

The second constraint is that the result of every instantiation in the body of an implemented method must be assigned to a variable in the variable change mapping and therefore must appear in the range of that mapping. This condition is in fact not strictly necessary but corresponds to a sort of "no waste" condition – the instantiation must return a new object and if this is not assigned to a variable it is just lost and there was therefore no point in creating it. This property is embodied in the function 'correct_list_ins', which is defined in terms of two more auxiliary functions 'instantiation_in_request_list' and 'instantiation_in_vble_change'. These check whether a given instantiation occurs in the range of the variable change mapping, respectively the list of requests in the body of an implemented method and their definitions again use the functions 'variable_change_body' and 'request_list_body' introduced immediately above.

```
value
  instantiation_in_request_list : Instantiation × Method → Bool
  instantiation_in_request_list(ins, m) ≡
    is_implemented(m) ∧
    Request_from_Instantiation(ins) ∈ elems request_list_body(m),

  instantiation_in_vble_change : Instantiation × Method → Bool
  instantiation_in_vble_change(ins, m) ≡
    is_implemented(m) ∧
    Request_from_Instantiation(ins) ∈ rng variable_change_body(m),

  correct_list_ins : Method → Bool
  correct_list_ins(m) ≡
    is_implemented(m) ⇒
      (
        ∀ i : Instantiation •
          instantiation_in_request_list(i, m) ⇒ instantiation_in_vble_change(i, m)
      )
```

The third property requires that the variable change mapping does not make assignments to formal parameters of an implemented method and is defined by the function 'is_correct_fparams'. This is in turn defined using the new auxiliary function 'changed_variables', which simply returns the set of all variables which appear on the left-hand side of assignments in the variable change mapping, and the auxiliary function 'set_f_params' defined in Section 3.1.

```
value
  changed_variables : Method ~→ G.Wf_Vble_Name-set
  changed_variables(m) ≡
    { v |
      v : G.Wf_Vble_Name, vs : G.Wf_Vble_Name-set •
        vs ∈ dom variable_change_body(m) ∧ v ∈ vs
    }
    pre is_implemented(m),

  is_correct_fparams : Method → Bool
  is_correct_fparams(m) ≡
    is_implemented(m) ⇒ changed_variables(m) ∩ G.set_f_params(f_params(m)) = {}
```

The fourth constraint is that an error method does not return a result, so its result is the empty set of variables. It is defined by the function 'is_error_method'.

```
value
  is_error_method : Method → Bool
  is_error_method(m) ≡ body(m) = error ⇒ meth_res(m) = {}
```

The remaining three constraints basically ensure that every assignment in the variable change mapping is consistent in the sense that the number of variables on the two sides of the assignment must be the same. The first function 'correct_inst_assig' deals with instantiations, the result of which is always a single variable representing the single object created. This object must therefore be assigned to a single variable.

```
value
  correct_inst_assig : Method → Bool
  correct_inst_assig(m) ≡
    is_implemented(m) ⇒
      let vm = variable_change_body(m) in
        (
          ∀ ins : Instantiation, vs : G.Variable_Name-set •
            vs ∈ dom vm ∧ vm(vs) = ins ⇒ card vs = 1
        )
      end
```

The second function 'correct_vble_assig' deals with sets of variables, in which case the two sets of variables (i.e. the domain value and the range value) must contain the same number of variables.

```
value
  correct_vble_assig : Method → Bool
  correct_vble_assig(m) ≡
    is_implemented(m) ⇒
      let vm = variable_change_body(m) in
        (
          ∀ vd, vr : Variables •
            vd ∈ dom vm ∧ vm(vd) = vr ⇒ card vd = card vr
        )
      end
```

Finally, the third function 'correct_collel_assig' deals with invocations of the reserved method 'collectionelement'. This method always returns a single object as its result, so as in the case of instantiations we again constrain the corresponding domain value in the variable change mapping to contain only one variable.

```
value
  correct_collel_assig : Method → Bool
  correct_collel_assig(m) ≡
    is_implemented(m) ⇒
      let vm = variable_change_body(m) in
        (
          ∀ inv : Invocation, vs : G.Variable_Name-set •
            vs ∈ dom vm ∧
            vm(vs) = inv ∧
            meth_name(call_sig(inv)) = G.collectionelement
              ⇒
            card vs = 1
        )
      end
```

These seven constraints are combined together in the function 'is_wf_method', which is then used as the defining predicate for the subtype 'Wf_Method' which represents well-formed methods.

```
value
  is_wf_method : Method → Bool
  is_wf_method(m) ≡
    receiver_in_call_vble(m) ∧
    correct_list_ins(m) ∧
    is_correct_fparams(m) ∧
    is_error_method(m) ∧
    correct_inst_assig(m) ∧
```

correct_vble_assig(m) ∧ correct_collel_assig(m)

type
Wf_Method = {| m : Method • is_wf_method(m) |}

The collection of all methods in a class is then modelled by the type 'Map_Methods', which associates the name of each method with its definition. Using a map here automatically ensures that no two methods in the same class have the same name. At this level there is only one constraint, namely that the reserved method names 'collectionadd', 'collectionremove' and 'collectionelement' cannot be used as the names of methods in the design. This is expressed using the function 'is_wf_class_method' and this is in turn used as the defining predicate of the subtype 'Class_Method' which then represents the well-formed collection of all methods in a class.

type
Map_Methods = G.Method_Name $\overrightarrow{m}$ Wf_Method,
Class_Method = {| m : Map_Methods • is_wf_class_method(m) |}

value
is_wf_class_method : Map_Methods → **Bool**
is_wf_class_method(m) ≡
G.collectionadd ∉ **dom** m ∧
G.collectionremove ∉ **dom** m ∧ G.collectionelement ∉ **dom** m

We again finish by defining some auxiliary functions which will be useful in later sections.

First the function 'is_defined' is analogous to the function 'is_implemented' defined above except that it checks whether a method is abstract.

value
is_defined : Wf_Method → **Bool**
is_defined(m) ≡ body(m) = defined

Next we define analogues of the functions 'instantiation_in_request_list' and 'instantiation_in_vble_change' which check the corresponding properties for invocations instead of instantiations. The functions 'invocation_in_request_list' and 'invocation_in_vble_change' thus check whether a given invocation occurs in the range of the variable change mapping, respectively the list of requests in the body of an implemented method.

value
invocation_in_request_list : Invocation × Method → **Bool**

```
invocation_in_request_list(inv, m) ≡
    is_implemented(m) ∧
    Request_from_Invocation(inv) ∈ elems request_list_body(m),

invocation_in_vble_change : Invocation × Wf_Method → Bool
invocation_in_vble_change(inv, m) ≡
    is_implemented(m) ∧
    Request_from_Invocation(inv) ∈ rng variable_change_body(m)
```

The function 'method_cut' is used to generate a new implemented method from an existing implemented method by removing from the body of the method all requests which occur before a given instantiation. This is in fact done using the function 'less' which simply iterates through a list of requests discarding elements until the required instantiation is found.

Note that there is no check that the instantiation actually occurs in the body of the method, in which case the body of the resulting method would be empty. Note also that the function 'method_cut' is under-specified – the variable change mapping in the new method could in principle be changed to any value provided of course that value satisfied all the consistency conditions. This is in fact not a problem since the method is only used as an auxiliary function within the function 'client_comment' (see Section 5.4) to check that requests occur in the correct order in the body and this is determined solely by the list of requests.

```
value
    method_cut : Method × Instantiation ~→ Method
    method_cut(m, ins) as mp
        post
            is_implemented(mp) ∧ f_params(mp) = f_params(m) ∧
                request_list_body(mp) = less(request_list_body(m), ins)
        pre is_implemented(m),

    less : Request* × Instantiation → Request*
    less(m, i) ≡ if hd (m) = i then tl (m) else less(tl (m), i) end
```

The function 'order' is also related to the order of requests in the body of a method. Specifically it checks whether the list of requests in the body of a method contain two given requests with the first request preceding the second. Note however that it takes no account of multiple occurrences of the same request in the request list so that, for example, if the second request occurs twice in the request list, once before the first request and once after, the function will still return true.

```
value
    order : Request × Request × Request* → Bool
    order(in1, in2, mlist) ≡
```

```
    (
      ∃ i, j : Nat •
        {i, j} ⊆ inds mlist ∧
        mlist(i) = in1 ∧ mlist(j) = in2 ∧ i < j
    )
```

The function 'is_one_one' checks if a variable change mapping is one-one, that is does not assign the same range value to different domain values.

```
value
  is_one_one : Variable_Change → Bool
  is_one_one(m) ≡
    (
      ∀ x, y : G.Variable_Name-set •
        {x, y} ⊆ dom m ∧ m(x) = m(y) ⇒ x = y
    )
```

The function 'has_invocation_param' checks whether a given method (m) is implemented and has in its request list an invocation to a given variable (v) of a given method (mn) with a given single actual parameter (p).

```
value
  has_invocation_param :
    Method × G.Variable_Name × G.Method_Name × G.Variable_Name → Bool
  has_invocation_param(m, v, mn, p) ≡
    let s = mk_Actual_Signature(mn, ⟨p⟩), i = mk_Invocation(v, s) in
      invocation_in_request_list(i, m)
    end
```

The function 'has_assignment' checks that a given method (m) is implemented and that its variable change map contains an assignment to a given single variable (tov) of the result of an invocation to another given variable (iv) of a given method (mn) with no parameters.

```
value
  has_assignment :
    Method × G.Variable_Name × G.Variable_Name × G.Method_Name → Bool
  has_assignment(m, tov, iv, mn) ≡
    is_implemented(m) ∧
    let
      vn = variable_change_body(m), s = mk_Actual_Signature(mn, ⟨⟩)
```

```
    in
      {tov} ∈ dom vn ∧ vn({tov}) = mk_Invocation(iv, s)
    end
```

The function 'has_assignment_invocation_param' checks whether a given method (m) is implemented and has two specific invocations in its body in the given order. The first of these invocations invokes one given method (m_1) on a given variable (v_2) with no actual parameters and assigns the result to a (dummy) variable (v_1) in the variable change map. The second invokes a second given method (m_2) on the same variable v_2, using the variable v_1 as the sole parameter of the invocation.

```
value
  has_assignment_invocation_param :
    Method × G.Variable_Name × G.Variable_Name × G.Method_Name ×
      G.Method_Name
    →
      Bool
  has_assignment_invocation_param(m, v1, v2, m1, m2) ≡
    is_implemented(m) ∧
    let
      vm = variable_change_body(m),
      s1 = mk_Actual_Signature(m1, ⟨⟩),
      s2 = mk_Actual_Signature(m2, ⟨v1⟩),
      inv1 = mk_Invocation(v2, s1),
      inv2 = mk_Invocation(v2, s2),
      r1 = Request_from_Invocation(inv1),
      r2 = Request_from_Invocation(inv2),
      rlb = request_list_body(m)
    in
      {v1} ∈ dom vm ∧
      vm({v1}) = inv1 ∧ inv2 ∈ elems rlb ∧ order(r1, r2, rlb)
    end
```

Finally, the function 'a_params_from_set' checks that the actual parameters of some method is some permutation of some given set of variables with no duplicates.

```
value
  a_params_from_set : G.Actual_Parameters × G.Variable_Name-set → Bool
  a_params_from_set(ap, vs) ≡ len ap = card vs ∧ elems ap = vs
```

3.3 Classes (C)

All the various elements which comprise a class – state variables, methods, and class type – have already been defined in Sections 3.1 and 3.2 above, so we simply combine these appropriately in the record type 'Design_Class' to obtain the formal specification of a class.

```
type
  Design_Class ::
    class_state : G.State
    class_methods : M.Class_Method
    class_type : G.Class_Type
```

However, state variables cannot be used as formal parameters of methods since this would lead to ambiguity, so we define the function 'is_wf_class' to capture this property and use this as the defining predicate for the subtype 'Wf_Class' of well-formed classes. The function 'method_in_class' used in the definition of 'is_wf_class' simply checks that a given method belongs to a given class.

```
value
  is_wf_class : Design_Class → Bool
  is_wf_class(c) ≡
    (
      ∀ m : M.Wf_Method •
        method_in_class(m, c) ⇒
        class_state(c) ∩ G.set_f_params(M.f_params(m)) = {}
    ),

  method_in_class : M.Wf_Method × Design_Class → Bool
  method_in_class(m, c) ≡ m ∈ rng class_methods(c)

type
  Wf_Class = {| c : Design_Class • is_wf_class(c) |}
```

The collection of all classes in a design is then modelled by the type 'Classes', which associates the name of each class with its definition. Again, using a map here automatically ensures that no two classes in the design have the same name. Other constraints on classes are defined in later sections.

```
type
  Classes = G.Class_Name ~m~> Wf_Class
```

Again we finish by defining some auxiliary functions which will be useful in later sections. The first, 'method_name_in_class', is very similar to the function 'method_in_class' defined above except it checks whether a class contains a method with a given name instead of with a given definition.

value
  method_name_in_class : G.Method_Name × Design_Class → **Bool**
  method_name_in_class(m, c) ≡ m ∈ **dom** class_methods(c)

The functions 'is_abstract_class' and 'is_concrete_class' check respectively whether the stated type of a class is abstract or concrete.

value
  is_abstract_class : Design_Class → **Bool**
  is_abstract_class(c) ≡ class_type(c) = G.abstract,

  is_concrete_class : Design_Class → **Bool**
  is_concrete_class(c) ≡ class_type(c) = G.concrete

Finally, the functions 'sig_invok_in_class' and 'sig_has_result' both check that the method name appearing in a given signature represents one of the methods in a given class. In addition, the function 'sig_invok_in_class' checks that the number of parameters in the signature is the same as the number expected by the method, while the function 'sig_has_result' checks that the method returns a result.

value
  sig_invok_in_class : Design_Class × M.Actual_Signature → **Bool**
  sig_invok_in_class(c, s) ≡
    let M.mk_Actual_Signature(n, p) = s, cm = class_methods(c) **in**
      n ∈ **dom** cm ∧ G.match_params(p, M.f_params(cm(n)))
    end,

  sig_has_result : Design_Class × M.Actual_Signature → **Bool**
  sig_has_result(c, s) ≡
    let M.mk_Actual_Signature(n, p) = s, cm = class_methods(c) **in**
      n ∈ **dom** cm ∧ M.meth_res(cm(n)) ≠ {}
    end

3.4 Relations(R)

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.

In the case of instantiation and inheritance relations, there can be at most one such relation between any pair of classes. The relation 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, different relations being distinguished by their names, which are essentially variable names except the names 'self' and 'super' may not be used. Association and aggregation relations also have associated source and sink cardinalities, which may be one or many.

We use the type 'Ref' to record the name and cardinalities of association and aggregation relations, the name being modelled using the type 'Wf_Vble_Name' in order to exclude the unwanted values and the cardinality by the type 'Card' (see Section 3.1). Then the variant type 'Relation_Type' models the type of a relation: for inheritance and instantiation relations it is just a constant of the appropriate name, while for aggregation and association relations it is a function of the appropriate name applied to a value of type 'Ref'. A relation as a whole is then modelled by the record type 'Design_Relation', which comprises the type of the relation and the names of its source and sink classes as explained above.

type
  Ref ::
    relation_name : G.Wf_Vble_Name
    sink_card : G.Card
    source_card : G.Card,
  Relation_Type ==
    inheritance | association(as_ref : Ref) | aggregation(ag_ref : Ref) | instantiation,
  Design_Relation ::
    relation_type : Relation_Type
    source_class : G.Class_Name
    sink_class : G.Class_Name

The well-formedness condition 'wf_relation' on a relation states that instantiation relations are not explicitly shown between a class and itself in the extended OMT diagram (every class is generally assumed to be able to instantiate itself) and that there cannot be inheritance relations between a class and itself since this would lead to essentially infinite inheritance structures. The auxiliary function 'is_assoc_or_aggr' simply checks whether a relation is either an association or an aggregation relation.

```
value
  wf_relation : Design_Relation → Bool
  wf_relation(r) ≡
    ~ is_assoc_or_aggr(r) ⇒ source_class(r) ≠ sink_class(r),

  is_assoc_or_aggr : Design_Relation → Bool
  is_assoc_or_aggr(r) ≡
    relation_type(r) ≠ inheritance ∧ relation_type(r) ≠ instantiation

type
  Wf_Relation = {| r : Design_Relation • wf_relation(r) |}
```

The set of all relations in a design must satisfy a number of consistency conditions.

The first of these extends the constraint that a class cannot have an inheritance relation with itself to rule out more complex transitive relationships which correspond to similarly infinite or otherwise meaningless structures. One obvious extension of this is that there cannot be any "loop" of inheritance relations in a design, that is a collection of classes c_1 to c_n such that c_{i+1} is a subclass of c_1 for all i and c_1 is a subclass of c_n. However, a similar loop of aggregation relations also does not make sense because an aggregation relation indicates that one object is a sub-object of another object so a loop of this form would mean that an object would effectively be a sub-object of itself and so would also correspond to an infinite structure. Similarly a loop in which one relation is an instantiation relation and all other relations are aggregation relations does not make sense because it implies that a sub-object (the source of the instantiation relation) is responsible for creating the object which contains it (the sink of the instantiation relation).

We define the auxiliary function 'exists_loop' to check whether a given set of relations contains some loop of relations (i.e. taking no account of the types of the relations so allowing mixed types of relation in the loop). This is in turn written in terms of the function 'exists_loop_from' which checks whether there is a chain of relations linking two given classes.

```
value
  exists_loop : Wf_Relation-set → Bool
  exists_loop(s) ≡
    (
      ∀ e : Wf_Relation •
        e ∈ s ⇒ exists_loop_from(source_class(e), source_class(e), s)
    ),

  exists_loop_from :
    G.Class_Name × G.Class_Name × Wf_Relation-set → Bool
  exists_loop_from(c1, c2, s) ≡
    (
      ∃ r : Wf_Relation •
```

```
        r ∈ s ∧
        source_class(r) = c1 ∧
        (sink_class(r) = c2 ∨ exists_loop_from(sink_class(r), c2, s))
    )
```

We also define auxiliary functions to check whether a given set of relations corresponds to one of the sets we wish to rule out – all aggregations (the function 'all_aggr' which is written in terms of another auxiliary function 'is_aggregation' which simply checks whether some given relation is an aggregation relation), all inheritance relations (the function 'all_inh'), and one instantiation relation together with a set of aggregation relations (the function 'inst_aggr_loop'). Note that in this last case the earlier constraint that instantiation relations between a class and itself are not shown implies that the loop must contain at least one aggregation relation.

```
value
  all_aggr : Wf_Relation-set → Bool
  all_aggr(s) ≡
    (∀ e : Wf_Relation • e ∈ s ⇒ is_aggregation(e)),

  is_aggregation : Design_Relation → Bool
  is_aggregation(r) ≡
    (∃ a : Ref • relation_type(r) = aggregation(a)),

  all_inh : Wf_Relation-set → Bool
  all_inh(s) ≡
    (
      ∀ e : Wf_Relation • e ∈ s ⇒ relation_type(e) = inheritance
    ),

  inst_aggr_loop : Wf_Relation-set → Bool
  inst_aggr_loop(s) ≡
    (
      ∃! e1 : Wf_Relation •
        e1 ∈ s ∧
        relation_type(e1) = instantiation ∧ all_aggr(s \ {e1})
    )
```

The function 'no_circularity' then captures the required property that a set of relations doesn't contain meaningless infinite loops – any non-empty subset s of relations which forms a loop cannot correspond to any of the unwanted loops described above.

```
value
  no_circularity : Wf_Relation-set → Bool
```

```
no_circularity(rs) ≡
  (
    ∀ s : Wf_Relation-set •
      s ≠ {} ∧ s ⊆ rs ∧ exists_loop(s) ⇒
        ∼ inst_aggr_loop(s) ∧ ∼ all_inh(s) ∧ ∼ all_aggr(s)
  )
```

The second consistency condition on the set of relations is that if two classes are related by an inheritance relation then there cannot be any other relation between the same two classes and in the same direction (though relations in the opposite direction are of course possible). We define three auxiliary functions to specify this property. The functions 'have_equal_sources' and 'have_equal_sinks' check whether two given relations have the same source class, respectively the same sink class, and the function 'is_compatible_relation' then uses these to check that two relations are *compatible*, that is if one of them is an inheritance relation and the other is some other kind of relation then they must have either different source classes or different sink classes (or both).

```
value
  is_compatible_relation : Wf_Relation × Wf_Relation → Bool
  is_compatible_relation(e1, e2) ≡
    relation_type(e1) = inheritance ∧ relation_type(e2) ≠ inheritance ⇒
      ∼ have_equal_sources(e1, e2) ∨ ∼ have_equal_sinks(e1, e2),

  have_equal_sources : Wf_Relation × Wf_Relation → Bool
  have_equal_sources((r1, r2)) ≡ source_class(r1) = source_class(r2),

  have_equal_sinks : Wf_Relation × Wf_Relation → Bool
  have_equal_sinks((r1, r2)) ≡ sink_class(r1) = sink_class(r2)
```

The third consistency constraint requires that association and aggregation relations in the design must be uniquely identified by their name. This property is again specified in terms of three auxiliary functions. The function 'is_association' is exactly analogous to the function 'is_aggregation' introduced above except that it checks whether some given relation is an association relation instead of an aggregation relation. The function 'vble_of_assoc_aggr' simply returns the name of the association or aggregation relation, which is a variable name. Finally, the function 'different_variable_name' checks that two different association or aggregation relations must have different names.

```
value
  different_variable_name : Wf_Relation × Wf_Relation → Bool
  different_variable_name(e1, e2) ≡
    is_assoc_or_aggr(e1) ∧ is_assoc_or_aggr(e2) ∧ (e1 ≠ e2) ⇒
```

```
        vble_of_assoc_aggr(e1) ≠ vble_of_assoc_aggr(e2),

    vble_of_assoc_aggr : Design_Relation ~→ G.Variable_Name
    vble_of_assoc_aggr(r) ≡
      if is_association(r) then
        relation_name(as_ref(relation_type(r)))
      else
        relation_name(ag_ref(relation_type(r)))
      end
      pre is_assoc_or_aggr(r),

    is_association : Design_Relation → Bool
    is_association(r) ≡
      (∃ a : Ref • relation_type(r) = association(a))
```

The function 'is_valid_relation' combines these last two constraints, and 'is_correct_relation' extends them from two relations to an arbitrary set of relations.

```
value
  is_valid_relation : Wf_Relation × Wf_Relation → Bool
  is_valid_relation(e1, e2) ≡
    is_compatible_relation(e1, e2) ∧ different_variable_name(e1, e2),

  is_correct_relation : Wf_Relation-set → Bool
  is_correct_relation(rs) ≡
    (
      ∀ e1, e2 : Wf_Relation •
        e1 ∈ rs ∧ e2 ∈ rs ⇒ is_valid_relation(e1, e2)
    )
```

Finally, we combine all three constraints in the function 'wf_relations' and use this as the defining predicate for the subtype 'Wf_Relations' of well-formed sets of relations.

```
value
  wf_relations : Wf_Relation-set → Bool
  wf_relations(rs) ≡ no_circularity(rs) ∧ is_correct_relation(rs)

type
  Wf_Relations = {| rs : Wf_Relation-set • wf_relations(rs) |}
```

As usual we complete this section by defining some auxiliary functions that will be useful in later sections.

The function 'sink_card' returns the sink cardinality of an association or aggregation relation.

```
value
  sink_card : Wf_Relation ~→ G.Card
  sink_card(r) ≡
    if is_association(r) then
      sink_card(as_ref(relation_type(r)))
    else
      sink_card(ag_ref(relation_type(r)))
    end
    pre is_assoc_or_aggr(r)
```

The function 'is_parent' checks whether a given class c_1 is an immediate superclass of the class c_2 with respect to some set of relations: the set of relations must contain an inheritance relation whose source class is c_1 and whose sink class is c_2.

```
value
  is_parent : G.Class_Name × G.Class_Name × Wf_Relations → Bool
  is_parent(c1, c2, rs) ≡
    (
      ∃ d : Wf_Relation •
        d ∈ rs ∧
        relation_type(d) = inheritance ∧
        source_class(d) = c1 ∧ sink_class(d) = c2
    )
```

This function is then used in the function 'is_superclass' to check the more general property that the class c_1 is a superclass of the class c_2 with respect to some set of relations: c_1 should be the parent of c_2 or a superclass of the parent of c_2.

```
value
  is_superclass : G.Class_Name × G.Class_Name × Wf_Relations → Bool
  is_superclass(c1, c2, rs) ≡
    is_parent(c1, c2, rs) ∨
    (
      ∃ c3 : G.Class_Name • is_parent(c3, c2, rs) ∧ is_superclass(c1, c3, rs)
    )
```

A class c_2 is a *leaf* in a hierarchy of classes starting from the class c_1 if c_1 is a superclass of c_2 and c_2 has no subclasses (i.e. there is no class c_3 whose parent is c_2).

value
is_leaf : G.Class_Name × G.Class_Name × Wf_Relations → **Bool**
is_leaf(c_1, c_2, rs) ≡
is_superclass(c_1, c_2, rs) ∧
∼
(
∃ c_3 : G.Class_Name • is_superclass(c_1, c_3, rs) ∧ is_parent(c_2, c_3, rs)
)

The function 'is_assoc_aggr_between' checks whether a relation r is an association or aggregation relation whose name, source class and sink class are v, c_1 and c_2 respectively.

value
is_assoc_aggr_between :
G.Variable_Name × G.Class_Name × G.Class_Name × Wf_Relation → **Bool**
is_assoc_aggr_between(v, c_1, c_2, r) ≡
source_class(r) = c_1 ∧
sink_class(r) = c_2 ∧
is_assoc_or_aggr(r) ∧ vble_of_assoc_aggr(r) = v

The function 'have_direct_assoc_aggr_rel' extends the same property to a set of relations, checking if there is some relation in the set which satisfies the function 'is_assoc_aggr_between'.

value
have_direct_assoc_aggr_rel :
G.Variable_Name × G.Class_Name × G.Class_Name × Wf_Relations → **Bool**
have_direct_assoc_aggr_rel(v, c_1, c_2, rs) ≡
(
∃ r : Wf_Relation • r ∈ rs ∧ is_assoc_aggr_between(v, c_1, c_2, r)
)

The function 'exists_assoc_aggr_relation_in_superclass' extends the same property to superclasses of the given class, that is it checks if there is an association or aggregation relation with the given name linking some superclass of c_1 to c_2.

value
exists_assoc_aggr_relation_in_superclass :
G.Variable_Name × G.Class_Name × G.Class_Name × Wf_Relations → **Bool**
exists_assoc_aggr_relation_in_superclass(v, c_1, c_2, rs) ≡
(

$\exists$ c_3 : G.Class_Name, r_1 : Wf_Relation •
is_superclass(c_3, c_1, rs) $\wedge$ $r_1 \in$ rs $\wedge$ is_assoc_aggr_between(v, c_3, c_2, r_1)
)

The two properties above are combined in the function 'exists_assoc_aggr_relation' which checks whether there is an association or aggregation relation linking the two classes directly or through a superclass.

value
exists_assoc_aggr_relation :
G.Variable_Name $\times$ G.Class_Name $\times$ G.Class_Name $\times$ Wf_Relations $\rightarrow$ **Bool**
exists_assoc_aggr_relation(v, c_1, c_2, rs) $\equiv$
have_direct_assoc_aggr_rel(v, c_1, c_2, rs) $\vee$
exists_assoc_aggr_relation_in_superclass(v, c_1, c_2, rs)

The function 'is_assoc_aggr_relation_in_superclass' is similar to 'is_assoc_aggr_between' except that it checks whether a relation r is an association or aggregation relation whose name, sink class and source class are v, c_2 and some superclass of c_1 respectively.

value
is_assoc_aggr_relation_in_superclass :
G.Class_Name $\times$ G.Class_Name $\times$ G.Variable_Name $\times$ Wf_Relations $\times$ Wf_Relation
$\rightarrow$
Bool
is_assoc_aggr_relation_in_superclass(c_1, c_2, v, rs, r) $\equiv$
r $\in$ rs $\wedge$
is_superclass(source_class(r), c_1, rs) $\wedge$
sink_class(r) = c_2 $\wedge$
is_assoc_or_aggr(r) $\wedge$ vble_of_assoc_aggr(r) = v

The actual association or aggregation relation with a given name linking two given classes either directly or through a superclass of the first is then given by the function 'assoc_aggr_relation'. The precondition ensures that the relation exists.

value
assoc_aggr_relation :
G.Class_Name $\times$ G.Class_Name $\times$ G.Variable_Name $\times$ Wf_Relations $\xrightarrow{\sim}$ Wf_Relation
assoc_aggr_relation(c_1, c_2, v, rs) **as** r_1
post
$r_1 \in$ rs $\wedge$

```
        if have_direct_assoc_aggr_rel(v, c1, c2, rs) then
            is_assoc_aggr_between(v, c1, c2, r1)
        else
            is_assoc_aggr_relation_in_superclass(c1, c2, v, rs, r1)
        end
    pre exists_assoc_aggr_relation(v, c1, c2, rs)
```

The function 'no_relation' requires that there should be no relation in a given set of relations which links classes c_1 and c_2, that is whose source is c_1 and whose sink is c_2.

```
value
    no_relation : G.Class_Name × G.Class_Name × Wf_Relations → Bool
    no_relation(c1, c2, rs) ≡
        ~ (
            ∃ r : Wf_Relation • r ∈ rs ∧ source_class(r) = c1 ∧ sink_class(r) = c2
        )
```

The function 'exists_inst_relation' checks whether there is an instantiation relation in a given set of relations whose sink class is c_2 and whose source class is c_1 or some superclass of c_1.

```
value
    exists_inst_relation :
        G.Class_Name × G.Class_Name × Wf_Relations → Bool
    exists_inst_relation(c1, c2, rs) ≡
        (
            ∃ r : Wf_Relation •
                r ∈ rs ∧
                relation_type(r) = instantiation ∧
                sink_class(r) = c2 ∧
                (
                    source_class(r) = c1 ∨
                    (
                        ∃ c3 : G.Class_Name •
                            is_superclass(c3, c1, rs) ∧ source_class(r) = c3
                    )
                )
        )
```

The final function in this section, 'vble_many_in_rel', checks whether a given set of relations contains an association or aggregation relation with given name and source class and with cardinality many.

value
vble_many_in_rel : G.Variable_Name × G.Class_Name × Wf_Relations → **Bool**
vble_many_in_rel(vd, c, dr) ≡
(
∃ r : Wf_Relation •
r ∈ dr ∧ is_assoc_or_aggr(r) ∧ vble_of_assoc_aggr(r) = vd ∧
source_class(r) = c ∧ sink_card(r) = G.many
)

3.5 Design Structure (DS)

A design as a whole then simply consists of a collection of classes and a collection of relations, the collection of classes being represented by the type 'Classes' defined in Section 3.3 and the collection of relations by the type 'Wf_Relations' defined in Section 3.4. A design is therefore modelled as a simple Cartesian product of these two types.

type
Design_Structure = C.Classes × R.Wf_Relations

There are of course many constraints that must apply to this combination of classes and relations in order for it to correctly model an object-oriented design.

The first constraint requires that both the source and the sink class of every relation in the collection of relations must be in the collection of classes. It is captured by the function 'is_defined_class'.

value
is_defined_class : Design_Structure → **Bool**
is_defined_class(c, r) ≡
(
∀ e : R.Wf_Relation •
e ∈ r ⇒ R.source_class(e) ∈ **dom** c ∧ R.sink_class(e) ∈ **dom** c
)

The second constraint relates to the inheritance of state variables: if a state variable v is declared locally in a class c then it cannot be declared in or inherited by any parent class of c. The function 'has_state_var' checks whether a given state variable is defined (i.e. either declared locally or inherited) in a given class, and the constraint as a whole is embodied in the function 'correct_state_hierarchy'.

```
value
  correct_state_hierarchy : Design_Structure → Bool
  correct_state_hierarchy(dsc, dsr) ≡
    (
      ∀ c, cp : G.Class_Name, v : G.Variable_Name •
        c ∈ dom dsc ∧
        v ∈ C.class_state(dsc(c)) ∧ R.is_parent(cp, c, dsr) ⇒
        ∼ has_state_var(v, cp, (dsc, dsr))
    ),

  has_state_var :
    G.Variable_Name × G.Class_Name × Design_Structure → Bool
  has_state_var(v, c, (dsc, dsr)) ≡
    c ∈ dom dsc ∧ v ∈ C.class_state(dsc(c)) ∨
    (
      ∃ c₂ : G.Class_Name •
        R.is_parent(c₂, c, dsr) ∧ has_state_var(v, c₂, (dsc, dsr))
    )
```

The third constraint extends the above to the case of multiple inheritance, when we must additionally rule out the possibility that a class inherits the same state variable from two superclasses which are not themselves related by inheritance. This is expressed using the function 'correct_state_hierarchy_multiple' which requires that any given state variable is defined in at most one parent of any class.

```
value
  correct_state_hierarchy_multiple : Design_Structure → Bool
  correct_state_hierarchy_multiple(dsc, dsr) ≡
    (
      ∀ c, cp₁, cp₂ : G.Class_Name, v : G.Variable_Name •
        R.is_parent(cp₁, c, dsr) ∧
        R.is_parent(cp₂, c, dsr) ∧
        cp₁ ≠ cp₂ ∧ has_state_var(v, cp₁, (dsc, dsr)) ⇒
        ∼ has_state_var(v, cp₂, (dsc, dsr))
    )
```

In fact a similar property applies to methods: any given method can also be defined in at most one parent of any class. (But note that we do not rule out the case in which a method is defined both in a superclass and a subclass because this situation is allowed and indeed is common in object-oriented systems: the method in the subclass simply overrides the method in the superclass.) The function 'has_method' determines whether a given method, identified by its name, is defined in a given class, where again this includes not only the possibility that the

method is declared locally but also that it is inherited from a superclass. The constraint is then captured by the function 'correct_method_hierarchy_multiple' which is essentially analogous to the function 'correct_state_hierarchy_multiple' defined immediately above.

value
correct_method_hierarchy_multiple : Design_Structure → **Bool**
correct_method_hierarchy_multiple(dsc, dsr) ≡
(
∀ c, cp_1, cp_2 : G.Class_Name, m : G.Method_Name •
R.is_parent(cp_1, c, dsr) ∧
R.is_parent(cp_2, c, dsr) ∧
$cp_1 \neq cp_2$ ∧ has_method(m, cp_1, (dsc, dsr)) ⇒
∼ has_method(m, cp_2, (dsc, dsr))
),

has_method : G.Method_Name × G.Class_Name × Design_Structure → **Bool**
has_method(m, c, (dsc, dsr)) ≡
c ∈ **dom** dsc ∧ C.method_name_in_class(m, dsc(c)) ∨
(
∃ c_2 : G.Class_Name •
R.is_parent(c_2, c, dsr) ∧ has_method(m, c_2, (dsc, dsr))
)

The function 'correct_multiple_inheritance' simply combines the two previous constraints.

value
correct_multiple_inheritance : Design_Structure → **Bool**
correct_multiple_inheritance(ds) ≡
correct_state_hierarchy_multiple(ds) ∧ correct_method_hierarchy_multiple(ds)

The next two constraints also apply to the inheritance of methods. The first, which is represented by the function 'not_allowed', states that if a method is implemented in some class it cannot be abstract in any subclass of that class, while the second is captured by the function 'is_impl_error_interf_inherited' and states that if a given method is declared locally both in a superclass and in a subclass and the superclass declares it as an error method then the subclass must also declare it as an error method.

value
not_allowed : Design_Structure → **Bool**
not_allowed(c, r) ≡
(

```
        ∀ c1 : G.Class_Name, m : G.Method_Name •
          c1 ∈ dom c ∧
          C.method_name_in_class(m, c(c1)) ∧
          M.is_implemented(C.class_methods(c(c1))(m)) ⇒
            ∼
              (
                ∃ c2 : G.Class_Name •
                  c2 ∈ dom c ∧
                  R.is_superclass(c1, c2, r) ∧
                  C.method_name_in_class(m, c(c2)) ∧
                  M.body(C.class_methods(c(c1))(m)) = M.defined
              )
    ),

  is_impl_error_interf_inherited : Design_Structure → Bool
  is_impl_error_interf_inherited(c, r) ≡
    (
      ∀ c1, c2 : G.Class_Name, m : G.Method_Name •
        c1 ∈ dom c ∧
        c2 ∈ dom c ∧
        C.method_name_in_class(m, c(c1)) ∧
        C.method_name_in_class(m, c(c2)) ∧
        M.body(C.class_methods(c(c1))(m)) = M.error ∧
        R.is_superclass(c1, c2, r) ⇒
          M.body(C.class_methods(c(c2))(m)) = M.error
    )
```

The next constraint requires that the sink of an instantiation relation cannot be an abstract class. This is because it is not allowed to build objects belonging to abstract classes. The function 'no_abstract_class_instantiated' captures this constraint.

```
value
  no_abstract_class_instantiated : Design_Structure → Bool
  no_abstract_class_instantiated(c, r) ≡
    (
      ∀ e : G.Class_Name •
        e ∈ dom c ∧ C.is_abstract_class(c(e)) ⇒
          ∼
            (
              ∃ d : R.Wf_Relation •
                d ∈ r ∧
                R.relation_type(d) = R.instantiation ∧ R.sink_class(d) = e
            )
    )
```

For the same reason, an abstract class must have at least one subclass. Alternatively, it must be the source of at least one inheritance relation in the design. This is expressed by the function 'exist_inheritance'. Note that this property effectively implies that all leaf classes must be concrete since otherwise they would have to have at least one subclass and therefore would not be leaf classes.

```
value
  exist_inheritance : Design_Structure → Bool
  exist_inheritance(c, r) ≡
    (
      ∀ e : G.Class_Name •
        e ∈ dom c ∧ C.is_abstract_class(c(e)) ⇒
          (
            ∃ d : R.Wf_Relation •
              d ∈ r ∧
              R.relation_type(d) = R.inheritance ∧ R.source_class(d) = e
          )
    )
```

A class that is declared as concrete must actually be concrete and therefore no methods in the class, including inherited methods, can be abstract – if a class has an abstract method the class must also be abstract. We define the auxiliary function 'inherits_defined_method' to check whether a class inherits an abstract method – it does if there is some method which is not declared locally but there is some parent class in which the method is declared as abstract or which inherits the abstract method. Then a class has no abstract methods (the function 'is_concrete_class') if no local methods are abstract and if no abstract methods are inherited. The constraint, 'verifying_concreteness', then states that every class whose type is concrete must satisfy this property 'is_concrete_class'.

```
value
  verifying_concreteness : Design_Structure → Bool
  verifying_concreteness(c, r) ≡
    (
      ∀ c₁ : G.Class_Name •
        c₁ ∈ dom c ∧ C.class_type(c(c₁)) = G.concrete ⇒
          is_concrete_class(c₁, (c, r))
    ),

  is_concrete_class : G.Class_Name × Design_Structure → Bool
  is_concrete_class(c₁, (c, r)) ≡
    c₁ ∈ dom c ∧
    (
      ∀ m : M.Method • C.method_in_class(m, c(c₁)) ⇒ ∼ M.is_defined(m)
```

```
    ) ∧
    ∼ inherits_defined_method(c1, (c, r)),

inherits_defined_method : G.Class_Name × Design_Structure → Bool
inherits_defined_method(c, (dsc, dsr)) ≡
    c ∈ dom dsc ∧
    (
        ∃ cp : G.Class_Name, m : G.Method_Name •
            ∼ C.method_name_in_class(m, dsc(c)) ∧
            R.is_parent(cp, c, dsr) ∧
            (
                cp ∈ dom dsc ∧
                C.method_name_in_class(m, dsc(cp)) ∧
                M.is_defined(C.class_methods(dsc(cp))(m)) ∨
                inherits_defined_method(cp, (dsc, dsr))
            )
    )
```

The next constraint 'valid_vble_used' states that every variable used in the body of an implemented method as a parameter of an instantiation or invocation or on the right-hand side of an assignment in the variable change map must be a state variable, a parameter of the method or a local variable. A variable is local (the auxiliary function 'is_local_variable') if it appears on the left-hand side of an assignment in the variable change map and it is not a state variable or a formal parameter.

```
value
    valid_vble_used : Design_Structure → Bool
    valid_vble_used(dsc, dsr) ≡
        (
            ∀ c : G.Class_Name, m : M.Method, v : G.Variable_Name •
                c ∈ dom dsc ∧
                C.method_in_class(m, dsc(c)) ∧
                M.is_implemented(m) ∧
                (
                    (
                        ∃ vs : M.Variables •
                            M.Request_or_Var_from_Variables(vs) ∈ rng
                                M.variable_change_body(m) ∧
                            v ∈ vs
                    ) ∨
                    (
                        ∃ inv : M.Invocation •
                            M.invocation_in_request_list(inv, m) ∧
                            v ∈ elems M.a_params(M.call_sig(inv))
```

```
            ) ∨
            (
               ∃ ins : M.Instantiation •
                  M.instantiation_in_request_list(ins, m) ∧
                  v ∈ elems M.a_params(ins)
            )
         ) ⇒
            has_state_var(v, c, (dsc, dsr)) ∨
            G.parameter_in_set(v, M.f_params(m)) ∨
            is_local_variable(v, m, c, (dsc, dsr))
   ),

   is_local_variable :
      G.Variable_Name × M.Method × G.Class_Name × Design_Structure → Bool
   is_local_variable(v, m, c, ds) ≡
      M.is_implemented(m) ∧
      v ∈ M.changed_variables(m) ∧
      ~ has_state_var(v, c, ds) ∧
      ~ G.parameter_in_set(v, M.f_params(m))
```

The function 'is_correct_design_class' simply combines the preceding four constraints.

```
value
   is_correct_design_class : Design_Structure → Bool
   is_correct_design_class(ds) ≡
      no_abstract_class_instantiated(ds) ∧
      exist_inheritance(ds) ∧
      verifying_concreteness(ds) ∧ valid_vble_used(ds)
```

The next constraint 'is_implemented_signature' requires that every method in the design which is declared as abstract in some class c_1 must eventually have a concrete (i.e. error or implemented) version in all lower branches of the class hierarchy. The existence of the concrete version of the method in the subclass hierarchies is specified using the auxiliary function 'is_implemented_signature_in_subclass'. Note that the constraint 'verifying_concreteness' implies that the class containing the abstract method must be abstract, from which it follows by the constraint 'exist_inheritance' that the class must have at least one subclass. The method must therefore be defined concretely in all leaf classes in the class hierarchy, though it may additionally have concrete definitions in intermediate classes in the hierarchy.

```
value
   is_implemented_signature : Design_Structure → Bool
   is_implemented_signature(c, r) ≡
```

```
    (
      ∀ c1 : G.Class_Name, m : G.Method_Name •
        c1 ∈ dom c ∧
        C.method_name_in_class(m, c(c1)) ∧
        M.is_defined(C.class_methods(c(c1))(m)) ⇒
          is_implemented_signature_in_subclass(m, c1, (c, r))
    ),

  is_implemented_signature_in_subclass :
    G.Method_Name × G.Class_Name × Design_Structure → Bool
  is_implemented_signature_in_subclass(m, sup, (c, r)) ≡
    (
      ∀ d : R.Wf_Relation •
        d ∈ r ∧
        R.relation_type(d) = R.inheritance ∧ R.source_class(d) = sup ⇒
          (
            C.method_name_in_class(m, c(R.sink_class(d))) ∧
            ∼ M.is_defined(C.class_methods(c(R.sink_class(d)))(m))
          ) ∨
          is_implemented_signature_in_subclass(m, R.sink_class(d), (c, r))
    )
```

The next set of constraints deal with the consistency of invocations. We first introduce some auxiliary functions which help with their formulation.

First, the function 'is_impl_method' checks whether the method invoked in a signature is implemented in a given class, where the method could be implemented locally or inherited from a superclass in which it is implemented. The related function 'is_impl_method_in_superclass' deals with the case of inheritance from a superclass – there must be some parent class in which the method is implemented.

```
value
  is_impl_method :
    G.Class_Name × M.Actual_Signature × Design_Structure → Bool
  is_impl_method(e, a, (c, r)) ≡
    e ∈ dom c ∧
    C.sig_invok_in_class(c(e), a) ∧
    M.is_implemented(C.class_methods(c(e))(M.meth_name(a))) ∨
    is_impl_method_in_superclass(e, a, (c, r)),

  is_impl_method_in_superclass :
    G.Class_Name × M.Actual_Signature × Design_Structure → Bool
  is_impl_method_in_superclass(e, a, (c, r)) ≡
    (
```

```
        ∃ cd : G.Class_Name •
            R.is_parent(cd, e, r) ∧ is_impl_method(cd, a, (c, r))
    )
```

The next pair of auxiliary functions 'exist_method' and 'exist_method_in_subclass' effectively check whether a particular class will be able to respond to the receipt of a particular signature, that is the method belongs to the implemented interface of the class so the class will be able to execute it. The method should either be implemented locally or it can be implemented in the subclasses of the class if the class is abstract (because in this case no objects belonging to the class itself can be created and all instances will in fact be instances of concrete subclasses).

```
value
    exist_method :
        G.Class_Name × M.Actual_Signature × Design_Structure → Bool
    exist_method(e, a, (c, r)) ≡
        e ∈ dom c ∧
        (
            is_impl_method(e, a, (c, r)) ∨
            C.is_abstract_class(c(e)) ∧ exist_method_in_subclass(e, a, (c, r))
        ),

    exist_method_in_subclass :
        G.Class_Name × M.Actual_Signature × Design_Structure → Bool
    exist_method_in_subclass(e, a, (c, r)) ≡
        (
            ∀ d : R.Wf_Relation •
                let s = R.sink_class(d), n = M.meth_name(a) in
                    d ∈ r ∧
                    s ∈ dom c ∧
                    R.relation_type(d) = R.inheritance ∧ R.source_class(d) = e ⇒
                    C.sig_invok_in_class(c(s), a) ∧
                    M.is_implemented(C.class_methods(c(s))(n)) ∨
                    exist_method_in_subclass(s, a, (c, r))
                end
        )
```

The function 'exist_method_with_res' checks whether the response by a class to the invocation of a particular signature will generate a result. The method invoked by the signature could be declared locally and return a result directly, or it could be inherited from a superclass, or the local class could be abstract and the method with the method being declared in all subclasses as returning a result.

```
value
```

```
exist_method_with_res :
   G.Class_Name × M.Actual_Signature × Design_Structure → Bool
exist_method_with_res(e, a, (c, r)) ≡
   e ∈ dom c ∧ C.sig_has_result(c(e), a) ∨
   (
      ∃ c1 : G.Class_Name •
         c1 ∈ dom c ∧
         C.sig_has_result(c(c1), a) ∧
         (
            R.is_superclass(c1, e, r) ∨
            (
               C.is_abstract_class(c(e)) ∧
               exist_method_in_subclass(e, a, (c, r)) ∧
               ∼
                  (
                     ∃ c2 : G.Class_Name •
                        R.is_superclass(e, c2, r) ∧
                        ∼ C.sig_has_result(c(c2), a)
                  )
            )
         )
   )
```

In the case where a particular method is defined in a particular class c, either locally or via inheritance from a superclass, the function 'class_of_method' returns the class which contains the definition of the method as seen by the class c. Of course if the method is declared locally in the class c then the result of this function is the class c. Otherwise, an arbitrary parent of c in which the method is also defined is chosen and the function recurses. Note that the well-formedness constraint 'correct_method_hierarchy_multiple' on multiple inheritance of methods ensures that in fact there is only one parent class in which the method can be defined, so the result of the function is in fact unique.

```
value
   class_of_method :
      G.Method_Name × G.Class_Name × Design_Structure ~→ G.Class_Name
   class_of_method(m, c, (dsc, dsr)) ≡
      if c ∈ dom dsc ∧ C.method_name_in_class(m, dsc(c)) then
         c
      else
         let
            c2 : G.Class_Name •
               R.is_parent(c2, c, dsr) ∧ has_method(m, c2, (dsc, dsr))
         in
```

```
        class_of_method(m, c2, (dsc, dsr))
    end
end
pre has_method(m, c, (dsc, dsr))
```

Allied to the above, the function 'method_of' returns the actual definition of the method as seen by the class c. It uses the above function to determine the class in which the method is actually declared, then simply returns the definition of the method which is local to that class.

```
value
  method_of :
    G.Class_Name × G.Method_Name × Design_Structure ~→ M.Method
  method_of(c, m, (dsc, dsr)) ≡
    let cd = class_of_method(m, c, (dsc, dsr)) in
      C.class_methods(dsc(cd))(m)
    end
    pre has_method(m, c, (dsc, dsr))
```

The first constraint on invocations is that all *self invocations* must be well-defined, that is if the body of one method in some class contains an invocation of another method in the same class this second method must belong to the implemented interface of the class. This property is captured by the function 'is_correct_self_invocation'.

```
value
  is_correct_self_invocation : Design_Structure → Bool
  is_correct_self_invocation(c, r) ≡
    (
      ∀ c1 : G.Class_Name, m : M.Method, inv : M.Invocation •
        c1 ∈ dom c ∧
        C.method_in_class(m, c(c1)) ∧
        M.invocation_in_request_list(inv, m) ∧
        M.call_vble(inv) = G.self ⇒
          exist_method(c1, M.call_sig(inv), (c, r))
    )
```

A similar property holds for *super invocations*, where the local method contains an invocation of a method in the interface of a parent class. In this case the second method must be implemented in a superclass of the current class. This property is captured by the function 'is_correct_super_invocation'.

```
value
  is_correct_super_invocation : Design_Structure → Bool
  is_correct_super_invocation(c, r) ≡
    (
      ∀ c1 : G.Class_Name, m : M.Method, inv : M.Invocation •
        c1 ∈ dom c ∧
        C.method_in_class(m, c(c1)) ∧
        M.invocation_in_request_list(inv, m) ∧
        M.call_vble(inv) = G.super ⇒
          is_impl_method_in_superclass(c1, M.call_sig(inv), (c, r))
    )
```

The next two functions 'signature_in_receiver' and 'signature_in_receiver_parameter' deal with an invocation to a method in another class, addressing the cases in which the variable representing the receiver of the invocation (the *call variable*) respectively is not and is a formal parameter of the method containing the invocation. In 'signature_in_receiver', there should be either an association or an aggregation relation with the same name as the call variable, and the method invoked must exist in the sink class of this relation. Note that we do not consider the case in which the invoked method is one of the reserved methods for manipulating collections since the classes to which these methods belong are unspecified. In 'signature_in_receiver_parameter' we can only impose a constraint if the type of the parameter (i.e. the class to which the object it represents belongs) which forms the call variable is defined in the signature of the method. Then the constraint requires that the method invoked must exist in that class.

```
value
  signature_in_receiver : Design_Structure → Bool
  signature_in_receiver(c, r) ≡
    (
      ∀ m : M.Method, e : G.Class_Name, inv : M.Invocation •
        e ∈ dom c ∧
        C.method_in_class(m, c(e)) ∧
        M.invocation_in_request_list(inv, m) ∧
        G.not_self_super(M.call_vble(inv)) ∧
        ∼ G.is_collection_method(M.meth_name(M.call_sig(inv))) ∧
        ∼ G.parameter_in_set(M.call_vble(inv), M.f_params(m)) ⇒
          (
            ∃ e1 : G.Class_Name •
              e1 ∈ dom c ∧
              R.exists_assoc_aggr_relation(M.call_vble(inv), e, e1, r) ∧
              exist_method(e1, M.call_sig(inv), (c, r))
          )
    ),

  signature_in_receiver_parameter : Design_Structure → Bool
```

```
signature_in_receiver_parameter(c, r) ≡
  (
    ∀ m : M.Method, e : G.Class_Name, inv : M.Invocation •
      e ∈ dom c ∧
      C.method_in_class(m, c(e)) ∧
      M.invocation_in_request_list(inv, m) ∧
      G.parameter_in_set(M.call_vble(inv), M.f_params(m)) ∧
      G.var(M.call_vble(inv)) ∉ elems M.f_params(m) ⇒
        let e1 = G.rol_of_parameter(M.call_vble(inv), M.f_params(m)) in
          exist_method(e1, M.call_sig(inv), (c, r))
        end
  )
```

The next constraint deals with invocations of the collection manipulation methods. Here the invocation must have one actual parameter and the call variable must represent an aggregation or association relation whose source class is the local class and whose cardinality is many.

```
value
  correct_inv_collection : Design_Structure → Bool
  correct_inv_collection(c, r) ≡
    (
      ∀ m : M.Method, c1 : G.Class_Name, inv : M.Invocation •
        c1 ∈ dom c ∧
        C.method_in_class(m, c(c1)) ∧
        M.invocation_in_request_list(inv, m) ∧
        G.is_collection_method(M.meth_name(M.call_sig(inv))) ⇒
          R.vble_many_in_rel(M.call_vble(inv), c1, r) ∧
          len M.a_params(M.call_sig(inv)) = 1
    )
```

Next, every invocation that appears on the right-hand side of some assignment in the variable change map must return a result. Again we consider the two cases in which the call variable of the invocation is not, respectively is a formal parameter of the containing method. In the first case, which is represented by the function 'signature_with_result', we again require that the call variable represents an association or aggregation relation and the method invoked returns a result in the sink class of that relation, and in the second, which is represented by the function 'signature_with_result_param', the type (class) of the parameter must again be declared in the signature and the method invoked must return a result in that class.

```
value
  signature_with_result : Design_Structure → Bool
  signature_with_result(c, r) ≡
```

```
    (
      ∀ m : M.Method, c1 : G.Class_Name, inv : M.Invocation •
        c1 ∈ dom c ∧
        C.method_in_class(m, c(c1)) ∧
        M.invocation_in_vble_change(inv, m) ∧
        ~ G.is_collection_method(M.meth_name(M.call_sig(inv))) ∧
        ~ G.parameter_in_set(M.call_vble(inv), M.f_params(m)) ⇒
          (
            ∃ c2 : G.Class_Name •
              c2 ∈ dom c ∧
              R.exists_assoc_aggr_relation(M.call_vble(inv), c1, c2, r) ∧
              exist_method_with_res(c2, M.call_sig(inv), (c, r))
          )
    ),

  signature_with_result_param : Design_Structure → Bool
  signature_with_result_param(c, r) ≡
    (
      ∀ m : M.Method, c1 : G.Class_Name, inv : M.Invocation •
        c1 ∈ dom c ∧
        C.method_in_class(m, c(c1)) ∧
        M.invocation_in_vble_change(inv, m) ∧
        G.parameter_in_set(M.call_vble(inv), M.f_params(m)) ∧
        G.var(M.call_vble(inv)) ∉ elems M.f_params(m) ⇒
          let c2 = G.rol_of_parameter(M.call_vble(inv), M.f_params(m)) in
            exist_method_with_res(c2, M.call_sig(inv), (c, r))
        end
    )
```

The last constraint on invocations is basically the converse of the above, namely that any invocation in the body of the method that returns a result must appear on the right-hand side of some assignment in the variable change map. This constraint is not strictly necessary but is included since otherwise the results of the invocations are lost and the invocations are then to a large extent redundant. Again, we consider separately the two cases in which the call variable of the invocation is not, respectively is a formal parameter of the containing method, these being represented by the two functions 'correct_inv_res' and 'correct_inv_res_param'. Note that the first case includes invocations to the method 'collectionelement' and both cases additionally check that assignment is to the correct number of variables.

```
value
  correct_inv_res : Design_Structure → Bool
  correct_inv_res(c, r) ≡
    (
      ∀ m : M.Method, c1, c2 : G.Class_Name, inv : M.Invocation •
```

```
            let M.mk_Invocation(v, s) = inv, mn = M.meth_name(s) in
               c1 ∈ dom c ∧
               c2 ∈ dom c ∧
               C.method_in_class(m, c(c1)) ∧
               M.invocation_in_request_list(inv, m) ∧
               (
                  mn = G.collectionelement ∨
                  (
                     ∼ G.parameter_in_set(v, M.f_params(m)) ∧
                     R.exists_assoc_aggr_relation(v, c1, c2, r) ∧
                     exist_method_with_res(c2, s, (c, r))
                  )
               ) ⇒
                  let
                     m2 = method_of(c2, mn, (c, r)),
                     n = card M.meth_res(m2),
                     vm = M.variable_change_body(m)
                  in
                     (
                        ∃ vs : G.Variable_Name-set •
                           vs ∈ dom vm ∧ card vs = n ∧ vm(vs) = inv
                     )
                  end
            end
      ),

   correct_inv_res_param : Design_Structure → Bool
   correct_inv_res_param(c, r) ≡
      (
         ∀ m : M.Method, c1 : G.Class_Name, inv : M.Invocation •
            let M.mk_Invocation(v, s) = inv, mn = M.meth_name(s) in
               c1 ∈ dom c ∧
               C.method_in_class(m, c(c1)) ∧
               M.invocation_in_request_list(inv, m) ∧
               G.parameter_in_set(M.call_vble(inv), M.f_params(m)) ∧
               G.var(M.call_vble(inv)) ∉ elems M.f_params(m) ⇒
                  let
                     c2 = G.rol_of_parameter(v, M.f_params(m)),
                     m2 = method_of(c2, mn, (c, r)),
                     n = card M.meth_res(m2),
                     vm = M.variable_change_body(m)
                  in
                     exist_method_with_res(c2, s, (c, r)) ⇒
                        (
                           ∃ vs : G.Variable_Name-set •
```

```
                        vs ∈ dom vm ∧ card vs = n ∧ vm(vs) = inv
                )
            end
        end
    )
```

All these constraints on invocations are then combined into the function 'is_correct_invocation'.

```
value
  is_correct_invocation : Design_Structure → Bool
  is_correct_invocation(ds) ≡
    is_correct_self_invocation(ds) ∧
    is_correct_super_invocation(ds) ∧
    signature_in_receiver(ds) ∧
    signature_in_receiver_parameter(ds) ∧
    correct_inv_collection(ds) ∧
    signature_with_result(ds) ∧
    signature_with_result_param(ds) ∧
    correct_inv_res(ds) ∧ correct_inv_res_param(ds)
```

The next constraints deal with properties of relations. First, for every instantiation in the body of some method in a class, there must be an instantiation relation linking that class to the class in the instantiation unless the two classes are the same. This property is captured by the function 'is_rqst_instantiation'.

```
value
  is_rqst_instantiation : Design_Structure → Bool
  is_rqst_instantiation(c, r) ≡
    (
      ∀ e : G.Class_Name, m : M.Method, inst : M.Instantiation •
        e ∈ dom c ∧
        C.method_in_class(m, c(e)) ∧
        M.instantiation_in_request_list(inst, m) ∧
        M.class_name(inst) ≠ e ⇒
          R.exists_inst_relation(e, M.class_name(inst), r)
    )
```

Next, the name of an association or aggregation relation cannot be the same as the name of any formal parameter of any method in the source class of the relation or in any of its subclasses. The source class itself is dealt with in the function 'is_correct_name_relation' and the subclasses in 'is_correct_name_rel_in_subclass'.

```
value
  is_correct_name_relation : Design_Structure → Bool
  is_correct_name_relation(c, r) ≡
    (
      ∀ d : R.Wf_Relation, m : M.Method •
        d ∈ r ∧
        R.is_assoc_or_aggr(d) ∧
        R.source_class(d) ∈ dom c ∧
        C.method_in_class(m, c(R.source_class(d))) ⇒
          ∼ G.parameter_in_set(R.vble_of_assoc_aggr(d), M.f_params(m))
    ),

  is_correct_name_rel_in_subclass : Design_Structure → Bool
  is_correct_name_rel_in_subclass(c, r) ≡
    (
      ∀ d : R.Wf_Relation, m : M.Method, c₂ : G.Class_Name •
        d ∈ r ∧
        R.is_assoc_or_aggr(d) ∧
        R.source_class(d) ∈ dom c ∧
        R.is_superclass(R.source_class(d), c₂, r) ∧
        C.method_in_class(m, c(c₂)) ⇒
          ∼ G.parameter_in_set(R.vble_of_assoc_aggr(d), M.f_params(m))
    )
```

Finally, there are some constraints on the results and formal parameters of methods. The first of these, which is captured by the function 'is_consistent_result', requires that if a method is declared in a superclass and in a subclass then the result of the method according to the two declarations must be the same except that it is possible for the declaration in the superclass to return a result and the declaration in the subclass to be error.

```
value
  is_consistent_result : Design_Structure → Bool
  is_consistent_result(c, r) ≡
    (
      ∀ m : G.Method_Name, c₁, c₂ : G.Class_Name •
        c₁ ∈ dom c ∧
        c₂ ∈ dom c ∧
        R.is_superclass(c₁, c₂, r) ∧
        C.method_name_in_class(m, c(c₁)) ∧
        C.method_name_in_class(m, c(c₂)) ⇒
          let
            m₁ = C.class_methods(c(c₁))(m), m₂ = C.class_methods(c(c₂))(m)
          in
            (M.meth_res(m₁) = {} ∧ M.meth_res(m₂) = {}) ∨
```

```
            (M.meth_res(m1) ≠ {} ∧ M.meth_res(m2) ≠ {}) ∨
            (M.meth_res(m1) ≠ {} ∧ M.body(m2) = M.error)
        end
    )
```

A similar constraint requires that if a method is declared in a superclass and in a subclass then the formal parameters of the method according to the two declarations must be the same.

```
value
  is_consistent_f_params : Design_Structure → Bool
  is_consistent_f_params(c, r) ≡
    (
      ∀ m : G.Method_Name, c1, c2 : G.Class_Name •
        c1 ∈ dom c ∧
        c2 ∈ dom c ∧
        R.is_superclass(c1, c2, r) ∧
        C.method_name_in_class(m, c(c1)) ∧
        C.method_name_in_class(m, c(c2)) ⇒
          M.f_params(C.class_methods(c(c1))(m)) =
            M.f_params(C.class_methods(c(c2))(m))
    )
```

Finally, the function 'correct_paramTyped_in_f_param' requires that the name of any class which is used to denote the type of any formal parameter in some method in the design must be one of the classes in the design.

```
value
  correct_paramTyped_in_f_param : Design_Structure → Bool
  correct_paramTyped_in_f_param(c, r) ≡
    (
      ∀ c1, c2 : G.Class_Name, m : M.Method, v : G.Variable_Name •
        c1 ∈ dom c ∧
        C.method_in_class(m, c(c1)) ∧
        G.paramTyped(v, c2) ∈ elems M.f_params(m) ⇒
          c2 ∈ dom c
    )
```

These last three constraints are combined in the function 'is_correct_res_f_param', and all constraints on the design are combined in the function 'is_wf_design_structure'. This is then used to construct the subtype 'Wf_Design_Structure' representing well-formed designs in the standard way.

```
value
  is_correct_res_f_param : Design_Structure → Bool
  is_correct_res_f_param(ds) ≡
    is_consistent_result(ds) ∧
    is_consistent_f_params(ds) ∧ correct_paramTyped_in_f_param(ds),

  is_wf_design_structure : Design_Structure → Bool
  is_wf_design_structure(ds) ≡
    is_defined_class(ds) ∧
    correct_state_hierarchy(ds) ∧
    correct_multiple_inheritance(ds) ∧
    not_allowed(ds) ∧
    is_impl_error_interf_inherited(ds) ∧
    is_correct_design_class(ds) ∧
    is_implemented_signature(ds) ∧
    is_correct_invocation(ds) ∧
    is_rqst_instantiation(ds) ∧
    is_correct_name_relation(ds) ∧
    is_correct_name_rel_in_subclass(ds) ∧
    is_correct_res_f_param(ds)

type
  Wf_Design_Structure = {| ds : Design_Structure • is_wf_design_structure(ds) |}
```

We conclude this section by defining a single auxiliary function 'invokes' which represents a generalisation of the possibly indirect invocation as seen, for example, in the invocation of the StoreCommand method by the Client class in the Command pattern (see [13] and the discussion of the Command pattern in [18]). In fact this function checks if a method m in a particular class (called client in the specification) invokes a method mn from a class c_2 on a given variable (called acommand in the specification) which is in fact the nth parameter of the invocation.

The function is defined recursively. In the base case, the invocation is direct, so there is an invocation inv_1 in the body of the method m whose call variable represents an association or aggregation directly linking the two classes concerned and which invokes mn with the correct actual parameter. In the recursive case, there is a method m_3 in an intermediate class c_3 contains a direct invocation of the method mn in the class c_2 and which is itself invoked, possibly indirectly, by the method m. We also take account of the fact that the position of the parameter representing the acommand variable might not be the same in the two invocations: it is the nth parameter of the second (direct) invocation from c_3 to c_2, and we find the position of the variable in the first invocation using the function 'position_of' defined in Section 3.1.

```
value
  invokes :
    M.Method × G.Method_Name × Nat × G.Variable_Name ×
```

```
        G.Class_Name × G.Class_Name × Wf_Design_Structure
    →
        Bool
invokes(m, mn, n, acommand, client, c2, (dsc, dsr)) ≡
    (
        ∃ inv1 : M.Invocation •
            M.invocation_in_request_list(inv1, m) ∧
            R.exists_assoc_aggr_relation(M.call_vble(inv1), client, c2, dsr) ∧
            M.a_params(M.call_sig(inv1))(n) = acommand ∧
            M.meth_name(M.call_sig(inv1)) = mn
    ) ∨
    (
        ∃ c3 : G.Class_Name, m3 : G.Method_Name, inv : M.Invocation, p : G.Parameter •
            R.exists_assoc_aggr_relation(M.call_vble(inv), c3, c2, dsr) ∧
            has_method(m3, c3, (dsc, dsr)) ∧
            let mp = method_of(c3, m3, (dsc, dsr)) in
                M.invocation_in_request_list(inv, mp) ∧
                M.meth_name(M.call_sig(inv)) = mn ∧
                p ∈ elems M.f_params(mp) ∧
                M.a_params(M.call_sig(inv))(n) = G.type_parameter(p) ∧
                let
                    var = M.a_params(M.call_sig(inv))(n),
                    pos = G.position_of(var, M.f_params(mp))
                in
                    invokes(m, m3, pos, acommand, client, c3, (dsc, dsr))
                end
            end
    )
```

4 Linking Designs to Patterns

A pattern represents an abstract "outline" or "skeleton" of a design, and in order to check whether a design matches a particular pattern we need to link the model of a design described and specified above to the design patterns.

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. A typical example of this is shown in Figure 3.

Both the correspondence between state variables and that between parameters involves only a *variable renaming*, which simply links names of variables in the design to those in the pattern. We model this using the type 'VariableRenaming'.

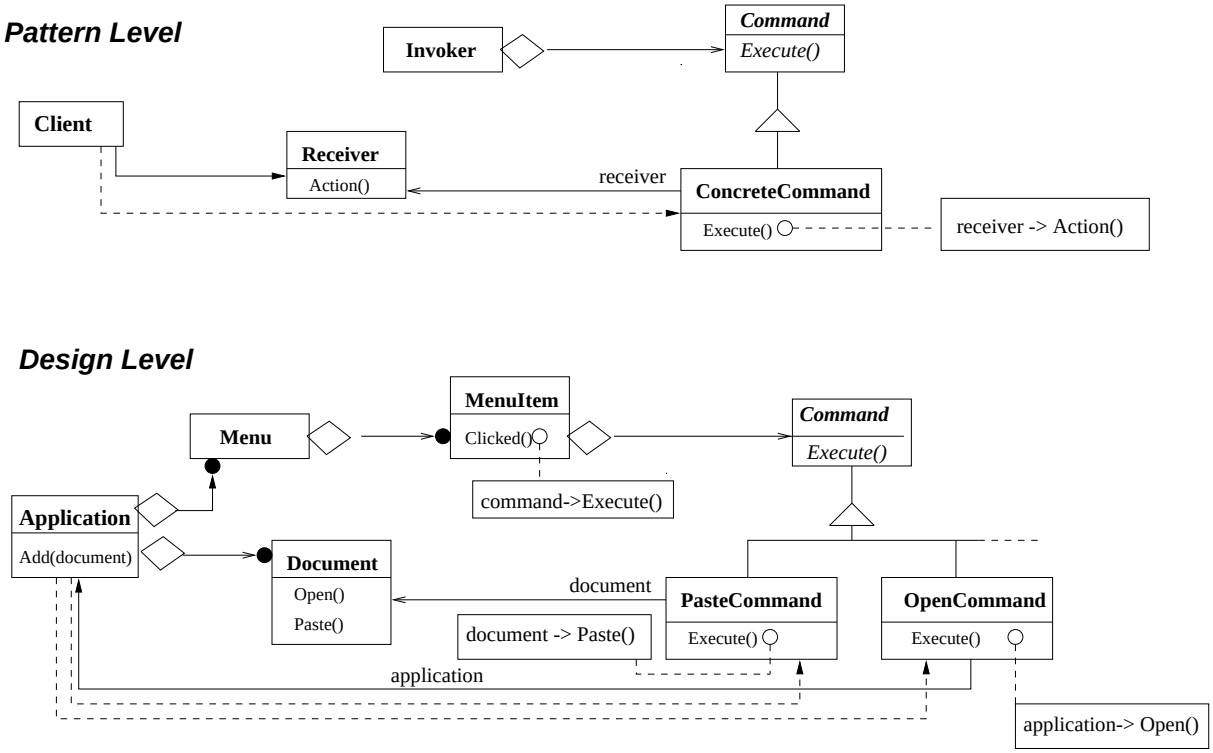

Figure 3: Linking the Design with the Pattern

type
VariableRenaming = G.Variable_Name $\overrightarrow{m}$ G.Variable_Name

The renaming for methods involves two parts, the first of which defines the correspondence between the names of the methods and the second of which relates their parameters. We define the type 'Method_Renaming' to consist of the method name in the pattern together with the variable renaming for the method's parameters. Then the type 'Method_and_Parameter_Renaming' links methods in the design to methods in the pattern by associating a method name with a value of the type 'Method_Renaming'. Note that the nested structure of the renaming is necessary because two different methods may have parameters with the same name.

type
Method_Renaming ::
method_name : G.Method_Name parameterRenaming : VariableRenaming,
Method_and_Parameter_Renaming = G.Method_Name $\overrightarrow{m}$ Method_Renaming

The renaming of a class has a similarly nested structure, the type 'ClassRenaming' 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 [12] 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'.

```
type
  ClassRenaming ::
    classname : G.Class_Name
    methodRenaming : Method_and_Parameter_Renaming
    varRenaming : VariableRenaming,
  Renaming = G.Class_Name -m-> ClassRenaming-set
```

In order for the renaming map to be well-formed, no class in the design can have an empty set of renamings (we model the fact that a class in the design plays no *role* in the pattern by simply omitting it from the domain of the renaming map) and the renamings of any one design class must all refer to different pattern classes (otherwise a single design class can have two contradictory renamings).

We specify these two properties using the functions 'images_not_empty' and 'different_images_class_name' respectively, and these are combined in the function 'is_wf_Renaming', which then forms the defining predicate of the subtype 'Wf_Renaming'.

```
value
  images_not_empty : Renaming → Bool
  images_not_empty(r) ≡ {} ∉ rng r,

  different_images_class_name : Renaming → Bool
  different_images_class_name(r) ≡
    (
      ∀ c : G.Class_Name, cr1, cr2 : ClassRenaming •
        c ∈ dom r ∧ cr1 ≠ cr2 ∧ cr1 ∈ r(c) ∧ cr2 ∈ r(c) ⇒
          classname(cr1) ≠ classname(cr2)
    ),

  is_wf_Renaming : Renaming → Bool
  is_wf_Renaming(r) ≡
    different_images_class_name(r) ∧ images_not_empty(r)

type
  Wf_Renaming = {| r : Renaming • is_wf_Renaming(r) |}
```

Then a design together with a renaming map which defines its correspondences to a given pattern is described by the simple Cartesian product type 'Design_Renaming'.

type
Design_Renaming = DS.Wf_Design_Structure × Wf_Renaming

There are again several consistency conditions that must be satisfied, but before considering those we introduce some useful auxiliary functions on the renaming alone.

The first of these, 'renaming_class_name', checks whether a class cd plays a given role cp under some renaming, and if so the second, 'class_renaming', returns the whole class renaming associated with that role.

value
renaming_class_name : G.Class_Name × G.Class_Name × Wf_Renaming → **Bool**
renaming_class_name(cd, cp, r) ≡
cd ∈ **dom** r ∧
(∃ c : ClassRenaming • c ∈ r(cd) ∧ classname(c) = cp),

class_renaming : G.Class_Name × G.Class_Name × Wf_Renaming $\xrightarrow{\sim}$ ClassRenaming
class_renaming(cd, cp, r) **as** c
post c ∈ r(cd) ∧ classname(c) = cp
pre renaming_class_name(cd, cp, r)

Next, the two functions 'method_renames_to' and 'state_var_renames_to' check respectively whether a method or a state variable plays a given role in a particular class renaming, and the function 'parameter_renames_to' checks whether a parameter plays a given role in a given method in a particular class renaming.

value
method_renames_to : G.Method_Name × ClassRenaming × G.Method_Name → **Bool**
method_renames_to(md, c, mp) ≡
let mr = methodRenaming(c) **in**
md ∈ **dom** mr ∧ method_name(mr(md)) = mp
end,

state_var_renames_to : G.Variable_Name × ClassRenaming × G.Variable_Name → **Bool**
state_var_renames_to(vd, c, vp) ≡
let vr = varRenaming(c) **in** vd ∈ **dom** vr ∧ vr(vd) = vp **end**,

parameter_renames_to :
G.Variable_Name × G.Variable_Name × G.Method_Name × ClassRenaming → **Bool**
parameter_renames_to(vd, vp, md, c) ≡
let mr = methodRenaming(c) **in**
md ∈ **dom** mr ∧

```
        let pr = parameterRenaming(mr(md)) in
          vd ∈ dom pr ∧ pr(vd) = vp
        end
    end
```

Using these functions we define a range of other functions which check that different entities play particular roles in the renaming as a whole. Thus, 'renaming_class_method' checks whether a class plays a particular role under the renaming and some method plays a given role within that class, and 'renaming_class_state' checks whether a class plays a particular role under the renaming and some state variable plays a given role within that class. Similar properties for other combinations of entities (two methods, one method and one state variable, and one method and one parameter of that method) are checked by the other three functions.

```
value
  renaming_class_method :
    G.Class_Name × G.Class_Name × G.Method_Name × G.Method_Name ×
      Wf_Renaming
    →
    Bool
  renaming_class_method(cd, cp, md, mp, r) ≡
    renaming_class_name(cd, cp, r) ∧
    let cr = class_renaming(cd, cp, r) in
      method_renames_to(md, cr, mp)
    end,

  renaming_class_state :
    G.Class_Name × G.Class_Name × G.Variable_Name ×
      G.Variable_Name × Wf_Renaming
    →
    Bool
  renaming_class_state(cd, cp, vd, vp, r) ≡
    renaming_class_name(cd, cp, r) ∧
    let cr = class_renaming(cd, cp, r) in
      state_var_renames_to(vd, cr, vp)
    end,

  renaming_class_method2 :
    G.Class_Name × G.Class_Name × G.Method_Name × G.Method_Name ×
      G.Method_Name × G.Method_Name × Wf_Renaming
    →
    Bool
  renaming_class_method2(cd, cp, md1, mp1, md2, mp2, r) ≡
    renaming_class_method(cd, cp, md1, mp1, r) ∧
    renaming_class_method(cd, cp, md2, mp2, r),
```

```
renaming_class_method_state :
    G.Class_Name × G.Class_Name × G.Method_Name × G.Method_Name ×
        G.Variable_Name × G.Variable_Name × Wf_Renaming
    →
    Bool
renaming_class_method_state(cd, cp, md, mp, vd, vp, r) ≡
    renaming_class_name(cd, cp, r) ∧
    let cr = class_renaming(cd, cp, r) in
        method_renames_to(md, cr, mp) ∧ state_var_renames_to(vd, cr, vp)
    end,

renaming_class_method_param :
    G.Class_Name × G.Class_Name × G.Method_Name × G.Method_Name ×
        G.Variable_Name × G.Variable_Name × Wf_Renaming
    →
    Bool
renaming_class_method_param(cd, cp, md, mp, vd, vp, r) ≡
    renaming_class_name(cd, cp, r) ∧
    let cr = class_renaming(cd, cp, r) in
        method_renames_to(md, cr, mp) ∧
        parameter_renames_to(vd, vp, md, cr)
    end
```

The next set of functions deal with the collection of all entities in a design which play given roles under some renaming: 'method_renaming_to' returns the set of all methods playing a particular role in some class playing a given role, while 'state_vars_renaming_to' similarly returns the set of all state variables playing a particular role in some class playing a given role.

```
value
    method_renaming_to :
        G.Class_Name × G.Class_Name × G.Method_Name × Wf_Renaming
            → G.Method_Name-set
    method_renaming_to(cd, cp, mp, r) ≡
        { md | md : G.Method_Name • renaming_class_method(cd, cp, md, mp, r) },

    state_vars_renaming_to :
        G.Class_Name × G.Class_Name × G.Variable_Name × Wf_Renaming
            → G.Variable_Name-set
    state_vars_renaming_to(cd, cp, vp, r) ≡
        { vd | vd : G.Variable_Name • renaming_class_state(cd, cp, vd, vp, r) }
```

We next define functions which return the number of entities in a design which play given roles

under some renaming: the three functions below deal with classes, methods and state variables respectively.

```
value
  quantity_of_classes : G.Class_Name × Wf_Renaming → Nat
  quantity_of_classes(cp, r) ≡
    card { cd | cd : G.Class_Name • renaming_class_name(cd, cp, r) },

  quantity_of_method : G.Class_Name × G.Method_Name × Wf_Renaming → Nat
  quantity_of_method(cp, mp, r) ≡
  card { md | cd : G.Class_Name, md : G.Method_Name •
                  renaming_class_method(cd, cp, md, mp, r) },

  quantity_of_vble :
    G.Class_Name × G.Class_Name × G.Variable_Name × Wf_Renaming → Nat
  quantity_of_vble(cd, cp, vp, r) ≡ card state_vars_renaming_to(cd, cp, vp, r)
```

The function 'quantities_of_variables' checks whether two classes playing two different roles under the renaming have the same number of state variables playing a given role.

```
value
  quantities_of_variables :
    G.Class_Name × G.Class_Name × G.Variable_Name × Wf_Renaming → Bool
  quantities_of_variables(cp1, cp2, vp, r) ≡
    (
      ∀ cd1, cd2 : G.Class_Name •
        renaming_class_name(cd1, cp1, r) ∧ renaming_class_name(cd2, cp2, r)
          ⇒
        quantity_of_vble(cd1, cp1, vp, r) = quantity_of_vble(cd2, cp2, vp, r)
    )
```

The functions 'has_role_in' and 'exists_method' check respectively whether a class plays some role in a given set of roles and whether a class has some method playing a given role, while the function 'share_role_in' checks whether there is some role in a given set of roles which is played by two given classes.

```
value
  has_role_in : G.Class_Name × G.Class_Name-set × Wf_Renaming → Bool
  has_role_in(cd, cps, r) ≡
    (
      ∃ cp : G.Class_Name • cp ∈ cps ∧ renaming_class_name(cd, cp, r)
```

```
  ),

  exists_method : G.Class_Name × G.Method_Name × Wf_Renaming → Bool
  exists_method(cp, mp, r) ≡
    (
      ∀ cd : G.Class_Name •
        renaming_class_name(cd, cp, r) ⇒
          let cr = class_renaming(cd, cp, r) in
            ∃ md : G.Method_Name • method_renames_to(md, cr, mp)
          end
    ),

  share_role_in :
    G.Class_Name × G.Class_Name × G.Class_Name-set × Wf_Renaming → Bool
  share_role_in(cd1, cd2, cps, r) ≡
    (
      ∃ cp : G.Class_Name •
        cp ∈ cps ∧ renaming_class_name(cd1, cp, r) ∧ renaming_class_name(cd2, cp, r)
    )
```

We now return to the constraints on the type 'Design_Renaming' which represents the combination of a design with a renaming.

The first constraints simply require that every entity which has a renaming under the renaming map must be in the design. This is captured by the function 'is_correct_domain', which is in turn written in terms of the three auxiliary functions 'domain_class_name', 'domain_method_parameter_name' and 'domain_variable_name' which respectively check that the constraint is satisfied by the classes, the methods and their parameters, and the state variables.

```
value
  is_correct_domain : Design_Renaming → Bool
  is_correct_domain(dr) ≡
    domain_class_name(dr) ∧ domain_method_parameter_name(dr) ∧
    domain_variable_name(dr),

  domain_class_name : Design_Renaming → Bool
  domain_class_name((dsc, dsr), r) ≡ dom r ⊆ dom dsc,

  domain_method_parameter_name : Design_Renaming → Bool
  domain_method_parameter_name(ds, r) ≡
    (
      ∀ c : G.Class_Name, cr : ClassRenaming, md : G.Method_Name •
        c ∈ dom r ∧ cr ∈ r(c) ∧ md ∈ dom methodRenaming(cr) ⇒
          DS.has_method(md, c, ds) ∧
```

```
                let m = DS.method_of(c, md, ds) in
                    dom parameterRenaming(methodRenaming(cr)(md)) ⊆
                    G.set_f_params(M.f_params(m))
                end
    ),

  domain_variable_name : Design_Renaming → Bool
  domain_variable_name(ds, r) ≡
    (
      ∀ c : G.Class_Name, cr : ClassRenaming, vd : G.Variable_Name •
        c ∈ dom r ∧ cr ∈ r(c) ∧ vd ∈ dom varRenaming(cr) ⇒
          DS.has_state_var(vd, c, ds)
    )
```

The next constraints relate to the consistency of renamings of methods within a class hierarchy. The first states that if a method plays some role in a superclass and also some role in a subclass then the two roles must be the same, and the second says that if a method plays some role in a superclass and the subclass inherits a different version of the method (i.e. the method is redefined locally or in an intermediate class) then the method must also play a role in the subclass.

```
value
  equal_meth_ren_in_hierarchy : Design_Renaming → Bool
  equal_meth_ren_in_hierarchy((dsc, dsr), r) ≡
    (
      ∀ c₁, c₂ : G.Class_Name, md : G.Method_Name, cr₁, cr₂ : ClassRenaming •
        c₁ ∈ dom r ∧
        c₂ ∈ dom r ∧
        R.is_superclass(c₁, c₂, dsr) ∧
        cr₁ ∈ r(c₁) ∧
        md ∈ dom methodRenaming(cr₁) ∧
        cr₂ ∈ r(c₂) ∧ md ∈ dom methodRenaming(cr₂) ⇒
          methodRenaming(cr₁)(md) = methodRenaming(cr₂)(md)
    ),

  meth_ren_in_hierarchy : Design_Renaming → Bool
  meth_ren_in_hierarchy((dsc, dsr), r) ≡
    (
      ∀ c₁, c₂ : G.Class_Name, md : G.Method_Name, cr₁, cr₂ : ClassRenaming •
        c₁ ∈ dom r ∧
        c₂ ∈ dom r ∧
        R.is_superclass(c₁, c₂, dsr) ∧
        cr₁ ∈ r(c₁) ∧
        cr₂ ∈ r(c₂) ∧
        md ∈ dom methodRenaming(cr₁) ∧
```

```
        DS.class_of_method(md, c2, (dsc, dsr)) ≠ c1 ⇒
          md ∈ dom methodRenaming(cr2)
    )
```

The final constraint applies only to matching designs against GoF patterns and derives from the fact that no method in the GoF patterns has more than one parameter. This constraint, which could be omitted if we wanted to generalise the matching to other forms of patterns, is described by the function 'card_images_of_parameters'.

```
value
  card_images_of_parameters : Design_Renaming → Bool
  card_images_of_parameters((dsc, dsr), r) ≡
    (
      ∀ c : G.Class_Name, m : G.Method_Name, cr : ClassRenaming •
        c ∈ dom r ∧ cr ∈ r(c) ∧ m ∈ dom methodRenaming(cr) ⇒
          card rng parameterRenaming(methodRenaming(cr)(m)) ≤ 1
    )
```

Finally, the constraints are combined together into the function 'is_wf_design_renaming' and this is used to define the subtype 'Wf_Design_Renaming' of 'Design_Renaming' in the usual way.

```
value
  is_wf_design_renaming : Design_Renaming → Bool
  is_wf_design_renaming(dr) ≡
    card_images_of_parameters(dr) ∧
    is_correct_domain(dr) ∧
    equal_meth_ren_in_hierarchy(dr) ∧ meth_ren_in_hierarchy(dr)

type
  Wf_Design_Renaming = {| pr : Design_Renaming • is_wf_design_renaming(pr) |}
```

5 Specifying Properties of Patterns

We conclude by defining a range of functions which capture properties of the GoF patterns in our model. Each basically defines a property that an entity in the design must satisfy if it is to be consistent with the corresponding entity (i.e. the role it plays) in the pattern. Since the section is rather long we divide it into subsections dealing with classes, state variables, methods and relations.

5.1 Specifying Properties of Classes in Patterns

The function 'exists_one' checks that there is a single class in the design which plays a given role in the pattern.

```
value
  exists_one : G.Class_Name × Wf_Design_Renaming → Bool
  exists_one(cp, (ds, r)) ≡
    (∃! cd : G.Class_Name • renaming_class_name(cd, cp, r))
```

Similarly, the function 'exists_role' checks that there is at least one class in the design which plays a given role in the pattern.

```
value
  exists_role : G.Class_Name × Wf_Design_Renaming → Bool
  exists_role(cp, (ds, r)) ≡
    (∃ cd : G.Class_Name • renaming_class_name(cd, cp, r))
```

The functions 'is_abstract_class' and 'is_concrete_class' check that all classes in the design which play a given role are abstract, respectively concrete.

```
value
  is_abstract_class : G.Class_Name × Wf_Design_Renaming → Bool
  is_abstract_class(cp, ((dsc, dsr), r)) ≡
    (
      ∀ cd : G.Class_Name •
        renaming_class_name(cd, cp, r) ⇒ C.is_abstract_class(dsc(cd))
    ),

  is_concrete_class : G.Class_Name × Wf_Design_Renaming → Bool
  is_concrete_class(cp, ((dsc, dsr), r)) ≡
    (
      ∀ cd : G.Class_Name •
        renaming_class_name(cd, cp, r) ⇒ C.is_concrete_class(dsc(cd))
    )
```

The function 'is_concrete' checks that every class in the design which plays the role cp_2 is a concrete subclass of every class which plays the role cp_1. It is mainly useful in cases where there is a unique class playing the role cp_1 so it is generally used in conjunction with the function 'exists_one'.

```
value
  is_concrete : G.Class_Name × G.Class_Name × Wf_Design_Renaming → Bool
  is_concrete(cp1, cp2, ((dsc, dsr), r)) ≡
    (
      ∀ cd1, cd2 : G.Class_Name •
        renaming_class_name(cd1, cp1, r) ∧ renaming_class_name(cd2, cp2, r) ⇒
          R.is_superclass(cd1, cd2, dsr) ∧ DS.is_concrete_class(cd2, (dsc, dsr))
    )
```

The function 'has_parent_direct' checks that every class in the design which plays the role cp_1 is a direct subclass of some class which plays the role cp_2.

```
value
  has_parent_direct : G.Class_Name × G.Class_Name × Wf_Design_Renaming → Bool
  has_parent_direct(cp1, cp2, ((dsc, dsr), r)) ≡
    (
      ∀ cd : G.Class_Name •
        renaming_class_name(cd, cp1, r) ⇒
          (
            ∃ cd2 : G.Class_Name •
              renaming_class_name(cd2, cp2, r) ∧ R.is_parent(cd2, cd, dsr)
          )
    )
```

The function 'single_role_in_subclasses' requires that every class in the design which is a subclass of the class cd plays at most one of the roles in the set of roles cps.

```
value
  single_role_in_subclasses :
    G.Class_Name × G.Class_Name-set × Wf_Design_Renaming → Bool
  single_role_in_subclasses(cd, cps, ((dsc, dsr), r)) ≡
    (
      ∀ cp1, cp2, cd1 : G.Class_Name •
        cp1 ∈ cps ∧
        cp2 ∈ cps ∧
        cp1 ≠ cp2 ∧
        R.is_superclass(cd, cd1, dsr) ∧
        renaming_class_name(cd1, cp1, r) ⇒
          ∼ renaming_class_name(cd1, cp2, r)
    )
```

The function 'hierarchy' checks that there is exactly one class (cd_1) in the design which plays the role cp_1; that this class does not play any of the roles in the set of roles cps_2; that every subclass of cd_1 plays at most one of the roles in the set of roles cps_1, none of the roles in the set of roles cps_2, and at least one of the roles in cps_1 if the subclass is a leaf class; and that if some subclass of cd_1 plays some role in cps_1 and some subclass of that subclass also plays some role in cps_1 then the two subclasses must share the same role from cps_1. This property in fact describes most of the hierarchies of classes found in the GoF patterns.

```
value
  hierarchy :
    G.Class_Name × G.Class_Name-set × G.Class_Name-set × Wf_Design_Renaming
      → Bool
  hierarchy(cp1, cps1, cps2, ((dsc, dsr), r)) ≡
    exists_one(cp1, ((dsc, dsr), r)) ∧
    let cd1 : G.Class_Name • renaming_class_name(cd1, cp1, r) in
      ∼ has_role_in(cd1, cps2, r) ∧
      single_role_in_subclasses(cd1, cps1, ((dsc, dsr), r)) ∧
      (
        ∀ cd2 : G.Class_Name •
          R.is_superclass(cd1, cd2, dsr) ⇒
            (R.is_leaf(cd1, cd2, dsr) ⇒ has_role_in(cd2, cps1, r)) ∧
            (cd2 ∈ dom r ⇒ ∼ has_role_in(cd2, cps2, r)) ∧
            (
              ∀ cd3 : G.Class_Name •
                R.is_superclass(cd2, cd3, dsr) ∧
                has_role_in(cd2, cps1, r) ∧
                has_role_in(cd3, cps1, r) ⇒
                  share_role_in(cd2, cd3, cps1, r)
            )
      )
    end
```

The function 'only_one_hierarchy' states that among all the classes in a design that play the role cp there is one class which is a superclass of all the others. It is used in particular in the specification of the Proxy pattern in [11] to describe the properties of the hierarchy of classes playing the RealSubject role.

```
value
  only_one_hierarchy : G.Class_Name × Wf_Design_Renaming → Bool
  only_one_hierarchy(cp, ((dsc, dsr), r)) ≡
    let
      s = { cd | cd : G.Class_Name • renaming_class_name(cd, cp, r) }
    in
```

```
        (
            ∃! r : G.Class_Name •
                r ∈ s ∧
                    (
                        ∀ r′ : G.Class_Name •
                            r′ ∈ s \ {r} ⇒ R.is_superclass(r, r′, dsr)
                    )
        )
    end
```

The function 'extends_interface' says that classes playing the role cp_2 extend the interface of classes playing the role cp_1 in some way, either by adding new state variables or new methods or by making abstract methods concrete. It is of course only really useful in situations in which the role cp_2 represents a subclass of the role cp_1, and also makes most sense when there is additionally a unique class playing the role cp_1.

```
value
    extends_interface : G.Class_Name × G.Class_Name × Wf_Design_Renaming → Bool
    extends_interface(cp1, cp2, ((dsc, dsr), r)) ≡
        (
            ∀ cd1, cd2 : G.Class_Name •
                renaming_class_name(cd1, cp1, r) ∧ renaming_class_name(cd2, cp2, r) ⇒
                    C.class_state(dsc(cd2)) ≠ {} ∨
                    (
                        ∃ md : G.Method_Name •
                            C.method_name_in_class(md, dsc(cd2)) ∧
                            (
                                DS.has_method(md, cd1, (dsc, dsr)) ⇒
                                    let m = DS.method_of(cd1, md, (dsc, dsr)) in
                                        M.is_defined(m)
                                    end
                            )
                    )
        )
```

The function 'has_private_interface_by_inh' requires that no method in any class playing the role cp_2 is invoked in any class playing the role cp_1. It is written in terms of the auxiliary function 'exists_class_invoking_method' which checks whether there is some method md_2 in some class cd_2 in the design which invokes the given method md in the class cd_1. This invocation may be through the body of the method md_2, in which case there should be an aggregation or association relation which corresponds to the variable of the invocation and which links the class cd_2 either to the class cd_1 or to one of its superclasses. Alternatively, the variable of the invocation could

be one of the typed parameters of the method md_2, in which case this class should be either the class cd_1 or one of its superclasses.

```
value
  has_private_interface_by_inh : G.Class_Name × G.Class_Name × Wf_Design_Renaming
  → Bool
  has_private_interface_by_inh(cp1, cp2, ((dsc, dsr), r)) ≡
    (
      ∀ cd1, cd2 : G.Class_Name, md : G.Method_Name •
        renaming_class_name(cd1, cp1, r) ∧
        renaming_class_name(cd2, cp2, r) ∧
        DS.has_method(md, cd2, (dsc, dsr)) ⇒
          ~ exists_class_invoking_method(cd1, md, (dsc, dsr))
    ),

  exists_class_invoking_method :
    G.Class_Name × G.Method_Name × DS.Wf_Design_Structure → Bool
  exists_class_invoking_method(cd1, md, (dsc, dsr)) ≡
    (
      ∃ cd2 : G.Class_Name, md2 : G.Method_Name, vd : G.Variable_Name,
        inv : M.Invocation •
        DS.has_method(md2, cd2, (dsc, dsr)) ∧
        let m = DS.method_of(cd2, md2, (dsc, dsr)) in
          M.invocation_in_request_list(inv, m) ∧
          M.meth_name(M.call_sig(inv)) = md ∧
          M.call_vble(inv) = vd ∧
          (
            (
              ∃ c1 : G.Class_Name •
                R.exists_assoc_aggr_relation(vd, cd2, c1, dsr) ∧
                (cd1 = c1 ∨ R.is_superclass(c1, cd1, dsr))
            ) ∨
            vd ∈ G.set_f_params(M.f_params(m)) ∧
            G.var(vd) ∉ elems M.f_params(m) ∧
            let
              c = G.rol_of_parameter(vd, M.f_params(m))
            in
              (c = cd1 ∨ R.is_superclass(c, cd1, dsr))
            end
          )
        end
    )
```

5.2 Specifying Properties of State Variables in Patterns

The function 'store_unique_vble' checks whether every class in the design playing a given role has a single state variable playing a given role.

```
value
  store_unique_vble : G.Class_Name × G.Variable_Name × Wf_Design_Renaming → Bool
  store_unique_vble(cp, vp, (ds, r)) ≡
    (
      ∀ cd : G.Class_Name •
        renaming_class_name(cd, cp, r) ⇒
        (
          ∃! vd : G.Variable_Name • renaming_class_state(cd, cp, vd, vp, r)
        )
    )
```

Similarly, the function 'store_vble' checks that every class in the design playing a given role has at least one state variable playing a given role.

```
value
  store_vble : G.Class_Name × G.Variable_Name × Wf_Design_Renaming → Bool
  store_vble(cp, vp, (ds, r)) ≡
    (
      ∀ cd : G.Class_Name •
        renaming_class_name(cd, cp, r) ⇒
        (
          ∃ vd : G.Variable_Name • renaming_class_state(cd, cp, vd, vp, r)
        )
    )
```

Finally, the function 'less_quantities_of_variables' requires that the number of state variables playing the role vp_1 in every class in the design playing the role cp_1 is not greater than the number of state variables playing the role vp_2 in any class in the design playing the role cp_2.

```
value
  less_quantities_of_variables :
    G.Class_Name × G.Variable_Name × G.Class_Name × G.Variable_Name ×
    Wf_Design_Renaming
      →
    Bool
```

```
less_quantities_of_variables(cp1, vp1, cp2, vp2, ((dsc, dsr), r)) ≡
  (
    ∀ cd1, cd2 : G.Class_Name, var : G.Variable_Name •
      renaming_class_name(cd1, cp1, r) ∧
      R.exists_assoc_aggr_relation(var, cd1, cd2, dsr) ∧
      renaming_class_name(cd2, cp2, r) ⇒
        quantity_of_vble(cd1, cp1, vp1, r) ≤ quantity_of_vble(cd2, cp2, vp2, r)
  )
```

5.3 Specifying Properties of Relations in Patterns

The extended OMT notation distinguishes association and aggregation relations, and when these relations are drawn in the OMT diagrams representing the structures of the patterns in the GoF catalogue [13] either one or the other is shown. However, in some cases it is possible to use an aggregation relation in the design where the GoF catalogue shows an association relation and vice versa. For example, the structure of the Command pattern (see Figure 1) shows an association relation between the Client and the Receiver classes whereas the sample design used in the motivation of the pattern in [13] has an aggregation relation between the classes playing these roles. In our model, therefore, we have three possible "implementations" for a relation that is shown in the GoF catalogue as an association or an aggregation: either it must be an association, or it must be an aggregation, or it could be either. The three values of the variant type 'Rel_Type' describe these three cases.

```
type
  Rel_Type == Association | Aggregation | AssAggr
```

The above type is used as the parameter t in the function 'has_assoc_aggr_reltype', which checks whether every class playing the role cp_1 is linked to every class playing the role cp_2 by an association or aggregation relation of cardinality ap and specific type t.

```
value
  has_assoc_aggr_reltype :
    G.Class_Name × G.Class_Name × Rel_Type × G.Card × Wf_Design_Renaming
      → Bool
  has_assoc_aggr_reltype(cp1, cp2, t, ap, ((dsc, dsr), r)) ≡
    (
      ∀ cd1, cd2 : G.Class_Name •
        renaming_class_name(cd1, cp1, r) ∧
        renaming_class_name(cd2, cp2, r) ⇒
          (
```

```
                ∃ vd : G.Variable_Name •
                    R.exists_assoc_aggr_relation(vd, cd1, cd2, dsr) ∧
                    let r1 = R.assoc_aggr_relation(cd1, cd2, vd, dsr) in
                        R.sink_card(r1) = ap ∧
                        (t = Aggregation ⇒ R.is_aggregation(r1)) ∧
                        (t = Association ⇒ R.is_association(r1))
                    end
            )
    )
```

The function 'has_assoc_aggr_var_ren' checks the same property and also checks that the relation corresponds to a state variable playing the role vp in the class cp_1.

```
value
  has_assoc_aggr_var_ren :
    G.Class_Name × G.Class_Name × Rel_Type × G.Variable_Name × G.Card ×
    Wf_Design_Renaming
      →
    Bool
  has_assoc_aggr_var_ren(cp1, cp2, t, vp, ap, ((dsc, dsr), r)) ≡
    (
      ∀ cd1, cd2 : G.Class_Name •
        renaming_class_name(cd1, cp1, r) ∧
        renaming_class_name(cd2, cp2, r) ⇒
          let cr1 = class_renaming(cd1, cp1, r) in
            ∃ vd : G.Variable_Name •
              state_var_renames_to(vd, cr1, vp) ∧
              R.exists_assoc_aggr_relation(vd, cd1, cd2, dsr) ∧
              let r1 = R.assoc_aggr_relation(cd1, cd2, vd, dsr) in
                R.sink_card(r1) = ap ∧
                (t = Aggregation ⇒ R.is_aggregation(r1)) ∧
                (t = Association ⇒ R.is_association(r1))
              end
          end
    )
```

The function 'has_assoc_aggr' checks that every class playing the role cp_1 is linked to at least one class playing the role cp_2 by either an association or an aggregation relation of given cardinality.

```
value
  has_assoc_aggr :
    G.Class_Name × G.Class_Name × G.Card × Wf_Design_Renaming → Bool
```

```
has_assoc_aggr(cp1, cp2, ap, ((dsc, dsr), r)) ≡
  (
    ∀ cd1 : G.Class_Name •
      renaming_class_name(cd1, cp1, r) ⇒
        (
          ∃ cd2 : G.Class_Name, vd : G.Variable_Name •
            renaming_class_name(cd2, cp2, r) ∧
            R.exists_assoc_aggr_relation(vd, cd2, cd1, dsr) ∧
            let r1 = R.assoc_aggr_relation(cd2, cd1, vd, dsr) in
              R.sink_card(r1) = ap
            end
        )
  )
```

The function 'has_assoc_aggr_com' checks that there is an association or aggregation relation between every class playing the role cp_1 and every class playing the role cp_2, except that in the case when a single class plays both roles no relation between that class and itself is required. This function is used specifically in the specification of the properties of the Command pattern in [18].

```
value
  has_assoc_aggr_com : G.Class_Name × G.Class_Name × Wf_Design_Renaming → Bool
  has_assoc_aggr_com(cp1, cp2, ((dsc, dsr), r)) ≡
    (
      ∀ cd1, cd2 : G.Class_Name •
        renaming_class_name(cd1, cp1, r) ∧
        renaming_class_name(cd2, cp2, r) ∧ cd1 ≠ cd2 ⇒
          (
            ∃ vd : G.Variable_Name •
              R.exists_assoc_aggr_relation(vd, cd1, cd2, dsr)
          )
    )
```

The function 'has_assoc_var_ren' requires that every class playing the role cp_1 has one association relation of given cardinality with a class playing the role cp_2 corresponding to each of its state variables playing the role vp.

```
value
  has_assoc_var_ren :
    G.Class_Name × G.Class_Name × G.Variable_Name × G.Card × Wf_Design_Renaming
      →
    Bool
```

```
has_assoc_var_ren(cp1, cp2, vp, ap, ((dsc, dsr), r)) ≡
  (
    ∀ cd1 : G.Class_Name, vd : G.Variable_Name •
      renaming_class_state(cd1, cp1, vd, vp, r) ⇒
        (
          ∃! cd2 : G.Class_Name •
            renaming_class_name(cd2, cp2, r) ∧
            R.exists_assoc_aggr_relation(vd, cd1, cd2, dsr) ∧
            let r1 = R.assoc_aggr_relation(cd1, cd2, vd, dsr) in
              R.is_association(r1) ∧ R.sink_card(r1) = ap
            end
        )
  )
```

The function 'has_unique_assoc_aggr_relation' checks that there is precisely one association or aggregation relation between any class playing the role cp_1 and any class playing the role cp_2. This is mainly useful in situations where there is a single class playing the role cp_2, in which case it states that every class playing the role cp_1 has precisely one association or aggregation relation which links it to that class, and indeed the function is used in just this way in the specification of the Command pattern in [18].

```
value
  has_unique_assoc_aggr_relation :
    G.Class_Name × G.Class_Name × Wf_Design_Renaming → Bool
  has_unique_assoc_aggr_relation(cp1, cp2, ((dsc, dsr), r)) ≡
    (
      ∀ cd1, cd2 : G.Class_Name •
        renaming_class_name(cd1, cp1, r) ∧
        renaming_class_name(cd2, cp2, r) ⇒
          (
            ∃! vd : G.Variable_Name •
              R.exists_assoc_aggr_relation(vd, cd1, cd2, dsr)
          )
    )
```

The function 'has_at_least_two_assoc_aggr' requires that every class playing the role cp_1 is linked by association or aggregation relations of given cardinality to at least two distinct classes playing the role cp_2.

```
value
  has_at_least_two_assoc_aggr :
    G.Class_Name × G.Class_Name × G.Card × Wf_Design_Renaming → Bool
```

```
has_at_least_two_assoc_aggr(cp1, cp2, ap, ((dsc, dsr), r)) ≡
  (
    ∀ cd1 : G.Class_Name •
      renaming_class_name(cd1, cp1, r) ⇒
        (
          ∃ cd2, cd3 : G.Class_Name, vd1, vd2 : G.Variable_Name •
            renaming_class_name(cd2, cp2, r) ∧
            renaming_class_name(cd3, cp2, r) ∧
            cd2 ≠ cd3 ∧
            R.exists_assoc_aggr_relation(vd1, cd1, cd2, dsr) ∧
            R.exists_assoc_aggr_relation(vd2, cd1, cd3, dsr) ∧
            let
              r1 = R.assoc_aggr_relation(cd1, cd2, vd1, dsr),
              r2 = R.assoc_aggr_relation(cd1, cd3, vd2, dsr)
            in
              R.sink_card(r1) = ap ∧ R.sink_card(r2) = ap
            end
        )
  )
```

The next function, 'has_instantiation', checks that every class playing the role cp_2 is the sink of an instantiation relation whose source is some class playing the role cp_1. Thus every class playing the role cp_2 is instantiated by at least one class playing the role cp_1.

```
value
  has_instantiation : G.Class_Name × G.Class_Name × Wf_Design_Renaming → Bool
  has_instantiation(cp1, cp2, ((dsc, dsr), r)) ≡
    (
      ∀ cd2 : G.Class_Name •
        renaming_class_name(cd2, cp2, r) ⇒
          (
            ∃ cd1 : G.Class_Name •
              renaming_class_name(cd1, cp1, r) ∧
              R.exists_inst_relation(cd1, cd2, dsr)
          )
    )
```

On the other hand, the function 'has_no_instantiation' says that there should be no instantiation relation in a design whose source and sink are classes playing the roles cp_1 and cp_2 respectively.

```
value
  has_no_instantiation : G.Class_Name × G.Class_Name × Wf_Design_Renaming → Bool
```

```
has_no_instantiation(cp1, cp2, ((dsc, dsr), r)) ≡
  (
    ∀ cd1, cd2 : G.Class_Name •
      renaming_class_name(cd1, cp1, r) ∧
      renaming_class_name(cd2, cp2, r) ⇒
        ~ R.exists_inst_relation(cd1, cd2, dsr)
  )
```

The function 'has_unique_assoc_aggr' combines the above with the function 'has_unique_assoc_aggr_relation' to check that there is precisely one association or aggregation relation between any class playing the role cp_1 and any class playing the role cp_2 and no instantiation relation between the two classes. Again this is mainly useful in situations where there is a single class playing the role cp_2.

```
value
  has_unique_assoc_aggr : G.Class_Name × G.Class_Name × Wf_Design_Renaming → Bool
  has_unique_assoc_aggr(cp1, cp2, dr) ≡
    has_unique_assoc_aggr_relation(cp1, cp2, dr) ∧
    has_no_instantiation(cp1, cp2, dr)
```

The function 'class_connected' checks that for every class playing the role cp_1 there is at least one class playing the role cp_2 to which it is linked by an association or aggregation relation, the relation being either direct or through a superclass of the class playing the role cp_1.

```
value
  class_connected : G.Class_Name × G.Class_Name × Wf_Design_Renaming → Bool
  class_connected(cp1, cp2, ((dsc, dsr), r)) ≡
    (
      ∀ cd1 : G.Class_Name •
        renaming_class_name(cd1, cp1, r) ⇒
          (
            ∃ cd2 : G.Class_Name, vd : G.Variable_Name •
              renaming_class_name(cd2, cp2, r) ∧
              R.exists_assoc_aggr_relation(vd, cd2, cd1, dsr)
          )
    )
```

The function 'equal_inst_asso' checks that there is an instantiation relation from a class playing the role cp_1 to another playing the role cp_2 if and only if the same two classes are linked in the opposite direction by an association or an aggregation relation, either directly or through a superclass of the class playing the role cp_1.

```
value
  equal_inst_asso : G.Class_Name × G.Class_Name × Wf_Design_Renaming → Bool
  equal_inst_asso(cp1, cp2, ((dsc, dsr), r)) ≡
    (
      ∀ cd1, cd2 : G.Class_Name •
        renaming_class_name(cd1, cp1, r) ∧
        renaming_class_name(cd2, cp2, r) ⇒
          R.exists_inst_relation(cd1, cd2, dsr) =
            (
              ∃ vd : G.Variable_Name •
                R.exists_assoc_aggr_relation(vd, cd2, cd1, dsr)
            )
    )
```

The function 'nro_inst_asso' states that every class playing the role cp_1 has the same number of instantiation relations linking it to classes playing the role cp_2 as it has methods playing the role mp.

```
value
  nro_inst_asso :
    G.Class_Name × G.Class_Name × G.Method_Name × Wf_Design_Renaming → Bool
  nro_inst_asso(cp1, cp2, mp, ((dsc, dsr), r)) ≡
    (
      ∀ cd1 : G.Class_Name •
        renaming_class_name(cd1, cp1, r) ⇒
          card
            { ci | ci : G.Class_Name •
              R.exists_inst_relation(cd1, ci, dsr) ∧ renaming_class_name(ci, cp2, r)
            } =
          card
            { md | md : G.Method_Name •
              renaming_class_method(cd1, cp1, md, mp, r)
            }
    )
```

The function 'classes_not_related' says that there are no relations between two different classes playing the same given role cp, and the function 'not_related_classes' generalises this to say that there are no relations between any class playing the role cp_1 and any different class playing the role cp_2.

```
value
  classes_not_related : G.Class_Name × Wf_Design_Renaming → Bool
```

```
classes_not_related(cp, ((dsc, dsr), r)) ≡
  (
    ∀ cd1, cd2 : G.Class_Name •
      renaming_class_name(cd1, cp, r) ∧
      renaming_class_name(cd2, cp, r) ∧ cd1 ≠ cd2 ⇒
        R.no_relation(cd1, cd2, dsr)
  ),

not_related_classes : G.Class_Name × G.Class_Name × Wf_Design_Renaming → Bool
not_related_classes(cp1, cp2, ((dsc, dsr), r)) ≡
  (
    ∀ cd1, cd2 : G.Class_Name •
      renaming_class_name(cd1, cp1, r) ∧
      renaming_class_name(cd2, cp2, r) ∧
      cd1 ≠ cd2 ⇒
        R.no_relation(cd1, cd2, dsr)
  )
```

The function 'has_ass_agg_var_ren_one_sink' states that every class playing the role cp_1 has a state variable playing the role vp which represents an association or aggregation relation of given type (t) and cardinality (ap) with a class c playing the role cp_2, all other classes playing the role cp_2 being subclasses of this class c. The function is used specifically to describe the relationship between the Proxy class and the hierarchy of RealSubject classes in the Proxy pattern (see [11]).

```
value
  has_ass_agg_var_ren_one_sink :
    G.Class_Name × G.Class_Name × Rel_Type × G.Variable_Name × G.Card ×
    Wf_Design_Renaming
      → Bool
  has_ass_agg_var_ren_one_sink(cp1, cp2, t, vp, ap, ((dsc, dsr), r)) ≡
    (
      ∀ cd1 : G.Class_Name •
        renaming_class_name(cd1, cp1, r) ⇒
          (
            ∃ vd : G.Variable_Name •
              renaming_class_state(cd1, cp1, vd, vp, r) ∧
              let
                s = { cd | cd : G.Class_Name • renaming_class_name(cd, cp2, r) }
              in
                (
                  ∃! c : G.Class_Name •
                    c ∈ s ∧
                    (
```

```
                    ∀ c′ : G.Class_Name •
                      c′ ∈ s \ {c} ⇒ R.is_superclass(c, c′, dsr)
                ) ∧
                R.exists_assoc_aggr_relation(vd, cd1, c, dsr) ∧
                let r1 = R.assoc_aggr_relation(cd1, c, vd, dsr) in
                    R.sink_card(r1) = ap ∧
                    (t = Aggregation ⇒ R.is_aggregation(r1)) ∧
                    (t = Association ⇒ R.is_association(r1))
                end
            )
        end
    )
)
```

5.4 Specifying Properties of Methods in Patterns

The function 'unique_method' checks whether every class in the design playing a given role has a single method playing a given role. Note that this additionally checks that no superclass can have more than one method playing the given role, which would in fact have to be the same method because of the consistency conditions on the renaming of inherited methods (see Section 4).

```
value
  unique_method : G.Class_Name × G.Method_Name × Wf_Design_Renaming → Bool
  unique_method(cp, mp, ((dsc, dsr), r)) ≡
    (
      ∀ cd : G.Class_Name •
        renaming_class_name(cd, cp, r) ⇒
          (
            ∃! md : G.Method_Name •
              renaming_class_method(cd, cp, md, mp, r)
          ) ∧
          (
            ∀ cd1 : G.Class_Name, cr2 : ClassRenaming •
            R.is_superclass(cd, cd1, dsr) ∧
            cd1 ∈ dom r ∧ cr2 ∈ r(cd1) ⇒
              card method_renaming_to(cd1, classname(cr2), mp, r) ≤ 1
          )
    )
```

The functions 'has_def_method', 'has_error_method' and 'has_impl_method' check that every class in the design playing a given role has at least one method which plays a given role and

which is abstract, error or implemented respectively.

```
value
  has_def_method : G.Class_Name × G.Method_Name × Wf_Design_Renaming → Bool
  has_def_method(cp, mp, ((dsc, dsr), r)) ≡
    (
      ∀ cd : G.Class_Name •
        renaming_class_name(cd, cp, r) ⇒
          (
            ∃ md : G.Method_Name •
              renaming_class_method(cd, cp, md, mp, r) ∧
              let c = DS.class_of_method(md, cd, (dsc, dsr)) in
                M.is_defined(C.class_methods(dsc(c))(md))
              end
          )
    ),

  has_error_method : G.Class_Name × G.Method_Name × Wf_Design_Renaming → Bool
  has_error_method(cp, mp, ((dsc, dsr), r)) ≡
    (
      ∀ cd : G.Class_Name •
        renaming_class_name(cd, cp, r) ⇒
          (
            ∃ md : G.Method_Name •
              renaming_class_method(cd, cp, md, mp, r) ∧
              let c = DS.class_of_method(md, cd, (dsc, dsr)) in
                M.body(C.class_methods(dsc(c))(md)) = M.error
              end
          )
    ),

  has_impl_method : G.Class_Name × G.Method_Name × Wf_Design_Renaming → Bool
  has_impl_method(cp, mp, ((dsc, dsr), r)) ≡
    (
      ∀ cd : G.Class_Name •
        renaming_class_name(cd, cp, r) ⇒
          (
            ∃ md : G.Method_Name •
              renaming_class_method(cd, cp, md, mp, r) ∧
              let c = DS.class_of_method(md, cd, (dsc, dsr)) in
                M.is_implemented(C.class_methods(dsc(c))(md))
              end
          )
    )
```

The functions 'has_all_def_method' and 'has_all_impl_method' are similar except that they check that in every class in the design playing a given role all methods which play a given role are abstract or implemented respectively.

```
value
  has_all_def_method : G.Class_Name × G.Method_Name × Wf_Design_Renaming → Bool
  has_all_def_method(cp, mp, ((dsc, dsr), r)) ≡
    (
      ∀ cd : G.Class_Name, md : G.Method_Name •
        renaming_class_method(cd, cp, md, mp, r) ⇒
          let c = DS.class_of_method(md, cd, (dsc, dsr)) in
            M.is_defined(C.class_methods(dsc(c))(md))
          end
    ),

  has_all_impl_method : G.Class_Name × G.Method_Name × Wf_Design_Renaming → Bool
  has_all_impl_method(cp, mp, ((dsc, dsr), r)) ≡
    (
      ∀ cd : G.Class_Name, md : G.Method_Name •
        renaming_class_method(cd, cp, md, mp, r) ⇒
          let c = DS.class_of_method(md, cd, (dsc, dsr)) in
            M.is_implemented(C.class_methods(dsc(c))(md))
          end
    )
```

The function 'has_method_without_res' checks that all methods playing the role mp in a class playing the role cp do not return a result.

```
value
  has_method_without_res :
    G.Class_Name × G.Method_Name × Wf_Design_Renaming → Bool
  has_method_without_res(cp, mp, (ds, r)) ≡
    (
      ∀ cd : G.Class_Name, md : G.Method_Name •
        renaming_class_method(cd, cp, md, mp, r) ⇒
          let m = DS.method_of(cd, md, ds) in M.meth_res(m) = {} end
    )
```

The function 'has_method_with_res' checks the converse, namely that all methods playing the role mp in a class playing the role cp do return a result. The functions 'has_method_with_res_with_ren' and 'has_method_with_result_class' represent specialisations of this property. The first says that the result is a state variable playing the role vp in the class playing the role cp, while the second

says that the result is a variable within the body of the method playing the role mp (provided the method is implemented, of course) which represents the result of an instantiation of a class playing the role cp_2.

```
value
  has_method_with_res : G.Class_Name × G.Method_Name × Wf_Design_Renaming → Bool
  has_method_with_res(cp, mp, (ds, r)) ≡
    (
      ∀ cd : G.Class_Name, md : G.Method_Name •
        renaming_class_method(cd, cp, md, mp, r) ⇒
          let m = DS.method_of(cd, md, ds) in M.meth_res(m) ≠ {} end
    ),

  has_method_with_res_with_ren :
    G.Class_Name × G.Method_Name × G.Variable_Name × Wf_Design_Renaming → Bool
  has_method_with_res_with_ren(cp, mp, vp, (ds, r)) ≡
    (
      ∀ cd : G.Class_Name, md : G.Method_Name •
        renaming_class_method(cd, cp, md, mp, r) ⇒
          (
            ∃ vd : G.Variable_Name •
              renaming_class_state(cd, cp, vd, vp, r) ∧
              let m = DS.method_of(cd, md, ds) in
                M.meth_res(m) = {vd}
              end
          )
    ),

  has_method_with_result_class :
    G.Class_Name × G.Method_Name × G.Class_Name × Wf_Design_Renaming → Bool
  has_method_with_result_class(cp, mp, cp2, ((dsc, dsr), r)) ≡
    (
      ∀ cd : G.Class_Name, md : G.Method_Name •
        renaming_class_method(cd, cp, md, mp, r) ⇒
          let m = DS.method_of(cd, md, (dsc, dsr)) in
            M.is_implemented(m) ⇒
              (
                ∃ cd2 : G.Class_Name, vd : G.Variable_Name, i : M.Instantiation •
                  renaming_class_name(cd2, cp2, r) ∧
                  vd ∈ M.changed_variables(m) ∧
                  M.class_name(i) = cd2 ∧
                  M.Request_from_Instantiation(i) =
                  M.variable_change_body(m)({vd}) ∧
                  M.meth_res(m) = {vd}
```

```
            )
        end
    )
```

The function 'result_of_Get_State' says that the result of every method playing the role mp in a class playing the role cp is the set of state variables playing the role vp in the same class. This is specifically used to describe the method GetState in the Memento pattern (see [18]). The function 'results_of_Get_State' is similar except that it requires only that the result of the method is a subset of the state variables playing the role vp, though it additionally requires that each of these state variables belongs to the result of at least one such method. This is specifically used to describe the method GetState in the Observer pattern (see [18]).

```
value
  result_of_Get_State :
    G.Class_Name × G.Method_Name × G.Variable_Name × Wf_Design_Renaming
      → Bool
  result_of_Get_State(cp, mp, vp, (ds, r)) ≡
    (
      ∀ cd : G.Class_Name, md : G.Method_Name •
        renaming_class_method(cd, cp, md, mp, r) ⇒
          let
            m = DS.method_of(cd, md, ds),
            vs = state_vars_renaming_to(cd, cp, vp, r)
          in
            M.meth_res(m) = vs
          end
    ),

  results_of_Get_State :
    G.Class_Name × G.Method_Name × G.Variable_Name × Wf_Design_Renaming
      → Bool
  results_of_Get_State(cp, mp, vp, (ds, r)) ≡
    (
      ∀ cd : G.Class_Name, md : G.Method_Name •
        renaming_class_method(cd, cp, md, mp, r) ⇒
          let
            m = DS.method_of(cd, md, ds),
            vs = state_vars_renaming_to(cd, cp, vp, r)
          in
            M.meth_res(m) ≠ {} ∧ M.meth_res(m) ⊆ vs
          end
    ) ∧
    (
      ∀ cd : G.Class_Name, vd : G.Variable_Name •
```

```
        renaming_class_state(cd, cp, vd, vp, r) ⇒
          (
            ∃ md : G.Method_Name •
              renaming_class_method(cd, cp, md, mp, r) ∧
              let m = DS.method_of(cd, md, ds) in
                vd ∈ M.meth_res(m)
              end
          )
    )
```

The function 'res_loc_vchge_inst_aparam_self' requires that every method playing the role mp in a class playing the role cp is implemented and contains an instantiation of a class playing the role cp_2, the result of this instantiation being assigned to a variable and this variable forming the result of the method. There should also be an instantiation relation between the two classes.

```
value
  res_loc_vchge_inst_aparam_self :
    G.Class_Name × G.Method_Name × G.Class_Name × Wf_Design_Renaming → Bool
  res_loc_vchge_inst_aparam_self(cp, mp, cp2, ((dsc, dsr), r)) ≡
    (
      ∀ cd : G.Class_Name, md : G.Method_Name •
        renaming_class_method(cd, cp, md, mp, r) ⇒
          (
            ∃ cd2 : G.Class_Name, vd : G.Variable_Name •
              R.exists_inst_relation(cd, cd2, dsr) ∧
              renaming_class_name(cd2, cp2, r) ∧
              let
                m = DS.method_of(cd, md, (dsc, dsr)),
                inst = M.mk_Instantiation(cd2, ⟨G.self⟩)
              in
                M.is_implemented(m) ∧
                {vd} ∈ dom M.variable_change_body(m) ∧
                M.Request_from_Instantiation(inst) =
                M.variable_change_body(m)({vd}) ∧
                M.meth_res(m) = {vd}
              end
          )
    )
```

The function 'res_local_var_change_inst_aparam_ren' requires that every method playing the role mp in a class playing the role cp is implemented and has a body which comprises precisely one instantiation followed by one invocation. The instantiation is an instantiation of the unique

class playing the role cp_2 with no parameters and the result of this instantiation is assigned to some variable, say v. Then the invocation invokes the unique method playing the role mp_2 from the class playing the role cp_2 on the variable v, the parameters of this invocation being all state variables playing the role vp. Finally, the variable v is returned as the result of the method playing the role mp.

```
value
  res_local_var_change_inst_aparam_ren :
    G.Class_Name × G.Method_Name × G.Variable_Name ×
      G.Class_Name × G.Method_Name × Wf_Design_Renaming → Bool
  res_local_var_change_inst_aparam_ren(cp, mp, vp, cp₂, mp₂, (ds, r)) ≡
    (
      ∀ cd, cd₂ : G.Class_Name, md, md₂ : G.Method_Name •
        renaming_class_method(cd, cp, md, mp, r) ∧
        renaming_class_method(cd₂, cp₂, md₂, mp₂, r) ⇒
          let
            vs = state_vars_renaming_to(cd, cp, vp, r),
            m = DS.method_of(cd, md, ds)
          in
            M.is_implemented(m) ∧
            let
              rlb = M.request_list_body(m),
              vm = M.variable_change_body(m),
              inst = M.mk_Instantiation(cd₂, ⟨⟩)
            in
              ∃ v : G.Variable_Name, inv : M.Invocation •
                vm = [ {v} ↦ inst ] ∧
                M.call_vble(inv) = v ∧
                M.a_params_from_set(M.a_params(M.call_sig(inv)), vs) ∧
                rlb = ⟨inst, inv⟩ ∧ M.meth_res(m) = {v}
            end
          end
    )
```

The function 'res_local_var_change_inv_aparam_ren' requires that every method playing the role mp which has a parameter playing the role vp_2 and belongs to a class playing the role cp is implemented and has a body which comprises precisely one invocation, this invoking the given method mp_2 on the unique state variable playing the role vp with the same parameter. The result of the invocation is returned as the result of the method.

```
value
  res_local_var_change_inv_aparam_ren :
    G.Class_Name × G.Method_Name × G.Variable_Name × G.Method_Name ×
```

```
        G.Variable_Name × Wf_Design_Renaming → Bool
    res_local_var_change_inv_aparam_ren(cp, mp, vp, mp2, vp2, (ds, r)) ≡
      (
       ∀
          cd, cd2 : G.Class_Name, md : G.Method_Name, vd, vd2 : G.Variable_Name •
            renaming_class_state(cd, cp, vd, vp, r) ∧
            renaming_class_method_param(cd, cp, md, mp, vd2, vp2, r) ⇒
              let
                m = DS.method_of(cd, md, ds),
                sig = M.mk_Actual_Signature(mp2, ⟨vd⟩),
                inv = M.mk_Invocation(vd2, sig)
              in
                M.is_implemented(m) ∧
                (
                  ∃ vd1 : G.Variable_Name •
                    {vd1} ∈ dom M.variable_change_body(m) ∧
                    M.Request_from_Invocation(inv) =
                    M.variable_change_body(m)({vd1}) ∧
                    M.request_list(M.body(m)) = ⟨inv⟩ ∧
                    M.meth_res(m) = {vd1}
                )
              end
      )
```

The next group of functions deal with the parameters of methods. The first, 'no_parameter_in_design', says that every method playing the role mp in some class playing the role cp has no (formal) parameters, while the second, 'images_ren_one_var_par_in_design', says that each such method has a single parameter and that parameter plays the role vp and the third, 'one_image_ren_pars_in_design', extends this further and requires that the single parameter is a typed parameter, the type being a class playing the role cp_2. The function 'all_pars_same_ren' places no restriction on the number of parameters (which could in fact be zero), but requires that all parameters play the given role vp, and the function 'fparams_var_ren' again does not restrict the number of parameters but requires that there must be at least one playing the role vp.

```
value
  no_parameter_in_design :
    G.Class_Name × G.Method_Name × Wf_Design_Renaming → Bool
  no_parameter_in_design(cp, mp, (ds, r)) ≡
    (
      ∀ cd : G.Class_Name, md : G.Method_Name •
        renaming_class_method(cd, cp, md, mp, r) ⇒
          let m = DS.method_of(cd, md, ds) in M.f_params(m) = ⟨⟩ end
    ),
```

```
images_ren_one_var_par_in_design :
   G.Class_Name × G.Method_Name × G.Variable_Name × Wf_Design_Renaming
      → Bool
images_ren_one_var_par_in_design(cp, mp, vp, (ds, r)) ≡
   (
      ∀ cd : G.Class_Name, md : G.Method_Name •
         renaming_class_method(cd, cp, md, mp, r) ⇒
            let m = DS.method_of(cd, md, ds) in
               ∃ p : G.Parameter •
                  M.f_params(m) = ⟨p⟩ ∧
                  renaming_class_method_param(cd, cp, md, mp,
                                              G.type_parameter(p), vp, r)
            end
   ),

one_image_ren_pars_in_design :
   G.Class_Name × G.Method_Name × G.Class_Name × G.Variable_Name ×
   Wf_Design_Renaming
      → Bool
one_image_ren_pars_in_design(cp1, mp, cp2, vp, (ds, r)) ≡
   (
      ∀ cd : G.Class_Name, md : G.Method_Name, vd : G.Variable_Name •
         renaming_class_method(cd, cp1, md, mp, r) ⇒
            renaming_class_method_param(cd, cp1, md, mp, vd, vp, r) ∧
            let m = DS.method_of(cd, md, ds) in
               ∃ cd1 : G.Class_Name •
                  M.f_params(m) = ⟨G.paramTyped(vd, cd1)⟩ ∧
                  renaming_class_name(cd1, cp2, r)
            end
   ),

all_pars_same_ren :
   G.Class_Name × G.Method_Name × G.Variable_Name × Wf_Design_Renaming
      → Bool
all_pars_same_ren(cp, mp, vp, (ds, r)) ≡
   (
      ∀ cd : G.Class_Name, md : G.Method_Name, vd : G.Variable_Name •
         renaming_class_method(cd, cp, md, mp, r) ∧
         let m = DS.method_of(cd, md, ds) in
            vd ∈ G.set_f_params(M.f_params(m))
         end ⇒
            renaming_class_method_param(cd, cp, md, mp, vd, vp, r)
   ),

fparams_var_ren :
```

```
    G.Class_Name × G.Method_Name × G.Variable_Name × Wf_Design_Renaming
      → Bool
  fparams_var_ren(cp, mp, vp, (ds, r)) ≡
    (
      ∀ cd : G.Class_Name, md : G.Method_Name •
        renaming_class_method(cd, cp, md, mp, r) ⇒
          let m = DS.method_of(cd, md, ds) in
            ∃ p : G.Parameter •
              p ∈ elems M.f_params(m) ∧
              renaming_class_method_param(cd, cp, md, mp,
                                          G.type_parameter(p), vp, r)
          end
    ),

  differents_params :
    G.Class_Name × G.Method_Name × G.Class_Name × Wf_Design_Renaming → Bool
  differents_params(cp1, mp, cp2, (ds, r)) ≡
    (
      ∀ cd1, cd2, cd3 : G.Class_Name, md1, md2 : G.Method_Name,
        vd1, vd2 : G.Variable_Name •
        renaming_class_method2(cd1, cp1, md1, mp, md2, mp, r) ∧
        md1 ≠ md2 ∧
        G.paramTyped(vd1, cd2) ∈
        elems M.f_params(DS.method_of(cd1, md1, ds)) ∧
        G.paramTyped(vd2, cd3) ∈ elems M.f_params(DS.method_of(cd1, md2, ds)) ⇒
          cd2 ≠ cd3
    )
```

The function 'params_vars_in_SetState_one' requires that every method playing the role mp in a class playing the role cp is implemented, has the same number of parameters as there are state variables playing the role vp in the same class, and has within its body assignments of its parameters to those state variables, each parameter being assigned to a different state variable. The function 'params_vars_in_SetState_many' is similar except that each method playing the role mp makes assignments to a subset of the state variables, with the additional constraint that each state variable must be assigned by at least one such method.

```
value
  params_vars_in_SetState_one :
    G.Class_Name × G.Method_Name × G.Variable_Name × Wf_Design_Renaming → Bool
  params_vars_in_SetState_one(cp, mp, vp, (ds, r)) ≡
    (
      ∀ cd : G.Class_Name, md : G.Method_Name •
        renaming_class_method(cd, cp, md, mp, r) ⇒
          let
```

```
            vs = state_vars_renaming_to(cd, cp, vp, r),
            m = DS.method_of(cd, md, ds),
            fp = M.f_params(m)
        in
            M.is_implemented(m) ∧
            len fp = card vs ∧
            let
                vm = M.variable_change_body(m),
                vss = { {v'} | v' : G.Variable_Name • v' ∈ vs }
            in
                vss ⊆ dom vm ∧
                M.is_one_one(vm / vss) ∧
                (
                    ∀ v : G.Variable_Name •
                        v ∈ vs ⇒
                        (
                            ∃ vrs : M.Variables •
                                vrs ⊆ G.set_f_params(fp) ∧ vm({v}) = vrs
                        )
                )
            end
        end
    ),

params_vars_in_SetState_many :
    G.Class_Name × G.Method_Name × G.Variable_Name × Wf_Design_Renaming → Bool
params_vars_in_SetState_many(cp, mp, vp, (ds, r)) ≡
    (
        ∀ cd : G.Class_Name, md : G.Method_Name •
            renaming_class_method(cd, cp, md, mp, r) ⇒
            let
                vs = state_vars_renaming_to(cd, cp, vp, r),
                m = DS.method_of(cd, md, ds),
                fp = M.f_params(m)
            in
                M.is_implemented(m) ∧
                len fp ≠ 0 ∧
                len fp ≤ card vs ∧
                let
                    vm = M.variable_change_body(m),
                    vss = { {v'} | v' : G.Variable_Name • v' ∈ vs },
                    varsets = vss ∩ dom vm
                in
                    varsets ≠ {} ∧
                    M.is_one_one(vm / varsets) ∧
```

```
                    (
                        ∀ v : G.Variable_Name •
                            {v} ∈ varsets ⇒
                                (
                                    ∃ vrs : M.Variables •
                                        vrs ⊆ G.set_f_params(fp) ∧ vm({v}) = vrs
                                )
                    )
                end
            end
        ) ∧
        (
            ∀ cd : G.Class_Name, vd : G.Variable_Name •
                renaming_class_state(cd, cp, vd, vp, r) ⇒
                    (
                        ∃ md : G.Method_Name •
                            renaming_class_method(cd, cp, md, mp, r) ∧
                            let m = DS.method_of(cd, md, ds) in
                                {vd} ∈ dom M.variable_change_body(m)
                            end
                    )
        )
```

The next batch of functions deals with the bodies of methods. The first, 'self_invocation', requires that the body of every method playing the role mp_1 in a class playing the role cp_1 contains a self-invocation of a method playing the role mp_2 in the same class.

```
value
    self_invocation :
        G.Class_Name × G.Method_Name × G.Method_Name × Wf_Design_Renaming → Bool
    self_invocation(cp, mp1, mp2, ((dsc, dsr), r)) ≡
        (
            ∀ cd : G.Class_Name, md1 : G.Method_Name •
                renaming_class_method(cd, cp, md1, mp1, r) ⇒
                    let m = DS.method_of(cd, md1, (dsc, dsr)) in
                        ∃ md2 : G.Method_Name, i : M.Invocation •
                            renaming_class_method(cd, cp, md2, mp2, r) ∧
                            M.invocation_in_request_list(i, m) ∧
                            M.meth_name(M.call_sig(i)) = md2 ∧
                            M.call_vble(i) = G.self
                    end
        )
```

The function 'exists_super_invocation' requires that every class playing the role cp contains at least one method playing the role mp whose body contains a super-invocation of the same method. Similarly, the function 'exists_super_self_inv' requires that the body of every method playing the role mp_2 in a class playing the role cp contains a super-invocation of the same method and a self-invocation of a method playing the role mp_1 in the same class.

```
value
  exists_super_invocation :
    G.Class_Name × G.Method_Name × Wf_Design_Renaming → Bool
  exists_super_invocation(cp, mp, (ds, r)) ≡
    (
      ∀ cd : G.Class_Name •
        renaming_class_name(cd, cp, r) ⇒
          (
            ∃ md : G.Method_Name, inv : M.Invocation •
              renaming_class_method(cd, cp, md, mp, r) ∧
              let m = DS.method_of(cd, md, ds) in
                M.invocation_in_request_list(inv, m) ∧
                M.call_vble(inv) = G.super ∧
                M.meth_name(M.call_sig(inv)) = md
              end
          )
    ),

  exists_super_self_inv :
    G.Class_Name × G.Method_Name × G.Method_Name × Wf_Design_Renaming
      → Bool
  exists_super_self_inv(cp, mp1, mp2, (ds, r)) ≡
    (
      ∀ cd : G.Class_Name, md2 : G.Method_Name •
        renaming_class_method(cd, cp, md2, mp2, r) ⇒
          (
            ∃ md1 : G.Method_Name, inv1, inv2 : M.Invocation •
              renaming_class_method(cd, cp, md1, mp1, r) ∧
              let m = DS.method_of(cd, md1, ds) in
                M.invocation_in_request_list(inv1, m) ∧
                M.call_vble(inv1) = G.super ∧
                M.meth_name(M.call_sig(inv1)) = md2 ∧
                M.invocation_in_request_list(inv2, m) ∧
                M.call_vble(inv2) = G.self ∧
                M.meth_name(M.call_sig(inv2)) = md1
              end
          )
    )
```

The function 'deleg_with_var' states that every method playing the role mp in a class playing the role cp is implemented and contains an invocation to each of the state variables playing the role vp in the same class of each method playing the role mp_2 in a class playing the role cp_2. It is used primarily in cases where there is a unique state variable playing the role vp and also a unique class playing the role cp_2. The function 'deleg_with_var_coll_aparam_ren' is similar except that the parameter mp_2 represents a method in the design (in particular one of the collection methods) and a parameter of the main method which plays the role vp_2 is passed directly as a parameter to the internal invocation.

```
value
  deleg_with_var :
    G.Class_Name × G.Method_Name × G.Variable_Name ×
      G.Class_Name × G.Method_Name × Wf_Design_Renaming → Bool
  deleg_with_var(cp, mp, vp, cp2, mp2, (ds, r)) ≡
    (
      ∀ cd, cd2 : G.Class_Name, vd : G.Variable_Name, md, md2 : G.Method_Name •
        renaming_class_method(cd2, cp2, md2, mp2, r) ∧
        renaming_class_method_state(cd, cp, md, mp, vd, vp, r) ⇒
          let m = DS.method_of(cd, md, ds) in
            ∃ i : M.Invocation •
              M.invocation_in_request_list(i, m) ∧
              M.meth_name(M.call_sig(i)) = md2 ∧ M.call_vble(i) = vd
          end
    ),

  deleg_with_var_coll_aparam_ren :
    G.Class_Name × G.Method_Name × G.Variable_Name ×
      G.Method_Name × G.Variable_Name × Wf_Design_Renaming → Bool
  deleg_with_var_coll_aparam_ren(cp, mp, vp, mp2, vp2, (ds, r)) ≡
    (
      ∀ cd : G.Class_Name, md : G.Method_Name, vd, vd2 : G.Variable_Name •
        renaming_class_method_param(cd, cp, md, mp, vd2, vp2, r) ∧
        renaming_class_state(cd, cp, vd, vp, r) ⇒
          let m = DS.method_of(cd, md, ds) in
            ∃ i : M.Invocation •
              M.invocation_in_request_list(i, m) ∧
              M.meth_name(M.call_sig(i)) = mp2 ∧
              M.call_vble(i) = vd ∧
              vd2 ∈ elems M.a_params(M.call_sig(i))
          end
    )
```

The function 'deleg_var_to_some_class' states that every method playing the role mp_1 in a class playing the role cp_1 is implemented and contains an invocation to a state variable playing the

role vp of a method playing the role mp_2 in some class playing the role cp_2, with the state variable representing an association or aggregation relation between the two classes.

```
value
  deleg_var_to_some_class :
    G.Class_Name × G.Method_Name × G.Variable_Name ×
      G.Class_Name × G.Method_Name × Wf_Design_Renaming → Bool
  deleg_var_to_some_class(cp1, mp1, vp, cp2, mp2, ((dsc, dsr), r)) ≡
    (
      ∀ cd1 : G.Class_Name, md1 : G.Method_Name •
        renaming_class_method(cd1, cp1, md1, mp1, r) ⇒
          let m = DS.method_of(cd1, md1, (dsc, dsr)) in
            M.is_implemented(m) ∧
            (
              ∃ cd2 : G.Class_Name, md2 : G.Method_Name, inv : M.Invocation •
                renaming_class_method(cd2, cp2, md2, mp2, r) ∧
                M.invocation_in_request_list(inv, m) ∧
                M.meth_name(M.call_sig(inv)) = md2 ∧
                let cr = class_renaming(cd1, cp1, r) in
                  state_var_renames_to(M.call_vble(inv), cr, vp)
                end ∧
                R.exists_assoc_aggr_relation(M.call_vble(inv), cd1, cd2, dsr)
            )
          end
    )
```

The function 'request_primitives' states that every method playing the role mp in a class playing the role cp is implemented and contains self invocations of all methods playing the role mp_2 in the same class.

```
value
  request_primitives :
    G.Class_Name × G.Method_Name × G.Method_Name × Wf_Design_Renaming → Bool
  request_primitives(cp, mp, mp2, (ds, r)) ≡
    (
      ∀ cd : G.Class_Name, dtem_meth, dpr_op : G.Method_Name •
        renaming_class_method2(cd, cp, dtem_meth, mp, dpr_op, mp2, r) ⇒
          (
            ∃ inv : M.Invocation •
              M.invocation_in_request_list(inv, DS.method_of(cd, dtem_meth, ds)) ∧
              M.meth_name(M.call_sig(inv)) = dpr_op ∧
              M.call_vble(inv) = G.self
          )
```

```
    )
```

The function 'ob_st_annotation' states that every method playing the role mp_1 in a class playing the role cp_1 is implemented and contains invocations of all methods playing the role mp_2 in classes playing the role cp_2, each invocation having a single parameter.

```
value
  ob_st_annotation :
    G.Class_Name × G.Class_Name × G.Method_Name × G.Method_Name ×
    Wf_Design_Renaming
      → Bool
  ob_st_annotation(cp1, cp2, mp1, mp2, (ds, r)) ≡
    (
      ∀ cd1, cd2 : G.Class_Name, md1, md2 : G.Method_Name •
        renaming_class_method(cd1, cp1, md1, mp1, r) ∧
        renaming_class_method(cd2, cp2, md2, mp2, r) ⇒
          (
            ∃ vdEl, vdVis : G.Variable_Name •
              let
                m = DS.method_of(cd1, md1, ds),
                s2 = M.mk_Actual_Signature(md2, ⟨vdVis⟩),
                inv1 = M.mk_Invocation(vdEl, s2)
              in
                M.invocation_in_request_list(inv1, m)
              end
          )
    )
```

The function 'Update_state_var_change_deleg_var' states that every method playing the role mp_1 in a class playing the role cp is implemented and makes assignments to every state variable playing the role vp_1 in the same class through invocations to state variables playing the role vp_2 of methods playing the role mp_2 in classes playing the role cp_2.

```
value
  Update_state_var_change_deleg_var :
    G.Class_Name × G.Method_Name × G.Variable_Name × G.Variable_Name ×
      G.Class_Name × G.Method_Name ×  Wf_Design_Renaming → Bool
  Update_state_var_change_deleg_var(cp, mp1, vp1, vp2, cp2, mp2, ((dsc, dsr), r)) ≡
    (
      ∀ cd : G.Class_Name, md : G.Method_Name, vd2 : G.Variable_Name •
        renaming_class_method_state(cd, cp, md, mp1, vd2, vp2, r) ⇒
          let
```

```
            m = DS.method_of(cd, md, (dsc, dsr)),
            vs = state_vars_renaming_to(cd, cp, vp1, r)
        in
            M.is_implemented(m) ∧
            vs ⊆ M.changed_variables(m) ∧
            (
                ∀ v : G.Variable_Name •
                    v ∈ vs ⇒
                        (
                            ∃ vset : G.Variable_Name-set, cd2 : G.Class_Name,
                              inv : M.Invocation •
                              let
                                vm = M.variable_change_body(m),
                                M.mk_Invocation(var, sig) = inv,
                                md2 = M.meth_name(sig)
                              in
                                vset ∈ dom vm ∧
                                v ∈ vset ∧
                                renaming_class_method(cd2, cp2, md2, mp2, r) ∧
                                var = vd2 ∧
                                R.exists_assoc_aggr_relation(vd2, cd, cd2, dsr)
                              end
                        )
            )
        end
    )
```

The function 'SetM_state_var_change_deleg_par' states that every method playing the role mp_1 in a class playing the role cp which has a parameter playing the role vp_2 is implemented and has a body which consists of a single invocation to its parameter of a method playing the role mp_2 in a class playing the role cp_2, the invocation having no parameters and the result of the invocation being assigned to the state variables playing the role vp_1 in the same class playing the role cp.

```
value
    SetM_state_var_change_deleg_par :
        G.Class_Name × G.Method_Name × G.Variable_Name × G.Variable_Name ×
            G.Class_Name × G.Method_Name × Wf_Design_Renaming → Bool
    SetM_state_var_change_deleg_par(cp, mp1, vp1, vp2, cp2, mp2, (ds, r)) ≡
        (
            ∀ cd : G.Class_Name, md1 : G.Method_Name, vd2 : G.Variable_Name •
                renaming_class_method_param(cd, cp, md1, mp1, vd2, vp2, r) ⇒
                    let
                        m = DS.method_of(cd, md1, ds),
```

```
            vs = state_vars_renaming_to(cd, cp, vp1, r)
        in
            M.is_implemented(m) ∧
            (
                ∃ cd2 : G.Class_Name, md2 : G.Method_Name •
                    renaming_class_method(cd2, cp2, md2, mp2, r) ∧
                    let
                        vm = M.variable_change_body(m),
                        rlb = M.request_list_body(m),
                        s = M.mk_Actual_Signature(md2, ⟨⟩),
                        inv = M.mk_Invocation(vd2, s)
                    in
                        vm = [ vs ↦ inv ] ∧ rlb = ⟨inv⟩
                    end
            )
        end
    )
```

The function 'use_interface' states that for every class cd_1 playing the role cp_1 and every method md_2 playing the role mp_2 in a class cd_2 which plays the role cp_2 there is a method in the class cd_1 which contains an invocation of the method md_2, the variable of the invocation representing an association or aggregation relation between the classes cd_1 and cd_2.

```
value
    use_interface :
        G.Class_Name × G.Class_Name × G.Method_Name × Wf_Design_Renaming → Bool
    use_interface(cp1, cp2, mp2, ((dsc, dsr), r)) ≡
        (
            ∀ cd1, cd2 : G.Class_Name, md2 : G.Method_Name •
                renaming_class_name(cp1, cd1, r) ∧
                renaming_class_method(cd2, cp2, md2, mp2, r) ⇒
                    (
                        ∃ md1 : G.Method_Name, i : M.Invocation •
                            DS.has_method(md1, cd1, (dsc, dsr)) ∧
                            let
                                c = DS.class_of_method(md1, cd1, (dsc, dsr)),
                                m = DS.method_of(cd1, md1, (dsc, dsr))
                            in
                                M.invocation_in_request_list(i, m) ∧
                                M.meth_name(M.call_sig(i)) = md2 ∧
                                R.exists_assoc_aggr_relation(M.call_vble(i), c, cd2, dsr)
                            end
                    )
        )
```

The function 'not_use_interface' states that classes playing the role cp_1 do not use the interface represented by methods playing the role mp_2 in classes playing the role cp_2, that is there is no method in any class playing the role cp_1 which contains an invocation of a method playing the role mp_2 in a class playing the role cp_2 and there is no association or aggregation relation between the two classes which corresponds to such an invocation.

```
value
  not_use_interface :
    G.Class_Name × G.Class_Name × G.Method_Name × Wf_Design_Renaming → Bool
  not_use_interface(cp1, cp2, mp2, ((dsc, dsr), r)) ≡
    (
      ∀ cd1, cd2 : G.Class_Name, md1, md2 : G.Method_Name, i : M.Invocation •
        renaming_class_name(cp1, cd1, r) ∧
        renaming_class_method(cd2, cp2, md2, mp2, r) ∧
        DS.has_method(md1, cd1, (dsc, dsr)) ∧
        let
          c = DS.class_of_method(md1, cd1, (dsc, dsr)),
          m = DS.method_of(cd1, md1, (dsc, dsr))
        in
          M.invocation_in_request_list(i, m)
        end ⇒
          M.meth_name(M.call_sig(i)) ≠ md2 ∧
          let c = DS.class_of_method(md1, cd1, (dsc, dsr)) in
            ~R.exists_assoc_aggr_relation(M.call_vble(i), c, cd2, dsr)
          end
    )
```

The function 'visitor' states that every method md playing the role mp in a class playing the role cp contains an invocation to a unique method playing the role mp_2 in a unique class playing the role cp_2, the invocation being to a parameter of md which plays the role vp and the only parameter of the invocation being 'self'.

```
value
  visitor :
    G.Class_Name × G.Method_Name × G.Variable_Name ×
      G.Class_Name × G.Method_Name × Wf_Design_Renaming → Bool
  visitor(cp, mp, vp, cp2, mp2, (ds, r)) ≡
    (
      ∀ cd : G.Class_Name, md : G.Method_Name, vd : G.Variable_Name •
        renaming_class_method_param(cd, cp, md, mp, vd, vp, r) ⇒
          (
            ∃! md2 : G.Method_Name, cd2 : G.Class_Name •
              renaming_class_method(cd2, cp2, md2, mp2, r) ∧
```

```
                    let
                        m = DS.method_of(cd, md, ds),
                        s = M.mk_Actual_Signature(md2, ⟨G.self⟩),
                        i = M.mk_Invocation(vd, s)
                    in
                        M.invocation_in_request_list(i, m)
                    end
            )
    )
```

The function 'different_create_iterator' checks that every method playing the role mp in a class playing the role cp is implemented and produces a single variable as result, this variable being in the variable change map. In addition, no two different methods playing this role have the same request or variable associated with the image of their result under their variable change map.

```
value
    different_create_iterator :
        G.Class_Name × G.Method_Name × Wf_Design_Renaming → Bool
    different_create_iterator(cp, mp, ((dsc, dsr), r)) ≡
        (
            ∀ cd : G.Class_Name, md1, md2 : G.Method_Name •
                renaming_class_method2(cd, cp, md1, mp, md2, mp, r) ∧
                md1 ≠ md2 ⇒
                    let
                        m1 = DS.method_of(cd, md1, (dsc, dsr)),
                        m2 = DS.method_of(cd, md2, (dsc, dsr))
                    in
                        M.is_implemented(m1) ∧
                        M.is_implemented(m2) ∧
                        card M.meth_res(m1) = 1 ∧
                        card M.meth_res(m2) = 1 ⇒
                            let
                                vd1 : G.Variable_Name • {vd1} = M.meth_res(m1),
                                vd2 : G.Variable_Name • {vd2} = M.meth_res(m2)
                            in
                                {vd1} ∈ dom M.variable_change_body(m1) ∧
                                {vd2} ∈ dom M.variable_change_body(m2) ∧
                                M.variable_change_body(m1)({vd1}) ≠
                                M.variable_change_body(m2)({vd2})
                            end
                    end
        )
```

The function 'never_operate' checks that classes playing the role cp contain no methods in which there is an invocation to a variable representing either an association or aggregation relation with a class playing the role cp_2.

```
value
  never_operate : G.Class_Name × G.Class_Name × Wf_Design_Renaming → Bool
  never_operate(cp, cp2, ((dsc, dsr), r)) ≡
    (
      ∀ cd, cd2 : G.Class_Name, md : G.Method_Name, inv : M.Invocation,
        vd : G.Variable_Name •
        renaming_class_name(cd, cp, r) ∧
        renaming_class_name(cd2, cp2, r) ∧
        R.exists_assoc_aggr_relation(vd, cd, cd2, dsr) ∧
        DS.has_method(md, cd, (dsc, dsr)) ∧
        M.invocation_in_request_list(inv, DS.method_of(cd, md, (dsc, dsr))) ⇒
          M.call_vble(inv) ≠ vd
    )
```

The function 'assign_invoke_param$_1$' checks that every method playing the role mp_1 and every state variable playing the role vp in a class playing the role cp_1 are related to every pair of methods playing the roles mp_2 and mp_3 in a class playing the role cp_2 via the function 'has_assignment_invocation_param' defined in Section 3.2 for some (dummy) variable v.

```
value
  assign_invoke_param1 :
    G.Class_Name × G.Variable_Name × G.Class_Name × G.Method_Name ×
      G.Method_Name × G.Method_Name × Wf_Design_Renaming → Bool
  assign_invoke_param1(cp1, vp, cp2, mp1, mp2, mp3, (ds, r)) ≡
    (
      ∀ cd1, cd2 : G.Class_Name, vd : G.Variable_Name,
        md1, md2, md3 : G.Method_Name •
        renaming_class_method_state(cd1, cp1, md1, mp1, vd, vp, r) ∧
        renaming_class_method2(cd2, cp2, md2, mp2, md3, mp3, r) ⇒
          let m = DS.method_of(cd1, md1, ds) in
            (
              ∃ v : G.Variable_Name •
                M.has_assignment_invocation_param(m, vd, v, md2, md3)
            )
          end
    )
```

The function 'assign_invoke_param$_2$' states that every method playing the role mp_1 in a class playing the role cp_1 is implemented and includes in its body an invocation to some variable v of

the method which plays the role mp_4 in a class playing the role cp_2, recording the result of this invocation in a state variable playing the role vp. Furthermore, for each method playing the role mp_1 there is at least one method playing the role mp_1 in the same class which is implemented and whose body includes an invocation to the same variable v of a method which plays the role mp_3 in the class playing the role cp_2, the variable vd being passed as the parameter to this invocation.

```
value
  assign_invoke_param2 :
    G.Class_Name × G.Variable_Name × G.Class_Name × G.Method_Name ×
      G.Method_Name × G.Method_Name × G.Method_Name × Wf_Design_Renaming
        → Bool
  assign_invoke_param2(cp1, vp, cp2, mp1, mp2, mp3, mp4, (ds, r)) ≡
    (
      ∀ cd1, cd2 : G.Class_Name, vd : G.Variable_Name,
        md1, md3, md4 : G.Method_Name •
        renaming_class_method_state(cd1, cp1, md1, mp1, vd, vp, r) ∧
        renaming_class_method2(cd2, cp2, md3, mp3, md4, mp4, r) ⇒
          (
            ∃ md2 : G.Method_Name, v : G.Variable_Name •
              renaming_class_method(cd1, cp1, md2, mp2, r) ∧
              let
                m1 = DS.method_of(cd1, md1, ds),
                m2 = DS.method_of(cd1, md2, ds)
              in
                M.has_assignment(m1, vd, v, md3) ∧
                M.has_invocation_param(m2, v, md4, vd)
              end
          )
    )
```

The function 'client_comment' states that every method playing the role mp_1 in a class playing the role cp_1 is implemented and contains in its body an instantiation of a class playing either the role cp_2 or the role cp_5. The instantiation of the cp_5 role receives no parameters, while the instantiation of the cp_2 role receives a single parameter which is generally a variable representing a relation between the class playing the role cp_1 and some class playing the cp_3 role but which can also be 'self' if the class playing the role cp_1 also plays the role cp_3. The result of this instantiation is assigned to a variable, and this variable is then passed as the sole parameter to an invocation of a method playing the role mp_2 in a class playing the role cp_4, this invocation being possibly indirect as described by the function 'invokes' defined in Section 3.5.

```
value
  client_comment :
```

```
    G.Class_Name × G.Class_Name × G.Class_Name × G.Class_Name × G.Class_Name ×
      G.Method_Name × G.Method_Name × Wf_Design_Renaming → Bool
client_comment(cp1, cp2, cp3, cp4, cp5, mp1, mp2, ((dsc, dsr), r)) ≡
  (
    ∀ cd1 : G.Class_Name, md1 : G.Method_Name •
      renaming_class_method(cd1, cp1, md1, mp1, r) ⇒
        let m = DS.method_of(cd1, md1, (dsc, dsr)) in
          M.is_implemented(m) ∧
          (
            ∃ ins : M.Instantiation •
              M.instantiation_in_request_list(ins, m) ∧
              (
                renaming_class_name(M.class_name(ins), cp2, r) ∨
                renaming_class_name(M.class_name(ins), cp5, r)
              )
          ) ∧
          (
            ∀ i : M.Instantiation •
              let M.mk_Instantiation(c, p) = i in
                M.instantiation_in_request_list(i, m) ∧
                (
                  renaming_class_name(c, cp2, r) ∨
                  renaming_class_name(c, cp5, r)
                ) ⇒
                  (
                    ∃ cd3, cd4 : G.Class_Name, md2 : G.Method_Name,
                      v : G.Variable_Name •
                      let vm = M.variable_change_body(m) in
                        {v} ∈ dom vm ∧
                        vm({v}) = i ∧
                        renaming_class_name(cd3, cp3, r) ∧
                        renaming_class_method(cd4, cp4, md2, mp2, r) ∧
                        (
                          renaming_class_name(c, cp2, r) ⇒
                            len p = 1 ∧
                            if cd3 = cd1 then
                              p = ⟨G.self⟩
                            else
                              R.exists_assoc_aggr_relation
                                (hd (p), cd1, cd3, dsr)
                            end
                        ) ∧
                        (renaming_class_name(c, cp5, r) ⇒ p = ⟨⟩) ∧
                        DS.invokes
                          (
```

```
                                    M.method_cut(m, i), md2, 1, v, cd1, cd4, (dsc, dsr)
                                )
                            end
                        )
                    end
                )
            end
    )
```

The function 'st_com_body' checks that every method which plays the role mp_1 in a class playing the role cp_1 is implemented, has no result, and has a single parameter which plays the role vp_1. In addition, the body of the method simply assigns this parameter to the state variables playing the role vp_2.

```
value
  st_com_body :
    G.Class_Name × G.Method_Name × G.Variable_Name × G.Variable_Name ×
      Wf_Design_Renaming → Bool
  st_com_body(cp1, mp1, vp1, vp2, ((dsc, dsr), r)) ≡
    (
      ∀ cd : G.Class_Name, md : G.Method_Name, vd1, vd2 : G.Variable_Name •
        renaming_class_state(cd, cp1, vd2, vp2, r) ∧
        renaming_class_method_param(cd, cp1, md, mp1, vd1, vp1, r) ⇒
          let
            m = DS.method_of(cd, md, (dsc, dsr)),
            vm = M.variable_change_body(m),
            rlb = M.request_list_body(m)
          in
            M.is_implemented(m) ∧
            rlb = ⟨⟩ ∧
            vm = [ {vd2} ↦ M.Request_or_Var_from_Variables({vd1}) ]
          end
    )
```

The function 'lvar_chng_inv_deleg' checks that every method which plays the role mp_1 in a class playing the role cp_1 is implemented and has a body which contains an invocation of a method playing the role mp_2 in a class playing the role cp_2, the result of which is assigned to a variable. Another method playing the same role mp_1 contains an invocation to this variable of a method playing the role mp_2 in a class playing the role cp_2. If the two invocations are in fact in the same method then they must appear in the order stated.

```
value
```

```
lvar_chng_inv_deleg :
   G.Class_Name × G.Method_Name × G.Class_Name × G.Method_Name ×
      G.Class_Name × G.Method_Name × Wf_Design_Renaming → Bool
lvar_chng_inv_deleg(cp1, mp1, cp2, mp2, cp3, mp3, ((dsc, dsr), r)) ≡
   (
      ∀ cd1, cd2, cd3 : G.Class_Name, md1a, md1b, md2, md3 : G.Method_Name •
         renaming_class_method2(cd1, cp1, md1a, mp1, md1b, mp1, r) ∧
         renaming_class_method(cd2, cp2, md2, mp2, r) ∧
         renaming_class_method(cd3, cp3, md3, mp3, r) ⇒
            (
               ∃ inv1, inv2 : M.Invocation, vd : G.Variable_Name •
                  DS.has_method(md1a, cd1, (dsc, dsr)) ∧
                  DS.has_method(md1b, cd1, (dsc, dsr)) ∧
                  let
                     c1a = DS.class_of_method(md1a, cd1, (dsc, dsr)),
                     m1a = DS.method_of(cd1, md1a, (dsc, dsr)),
                     c1b = DS.class_of_method(md1b, cd1, (dsc, dsr)),
                     m1b = DS.method_of(cd1, md1b, (dsc, dsr))
                  in
                     M.is_implemented(m1a) ∧
                     vd ∈ M.changed_variables(m1a) ∧
                     M.variable_change_body(m1a)({vd}) =
                     M.Request_from_Invocation(inv1) ∧
                     M.meth_name(M.call_sig(inv1)) = md2 ∧
                     R.exists_assoc_aggr_relation(M.call_vble(inv1), c1a, cd2, dsr) ∧
                     M.invocation_in_request_list(inv2, m1b) ∧
                     M.call_vble(inv2) = vd ∧
                     M.meth_name(M.call_sig(inv2)) = md3 ∧
                     R.exists_assoc_aggr_relation(vd, c1a, cd3, dsr) ∧
                     (m1a = m1b ⇒ M.order(inv1, inv2, M.request_list_body(m1a)))
                  end
            )
   )
```

Finally, the function 'res_chng_vble_alternative' checks that every method which plays the role mp_1 in a class playing the role cp_1 is implemented, has a parameter playing the role vp_2, has a body which contains an alternative, assigns the result of evaluating that alternative to a variable, and returns the value of that variable as its result. The first block of the alternative contains an invocation of the method mp_2 with the above parameter on a state variable playing the role vp_1. The second block contains an instantiation of a class playing the role cp_2, followed by an invocation of the method mp_3 to the same state variable, the result of the instantiation being a parameter of the invocation.

```
value
```

```
res_chng_vble_alternative :
   G.Class_Name × G.Method_Name × G.Variable_Name × G.Variable_Name ×
      G.Method_Name × G.Class_Name × G.Method_Name × Wf_Design_Renaming
         → Bool
res_chng_vble_alternative(cp1, mp1, vp1, vp2, mp2, cp2, mp3, (ds, r)) ≡
   (
      ∀ cd1, cd2 : G.Class_Name, md : G.Method_Name, vd1, vd2 : G.Variable_Name •
         renaming_class_method_param(cd1, cp1, md, mp1, vd2, vp2, r) ∧
         renaming_class_state(cd1, cp1, vd1, vp1, r) ∧
         renaming_class_name(cd2, cp2, r) ⇒
         let
            m = DS.method_of(cd1, md, ds),
            sig = M.mk_Actual_Signature(mp2, ⟨vd2⟩),
            inv1 = M.mk_Invocation(vd1, sig)
         in
            ∃ vd3 : G.Variable_Name, blk1, blk2 : M.Block, inst : M.Instantiation,
               inv2 : M.Invocation •
               M.is_implemented(m) ∧
               M.Request_from_Invocation(inv1) ∈ elems blk1 ∧
               M.Request_from_Instantiation(inst) ∈ elems blk2 ∧
               M.class_name(inst) = cd2 ∧
               M.Request_from_Invocation(inv2) ∈ elems blk2 ∧
               M.call_vble(inv2) = vd1 ∧
               M.meth_name(M.call_sig(inv2)) = mp3 ∧
               vd3 ∈ elems M.a_params(M.call_sig(inv2)) ∧
               M.order(inst, inv2, blk2) ∧
               vd3 ∈ M.changed_variables(m) ∧
               M.variable_change_body(m)({vd3}) =
               M.Request_from_Alternative(blk1, blk2) ∧
               M.meth_res(m) = {vd3}
         end
   )
```

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. Combining the model with specifications of the specific properties of individual GoF patterns as in [18, 11, 4] then gives a way of formally determining whether or not a given design matches a given pattern, thus allowing designers to be sure, as well as to demonstrate to others that they are using the patterns correctly and consistently.

The model can also help designers to understand the properties of the GoF patterns clearly, and indeed the analyses of the various GoF patterns using the model presented in [18, 11, 4] have identified a number of inconsistencies and incompletenesses in the informal descriptions of a number of patterns and has led to the proposal of modified pattern structures which resolve these problems.

The work presented here concentrates on matching a subset of a design to a single pattern at a time, whereas in practice a design is of course likely to be based around several different patterns and may even comprise several instances of the same pattern. This can be taken into account by generalising the renaming map (see [3]).

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 based on the extended OMT notation. We also believe that our work could form a strong basis for a similar model of an object-oriented design based on the UML notation (http://www.omg.org/uml), and we propose to investigate this in the future.

References

[1] C. Alexander, S. Ishikawa, M. Silverstein, M. Jacobson, I. Fiksdahl-King, and S. Angel. *A Pattern Language*. Oxford University Press, 1977.

[2] Brad Appleton. Patterns and Software: Essential Concepts and Terminology. http://www.enteract.com/~bradapp, November 1997.

[3] Gabriela Aranda and Richard Moore. Formally Modelling Compound Design Patterns. Technical Report 225, UNU/IIST, P.O. Box 3058, Macau, December 2000.

[4] Gabriela Aranda and Richard Moore. GoF Creational Patterns: A Formal Specification. Technical Report 224, UNU/IIST, P.O. Box 3058, Macau, December 2000.

[5] 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.

[6] Alejandra Cechich and Richard Moore. A Formal Specification of GoF Design Patterns. Technical Report 151, UNU/IIST, P.O.Box 3058, Macau, January 1999. Presented at and published in the proceedings of 6th Asia-Pacific Software Engineering Conference (APSEC'99) Takamatsu, Japan, December 7-10, 1999, IEEE Computer Society Press, pp. 284–291.

[7] 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.

[8] Peter Coad. *Object Models - Strategies, Patterns, and Applications.* Prentice-Hall, 1995.

[9] 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.

[10] A. Eden, Y. Hirshfeld, and A. Yehudai. LePUS - A Declarative Pattern Specification Language. http://www.math.tau.ac.il/~eden/bibliography.html.

[11] Andres Flores and Richard Moore. GoF Structural Patterns: A Formal Specification. Technical Report 207, UNU/IIST, P.O. Box 3058, Macau, August 2000. Presented at and published in the proceedings of the IASTED International Conference on Applied Informatics (AI 2001), Innsbruck, Austria, 19-22 February 2001, pp. 625–630.

[12] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison Wesley, 1995.

[13] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison Wesley Professional Computing Series. Addison Wesley, 1995.

[14] 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.

[15] L. Lamport. The Temporal Logic of Actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, May 1994.

[16] Tommi Mikkonen. Formalizing Design Patterns. In *Proceedings of the International Conference on Software Engineering ICSE'98*, pages 115–124. IEEE Computer Society Press, 1998.

[17] The RAISE Language Group. *The RAISE Specification Language.* BCS Practitioner Series. Prentice Hall, 1992. Available from Terma A/S. Contact jnp@terma.com.

[18] Luis Reynoso and Richard Moore. GoF Behavioural Patterns: A Formal Specification. Technical Report 201, UNU/IIST, P.O. Box 3058, Macau, May 2000. Presented at and published in the proceedings of the ACIS 2nd International Conference on Software Engineering, Artificial Intelligence, Networking & Parallel/Distributed Computing (SNPD'01), Nagoya, Japan, August 2001, pp. 262–270.

[19] J. Rumbaugh. *Object-Oriented Modeling and Design.* Prentice Hall, 1991.