Specifying Behavioural Features of Design Patterns in First Order Logic

Ian Bayley and Hong Zhu
Department of Computing, Oxford Brookes University, Wheatley, Oxfordshire, OX33 1HX, UK

Abstract

The formal specification of design patterns is widely recognised as being vital to their effective and correct use in software development. It can clarify the concepts underlying patterns, eliminate ambiguity and thereby lay a solid foundation for tool support. This paper further advances an approach that uses first order predicate logic to specify design patterns by capturing the dynamic behaviour represented in sequence diagrams. A case study of all 23 patterns in the Gang of Four catalogue demonstrates that it can not only capture dynamic features but also simplify the specification of structural properties.

1 Introduction

Software design patterns are a technique for documenting solutions to recurring design problems and for sharing design expertise in an application-independent fashion [4, 3, 8]. They are commonly presented in Alexandrian form, in which design principles are first explained in informal English, and then clarified with illustrative class diagrams and specific code examples [8]. This format is informative enough for humans to understand the design principle and to learn how to apply patterns to solve their own problems, but it is also informal enough to risk ambiguity. It is widely recognised that poorly presented patterns can result in poor system quality [18]. The formal specification of patterns employs mathematical theories and notations that help to eliminate this ambiguity and to clarify the notions underlying the patterns. Moreover, it is not enough to understand individual patterns in isolation. They need to be catalogued [21] and combined to solve real-world problems. Formal specification can also lay a solid foundation for reasoning about properties of patterns and the relationships between them, as shown in [1]. However, as discussed in section 2, despite of the large research effort in the past few years, existing techniques of pattern specification have not satisfactorily captured all features of software patterns, especially the dynamic features.

This paper is concerned with the specification of dynamic features of patterns. It advances the approach proposed in [1] by employing a first-order predicate logic defined on a domain containing both class and sequence diagrams. We report a successful case study of the patterns in the Gang of Four (GoF) book [8].

The remainder of the paper is organised as follows. Section 2 briefly reviews related work and discusses the difficulties of specifying behavioural features. Section 3 describes the proposed approach. Section 4 illustrates the proposed method by a number of examples. Section 5 analyses the results of the case study on the patterns of the GoF book [8]. Section 6 concludes the paper with a discussion of the advantages of the approach and directions for future work.

2 Related work and open problems

In the past few years, many research efforts have focused on the formal specification of software design patterns. Existing work can be classified into two categories. The first category proposes special-purpose formal languages or semi-formal graphic modeling languages in order to define patterns rigorously. The second category, to which our work belongs, simply employs or adapts existing formal or semi-formal languages.

Among the work in the first category is the Design Pattern Modelling Language of Mapelsden et al [14], which defines a whole new language just for patterns. Similarly, Eden [5, 6] devised a new graphical language LePUS for the purpose of modelling patterns. In the second category, Taibi [19, 20] formalises class diagrams as relations between program elements, specifies post-conditions with predicate logic and describes the desired behavior with temporal logic. Mikkonen [15] formalises temporal behaviours in a temporal logic of actions. Le Guennec et al [9] extend the UML meta-model to incorporate collaboration occurrences and use the Object Constraint Language (OCL) to constrain the collaborations. Mak et al [13] define the notion of collaborations by extending UML to action semantics. France et al [7] also uses UML meta-modelling facility to describe the structural properties in class diagrams and dynamic properties in sequence diagrams, and semantic information by templates of OCL constraints. Zdun et al [22] identified architectural primitives that occur in patterns from the component-and-connector view. Finally, Kodituwakku and Bertok [10] use category theory to formally define the relationships between patterns and study

0730-3157/08 $25.00 © 2008 IEEE
DOI 10.1109/COMPSAC.2008.67

the mathematical structure of pattern organisations. While each of these approaches are demonstrated with examples, it remains an open question whether they can be used to specify all design patterns.

Tools have also been developed to recognise patterns by analysing source code [16]. Also focusing at the code level, Lano *et al.* [11] consider patterns to be transformations from flawed solutions to improved solutions. Lauder and Kent [12] propose a three layer modeling approach consisting of role models, type models and concrete class models. The disadvantage of focusing at code level is that many behavioural properties are hard to determine. Moreover, although tools like PINOT [16] are desirable, design-level tools are preferable as they would help designers avoid errors at the earlier design stage. Better still would be to develop tools like PINOT in such a manner that they can be proven correct with respect to a formal specification.

Recently, Bayley and Zhu [1] have also advanced a method for the formal specification of patterns using predicate logic defined on the domain of UML class diagrams. All 23 patterns in [8] are formally specified as predicates in the first-order logic on the domain of class diagrams. They demonstrated how a concrete design represented in a UML class diagram can be recognised as an instance of a pattern by proving the satisfaction of the predicate. They also shown how properties of design patterns can be proved in first-order logic. This approach has many advantages over its rivals. First, the specifications are easy to understand and readable by both humans and computers. The notation is expressive too as demonstrated by its successful application to all 23 patterns in the GoF book [8]. Moreover, reasoning about the properties of patterns and their relationships can be done using inference in first-order logic, which is well-understood and supported by software tools. A similar alternative approach is simply to use OCL, but OCL is not designed for the meta-level and even when lifted, it cannot specify the *absence* of a relationship between classes. Other problems with OCL were noted by France et al in [7].

However, although the work in [1] characterises well the structural properties of patterns, by relying on the design information contained in class diagrams, it shares with most other approaches (an exception being [9] and [7]) the major flaw that dynamic properties cannot be captured. These are the properties we observe at runtime. Examples include a message being sent to all instances of a class and a specific message being sent to a specific object only after a certain event happens at runtime. For instance, the Observer pattern [8] has the dynamic property that *"all observers are notified whenever the subject undergoes a change in state."* Dynamic properties are usually stated as comments in the class diagram, and/or as explanatory text in the Alexandrian form, or at best, illustrated using sequence diagrams. The ambiguities in both of these forms is the reason why such properties are difficult to specify. As with structural properties in [1], the formalisation of behavioural properties is all about clarifying the underlying principles.

This paper advances the approach proposed in [1] by specifying dynamic properties, based on information contained in UML sequence diagrams. The choice of sequence diagram is because they are more widely used than other diagrams of this sort, and contain most of the important information about dynamic behaviour.

3 Specification of patterns as meta-modelling

Each pattern is a subset of design models with certain structural and behavioural features. Therefore, the formal specification of patterns is a meta-modelling problem. As in [1], our approach to meta-modelling is first to define the domain of all models by an abstract syntax in the meta-notation GEBNF [23], which stands for Graphic Extension of BNF. It extends traditional BNF notation with a 'reference' facility to define the graphic structures of diagrams. Then, for each design pattern, we define a first-order predicate to constrain the models such that each model that satisfies the predicate is an instance of the pattern. So, a meta-model in our approach comprises an abstract syntax in GEBNF plus a first-order predicate.

3.1 The Domain of Models

In this subsection, we first review the meta-notation GEBNF [23, 1] and then use it to define the domain of models for class diagrams and sequence diagrams.

3.1.1 GEBNF Notation

In GEBNF, the abstract syntax of a modeling language is defined as a tuple $\langle R, N, T, S\rangle$, where N is a finite set of non-terminal symbols, and T is a finite set of terminal symbols, each of which represents a set of values. Furthermore, $R \in N$ is the root symbol and S is a finite set of production rules of the form $Y ::= Exp$, where $Y \in N$ and Exp can be in one of the following forms.

$$L_1 : X_1, L_2 : X_2, \cdots, L_n : X_n$$
$$X_1 | X_2 | \cdots | X_n$$

where L_1, L_2 , $\cdots$, L_n are field names, and X_1, X_2 , $\cdots$, X_n are the fields. Each field can be in one of the following forms: Y, $Y*$, $Y+$, $[Y]$, $\underline{Y}$, where $Y \in N \cup T$.

The meaning of the meta-notation is given in Table 1. Note that where an element is underlined, it is a reference to an existing element on the diagram as opposed to the introduction of a new element.

Table 1. Meanings of the GEBNF Notation

Notation	Meaning	Example and explanation
$X_1 \mid \ldots \mid X_n$	Choice of $X_1, X_2, \ldots, X_n$	*ActorNode* | *UseCaseNode* means that the entity is either an actor node or a use case node.
$L_1{:}\,X_1$, $L_2{:}\,X_2$, … $L_k{:}\,X_k$	Ordered sequence consists of k fields of type $X_1, X_2, \ldots, X_k$ that can be access by the field names $L_1, L_2, \ldots, L_k$.	*ClassName: Text Attributes: Attribute* Methods: Method** means that the entity consists of three parts called *Classname*, *Attributes* and *Methods* respectively.
X*	Repetition of X (include null)	*Diagram** means that the entity consists of a number N of diagrams, where $N \geq 0$.
X+	Repetition of X (exclude null)	*Diagram*+ means that the entity consists of a number N of diagrams, where $N \geq 1$.
[X]	X is optional	[*Actor*]: element of actor is optional.
$\underline{X}$	Reference to an exiting element of type X in the model	*ClassNode* is a reference to an existing class node.

3.1.2 Class Diagrams

The GEBNF definition of UML class diagrams is obtained from [17] by removing those attributes not required to describe patterns, and by flattening the hierarchy in to eliminate some meta-classes for simplicity.

A class diagram consists of classes, linked with association, inheritance and whole-part ($compag$ for $comp$osite or aggregate) relations between them. A class has a name, attributes, and operations.

$$ClassDiagram ::= classes : Class^+,$$
$$assocs : Rel^*, inherits : Rel^*, compag : Rel^*$$
$$Class ::= name : String,$$
$$[attrs : Property^*], [opers : Operation^*]$$

Here, $String$ denotes the type of strings of characters.

An operation has a name, parameters and five flags. Each parameter has a name, type, optional multiplicity information and direction. Since return values play much the same role as out parameters, they are treated as just another sort of parameter, as in [17].

$$Operation ::= name : String, [params : Parameter^*],$$
$$[isAbstract : Bool], [isQuery : Bool],$$
$$[isLeaf : Bool], [isNew : Bool], [isStatic : Bool]$$
$$Parameter ::= [name : String], [type : Type],$$
$$[direction : ParaDirKind], [mult : Multiplicity]$$
$$ParaDirKind ::= \text{“}in\text{”} \mid \text{“}inout\text{”} \mid \text{“}out\text{”} \mid \text{“}return\text{”}$$
$$Multiplicity ::= [lower : Natural],$$
$$[upper : Natural \mid \text{“}*\text{”}]$$

Here, $Natural$ denotes the type of natural numbers and $Bool$ denotes the type of boolean values.

A property has a name, type, multiplicity information and a flag $isStatic$.

$$Property ::= name : String, type : Type,$$
$$[isStatic : Bool], [mult : Multiplicity]$$

Similarly, relationships between classes can be defined as follows.

$$Rel ::= [name : String], source : End, end : End$$
$$End ::= node : \underline{Class},$$
$$[name : String], [mult : Multiplicity]$$

In the sequel, when there is no risk of confusion, we will also use the name field of a classifier as its identifier.

3.1.3 Sequence Diagrams

A sequence diagram is an ordered collection of messages sent between lifelines. Each lifelines has a class and a collection of activations. It can be either an object lifeline ($isStatic = false$), in which case they may have a name, or a class lifeline ($isStatic = true$), in which case they don't. Here, we need only consider synchronous messages for the sake of simplicity.

$$SequenceDiagram ::=$$
$$lifelines : Lifeline^*, msgs : Message^*,$$
$$ordering : (\underline{Message}, \underline{Message})^*$$
$$Lifeline ::= activations : \underline{Activation}^*,$$
$$className : String, [objectName : String],$$
$$isStatic : Bool$$

The actions of sending, receiving and returning from (activations started by) messages are all events, so both activations and messages must refer to events. Messages also refer to operations in the class diagrams, which include parameters, and hence return values.

$$Activation ::= start, finish : Event, others : Event^*$$
$$Message ::= send, receive : \underline{Event}, sig : \underline{Operation}$$

3.2 Predicates on Diagrams

As shown in [23], an abstract syntax in GEBNF induces a first-order language for writing first-order predicates as constraints on the models.

In a GEBNF definition, every field $f : X$ of a term T introduces a function $f : T \to X$. Function application is written $x.f$ for function f and argument x. For example, because $opers : Operation^*$ is a field of $Class$, then $C.opers$ denotes the set of operations in class C.When there is just one class diagram or one sequence diagram, functions on them are written without their arguments, as $classes$, $lifelines$ etc. From functions deduced from GEBNF syntax, first-order predicates can be defined as usual using relations and operators on sets and basic data types and logic connectives and quantifiers. Further functions and relations can be defined as usual in the first-order logic. For the sake of readability, we will also use infix and prefix forms for defined functions and relations. Thus, we then write the application of function f to argument x with the more conventional prefix notation $f(x)$. Here follows some functions and relations used in many patterns.

Let $bounds(x) = (x.mult.lower, x.mult.upper)$, for $x : End$. We write $C_1 \diamond\!\longrightarrow C_2$ for the relation $r \in compag$ such that $r.source.node = C_1$, $r.end.node = C_2$, $bounds(r.source) = (1,1)$ and $bounds(r.end) = (1,1)$. Let $C \diamond\!\longrightarrow^* C'$ be similar but with $bounds(r.end) =$

$(1, *)$. Let $\longrightarrow$ and $\longrightarrow^*$ be the equivalent syntactic sugar for $assocs$.

Let C be a class. Then $subs(C)$ denotes the set of concrete subclasses of C. $C..op$ denote the redefinition of op for class C. We define $isAbstract(C) \equiv \exists op \in C.opers \cdot (op.isAbstract)$ and we write $allAbstract(ops)$ when $op.isAbstract$ is true for every $op \in ops$.

Let m and m' be messages. We will write $m < m'$, if $(m, m') \in ordering$. We define $fromAct(m)$ to be the unique activation a such that $m.send \in a.others$, $fromLL(m)$ to be the unique lifeline l such that $fromAct(m) \in l.activations$, and $fromClass(m)$ to abbreviate $fromLL(m).class$. Similarly, we define $toAct(m)$, $toLL(m)$ and $toClass(m)$. Finally, $trigs(m, m')$ means that message m starts (or "triggers") an activation that sends message m' or, more formally, $toAct(m) = fromAct(m')$.

For operations op and op' we define $calls$ below, and promote it to classes. A much-used predicate is $callsHook$, defined when an operation calls another at the root of an inheritance hierarchy.

$$calls(op, op') \equiv \exists m, m' \in msgs \cdot (m.sig = op \wedge m'.sig = op' \wedge trigs(m, m'))$$

$$calls(C, C') \equiv \exists m \in msgs \cdot (fromClass(m) = C \wedge toClass(m) = C')$$

$$callsHook(op, op') \equiv \exists C \in subs(C') \cdot calls(op, C.op')$$

For all messages m and objects o, we define that $hasReturnParam(m, o)$ is true if o is the return parameter for m. If there is only one such o for a message m, we write $returns(m) = o$.

3.3 Consistency Constraints

We define patterns only for design models that are well-formed and consistent with respect to a set of constraints [23]. There are so-called *intra-diagram* constraints, which affect the diagrams in isolation, and *inter-diagram* constraints, which concern the way that two diagrams must work together. Inter-diagram constraints for class diagrams include the constraints on operations already mentioned, the inheritance of attributes, operations and associations, that all classes must have different names (and operations and attributes within the same class must do too), that every abstract class is subclassed by a concrete class and that inheritance is irreflexive, and so on. For sequence diagrams, we require that every message must start an activation.

$$\forall m \in msgs \cdot \exists l \in lifelines, a \in activations(l) \cdot (m.receive = a.start)$$

Note that this constraint would not be necessary for parallel machines where two versions of the same operation can be executed simultaneously.

Inter-diagram constraints between the class and sequence diagrams, include the following.

- Every message to an activation must be for an operation of a concrete class:

  $$\forall m \in msgs \cdot (m.sig \in toClass(m).opers \wedge \neg isAbstract(toClass(m)))$$

- if a message is for a static operation, the lifeline must be a class lifeline; but if a message is for a non-static operation, the lifeline must be an object lifeline:

  $$\forall m \in msgs \cdot (m.sig.isStatic \Rightarrow toLL(m).isStatic \wedge \neg m.sig.isStatic \Rightarrow \neg toLL(m).isStatic)$$

- every class in the class diagram must appear in the sequence diagram:

  $$\forall C \in classes \cdot \exists l \in lifelines \cdot (l.class = C.name)$$

Descriptions of patterns in the literature sometimes violate such consistency constrains. For example, in [8], the Builder pattern breaks the final constraint with the Product class.

4 Examples of Formalisation

Both the structural and behavioural features of patterns can be formally specified as predicates on diagrams in the same way as consistency constraints. We have successfully specified all 23 patterns in the GoF book [8]. Here, we only give some examples to illustrate the style. A complete list of all specifications can be found in [2].

For each pattern, the formal specification consists of three parts. The first part, entitled *Components*, declares a set of variables, which are existentially quantified over the scope of all predicates in the specification of the pattern. In this way, it sets the background for the formulae by asserting the existence of certain components in the system design. The second part, entitled *Static Conditions*, consists of a number of predicates for the structural relations between the components. Such predicates can be evaluated using the information contained in the class diagram of a design. The third part, entitled *Dynamic Conditions*, consists of a number of predicates for the dynamic behaviour of the system, using information in the sequence diagram of a design, and sometimes in the class diagram too. In the latter case, consistency between the diagrams is ensured by the consistency constraints in subsection 3.3. We omit the text descriptions, context and solutions to save space, but we include the diagrams from the GoF book for the sake of readability.

We start with a simple example, the Adapter pattern.

4.1 Adapter

The structure of Adapter pattern is shown in Figure 1. There are four participants, $Target$, $Client$, $Adapter$ and

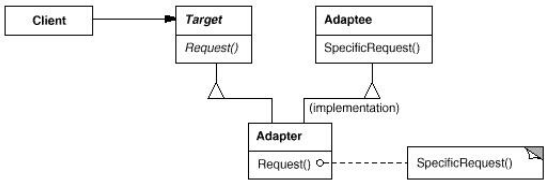

Figure 1. Adapter pattern class diagram

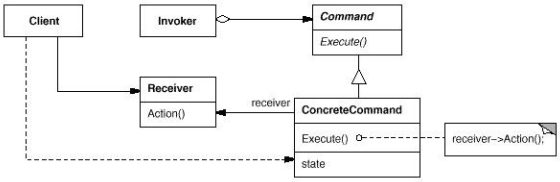

Figure 2. Command pattern class diagram

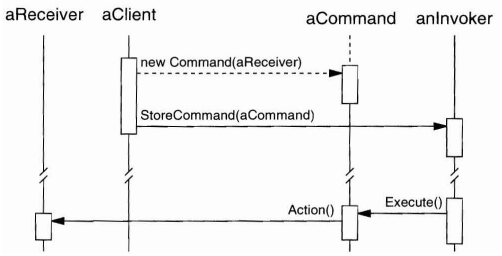

Figure 3. Command pattern seq diagram

$Adaptee$, so they are all declared as components, with the exception of $Client$ which may not necessarily be a specific class in the system, although it often is.

Components

- $Target, Adapter, Adaptee \in classes$
- $requests \in Target.opers$
- $specreqs \in Adaptee.opers$

The most important property of $Client$ is that it only accesses and depends on $Target$ but not any other components, such as $Adaptee$. This illustrates a common situation, in which there is a relationship from a class $Client$ to the root of a class hierarchy. It means that if a message is sent from a class that is not explicitly mentioned in the pattern then the operation must be declared in the root class. So, we write $CDR(C)$, short for client depends on root, where C is the root class. Formally,

$$\begin{aligned} &CDR(C) \equiv \\ &\quad \forall m \in msgs \cdot (toClass(m) \in subs(C) \\ &\quad \Rightarrow m.sig \in toClass(m).opers \\ &\quad \wedge \exists o \in opers.C \cdot (toClass(m).o = m.sig)) \end{aligned}$$

So the structural features of the Adapter pattern can be specified as follows.

Static Conditions

- $Adapter \longrightarrow\!\!\triangleright\ Target$
- $Adapter \longrightarrow Adaptee$
- $CDR(Target)$

The key dynamic feature of the Adapter pattern is that for every client call to the Adapter's operations, the Adapter calls the Adaptee's operations to carry out the request. This can be specified as follows.

Dynamic Conditions

- a request is delegated to a specific object

$$\forall o \in requests \cdot \exists o' \in specreqs \cdot (calls(o, o'))$$

A complete specification of the Adapter pattern can be assembled from the three parts by removing the comments in English, which were inserted for the sake of readability.

Note that the specification given above is for Object Adapter. The Class Adapter pattern has the static condition $Adapter \longrightarrow\!\!\triangleright\ Adaptee$ instead of $Adapter \longrightarrow Adaptee$, and we must also capture the condition that it is only the $Adapter$ class that can send a message to the $Adaptee$.

$$\begin{aligned} &\forall m \in msgs \cdot (toClass(m) = Adaptee \Rightarrow \\ &\quad fromClass(m) = Adapter) \end{aligned}$$

Note that Adapter is typical of the structural patterns in the GoF catalogue, as it has rich structural features, but also some dynamic features. Note too that the structural features of the pattern are specified more simply and clearly than [1], where only the class diagram is used. In fact, this is true for almost all patterns in the GoF catalog [8]. See Section 5 for more details.

4.2 Command

Command is typical of the behavioural patterns in the GoF catalog, in that it is rich in dynamic features. Figure 2 shows the structure of the pattern, as captured in the $Component$ and $StaticCondition$ parts of the specification, below.

Components

- $Command, ConcreteCommand,$ $Invoker, Receiver \in classes,$
- $execute \in Command.opers,$
- $action \in Receiver.opers$

Static Conditions

- $Invoker \diamond\!\!\longrightarrow Command$
- $ConcreteCommand \longrightarrow Receiver$
- $ConcreteCommand \longrightarrow\!\!\triangleright\ Command$
- $execute.isAbstract$
- $\neg isAbstract(ConcreteCommand)$

In the GoF catalogue, the dynamic features of a pattern are described in the Collaborations section, which is sometimes illustrated by a sequence diagram. The sequence diagram for Command pattern is given in Figure 3.

To specify the dynamic features of a pattern, we often split the *Dynamic Conditions* into two sub-parts: the *An-*

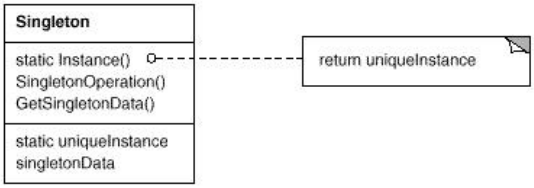

Figure 4. Singleton pattern class diagram

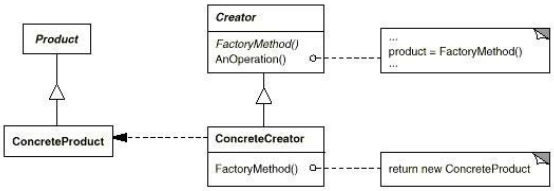

Figure 5. Factory method class diagram

tecedent and the *Consequent*. The former specifies the condition or scenario in which the behavior happens. The latter specifies the behavior itself. For the Command pattern, the trigger is a call to the method $execute$.

Dynamic Conditions - Antecedent

- when a command is executed then

$$\forall me \in msgs \cdot \\ me.sig = ConcreteCommand.execute$$

Dynamic Conditions - Consequent

- the invoker is responsible, and

$$fromLL(me).class = Invoker$$

- the receiver will perform an action at once and

$$\exists ma \in msgs \cdot (calls(me, ma) \land ma.sig = action)$$

- the command to be executed is created and

$$\exists mn \in msgs \cdot \\ isNew(mn.sig) \land toLL(mn) = toLL(me)$$

- the command is stored in the invoker and

$$\exists ms \in msgs \cdot (ms.sig = storeCommand \land \\ fromAct(ms) = fromAct(mn))$$

- the command was created with the receiver before the command was stored before it was executed

$$(mn < ms) \land (ms < me) \land \\ hasParam(mn, toLL(ma).name) \land \\ hasParam(ms, toLL(mn).name)$$

This captures the dynamic information that would have been missed had we restricted our attention to the static properties considered by [1]. This is particularly important for patterns where the static properties are trivial, such as the single-class Singleton pattern.

4.3 Singleton

The Singleton pattern is a creational pattern with a simple structure shown in Figure 4, but with dominant dynamic behaviour, though its GoF description contains no sequence diagram.

Components

- $Singleton \in classes$
- $getInstance \in Singleton.opers$

Static Conditions

- $getInstance.isStatic$

Dynamic Conditions - Antecedent

- when a new $Singleton$ object is created

$$\forall m \in msgs \cdot \\ isNew(m.sig) \land toClass(m) = Singleton$$

Dynamic Conditions - Consequent

- this must be triggered by a request for an instance

$$\exists m' \in msgs \cdot \\ (m'.sig = getInstance \land calls(m', m))$$

- there cannot be any earlier requests for an instance

$$\forall m'' \in msgs \cdot \\ (m'' < m' \Rightarrow m''.sig \neq getInstance)$$

- any subsequent request for an instance will return the same instance

$$\forall m''' \in msgs \cdot \\ (m' < m''' \land m'''.sig = getInstance \\ \Rightarrow returns(m') = returns(m'''))$$

Note that sequence diagrams do have a limitation here, in that they cannot be used to state explicitly the intent that only one instance is created, nor that the instance is retrieved from a field. Instead both of these must be and can be inferred from the dynamic conditions given above.

4.4 Factory Method

As discussed in Section 1, patterns when documented informally will inevitably contain ambiguities or even inaccuracies, so often, as with the Factory Method pattern, we must choose between alternatives.

Let us first introduce a predicate $isMakerFor(op, C)$, which is true if op starts an activation that creates and returns an object of class C. Formally,

$$isMakerFor(op, C) \equiv \exists m \in msgs \cdot (m.sig = op \\ \Rightarrow \exists m' \in msgs \cdot (isNew(m'.sig) \\ \land calls(m, m') \land toClass(m') = C \\ \land returns(m) = toLL(m').name))$$

Then Factory Method can be specified as follows.

Components

- $Creator, Product \in classes$
- $factoryMethod \in Creator.opers$

Static Conditions

- $factoryMethod.isAbstract$
- foreach creator subclass there is one product subclass

$$\forall C \in subs(Creator) \cdot \exists! P \in subs(Product)$$

- furthermore, denoting witness P by $f(C)$, then f is a total bijection.

Dynamic Conditions

- for every creator subclass, the factory method creates that unique product subclass:

$$\forall C \in subs(Creator) \cdot \\ isMakerFor(C..factoryMethod, f(C))$$

Now for the alternative formulations. First, in [5] Eden allows there to be several factory methods rather than just one as in the above. Second, one could argue for $\neg factoryMethod.isLeaf$ instead of $factoryMethod.isAbstract$. Third, the operation $AnOperation \in Creator.opers$ is not essential to the Factory pattern. But, if it is added to the Components section, the condition $calls(AnOperation, FactoryMethod)$ should also be added to the Dynamic Conditions.

5 Analysis of case study

We now report the findings of our case study where we formally specified all 23 patterns in the GoF book [8].

First, every specification is simpler in its structural features than, for example, those in [1]. Our notations match more closely the arrows of UML class diagrams. More importantly though, when only a class diagram was available, the behavioural features were expressed as static conditions. Now, they can be expressed more naturally using sequence diagrams. For example, the calls relation between operations was previously defined as a dependency relation between operations, but it is more naturally expressed in sequence diagrams. This allows us to choose the simpler option when one notion can be expressed in two different ways. The consistency assumption, itself specified with first-order predicates, also allows us to reduce redundancy by removing equivalent expressions. In Table 2 the column entitled *Simpler structural properties* shows where we have been able to make simplifications.

Second, as one would expect, sequence diagrams enable us to characterise dynamic properties more accurately and adequately. A class diagram can dictate that one method calls another, as discussed in Section 2, and this can be enough for some patterns but others require more information, such as the temporal ordering of messages, which must come from sequence diagrams. In Table 2 the column entitled *Improved behavioural properties* indicates that 5 patterns are specified without an obvious improvement, and 12 patterns show a slight improvement when the additional information contained in sequence diagrams is used, but for 6 patterns, the improvement is significant.

Table 2. Findings of the Case Study

Pattern	Simpler structural properties	Improved behavioral properties	Many alternatives	Specified adequately
Abstract Factory	✓	✓	✓	✓
Adaptor	✓	✓	×	✓
Bridge	✓	×	✓	×
Builder	✓	✓✓	✓	×
Chain of Respons.	✓	×	✓	✓
Command	✓	✓✓	✓	✓
Composite	✓	✓	✓	✓
Decorator	✓	✓	✓	×
Facade	×	✓	×	✓
Factory	✓	✓	✓	✓
Flyweight	×	×	×	×
Interpreter	✓	✓	×	✓
Iterator	✓	✓	×	✓
Mediator	✓	✓	×	✓
Memento	×	✓✓	×	✓
Observer	✓	✓✓	✓	×
Prototype	✓	✓	✓	✓
Proxy	✓	✓	✓	×
Singleton	×	✓✓	×	✓
State	✓	✓	×	✓
Strategy	✓	×	×	✓
Template	✓	×	×	✓
Visitor	✓	✓✓	×	✓

Third, some patterns have alternative non-equivalent specifications, as noted in [1]. Sometimes there is ambiguity even in the best documented patterns and clarification is needed to select between the alternatives. Sometimes, however, each alternative is a different specialisation of the pattern. These alternatives, each with their pros and cons, may form sub-patterns if they are significant enough in practice. An advantage of the method is that we can now document each of the subpatterns separately, exploring the alternatives without having to commit to any of them. The reader can then make an informed choice. In Table 2, the column entitled *Many alternatives* shows there are 11 patterns for which there were many equally valid alternatives.

Finally, as UML diagrams contain only some information about the system and at a high level of abstraction, one may find a specification based on these does not fully express all the properties required. Three examples of this now follow.

In the Builder pattern, the BuildPart operations in the Builder class must each build a different part of the Product, and the first creates the object of class Product. This cannot be accurately expressed. The rest of this pattern can be captured adequately, however, and better than in [1] as

the sequence diagram is constrained.

In the Composite pattern, the Composite class must propagate messages sent to it to each of its children, but without an object diagram, we cannot tell which lifelines must be the target of the messages. Naturally, this is also a problem with the Interpreter pattern, but we can at least dictate that the recursive calls are parameterised by the same Context object.

In the Flyweight pattern, since the Flyweight class has two different subclasses, one holding the intrinsic state and the other holding the extrinsic state, the missing parts of the state should be passed to operations on the former. This cannot be fully expressed, too, because such information cannot be included in design models of UML class and sequence diagrams.

In Table 2, the column entitled *Specified adequately* indicates whether the structural and behavioural features have been fully specified; in 6 out of 23 patterns it has not.

6 Conclusion

In this paper, we further advanced the approach proposed in [1]. The advantages of the approach as demonstrated in [1] and summarised in Section 2 apply here too but we omit the demonstrative examples to save space.

The main contribution of this paper is an investigation into the behavioural features of patterns. The case study shows that the method improves the accuracy and adequacy of formal specification of design patterns for almost all patterns in the GoF book, with the exception of the Flyweight pattern.

For future work, we believe that a generic pattern-based software design tool based on formal specifications could make a significant impact on the practical use of patterns in software development and to ensure the quality of software patterns and pattern languages. It will also be interesting to conduct further case studies of the method with more complicated patterns, such as patterns in distributed systems where dynamic behavioral features play a dominant role. We also plan to formally define when patterns can be composed into larger patterns like Model-View-Controller[8].

References

[1] I. Bayley and H. Zhu. Formalising design patterns in predicate logic. In *Proc. SEFM'07*, London, 2007.

[2] I. Bayley and H. Zhu. Specifying behavioural features of design patterns. Tech. Report DOC-TR-08-01, Dept. of Comp., Oxford Brookes Univ., Oxford, UK, 2008.

[3] S. Berczuk. Finding solutions through pattern languages. *IEEE Computer*, 27(12):75–76, Dec. 1995.

[4] P. Coad. Object-oriented patterns. *Communications of the ACM*, 35(9):152– 159, September 1992.

[5] A. H. Eden. Formal specification of object-oriented design. In *Int. Conf. on Multidisciplinary Design in Engineering, Montreal, Canada*, November 2001.

[6] A. H. Eden. A theory of object-oriented design. *Information Systems Frontiers*, 4(4):379–391, 2002.

[7] R. B. France, D.-K. Kim, S. Ghosh, and E. Song. A uml-based pattern specification technique. *IEEE Trans. Softw. Eng.*, 30(3):193–206, 2004.

[8] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns - Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[9] A. L. Guennec, G. Sunyé, and J.-M. Jézéquel. Precise modeling of design patterns. In *Proc. UML'02*, LNCS 1939, 482–496. Springer, 2000.

[10] S. R. Kodituwakku and P. Bertok. Pattern categories: a mathematical approach for organizing design patterns. In *Proc. CRPIT'02*, 63 – 73, Melbourne, Australia, June 2003. Australia Computer Society, Inc.

[11] K. Lano, J. C. Bicarregui, and S. Goldsack. Formalising design patterns. In *BCS-FACS Northern Formal Methods Workshop*, September 1996.

[12] A. Lauder and S. Kent. Precise visual specification of design patterns. In *Proc. ECOOP'98*, LNCS 1445, 114–134, Springer, 1998.

[13] J. K. H. Mak, C. S. T. Choy, and D. P. K. Lun. Precise modeling of design patterns in uml. In *Proc. ICSE'04*, 252–261, 2004.

[14] D. Mapelsden, J. Hosking, and J. Grundy. Design pattern modelling and instantiation using dpml. In *Proc. CRPIT '02*, 3–11. Australian Comp. Society, 2002.

[15] T. Mikkonen. Formalizing design patterns. In *Proc. ICSE'98*, 115–124. IEEE CS, April 1998.

[16] N. Nija Shi and R. Olsson. Reverse engineering of design patterns from java source code. In *Proc. ASE'06*, 123–134, September 2006.

[17] OMG. Unified modeling language: Superstructure, version 2.0, formal/05-07-04.

[18] PLAC'2007. The first international workshop on patterns languages: Addressing challenges. http://www.engr.sjsu.edu/ fayad/workshops/PLAC07, Accessed on 12 Sept. 2007 2007.

[19] T. Taibi. Formalising design patterns composition. *IEE Proc. on Software*, 153(3):126–153, June 2006.

[20] T. Taibi, D. Check, and L. Ngo. Formal specification of design patterns-a balanced approach. *Journal of Object Technology*, 2(4), July-August 2003.

[21] T. Winn and P. Calder. A pattern language for pattern language structure. In *Proc. CRPIT'02*, 45–58. Australia Computer Society, Inc., June 2003.

[22] U. Zdun and P. Avgeriou. Modelling architectural patterns using architectural primitives. In *Proc. OOPLSA'05*, 133–146, 2005.

[23] H. Zhu and L. Shan. Well-formedness, consistency and completeness of graphic models. In *Proc. UKSIM'06*, 47–53, April 2006.