Formal Specification of the Variants and Behavioural Features of Design Patterns

Ian Bayley and Hong Zhu

Department of Computing and Electronics, Oxford Brookes University, Oxford OX33 1HX, UK

Abstract

The formal specification of design patterns is widely recognized 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 a formal meta-modelling approach that uses first order predicate logic to specify design patterns. In particular, it specifies both structural and behavioural features of design patterns and systematically captures the variants in a well-structured format. The paper reports a case study involving the formal specification of all 23 patterns in the Gang of Four catalog. It demonstrates that the approach improves the accuracy of pattern specifications by covering variations and clarifying the ambiguous parts of informal descriptions.

Key words:
Design patterns, formal specification, predicate logic, graphic modeling, sequence diagrams, variants of patterns

1. Introduction

Design patterns are a technique for documenting solutions to recurring design problems and for sharing design expertise in an application-independent fashion (Coad, 1992; Berczuk, 1995; Gamma et al., 1995). They are commonly presented in Alexandrian form, in which design principles are first explained in informal English, and then clarified with illustrative diagrams and specific code examples (Gamma et al., 1995). This format is informative enough for humans to understand the design principle and to learn how to apply patterns to solve their own problems.

However, as Eden and Hirshfeld (2001) pointed out, the Alexandrian format is very informal and hence brings such ambiguity that it is often a matter of dispute whether an implementation conforms to a pattern or not. Furthermore, it is now widely recognised that poor presentation of patterns can lead to poor system quality (PLAC, 2007), and can actually impede maintenance and evolution, according to the empirical studies in (Khomh and Guéhéneuc, 2008), which suggests that patterns should be used with caution. Mathematical notations can help eliminate this ambiguity by clarifying the underlying notions. So it is no surprise that the past few years have seen much research into formal pattern specification but, as we argue in Section 2, the full potential of formal pattern specification has still not yet been realised.

Another weakness of the Alexandrian format is that it presents knowledge in an unstructured way. Each pattern is described separately, with relationships to other patterns merely indicated. But understanding individual patterns in isolation is not enough. They need to be cataloged

(Winn and Calder, 2003) and combined to solve real-world problems. Formal specification can not only remove ambiguity, as discussed above, but also lay a solid foundation for reasoning about their properties and their inter-relationships.

In our approach, we define the abstract syntax of UML, in which the software design models are represented, using the meta-notation Graphical Extension of BNF (GEBNF). From this GEBNF definition, we can systematically derive primitive predicates on these design models, and thus a First-Order predicate Logic (FOL) language, in which design patterns can be formally specified as predicates on designs, predicates that are satisifed if and only if the design conforms to the pattern. They are written in a scheme that explicitly demarcates the structural features, behavioural features and allowable variants.

Our main contributions here are:

- the meta-notation GEBNF
- the GEBNF definition of a non-trivial subset of UML with class and sequence diagrams
- the mechanism for deriving primitive predicates on models
- the formal scheme for specifying design patterns as complex predicates on models.

Other related contributions mentioned only briefly here, for reasons of space, are:

- the specification of all 23 patterns in the GoF book (we show just some of these)
- a prototype software tool called LAMBDES-DP for recognising patterns
- the formal proof and definition of various properties of patterns
- experiments on the 23 patterns that show the tool has a high precision and recall rate
- a definition of transformations on patterns such as lifting
- a definition, using these, of pattern composition as an operation on predicates

The remainder of the paper is organized as follows. Section 2 reviews related work and discusses the problems associated with them. Section 3 describes our formal meta-modeling framework, which is based on the abstract syntax definition of modeling language in GEBNF and on the induced first-order logic. Section 4 presents the scheme in which design patterns are specified and illustrates it with a number of examples. Section 5 reports a case study on the formal specification of the patterns in the GoF book (Gamma et al., 1995) and the main results of the experiments with the tool LAMBDES-DP. Section 6 concludes the paper with a discussion of the directions for future work.

2. Related Work and Associated Problems

Some recent research efforts have adapted existing modeling notations and formal specification techniques, while others have developed new languages specifically for the purpose. In this section, we divide the existing work into modelling and meta-modelling approaches. The first specifies design patterns as a model of software systems, i.e. as a set of features within its program code structure and/or run-time behaviors. The second approach, to which our work belongs, specifies design patterns as meta-models, i.e. as a set of features of the models themselves.

2.1. The Modeling Approach

Of the two approaches, this was the first to be explored, and it can even be seen within GoF itself, where each design pattern is illustrated with an OMT class diagram. But even when accompanied by a sequence diagram, a class diagram is not enough to capture all the information conveyed by a design pattern. So many researchers proposed extensions to object-oriented models. For example, Lauder and Kent (1998) proposed a three-layer modeling approach consisting of role models, type models and concrete class models. Lano et al. (1996) also focused at the code-level and treated patterns as transformations from flawed solutions to improved solutions. Guennec et al. (2000) extended the UML meta-model to incorporate collaboration occurrences and to use the Object Constraint Language (OCL) to constrain the collaborations. Zdun and Avgeriou (2005) identified architectural primitives that occur in patterns from the component-and-connector view. Mak et al. (2004) define the notion of collaborations by extending UML to action semantics.

Existing formal specification techniques have also been applied. Mikkonen (1998) formalizes the temporal behaviors of software in a temporal logic of actions. Taibi et al. (2003) formalize structural features as relations between program elements, specify post-conditions with predicate logic and describe the desired behavior with temporal logic. Taibi (2006) also investigated how pattern composition can be defined in this framework.

Dozens of tools have been developed to recognize patterns by analyzing source code. Examples include FUJABA (Niere et al., 2002), HEDGEHOG (Blewitt et al., 2005) and PINOT (Nija Shi and Olsson, 2006); see (Dong et al., 2007) for a recent survey and review. These tools often use an intermediate representation of program structure and behavior features. The most common of these representations include logic programs (Krämer and Prechelt, 1996; Huang et al., 2005), relational databases (Seemann and von Gudenberg, 1998) and the first-order logic predicates (Beyer et al., 2005). They can all be regarded as model-level pattern specifications. They are matched by the tool with design information extracted from the source code. But the level of abstraction is too low and the specifications are, of necessity, overwhelmed with details related to programming language-specific issues. This makes it difficult to achieve high precision and recall in pattern recognition (Dong et al., 2007). Even worse, some behavioral properties cannot be extracted from source code because they are non-determinable in general.

A further problem is that “formal specification languages were not molded to express implementation details”, as Eden et al. (1997) put it. They describe external characteristics precisely without specifying implementation details at all, but a design pattern is all about the implementation-oriented solution to a kind of problem. Graphic modeling languages like UML are not expressive enough either. GoF’s own descriptions of design patterns have to rely mainly on informal notes, using diagrams only as illustration, because in the absence of formalisation they are neither precise nor general enough, as is widely recognised. Finally, while tools like PINOT (Nija Shi and Olsson, 2006) are desirable, design-level tools are better still as they would minimise errors at the design stage, which is earlier. In which case, design patterns must be specified in terms of the structural and behavioural features of design models rather than source code. This naturally motivates the second approach, to which our work belongs.

2.2. The Meta-Modeling Approach

The *LePUS* approach, advanced by Eden (2001, 2002), is to include both modelling and meta-modelling facilities in the same language. To support modelling, LePUS has ground entities from object-oriented design such as classes, attributes and methods, connected by ground relations such as inherit, invoke and create. To support meta-modelling, LePUS has typed sets of these ground entities plus some rudiments, which reflect the common building blocks within GoF design patterns. Examples of these are called *clan*, *tribe*, *hierarchy* and *bijection*, and their semantics are all formally defined in predicate logic. So each well-formed LePUS diagram is equivalent to a formula $\varphi(x_1, x_2, \cdots, x_n)$ in first-order predicate logic, where the free variables $x_1, x_2, \cdots, x_n$ are the participants in the pattern and the relation φ between them represents the collaborations. Relationships between patterns can be described as logic relations (Eden and Hirshfeld, 2001). Recently, the visual notations and the underlying first-order logic language have been formally defined (Eden et al., 2007) and referred to as *LePUS3* and *Class-Z*, respectively. A tool has been implemented to extract information from Java programs and represent it in the form of LePUS3 diagrams. It can then decide and prove automatically whether or not the program satisfies a pattern specification written in LePUS3 (Gasparis et al., 2008b,a). However, LePUS's rudiments cover only the static structural features of design patterns and they ignore dynamic behavioral features.

Another well-known work in this category is the *Design Pattern Modeling Language* (*DPML*) of Maplesden et al. (2001, 2002). In this language, design pattern solutions are modeled as a collection of participants, representing structural features such as classes and methods, plus associated constraints and dimensions. Constraints either relate to a single object realising a participant, in which case they are written as natural language annotations within curly brackets, or two such objects, in which case, they are binary relations drawn as lines with arrowheads between the two participants. These are predefined and examples include *implements*, *extends*, *realizes*, *declared in*, *defined in* and *refers to*. Dimensions specify the set of objects playing a role and dimension keys can be used to specify that two or more participants have the same dimension ie the same number of instances of the roles. When the binary relations are applied to participants associated with dimensions, they can be mapped to a *total*, *regular*, *complete* or *incomplete* relation between the sets of objects. The facilities for modelling are kept more separate from those for meta-modelling than is the case with LePUS.

DPML also defines a set of visual notations for specifying the instance models of patterns. A tool called DPTool, reported in (Maplesden et al., 2002), supports both pattern specification, and the checking of pattern conformance, with respect to a UML model. DPML does have some shortcomings though that affect its expressivity and precision. For a start, the constraints on participants are informal and again, only structural features of design patterns can be specified. Also, some important issues are not addressed and thus remain open problems. These include how to reason about patterns, how to compose them, and how to specify variants without having to use a separate DPML diagram for each one.

LePUS and DPML both mixed modelling and meta-modelling together in the same language but the Role-Based Meta-modeling Language *RBML* proposed by France et al. (2004) took a strict meta-modeling approach. RBML extends UML for meta-modelling in a UML-like notation. A pattern is viewed as a meta-model so each instance of the pattern is a model in UML. The participants of a pattern and the relations between them are represented in graphic notation as roles. Further constraints are represented in the Object Constraint Language OCL, and the semantics of methods and attributes can be defined as OCL templates, instantiated for each pattern. An RBML meta-model can be translated into a UML meta-model in the form of a UML class diagram. We can also determine whether a UML model conforms to a design pattern specified by an RBML meta-model. The expressiveness of RBML has been demonstrated by defining the meta-models of class diagrams, sequence diagrams and state machine diagrams for such design patterns as Observer and Visitor (Kim, 2004).

Based on RBML specifications like these, Kim and Lu (2006) proposed a logic programming approach to identifying patterns in UML class diagrams. In this approach, a UML model is a set of Prolog facts and a pattern is a Prolog query. They illustrated the approach with the Visitor pattern, but without the details of how to translate UML designs into Prolog programs, nor how to translate RBML pattern specifications into Prolog queries. More recently, Kim and Shen (2007) developed an algorithm to check if a UML model conforms to a pattern specified in RBML. They have implemented it as a tool called RBMLCC, an add-on component of IBM Rational Rose. They also report a case study where 7 of the 23 GoF patterns are specified (Kim and Shen, 2008). However, the tool does not fully support RBML yet. In particular, OCL constraints and OCL templates are omitted.

Another work in this broad category, also with tool support, is that on the Pattern Description Language *PDL* (Albin-Amiot et al., 2001), though PDL diagrams are, strictly speaking, neither 'models' nor 'meta-models'. PDL is defined by an extended meta-model of UML and again, design patterns are specified in a graphical notation. A diagram that conforms to this extended meta-model is called an *abstract model* and it must be 'instantiated' into a *concrete model* as an instance of the pattern. However, this instantiation is not simply the conformance relation from model to meta-model seen in the OMG's four-layer meta-modeling architecture. For this reason, Elaasar et al. (2006) propose a Pattern Modeling Framework *PMF*. It extends the Meta Object Facility (MOF) (i.e. the M3 layer of OMG's four-layer meta-modeling architecture) to define a meta-modeling language called *Epattern* with which one can specify patterns in any MOF-compliant modeling language at the M2 layer and take advantages of existing modeling and meta-modeling tools.

In general, the graphic meta-modeling approach suffers from several drawbacks. First, meta-models are difficult to understand. This is partly solved in RBML, DPML and PDL by introducing new graphic notations for meta-models, but the semantics for these are complex and have not been formally defined. Secondly, graphic meta-models are ambiguous as all UML-based languages are, since UML is itself informal. LePUS has its semantics formally specified, but its notation is not widely used by designers and it does not handle behavioural features. Thirdly, graphic meta-models are not expressive enough to specify patterns accurately. RBML uses OCL to compensate but OCL is designed for modelling and has expressiveness issues when used in meta-modelling, for which there is no tool support anyway (France et al., 2004). Fourthly, graphic meta-models do not support formal reasoning about design patterns, such as that needed to compose two patterns or to decide whether one pattern is a special case of another. Finally, and most importantly, variants cannot be specified other than by using a different diagram for each one. As we shall see, the Adapter patterns has two variants: Object Adapter, with an association relation between the Adapter and Adaptee, and Class Adapter with an inheritance relation. These cannot be depicted both on one diagram.

2.3. Previous work

This paper presents our research work from the past few years. Some preliminary results have been reported at conferences. The meta-notation GEBNF was first proposed in (Zhu and Shan, 2006), but it has been simplified and formalised in this paper. Its use in design patterns was first proposed and outlined in (Bayley and Zhu, 2007), but only structural features were included. In (Bayley and Zhu, 2008b), behavioural features were specified as well, thereby improving the accuracy of pattern specification, and a case study of all 23 GoF patterns was also reported. In (Bayley and Zhu, 2008a), a family of composition operators on design patterns was defined with the help of transformations such as lifting. More recently, in (Zhu et al., 2009), the LAMBDES-DP tool was reported, together with the pattern recognition experiments that were performed on it. This paper further advances the approach by extending the scheme for pattern specification with alternative and optional conditions. We show that such variations can be specified formally and systematically in the first-order logic. This is difficult to achieve in graphic or meta-modelling languages, if not completely impossible.

3. Formal Meta-Modeling in GEBNF and First Order Logic

Each pattern is a subset of design models with certain structural and behavioral features. So formal specification of patterns is a meta-modeling problem. As in (Bayley and Zhu, 2007), our approach to meta-modeling begins by first defining the domain of all models in an abstract syntax for modeling languages written in the meta-notation GEBNF (Zhu and Shan, 2006), which stands for Graphic Extension of BNF. 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. Such a predicate is written in a first-order language induced from the abstract syntax definition of the graphic modeling language. So, a meta-model in our approach comprises an abstract syntax in GEBNF plus a first-order predicate.

In this section, we first formally define the meta-notation GEBNF (Zhu and Shan, 2006; Bayley and Zhu, 2007) and then use it to define the domain of models for UML class diagrams and sequence diagrams.

3.1. Graphical Extension of BNF

As is the case with the BNF definition of the syntax of a programming language, a GEBNF definition of the syntax of a modeling language is a set of syntax rules defining non-terminal symbols based on terminal symbols. The extensions that GEBNF brings to BNF are twofold. The first is *field naming*, which enables a set of function symbols to be deduced from a syntax definition to form a signature of a first-order predicate logic language. The second is the facility for *referential occurrences* of non-terminal symbols, so that two-dimensional structures like graphs can be defined.

3.1.1. GEBNF Meta-Notation

Definition 1. (GEBNF meta-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 syntax rules of the form

$$Y ::= L_1 : X_1, L_2 : X_2, \cdots, L_n : X_n,$$

where $Y \in N$, L_1, L_2 , $\cdots$, L_n are called *field names*, and X_1, X_2 , $\cdots$, X_n are the fields. Each field can be an expression, which is inductively defined as follows.

- For all $Y \in N \cup T$, Y is an expression.
- For all $Y \in N$, $\underline{Y}$ is an expression.
- If Y is an expression, Y^*, Y^+ and $[Y]$ are expressions.
- If $Y_1, Y_2, \cdots, Y_n$ are expressions, $Y_1 \mid Y_2 \mid \cdots \mid Y_n$ is an expression. □

Table 1: Meanings of the GEBNF Notation

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

The meaning of the meta-notation is explained in Table 1. Each terminal and non-terminal symbol denotes a type of entities. Terminal symbols denote the basic atomic entities like $String$, the set of strings. Non-terminal symbols denote the constructs of the modelling. And finally, the elements in the set of entities denoted by the root symbol are the models of the language.

If a non-terminal symbol is defined as $Y ::= L_1 : X_1, L_2 : X_2, \cdots, L_n : X_n$, then Y denotes a set of entities that are n-tuples with elements in the sets denoted by X_1, X_2, $\cdots$, X_n, respectively. In other words, each entity of type Y is constructed from n elements of type $X_1, X_2, \cdots, X_n$. The k'th element in the tuple can be accessed through the field name L_k, for every $1 \leq k \leq n$, and we write $a.L_k$ for the k'th element of a, if a is an entity of Y.

As an example, consider the following definition of directed graphs in GEBNF.

$$\begin{aligned} &Graph ::= nodes : Node^+, edges : Edge^* \\ &Node ::= name : String, weight : [Real] \\ &Edge ::= from : \underline{Node}, to : \underline{Node}, weight : Real \end{aligned}$$

where $Graph$ is the root symbol, $Graph$, $Node$ and $Edge$ are non-terminal symbols, and $String$ and $Real$ are terminal symbols.

This definition consists of three syntax rules, one on each line. It states that a graph consists of a non-empty set of nodes and a set of edges. Each node has a name, which is a string of characters, and may have an optional weight, which is a real number. Each edge is from one node to another, and has a weight, which is a real number.

If a symbol $X \in T \cup N$ occurs on the right-hand side of the definition of non-terminal symbol Y, we say that X is *directly reachable* from Y through a field name. For example, $Node$ and $Edge$ are directly reachable from $Graph$. We define the *reachable* relation as the transitive closure of the directly reachable relation. If there is a non-terminal symbol that is not reachable from the root symbol R, then its entities do not play any role in the construction of any model. In this case, we say that the syntax of the modelling language is not well-defined. We also say this when a non-terminal symbol is used but not defined. More formally, we have:

Definition 2. (Well-defined syntax)

An abstract syntax definition $\langle R, N, T, S \rangle$ in GEBNF is *well-defined* if it satisfies the following two conditions.

1. *Completeness*. For each non-terminal symbol $X \in N$, there is one and only one syntax rule $s \in S$ that defines X, i.e. for which X is the left-hand-side.
2. *Reachability*. For each non-terminal symbol $X \in N$, X is reachable from the root R of the syntax definition. □

Obviously, the syntax of directed graphs given above is well-defined.

3.1.2. Induced First-Order Language

Consider the syntax definition of directed graphs given above. The first syntax rule actually introduces two functions *nodes* and *edges*, which maps from a graph to its non-empty set of nodes and the set of edges, respectively. That is, if g is a graph, then $g.nodes$ is the set of nodes in g. In fact, every field $f : X$ in the definition of a symbol Y introduces a function $f : Y \to X$. Function application is written $a.f$ for function f and argument a of type Y.

In general, given a well-defined syntax, a set of function symbols and their types can be derived as follows.

Definition 3. (Induced functions)

A syntax rule "$A ::= \cdots, f : B, \cdots$" introduces a function symbol f whose domain is of type A and range is of type $[\![B]\!]$, where

- $[\![B]\!] = C$, when $B = C$ for symbol $C \in T \cup N$, so f is a total function from entities of type A to entities of type C;
- $[\![B]\!] = C$, when $B = \underline{C}$ for non-terminal symbol $C \in N$, so f is a total function from entities of type A to entities of type C, as above;
- $[\![B]\!] = \mathbb{P}([\![C]\!])$, when $B = C^*$, so f is a total function from entities of type A to the sets of entities of type $[\![C]\!]$.
- $[\![B]\!] = \mathbb{P}([\![C]\!]) - \emptyset$, when $B = C^+$, so f is a total function from entities of type A to the non-empty sets of entities of type $[\![C]\!]$;
- $[\![B]\!] = [\![C]\!] \cup \{\bot\}$, when $B = [C]$, where $\bot$ means *undefined*, so f is a partial function from entities of type A to elements of $[\![C]\!]$.
- $[\![B]\!] = \bigcup_{i=1}^{n}([\![C_i]\!])$, when $B = C_1|C_2|\cdots|C_n$, so f is a function from entities of type A to the disjoint union of the sets $C_1, C_2, \cdots, C_n$; in other words, for all $x \in A$, $x.f$ is in one of the types $[\![C_1]\!], [\![C_2]\!], \cdots, [\![C_n]\!]$. □

For example, the function *weight* introduced by the second syntax rule of directed graphs is a *partial* function from nodes to real numbers, because the clause *weight* : *Real* is optional. Therefore, a node n may be associated with no weight. In such a case, $n.weight$ is undefined and we write $n.weight = \bot$.

An occurrence of a non-terminal symbol X in the form of $\underline{X}$ on the right-hand-side of a syntax rule is called a *referential occurrence*, where the underline is called the *reference modifier*; otherwise, it is called a *creative occurrence*. It is worth noting that although a referential occurrence of a non-terminal symbol in a syntax rule introduces a function that is of the same range type as the creative occurrence of the symbol, the function has different properties, and thus the structure of the model is different. For example, if the syntax definition of *Edge* is replaced by the following rule, i.e. when the reference modifier on *Node* is removed from the original rule,

$$Edge ::= from : Node, to : Node, weight : Real,$$

each edge will introduce two new nodes, i.e. for all edges $e \neq e' \in Edges$, we have that $e.from \neq e'.from$. Moreover, for all edges e, we have that the node $e.from$ must be different from the node $e.to$, i.e. $e.from \neq e.to$. In contrast, the original definition allows $e.from = e.to$, $e.from = e'.from$ and $e.to = e'.to$ to be true for some edges e and e'.

In general, the function symbol induced from a field that contains a creative occurrences of a non-terminal symbol represents an *injective* function. Moreover, any two such injective functions of the same range type must have *disjoint images*.

Given a well-defined GEBNF syntax $\langle R, N, T, S \rangle$ of modeling language $\mathcal{L}$, let F be the set of function symbols $f : X \rightarrow Y$ derived from the syntax rules in S as defined in Definition 3.

From the set F of function symbols deduced from GEBNF syntax, a first-order language can be defined as usual using variables, relations and operators on sets and relations and operators on basic data types of terminal symbols and equality and logic connectives *or* $\vee$, *and* $\wedge$, *not* $\neg$, *implication* $\rightarrow$ and *equivalent* $\equiv$, and quantifiers *for all* $\forall$ and *exists* $\exists$.

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 may also write the application of function f to argument x with the more conventional prefix notation $f(x)$.

For example, the set of nodes in a graph g that have no weight associated with them can be formally defined using the functions introduced in the syntax definition as follows.

$$UnweightedNodes(g) = \{n | n \in g.nodes \wedge n.weight = \bot\}$$

3.1.3. Modeling and Meta-Modeling

Given an GEBNF definition of the abstract syntax of a modeling language, we now define what is a syntactically valid model.

Definition 4. (Well-formed model)

A well-formed model m in the language $\mathcal{L}$, written $m \in \mathcal{L}$, is a mathematical structure $m = \langle \mathbf{E}, \mathbf{F} \rangle$ that consists of a collection $\mathbf{E}$ of sets E_x ($x \in T \cup N$), and a collection $\mathbf{F}$ of functions φ_f ($f \in F$) such that for every $f \in F$, φ_f has the corresponding domain and range on $\mathbf{E}$ as f on $T \cup N$ defined in Definition 3 and φ_f also satisfies the corresponding restrictions on injectiveness and disjointness on images. □

Let $f(x_1, x_2, \cdots, x_n)$ be a formula in the first-order language that contains free variables $x_1, x_2, \cdots, x_n$. Let $m = \langle \mathbf{E}, \mathbf{F} \rangle$ be a valid model, and α be an assignment of the free variables in f to the elements of m. The formula $f(x_1, x_2, \cdots, x_n)$ can be evaluated by interpreting function symbols f in F by the corresponding functions φ_f in the model $m = \langle \mathbf{E}, \mathbf{F} \rangle$. We write $Eva_\alpha(f, m)$ to denote the value obtained when f is evaluated on m in the context of α. If this is true for a ground predicate f, meaning a truth-valued formula without free variables, we say that the m *satisfies* f and write $m \models f$. We write $p \vdash q$ if we can deduce formula q from formula p in the first-order logic. By the semantic consistency of first order languages, we have the following proposition.

Proposition 1.

Let p and q be ground predicates on models in a well-defined modeling language $\mathcal{L}$ defined in GEBNF. If $p \vdash q$, then for all models $m \in \mathcal{L}$, $m \models p$ implies $m \models q$. □

Meta-modeling defines a subset of models in a modeling language such that each model of the subset has a specific property. So if the abstract syntax of a modeling language is defined in

GEBNF, meta-modeling can be performed by defining a predicate p such that $\{m \mid m \models p\}$ is the required subset of models.

For example, consider the directed graphs defined above. The set of strongly connected graphs can be defined as the set of models that satisfy the following predicate.

$$\forall x, y \in nodes \cdot (x\ reaches\ y),$$

where predicate $(x\ reaches\ y)$ is defined as follows.

$$(x\ reaches\ y) \Leftrightarrow$$
$$\exists e \in edges \cdot (x = e.from \wedge y = e.to) \vee \exists z \in nodes \cdot ((x\ reaches\ z) \wedge (z\ reaches\ y))$$

The set of acyclic graphs can be defined as the set of models that satisfy the following predicate.

$$\forall x, y \in nodes \cdot ((x\ reaches\ y) \Rightarrow x \neq y)$$

The set of connected graphs can be defined as follows.

$$\forall x \neq y \in nodes \cdot ((x\ reaches\ y) \vee (y\ reaches\ x)).$$

Finally, a tree can be defined as satisfying the following condition.

$$\exists x \in nodes \cdot (\forall y \in nodes \cdot (x\ reaches\ y)) \wedge \forall e, e' \in edges \cdot (e.to = e'.to \Rightarrow e = e')$$

In the same way, we will define design patterns by first defining the abstract syntax of UML class diagrams and sequence diagrams and then specifying the predicates that their instances, i.e. models, must satisfy.

3.2. Abstract Syntax of UML

This subsection gives a definition of the abstract syntax of a simplified UML modeling language.

3.2.1. Class Diagrams

The GEBNF definition of UML class diagrams is obtained from (OMG, 2004) by removing those attributes not required to describe patterns, and by flattening the hierarchy 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 *ag*gregate) relations between them. A class has a name, attributes, and operations.

$$\begin{array}{rcl} ClassDiagram & ::= & classes : Class^{+}, assocs : Rel^{*}, inherits : Rel^{*}, compag : Rel^{*} \\ Class & ::= & name : String, [attrs : Property^{*}], [opers : Operation^{*}] \end{array}$$

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 UML 2.0 (OMG, 2004).

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

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.

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

3.2.2. 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.

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

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.

$$\begin{array}{rcl} Activation & ::= & start, finish : Event, others : Event^{*} \\ Message & ::= & send, receive : \underline{Event}, sig : \underline{Operation} \\ Event & ::= & actor : \underline{Activation} \end{array}$$

3.2.3. Predicates on UML Diagrams

When there is just one class diagram or one sequence diagram, functions on each of these are written without their arguments, as *classes*, *lifelines* etc. Here now 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 and $C..op$ denotes 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.

$$\begin{array}{rcl} calls(op, op') & \equiv & \exists m, m' \in msgs \cdot (m.sig = op \land m'.sig = op' \land trigs(m, m')) \\ calls(C, C') & \equiv & \exists m \in msgs \cdot (fromClass(m) = C \land toClass(m) = C') \\ callsHook(op, op') & \equiv & \exists C \in subs(C') \cdot calls(op, C.op') \end{array}$$

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, then 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 (Zhu and Shan, 2006). 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.

1. every message to an activation must be for an operation of a concrete class:

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

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

$$\begin{aligned}\forall m \in msgs \cdot (m.sig.isStatic \Rightarrow toLL(m).isStatic \wedge \\ \neg m.sig.isStatic \Rightarrow \neg toLL(m).isStatic)\end{aligned}$$

3. 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 (Gamma et al., 1995), the Builder pattern breaks the final constraint with the Product class.

4. Pattern Specification Scheme

In (Bayley and Zhu, 2007, 2008b), an informal scheme was advanced for presenting pattern specifications in a readable manner. In this section, we further develop the scheme by introducing new constructs to specify variants of patterns. This is illustrated by examples. All 23 patterns in the GoF book (Gamma et al., 1995) have been specified in this scheme. The case study will be reported in the next section.

4.1. Overall Structure

For each pattern, the formal specification consists of an identifier for the name of the pattern and three parts. The first part, entitled *Components*, declares a set of variables, which are existentially quantified over the scope of all predicates in the pattern specification. In this way, it sets the context 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 behavior of the system, using information in

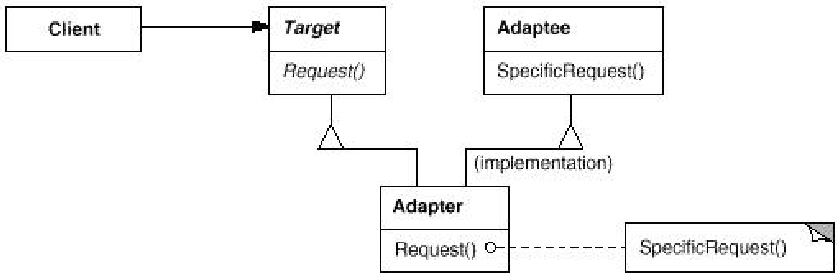

Figure 1: Class Diagram of Class Adapter Pattern

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 Class Adapter pattern, to illustrate the overall structure of pattern specification. The class diagram is shown in Figure 1. There are four participants, *Target*, *Client*, *Adapter* and *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

1. $Target, Adapter, Adaptee \in classes$
2. $requests \in Target.opers$
3. $specreqs \in Adaptee.opers$

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,

$$CDR(C) \equiv \forall m \in msgs \cdot (toClass(m) \in subs(C) \\ \Rightarrow m.sig \in toClass(m).opers \wedge \exists o \in C.opers \cdot (toClass(m).opers = m.sig))$$

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

STATIC CONDITIONS

1. $Adapter \multimap\!\!\triangleright Target$
2. $Adapter \multimap\!\!\triangleright Adaptee$
3. $CDR(Target)$
4. $requests \subseteq Target.opers$
5. $specreqs \subseteq Adaptee.opers$

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

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

In general, the overall structure of a pattern specification is defined by the following BNF syntax, where the terminal symbols are in bold font and non-terminal symbols are in italic font.

PatternSpec	::=	**Pattern Name :** *Identifier* **;**
		ComponentDeclaration
		StaticCondition
		[*DynamicCondition*]
ComponentDeclaration	::=	**Components :** *CompDecls* **.**
CompDecls	::=	([*Number*|*Label*] *Variable* **:** *Type*) [**;** *CompDecls*]
StaticCondition	::=	**Static Condition :** *Conditions* **.**
DynamicCondition	::=	**Dynamic Condition :** *Conditions* **.**
Conditions	::=	[*Explanation*] [*Number*|*Label*] **:** *Condition* [**;** *Conditions*]
Condition	::=	*Expression* | *AntecedentConsequent*
		| *Alternatives* | *Options* | *Dependent*

where *Explanation* is a string of characters, and *Expression* is a predicate on the domain of UML class diagrams and sequence diagrams. The other forms of conditions will be discussed later.

Let a pattern P be specified in the above form as follows.

Pattern Name : P
Components:
  $var_1 : Type_1;\ \ var_2 : Type_2; \cdots\ \ var_n : Type_n.$
Static Condition:
  $LabelS_1 : Ps_1;\ \ LabelS_2 : Ps_2; \cdots;\ \ LabelS_m : Ps_m.$
Dynamic Condition:
  $LabelD_1 : Pd_1;\ \ LabelD_2 : Pd_2; \cdots;\ \ LabelD_k : Pd_k.$

The semantics of this is the predicate

$$\exists var_1 : Type_1 \exists var_2 : Type_2 \cdots \exists var_n : Type_n \cdot (Ps \wedge Pd),$$

where $Ps = Ps_1 \wedge Ps_2 \wedge \cdots \wedge Ps_m$ and $Pd = Pd_1 \wedge Pd_2 \wedge \cdots \wedge Pd_k$.

Note that Adapter is typical of the structural patterns in the GoF catalog, in that 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 in (Bayley and Zhu, 2007), where only the class diagram is used. In fact, this is true for almost all patterns in the GoF catalog (Gamma et al., 1995). See Section 5 for more details.

Note too that the specification given above is for Class Adapter, which is a variant of the Adapter pattern. There is another variant called Object Adapter. The specification of both Class Adapter and Object Adapter as variants of the more general Adapter Pattern will be given in Subsection 4.3. Now, we first discuss how more complicated behavior can be specified in a more readable format.

4.2. Specifying Complicated Behaviors

Command is typical of the behavioral 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.

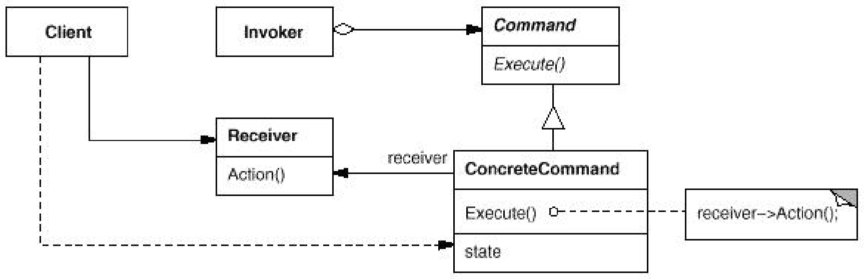

Figure 2: Command pattern class diagram

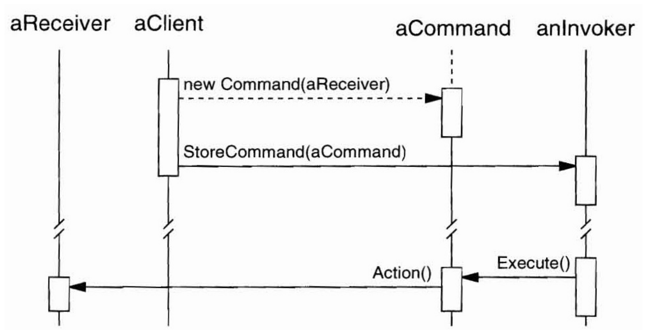

Figure 3: Command pattern seq diagram

Components

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

Static Conditions

1. $Invoker \diamond\!\!\longrightarrow Command$
2. $ConcreteCommand \longrightarrow Receiver$
3. $ConcreteCommand \longrightarrow\!\!\triangleright Command$
4. $execute.isAbstract$
5. $\neg isAbstract(ConcreteCommand)$

In the GoF catalog, 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 *Antecedent* 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:

1. when a command is executed

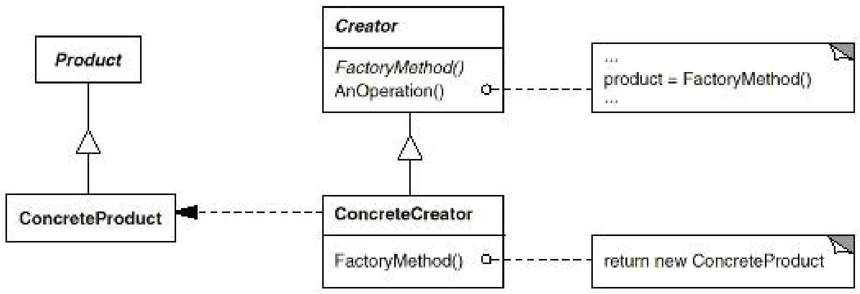

Figure 4: Factory method class diagram

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

CONSEQUENT:

1. the invoker is responsible,
$$fromLL(me).class = Invoker$$
2. the receiver will perform an action at once
$$\exists ma \in msgs \cdot (calls(me, ma) \wedge ma.sig = action)$$
3. the command to be executed is created
$$\exists mn \in msgs \cdot isNew(mn.sig) \wedge toLL(mn) = toLL(me)$$
4. the command is stored in the invoker
$$\exists ms \in msgs \cdot (ms.sig = storeCommand \wedge fromAct(ms) = fromAct(mn))$$
5. the command was created with the receiver before the command was stored before it was executed
$(mn < ms) \wedge (ms < me) \wedge hasParam(mn, toLL(ma).name) \wedge hasParam(ms, toLL(mn).name)$

This captures the dynamic information that would have been missed had we restricted our attention to the static properties considered by (Bayley and Zhu, 2007). This is particularly important for patterns where the static properties are trivial, such as the single-class Singleton pattern.

In general, an antecedent-consequent condition has the syntax defined by the following BNF rules.

$$AntecedentConsequent \quad ::= \quad \textbf{Antecedent :}\ conditions\ \textbf{; Consequent :}\ Conditions;$$

If a condition P is split into an antecedent P_A and a consequent P_C, the condition P is equivalent to $P_A \Rightarrow P_C$.

4.3. Specifying Variants

As discussed in Section 1, patterns that are documented informally will inevitably contain ambiguities or even inaccuracies so often, as with the Factory Method pattern, we must choose between alternatives and cover a number of variants so that patterns can be specified as structured knowledge.

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,

$$\begin{aligned}&isMakerFor(op, C) \equiv \exists m \in msgs \cdot (m.sig = op\\&\quad\Rightarrow \exists m' \in msgs \cdot (isNew(m'.sig) \wedge calls(m, m') \wedge toClass(m') = C\\&\qquad \wedge returns(m) = toLL(m').name))\end{aligned}$$

Then Factory Method can be specified as follows without covering variants.

PATTERN NAME : *Factory Method (without variants)*

COMPONENTS

1. $Creator, Product \in classes$
2. $factoryMethod \in Creator.opers$

STATIC CONDITIONS

1. $factoryMethod.isAbstract$
2. foreach creator subclass there is one product subclass
$$\forall C \in subs(Creator) \cdot \exists! P \in subs(Product)$$
3. furthermore, denoting witness P by $f(C)$, then f is a total bijection.

DYNAMIC CONDITIONS

1. 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, Eden (2001) allows there to be several factory methods rather than just one as above. Thus, an alternative to Component Declaration 2 is

$$factoryMethods \subseteq Creator.opers.$$

Second, for Static Condition 1, 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.

To enable alternatives and variations to be systematically specified, we introduce keywords '*Optional*', '*Alternatives*' and '*Depends on*', and '*In case of*' in the structure. For example, the specification of Factory Method pattern thus becomes the following.

PATTERN NAME : *Factory Method*

COMPONENTS

1. $Creator, Product \in classes$
2. ALTERNATIVES:
   (a) Single factory method: $factoryMethod \in Creator.opers$
   (b) Multiple factory methods: $factoryMethods \subseteq Creator.opers$
3. OPTIONAL: $AnOperation \in Creator.opers$

STATIC CONDITIONS

1. DEPENDS ON ALTERNATIVES OF Components Declaration 2:
   (a) IN CASE OF Single factory method, ALTERNATIVES:
      i. Stronger condition: $factoryMethod.isAbstract$
      ii. Weaker condition: $\neg factoryMethod.isLeaf$
   (b) IN CASE OF Multiple factory methods, ALTERNATIVES:
      i. A: $\forall fm \in factoryMethods \cdot (fm.isAbstract)$
      ii. B: $\forall fm \in factoryMethods \cdot (\neg fm.isLeaf)$
      iii. C: $\forall fm \in factoryMethods \cdot (\neg fm.isLeaf \vee fm.isAbstract)$
2. for each creator subclass there is one product subclass
$$\forall C \in subs(Creator) \cdot \exists! P \in subs(Product)$$
3. furthermore, denoting witness P by $f(C)$, then f is a total bijection.

DYNAMIC CONDITIONS

1. for every creator subclass, the factory method creates that unique product subclass:
$$\forall C \in subs(Creator) \cdot isMakerFor(C..factoryMethod, f(C))$$
2. DEPENDS ON THE OPTION OF Component 3:
   (a) IN CASE WHERE THE OPTION IS TRUE:
      i. DEPENDS ON ALTERNATIVES OF Component 2 :
         A. IN CASE OF Single factory method:
$$calls(AnOperation, FactoryMethod)$$
         B. IN CASE OF Multiple factory methods:
$$\exists fm \in FactoryMethods \cdot calls(AnOperation, fm)$$

In general, the syntax of Alternatives, Optional and Dependent conditions is defined by the following BNF formulas.

$$\begin{array}{rcl}
Alternatives & ::= & \textbf{Alternatives : } AlterConds \\
AlterConds & ::= & Label \textbf{ : } Condition \text{ [} \textbf{;} \text{ } AlterConds\text{]} \\
Options & ::= & \textbf{Optional : } condition \\
Dependent & ::= & \textbf{Depends on } (\textbf{alternatives} \mid \textbf{option}) \; Label \;\; CaseConds \\
CaseConds & ::= & \textbf{In case of } Label \textbf{ : } Condition \text{ [;} CaseConds\text{]} \mid \\
 & & \textbf{In case where the option } (\textbf{is true} \mid \textbf{is false}) \textbf{ : } Condition
\end{array}$$

The semantics of these constructs are as follows. A condition that has the *Alternative* structure in the following form is equivalent to $P_A \vee P_B \vee \cdots \vee P_C$.

$$\begin{array}{l} Label_P: \ \textsc{Alternatives}: \\ \quad Label_A : P_A; \\ \quad Label_B : P_B; \\ \quad \ldots \\ \quad Label_C : P_C. \end{array}$$

A condition that has the *Depends on alternatives* structure in the following form is equivalent to the condition $(P_A \Rightarrow Q_A) \wedge (P_B \Rightarrow Q_B) \wedge \cdots \wedge (P_C \Rightarrow Q_C)$.

$$\begin{array}{l} \textsc{Depends on alternatives of } Label_P: \\ \quad \textsc{In case of } Label_A: \ Q_A; \\ \quad \textsc{In case of } Label_B: \ Q_B; \\ \quad \ldots \\ \quad \textsc{In case of } Label_C: \ Q_C. \end{array}$$

A condition R in the *Optional* structure in the following form can be omitted.

$$Label_R : \textsc{Optional} : P_R;$$

However, it assigns a name R to the predicate P_R so that the following condition S in the *Depends on option* structure is equivalent to $(P_R \Rightarrow S_{true}) \wedge (\neg P_R \Rightarrow S_{false})$.

$$\begin{array}{l} Label_S: \ \textsc{Depends on option } Label_R: \\ \quad \textsc{In case of where the option is true}: \ S_{true}; \\ \quad \textsc{In case of where the option is false}: \ S_{false}. \end{array}$$

By specifying the variants of a pattern, we can organise design knowledge in a much more structured way. Let's now back to the variants of Adapter pattern and see how both Class Adapter and Object Adapter can be specified within one framework.

As shown in Figure 5, the Object Adapter pattern has the static condition $Adapter \longrightarrow Adaptee$ instead of $Adapter \multimap\!\!\!\triangleright\ Adaptee$. And, we must also capture the condition that it is only the *Adapter* class that can send a message to the *Adaptee*.

$$\forall m \in msgs \cdot (toClass(m) = Adaptee \Rightarrow fromClass(m) = Adapter)$$

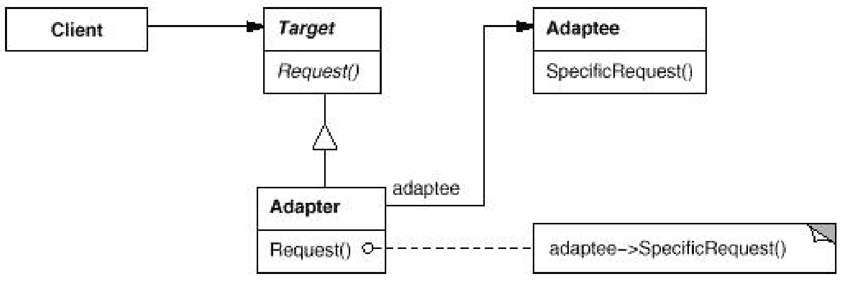

Figure 5: Class Diagram of Object Adapter Pattern

So an alternatives clause is introduced in the static condition, while the above dynamic condition is added as a dependent condition only applicable to the Object Adapter. Thus, we have the following.

PATTERN NAME : *Adapter*
COMPONENTS

1. $Target, Adapter, Adaptee \in classes$
2. $requests \subseteq Target.opers$
3. $specreqs \subseteq Adaptee.opers$

STATIC CONDITIONS

1. $Adapter \multimap\!\!\triangleright Target$
2. ALTERNATIVES:
   (a) Object adapter: $Adapter \longrightarrow Adaptee$;
   (b) Class adapter: $Adapter \multimap\!\!\triangleright Adaptee$;
3. $CDR(Target)$

DYNAMIC CONDITIONS

1. A request is delegated to a specific object

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

2. DEPENDS ON ALTERNATIVES OF Static Condition 2:
   (a) IN CASE OF Object adapter, only the *Adapter* class can send a message to the *Adaptee*,

$$\forall m \in msgs \cdot (toClass(m) = Adaptee \Rightarrow fromClass(m) = Adapter)$$

5. Case Study of the GoF Patterns

We have formally specified all 23 design patterns in the GoF catalog (Gamma et al., 1995). In this section we discuss the findings of the case study.

To begin with, let us first discuss the inevitable ambiguity surrounding the informal descriptions in (Gamma et al., 1995) and how the formal specifications were formed out of them. The generic class and sequence diagrams were the main source of ambiguity. As graphic notations have limited expressiveness, the generic features of design patterns have to be conveyed through illustrative uses of the notation. The most extreme example of this is the *Facade* pattern, whose class diagram given in the GoF book is not even well-formed and cannot be taken at face-value in terms of either the number of classes or their inter-connections as shown in Figure 6.

It was reasonably straightforward in this case to tell what was meant by the diagram. In many other cases, more than one interpretation was possible, as noted in (Bayley and Zhu, 2007), and the ability to specify these variants systematically is the main contribution of this paper. We now consider the expressiveness of our notation for this.

5.1. Expressiveness

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 valid specialization 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 proposed in this paper is that we can now document the sub-patterns systematically, exploring the alternatives

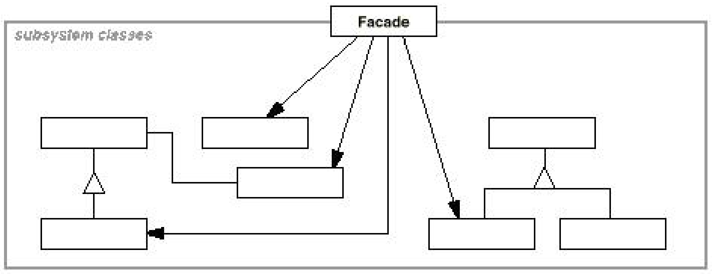

Figure 6: Class Diagram of Facade Pattern

without having to commit to any of them. The reader can then make an informed choice. We now discuss what sort of variants can exist. For a start, in general, we declare component variables for the classes named on the class diagram, but there are exceptions.

First, the only classes declared are those that need to be referred to in the conditions. For example, in the GoF book, the class diagram for *State* pattern contains two subclasses of the *State* class named *ConcreteStateA* and *ConcreteStateB*, respectively. They are used to illustrate the existence of an arbitrary number of concrete state classes. There are two possible ways to specify such a set of concrete state classes. The first is to explicitly declare a component $ConcreteStates \subseteq classes$ and to have a static condition that $\forall x \in ConcreteStates \cdot (x \multimap\!\!\triangleright State)$. The second is to call the set $subs(State)$ in all the conditions as we did in (Bayley and Zhu, 2008b). There is a subtle difference between these two specifications in that $ConcreteStates$ could be a proper subset of $subs(State)$. It is unclear from the description in GoF whether the pattern allows the *State* class to have a subclass that is not a concrete state. If it does not, we need to have a static condition $\forall x \in classes \cdot (x \multimap\!\!\triangleright State \Rightarrow x \in ConcreteStates)$. We can now use the optional condition facility to cover both cases and leave the choice to the reader.

Similarly, a class is often shown in a class diagram to have only one subclass, and as we saw with *Factory Method*, this can be generalized to more than one class. There are some places were this would complicate the conditions though, such as in *Command* and *Proxy*, and there we leave the conditions specialized to one class. We make a similar generalization where only one subclass is shown to meet a condition, as with the *Leaf* subclass of *Composite* pattern, since the accompanying Motivation section shows several. We could have done this for the *Composite* class too, but decided not to, for simplicity.

Some classes can be omitted. For example, the *ObjectStructure* class of *Visitor* pattern is only there to communicate through untranslatable English that the elements are to be found in a collection of some sort. Moreover, the ubiquitous class marked *Client* is omitted from nearly all patterns, because its purpose is to indicate the class on which the rest of the program depends. Usually, this is a class at the root of an inheritance and there the *CDR* predicate is used. Where the relationship is not a simple dependency, the *Client* is included in the variables. This occurs in both the *Prototype* pattern and the *Interpreter* pattern, where there is an extra dependency to the class Context.

A component can also be introduced for unnamed class nodes in the class diagram. This is seen in the *Facade* pattern where variable *behind* represents the set of classes behind the facade class and *rest* represents the rest of the system. Each class diagram implies that each class must be different, but since these conditions do not seem to be absolutely necessary, they are omitted in our previous work (Bayley and Zhu, 2008b). These conditions can now be specified as optional conditions too.

Even the properties of a class can be ambiguous. In general, where a class is indicated as abstract, we read this as saying that the class could be abstract but does not need to be. Clearly, if an operation is shown as abstract then the constraints on class diagrams imply that the class must be abstract too. However, it also make sense to use the weaker alternative that the method can be overridden in a subclass, as with the *Factory Method* pattern. Again, such a condition can be specified explicitly using an optional condition or alternative conditions.

The operations listed for a class may or may not be the only operations in the class. If they are all abstract then the class can be considered as an interface, as with the *Strategy* and *Abstract-Factory* classes in the patterns of those names, but whether to include this in the conditions is an arbitrary choice, and thus another appropriate use of the optional condition and alternative conditions. In the case of *Template Method*, the abstract requirement could be replaced by non-leaf for a subtly different requirement.

Where one operation is listed for a class, it can be taken as representative of several operations. For example, *Composite* could be defined to have several operations of the sort named operation, and so too could *Decorator*. For the *Decorator* class, it is implied that subclasses have more attributes or more operations or both, and this could be enforced but, for simplicity, it is not. The implication is that the subclasses add behavior. This could also be true of the subclasses of *Abs* in the *Bridge* pattern but there it is not implied.

For the *Abstract Factory* pattern, the diagram implies that the products and creation operations are in bijection, i.e. each product is created by one operation (where an operation is identified both by its name and class) and vice versa, but this need not necessarily be true. A similar bijection is placed in the conditions for the *Visitor* pattern. These conditions are, therefore, optional.

In conclusion, the case study has demonstrated that the extended scheme is expressive enough for specifying variants of patterns.

5.2. Readability

In previous work (Bayley and Zhu, 2008b), we found that by specifying the behavioral features we could make the structural features much simpler than before (Bayley and Zhu, 2007). Our notations then match more closely the arrows of UML class diagrams. More importantly though, when only a class diagram was available, the behavioral 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 as a feature of 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.

As one would expect, sequence diagrams enable us to characterize 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.

The introduction of facilities to specify variants inevitably increases the complexity of pattern specifications. However, as demonstrated in Section 4, the specification remains readable, and more so than the equivalent mathematical expression.

5.3. Adequacy

Finally, as UML diagrams contain only some information about the system and at a high level of abstraction, one may find that a specification based on them 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 without in (Bayley and Zhu, 2007) because now 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 of the 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 Observer pattern, we have the same problem as we do with the Composite pattern. This problem can be resolved by using frames in the sequence diagram. Frames are omitted from the case study though for simplicity as only a few patterns need them.

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, either, because such information cannot be included in design models of UML class and sequence diagrams.

In the Iterator pattern, the operations First, Next, IsDone and CurrentItem are mentioned but their semantics cannot be captured in UML. Assignments cannot be captured in UML either so we cannot specify in the case of the Singleton class, for example, that the instance is set to null at creation time. A solution to this problem has been advanced in RBML by France et al. (2004), which is to use OCL template to specify the semantics of operations. This can be easily incorporated into our formal meta-modeling approach to the specification, but there is not sufficient tool support for using OCL at meta-modeling level.

Finally, for the State class, the exact interaction within the implementation, whereby the handler requests a State subclass instance to give back to the handler, is not specified but it could be.

5.4. Tool Support

In (Zhu et al., 2009), we report a tool LAMBDES-DP for recognising design patterns in UML models. We have conducted some experiments to see how accurately it does this. There are two types of errors that could occur in pattern recognition. A false positive error means that a design is incorrectly recognized as an instance of a pattern because the specification of the pattern is not restrictive enough to rule out incorrect uses. In contrast, a false *negative* error means that a design is incorrectly *rejected* as an instance of a pattern because the specification is too *restrictive* to take into account some valid uses. We produced models based on the GoF diagrams for each design pattern and used the tool to look for the pattern within the design. In each case, the tool found the pattern so the false negative error rate was 0%. Similarly, we used the tool to look for every other pattern in each design. When only the structural features of design patterns were used, the rate of false positive errors was 22%. In contrast, when behavioral features were also used to check against the sequence diagrams in the designs, the tool did not find the other patterns before time out, so the false positive error rate was also 0%, if failure to find is interpreted as rejection. This confirms the assertion in (Bayley and Zhu, 2008b) that structural features on their own are not sufficient to identify patterns.

6. Conclusion

In this paper, we systematically presented and advanced a meta-modelling approach to the formalisation of design patterns, one that captures both structural and behavioural features and that specifies variants systematically in a readable formal notation. It enables formal reasoning about patterns and their composition and transformation (Bayley and Zhu, 2008a), and facilitates automatic tool support for applying patterns at the design stage (Zhu et al., 2009). The advantages of the approach are demonstrated by examples and justified by case studies and experiment results.

For future work, we will further develop our prototype tool LAMBDES-DP. Also, it will be interesting to conduct further case studies, perhaps into the patterns for distributed systems, where dynamic behavioral features play a dominant role.

References

Albin-Amiot, H., Cointe, P., Guéhéneuc, Y.-G., Jussien, N., 2001. Instantiating and detecting design patterns: Putting bits and pieces together. In: Proceedings of the 16th IEEE International Conference on Automated Software Engineering (ASE 2001), 26-29 November 2001, Coronado Island, San Diego, CA, USA. IEEE Computer Society, pp. 166–173.

Bayley, I., Zhu, H., 2007. Formalising design patterns in predicate logic. In: Proceedings of the 5th IEEE International Conference on Software Engineering and Formal Methods (SEFM 2007). pp. 25–36.

Bayley, I., Zhu, H., 2008a. On the composition of design patterns. In: Zhu, H. (Ed.), Proceedings of the 8th Intenational Conference on Quality Software (QSIC'08). IEEE Computer Society, Oxford, UK., pp. 27–36.

Bayley, I., Zhu, H., 2008b. Specifying behavioural features of design patterns in first order logic. In: Proceedings of the IEEE 32nd International Computer Software and Applications Conference (COMPSAC 2008). pp. 203–210.

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

Beyer, D., Noack, A., Lewerentz, C., 2005. Efficient relational calculation for software analysis. IEEE Trans. Software Eng. 31 (2), 137–149.

Blewitt, A., Bundy, A., Stark, I., 2005. Automatic verification of design patterns in Java. In: Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering (ASE 2005). ACM Press, pp. 224–232. URL `http://www.inf.ed.ac.uk/ stark/autvdp.html`

Coad, P., 1992. Object-oriented patterns. Communications of the ACM 35 (9), 152– 159, special issue on analysis and modeling in software development.

Dong, J., Zhao, Y., Peng, T., 2007. Architecture and design pattern discovery techniques - a review. In: Arabnia, H. R., Reza, H. (Eds.), Software Engineering Research and Practice. CSREA Press, pp. 621–627.

Eden, A. H., 2001. Formal specification of object-oriented design. In: International Conference on Multidisciplinary Design in Engineering, Montreal, Canada.

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

Eden, A. H., Gasparis, E., Nicholson, J., 2007. LePUS3 and Class-Z reference manual. Tech. Rep. CSM-474, University of Essex, UK, ISSN 1744-8050.

Eden, A. H., Hirshfeld, Y., 2001. Principles in formal specification of object oriented design and architecture. In: Proceedings of the 2001 Conference of the Centre for Advanced Studies on Collaborative Research (CASCON '01). IBM Press, pp. 3–18.

Eden, A. H., Yehudai, A., Gil, J., 1997. Precise specification and automatic application of design patterns. In: ASE. pp. 143–152.

Elaasar, M., Briand, L. C., Labiche, Y., 2006. A metamodeling approach to pattern specification. In: Proceedings of 9th International Conference on Model Driven Engineering Languages and Systems, (MoDELS 2006), Genova, Italy, October 1-6, 2006. Vol. 4199 of Lecture Notes in Computer Science. Springer, pp. 484–498.

France, R. B., Kim, D.-K., Ghosh, S., Song, E., 2004. A UML-based pattern specification technique. IEEE Trans. Softw. Eng. 30 (3), 193–206.

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

Gasparis, E., Eden, A. H., Nicholson, J., Kazman, R., 2008a. The design navigator: charting Java programs. In: Proceedings of the 30th International Conference on Software Engineering (ICSE 2008), Leipzig, Germany, May 10-18, 2008, Companion Volume. pp. 945–946.

Gasparis, E., Nicholson, J., Eden, A. H., 2008b. LePUS3: An object-oriented design description language. In: Proceedings of 5th International Conference on Diagrammatic Representation and Inference (Diagrams 2008), Herrsching, Germany. Vol. 5223 of Lecture Notes in Computer Science. Springer, pp. 364–367.

Guennec, A. L., Sunyé, G., Jézéquel, J.-M., 2000. Precise modeling of design patterns. In: Proceedings of the Third International Conference on The Unified Modeling Language (UML 2000). York, UK. Vol. 1939 of Lecture Notes in Computer Science. Springer, pp. 482–496.

Huang, H., Zhang, S., Cao, J., Duan, Y., 2005. A practical pattern recovery approach based on both structural and behavioral analysis. J. Syst. Softw. 75 (1-2), 69–87.

Khomh, F., Guéhéneuc, Y.-G., 2008. Do design patterns impact software quality positively? In: 12th European Conference on Software Maintenance and Reengineering (CSMR 2008), April 1-4, 2008, Athens, Greece. IEEE, pp. 274–278.

Kim, D.-K., 2004. A meta-modeling approach to specifying patterns. Ph.D. thesis, Colorado State University, Fort Collins, CO, USA.

Kim, D.-K., Lu, L., 2006. Inference of design pattern instances in UML models via logic programming. In: Proceedings of the 11th International Conference on Engineering of Complex Computer Systems (ICECCS 2006), Stanford, California, USA. IEEE Computer Society, pp. 47–56.

Kim, D.-K., Shen, W., 2007. An approach to evaluating structural pattern conformance of UML models. In: Proceedings of the 2007 ACM Symposium on Applied Computing (SAC'07), Seoul, Korea. ACM Press, pp. 1404–1408.

Kim, D.-K., Shen, W., 2008. Evaluating pattern conformance of UML models: a divide-and-conquer approach and case studies. Software Quality Journal 16 (3), 329–359.

Krämer, C., Prechelt, L., 1996. Design recovery by automated search for structural design patterns in object-oriented softwar. In: 3rd Working Conference on Reverse Engineering (WCRE'96). IEEE Computer Society, pp. 208–215.

Lano, K., Bicarregui, J. C., Goldsack, S., 1996. Formalising design patterns. In: BCS-FACS Northern Formal Methods Workshop, Ilkley, UK. pp. 11(1 – 20).

Lauder, A., Kent, S., 1998. Precise visual specification of design patterns. In: Lecture Notes in Computer Science Vol. 1445. ECOOP'98, Springer, pp. 114–134.

Mak, J. K. H., Choy, C. S. T., Lun, D. P. K., 2004. Precise modeling of design patterns in UML. In: 26th International Conference on Software Engineering (ICSE'04). pp. 252–261.

Maplesden, D., Hosking, J., Grundy, J., 2001. A visual language for design pattern modelling and instantiation. In: Proceedings of IEEE International Symposium on Human-Centric Computing Languages and Environments (HCC'01). pp. 338–339.

Maplesden, D., Hosking, J., Grundy, J., 2002. Design pattern modelling and instantiation using DPML. In: Proceedings of the Fortieth International Conference on Tools Pacific (TOOLS Pacific 2002). Australian Computer Society, Inc., Darlinghurst, Australia, Australia, pp. 3–11.

Mikkonen, T., 1998. Formalizing design patterns. In: Proc. of ICSE'98, Kyoto, Japan. IEEE Computer Society, pp. 115–124.

Niere, J., Schäfer, W., Wadsack, J. P., Wendehals, L., Welsh, J., 2002. Towards pattern-based design recovery. In: Proceedings of the 22rd International Conference on Software Engineering (ICSE 2002), Orlando, Florida, USA. pp. 338–348.

Nija Shi, N., Olsson, R., 2006. Reverse engineering of design patterns from java source code. In: Proc. of ASE'06, Tokyo, Japan. pp. 123–134.

OMG, 2004. Unified modeling language: Superstructure, version 2.0, formal/05-07-04.

PLAC, 2007. The first international workshop on patterns languages: Addressing challenges. Available online at URL: http://www.engr.sjsu.edu/ fayad/workshops/PLAC07, accessed on 12 Sept. 2007.

Seemann, J., von Gudenberg, J. W., 1998. Pattern-based design recovery of Java software. In: SIGSOFT '98/FSE-6: Proceedings of the 6th ACM SIGSOFT international symposium on Foundations of software engineering. pp. 10–16.

Taibi, T., 2006. Formalising design patterns composition. Software, IEE Proceedings 153 (3), 126–153.

Taibi, T., Check, D., Ngo, L., 2003. Formal specification of design patterns-a balanced approach. Journal of Object Technology 2 (4).

Winn, T., Calder, P., 2003. A pattern language for pattern language structure. In: Proceedings of the 2002 conference on Pattern languages of programs (CRPIT'02). Australia Computer Society, Inc., pp. 45–58.

Zdun, U., Avgeriou, P., 2005. Modelling architectural patterns using architectural primitives. In: 20th annual ACM SIGPLAN conference on Object-Oriented Programming, Systems, Languages and Applications (OOPLSA'05), San Diego, California. pp. 133–146.

Zhu, H., Shan, L., 2006. Well-formedness, consistency and completeness of graphic models. In: Proc. of UKSIM'06, Oxford, UK. pp. 47–53.

Zhu, H., Shan, L., Bayley, I., Amphlett, R., 2009. A formal descriptive semantics of UML and its applications. In: Lano, K. (Ed.), UML 2 Semantics and Applications. John Wiley & Sons, Inc., (In press).

Zhu, H., Bayley, I., Shan, L., Amphlett, R., 2009. Tool support for design pattern recognition at model level. In: Pro-

ceedings of the IEEE 33nd International Computer Software and Applications Conference (COMPSAC’09), Seattle, Washington, USA. (In press)