Towards a formal model of patterns and pattern languages

Alireza Rouhi, Bahman Zamani[1]

Department of Software Engineering, University of Isfahan, Isfahan, Iran
{rouhi,zamani}@eng.ui.ac.ir

Abstract

Context: Software patterns are solutions to recurring design problems. Patterns behave socially in the forms of Pattern Languages (PLs) to resolve problems. In a simple statement, a PL is a network of related patterns which provides a step-by-step process to solve a given problem in a specific context. Despite the popularity of applying PLs in practice, lack of formal basis on the patterns' inter-relationships in general, and PLs in particular, makes development of supporting tools difficult.

Objective: Based on the algebra of design patterns, we aimed at presenting a new formalism for patterns and PLs.

Method: We revised and extended the Graphic extension of EBNF (GEBNF) and applied it to model Unified Modeling Language (UML) class and sequence diagrams which are required to model the popular design patterns. Also, we presented the formal semantics of commonly used patterns' inter-relationships which pave the way for presenting our PL formalism. To show the applicability of our presented formalism on patterns and PLs, we presented the formal model of the Broker PL as a case study.

Results: Use of the presented formalism to model patterns and PLs will make them more expressive, readable, and understandable as well.

Conclusion: We have proposed a formalism for defining a PL in a more accurate manner. This formalism facilitates formal verification of patterns and PLs, particularly when it is intended to build a tool for such verification.

*Corresponding author
Email address: zamani@eng.ui.ac.ir (Bahman Zamani)

Keywords: Formal modeling, Revised and extended GEBNF, Design patterns, Patterns' inter-relationships, Pattern languages
2010 MSC: 00-01, 99-00

1. Introduction

Patterns are reusable assets that help transfer the knowledge of domain experts to novices [1, 2]. Despite the popularity of applying patterns in many fields and specifically in software engineering, lack of formal foundation makes it difficult to develop support tools in order to utilize the full power of software patterns [3, 4, 5, 6].

Undoubtedly, application of patterns in a design model improves the quality of produced artifacts and reduces software maintenance costs [7, 8]. In addition, considering patterns' inter-relationships on their application would make a design more concise and more cohesive than applying patterns in isolated form [3, 5, 9]. Also, the best practice in the application of patterns regarding their inter-relationships is the application of PLs to solve problems as a whole [5, 10]. A PL is defined as a collection of inter-related patterns which can be used to solve a set of specific software development problems [9] like developing a distributed Object Request Broker (ORB) middleware [10], an enterprise application architecture [11], to name a few.

The rapid growth of patterns and their inter-relationships will cause a designer to get into trouble on verifying the applied patterns and PLs. As a result, applying patterns and PLs will need verification tool support [12]. One of the main drawbacks on developing the verification tools for PLs is that the patterns and PLs often have been presented informally or semi-formally in literature [1, 9, 11]. Therefore, to facilitate the process of developing supporting tools on verifying the applied patterns and PLs in practice, we need to formalize the underlying concepts [3, 6]. Although the literature of the last two decades of software patterns, beginning from the publishing of the seminal Gang-of-Four (GoF) design patterns book [1], illustrates several researches which have

been concentrated on formalizing single patterns [3, 13, 14, 15, 16], a few of the current researches consider the patterns' inter-relationships and patterns' compositions [17, 18, 19, 20] as well. Thus, we examine the main researches [5, 6, 21] which provide a solid foundation to our ongoing research regarding formalizing patterns and PLs.

Zdun [6] has presented the idea of using a formal grammar like Backus-Naur Form (BNF) to provide a systematic pattern selection approach from a collection of related patterns. This grammar uses annotations representing the consequences of applied patterns on the design quality goals.

Following the proposed grammar of Zdun [6], the verification problem of applying patterns of a given PL has been addressed by Zamani and Butler [5] with introducing the Pattern Language Verifier (PLV) concept. However, their presented PLV is limited to only one PL, that means it verifies (only) the Patterns of Enterprise Application Architecture (PofEAA) PL's patterns [11] on a design model.

The main purpose of our research is to generalize the idea of PLV [5], such that for any given PL it is possible to build the corresponding PLV, automatically. To make this happen, one needs a formalism to describe the patterns of a PL as well as the relationships between the patterns. Without such a formalism, there will be ambiguities in the construction of a PLV for the given PL. For instance, in PofEAA PL [11], Fowler has classified all the patterns into layers and sub-layers. This classification has been used by Zamani and Butler [5] to verify the *pattern-layer* relationships–i.e, whether a selected and applied pattern in a specific layer is allowed or not–brings severe ambiguities on distinguishing the Layers pattern [22] from the patterns' layers of the language where a Service Layer pattern [11] is placed in the *Service* layer (see Table 11 in [23, p. 90]). As we know, the known Layers pattern [24, 25] can decompose a large and complex monolithic system–which is a mixed of high and low-level abstractions–into layers. The number of layers and services that each layer provides must be selected carefully. In addition, each layer can only use the provided services of its underneath layer. In fact, based on our proposed formal model, there is no need to

include the classification/layering concept in a PL definition at all. By adopting the proposed formal PL definition, even though a classification/layering is inevitable, applying a Layers pattern and placing the patterns in the appropriate layers can resolve the problem. As a result, the problem of verifying patterns' relationships of PofEAA PL is reduced to the *pattern-pattern* relationships.

This paper aims to present a nearly unified formalism for the popular patterns and PLs. This formalism will be applicable to patterns of any PL which its patterns' solutions are represented by UML class and sequence diagrams.

Even though a few formal models can be found in literature [18, 26, 27], which are discussed in the next section, the formal model of [21] is relatively complete regarding formalizing single patterns. Zhu and Bayley [21] have presented an algebra for the GoF design patterns [1] and pattern compositions.

To the best of our knowledge, the pattern composition approach of Zhu and Bayley [21], by exploiting from the *commutative* and *associative* relation * (see Table 5 for more detail), solves problems in small cases similar to the Request Handling Framework which applies $\langle \mathit{Command}, \mathit{CommandProcessor}, \mathit{Memento}, \mathit{Strategy}, \mathit{Composite} \rangle$ patterns with a limited patterns' inter-relationships (see [21] for more detail). Also, the pattern composition formalism does not consider the current forces of the design context as well as the consequences of the applied patterns. Therefore, there is not any pattern sequence proposal to the designer from the pattern composition formalism. In other words, this is the designer who examines the candidate patterns and selects the appropriate patterns to compose. As a result, since the decision of the appropriate ordering of patterns is the designer responsibility, the produced design will probably be incorrect and cannot be free of contradictions as well.

Thus, towards presenting a formal model of patterns' inter-relationships and PLs, we investigated the algebra of [21] and reached to the following question: Is it possible to formalize a PL using the algebra of design patterns of Zhu and Bayley [21] or not? In this paper, we will try to answer the posed "question" after presenting our proposed formalism for a given PL.

The presented formalism of single patterns is based on the formal model of

Zhu and Bayley [21]. Since most of the popular pattern collections use UML to specify their pattern solutions (the GoF design patterns [1], PofEAA [11], and the Broker PL [22, 28], to name a few), the abstract syntax of UML is specified using the GEBNF notation [3]. This specification includes class diagrams as well as sequence diagrams. Note that these diagrams are adequate to model the structural and behavioral features of the aforementioned commonly used pattern collections. Because the order of fields is required in some GEBNF production rules like the *quantifier formula* (Rule 35) definition in the pattern specification (Section 5), we add the ordered production rule–an ordered sequence of fields–to the GEBNF notation. Then, inspired from the idea of the induced functions [3], resulting from the UML's representing production rules as well as some other auxiliary functions, a formal model of patterns in the revised and extended GEBNF is presented. Also, the formal semantics of popular patterns' inter-relationships are presented which pave the way for formalizing a given PL. In the context of a given pattern of a PL, the systematic pattern selection method of Zdun [6] helps pick a single pattern from several candidate related patterns. Finally, to show the applicability of the proposed formalism, the formal model of Broker PL [10, 22, 28] is presented as a case study.

The main contributions of this paper include:

- Revising and extending the GEBNF metamodeling notation of Bayley and Zhu [3],
- Applying the GEBNF notation to model a design pattern inspired from the pattern specification scheme of Bayley and Zhu [3],
- Presenting a first-order predicate logic formalism for a given PL inspired from the automata theory's transition function [29] and the systematic pattern selection method of Zdun [6],
- Presenting a formal semantics for the popular patterns' inter-relationships,
- Illustrating the applicability of the proposed formalism regarding patterns and PLs on formalizing the Broker PL [22, 28] as a case study,

- Presenting an eclipse-based plug-in parser to write and validate the GEBNF models [30], and
- Presenting an eclipse-based plug-in parser to write and validate the design patterns in the presented formalism [31].

The rest of paper is organized as follows. Section 2 explores the related works on formalizing patterns and PLs. Section 3 presents the revised and extended GEBNF metamodeling notation. Section 4 applies the revised and extended GEBNF notation to model the UML class and sequence diagrams. Section 5 presents a pattern specification scheme in GEBNF and the specification of Broker pattern as an example. Section 6 is dedicated to the proposed PL formalism. Section 7 presents a formal model of the Broker PL as a case study which shows the presented formalism applicability. Section 8 discusses the presented formalism with related works. Section 9 presents the developed tools regarding the GEBNF and the pattern specification models. Finally, Section 10 concludes the paper and directs for future works.

2. Related Work

In this section, we present the mainstream and the state of research on patterns and PLs formalization in literature. One classification which has been presented in [4] classifies the current researches on the design patterns into two main categories: 1) Pattern Application, and 2) Pattern Detection and Identification. According to this classification, current research belongs to the first category and will provide a foundation for applying patterns and PLs.

The related works on formalizing patterns can be classified into three main categories: *single* patterns, *composition/combination* of patterns, and *PLs*. Most of the existing researches are limited to single patterns [13, 14, 15, 16] and combination of patterns [17, 18, 19, 20]. Note that among other purposes of presenting a formalism to describe a single pattern, having a unified visual approach will be helpful to improve pattern authors' and pattern users' perception [14, 19, 20, 32].

Table 1: Literature review on formalizing patterns and their inter-relationships

Research	**Pattern**			**Aspect**	
	Single	**Composition**	**PL**	**Structural**	**Behavioral**
Zamani and Butler [5]		✓	✓	✓	
Zdun [6]		✓	✓		
France et al. [14]	✓			✓	✓
Bayley and Zhu [17]	✓	✓		✓	✓
Bottoni et al. [18]	✓	✓		✓	✓
Taibi and Ngo [20]	✓	✓		✓	✓
Dong et al. [36]	✓	✓		✓	✓
Eden et al. [37]	✓			✓	✓

To the best of our knowledge, there is no language in literature which demonstrates PLs without considering the implementation issues. In other words, 175 there is no formalism for representing PLs at all. Also, it must be mentioned that most of the existing pattern catalogs and pattern collections have been written with the natural languages [26, 33, 34, 35]. Therefore, a few formal techniques regarding the representation of patterns and their inter-relationships are found in literature [18, 26, 27].

180 Table 1 presents a summary of the related works on formalizing patterns and their inter-relationships. Here, we examine the sources based on their formalism on **single** patterns, **composition/combination** of patterns, and **PLs**. In addition, since most of the resources in literature discuss the design patterns from two aspects, i.e., the **structural** and **behavioral**, we also include them 185 in Table 1.

Zdun [6] presents a systematic pattern selection approach to pick one solution among several candidate patterns and PLs. Note that these patterns are documented by different authors in various pattern forms. So, a formal BNF-like grammar is presented. This grammar is used to represent the given PL which 190 is annotated with the consequences of patterns on the system quality goals.

Also, it uses a design space analysis technique to analyze the solution space of a complex design. Of course, this approach is not applicable to resolve very large design problems. Because of the complexity of decision making on selecting pattern variants or pattern alternatives and inadequacy of the grammar annotations, the Questions, Options, Criteria (QOC) design space analysis has been used. This technique (inspired from the Human-Computer Interaction (HCI) field) in combination with the annotated grammar provides enough rationales for pattern selection and decision making. Here, the problem domain is mapped to a set of solution patterns and Pattern Sequences (PSs). This research aims at presenting a grammar to support tools and managing efficiently the design space to cope with the complexity of selecting related patterns among PLs.

Zamani and Butler [5] recently introduced the verification problem of a design model which applies the patterns of PofEAA [11]. The given model is verified to detect and rectify the underlying bugs and problems. As a result, the quality of the obtained model will improve. The authors have extracted a set of patterns from [11] in a PL form and have represented its annotated grammar (inspired from the approach of Zdun [6]). Also, exploiting from the compiler of a typical programming language, they have introduced the PLV concept. The PLV here verifies the applied patterns of PofEAA [11] on a given design model.

Elaasar [38] discusses the detection of patterns and anti-patterns in design models of Object Management Group (OMG) Meta Object Facility (MOF) metamodel-based languages. Here, the domain-specific patterns are specified with a new domain-specific visual language called Visual Pattern Modeling Language (VPML). The VPML metamodel is small and its defined concepts and relations are familiar to the pattern experts. A prototype of this tool is implemented on the Eclipse platform. Furthermore, its semantics are defined with mappings to Query, View, Transformations (QVT)-R [39] which is a formal and general purpose model transformation language.

Eden [40] has developed LanguagE for Patterns Uniform Specification (LePUS) which is a formal and visual language for the *CodeCharts* specifications [37, 41]. LePUS has more readable structures and design motifs to specify the object-

oriented programming concepts in general and design patterns in particular. Each pattern formula is shown by logic statements. Given any program model and the specification formula of a specific pattern, the model conforms to the pattern if and only if it satisfies the formula representing pattern or its logic statements. Also, Nicholson et al. [32] present a toolkit to verify whether a given model conforms to the design patterns (for example, the Composite pattern of the GoF catalog [1]) in the LePUS language or not.

Bayley and Zhu [42] following [43] have proposed a formal language as well as a set of operators for composing related patterns. These operators have been used to compose the GoF design patterns [1] and sketch the patterns' inter-relationships. The presented set of operators here is more precise and fine-grained than their previous universal composition operator which has been presented in [43].

The theory and formal language of Eden [40, 44] is very similar to the pattern specification scheme of Zhu and Bayley [21]. However, only the static structural features of object-oriented design patterns are covered by the Eden's approach and the dynamic behavioral features are ignored [3]. So, we aim to base our formal theory of patterns on the theory of Bayley and Zhu [3] which supports metamodeling of both the structural and behavioral features of design patterns. We revise and extend the GEBNF metamodeling notation [42] to provide more abstraction and flexibility in the specification of rules. Then, we apply the revised notation to model class and sequence diagrams of UML which is the popular modeling language of design pattern solutions [1, 11]. Also, inspired from the notion of systematic pattern selection method of Zdun [6] and exploiting from the induced functions of the UML class and sequence diagrams, we present a formal model for a given PL.

3. GEBNF Metamodeling: Revised and Extended

GEBNF has extended Extended BNF (EBNF)–the extension of BNF which is used to define the syntax of programming languages–to make possible the

definition of graphical modeling languages like UML [3]. First, using a facility called “reference” for nonterminal referencing, GEBNF simplifies the specification of graphical structures like UML class diagrams and sequence diagrams. Second, through the introduction of “field labeling” feature in GEBNF, it is 255 straightforward to extract functions representing the GEBNF syntax definition for the construction of a first-order logic predicate language [3].

The abstract syntax of each modeling language in GEBNF is defined as a quadruple $\langle N, T, R, S\rangle$. Here, N denotes a finite set of nonterminals (which are strings beginning with an upper-case letter), T denotes a finite set of terminals 260 (which are surrounded by double quotation marks, ‘ ’), $R \in N$ is the root symbol and S is the finite set of production rules in the form of $Y ::= \mathit{Exp}$, where $Y \in N$ and Exp takes one of the forms that are displayed in Table 2.

The X_i’s, i.e., fields that are declared in Table 2 will take one of the forms that are displayed in Table 3.

265 In the Table 3, $Z \in N \cup T$, i.e., the field Z can be *nonterminal* or *terminal*. The L_i’s which are field labels in the GEBNF notation (and intentionally neglected from the language quadruple, i.e., $\langle N, T, R, S\rangle$) will be helpful on the extraction of the induced functions from the production rules.

4. The Abstract Syntax of UML in the Revised and Extended GEBNF

270 Since the complete abstract syntax of UML is too lengthy to present here, we have presented the syntax in [25]. So, a short list of rules has been included here to show the various production rules of a simplified UML model in the revised and extended GEBNF.

Each UML model to model the structural or behavioral features of a given 275 domain is defined as an unordered list of a class diagram field, *ClassDiag*, and an optional sequence diagram field, *SequenceDiag*, as follows (Rule 1).

To model *Class* and *Interface* classifiers, we use label *classes* which denotes a nonempty set of *Class* field. In addition, we can have relationships among the constituent nodes of a class diagram. Because the order of the constituent

Table 2: Meanings of GEBNF notation (revised and extended from [3, 43, 45])

Notation	**Meaning**
$Y ::= L_1 : X_1,$ $L_2 : X_2,$ $...,$ $L_n : X_n$	Y is defined as an *ordered list* of n label:fields, i.e., $X_1, X_2, ..., X_n$. Here, the sequence of fields are separated by comma (e.g., Rule 22).
$Y ::= X_1 \mid X_2 \mid ... \mid X_n$	Y is defined as a *list of choices*, i.e., fields X_1, X_2 or X_n. Here, the order of fields is unimportant (e.g., Rule 19).
$Y ::= L_1 : X_1;$ $L_2 : X_2;$ $...;$ $L_n : X_n$	Y is defined as an *unordered list* of n label:fields. Here, the fields are separated by semicolon (e.g., Rule 4).
$Y ::= L_1 : X_1, ..., L_k : X_k;$ $L_{k+1} : X_{k+1};\ ...;\ L_n : X_n$	Y is defined as a *mixed list* of n label:fields, i.e., $X_1, ..., X_k$ list is ordered and $X_{k+1}, ..., X_n$ list is unordered (e.g., Rule 5). Here, it is possible to define some fields as alternation too. For example, X_k can be defined as $X_{k'} \mid X_{k'+1} \mid ...$.

Table 3: Fields of a rule's right-hand side in GEBNF notation

Field	Description
‘ ’	Double quotes represents a terminal (e.g., terminal ‘return’ in Rule 26).
Z	Specifies the basic form of a field (e.g., String field in Rule 3).
Z^*	Specifies zero or more repetition of field Z (including *null* situation), i.e., a set of Z fields (e.g., Relation field in Rule 2).
Z^+	Specifies the field Z has repetition (excluding *null* situation), i.e., a nonempty set of Z fields (e.g, Lifeline field in Rule 7).
$[Z]$	Specifies the field Z is optional which is surrounded by square brackets (e.g., String field name in Rule 4). Note that this field can be a list too (e.g., Parameters of a function in Rule 21).
$\underline{Z}$	This means the Z field has been defined as a reference which is denoted by an *underlined* character similar to a hyperlink in a HTML document which guides a user to a web page (e.g., Activation field in Rule 11).
(Z, Z)	*Ordered pair*, says the first field has precedence in occurrence to the second field. For example, in sequence diagrams, given $(M_1 : \mathit{Message}, M_2 : \mathit{Message})$ means message M_1 should appear before the message M_2 (e.g., Ordered message pair field in Rule 7).

280 fields of a class diagram is irrelevant, we apply the unordered list to define a class diagram (Rule 2). An *Operation* field defines an ordered list including an optional visibility qualifier and a signature along with another unordered list which includes boolean field labels, *isAbstract*, *isQuery*, *isLeaf*, and *isStatic*.

A sequence diagram is defined as an unordered list of a nonempty set of 285 lifelines field labeled with *lifelines*, a nonempty set of message field labeled with *msgs* which are exchanged among lifelines, and a set of zero or more ordered-pairs of messages (Rule 7). Each lifeline is defined as an ordered list of a lifeline head (Rule 9) field labeled with *name* and zero or more activations. We define the *isStatic* boolean flag to determine whether the lifeline head refers to a class 290 (*isStatic* = *true*) or an optional object name followed by a ':' and the referred class (*isStatic* = *false*). Each message is defined by an ordered list of an optional sender event and an optional receiver event with an unordered list of the message identifier, the optional signature of message target, an optional message kind, and an optional message sort. Each *Property* is defined as an ordered list of 295 an optional qualifier field *VisibilityKind* (either *public, '+'*; *private '-'*; *protected '#'*; or *package, '∼'*); a required name followed by an optional type, an optional multiplicity, an optional default value, and an optional set of modifiers specifying the additional characteristics of the property. Here, for the sake of brevity, we left-out the *Property*'s rule definition from the grammar list (see [25] for more 300 detail).

$$
\begin{aligned}
\mathit{UMLModel} &::= \mathit{cd} : \mathit{ClassDiag};\ [\mathit{sd} : \mathit{SequenceDiag}] && (1)\\
\mathit{ClassDiag} &::= \mathit{classes} : \mathit{Class}^{+};\ \mathit{rels} : \mathit{Relation}^{*} && (2)\\
\mathit{Class} &::= \mathit{name} : \mathit{String}, \mathit{attrs} : \mathit{Property}^{*}, \mathit{opers} : \mathit{Operation}^{*}; &&\\
&\quad [\mathit{isAbstract} : \mathit{Boolean}];\ [\mathit{isInterface} : \mathit{Boolean}]; &&\\
&\quad [\mathit{isAssocClass} : \mathit{Boolean}];\ [\mathit{isFinal} : \mathit{Boolean}] && (3)\\
\mathit{Relation} &::= [\mathit{name} : \mathit{String}];\ \mathit{ends} : \mathit{End}^{+}; &&\\
&\quad \mathit{kind} : \mathit{Kind};\ [\mathit{isDerived} : \mathit{Boolean}]; && (4)\\
\mathit{Operation} &::= [\mathit{visibility} : \mathit{VisibilityKind}], \mathit{sig} : \mathit{Signature}; &&\\
&\quad [\mathit{isAbstract} : \mathit{Boolean}];\ [\mathit{isQuery} : \mathit{Boolean}]; &&\\
&\quad [\mathit{isLeaf} : \mathit{Boolean}];\ [\mathit{isStatic} : \mathit{Boolean}] && (5)\\
\mathit{Signature} &::= \mathit{name} : \mathit{String}, \mathit{params} : \mathit{Parameter}^{*}, &&\\
&\qquad [\text{‘ : ’}, [\mathit{type} : \mathit{Type}], [\text{‘[’}, \mathit{mult} : \mathit{Multiplicity}, \text{‘]’}], &&\\
&\qquad\quad [\mathit{prop} : \mathit{OperProperty}^{*}]] && (6)\\
\mathit{SequenceDiag} &::= \mathit{lifelines} : \mathit{Lifeline}^{+};\ \mathit{msgs} : \mathit{Message}^{+}; &&\\
&\quad \mathit{msg_order} : (\underline{\mathit{Message}}, \underline{\mathit{Message}})^{*} && (7)\\
\mathit{Lifeline} &::= \mathit{name} : \mathit{LifelineHead}, \mathit{activations} : \mathit{Activation}^{*}; &&\\
&\quad \mathit{isStatic} : \mathit{Boolean} && (8)\\
\mathit{LifelineHead} &::= [\mathit{object_name} : \mathit{String}], \text{‘ : ’}, c : \underline{\mathit{Class}} && (9)\\
\mathit{Activation} &::= \mathit{start} : \mathit{Event}, \mathit{finish} : \mathit{Event}, \mathit{others} : \mathit{Event}^{*} && (10)\\
\mathit{Event} &::= \mathit{lifeline} : \mathit{Lifeline}, \mathit{actor} : \underline{\mathit{Activation}} && (11)\\
\mathit{Message} &::= [\mathit{sender} : \underline{\mathit{Event}}], [\mathit{receiver} : \underline{\mathit{Event}}];\ \mathit{msgID} : \mathit{MessageID}; &&\\
&\quad [\mathit{sig} : \underline{\mathit{Signature}}];\ [\mathit{kind} : \mathit{MessageKind}]; &&\\
&\quad [\mathit{sort} : \mathit{MessageSort}] && (12)
\end{aligned}
$$

Now, after presenting the abstract syntax of UML class and sequence diagrams, we discuss the improvements of our presented syntax compared with the

previous syntax of the simplified UML in [3]:

Field labeling instead of Field naming: Production rules and their induced functions are more expressive and understandable with field *labeling* than field *naming*. For instance, most of the defined production rules like Class (Rule 3), Relation (Rule 4), just to name a few, have a name string field. Field naming causes ambiguity on the name fields of the mentioned rules, i.e, the *name* term can stand for the rule's string field *name* while it is used for naming the rule's fields.

More abstraction on the rule definition: For example, $ClassDiagram ::= ..., assocs : Rel^*, inherits : Rel^*, compag : Rel^*$ rule which distinguishes relationships (compared with the Rule 2) and defines fields as an ordered list while the unordered list has more abstraction here [3, p. 6].

Resolving inaccuracies and ambiguous usage of the grammar syntax: For example, $[mult : Multiplicity]$ in the *Property* rule and $mult : [Multiplicity]$ in the *End* rule, which must be in the $[label : field]$ format in all of the defined rules. The Lifeline rule places the *class* name field before than the *object* name field while it must be reversed [3, p. 6].

Incompleteness in the production rules' definition The production rule like *Multiplicity* is specified so that all fields are optional. This has contradiction with its required upper field. In addition, the optional declaration of a field denotation like $[Z^*]$ is equivalent to Z^*, i.e., there is no need to introduce it as an optional field too.

Completeness in terms of the number of rules, 29 [25] compared to 14 [3]: With the relatively complete list of rules presented here it is possible to specify various collections of patterns like GoF [1], PofEAA [11], and the Broker PL [9].

4.1. Induced First-Order Logic Functions

330 Based on the approach that Zhu [46] proposes, the construction of the induced first-order logic functions is as follows:

- Each field label in the right-hand side of any syntactic rule defines a function from the domain of the rule's left-hand side to the range of the label's corresponding field. For example, label *classes* in the class diagrams, de-
335 fines a function from the domain *ClassDiag* to the *Class* nodes of the diagram. Even though this instruction is applicable to all field labels without exception, but some fields are defined only to determine the alternative choices of other fields. For example, the *isStatic* field label in the lifeline rule, determines the status of a lifeline head name-*true* for a class lifeline
340 and *false* for the object lifeline.

- Because of the existence of the common field labels in different rules, like *name* field label in *Class*, *Operation*, etc., their induced function will be overloaded. For example, overloading $name : Class \rightarrow String$ and $name : Operation \rightarrow String$.

345 For the sake of conciseness and brevity, we ignore the list of the extracted induced functions here (see [25] for the function lists).

4.2. Auxiliary Functions

Now, based on the induced functions extracted from the rules defined in Section 4, we define auxiliary functions which are required later to specify for-
350 mally the design patterns in general (see [25] for the complete list of functions). For the sake of simplicity, on applying and referencing to the functions which their domains are trivial like *ClassDiag*, we will intentionally omit these arguments and instead use the function names. For example, instead of writing $classes(ClassDiag)$ to refer for the set of class nodes, we will use *classes* instead.
355 To check the existence of the relationships like *association*, *dependency*, *composition*, *aggregation*, *inheritance*, and *realization*, we use the functions *associates*, *depends*, *composes*, *aggregates*, *inherits*, *realizes*, respectively (note that

all of these functions with the same form, $function_name : (Class \times Class) \nrightarrow Boolean$, check the existence of the mentioned relationships between the given
360 two classifiers). Table 4 lists more functions (see [25] for more detail on the formal definition of listed functions).

Table 4: Functions representing the inter-classifiers' relationships (revised from [3])

Function	**Explanation**
$isInterface : Class \nrightarrow Boolean$	Checks whether or not the given classifier is an interface.
$subs : Class \rightarrow \mathbb{P}\ Class$	Returns the subclasses of the given classifier.
$calls : (Operation \times Operation) \nrightarrow Boolean$, $calls : (Class \times Class) \nrightarrow Boolean$	Returns true if the first operand calls the second operand. The first function considers the hook method call too.
$precedes : (Message \times Message) \nrightarrow Boolean$	Given a message pair, checks whether or not the first message has occurred before than the second one in time.

4.3. Semantic Constraints

To complete the abstract syntax definition of UML, we exploit the induced and auxiliary functions (defined in Section 4.1 and Section 4.2) to specify the
365 complement first-order logic constraints of the UML design models (note that some of the Object Constraint Language (OCL) equivalents of these constraints have been presented in [47]). See [25] to find the complete list of the presented constraints.

1. A classifier in a generalization relationship cannot be general and specific simultaneously in a hierarchy:

$$\forall\, c : Class \bullet c \in classes \Rightarrow c \notin subs(c)$$

2. The general classifier in a generalization relationship cannot be final:

$$\forall\, c_i, c_j : \mathit{Class} \mid c_i \in \mathit{classes} \wedge c_j \in \mathit{classes} \bullet$$
$$\mathit{inherits}(c_j, c_i) = \mathit{true} \Rightarrow \neg\, \mathit{isFinal}(c_i)$$

3. For the following model elements, the mutually exclusive constraints must
370 be held:
   (a) Each class can be defined as an abstract or final exclusively:

$$\forall\, c : \mathit{Class} \bullet c \in \mathit{classes} \Rightarrow$$
$$(((\neg\, \mathit{isAbstract}(c) \wedge \mathit{isFinal}(c)) \vee (\mathit{isAbstract}(c) \wedge \neg\, \mathit{isFinal}(c)))$$
$$\vee$$
$$(\neg\, \mathit{isAbstract}(c) \wedge \neg\, \mathit{isFinal}(c)))$$

   (b) Each operation which is defined in a class can be set as a leaf or abstract; and as an abstract or static exclusively:

$$\forall\, c : \mathit{Class};\ o : \mathit{Operation} \bullet c \in \mathit{classes} \wedge o \in \mathit{opers}(c) \Rightarrow$$
$$(((\mathit{isLeaf}(o) \wedge \neg\, \mathit{isAbstract}(o)) \vee (\neg\, \mathit{isLeaf}(o) \wedge \mathit{isAbstract}(o)))$$
$$\wedge$$
$$((\mathit{isAbstract}(o) \wedge \neg\, \mathit{isStatic}(o)) \vee (\neg\, \mathit{isAbstract}(o) \wedge \mathit{isStatic}(o))))$$

   (c) Each relation's end can be aggregate or composite exclusively:

$$\forall\, r : \mathit{Relation};\ \mathit{end} : \mathit{End} \bullet r \in \mathit{rels} \wedge \mathit{end} \in \mathit{ends}(r) \Rightarrow$$
$$((\mathit{isComposite}(\mathit{end}) \wedge \neg\, \mathit{isAggregate}(\mathit{end}))$$
$$\vee$$
$$(\neg\, \mathit{isComposite}(\mathit{end}) \wedge \mathit{isAggregate}(\mathit{end})))$$

5. Pattern Specification

Now, after formalizing the domain of models, i.e., the abstract syntax of UML as well as the semantic constraints, we can present the formal model of any given pattern. Each pattern is specified as a first-order logic predicate. As
375 a result, a model which satisfies the predicate of a given pattern will be an instance of the mentioned pattern. A design pattern is defined as below [21]:

Definition 1. *(Formal Specification of Design Patterns). Each design pattern is defined as a triple member sequence* $\langle V, Pr_s, Pr_d \rangle$*.* V *is a set of free variables/components (appeared in the predicates* Pr_s *and* Pr_d*) which is represented as* $V = \{v_1 : T_1,\ v_2 : T_2,\ ...,\ v_n : T_n\}$ *with the corresponding variable types,* T_i*'s,* $i = 1..n$*. These types include basic types such as class, operation, property, lifeline, message, etc. or a set type such as* $\mathbb{P}\,T$ *(i.e., power set of type* T*, for example,* $\mathbb{P}\,Class$*) or* $\mathbb{P}(\mathbb{P}(T))$*, etc.* Pr_s *denotes the structural features of a design pattern (corresponding to the pattern's class diagram). And finally,* Pr_d *denotes the predicates representing the behavioral features of the specified design pattern (corresponding to the pattern's sequence diagram). To simplify the pattern specifications,* $Vars(P)$ *and* $Spec(P)$ *are used to denote the declared variables of pattern* P*, i.e.,* V*, and its predicate part, i.e.,* $Pr_s \wedge Pr_d$*, respectively.*

In the following, Section 5.1 presents the GEBNF definition of a design pattern. To clarify the specification of a given pattern in GEBNF, Section 5.2 is dedicated to the Broker pattern specification. Finally, Section 5.3 discusses applying patterns on a design model and the model conformance to a given pattern specification.

5.1. Definition of a Design Pattern Using the Revised GEBNF

Inspired from the pattern specification scheme of Bayley and Zhu [3], we specify each pattern in GEBNF as the following rule (Rule 13). It says a pattern includes a *name*, *components*, *static conditions*, and *dynamic conditions*, in order. Each *Formula* is a well-formed formula in the first-order logic [48, 49].

Each pattern is defined as an ordered list of pattern name, pattern components/variables (one or more), pattern structural/static conditions (zero or

more), and pattern behavioral/dynamic conditions (zero or more).

$$\begin{aligned} Pattern \quad ::= \quad & \textbf{patternName} : String, \\ & \textbf{components} : Declaration^{+}, \\ & \textbf{staticConditions} : StaticCondition^{*}, \\ & \textbf{dynamicConditions} : DynamicCondition^{*} \end{aligned} \tag{13}$$

The pattern's declaration field defines each pattern's constituent components/variables. Each declaration can be defined as a set inclusion (Rule 14), e.g.

405 $call_server() \in opers(Client)$ or a set membership, e.g. $\{forward_message(), transmit_message()\} \subseteq opers(Bridge)$. Each member is defined as an ordered-list of identifiers–which can include a list of classifier identifiers, function names, or attribute names (Rule 17).

$$Declaration \quad ::= \quad SetInclusion \mid SetMembership \tag{14}$$

$$SetInclusion \quad ::= \quad \text{‘}\{\text{’}, Member, [\text{‘},\text{’}, Member]^{*}, \text{‘}\}\text{’}, \text{‘} \subseteq \text{’}, Type \tag{15}$$

$$SetMembership \quad ::= \quad Member, \text{‘} \in \text{’}, Type \tag{16}$$

$$Member \quad ::= \quad id : String, [\text{‘(’}, [id : String, [\text{‘(’}, \text{‘)’}], [\text{‘},\text{’}]]^{*}, \text{‘)’}] \tag{17}$$

Production *Type* (Rule 18) defines various types which can be basic type,

410 function inclusion, power-set type or user-defined types.

$$\begin{aligned} Type \quad ::= \quad & BasicType \mid FuncInclusion \mid \\ & \text{‘}\mathbb{P}\text{’}, Type \mid id : String \end{aligned} \tag{18}$$

$$\begin{aligned} BasicType \quad ::= \quad & Boolean \mid Integer \mid String \mid Value \mid \\ & \underline{Class} \mid \underline{Operation} \mid \underline{Property} \mid \underline{Lifeline} \mid \\ & \underline{Message} \mid \underline{Parameter} \end{aligned} \tag{19}$$

$$Boolean \quad ::= \quad \text{‘}true\text{’} \mid \text{‘}false\text{’} \tag{20}$$

Each function inclusion (Rule 21) is defined by an ordered-list of function name, zero or more parameters, and an optional range value which is preceded by a comparison operator, if any. Each parameter (Rule 22) can be a classifier

identifier or a function name which is followed by an optional list. We use the
415 '::' notation to specify and resolve the scope of a function/method name or an attribute name of a classifier. In addition, to specify and resolve the scope of a function/method parameter name, we use the '♯' notation. The functions' range-values (Rule 23) can be a boolean, visibility-kind, direction-kind, basic type, function inclusion, or other string values like the '*' multiplicity.

$$\begin{aligned}
\mathit{FuncInclusion} &::= \mathit{funcName} : \mathit{String}, [\text{`('}, \mathit{Param}^*, \text{`)'}], \\
&\quad [\mathit{CompOp}, \mathit{RangeType}] && (21) \\
\mathit{Param} &::= \mathit{FuncInclusion} \mid \mathit{id} : \mathit{String}, \\
&\quad [\text{`::'}, \mathit{id} : \mathit{String}, [\text{`('}, \text{`)'}], [\text{`}\sharp\text{'}, \mathit{id} : \mathit{String}]] && (22) \\
\mathit{RangeType} &::= \mathit{Boolean} \mid \mathit{VisibilityKind} \mid \mathit{DirectionKind} \mid \\
&\quad \mathit{BasicType} \mid \mathit{FuncInclusion} \mid \mathit{String} && (23) \\
\mathit{CompOp} &::= \text{`}<\text{'} \mid \text{`}\leq\text{'} \mid \text{`}>\text{'} \mid \text{`}\geq\text{'} \mid \text{`}=\text{'} && (24) \\
\mathit{VisibilityKind} &::= \text{`}\mathit{public}\text{'} \mid \text{`}\mathit{private}\text{'} \mid \text{`}\mathit{protected}\text{'} \mid \text{`}\mathit{package}\text{'} && (25) \\
\mathit{DirectionKind} &::= \text{`}\mathit{in}\text{'} \mid \text{`}\mathit{inout}\text{'} \mid \text{`}\mathit{out}\text{'} \mid \text{`}\mathit{return}\text{'} && (26)
\end{aligned}$$

420 The static and dynamic conditions of each pattern are defined by a *Formula* field. Each formula (Rule 29) can be negative or positive. In addition, each formula can be a *simple*, *quantifier*, or mixed of both simple/quantifier type (Rule 31).

$$\begin{aligned}
\mathit{StaticCondition} &::= \mathit{Formula} && (27) \\
\mathit{DynamicCondition} &::= \mathit{Formula} && (28) \\
\mathit{Formula} &::= [\mathit{UnaryConn}], \mathit{FormulaType} && (29) \\
\mathit{UnaryConn} &::= \text{`}\neg\text{'} && (30) \\
\mathit{FormulaType} &::= \mathit{SimpleFormula} \mid \mathit{QuantFormula} \mid \\
&\quad \text{`('}, \mathit{MixedFormula}, \text{`)'} && (31)
\end{aligned}$$

Each simple formula is defined as an alternation of an ordered-list of a left-
425 hand side operand, comparison operator, and right-hand side operand; a set

inclusion; a set membership; or, a function inclusion. The left-hand side operand can be a classifier, an operation, an attribute, or a parameter name.

$$
\begin{aligned}
\mathit{SimpleFormula} &::= \mathit{LHSOperand}, \mathit{CompOp}, \mathit{RHSOperand} \mid \\
&\quad \mathit{SetInclusion} \mid \mathit{SetMembership} \mid \mathit{FuncInclusion} && (32)\\
\mathit{LHSOperand} &::= \mathit{String} && (33)\\
\mathit{RHSOperand} &::= \mathit{String} \mid \mathit{Boolean} \mid \mathit{Integer} \mid \text{`}*\text{'} && (34)
\end{aligned}
$$

Each quantifier formula (Rule 35) can be defined as an ordered-list of a quantifier symbol (‘∀’ or ‘∃’), quantifier types–which can be a list of variable(s):type–
430 an optional such that formula preceded by a ‘|’, an optional at sign formula preceded by a ‘•’, and the consequent formula preceded by a ‘⇒’ or ‘⇔’ symbol.

$$
\begin{aligned}
\mathit{QuantFormula} &::= \mathit{Quantifier}, \mathit{QTypes}, [\text{`}\mid\text{'}, \mathit{Formula}], \\
&\quad [\text{`}\bullet\text{'}, \mathit{Formula}], \mathit{ConseqConn}, \mathit{Formula} && (35)\\
\mathit{Quantifier} &::= \text{`}\forall\text{'} \mid \text{`}\exists\text{'} && (36)\\
\mathit{QTypes} &::= \mathit{VarType}, [\text{`};\text{'}, \mathit{VarType}]^* && (37)\\
\mathit{VarType} &::= \mathit{id} : \mathit{String}, [\text{`},\text{'}, \mathit{id} : \mathit{String}]^*, \text{`}:\text{'}, \mathit{Type} && (38)\\
\mathit{ConseqConn} &::= \text{`}\Rightarrow\text{'} \mid \text{`}\Leftrightarrow\text{'} && (39)
\end{aligned}
$$

Each mixed formula (Rule 40) is defined as an ordered list of a formula followed by an optional mixed formula which can be a disjunction/conjunction kind.

$$
\begin{aligned}
\mathit{MixedFormula} &::= \mathit{Formula}, \mathit{MixedFormulaRHS} && (40)\\
\mathit{MixedFormulaRHS} &::= [\text{`}\vee\text{'}, \mathit{Formula}]^* \mid [\text{`}\wedge\text{'}, \mathit{Formula}]^* && (41)
\end{aligned}
$$

435 *Pattern Specification Consistency Constraints.* In the set inclusion rule (15) and the set membership rule (16), the types of operands for ‘ ∈ ’ and ‘ ⊆ ’ must be matched with each other, i.e., their right-hand side operator must be super type of the left-hand side one. To avoid confusion in the circumstances like *classes* function which gives us all the classes defined in a class diagram and $\mathbb{P}\,\mathit{Class}$

440 which are all possible classes of design models, we assume $classes \subseteq \mathbb{P}\,Class$. For the sake of conciseness, the basic type rule (19) has included only some of the permitted types of the GEBNF model of UML presented in Section 4.

5.2. An example: the Broker pattern

In this section, to clarify the pattern specification using the presented GEBNF 445 rules, we specify the Broker pattern [9, 24].

The communication between a client and server in a centralized system is simple and straightforward. In a simple scenario, a client sends a request for a service and the server runs and returns the result directly. However, this interaction has many challenges in the distributed systems, different address spaces, 450 protocols, platforms and roughly speaking, in the communicating systems which are heterogeneous in many aspects. One of the popular solution to resolve these problems is applying a broker pattern to design a communication middleware, for example, the OMG Common Object Request Broker Architecture (CORBA) and the Microsoft .Net Remoting [22]. Figure 1 and Figure 2 show the class 455 diagram of the Broker pattern (changed from [24]) and a simple scenario of its sequence diagram, respectively. For the sake of simplicity, we omit intentionally some of the listed operations of the source class diagram here. Also, we assume that the exchanged messages in the sequence diagram are synchronous, i.e., sending a message will block the message sender until it receives the message 460 response. In addition, the *Server* in the sequence diagram scenario is assumed local to the *Client*.

Following, we sketch the Broker pattern specification using GEBNF. The specification is divided into three parts: *components*, *staticConditions*, and *dynamicConditions* corresponding to their counterparts $V(Broker)$, $Pr_s(Broker)$, 465 and $Pr_d(Broker)$ in Definition 1, respectively. To clarify the applied rules in the specification, we commented rule numbers next to the predicates. For the sake of readability, we left-out intentionally the comments of similar predicates.

- **Broker** `//Rule 13`

- **components:**

470 $\{Client, C_Proxy, C_API, Broker, Bridge, S_API, S_Proxy,$
$Server\} \subseteq classes;$ //R 13,R 14,R 15,R 17,R 18,R 2,R 3
$call_server() \in opers(Client);$
//R 13,R 14,R 16,R 17,R 18,R 2,R 3,R 5
$\{send_request(), return()\} \subseteq opers(C_Proxy);$
475 //R 13,R 14,R 15,R 17,R 18,R 2,R 3,R 5
$call_server() \in opers(C_API);$
$\{find_server(), forward_request(), find_client(),$
$forward_response()\} \subseteq opers(Broker);$
$\{forward_message(), transmit_message()\} \subseteq opers(Bridge);$
480 $register_service() \in opers(S_API);$
$\{call_service(), send_response()\} \subseteq opers(S_Proxy);$
$run_service() \in opers(Server);$

- **staticConditions:**

$associates(Client, C_Proxy) \wedge$
485 //R 13,R 27,R 29,R 31,R 32,R 21,R 22
$associates(C_Proxy, Client) \wedge$
$associates(Server, S_Proxy) \wedge associates(S_Proxy, Server) \wedge$
$associates(Broker, Broker) \wedge associates(Broker, Bridge) \wedge$
$associates(Bridge, Broker) \wedge associates(Bridge, Bridge) \wedge$
490 $associates(C_Proxy, Broker) \wedge associates(Broker, C_Proxy) \wedge$
$associates(S_Proxy, Broker) \wedge associates(Broker, S_Proxy) \wedge$
$depends(Client, C_API) \wedge depends(Server, S_API) \wedge$
$composes(Broker, C_API) \wedge composes(Broker, S_API) \wedge$
$isInterface(C_API) \wedge$
495 $isInterface(S_API)$

- **dynamicConditions:**

$\forall\, m : Message \bullet (m \in msgs \wedge$
//R 13,R 28,R 29,R 35,R 40,R 41,R 31,R 32,R 7,R 12,R 16,R 18,R 39

$name(c(name(lifeline(sender(m))))) = Client \wedge$

500 //R 29,R 31,R 32,R 21,R 22,R 12,R 11,R 8,R 9,R 3,R 18

$name(c(name(lifeline(receiver(m))))) = Client \wedge$

$sig(m) = call_server()) \Rightarrow$

//R 29,R 31,R 32,R 21,R 22,R 23,R 24,R 21,R 12,R 5,R 18

$(\exists M_c : Message \bullet (M_c \in msgs \wedge$

505 $sig(M_c) = send_request() \wedge$

$name(c(name(lifeline(sender(M_c))))) = Client \wedge$

$name(c(name(lifeline(receiver(M_c))))) = C_Proxy) \Rightarrow$

$(precedes(m, M_c) \wedge$

$calls(call_server(), send_request())) \wedge$

510 $\exists M_b : Message \bullet (M_b \in msgs \wedge$

$sig(M_b) = forward_request() \wedge$

$name(c(name(lifeline(sender(M_b))))) = C_Proxy \wedge$

$name(c(name(lifeline(receiver(M_b))))) = Broker) \Rightarrow$

$(precedes(M_c, M_b) \wedge$

515 $calls(send_request(), forward_request())) \wedge$

$\exists M_{sp} : Message \bullet (M_{sp} \in msgs \wedge$

$sig(M_{sp}) = call_service() \wedge$

$name(c(name(lifeline(sender(M_{sp}))))) = Broker \wedge$

$name(c(name(lifeline(receiver(M_{sp}))))) = S_Proxy) \Rightarrow$

520 $(precedes(M_b, M_{sp}) \wedge$

$calls(forward_request(), call_service())) \wedge$

$\exists M_s : Message \bullet (M_s \in msgs \wedge$

$sig(M_s) = run_service() \wedge$

$name(c(name(lifeline(sender(M_s))))) = S_Proxy \wedge$

525 $name(c(name(lifeline(receiver(M_s))))) = Server) \Rightarrow$

$(precedes(M_{sp}, M_s) \wedge$

$calls(call_service(), run_service())) \wedge$

$\exists M_{rsp} : Message \bullet (M_{rsp} \in msgs \wedge$

$sig(M_{rsp}) = send_response() \wedge$

530 $name(c(name(lifeline(sender(M_{rsp})))) = Server \wedge$

$name(c(name(lifeline(receiver(M_{rsp})))) = S_Proxy) \Rightarrow$

$(precedes(M_s, M_{rsp}) \wedge$

$calls(run_service(), send_response())) \wedge$

$\exists\, M_{rb} : Message \bullet (M_{rb} \in msgs \wedge$

535 $sig(M_{rb}) = forward_response() \wedge$

$name(c(name(lifeline(sender(M_{rb})))) = S_Proxy \wedge$

$name(c(name(lifeline(receiver(M_{rb})))) = Broker) \Rightarrow$

$(precedes(M_{rsp}, M_{rb}) \wedge$

$calls(send_response(), forward_response())) \wedge$

540 $\exists\, M_{rbb} : Message \bullet (M_{rbb} \in msgs \wedge$

$sig(M_{rbb}) = find_client() \wedge$

$name(c(name(lifeline(sender(M_{rbb})))) = Broker \wedge$

$name(c(name(lifeline(receiver(M_{rbb})))) = Broker) \Rightarrow$

$precedes(M_{rb}, M_{rbb}) \wedge$

545 $\exists\, M_{rcp} : Message \bullet (M_{rcp} \in msgs \wedge$

$sig(M_{rcp}) = return() \wedge$

$name(c(name(lifeline(sender(M_{rcp})))) = Broker \wedge$

$name(c(name(lifeline(receiver(M_{rcp})))) = C_Proxy) \Rightarrow$

$(precedes(M_{rbb}, M_{rcp}) \wedge$

550 $calls(find_client(), return()))$

)

Note that, with the simplicity in mind, we omit intentionally the specification of multiplicities of the relationships' ends in the presented specification. Also, to show the applicability of our presented notation, we specified other patterns
555 of the Broker PL [22, 28] as a case study [25].

5.3. Applying Patterns on a Design Model

Now, we move on to apply and verify a design pattern in practice. A simple question which maybe asked here is: How to verify a given model has applied

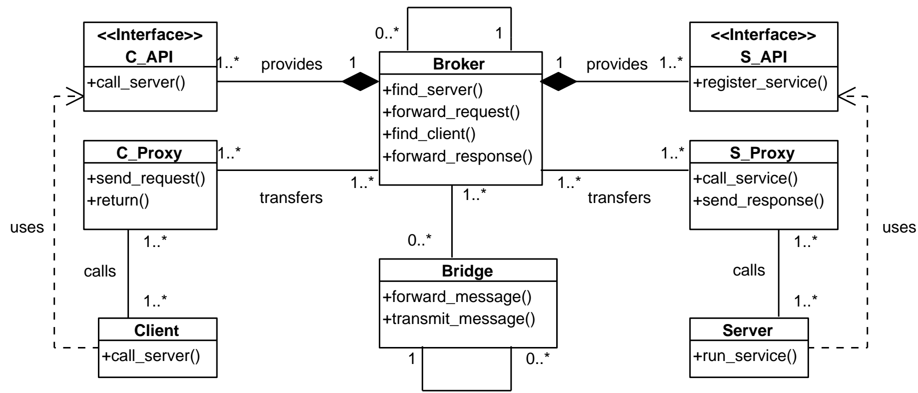

Figure 1: The Broker pattern class diagram (changed from [24])

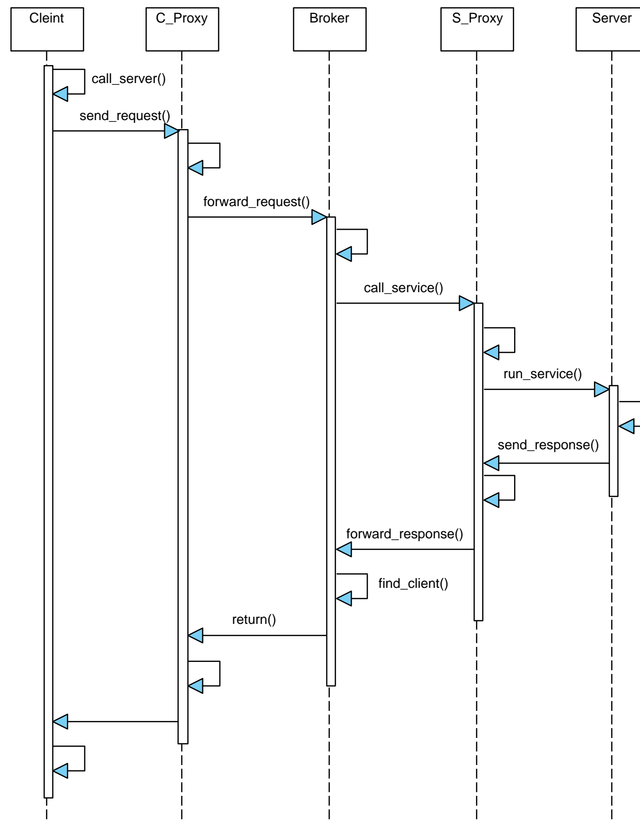

Figure 2: The Broker pattern sequence diagram (changed from [24])

a design pattern correctly? To answer this question, we need to define a design model conformance to a design pattern. In Definition 2, to specify formally that a given model satisfies a predicate, we exploit the abstract syntax of the UML model which has been presented in Section 4.

Definition 2. *(Conformance of a Design Model to a Design Pattern). A given design model m conforms to a pattern P if m satisfies $Spec(P)$ which is denoted by $m \models Spec(P)$. We formalize the satisfaction as follows (the expression, $Vars(P) \leadsto s$, assumed to bind and match some elements of $Vars(P)$ with the elements of set s):*

$$
\begin{aligned}
&\forall\, m : UMLModel;\; P : Pattern \bullet \\
&\quad m \models Spec(P) \Leftrightarrow \\
&\qquad \exists\, s : \mathbb{P}\, Class;\; r : \mathbb{P}\, Relation \bullet s \subseteq classes(cd(m)) \wedge r \subseteq rels(cd(m)) \wedge \\
&\qquad\quad (\forall\, r_i : Relation;\; e : End \mid r_i \in r \wedge e \in ends(r_i) \bullet node(e) \in s) \Rightarrow \\
&\qquad\qquad Pr_s(P)[Vars(P) \leadsto s] \wedge Pr_d(P)[Vars(P) \leadsto s]
\end{aligned}
$$

In other words, a UML model m conforms to a pattern P iff we can find at least an assignment of $Vars(P)$ to some elements of model m such that evaluating $Spec(P)$–i.e., $Pr_s(P) \wedge Pr_d(P)$–in the context of m yields true [21].

Given patterns P, Q; design model m; and constraint c; Table 5 shows a summary of the algebra of design patterns [21].

6. Formalizing a PL

In this section, we will try to give a formal specification for a given PL. Therefore, to base our formal definition, we need a comprehensive and relatively complete informal definition of a typical PL.

PL Informal Definition. There are several informal definitions for PL in literature [5, 9]. Here, we present our accepted PL definition of [9] which we quote and adapt as follows (see [25, 23] for more discussion on a PL informal definition):

Table 5: Summary of the Algebra of Design Patterns (adapted from [21])

Expression	**Description**
$P \preccurlyeq Q$	*Specialization Relation.* P is an special case of Q if $\forall\, m \bullet m \models Spec(P) \Rightarrow m \models Spec(Q)$. In other words, $P \preccurlyeq Q$, if $Spec(P) \Rightarrow Spec(Q)$. Furthermore, $\preccurlyeq$ is a *preorder relation* with **FALSE** and **TRUE** patterns as its bottom and top, respectively. Pattern *TRUE* is the pattern such that $\forall\, m \bullet m \models TRUE$. Pattern *FALSE* is the pattern such that $\nexists m \bullet m \models Spec(FALSE)$. So, $FALSE \preccurlyeq P \preccurlyeq TRUE$.
$P \approx Q$	*Equivalence Relation.* P is equivalent to Q iff $P \preccurlyeq Q \wedge Q \preccurlyeq P$.
$P[c]$	*Restriction Operator.* It is defined formally as (1) $Vars(P[c]) = Vars(P)$, (2) $Spec(P[c]) = (Spec(P) \wedge c)$. In other words, this operator only restricts the predicate part of the design pattern.
$P * Q$	*Composition Operator.* The $P * Q$ is defined formally as (1) $Vars(P * Q) = Vars(P) \uplus Vars(Q)$, and (2) $Spec(P * Q) = Spec(P) \wedge Spec(Q)$ such that by the definitions of the partial and surjective functions $f_1 : Vars(P * Q) \twoheadrightarrow Vars(P)$ and $f_2 : Vars(P * Q) \twoheadrightarrow Vars(Q)$ which map the components of the composition of P and Q (i.e., the disjoint set union of variables declared in patterns P and Q) to the original components of them, respectively, the components of both patterns P and Q satisfying the predicate $\forall\, v_1 : Vars(P), v_2 : Vars(Q) \Rightarrow f_1^{\sim}(v_1) = f_2^{\sim}(v_2)$ will determine those elements of the composition which play the same role in both of patterns P and Q (see Table A.11 for more detail on the used functions). Also, for all patterns p, q, and r, the $*$ is *commutative* and *associative*, i.e., $p * q \approx q * p \wedge (p * q) * r \approx p * (q * r)$.
$P\#(V \bullet c)$	*Extension Operator.* Given a variable set V disjoint from variables declared in P, i.e. $Vars(P) \cap V = \emptyset$; the extension of P to accommodate V as well as satisfying constraint c is defined as follows: (1) $Vars(P\#(V \bullet c)) = Vars(P) \cup V$; (2) $Spec(P\#(V \bullet c)) = Spec(P) \wedge c$.

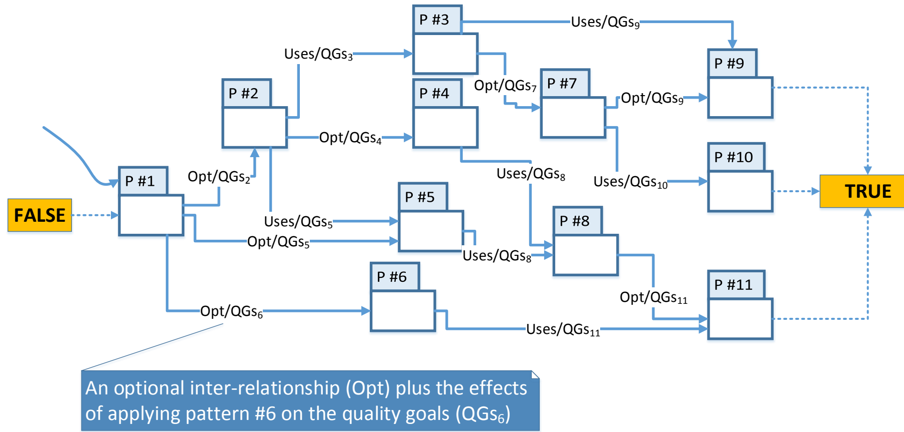

Figure 3: A sample PL which clarifies our presented PL definition

"A network of tightly interwoven patterns that defines a process for systematically resolving a set of related and interdependent software development problems [through satisfying constraint forces of the underlying problem domain or context]." [9]

Figure 3 shows a sample PL with the *optional/uses* patterns' inter-relationships that clarifies our accepted definition. Here, *FALSE* and *TRUE* patterns are trivial patterns which we augment them to the network of a PL by the dotted connectors. Patten $P\#1$ is the *initial* pattern of the PL. The links between patterns specify the relationship and the consequences of applying the latter pattern on the quality goals in the context of the former. For example, in the context of $P\#2$, applying the required pattern $P\#5$ affects on the quality goals which are listed in the set QGs_5.

In the following, based on the accepted informal definition of a PL, a formal definition for a PL is presented in Section 6.1. Section 6.2, compared to the Definition 2, defines the conformance of a design model to a given PL.

6.1. Formal Definition of a PL

Now, we are ready to provide a formal definition for a typical PL. This definition is based on the informal definition which is presented in the previous

section.

595 **Definition 3.** *(Pattern Language). A pattern language named plname, denoted by* L_{plname} *is defined as a quadruple* $\langle P \cup \{FALSE, TRUE\}, R, p_i, \delta \rangle$ *where*

- $P : \mathbb{P}\, Pattern$*, is the set of patterns involved in* L_{plname}*;* $FALSE, TRUE : Pattern$*, are trivial patterns of* L_{plname} *such that* $\forall\, p : Pattern \mid p \in P \bullet FALSE \preccurlyeq p \wedge p \preccurlyeq TRUE$*,*

600
- $R : \mathbb{P}(Pattern \times Relationship \times Pattern)$*, is the set of existing patterns' inter-relationships of the language* L_{plname}*,*
- $p_i : Pattern$*, is the initial pattern of* L_{plname}*, and*
- δ*, the transition function (inspiring from the notion of automata transition function [29]) which for a given pattern from* L_{plname} *returns a sequence*
605 *of patterns which can be applied successively in the context of the given pattern. (Note that* $[Pattern, QGoal]$*, assumed to be predefined given sets– borrowed from the Z notation counterparts [49]–, i.e., patterns and the quality goals of a given design, respectively)*

In the following, the transition function δ is specified in Section 6.1.1. Also,
610 the patterns' inter-relationships, i.e, R is defined in Section 6.1.2. A summary of the used Z algebraic notation [49] in our presented axiomatic definitions are included in AppendixA.

6.1.1. Transition Function: δ

This function is defined using the following auxiliary functions: *succ*, *iseqComp*, *effects*, *conseqs*, and *opt*. The function *succ*–a shorthand for *successor*–is defined as below. This function using the patterns' inter-relationships returns the related patterns of a given pattern.

$$succ : Pattern \nrightarrow \mathbb{P}\, Pattern$$

$$\forall\, p : Pattern \mid p \approx TRUE \bullet succ(p) = \emptyset$$
$$\forall\, p : Pattern;\ prp : Pattern \times Relationship \times Pattern \bullet$$
$$\quad p \not\approx TRUE \wedge prp \in R \wedge prp.1 = p \Rightarrow prp.3 \in succ(p)$$

Applying each pattern to resolve a problem in a context results some con-
615 sequences. To select and apply a pattern from a set of candidate patterns, it is required to compare and filter out unnecessary patterns. Of course, a pattern will be applied in the next step, firstly, if it is one of the next essential candidates, and secondly, in terms of pattern consequences, applying a candidate maximizes the benefits and minimizes the liabilities. We use the function *opt* to select an
620 optimal pattern. To discriminate the pattern application consequences, we use the *conseqs* function.

Given two patterns, p and q, the function *conseqs* returns the consequences of applying q in the context of p. Also, the consequences of applying a pattern are defined as an injective sequence of pairs. Each pair illustrates the effect of
625 applying a pattern on a quality goal. The effect can be positive or negative that assigning its value is based on the designer choice–this idea has been inspired from Zdun's approach [6] which selects applicable candidate patterns based on their effects on the quality goals of a design. To compare the consequences of applying two given candidate patterns, we define the function *iseqComp* (here,
630 the *iseq* is the injective sequence function, see AppendixA for more detail).

$$
\begin{array}{l}
\mathit{iseqComp} : (\mathrm{iseq}(\mathit{QGoal} \times \mathbb{Z}) \times \mathrm{iseq}(\mathit{QGoal} \times \mathbb{Z})) \rightarrow\!\!\!\!\!+ \mathit{Boolean} \\
\hline
\forall X, Y : \mathrm{iseq}(\mathit{QGoal} \times \mathbb{Z}) \bullet \\
\quad (\forall i : 1 \mathrel{..} \#X \mid (X i).1 = (Y i).1 \bullet (X i).2 \geq (Y i).2 \Rightarrow \\
\qquad \mathit{iseqComp}(X, Y) = \mathit{true}) \quad \vee \\
\quad (\exists i : 1 \mathrel{..} \#X \mid (X i).1 = (Y i).1 \bullet (X i).2 < (Y i).2 \Rightarrow \\
\qquad \mathit{iseqComp}(X, Y) = \mathit{false})
\end{array}
$$

Applying each pattern in a given context will have a side-effect (or consequence) which is defined as a free type:

$$\mathit{Effect} ::= \mathit{Benefit} \mid \mathit{Liability}$$

To get the consequences of applying a pattern on the quality goals in the context of a given pattern, we use *effects* and *conseqs* functions (the mappings of these functions must be provided by the PL writer):

$$effects : (Pattern \times QGoal) \rightarrow\!\!\!\!\!\!+ \; Effect$$

$$conseqs : (Pattern \times Pattern) \rightarrow\!\!\!\!\!\!+ \; \mathrm{iseq}(QGoal \times \mathbb{Z})$$

$$\begin{array}{l}
\forall p, q : Pattern \mid p \in P \land q \in P \bullet q \in succ(p) \land q \neq TRUE \Rightarrow \\
\quad (\forall qg : QGoal \bullet effects(q, qg) = Benefit \Rightarrow \\
\quad\quad (\exists pair : QGoal \times \mathbb{Z} \mid pair \in \mathrm{ran}(conseqs(p, q)) \bullet \\
\quad\quad\quad pair.1 = qg \land pair.2 > 0) \\
\quad \lor \\
\quad (\forall qg : QGoal \bullet effects(q, qg) = Liability \Rightarrow \\
\quad\quad (\exists pair : QGoal \times \mathbb{Z} \mid pair \in \mathrm{ran}(conseqs(p, q)) \bullet \\
\quad\quad\quad pair.1 = qg \land pair.2 < 0))
\end{array}$$

To select an optimal pattern from a set of given candidate patterns in a context, we use *opt* function which is defined as below:

$$opt : Pattern \rightarrow\!\!\!\!\!\!+ \; Pattern$$

$$\begin{array}{l}
\forall p : Pattern \mid p \in P \bullet \\
\quad (\exists t : Pattern \bullet \\
\quad\; t \in succ(p) \land \\
\quad\; (\forall q : Pattern \mid q \in succ(p) \land q \neq t \bullet \\
\quad\quad iseqComp(conseqs(p, t), conseqs(p, q)) = true) \\
\quad \Rightarrow opt(p) = t)
\end{array}$$

Now, we come back to the transition function, δ. The aim of this function is to select an optimal path (which maximizes the benefits and minimizes the liabilities in terms of the consequences of the selected patterns) in a step-by-step manner through the network of patterns of the given PL. The selected patterns
635 must satisfy the expected quality goals of the designer. The function executes recursively until it reaches to *TRUE* pattern. It returns the selected optimal candidate patterns as an injective sequence.

Table 6: Literature review on the patterns' inter-relationships (granularity aspect)

Research	PLs	PSs	Patterns' inter-relationships			
			uses	**refines**	**conflicts**	**competes**
Zamani and Butler [5]	✓	✓	✓	✓	✓	✓
Zdun [6]	✓	✓	✓			
Zhu and Bayley [21]			✓	✓	✓	
Buschmann et al. [22]	✓	✓	✓	✓	✓	✓
Porter et al. [27]	✓	✓				
Hakeem et al. [50]	✓	✓				
Noble [51]			✓	✓	✓	
Zimmer [52]			✓	✓	✓	

$$\delta : Pattern \pfun \operatorname{iseq} Pattern$$

$$\begin{array}{l} \forall\, p : Pattern \mid p \in P \bullet \\ \quad (\exists\, q : Pattern \mid q \in P \land q = opt(p) \bullet \\ \qquad q \approx TRUE \Rightarrow \delta(p) = \langle\rangle \\ \qquad\quad \lor \\ \qquad q \not\approx TRUE \Rightarrow \delta(p) = \langle q \rangle \frown \delta(q)) \end{array}$$

6.1.2. Patterns' inter-relationships: R

There are many sources in literature [5, 6, 21, 22, 27, 50, 51, 52] which discuss commonly used patterns' inter-relationships (see Table 6).

In the PL formal definition (Definition 6.1), the relationships among patterns, R has been specified as a set of triples $Pattern \times Relationship \times Pattern$. Here, based on the popular and primary patterns' inter-relationships which have been introduced by Zamani and Butler [5] and Buschmann et al. [22], the *Relationship* set is specified as the following free type definition:

$$Relationship ::= uses \mid refines \mid conflicts \mid competes$$

These relationships are enough to specify the commonly used patterns' inter-relationships in the catalogs like GoF [1] and PLs like PofEAA [11, 23] and

Broker [22, 28].

Now, using the theory of composibility of design patterns in [21] and LePUS
650 formulas regarding the patterns' inter-relationships [44], the formal semantics of patterns' inter-relationships are specified as follows: (Note that the patterns' inter-relationships are considered from their solution point of view not from the problems they address; for example, pattern X *uses* pattern Y means pattern X uses pattern Y in its solution.)

Definition 4. *(Formal Semantics of patterns' inter-relationships). Given two related patterns* p *and* q *of* L_{plname} *as well as the pattern inter-relationship* r*, we define* $Pred(r)$ *as follows that must be satisfied in a design model which applies* p *or* q *or both of them:*

$$\forall\, p, q : Pattern;\ r : Relationship \mid$$

$$p \in P \wedge q \in P \wedge (p, r, q) \in R \wedge q \in succ(p) \bullet$$

$$r = uses \Rightarrow Spec(p) \wedge Spec(q) \wedge$$

$$\exists\, s : Vars(p);\ t : Vars(q) \bullet \forall\, v_p \in s \Rightarrow$$

$$\exists\, v_q \in t \bullet depends(v_p, v_q) \wedge \neg\, depends(v_q, v_p)$$

`// Conformance of a design model to p uses q`

$$\vee$$

$$r = refines \Rightarrow Spec(p) \Rightarrow Spec(q)$$

`// Conformance of a design model to p implies`

`// its conformance to q, i.e., ` $refines \equiv \preccurlyeq$

$$\vee$$

$$r = conflicts \Rightarrow (Spec(p) \wedge Spec(q)) \Rightarrow false$$

`// Conformance of a design model to both of`

`// p and q implies False statement,`

`//i.e., a contradiction`

$$\vee$$

$$r = competes \Rightarrow Spec(p) \vee Spec(q) \vee Spec(p * q)$$

`// Conformance of a design model to`

`// p or q, or a compositions of p and q`

655 *6.2. Conformance of a Design Model to a PL*

Each applied pattern of a given PL will see a sequence of patterns which have been applied previously beginning from the initial pattern p_i of the language. We define formally the *prefix* of a given pattern p, i.e., all the prerequisite patterns of p as following:

660 **Definition 5.** *(Pattern Prefix). Given a pattern p from a PL, the prefix of pattern p is defined as a sequence of tuples* $S = \langle (p_0, r_0, p_1), ..., (p_{k-1}, r_{k-1}, p_k) \rangle, k >= 1$ *such that:*

1. $(head\ S).1 = p_i \wedge (last\ S).3 = p,$ *and*

2. $\forall\, prp : \mathrm{ran}\, S \bullet prp \in R$, *and*

665 3. $\forall\, i : 1..\#S \bullet \#S > 1 \Rightarrow (S\ i).3 \in succ((S\ i).1)$.

The function *prefix* which returns a sequence of patterns representing a given pattern prefix is specified formally as follows:

$$
\begin{array}{|l}
prefix : Pattern \rightarrowtail \mathrm{seq}\, Pattern \\
\hline
\forall\, p : Pattern \mid p \in P \bullet p \approx FALSE \Rightarrow prefix(p) = \langle\rangle \\
\forall\, p : Pattern \mid p \in P \wedge p \not\approx FALSE \bullet \\
\quad (\exists\, prp : Pattern \times Relationship \times Pattern \mid prp \in R \bullet p = prp.3 \Rightarrow \\
\qquad prefix(p) = prefix(prp.1) \frown \langle prp.3 \rangle)
\end{array}
$$

Now, after specifying the semantics of the popular patterns' inter-relationships and the pattern prefix, a design model conformance to a given PL can be specified with the following definition:

670

Definition 6. *(Conformance of a Design Model to a PL). A design model m conforms to a given PL, $L_{plname} = \langle P \cup \{TRUE, FALSE\}, R, p_i, \delta \rangle$ if and only if:*

1. *For any given pattern that m conforms to, it must conform to the pattern prefix too:*

675

$$\forall\, p : Pattern \mid p \in P \bullet m \models Spec(p) \Rightarrow (\forall\, q : Pattern \mid q \in P \bullet q \in prefix(p) \Rightarrow m \models Spec(q)),\ and$$

2. *Given two patterns, p and q, such that pattern q is the immediate successor of p regarding the relationship of r in the given PL, the semantic of relationship r, i.e., $Pred(r)$ must be preserved in m according to the Definition 4:*

680

$$\forall\, p, q : Pattern;\ r : Relationship \mid p \in P \wedge q \in P \wedge (p, r, q) \in R \wedge q \in succ(p) \bullet m \models (Spec(p) \vee Spec(q)) \Rightarrow m \models Pred(r).$$

7. Case Study: the Broker PL

685 To illustrate the applicability of our PL formal definition, we specified formally the Broker PL [22, 53]. Figure 4 depicts the constituent patterns of the language and their relationships. For the sake of conciseness, we present the patterns, the resolved problems, and the consequences (in the forms of benefits and liabilities) of applying patterns in Table 7 and Table 8.

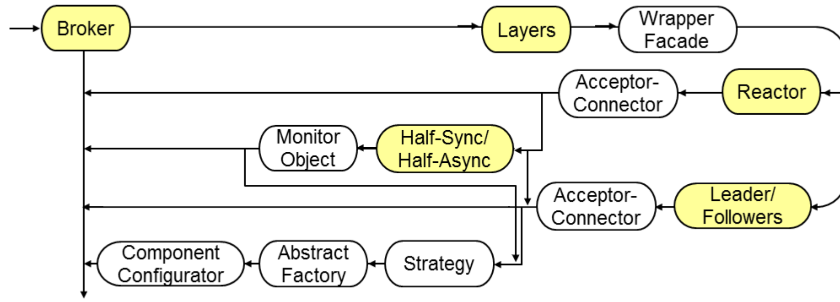

Figure 4: A sample PL to construct a broker in the concurrent and networked systems (adapted from [22])

$$
\begin{aligned}
P &= \{P_B, P_L, P_{WF}, P_R, P_{AC}, P_{LF}, P_{HSA}, P_{MO}, P_S, P_{AF}, P_{CC}\} \\
R &= \{(P_B, uses, P_L), (P_L, uses, P_{WF}), (P_R, competes, P_{LF}), \\
&\quad (P_{WF}, uses, P_R), (P_{WF}, uses, P_{LF}), (P_R, uses, P_{AC}), \\
&\quad (P_{LF}, uses, P_{AC}), (P_{LF}, competes, P_{HSA}), (P_{AC}, uses, P_{HSA}), \\
&\quad (P_{AC}, uses, P_S), (P_{HSA}, uses, P_{MO}), (P_{MO}, uses, P_S), \\
&\quad (P_S, uses, P_{AF}), (P_{AF}, uses, P_{CC})\}
\end{aligned}
$$

Table 7: Patterns of the Broker PL and the problems they resolve (adapted from [28, 9, 54])

Pattern [ID]	**Problem**
Broker[P_B]	Defining the broker's base-line architecture
Layers[P_L]	Structuring broker internal design to enable reuse & clean separation of concerns
WrapperFacade[P_{WF}]	Encapsulating low-level system functions to enhance portability
Reactor[P_R]	Demultiplexing the broker core events efficiently
Acceptor/Connector[P_{AC}]	Managing broker connections efficiently
Leader/Followers[P_{LF}]	Providing an efficient concurrency model and enhancing predictability
HalfSync/HalfAsync[P_{HSA}]	Enhancing broker scalability by processing requests concurrently
MonitorOpject[P_{MO}]	Efficiently synchronize the Half-Sync/Async request queue
Strategy[P_S]	Interchanging internal broker mechanisms transparently
AbstractFactory[P_{AF}]	Consolidating broker mechanisms into groups of semantically compatible strategies
ComponentConfigurator[P_{CC}]	Configuring dynamically the consolidated broker strategies

Table 8: The Broker PL's patterns and their consequences, i.e., the benefits and liabilities (adapted from [9, 28, 54])

ID	Benefits	Liabilities
P_B	{simplicity, modularity, encapsulation}	$\emptyset$
P_L	{encapsulation}	{performance}
P_{WF}	{portability, testability, stability, encapsulation}	$\emptyset$
P_R	{flexibility, performance, encapsulation, transparency}	$\emptyset$
P_{AC}	{encapsulation}	{complexity}
P_{LF}	{predictability}	{applicability}
P_{HSA}	{simplicity, performance, scalability}	$\emptyset$
P_{MO}	{simplicity, availability}	{deadlock}
P_S	{extendibility}	{performance}
P_{AF}	{low coupling, flexibility, encapsulation}	$\emptyset$
P_{CC}	{flexibility, availability}	$\emptyset$

$$\begin{aligned} p_i &= P_B \\ QGoal &::= \textit{simplicity} \mid \textit{modularity} \mid \textit{encapsulation} \mid \textit{portability} \mid \\ &\quad\ \textit{testability} \mid \textit{stability} \mid \textit{flexibility} \mid \textit{performance} \mid \textit{transparency} \mid \\ &\quad\ \textit{predictability} \mid \textit{applicability} \mid \textit{scalability} \mid \textit{availability} \mid \\ &\quad\ \textit{deadlock} \mid \textit{extendibility} \mid \textit{low_coupling} \mid \textit{complexity} \end{aligned}$$

690 To characterize the transition function, δ, we must define the *succ* function first. Table 9 displays the mappings of this function.

At the second step, we present the *conseqs* function mappings. Given two patterns, p and q, this function returns the consequences of applying q in the context of p. Table 10 displays the mappings of *conseqs* function.

695 Now, using the mappings of the *conseqs* function, we can run the δ function to extract required patterns in the form of PSs which will guide us on designing

Table 9: Mappings of *succ* function

Input	**Output**
$FALSE$	$\{P_B\}$
P_B	$\{P_L\}$
P_L	$\{P_{WF}\}$
P_{WF}	$\{P_R, P_{LF}\}$
P_R	$\{P_{AC}\}$
P_{LF}	$\{P_{AC}\}$
P_{AC}	$\{TRUE, P_{HSA}, P_S\}$
P_{HSA}	$\{P_L\}$
P_L	$\{P_{MO}\}$
P_{MO}	$\{TRUE, P_S\}$
P_S	$\{P_{AF}\}$
P_{AF}	$\{P_{CC}\}$
P_{CC}	$\{TRUE\}$

a Broker architecture. Thus, for the sake of conciseness and simplicity of extracting candidate patterns by following δ, we do not rewrite the function body here.

700 Here, we present our findings in the presented case study. For the sake of conciseness, we have documented the full specifications of patterns in [25]:

The incomplete informal definition of patterns: One of the main obstacles in the formal specification of patterns is the lack of their informal definition in language and illustration. For example, despite the applicability of 705 the Broker PL in some projects like ADAPTIVE Communication Environment (ACE) [10], there are ambiguities in terms of declaring the pattern participants, for example, the Broker pattern and its representing class and sequence diagrams.

The pattern variants and adequacy of our specifications: Inherently, some pat-710 terns have variations on their components. For example, we can see these

Table 10: Mappings of *conseqs* function

Input	**Output (without assigning numeric values)**	
	Benefit	**Liability**
$FALSE, P_B$	{simplicity, modularity, encapsulation}	$\emptyset$
P_B, P_L	{encapsulation}	{performance}
P_L, P_{WF}	{portability, testability, stability, encapsulation}	$\emptyset$
P_R, P_{LF}	{predictability}	{applicability}
P_{WF}, P_R	{flexibility, performance, encapsulation, transparency}	$\emptyset$
P_{WF}, P_{LF}	{predictability}	{applicability}
P_R, P_{AC}	{encapsulation}	{complexity}
P_{LF}, P_{AC}	{encapsulation}	{complexity}
P_{LF}, P_{HSA}	{simplicity, performance, scalability}	$\emptyset$
P_{AC}, P_{HSA}	{simplicity, performance, scalability}	$\emptyset$
P_{AC}, P_S	{extendibility}	{performance}
P_{HSA}, P_L	{encapsulation}	{performance}
P_L, P_{MO}	{simplicity, availability}	{deadlock}
P_{MO}, P_S	{extendibility}	{performance}
P_{MO}, P_{AF}	{low_coupling, flexibility, encapsulation}	$\emptyset$
P_{AF}, P_{CC}	{flexibility, availability}	$\emptyset$

variations on the number of concrete states in the State pattern, the number of broker components in the Broker pattern, and the number of layers in the Layers pattern, to name a few. However, the presented formalism extracted from the appropriate quantifier formulas by iterating over each pattern's components, makes it possible to resolve the mentioned problems of the pattern variants (for example, see the Layers pattern specification in [25]). As a result, the mentioned forms of the pattern variants do not restrict the scope of the pattern specifications and their correctness.

The expressiveness of rules representing patterns: Because of the inherent concurrent nature that exists in the distributed systems, representing the state diagram of patterns can also simplify the system state understanding and formalizing as well. For example, although we formalized the Leader/Followers pattern only by using the rules of its class and sequence diagrams, formalizing by augmenting the state diagrams would be more straightforward and expressive.

The readability of pattern specifications: Like other formal notations, the presented specifications promote comprehension and understandability of design models. However, the underlying applied formal basics requires designer experience on the formal notations like Z and the first-order logic predicates [49].

8. Discussion

As stated in Section 2, only Zhu and Bayley [21] have relatively presented a complete formalism for composing design patterns in general, and the GoF design patterns [1] in particular. Now, let's come back to the earlier raised question: *Is it possible to define a PL by applying the algebra of design patterns which is presented in Zhu and Bayley [21]?*

From our point of view and regarding the presented PL formal definition, the answer to this question is "no"! Referring back to the presented PL definition,

a PL is more than just composing some related patterns. Note that we must consider the "consequences" of applying each selected pattern, one-by-one, on resolving a design problem as a whole. Pattern composition does not consider the pattern consequences at all [5, 45].

In addition, as stated in [21], the $*$ operator is the primary operator which is used to compose patterns to each other in the pattern expressions. This operator is a *commutative* and an *associative* operator (see Table 5). As a result, the pattern expression resulting from the application of the $*$ operator must be commutative and associative which contradicts with any pattern application ordering definition. Thus, because of the importance of ordering on applying patterns regarding patterns' inter-relationships, we believe that it is impossible to express any PL through compositions [17, 21].

For example, in the Broker PL [22, 28], the *Broker* pattern **uses** the *Layers* pattern to provide simplicity, modularity, and encapsulation. This relationship is not commutative. Therefore, it cannot be expressed by using only the $*$ operator. As a result, the Broker PL is a counterexample to the "yes" answer of the above question.

To summarize, we can compare the proposed formalism of patterns and PLs with the pattern composition in [21] as follows:

The scale of problem solving To the best of our knowledge, the pattern composition approach, by exploiting from the *commutative* and *associative* relation *, solves problems in small cases similar to the Request Handling Framework which applies a few patterns with a limited patterns' inter-relationships. In fact, there is no limit on the number of patterns and their inter-relationships in our proposed formalism.

Pattern selection and recommendation In the pattern composition approach, this is the designer who decides to choose an appropriate ordering of patterns. Undoubtedly, this decision-making process without considering the forces of the current context will bring unavoidable contradictions and inconsistencies to the design. In the proposed formalism, based on the forces

of the current context and the applied patterns, if any, the appropriate
770 patterns, in the form of PSs, are recommended to the designer.

9. Tool support

After revising and extending the GEBNF grammar of [3] which has been presented in Section 3, it is required to provide a tool for validating automatically the applied grammar in practice. With some modification to the presented
775 grammar, we utilized the Xtext tool [55, 56] and generated the GEBNF parser which is introduced in Section 9.1. Also, to write and validate automatically the formal model of patterns conforming the presented pattern specification model in Section 5, we developed a parser named Pattern Specification Language (PSL) parser which is introduced in Section 9.2.

780 *9.1. The GEBNF model parser*

Since manually validating the revised and extended GEBNF models, especially the specification of production rules regarding the abstract syntax of UML class and sequence diagrams (Section 3), is time-consuming and an error-prone process, we developed an eclipse-based plug-in editor to write and validate the
785 applied GEBNF grammar models using Xtext. To show the applicability of our generated plug-in, we validated our presented abstract syntax of UML class and sequence diagrams along with the pattern specification model. Using the generated parser, we found and corrected numerous typing errors of the earlier presented grammar's production rules (see [25, 30] for more detail).

790 *9.2. The PSL parser*

To present a parser tool for writing and validating the specification of design patterns, we developed an Xtext-based plug-in parser editor named PSL parser to write and validate the specification of design patterns in the presented formalism (Section 5.1). To show the applicability of the generated parser plug-in, we
795 revised and validated our presented patterns of the Broker PL in [25] as a case study. Utilizing the presented PSL parser tool, we found and corrected several

typing and logical errors on the earlier versions of the presented specifications and the pattern specification model as well (see [25, 30, 31] for more detail).

10. Conclusion

Because of the rapid growth of patterns in number, as well as the inherent patterns' inter-relationships, formalizing patterns and PLs is needed to facilitate the transfer of experts' knowledge to novices and to develop support tools. Therefore, we revised the GEBNF notation of Bayley and Zhu [3] to provide more abstract and flexible rule definition. Then, we presented the abstract syntax definition of UML, i.e., the class and sequence diagrams (which are enough to model the structural and behavioral features of popular patterns like the GoF design patterns [1] and the architectural patterns of the Broker PL [22, 28]). Inspired from the notion of Bayley and Zhu [3], we extracted the induced functions from the obtained GEBNF production rules.

Also, to pave the way for the formal specification of patterns, we defined other auxiliary functions (for more information on the complete list of functions refer to [25]). Inspired from Bayley and Zhu [3], we presented the GEBNF notation of a pattern specification scheme. Furthermore, in order to show the applicability of the proposed notation, we formalized and presented all of the 11 patterns of the Broker PL as a case study (for the sake of conciseness, we presented the specification of all patterns other than the *Broker* pattern in [25]).

To provide a foundation on formalizing a given PL, after a literature review on the commonly used patterns' inter-relationships, we presented a formal semantic for each of the presented relationships, i.e., *uses*, *refines*, *competes*, and *conflicts*. Inspired from the idea of systematic pattern selection approach of Zdun [6] and the informal PL definition of Buschmann et al. [22] and Zamani and Butler [5], we presented a formal definition for a given PL. In addition, to illustrate the applicability of our presented PL formalism, we presented a formal model of the Broker PL as a case study. Note that we used the *Z/EVES* tool

825 suite[1] [57] to type check the axiomatic definitions of defined functions.

In a simple statement, we presented a formal metamodel for patterns and PLs. For future work, we intend to transform this metamodel to an executable framework. The framework will provide tool support to model and verify the application of PLs on the design models.

830 AppendixA. The Z algebraic notation

Table A.11 illustrates a summary of the used Z algebraic notation in our presented axiomatic definitions. For more detail, see [49].

References

[1] E. Gamma, R. Helm, R. Johnson, J. Vlissides, Design Patterns: Elements 835 of Reusable Object-Oriented Software, Addison-Wesley Professional, 1st edn., 1994.

[2] D. C. Schmidt, M. Fayad, R. E. Johnson, Software patterns, Communications of the ACM 39 (10) (1996) 37–39.

[3] I. Bayley, H. Zhu, Formal specification of the variants and behavioural 840 features of design patterns, Journal of Systems and Software 83 (2) (2010) 209–221.

[4] A. Rouhi, B. Zamani, Design Patterns: Current Challenges, Trends, and Research Directions, Tech. Rep. UI-SE-MDSERG-2015-02, Model-Driven Software Engineering Research Group, University of Isfahan, Isfahan, 845 Iran, URL `http://www.bahmanzamani.com/TR/UI-SE-MDSERG-2015-02.pdf`, 2015.

[5] B. Zamani, G. Butler, Pattern Language Verification in Model Driven Design, Information Sciences 237 (2013) 343–355.

[1] http://czt.sourceforge.net/eclipse/zeves/

Table A.11: Z algebraic notation, X and Y are sets of elements here

Notation	Description
$X \leftrightarrow Y == \mathbb{P}(X \times Y)$	The set of all relations between X and Y, i.e, the power set of pairs which are drawn from the set of X and Y, respectively.
$f : X \nrightarrow Y$	f is a partial function which maps some of the elements of the domain X to the range Y.
$f : X \twoheadrightarrow\!\!\!\!\!\!+\; Y$	f is a partial and surjective function which maps some of the elements of the domain X to the complete range of Y, i.e., $\operatorname{ran} f = Y$.
$f : X \rightarrowtail\!\!\!\!\!\!+\; Y$	f is a partial injective function.
$f^{\sim}$	The inverse of function f.
$f : X \rightarrowtail Y$	f is a total injective function.
$\operatorname{dom} f$	The domain of the function f, i.e., $\operatorname{dom} f = \{x : X;\ y : Y \mid x \mapsto y \in f \bullet x\}$.
$\operatorname{ran} f$	The range of the function f, i.e., $\operatorname{ran} f = \{x : X;\ y : Y \mid x \mapsto y \in f \bullet y\}$.
$f : X \rightarrow Y$	f is a total function which maps all of the elements of the domain X to the range Y, i.e, $\operatorname{dom} f = X$.
$\operatorname{seq} X$	A finite sequence of elements drawn from the set X, i.e., $\operatorname{seq} X == \{s : \mathbb{N} \twoheadrightarrow\!\!\!\!\!\!+\; X \mid \exists\, n : \mathbb{N} \bullet \operatorname{dom} s = 1 \mathinner{..} n\}$.
$\operatorname{iseq} X$	A finite injective sequence of elements drawn from the set X, i.e., $\operatorname{iseq} X == \{s : seqX \mid s \in \mathbb{N} \rightarrowtail X\}$.
$s \frown t$	The concatenation of sequences s and t. For example, for $s = \langle 1, 2\rangle$, and $t = \langle 5\rangle$, $s \frown t = \langle 1, 2, 5\rangle$.
$\#s$	The number of elements in a finite set/sequence s , i.e., the set/sequence size.

[6] U. Zdun, Systematic pattern selection using pattern language grammars and design space analysis, Software: Practice and Experience 37 (9) (2007) 983–1016.

[7] P. Hegedűs, D. Bán, R. Ferenc, T. Gyimóthy, Myth or reality? analyzing the effect of design patterns on software maintainability, in: Computer Applications for Software Engineering, Disaster Recovery, and Business Continuity, Springer, 138–145, 2012.

[8] B. Huston, The effects of design pattern application on metric scores, Journal of Systems and Software 58 (3) (2001) 261–269.

[9] F. Buschmann, K. Henney, D. C. Schmidt, Pattern-Oriented Software Architecture: On Patterns and Pattern Languages, vol. 5, John Wiley & Sons, 2007.

[10] D. C. Schmidt, C. Cleeland, Applying patterns to develop an extensible ORB middleware, Communications Magazine, IEEE 37 (4) (1999) 54–63.

[11] M. Fowler, Patterns of enterprise application architecture, Addison-Wesley Longman Publishing Co., Inc., 2002.

[12] M. Riaz, T. Breaux, L. Williams, How have we evaluated software pattern application? A systematic mapping study of research design practices, Information and Software Technology 65 (2015) 14–38.

[13] S. Dey, Formal specification of structural and behavioral aspects of design patterns., Journal of Object Technology 9 (6) (2010) 99–126.

[14] R. B. France, D.-K. Kim, S. Ghosh, E. Song, A UML-based pattern specification technique, Software Engineering, IEEE Transactions on 30 (3) (2004) 193–206.

[15] B. Hamid, C. Percebois, A Modeling and Formal Approach for the Precise Specification of Security Patterns, in: Engineering Secure Software and Systems, Springer, 95–112, 2014.

[16] S.-K. Kim, D. Carrington, A formalism to describe design patterns based on role concepts, Formal aspects of computing 21 (5) (2009) 397–420.

[17] I. Bayley, H. Zhu, A formal language for the expression of pattern compositions, International Journal on Advances in Software 4 (3 and 4) (2012)
880 354–366.

[18] P. Bottoni, E. Guerra, J. de Lara, A language-independent and formal approach to pattern-based modelling with support for composition and analysis, Information and Software Technology 52 (8) (2010) 821–844.

[19] J. Dong, P. S. Alencar, D. D. Cowan, S. Yang, Composing pattern-based
885 components and verifying correctness, Journal of Systems and Software 80 (11) (2007) 1755–1769.

[20] T. Taibi, D. C. L. Ngo, Formal specification of design pattern combination using BPSL, Information and Software Technology 45 (3) (2003) 157–170.

[21] H. Zhu, I. Bayley, An algebra of design patterns, ACM Transactions on
890 Software Engineering and Methodology (TOSEM) 22 (3) (2013) 23.

[22] F. Buschmann, K. Henney, D. C. Schmidt, Pattern-Oriented Software Architecture: A Pattern Language for Distributed Computing, vol. 4, John Wiley & Sons, 2007.

[23] B. Zamani, On verifying the use of a pattern language in model driven
895 design, Dissertation, Concordia University, 2009.

[24] F. Bushmann, R. Meunier, H. Rohnert, P. Sommerlad, M. Stal, Pattern-oriented software architecture: A system of patterns, vol. 1, John Wiley & Sons, 1996.

[25] A. Rouhi, B. Zamani, Formalizing Patterns and Pattern Languages: A
900 Case Study Approach, Tech. Rep. UI-SE-MDSERG-2015-01, Model-Driven Software Engineering Research Group, University of Isfahan, Isfahan,

Iran, URL http://www.bahmanzamani.com/TR/UI-SE-MDSERG-2015-01.pdf, 2015.

[26] S. R. Kodituwakku, P. Bertok, Pattern categories: a mathematical approach for organizing design patterns, in: Proceedings of the 2002 conference on Pattern languages of programs-Volume 13, Australian Computer Society, Inc., 63–73, 2003.

[27] R. Porter, J. O. Coplien, T. Winn, Sequences as a basis for pattern language composition, Science of Computer Programming 56 (1) (2005) 231–249.

[28] D. C. Schmidt, M. Stal, H. Rohnert, F. Buschmann, Pattern-Oriented Software Architecture, Patterns for Concurrent and Networked Objects, vol. 2, John Wiley & Sons, 2000.

[29] P. Linz, An introduction to formal languages and automata, Jones & Bartlett Publishers, 2011.

[30] A. Rouhi, B. Zamani, An Xtext Generated Parser to Validate the Revised and Extended GEBNF Applications, Tech. Rep. UI-SE-MDSERG-2015-03, Model-Driven Software Engineering Research Group, University of Isfahan, Isfahan, Iran, URL http://www.bahmanzamani.com/TR/UI-SE-MDSERG-2015-03.pdf, 2015.

[31] A. Rouhi, B. Zamani, The Xtext Generated Parser for Specifying Design Patterns: The PSL Editor, Tech. Rep. UI-SE-MDSERG-2016-05, Model-Driven Software Engineering Research Group, University of Isfahan, Isfahan, Iran, URL http://www.bahmanzamani.com/TR/UI-SE-MDSERG-2016-05.pdf, 2016.

[32] J. Nicholson, A. H. Eden, E. Gasparis, R. Kazman, Automated verification of design patterns: A case study, Science of Computer Programming 80 (2014) 211–222.

[33] A. Blewitt, Spine: Language for Pattern Verification, Design Pattern Formalization Techniques. IGI Global (2007) 109–122.

930 [34] J. Dong, S. Yang, Visualizing design patterns with a UML profile, in: Human Centric Computing Languages and Environments, 2003. Proceedings. 2003 IEEE Symposium on, IEEE, 123–125, 2003.

[35] E. Gasparis, Design Patterns Formalization Techniques, chap. LePUS: A formal language for modeling design patterns, IGI Publishing, Hershey,
935 357–372, 2007.

[36] J. Dong, T. Peng, Y. Zhao, Automated verification of security pattern compositions, Information and Software Technology 52 (3) (2010) 274–295.

[37] A. H. Eden, E. Gasparis, J. Nicholson, R. Kazman, Modeling and visualizing object-oriented programs with Codecharts, Formal Methods in System
940 Design 43 (1) (2013) 1–28.

[38] M. E. Elaasar, An Approach to Design Pattern and Anti-Pattern Detection in MOF-Based Modeling Languages, Dissertation, Carleton University, 2012.

[39] I. Kurtev, State of the art of QVT: A model transformation language stan-
945 dard, in: Applications of graph transformations with industrial relevance, Springer, 377–393, 2008.

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

[41] A. H. Eden, Codecharts: Roadmaps and blueprints for object-oriented pro-
950 grams, John Wiley & Sons, 2011.

[42] I. Bayley, H. Zhu, On the composition of design patterns, in: Quality Software, 2008. QSIC'08. The Eighth International Conference on, IEEE, 27–36, 2008.

[43] I. Bayley, H. Zhu, Specifying behavioural features of design patterns in first
955 order logic, in: Computer Software and Applications, 2008. COMPSAC'08. 32nd Annual IEEE International, IEEE, 203–210, 2008.

[44] A. H. Eden, J. Y. Gil, Y. Hirshfeld, A. Yehudai, Towards a Mathematical Foundation for Design Patterns, Tech. Rep. 1999-004, Department of Information Technology, Uppsala University, 1999.

960 [45] I. Bayley, H. Zhu, Formalising design patterns in predicate logic, in: Software Engineering and Formal Methods, 2007. SEFM 2007. Fifth IEEE International Conference on, IEEE, 25–36, 2007.

[46] H. Zhu, An institution theory of formal meta-modelling in graphically extended BNF, Frontiers of Computer Science 6 (1) (2012) 40–56.

965 [47] O. M. Group, OMG Unified Modeling Language (OMG UML), Superstructure. Version 2.4.1 document 2011-08-06 URL `http://www.omg.org/spec/UML/2.4.1/Superstructure`.

[48] R. Brachman, H. Levesque, Knowledge representation and reasoning, Elsevier, 2004.

970 [49] J. Woodcock, J. Davies, Using Z: specification, refinement, and proof, Prentice-Hall, Inc., 1996.

[50] H. Hakeem, H. Guan, H. Yang, A Framework of Patterns Applicability in Software Development, in: Computer Software and Applications Conference Workshops (COMPSACW), 2014 IEEE 38th International, IEEE,
975 486–491, 2014.

[51] J. Noble, Classifying relationships between object-oriented design patterns, in: Software Engineering Conference, 1998. Proceedings. 1998 Australian, IEEE, 98–107, 1998.

[52] W. Zimmer, Relationships between design patterns, Pattern languages of
980 program design 1 (1995) 345–364.

[53] D. C. Schmidt, C. O'Ryan, M. Kircher, I. Pyarali, F. Buschmann, Leader/Followers, in: University of Washington. http://www.cs.wustl.edu/˜schmidt/PDF/lf.pdf, Citeseer, 2000.

[54] D. C. Schmidt, Overview of Pattern Relationships, URL https://class.coursera.org/posa-001/, 2013.

[55] M. Eysholdt, H. Behrens, Xtext: implement your language faster than the quick and dirty way, in: Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion, ACM, 307–309, 2010.

[56] A. Bergmayr, M. Wimmer, Generating Metamodels from Grammars by Chaining Translational and By-Example Techniques., in: MDEBE@ MoDELS, 22–31, 2013.

[57] M. Saaltink, The Z/EVES system, in: ZUM'97: The Z Formal Specification Notation, Springer, 72–85, 1997.