Available online at www.sciencedirect.com

ScienceDirect

Electronic Notes in Theoretical Computer Science 219 (2008) 51–66

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

A Rule-based Method to Match Software Patterns Against UML Models

D. Ballis[a] A. Baruzzo[a] M. Comini[a]

[a] *Dipartimento di Matematica e Informatica (DIMI), University of Udine,*
Via delle Scienze 206, 33100 Udine, Italy.

Abstract

In a UML model, different aspects of a system are covered by different types of diagrams and this bears the risk that an overall system specification becomes barely tractable by the designer. When the model grows, it is likely that the architectural integrity will be compromised by extensions and bug-fixing operations. Hence, it is important to provide means to help designers to search in big models for particular instances of some variable schema of UML models (design patterns) they construct. This can help them both to find potential problems in the architecture design and to ensure that intended architectural choices had not been broken by mistake. In this paper we propose a rule-based method to find matches of design patterns into a UML model. The method is general enough to tackle most patterns and antipatterns.

Keywords: Rule-based domain specific language, Pattern Matching, UML Design Patterns, UML formal specification

---

1 Introduction

Maintenance is recognized as the most expensive activity of the software development process. Numerous techniques and processes have been proposed to ease this task. One of the most influential proposals are design patterns [8], a collection of elegant and reusable solutions to recurring design problems. Design patterns have been quickly adopted by the object-oriented community because they ease designing, understanding, and re-engineering software.

As a complement to "good" solutions, proposed by design patterns, several authors formalized typical design defects (e.g. antipatterns, code smells) that hinder maintenance by decreasing software quality. The Blob [5] (or God class) and (unnecessary) circular dependencies are typical examples of such flaws that cause large object-oriented programs to be expensive to maintain.

However, all these proposals, at the boundary of programming languages and design models, suffer from a lack of formalism. For this reason, their application remains empirical and manually performed (which is certainly tedious and error prone). We believe that this task should be automated or at least, be assisted.

doi:10.1016/j.entcs.2008.10.034

In particular one of the most desirables automation is an a posteriori *detection* of design patterns and antipatterns, in order to help designers to focus their attention only to specific components of a big project, which probably they could not easily spot by themselves.

In this area, there have been lots of proposals. Without pretending to be exhaustive, we could categorize them as follows.

Exact (canonicals) pattern detection (like [14,3]) which provides tools that aim at reconstructing the presence of some of the most common design patterns from the code. The patterns which are identified are hard-coded in the tools. This unfortunately has a quite limited pragmatic applicability, as plenty of small variations (still provenly solid) of recognizable patterns are present in real systems, and these tools would not find them. Moreover, the designer might desire to expressly find his customized "variations over the theme".

Exact antipattern detection (like [10]) aim instead to detect antipatterns. Also in this case only predefined antipatterns can be detected.

Approximate pattern detection tries to encompass the rigidity of exact pattern detection, with approximate matches. For example [12] does a sort of approximated graph matching[1] with suitable similarity measures, while [9] represents patterns and models as strings and then does string matching. Instead [2,1] defines patterns at the meta-model level and find matches by solving Constraint Satisfaction Problems.

However, since algorithms cannot distinguish the meaning of user customizations, the number of presumed instances of patterns can be quite huge (as tables in [9] show clearly), sensibly reducing the usefulness of the results [2].

Pattern description languages approaches like [7,6] instead introduce a (graphical) formal language (LePUS) which provides a much higher expressive power. Indeed LePUS is not an *ad hoc* collection of loosely related concepts but instead originates from an insight on a small number of basic building blocks that are ubiquitous in object-oriented design. One can easily define LePUS descriptions of common design patterns, such as Proxy, Visitor, and Composite [8]. With this approach the designer can easily redefine the descriptions of canonical patterns to look for his customizations, or define new ones from scratch, or either simply look for arbitrary compositions of other patterns at the same time.

We prefer this approach. However, LePUS is not powerful enough to express descriptions of several relevant properties, amongst all (for example) circular dependencies and, in general, antipatterns [5], which are essential to identify parts of a complex model which do not conform to the expected design properties. Furthermore the proposed prototype works only on code and not directly on UML models.

---

[1] after having reduced the model to limit complexity of the algorithm.

[2] This is probably due to the fact that design patterns are based on a very limited set of simple structures (typically one or two hierarchies and a bunch of classes connected together with some relations such as dependencies or associations). Hence, it is very likely that these simple structures can be very similar to substructures of real-world systems which are not instances of patterns.

In this paper we propose a general language that can express *at the same time* patterns and antipatterns (which we will call simply patterns), plus a rule-based matching algorithm to find all instances of a pattern in the graph which underlies the designers' diagrams. Moreover, whenever code is also provided, matching information can be (if needed) extracted from the code.

The paper is structured as follows. In Section 2 we present our representation of class diagrams. In Section 3 we introduce a language that can express either patterns and antipatterns. In Section 4 we provide the semantics of patterns in terms of suitable sub-graphs of the class diagram graph (i.e., the merge of all class diagrams). In Section 5 we describe a matching algorithm obtained from the semantic description that efficiently finds all instances of a pattern.

2 Class diagrams

Throughout this paper, we formalize the merge of all the class diagrams of an UML model as a graph-shaped structure over a set C of classes and a set A of relations (labeled arrows) between classes. We will often call it *the* class diagram. For a thorough explanation of UML diagrams see [11,4].

A formal description of our representation of class diagrams is provided below.

2.1 Class diagram representation

Let us consider two infinite sets, namely $\mathcal{CN}$and $\mathcal{MN}$, which respectively represent the set of *class names* and the set of *method names*. Given a set of basic types $\mathcal{B}$ (including `int`, `bool`, `char`, `float`, `double` *etc.*), we denote a method *signature* by notation $\mathtt{s}_1, \ldots, \mathtt{s}_n \rightarrow \mathtt{s}$, where $\mathtt{s}_1, \ldots, \mathtt{s}_n, \mathtt{s} \in \mathcal{B} \cup \mathcal{CN}$. A *method* $m : \mathtt{s}_1, \ldots, \mathtt{s}_n \rightarrow \mathtt{s}$ consists of a method name m and a method signature $\mathtt{s}_1, \ldots, \mathtt{s}_n \rightarrow \mathtt{s}$. For what concerns our purpose attributes can be considered as a degenerate case of a method which has no formal parameters. Hence, the set of all the methods (and attributes) we can build over method names and method signatures is defined as follows:

$$\mathit{Methods} := \{m : \mathtt{s}_1, \ldots, \mathtt{s}_n \rightarrow \mathtt{s} \mid n \geq 0, m \in \mathcal{MN}, \mathtt{s}_1, \ldots, \mathtt{s}_n, \mathtt{s} \in \mathcal{B} \cup \mathcal{CN}\}$$

Actually in the tool we also consider all typical method properties like visibility, staticity, stereotypes, parameter passing modalities (in, out, ...), *etc.* Besides, attributes are handled with a separate set. For the sake of simplicity we will not clutter this presentation with all these details.

Let *CProps* be the set of all class properties (like `abstract`, `concrete`, `static`, *etc.*) that are applicable to classes. A *class* is any term of a suitable ground term algebra of the form $c(P, M)$, where c is a class name, P is a set of class properties, and M is a set of methods. Therefore, the set of all classes is defined as:

$$\mathit{Classes} := \{c(P, M) \mid c \in \mathcal{CN}, P \subseteq \mathit{CProps}, M \subseteq \mathit{Methods}\}$$

Class diagrams relations are modeled by means of labeled arrows between classes (like `inh`, `aggr`, `comp`, `assoc`, `dep`, *etc.*) possibly annotated (with stereotypes, roles,

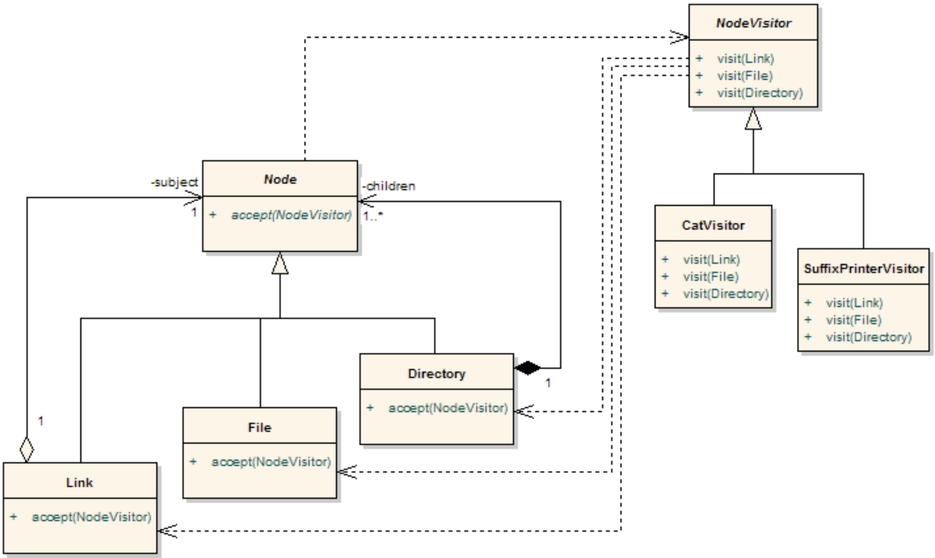

Figure 1. An example of class diagram (taken from [13])

etc.). We denote the set of all arrow labels by $Kinds$, and the set of all annotations by $Annot$. Hence, given $Arrows := Classes \times Classes \times Kinds \times Annot$, a *class diagram* is a pair $\mathcal{D} := (C, A)$ where C is a finite subset of $Classes$, and A is a finite subset of $Arrows$.

Example 2.1 Our representation of the diagram in Figure 1 is the pair $\mathcal{D}_E := (C_E, A_E)$ with

$$\begin{aligned}
C_E := \{&\mathtt{NodeVisitor}(\{\mathtt{abstract}\}, \{\mathtt{visit} : \mathtt{Link} \rightarrow \mathtt{void},\\
&\qquad\qquad\qquad\qquad\qquad\quad \mathtt{visit} : \mathtt{File} \rightarrow \mathtt{void},\\
&\qquad\qquad\qquad\qquad\qquad\quad \mathtt{visit} : \mathtt{Directory} \rightarrow \mathtt{void}\}),\\
&\mathtt{CatVisitor}(\{\mathtt{concrete}\}, \{\mathtt{visit} : \mathtt{Link} \rightarrow \mathtt{void},\\
&\qquad\qquad\qquad\qquad\qquad \mathtt{visit} : \mathtt{File} \rightarrow \mathtt{void},\\
&\qquad\qquad\qquad\qquad\qquad \mathtt{visit} : \mathtt{Directory} \rightarrow \mathtt{void}\}),\\
&\mathtt{SuffixPrinterVisitor}(\{\mathtt{concrete}\}, \{\ldots\}),\\
&\mathtt{Node}(\{\mathtt{abstract}\}, \{\mathtt{accept} : \mathtt{NodeVisitor} \rightarrow \mathtt{void}\}),\\
&\mathtt{Link}(\{\mathtt{concrete}\}, \{\mathtt{accept} : \mathtt{NodeVisitor} \rightarrow \mathtt{void}\}),\\
&\mathtt{File}(\{\mathtt{concrete}\}, \{\mathtt{accept} : \mathtt{NodeVisitor} \rightarrow \mathtt{void}\}),\\
&\mathtt{Directory}(\{\mathtt{concrete}\}, \{\mathtt{accept} : \mathtt{NodeVisitor} \rightarrow \mathtt{void}\})\}
\end{aligned}$$

$$\begin{aligned}A_E := \{&\texttt{Node(...)} \xrightarrow{\texttt{dep}} \texttt{NodeVisitor(...)},\\ &\texttt{CatVisitor(...)} \xrightarrow{\texttt{inh}} \texttt{NodeVisitor(...)},\\ &\texttt{SuffixPrinterVisitor(...)} \xrightarrow{\texttt{inh}} \texttt{NodeVisitor(...)},\\ &\texttt{NodeVisitor(...)} \xrightarrow{\texttt{dep}} \texttt{Link(...)},\quad \texttt{NodeVisitor(...)} \xrightarrow{\texttt{dep}} \texttt{File(...)},\\ &\texttt{NodeVisitor(...)} \xrightarrow{\texttt{dep}} \texttt{Directory(...)},\quad \texttt{Link(...)} \xrightarrow[subject]{\texttt{aggr}} \texttt{Node(...)},\\ &\texttt{Link(...)} \xrightarrow{\texttt{inh}} \texttt{Node(...)},\quad \texttt{File(...)} \xrightarrow{\texttt{inh}} \texttt{Node(...)},\\ &\texttt{Directory(...)} \xrightarrow{\texttt{inh}} \texttt{Node(...)},\quad \texttt{Directory(...)} \xrightarrow[children]{\texttt{comp}} \texttt{Node(...)}\}\end{aligned}$$

Note that for the sake of conciseness we did not show all details (like relation annotations, methods, etc).

3 A language to express software patterns

In this section, we present the syntax and an informal specification of our language for design pattern detection. In the next section we present its formal semantics. For a thorough explanation of design patterns see [8].

3.1 Class patterns

Class patterns are basically non-ground terms providing "templates" for the classes of a class diagram. We define the set of all class patterns *CPatterns* as the typed non-ground extension of the ground term algebra used to define the set *Classes*. A *class pattern* is therefore a class that may contain variables. Note that we consider typed variables, which play the role of placeholders for any unknown part of a class. We thus assume to have infinite sets of variables for each type (class, class name, method, method name, set of methods, *etc.*). We denote $t :: \tau$ a term (thus including a variable) t of type τ. In particular, we denote the set of variables of type $\mathcal{CN}$ by $V_{\mathcal{CN}}$.

3.2 Class diagram patterns

Class diagram patterns formalize general structures to be matched against class diagrams. In the following, we assume that x, y are variables of type *CPatterns*, $a, b :: \mathcal{CN}$, and $v, w \in V_{\mathcal{CN}}$. Formally, a *class diagram pattern* (or simply pattern) p is defined by means of the following BNF-like grammar:

$$\begin{aligned}p ::= &\ \texttt{inh}(x,y) \mid \texttt{dep}(x,y) \mid \texttt{assoc}(x,y) \mid \texttt{aggr}(x,y) \mid \texttt{comp}(x,y) \mid \texttt{any}(x,y)\\ &\texttt{star}(a,b,sp) \mid \texttt{path}(a,b,sp) \mid \texttt{span}(a,sp) \mid\\ &\texttt{hierarchy}(x) \mid \texttt{all2one}(a,b,sp) \mid \texttt{one2all}(a,b,sp) \mid\\ &\texttt{onto}(a,b,sp) \mid \texttt{into}(a,b,sp) \mid \texttt{iso}(a,b,sp) \mid\\ &p \oplus p\\ sp ::= &\ \lambda v\, w.p\end{aligned}$$

Observe that class name variables in a pattern might be bound via an abstraction binder λ. By $\text{free}(p)$ we denote the set of all free variables of p, according to standard definitions[3]. For instance, let $a' \in V_{\mathcal{CN}}$, consider the pattern p [4]

$$\texttt{span}(a', \lambda v\, w.\texttt{comp}(v(\emptyset, \{m\}), w(\emptyset, \emptyset)) \oplus \texttt{aggr}(w(\emptyset, \emptyset), v(\emptyset, \{m\}))).$$

Then, p is a legal pattern with respect to the grammar we defined above and $\text{free}(p) = \{a' :: \mathcal{CN}, m :: \textit{Methods}\}$.

We call *Pat* the set of all class diagram patterns, while *Gpat* represents the set of all ground class diagram patterns, i.e., patterns which do not contain any free variable.

Roughly speaking, our pattern language is equipped with constructs to recognize simple class diagrams relations (such as inheritance, aggregation, *etc.*), as well as to perform more complex matches against the concrete class diagram.

In order to handle constructs like $\texttt{hierarchy}(x)$ (which, as we will explain better in a while, is intended to detect a inheritance hierarchy rooted at x) we need to "stretch" the usual meaning of matching. Instead of obtaining a substitution for the pattern which makes it identical to the matched term, we (possibly) need in return a subgraph. For example to "match" $\texttt{hierarchy}(x)$ we need to find within the class diagram the whole hierarchy rooted at x. Thus in our setting, matching a pattern with respect to a given class diagram corresponds to extracting the set of all the class diagram relations which represent the matched substructure. In other words, the execution of a pattern p against a class diagram $\mathcal{D}$ returns a subset S of $\wp(\textit{Arrows})$ such that each $s \in S$ is a match of p with respect to $\mathcal{D}$. When the execution returns $S = \emptyset$, no match is found and we say that the pattern p fails on $\mathcal{D}$. Thus in the following we will use expressions like "the match of pattern p extracts the set of relations R" meaning that we (non-deterministically) found a subgraph R which is an instance of p.

We will provide the formal semantics of the language in Section 4. In the rest of this section, we briefly illustrate the intended behavior of our basic constructs by means of intuitive examples which refer to the class diagram of Figure 1.

By $sp(a, b)$, we denote the pattern which is obtained from $sp = \lambda v\, w.p$ by replacing v with a and w with b in p.

$arrow(x, y)$, $arrow \in \{\texttt{inh}, \texttt{dep}, \texttt{assoc}, \texttt{aggr}, \texttt{comp}\}$ These constructs allow to define patterns modeling the class diagram relations (i.e., inheritance, dependency, association, aggregation, composition arrows) from a class pattern x to a class pattern y. For instance, the pattern

$$\texttt{comp}(a(\{\texttt{concrete}\}, \{w\}), \texttt{Node}(\emptyset, \{w\}))$$

---

[3] In particular, as the only variables that can be bound by λ are of type $\mathcal{CN}$, all variables of other types are always free.

[4] Note that $m :: \textit{Methods}$ and $a, a', v, w :: \mathcal{CN}$.

matches, against $\mathcal{D}_E$ of Example 2.1,

$$\{\texttt{Directory}(\{\texttt{concrete}\},\{\texttt{accept}:\texttt{NodeVisitor}\rightarrow\texttt{void}\})\xrightarrow[\textit{children}]{\texttt{comp}}\\ \texttt{Node}(\{\texttt{abstract}\},\{\texttt{accept}:\texttt{NodeVisitor}\rightarrow\texttt{void}\})\}$$

after instantiating variable a with `Directory` and variable w with `accept : NodeVisitor → void`.

$\underline{\texttt{any}(\boldsymbol{x},\boldsymbol{y})}|$ This construct allows to match any class diagram relation from class pattern x to class pattern y independently of the arrow label. For instance,

$$\texttt{any}(a(\emptyset,\emptyset),\texttt{NodeVisitor}(\{\texttt{abstract}\},\emptyset))$$

matches (against $\mathcal{D}_E$ of Example 2.1) the three arrows

$$\begin{aligned}&\{\{\texttt{Node}(\ldots)\xrightarrow{\texttt{dep}}\texttt{NodeVisitor}(\ldots)\},\\ &\ \{\texttt{CatVisitor}(\ldots)\xrightarrow{\texttt{inh}}\texttt{NodeVisitor}(\ldots)\},\\ &\ \{\texttt{SuffixPrinterVisitor}(\ldots)\xrightarrow{\texttt{inh}}\texttt{NodeVisitor}(\ldots)\}\}\end{aligned}$$

when a is respectively replaced by `Node`, `NodeVisitor`, and `SuffixPrinterVisitor`.

$\underline{\texttt{path}(\boldsymbol{a},\boldsymbol{b},\boldsymbol{sp})}|$ This construct allows to find a concatenated sequence of matches of subpattern sp starting from a class identified by a and ending in a class identified by b. For instance, the pattern

$$\texttt{path}(\texttt{Node},\texttt{Directory},\lambda a\,b.\texttt{dep}(a(\{\texttt{abstract}\},\emptyset),b(\{\texttt{concrete}\},\emptyset)))$$

matches the unique dependency path from the abstract class `Node` to the concrete class `Directory`, i.e.,

$$\{\texttt{Node}(\ldots)\xrightarrow{\texttt{dep}}\texttt{NodeVisitor}(\ldots),\quad\texttt{NodeVisitor}(\ldots)\xrightarrow{\texttt{dep}}\texttt{Directory}(\ldots)\}$$

$\underline{\texttt{hierarchy}(\boldsymbol{x})}|$ The construct `hierarchy` selects all the inheritance arrows of a given hierarchy whose root class matches the class pattern x. For example,

$$\texttt{hierarchy}(\texttt{NodeVisitor}(\{\texttt{abstract}\},\{w_1,w_2,w_3\}))$$

matches the whole hierarchy (consisting, in this case, of two inheritance arrows)

$$\begin{aligned}&\{\texttt{CatVisitor}(\ldots)\xrightarrow{\texttt{inh}}\texttt{NodeVisitor}(\ldots),\\ &\ \texttt{SuffixPrinterVisitor}(\ldots)\xrightarrow{\texttt{inh}}\texttt{NodeVisitor}(\ldots)\}\end{aligned}$$

$\underline{\texttt{span}(\boldsymbol{a},\boldsymbol{sp})}|$ The `span` operator extracts all the class diagram relations matching the union of the patterns $sp(a, z)$, for any $z \in \mathcal{CN}$. For example, the pattern

$$\texttt{span}(\texttt{NodeVisitor},\lambda a\,b.\texttt{dep}(a(\{\texttt{abstract}\},\{w_1,w_2,w_3\}),b(\emptyset,\emptyset))$$

matches

$$\{\texttt{NodeVisitor}(\ldots) \xrightarrow{\texttt{dep}} \texttt{Link}(\ldots),\quad \texttt{NodeVisitor}(\ldots) \xrightarrow{\texttt{dep}} \texttt{File}(\ldots),$$
$$\texttt{NodeVisitor}(\ldots) \xrightarrow{\texttt{dep}} \texttt{Directory}(\ldots)\}$$

$\texttt{star}(a, b, sp)$ The `star` operator returns the set of all the class diagram relations matching the subpattern $sp(a, b)$. The application of this construct never fails, since —when no match is found— it returns the set containing the empty set. Consider the pattern

$$\texttt{star}(\texttt{File}, b, \lambda v\, w.\texttt{aggr}(v(\{\texttt{concrete}\}, \emptyset), w(\emptyset, \emptyset)))$$

as there are no outgoing aggregation arrows from the concrete class `File`, the `star` operator returns $\{\emptyset\}$. On the other hand, the pattern

$$\texttt{aggr}(\texttt{File}(\{\texttt{concrete}\}, \emptyset), b(\emptyset, \emptyset))$$

would fail on the given class diagram.

$\texttt{all2one}(a, b, sp)$ Let H be a hierarchy whose root class has a name matching a, and L be the set of all the class names of the leaf classes of H. Let cn be a class name matching b. If $sp(l, cn)$ does not fail, for $l \in L$, then the `all2one` operator returns the union of all $sp(l, cn)$, $l \in L$, otherwise it fails. For instance,

$$\texttt{all2one}(\texttt{NodeVisitor}, \texttt{NodeVisitor},$$
$$\lambda a\, b.\texttt{inh}(a(\{\texttt{abstract}\}, \emptyset), b(\{\texttt{abstract}\}, \emptyset)))$$

generates all the inheritance arrows corresponding to the hierarchy rooted by the class `NodeVisitor` [5].

$\texttt{one2all}(a, b, sp)$ This construct behaves symmetrically to the `all2one` operator. Let H be a hierarchy whose root class has a name matching b, and L be the set of all the class names of the leaf classes of H. Let cn be a class name matching a. If $sp(cn, l)$ does not fail, for $l \in L$, then the `all2one` operator returns the union of all $sp(cn, l)$, $l \in L$, otherwise it fails.

$morph(a, b, sp)$, $morph \in \{\texttt{onto}, \texttt{into}, \texttt{iso}\}$ *morph* constructs allow to recognize surjective (`onto`), injective (`into`), and bijective (`iso`) "morphisms" between the leaf classes of two hierarchies whose root class names match a and b respectively. Morphisms are represented by the subpattern *sp*. As an example, we might employ the `iso` construct in order to find a one-to-one correspondence between leaf classes of two hierarchies with respect to the dependency arrow kind as required to specify the Factory Method pattern [8]. In this case, the pattern could be $\texttt{iso}(a, b, \lambda v\, w.\texttt{dep}(v, w))$, where a and b are two class names representing two root classes in the class diagram.

---

[5] This is indeed a completely artificial example, but with the model of Figure 1 we cannot show more natural examples of the `all2one` constructor.

$p \oplus p$ $\oplus$ is a binary, associative and commutative operator, which can be employed to build compound patterns starting from simpler ones. The pattern $p_1 \oplus p_2$ fails to find a match, whenever either p_1 or p_2 fail on the given class diagram. Otherwise the union of the matches computed by the compound pattern is delivered. For instance, the pattern $\texttt{aggr}(a(\emptyset,\emptyset), b(\emptyset,\emptyset)) \oplus \texttt{any}(a(\emptyset,\emptyset), b(\emptyset,\emptyset))$ matches

$$\{\texttt{Link}(\ldots) \xrightarrow[subject]{\texttt{aggr}} \texttt{Node}(\ldots), \quad \texttt{Link}(\ldots) \xrightarrow{\texttt{inh}} \texttt{Node}(\ldots)\}$$

while $\texttt{hierarchy}(cp1) \oplus \texttt{hierarchy}(cp2) \oplus \texttt{dep}(cp1, cp2)$ searches for hierarchies, whose root classes match class patterns *cp1* and *cp1* and that are connected through a dependency arrow.

Example 3.1 The Proxy pattern [8] can be represented with our syntax as

$$\begin{array}{l}\texttt{aggr}(proxy(\{\texttt{concrete}\}, \{m\}), realsubj(\{\texttt{concrete}\}, \{m\})) \oplus \\ \texttt{inh}(proxy(\{\texttt{concrete}\}, \{m\}), subj(\{\texttt{abstract}\}, \{m\})) \oplus \\ \texttt{inh}(realsubj(\{\texttt{concrete}\}, \{m\}), subj(\{\texttt{abstract}\}, \{m\}))\end{array}$$

Example 3.2 The Composite pattern can be represented with our syntax as

$$\begin{array}{l}\texttt{comp}(composite(\{\texttt{concrete}\}, \emptyset), component(\{\texttt{abstract}\}, \emptyset)) \oplus \\ \texttt{path}(composite, component, \lambda a\, b.\texttt{inh}(a(\{\texttt{concrete}\}, \emptyset), b(\{\texttt{abstract}\}, \emptyset))) \oplus \\ \texttt{hierarchy}(component(\{\texttt{abstract}\}, \emptyset))\end{array}$$

Example 3.3 The Visitor pattern can be represented with our syntax as

$$\begin{array}{l}\texttt{dep}(element(\{\texttt{abstract}\}, \emptyset), visitor(\{\texttt{abstract}\}, \emptyset)) \oplus \\ \texttt{one2all}(visitor, element, \lambda a\, b.\texttt{dep}(a, b(\{\texttt{concrete}\}, \emptyset))) \oplus \\ \texttt{hierarchy}(element) \oplus \texttt{hierarchy}(visitor)\end{array}$$

Example 3.4 The Abstract Factory pattern can be represented as follows:

$$\begin{array}{l}\texttt{span}(absFact(\{\texttt{abstract}\}, \emptyset), \\ \qquad\qquad \lambda v\; absProd.\texttt{iso}(v, absProd(\{\texttt{abstract}\}, \emptyset), \lambda w\, z.\texttt{dep}(w, z)))\end{array}$$

This actually is a fruitful example of usage of nesting of subpatterns, in order to report just a single instance of Abstract Factory instead of several disjoint instances (one for each abstract product) as other methods do.

Example 3.5 The Circular Dependencies *antipattern*, which is used to detect any circular path in a class diagram, can be represented with our syntax as

$$\texttt{path}(a, b, \lambda v\, w.\texttt{any}(v(\emptyset,\emptyset), w(\emptyset,\emptyset)) \oplus \texttt{path}(b, a, \lambda v\, w.\texttt{any}(v(\emptyset,\emptyset), w(\emptyset,\emptyset))$$

4 Rule-based pattern semantics

In this section we provide a rule-based semantics formalizing the behavior of the language we presented in Section 3. Basically, given a class diagram $\mathcal{D}$, we first define an evaluation function $[\![\cdot]\!]_{\mathcal{D}}\colon \mathit{Gpat} \rightarrow \wp(\mathit{Arrows})$ such that, for any ground pattern p (i.e., a pattern without free variables), $[\![p]\!]_{\mathcal{D}}$ returns a set of sets of class diagram relations. Each set of class diagram relation represents a possible match of the pattern p against $\mathcal{D}$. Then we lift $[\![\cdot]\!]_{\mathcal{D}}\colon \mathit{Gpat} \rightarrow \wp(\mathit{Arrows})$ to an evaluation function $[\![\cdot]\!]_{\mathcal{D}}\colon \mathit{Pat} \rightarrow \wp(\mathit{Arrows})$ [6] to manage non-ground patterns.

We formalize the evaluation function $[\![\cdot]\!]_{\mathcal{D}}\colon \mathit{Gpat} \rightarrow \wp(\mathit{Arrows})$ by induction on the syntax of the language constructs described in Section 3. Note that each case can be directly translated into (possibly conditional) rules which can be easily implemented using any functional language (see Section 5).

In order to give the formal definition of the evaluation function, we need the following auxiliary notions. Let S be a set of sets, then $\mathrm{flat}(S) := \bigcup_{X \in S} X$. Consider a root class x with class name cn of some hierarchy, then $\mathrm{lf}(cn)$ is the set of all the class names of the leaf classes of the hierarchy rooted by x. [7]

Definition 4.1 [**evaluation function**] Let $\mathcal{D}$=(C,A) be a class diagram. Let $\mathit{arrow} \in \{\mathtt{inh}, \mathtt{dep}, \mathtt{assoc}, \mathtt{aggr}, \mathtt{comp}\}$, x, y, $a_1(P_1, M_1)$, $a_2(P_2, M_2) \in \mathit{Classes}$, $a, b \in \mathcal{CN}$, and $p, p_1, p_2 \in \mathit{Gpat}$.

$$
\begin{aligned}
&[\![\mathit{arrow}(a_1(P_1,M_1), a_2(P_2,M_2))]\!]_{\mathcal{D}} := \\
&\quad \{\{e\} \mid e \equiv (a_1(P',M'), a_2(P'',M''), \mathit{arrow}, an) \in A, \\
&\qquad\quad M_1 \subseteq M', P_1 \subseteq P', M_2 \subseteq M'', P_2 \subseteq P''\} \\
&[\![\mathtt{any}(a_1(P_1,M_1), a_2(P_2,M_2))]\!]_{\mathcal{D}} := \\
&\quad \{\{e\} \mid e \equiv (a_1(P',M'), a_2(P'',M''), \mathit{arrow}, an) \in A, \\
&\qquad\quad \mathit{arrow} \in \{\mathtt{inh}, \mathtt{dep}, \mathtt{assoc}, \mathtt{aggr}, \mathtt{comp}\}, M_1 \subseteq M', P_1 \subseteq P', \\
&\qquad\quad M_2 \subseteq M'', P_2 \subseteq P''\} \\
&[\![\mathtt{star}(a, b, sp)]\!]_{\mathcal{D}} := \{\emptyset \cup \{\mathrm{flat}([\![sp(a,b)]\!]_{\mathcal{D}})\}\} \\
&[\![\mathtt{span}(a, sp)]\!]_{\mathcal{D}} := \{\mathrm{flat}(\bigcup_{b \in \mathcal{CN}} [\![sp(a,b)]\!]_{\mathcal{D}})\} \\
&[\![\mathtt{hierarchy}(x)]\!]_{\mathcal{D}} := \{\mathrm{flat}(\{\mathrm{flat}(\mathtt{inh}(z,x)) \cup \mathtt{hierarchy}(z) \mid \\
&\qquad\qquad\qquad\qquad z \in \mathit{Classes}, \mathtt{inh}(z,x) \neq \emptyset\})\} \\
&[\![\mathtt{all2one}(a, b, sp)]\!]_{\mathcal{D}} := \begin{cases} \{\mathrm{flat}(\bigcup_{z \in \mathcal{CN}} [\![sp(z,b)]\!]_{\mathcal{D}})\} & \text{if } Q_{a2o}(a,b) \\ \emptyset & \text{otherwise} \end{cases}
\end{aligned}
$$

where $Q_{a2o}(a,b) := \forall z \in \mathcal{CN}, z \in \mathrm{lf}(a), [\![sp(z,b)]\!]_{\mathcal{D}} \neq \emptyset$

$$
[\![\mathtt{one2all}(a, b, sp)]\!]_{\mathcal{D}} := \begin{cases} \{\mathrm{flat}(\bigcup_{z \in \mathcal{CN}} [\![sp(a,z)]\!]_{\mathcal{D}})\} & \text{if } Q_{o2a}(a,b) \\ \emptyset & \text{otherwise} \end{cases}
$$

---

[6] By abuse of notation, the functional symbol $[\![\cdot]\!]_{\mathcal{D}}$ is overloaded to deal with non-ground patterns.

[7] Classes are univocally identified by means of their class name.

where $Q_{o2a}(a,b) := \forall z \in \mathcal{CN}, z \in \mathrm{lf}(b), [\![sp(a,z)]\!]_{\mathcal{D}} \neq \emptyset$

$$[\![p_1 \oplus p_2]\!]_{\mathcal{D}} := \{X_1 \cup X_2 \mid X_1 \in [\![p_1]\!]_{\mathcal{D}}, X_2 \in [\![p_2]\!]_{\mathcal{D}}\}$$

$$[\![\mathtt{path}(a,b,sp)]\!]_{\mathcal{D}} := [\![sp(a,b)]\!]_{\mathcal{D}} \cup \left(\bigcup_{z \in \mathcal{CN}} [\![sp(a,z)]\!]_{\mathcal{D}} \oplus \mathtt{path}(z,b,sp) \right)$$

$$[\![\mathtt{onto}(a,b,sp)]\!]_{\mathcal{D}} := \begin{cases} \{\mathrm{flat}(\bigcup_{z_1,z_2 \in \mathcal{CN}} [\![sp(z_1,z_2)]\!]_{\mathcal{D}})\} & \text{if } Q_{surj}(a,b) \\ \emptyset & \text{otherwise} \end{cases}$$

where $Q_{surj}(a,b) := \forall z_1 \in \mathcal{CN}, \exists z_2 \in \mathcal{CN}, z_1 \in \mathrm{lf}(a), z_2 \in \mathrm{lf}(b), [\![sp(z_1,z_2)]\!]_{\mathcal{D}} \neq \emptyset$.

The definition for $\mathtt{into}(a,b,sp)$ and $\mathtt{iso}(a,b,sp)$ is similar to the one presented for $\mathtt{onto}(a,b,sp)$, and can be obtained by simply replacing the predicate $Q_{surj}(a,b)$ with a suitable predicate $Q_{inj}(a,b)$ (respectively, $Q_{biject}(a,b)$) modeling an injective (respectively, bijective) morphism.

The function $[\![\cdot]\!]_{\mathcal{D}}\colon Gpat \to \wp(Arrows)$ can be easily extended to manage non-ground patterns. We simply take the union of the evaluations of all the ground instances of the considered non-ground pattern.

More formally, given a non-ground pattern p, $[\![\cdot]\!]_{\mathcal{D}}\colon Pat \to \wp(Arrows)$ is defined as

$$[\![p]\!]_{\mathcal{D}} := \bigcup_{\overline{p} \in \langle p \rangle} [\![\overline{p}]\!]_{\mathcal{D}}$$

where $\langle p \rangle$ is the set containing all the ground patterns which are instances of p.

5 Matching method

We have developed a nondeterministic matching method which computes (by need) the aforementioned ground pattern semantics. A prototype of the algorithm, which is coded in Curry (as well as in Haskell[8]), implements an optimized version of the evaluation function $[\![\cdot]\!]_{\mathcal{D}}\colon Pat \to \wp(Arrows)$ which shrinks the search space in several ways.

Roughly speaking, we evaluate a (non necessarily ground) pattern p by induction on the syntax, while generating instances for variables only when needed. Moreover (within the process), whenever we have to match a pattern variable, we apply the resulting substitution on the whole p (similarly to what happens when using narrowing in a functional logic setting) to generate more instantiated subpatterns of p to narrow the search space.

In order to reduce the degree of nondeterminism, the algorithm employes a destructive matching mechanism; that is, when a nondeterministic match is found, the matched substructure is removed from the class diagram representation, and so the search proceeds on a smaller data structures. This approach preserves the completeness of the matching with respect to the pattern semantics, since any ground arrow

---

[8] We tried both to experiment the differences in the two languages.

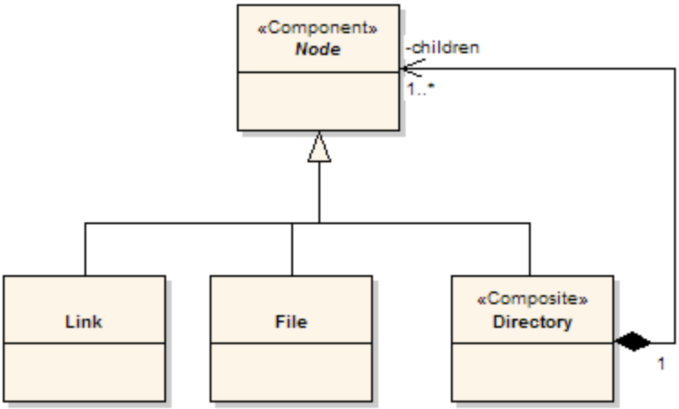

Figure 2. Recognized instance of the Composite pattern

cannot be matched twice by two distinct subpatterns of a given pattern. Moreover, this avoids the generation of multiple versions of the same solutions.

Let us present in much detail what happens for pattern $\mathtt{path}(a, b, sp) \oplus p$, where $sp := \lambda v\, w.p'$. For the operator `path` we have two rules (which have to be tried nondeterministically).

Base case We "apply" sp to a and b by matching v and w with a and b. If the match succeeds then p', which has been instantiated by the match, is recursively tried for match. If it succeeds then its result is returned. The "upper level" then tries p for match and if this succeeds the union of each result is returned.

Inductive case We generate a fresh variable z. Then we proceed as in the base case but for variables a and z. If we succeed then we recursively try $\mathtt{path}(z, b, sp)$ and if it succeeds then the union of each result is returned. (Then the "upper level" will go on with p).

For the sake of conciseness we do not present all cases, as all rules of the matching are derived analogously from the definition of the semantics.

5.1 Examples of matches

Example 5.1 Let us show an example of a simple pattern that can also be discovered by *ad hoc* tools. Consider the Composite Pattern p_{com} formalized in Example 3.2. The execution of the match of p_{com} against $\mathcal{D}_E$ of Example 2.1 is

$$\begin{aligned}\{\{&\mathtt{Link}(\ldots) \xrightarrow{\mathtt{inh}} \mathtt{Node}(\ldots), \quad \mathtt{File}(\ldots) \xrightarrow{\mathtt{inh}} \mathtt{Node}(\ldots),\\ &\mathtt{Directory}(\ldots) \xrightarrow{\mathtt{inh}} \mathtt{Node}(\ldots),\\ &\mathtt{Directory}(\ldots) \xrightarrow[\mathit{children}]{\mathtt{comp}} \mathtt{Node}(\ldots)\}\}\end{aligned}$$

Figure 2 illustrates the (graphical version of the) outcome of the execution.

Example 5.2 Consider the compound pattern p_{vis} formalized in Example 3.3. Figure 3 illustrates the outcome of the execution of p_{vis} against $\mathcal{D}_E$.

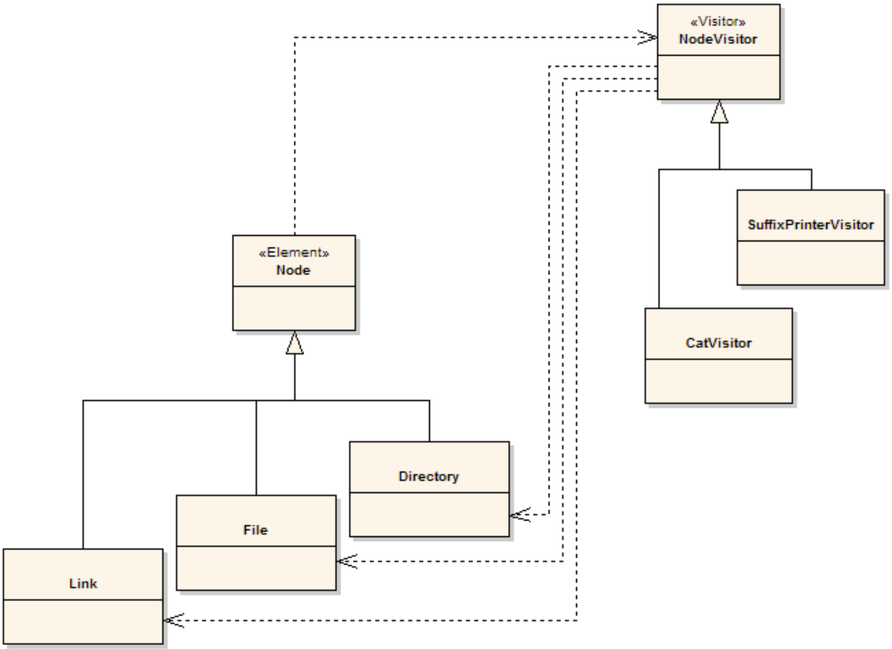

Figure 3. Recognized instance of the Visitor pattern

Example 5.3 Now let us see what happens with variants of canonical patterns. The designer which produced the diagram of Figure 1 did not employ a canonical Proxy pattern [8] to model a proxy architecture in his class diagram; instead, he changed it by "moving" the original aggregation between "Proxy" and "Real Subject" classes, to an aggregation between "Proxy" and "Subject" classes.

Our implementation reports indeed no matches of the Proxy pattern formalized in Example 3.1. But if we change the target pattern, according to the designer's variation, in

$$\begin{aligned}&\texttt{aggr}(proxy(\{\texttt{concrete}\},\emptyset), subj(\{\texttt{abstract}\},\emptyset)) \oplus\\ &\texttt{hierarchy}(subj(\{\texttt{abstract}\},\emptyset))\end{aligned}$$

our matching algorithm finds instead a solution. The outcome of its execution is illustrated in Figure 4.

Example 5.4 Let us now see an example with an antipattern which cannot be expressed with LePUS. Consider the pattern which we formalized in Example 3.5.

By executing such a pattern against $\mathcal{D}_E$, we are able to detect all the cycles appearing in the diagram. Concretely, we recognize all five cycles involving the classes `Node`, `NodeVisitor`, `Directory`, `File` and `Link`.

5.2 Some details about the tool

In our prototype, for the moment, in order to simplify the implementation, the operator $\oplus$ is not commutative and subpatterns are matched left to right. This leads to the necessity to write patterns in the right order.

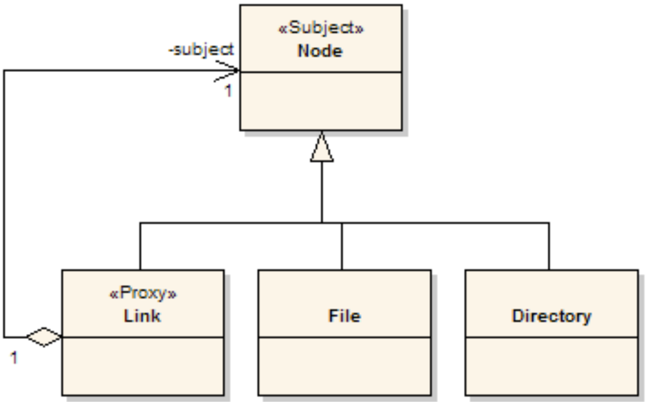

Figure 4. Recognized instance of the Proxy pattern variant

```
data Relation = Relation RelationKind Stereotype Type Type
data RelationKind = WithoutProp WithoutPropKind
                  | WithProp AggregationKind RelProp
data Type = Class TyName AbsKind Stat Stereotype Attrs Methods
          | Interface UmlName Methods
data TyName = TyName UmlName [TyName]
type Attrs = Set Attribute
type Methods = Set Method
data Attribute = Attribute Visib Stereotype UmlName TyName
data Method = Method AbsKind Visib Stereotype UmlName Params TyName
newtype Params = Params [Parameter]
data Parameter = Parameter ParameterKind UmlName TyName
```

Figure 5. An excerpt from the data structures of the prototype

As told before, actually within the prototype we handle all the details of class diagrams. See the excerpt from the data structures of the prototype in Figure 5 to have an idea.

We are currently working on a (quite technical) extension of the tool. As the information which is contained in class diagrams can be synthesized by inspecting source code, we are integrating into the matching mechanism the possibility to extract class information directly from the code. The difficult part of this is just to build a parser for the target language, then it is just a matter of visiting the parse tree.

6 Conclusions and future work

In this paper we proposed a formal language for describing both design patterns and antipatterns. We provided a small number of basic building blocks, that are ubiquitous in object-oriented design, plus some language connectives to glue other blocks as desired, instead of giving an *ad hoc* collection of loosely related concepts. With this language one can easily define descriptions of common design patterns, as well as customized variations or arbitrary compositions of other patterns.

We developed a rule-based matching method that finds all instances of a pattern in the designers' diagrams. Moreover, whenever code is also provided matching information can be (if needed) extracted from the code. The result encompasses exact *ad hoc* proposals (like [14,3,10]), approximate ones (like [12,9,2,1]), as well as more expressive ones like [7,6].

We have implemented a prototype in Curry (and Haskell).

We are now working on several refinements of both our theoretical method and prototype. We are developing a semantically equivalent graphical version of the proposed patterns (as the one of [7,6]). This could be the key for the adoption of our tool in the UML community, as designers could use it without much cognitive load. Furthermore, the prototype could be seamlessly integrated within a UML editor.

We are modifying the matching method to have another interesting behavior. As it can happen that a presumed pattern is no longer found because it has been broken, we are changing the method to suggest/show which modifications should be performed in a diagram to *fix* the pattern.

Another interesting extension that we would like to carry over is relative to sequence diagrams. Several common patterns are supplied with natural language notes which specify constraints over methods definitions [8]. We would like to extend our patterns with sequence-patterns which have to match over UML model's sequence diagrams (or suitable methods calls in the code) to be able to fully assure that we have found a proper pattern instance.

References

[1] H. Albin-Amiot, P. Cointe, Y.-G. Gueheneuc, and N. Jussien. Instantiating and detecting design patterns: Putting bits and pieces together. In *Proc. of the 6th IEEE International Conference on Automated Software Engineering (ASE 2001)*, pages 166–173, Los Alamitos, CA, USA, 2001. IEEE Computer Society.

[2] H. Albin-Amiot and Y.-G. Guhneuc. Meta-modeling design patterns: Application to pattern detection and code synthesis. In Bedir Tekinerdogan, editor, *Proceedings of ECOOP Workshop on Automating Object-Oriented Software Development Methods*, 2001.

[3] F. Bergenti and A. Poggi. Improving uml design using automatic design pattern detection. In *Proc. 12th. International Conference on Software Engineering and Knowledge Engineering (SEKE 2000)*, pages 336–343, 2000.

[4] G. Booch, J. Rumbaugh, and I. Jacobson. *Unified Modeling Language User Guide, The (2nd Edition)*. Addison-Wesley Professional, 2005.

[5] W. J. Brown, R. C. Malveau, H. W. McCormick, III, and T. J. Mowbray. *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis*. John Wiley & Sons, New York, 1998.

[6] A. H. Eden. Formal Specification of Object-Oriented Design. In *Proc. Int'l Conf. Multidisciplinary Design in Engineering CSME-MDE 2001, Montreal, Canada*, 2001.

[7] A. H. Eden. LePUS: A Visual Formalism for Object-Oriented Architectures. In *Proc. 6th World Conf. Integrated Design and Process Technology—IDPT 2002, Pasadena, CA, USA*, 2002.

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

[9] O. Kaczor, Y.-G. Gueheneuc, and S. Hamel. Efficient identification of design patterns with bit-vector algorithm. In *CSMR '06: Proceedings of the Conference on Software Maintenance and Reengineering*, pages 175–184, Washington, DC, USA, 2006. IEEE Computer Society.

[10] N. Moha, Y.-G. Gueheneuc, and P. Leduc. Automatic generation of detection algorithms for design defects. In *ASE '06: Proceedings of the 21st IEEE International Conference on Automated Software Engineering (ASE'06)*, pages 297–300, Washington, DC, USA, 2006. IEEE Computer Society.

[11] Object Management Group. *UML 2.0 Superstructure Specification, v2.0.* Document – formal/05-07-04 (UML Superstructure Specification, v2.0).

[12] N. Tsantalis, A. Chatzigeorgiou, G. Stephanides, and S.T. Halkidis. Design pattern detection using similarity scoring. *IEEE Transactions on Software Engineering*, 32(11):896–909, 2006.

[13] J. M. Vlissides. *Pattern Hatching: Design Patterns Applied.* Addison-Wesley, Reading, MA, 1995.

[14] M. Vokác. An efficient tool for recovering Design Patterns from C++ Code. *Journal of Object Technology*, 5(1):139–157, 2006.