A Model-Based Graph-Matching Approach for Design Patterns Detection

Mario Luca Bernardi
Università degli Studi del Sannio
153 PUBLICATIONS **1,141** CITATIONS
SEE PROFILE

Marta Cimitile
UnitelmaSapienza University of Rome
155 PUBLICATIONS **1,218** CITATIONS
SEE PROFILE

Giuseppe A. Di Lucca
Università degli Studi del Sannio
124 PUBLICATIONS **2,424** CITATIONS
SEE PROFILE

A Model-Driven Graph-Matching Approach for Design Pattern Detection

Mario Luca Bernardi
Department of Engineering
University of Sannio, Italy
mlbernar@unisannio.it

Marta Cimitile
Faculty of Economics
Unitelma Sapienza University,Italy
marta.cimitile@unitelma.it

Giuseppe Antonio Di Lucca
Department of Engineering
University of Sannio, Italy
dilucca@unisannio.it

***Abstract*—In this paper an approach to automatically detect Design Patterns (DPs) in Object Oriented systems is presented. It allows to link system's source code components to the roles they play in each pattern. DPs are modelled by high level structural properties (e.g. inheritance, dependency, invocation, delegation, type nesting and membership relationships) that are checked against the system structure and components. The proposed metamodel also allows to define DP variants, overriding the structural properties of existing DP models, to improve detection quality. The approach was validated on an open benchmark containing several open-source systems of increasing sizes. Moreover, for other two systems, the results have been compared with the ones from a similar approach existing in literature. The results obtained on the analyzed systems, the identified variants and the efficency and effectiveness of the approach are thoroughly presented and discussed.**

***Index Terms*—Reverse Engineering, Design Patterns Mining, Software Comprehension, Source Code Analysis**

I. Introduction

Detection of Design Patterns (DPs) instances in Object Oriented (OO) software systems can help to better structure, understand, maintain and reuse them [1]. Indeed, the lack of adequate documentation in a software system may make it hard to understand which are the adopted design solutions and patterns and where (in which code components) they are implemented [2, 3]. To address these issues, several methodologies, approaches and tools have been proposed in the literature in the last twenty years. Most of these approaches, however, are sensitive to structural differences of searched patterns with respect their specifications. Moreover, the kind of properties considered to specify a pattern are often fixed and limited. In this paper we propose an approach, centered on a metamodel, representing both the software system and the patterns to be detected as graphs. These graphs exploit a wider set of high level properties related to the source code elements, the static relationships among them, and their behavior. The patterns identification is performed by traversing these graphs and by annotating the source code elements of the system with information on the roles they play in each pattern model.

To improve detection quality and flexibility, the proposed approach organizes the design pattern models as a hierarchy of declarative specifications in order to take into account structural implementation differences. A variant can be expressed as a set of changes to an existing specification by adding, removing or relaxing properties. Hence, a new pattern specification can be derived from an existing one (to detect a variant) or written from scratch (to detect a new kind of pattern), with no impact on the algorithm or tool. This approach also helps to reduce the size of the search space since variants share part of the search tree giving more chances to prune it.

An eclipse-based tool, called Design Pattern Finder (DPF) was developed to provide an automatic support to the approach. The description of the tool is out of the scope of this paper; however it is reported in [4].

The approach has been assessed by applying it to an open benchmark comprised of 11 Java systems (of which QuickUML 2001, Lexi v0.1.1 alpha, JUnit v3.7, JHotDraw v5.1, MapperXML v1.9.7, Nutch v0.4 and PMD v1.8) proposed in [5] and [6]. Finally, for other two systems, we compared our results with the ones obtained using the Design Pattern Detector (DPD) Tool proposed in [2]. In this case the results have been validated by experts in order to evaluate precision and recall considering the true positives of both tools as an (incomplete) gold standard.

This paper improves and enhances our previous preliminary investigation reported in [7]. The improvements and enhancements are mainly referred to: (i) a more complete implementation of the metamodel used by the detection algorithm; (ii) the definition of a larger and richer set of detected patterns; (iii) the development of the prototype DPF tool; (iv) the discussion of the results provided by DPF on eight systems, comparing them with results provided by both an open benchmark and DPD tool.

The paper is structured as follows. In Section II relevant related works are discussed. Section III presents the metamodel defined to represent the pattern structure in terms of properties. Section IV presents the graph-matching approach to detect patterns in a Java system. Section V discusses the case study. Section VI contains conclusive remarks and briefly discusses future work.

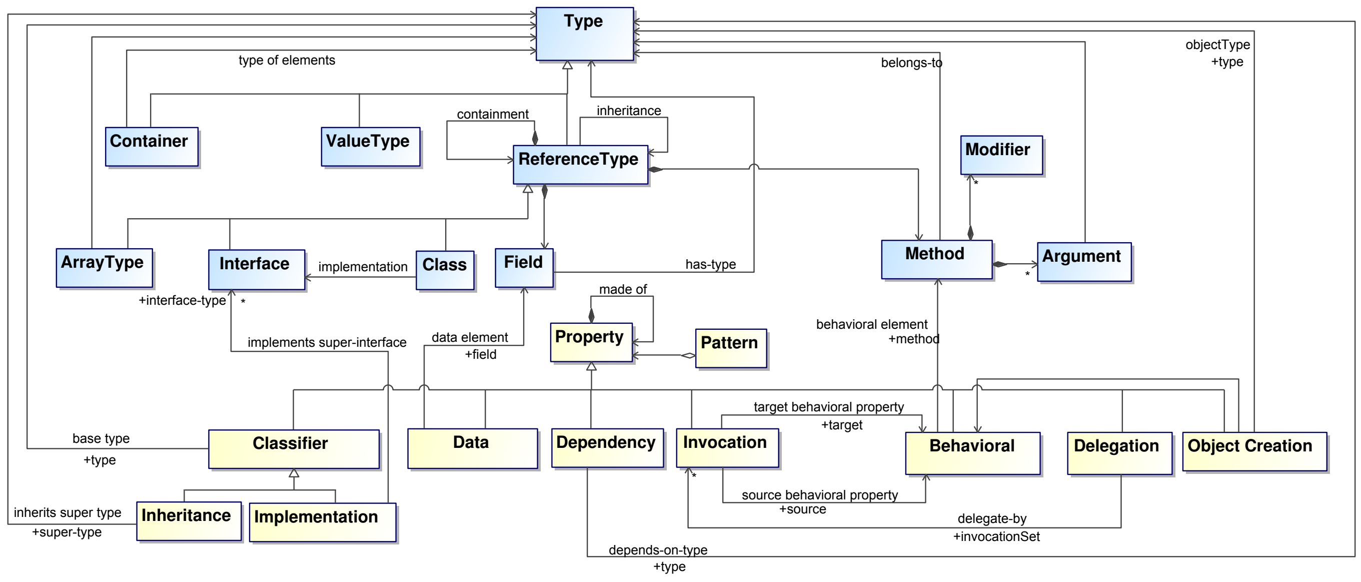

Fig. 1. The meta-model represented as a UML class diagram

II. Related Work

Some reviews on current techniques and tools for discovering architecture and design patterns from OO systems, are provided in [8] and [9]. In the last work, the authors classified pattern recovery techniques according to the type of analysis used and the searching methodology adopted. For brevity, we focus on the searching method classification (a wider discussion is in the technical report available at [4]).

Several design pattern recovery techniques use queries on DB for detection [10, 11, 12]. They provide an intermediate representation of the source code (i.e. ASG, AST, XMI, metadata and UML structures) and then use SQL queries to extract pattern related information. The limit of these approaches is that SQL languages are not suitable to easily represent and query the complex graph structure generated by system ASTs. These techniques are used mainly for simple structural and creational patterns and they only partially support the recovery of behavioral ones.

Metric based techniques compute program related metrics (i.e. generalizations, aggregations, associations, interface hierarchies) from different source code representations and compare their values with source code DP metrics. These techniques [13, 14, 15] are computationally efficient because metric computation is less expensive than structural pattern recognition and do not require heuristic approach to reduce the search space through filtration [16]. Their precision and recall are usually low; moreover they were experimented on few design patterns in literature.

XPG formalism and parsing techniques use SVG (scalable vector graphics) format for the intermediate representation of the source code and represent design patterns in a visual language by mapping the visual language grammar of each pattern with the graph representation. In [17], De Lucia et al. present some case studies of recovering structural design patterns from OO source code. They use a recovery technique based on the parsing of visual languages, and supported by a visual environment automatically produced by a grammar based visual environment generator. They give a precise visualization but in our knowledge, the existing experimentation is limited to few patterns.

UML structures, represented as matrices, are used in several works [2, 18, 14] to model structural and behavioral information of software systems. These techniques are applied to match a DP template matrix with the matrix generated for the system. In particular, a DP detection methodology based on similarity scoring between graph vertexes is proposed in [2]. The approach is able to also recognize patterns that are slightly modified from their standard representation. It exploits the fact that patterns reside in one or more inheritance hierarchies (in order to reduce the size of the graphs to which the algorithm is applied). These approaches are computationally efficient and have good precision and recall rates. Their limit is that they miss to detect pattern variants of similar design patterns. Furthermore, they are limited only to the patterns represented by matrices, thus it is not suitable to be easily extended.

Finally, there are some well known techniques that cannot be classified in the above categories (e.g. fuzzy reasoning, bit vector compression, minimum key structure method, predicate and rho calculus, dynamic analysis using run-time execution traces, formal methods based on semantic, machine learning based approaches and concept analysis) but that are good as a complement to improve the structural methods cited above.

```
pattern observer {
        type AS(1) {
                has method A,R;
                has method N;
                has container o of type AO;
        }
        type AO(1) {
                has method U;
        }
        type CO(*) {
                inherits-from AO;
        }
        type CS(*) {
                inherits-from AS;
                has constructor c {
                     object-creation o;
                }
                overrides methods [A,R] each {
                        delegates to o;
                }
                overrides method N each {
                        delegates to o;
                        calls U in AO.U;
                }
         }
}
```

Fig. 2. An Example of DSL instance: the Observer Pattern specification

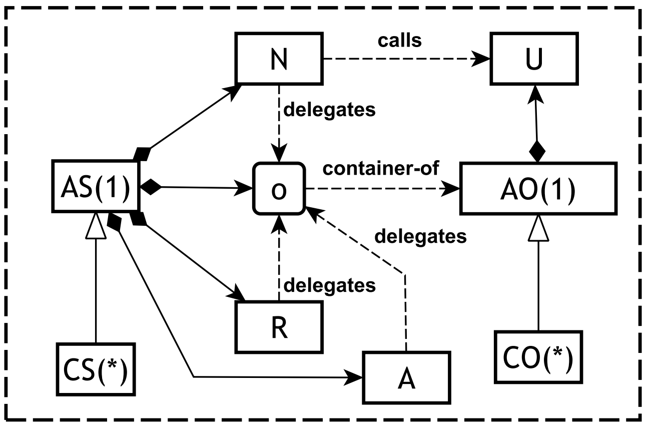

Fig. 3. An excerpt of the graph related to the DSL for the Observer Pattern

In [19] a tool for design pattern detection and software architecture reconstruction is proposed. The tool uses a mixed structural and metric approach to detect pattern instances.

Recent studies have been also focused on the formalization of empirical evaluation criteria [18, 17, 20]. Each applied technique should be evaluated using well defined criteria and different authors have proposed taxonomies and related frameworks to perform such evaluations.

The proposed approach is based on a system metamodel that is able to represent elements down to statements and expressions. This allows to reason about behavioral properties that can be used (*i*) to improve search space reduction and (*ii*) to distinguish between patterns that have the same structure but behaves (or are used) in different ways [2].

III. The Meta-model and DSL

The approach defines and exploits a meta-model and a Domain Specific Language (DSL) to model the structure of both the software system and the design patterns to be detected.

The Figure 1 depicts the defined meta-model as a class diagram. The system structure, shown in the upper part of Figure 1, is modeled as a set of Types (i.e., Container, Value, Reference, and Compound Types[1]) along with their relationships. A Design Pattern (the bottom part of the diagram) is modeled as a set of Properties characterising its structure and behaviour. The *Classifier* Property introduces a Reference Type (Class or Interface) in a pattern specification (or overrides its definition). It allows to model a pattern role specifying its structure and relationships with other types. It can be used to define constraints on the role super-type or its implemented interfaces. The *Data* Property models a new field in an existing reference type (or override one of existing fields). The specification of a Data property must provide the type of the new field, or the base type for compound types (like in the case of an array or a generic List). The *Behavioral* Property defines, or overrides, a method in an existing Classifier. The definition includes the specification of return type and arguments. This property is used to define required (or optional) behaviors of a type in a design pattern specification. The *Dependency* Property describes a generic use dependency between pattern elements. An example is the dependency between a method and a field declared in its body definition. The *Invocation* Property models a call between methods of reference types defined in the pattern specification. The *Delegation* Property models a mapping between a set of methods of a Class and a set of methods of any existing reference type defined in the pattern specification. This allows to take into account the delegation for the patterns that require it. Finally the *Object Creation* Property models the object creation specifying the method or the scope in which an object needs to be created and the type of the created object. This property is used to model patterns expressing mandatory object instantiations (it is needed in almost all creational patterns).

[1]Compound types are treated as separated types since they must specify the base type of compound. In this class are also arrays and generic types.

Each pattern, in order to be detected, needs to be modeled writing a pattern specification using a DSL, defined according to the meta-model of Figure 1. The defined DSL takes inspiration from many languages and systems proposed in the past, namely the SDF, Crocopat and Grok [21, 22, 23]. The main goals driving the definition of DSL are briefly summarized below:

- a specification should be writable by the analyst with reduced effort;
- the DSL should allow to express constraints on source code structure and behavior to model complex DPs;

- the DSL should support the definition of pattern variants (using inheritance among specifications) to foster reuse;

Since the aim of this paper is to present the detection approach, the complete definition of DSL syntax is not covered here (however more details and pattern specifications are available at [4]).

As an example how a pattern is modeled using the DSL, let us consider a classic Observer DP (supporting only a single kind of event for each notify method) as proposed in literature [24]. The Figure 2 shows the DSL specification for such an Observer DP. Each specification is just a sequence of *type* blocks: each block specifies the set of properties that must hold for a role in the pattern (including the constraints on the allowed multiplicity, reported in the brackets just after the type name - if no brackets follow the type name the default multiplicity is 1).

As shown in the Figure 2, the Observer specification requires:

- a single AbstractObserver (AO) and several ConcreteObservers (CO);
- a single AbstractSubject (AS) and several ConcreteSubjects (CS);
- a container of AbstractObservers to be defined in the ConcreteSubject (the field “o”);
- the methods *A* and *R* (that play roles of add and remove) to be defined in the AbstractSubject and overridden in ConcreteSubjects;
- a Delegation to be defined between A and R of ConcreteSubject and the add/remove methods Container type;
- the notify method (called “N”); the method N must contain an invocation towards the update method U of the AbstractObserver classifier;
- an object creation (to initialize the container field “o”) in the constructor of the ConcreteSubject type.

Each specification can be translated to a graph in which elements are nodes and properties are labelled edges. This graph, as better explained in Section IV, is part of the input for a two-pass graph-matching detection algorithm. The Figure 3 shows an excerpt of the graph representing the Observer as described in the specification of Figure 2. The graph in Figure 3 reports key elements specified in the DSL together with the relationships among them [2]. A variant of this Observer can be defined, easily, deriving it from the DSL specification of Figure 2. Structural elements that need to be changed can be overridden. For instance, Figure 6 shows the graph of a common multi-event Observer. This variant redefines the elements “N” and “U” as sets of methods to take into account different kinds of events and notification handlers.

[2]To keep figure concise, each node/edge is labelled with the initial letter of the corresponding field or method in the DSL. Moreover not all properties are represented, as for type CS which has several overrides and a constructor that are all omitted.

IV. The Detection Process

The pattern detection process comprises the following main steps:

- **definition of the patterns specifications repository**: each specification is written according to the proposed DSL and organized in a catalog stored into a repository.
- **pattern Models instantiation**: the DSL specifications are parsed to generate the Design Pattern Graphs (DPGs) to be detected.
- **system source code analysis**: the source code of the system under study is parsed and the complete ASTs of the system are produced.
- **generation of an instance of the system model**: a traversal of the system ASTs is performed to generate an instance of the system model, also represented as a graph (called the System Graph - S). Rapid type analysis (RTA), class flattening and inlining of not public methods are exploited in order to build a system’s representation suitable for the matching algorithm[3].
- **design patterns matching**: S is traversed and a matching algorithm is performed to identify implemented patterns. During the detection each pattern instance is mapped to the corresponding matching design pattern graph.

A DPG can be regarded as an attributed graph specifying a set of predicates on the attributes that must hold: a DPG is a pair $DP = (P, AC)$, where P is an attributed graph and AC is a set of predicates on the attributes. When a DPG is matched onto a System Graph, the binding between them can be used to access the sub-graph on the system (either the sub-graph structure or attributes and properties on nodes and edges).

The specification expressed as a DPG is rewritten by means of a set of predicates AC_u and AC_e on, respectively, individual nodes and edges. For each node u in the pattern DP, there is a set of candidate matched nodes in S for which the constraints AC_u hold. These nodes define a partially matched DPG referred as *candidate neighborhood* of node u and denoted by $\phi(u)$.

The search space of a DPG on a system graph S is defined by the candidate neighborhood in S of all DPG nodes. It corresponds to the Cartesian product of the candidate neighborhood for each DPG node: $\phi(u_1), \times \ldots \times, \phi(u_k) \in S$, where $u_1, \ldots, u_k \in DP$.

The Figure 4 outlines the identification algorithm. It is executed in two phases. The first one starts at line 5 by calling (line 7) the *forwardNeighborhoodAnalysis* function (lines 13-21) which computes the candidate neighborhood for each node u_h in the design pattern graph. This function ends after performing a pruning step of the search

[3]Note that RTA is used to handle late binding and hence the computed call graph reports a super-set of the real calls that can be executed at run-time. This however only lowers the precision in very few cases. A discussion on the impact on the detection quality is however reported in threats to validity section.

```
 1
 2
 3  List<MatchedGraph> instances = ...;
 4
 5  void start()
 6  begin
 7   forwardNeighborhoodAnalysis(DP)
 8   for i = 1 to k do
 9    Match(i);
10   end
11  end
12
13  void forwardNeighborhoodAnalysis(DP)
14  begin
15   foreach node u in DP do
16      $\phi(u)$ = { v in V (S) | $AC_u$ (v) = true }
17      (1) computation $\phi(u)$
18      (2) reduce $\phi(u_1) \ldots \phi(u_k)$ using
19          lookahead and properties constraint
20   end
21  end
22
23  void Match(i)
24  begin
25   foreach v in $\phi(u_i)$ | v is free do
26     if not checkNeighborhoodBindings($u_i$ ,v)
27       then continue;
28     $\phi(u_i)$ = v;
29     if i < |V(DP)| then Match(i + 1);
30       else
31       if $AC_\phi$ (S) then
32         instances.add($\phi()$);
33       end
34  end
35
36  boolean checkNeighborhoodBindings($u_i$ , v)
37  begin
38       foreach edge e($u_i$ , $u_j$ ) in E(DP), j < i do
39        if (edge $e_1(v, \phi(u_j))$ not in E(S))
40            or (not $AC_e(e_1)$) then
41            return false;
42        end
43       return true;
44  end
```

Fig. 4. A sketch of the detection algorithm

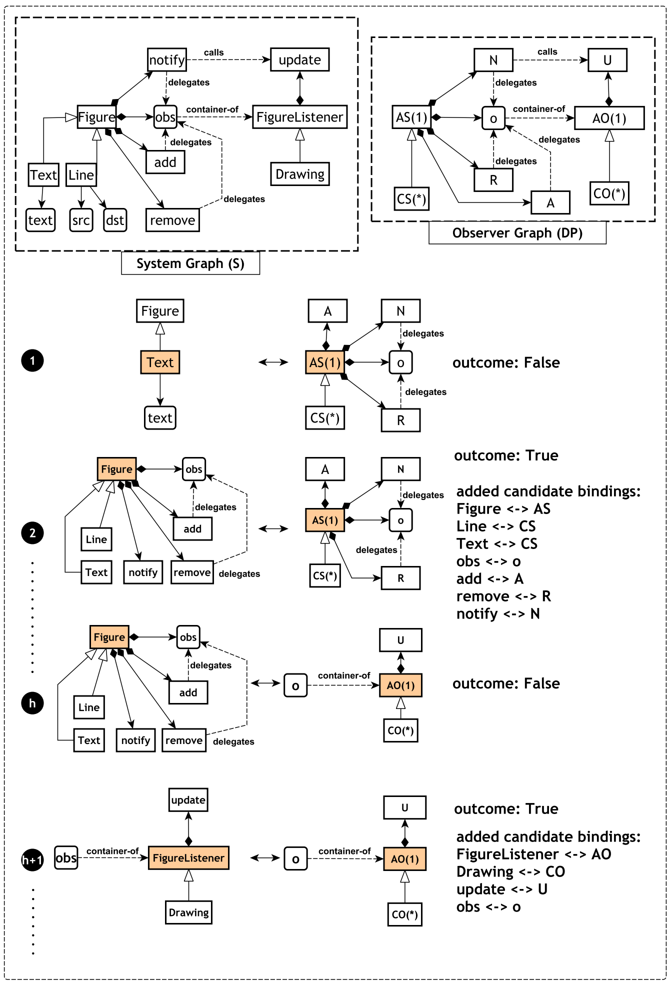

Fig. 5. A running example showing the candidate neighborhood analysis

space (better described in the following). The second phase (on lines 23-44) performs a search, over the search space $\phi(u_1) \times \ldots \times \phi(u_k)$, using a depth-first traversal, to find a sub-graph isomorphisms. The $Match(i)$ function iterates on the i_{th} node to find valid bindings for that node. Procedure $checkNeighborhoodBindings(ui, v)$ examines if u_i can be mapped to v by considering their edges and attributes. Line 28 maps the node u_i to v. Lines 29-33 continue to search for the next node or, when the last node is reached, evaluate the predicate AC to check constraints. If it is true, then a valid binding $\phi : V(DP) \rightarrow V(S)$ has been found and is added to the list $\phi()$(line 32). Since the worst-case complexity of the matching algorithm is $O(n^k)$, where $n = |S|$ and $k = |P|$, to make the algorithm usable on real systems, a search space reduction technique must be used. Our approach uses system and pattern information to reduce the size of search space. The reduction step exploits a look-ahead requiring, for each node u_i of the DPG, a valid (partial) binding of the neighborhood sub-graph centered in u_i at a fixed distance r from it.

For each candidate neighborhood, structural information (e.g. nodes and edges) and predicates on attributes (types and properties of nodes and edges) are used to prune matches that would not produce acceptable solutions. This neighborhood knowledge can be exploited to prune infeasible sub-graph at an early stage and obtain a reduced set of candidates on which to perform the full depth first matching (that is resource- and time- expensive). There is a trade-off with respect to how candidate neighborhood sub-graphs are built. They increase pruning power as the look-ahead increases, but their construction is of polynomial complexity (with respect to look-ahead). Our current implementation uses a look-ahead equals to 1 (immediate neighborhood). We found no improvement with a look-ahead equals to 2 since even if for some patterns (those with a rich structure or highly constrained) time was greatly reduced, for others the wider neighborhood analysis increased the total time (i.e., the average time remained almost the same).

The Figure 5 reports a simple example of the detection process. It shows the candidate neighborhood analysis, related to the pattern specification of Figure 2, performed on some nodes of the Observer DPG on a small S (represented in the Figure 5 respectively by DP and S) and the resulting bindings. For each pair of nodes $u_i \in V(DP)$ and $v_j \in V(S)$, the neighborhood sub-graphs of u_i and v_j are matched to find the candidate neighborhood. The Figure 5 just reports few interesting cases. Step 1 considers the pair of nodes (Text, AS) and hence the immediate neighborhoods of respectively DP and S are considered. In this case the match fails since DPG neighborhood is not a sub-graph of S. The step 2 on the pair (AS, Figure) results in a successful match since AS neighborhood is congruent with the one of Figure and all the constraints are satisfied (nodes and edges are of the same types, and multiplicity constraints are met). Hence several conditioned bindings are established for the matched candidate neighborhood. The step h is for the pair (Figure, AO). Due to structural differences this match fails (correctly) avoiding an infeasible candidate neighborhood (since the Figure is not an AbstractObserver). This is because the structure of the Observer DP has a quite good pruning power. Simpler patterns may generate a higher number of candidate neighborhoods that must be taken into account in the second phase of the algorithm in which the full depth-first matching performed increases time and space requirements. Hence, to further reduce the search space, the algorithm is executed on all the DPGs in the specification repository and the previous bindings are taken into account when performing the subsequent candidate neighborhood analysis.

For some pattern model, the same element could be bound, in the general case, to several patterns. This however is not true for all pattern elements. For instance, the binding of the visit() for the Visitor pattern method should not allow bindings of another patterns (e.g. like the execute() method of a Command or the notify() method of an Observer). When a pattern model explicitly forbids multiple bindings for a pattern element, existing established bindings of already analyzed patterns are used as further constraints to improve the search space reduction (pruning infeasible bindings as early as possible).

Variants are handled in the same way as other specifications, with no special treatment within the detection process. The only difference regards how their sub-graphs are built (taking into account the specification inheritance relationships and using a flattening approach). The resulting variant graph contains both the properties inherited from their super-specification and the overridden ones.

V. Case Study

To validate the correctness, effectiveness and efficiency of the proposed approach we applied it to several OO systems. A first group contains eleven open source java software systems of increasing sizes from the publicly available benchmarks proposed in [5, 6]. For these systems we considered the benchmark as the gold standard to evaluate precision and recall. In this paper, due to space constraints, the discussion is limited to the selection of the first six systems reported in the first column of Table I. The second group contains two larger systems (JHotdraw 7 and Apache Avro 1.6.x) selected to perform a direct comparison with a similar design patterns mining tool (DPD, version 4.5) proposed by Tsantalis in [2].

TABLE I
Analyzed systems characteristics

System Name	Version	Size (KLOC)	#Types	#Methods
Junit	3.7	4,9K	104	648
Lexi	0.1	7,1K	100	677
JHotDraw	5.1	8,9K	174	1316
QuickUML	2.1	9,2K	230	1082
Nutch	0.4	23,6K	335	1854
PMD	1.8	41,5K	519	3665
JHotDraw	7.0	78,5K	567	4185
Apache Avro	1.6	125,2K	1195	8451

Table I reports the main structural characteristics of the systems. Systems, in both groups, were chosen of increasing sizes to evaluate the scalability of the algorithm and to validate the quality of results on a large code base.

The DPs considered in this case study are the most common ones proposed in literature (e.g., Factory Method, Prototype, Singleton, Adapter, State, Strategy, Composite, Decorator, Observer, Template Method, Memento, Bridge, Command, Proxy, and Visitors), whose DSL specifications were stored in the catalog repository. In particular, the catalog used in the case study consisted of 21 DSL specifications related to 18 different DPs. The validation was performed using the DPF tool developed to support the approach. It was implemented as a set of Eclipse plugins based upon JDT (to extract information on the systems' static structure down to code statement level), and upon the EMF framework (to implement the metamodel).

According to [9], design pattern recovery techniques can be evaluated by computing precision and recall [25], in order to assess their effectiveness and correctness. To compute recall and precision we assume that a pattern instance can be classified into one of four categories:

- true-positive (T_P: correctly found),
- false-positive (F_P: incorrectly found),
- true-negative (T_N: correctly missed),
- false-negative (F_N: incorrectly missed).

On that base, precision is defined as the ratio of correctly found occurrences to occurrences provided by the tool and is given by:

$$Precision = T_P/(T_P + F_P) \tag{1}$$

Recall is the ratio of correctly found occurrences to all correct occurrences and is given by:

$$Recall = T_P/(T_P + F_N). \tag{2}$$

The Gold Standard (GS) used as reference is the set of all true positive instances. As said it was computed in a different way for the two systems groups.

For the first group, it was used the open benchmark provided in [5, 6]. We assumed it to be correct and complete. More information on the results of our findings are discussed in the following of the section.

For JHotDraw 7 and Apache Avro 1.6, the GS was computed using the correct results produced by both DPF and DPD tools. Hence it could lack pattern instances missed by both tools (overestimating recall) but allows to perform a direct (and reliable) comparison on precision.

A. *Results on Benchmark*

The study was carried on by applying DPF to find the DPs contained in the first group of six java software systems reported in Table I.

Table II, for each of the analyzed systems, reports: the name of the DPs searched in the code (first column); the number of true positive instances as provided by the benchmark (column labelled by GS); the number of each searched pattern detected by the proposed approach (column D); the number of true positive found by DPF (column Tp), the number of false positive found by DPF (column Fp), i.e. the number of DPs instances detected by DPF but not present in the benchmark. The last two columns report respectively precision (P) and recall (R) computed on the results as provided by DPF (using the benchmark as the gold standard).

As shown in Table II, the values for precision are greater than 0.8 with an average value of 0.89, while the values for recall are greater than 0.75 with an average value of 0.93. These values are also consistent for increasing system sizes. Patterns like Command, Composite, Observer but also Visitor (that is based on double dispatch) are more precisely identified since their specifications include both static and behavioral relationships. This is confirmed by the number of false positives that is lower than patterns with a less constrained structure or with limited or absent behavioral properties. This, as highlighted in the Section IV, is due to the higher pruning power of complex DPGs with respect to the simpler ones.

As results show, the proposed approach is able (depending on how specifications are written) to distinguish among patterns that have the same static structure but different behaviors. For example, for the Command pattern, in order to distinguish it from the Adapter one (the object version), the specification uses the invocation property requiring the *execute* method, in the concrete subclass, to invoke a method of a class bound to a Command. The same happens for Composite and Decorator, where the Decorator is required to specify a delegation towards the decorated object.

TABLE II
DPF RESULTS ON THE OPEN BENCHMARK

System→	QuickUML 2001					
↓Design Pattern	GS	D	T_P	F_P	P	R
Abstract Factory	16	17	16	1	0,94	1
Builder	12	11	10	1	0,91	0,83
Command	6	6	5	1	0,83	0,83
Composite	29	28	27	1	0,96	0,93
Observer	17	18	17	1	0,94	1
Singleton/spec{gof-relaxed}	1	1	1	0	1	1
Singleton	1	1	1	0	1	1
System→	Lexi v0.1.1 alpha					
↓Design Pattern	GS	D	T_P	F_P	P	R
Builder	5	6	5	1	0,83	1
Observer	11	10	9	1	0,9	0,82
Singleton/spec{gof-relaxed}	2	3	2	1	0,67	1
Singleton	2	2	2	0	1	1
System→	JUnit v3.7					
↓Design Pattern	GS	D	T_P	F_P	P	R
Composite	39	34	33	1	0,97	0,85
Decorator	37	31	30	1	0,97	0,81
Iterator	6	6	5	1	0,83	0,83
Observer	14	12	11	1	0,92	0,79
Singleton/spec{gof-relaxed}	2	3	2	1	0,67	1
Singleton	2	3	2	1	0,67	1
System→	JHotDraw v5.1					
↓Design Pattern	GS	D	T_P	F_P	P	R
Adapter	32	29	27	2	0,93	0,84
Command	25	25	22	3	0,88	0,88
Composite	30	32	29	3	0,91	0,97
Decorator	23	23	20	3	0,87	0,87
Factory Method	55	52	49	3	0,94	0,89
Observer	29	32	29	3	0,91	1
Prototype	26	25	22	3	0,88	0,85
Singleton/spec{gof-relaxed}	2	7	2	5	0,29	1
Singleton	2	2	2	0	1	1
State	18	25	18	7	0,72	1
Strategy	34	41	34	7	0,83	1
Template Method	24	29	22	7	0,76	0,92
System→	Nutch v0.4					
↓Design Pattern	GS	D	T_P	F_P	P	R
Adapter	7	8	7	1	0,88	1
Bridge	25	22	21	1	0,95	0,84
Command	14	15	14	1	0,93	1
Iterator	7	7	6	1	0,86	0,86
Memento	15	15	14	1	0,93	0,93
Singleton	1	1	1	0	1	1
Strategy	8	9	8	1	0,89	1
Template Method	14	15	14	1	0,93	1
System→	PMD v1.8					
↓Design Pattern	GS	D	T_P	F_P	P	R
Adapter	4	5	4	1	0,8	1
Builder	12	10	9	1	0,9	0,75
Composite	8	8	7	1	0,88	0,88
Factory Method	15	15	14	1	0,93	0,93
Iterator	4	5	4	1	0,8	1
Observer	11	12	11	1	0,92	1
Proxy	3	4	3	1	0,75	1
Template Method	2	3	2	1	0,67	1
Visitor	5	5	5	0	1	1

Variant Analysis: The main goals in supporting variants of existing pattern specifications is to improve both detection quality and flexibility of the approach (to make it easily adaptable for new patterns). We found that the number of false negatives was dramatically reduced by adding new variants inheriting existing specifications and taking into account the structural differences that caused the tool to miss them. The false negatives ($GS - T_p$ in Table II) were related to patterns implemented differently from what assumed in the specification (our catalog is, for the most part, based on the definitions given in literature [24, 26]). An example is the case of Singleton pattern for JHotDraw v5.1. DPF was configured with two specifications (for convenience separately shown in Table II). The first one was related to the Singleton definition given in literature [24], implemented with a final class, a private constructor and a public static getter method. A relaxed specification, called "relaxed-gof", removes the private constructor and final class constraints. With these changes the number of detected Singleton instances changes from two to seven, lowering the precision. However, the five false positives could be considered valid Singleton instances since they only violate one Singleton property and are used as Singleton in the system (in this case precision and recall became both one). The same happens in the Lexi system for which an instance of Singleton has a public constructor and was not considered in the benchmark due to the strict specification used to identify it.

B. A Comparion with DPD Tool

For the last two systems of Table I, a comparison between results obtained by DPF and those obtained using the similarity scoring approach proposed by Tsantalis [2] was performed. Tsantalis' tool (Design Pattern Detector Tool - DPD) was chosen mainly because it adopts a similar technique (exploiting a subset of the same information, even if used in a different way). Table III, for JHotDraw 7 and Apache Avro 1.6 systems reports the name of the detected DPs (first column) and the number of pattern considered as gold standard (GS). The remaining columns in the table, for each tool, reports the number of detected (D), true positives (Tp), false positives (Fp) and the values of Precision (P) and Recall (R).

The first consideration about the results is related to the presence of false positive and false negative. The ratio of false positive is less than 0.4% and 2% for respectively DPF and Tsantalis that is quite acceptable for both tools. However for some patterns, and for both the approaches, the number of false positive is particularly higher than for other patterns. This happens, for DPF on JHotDraw 7, for Template Method pattern: in this case, of 110 instances, 10 instances were not template methods. Inspecting those cases revealed a bug in the pattern specification. Although correct, it was too relaxed and this caused some internal (private) helper methods to be considered as Template Methods. For the Observer pattern the results were similar, since the approach detected 105 observers instances (one for each concrete participant) but 9 of them were not Observers.

The case of Observer design pattern is also interesting for what concerns the detection of pattern variants. DPF detected

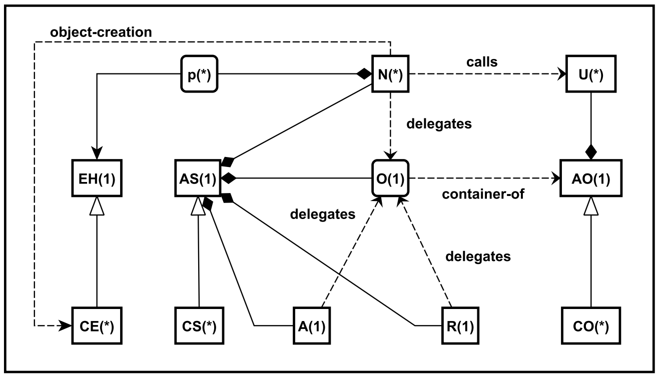

Fig. 6. An excerpt of the DPG of the multi-event Observer variant

96 true Observer instances on JHotDraw 7 (with 9 false positives) and 48 instances on Apache Avro 1.6 (no false positives found). In this analysis, our pattern specification repository was comprised of 3 variants for the Observer pattern. The first one exploits standard Java types (Observable class and Listener interface) while the other requires an abstract type for both Subject and Observer roles. The third one defines a multi-event observer in which the event context is created by a private helper method (fire$\langle Event \rangle$ methods) and passed to the corresponding callback on the listener (taking the context as parameter). The structure of this specification, reported in Figure 6, is quite different from the one proposed in literature. Inspecting the matched instances we found that, for the both JHotDraw 7 and Apache Avro 1.6 systems, the 96 and 48 instances respectively were all variants of the third type. This also explains why DPD Tool, that is based on the classic variant, was not able to find observers on these two systems.

C. Performance Issues

When running the DPF tool, we have measured the execution times for each step of the detection process.

Table IV reports the measured values for each of the analyzed systems.

The pattern matching step is the most CPU time consuming. We cannot show detection times for each pattern since our approach uses the successful identifications across pattern specifications as constraints to improve the performance and hence the detection times are dependent on this. However, we calculated the average time to detect a single pattern and it resulted to be comparable to the other structural approaches. The total times in Table IV, show that DPF exhibits a better scalability with respect to DPD Tool.

Experimentation performed for tuning the patterns specifications, showed that performances can be considerably improved by identifying structural and behavioral constraints that are effective at identifying a well defined variant of a pattern. Hence the approach is more effective when specifications are

TABLE III
PRECISION AND RECALL FOR JHOTDRAW 7 AND APACHE AVRO 1.6

System→	JHotDraw 7											Apache Avro 1.6										
Tool→		Design Pattern Finder					DPD Tool						Design Pattern Finder					DPD Tool				
↓Design Pattern	GS	D	T_P	F_P	P	R	D	T_P	F_P	P	R	GS	D	T_P	F_P	P	R	D	T_P	F_P	P	R
Observer	96	105	96	9	0.91	1	0	0	0	0	0	48	48	48	0	1	1	0	0	0	0	0
Singleton	32	32	32	0	1	1	31	30	1	0.96	0.93	27	27	26	1	0.96	0.96	21	18	3	0.85	0.66
Factory Method	20	18	18	0	1	0.9	4	2	2	0.5	0.1	12	12	12	0	1	1	1	1	0	1	0.1
Template Method	110	110	100	10	0.9	0.9	16	10	6	0.625	0.09	32	28	28	0	1	0.875	13	9	4	0.69	0.28
Adapter/Command	155	123	120	3	0.97	0.77	79	71	8	0.89	0.45	33	30	30	0	1	0.90	12	9	3	0.75	0.27
Decorator	6	6	6	0	1	1	4	4	0	1	0.67	6	5	5	0	1	0.83	6	6	0	1	1
Prototype	46	34	31	3	0,91	0,67	113	40	73	0,35	0,86	5	7	5	2	0.71	1	0	0	0	0	0
State Strategy	194	168	165	3	0.92	0.85	213	171	42	0.80	0.88	116	137	114	23	0.82	0.98	18	18	0	1	0.15
Composite	6	5	5	0	1	0.83	6	6	0	1	1	0	—	—	—	—	—	—	—	—	—	—

structured in a hierarchy and each specification is dedicated to specific pattern variants.

TABLE IV
EXECUTION TIMES OF THE DESIGN PATTERNS DETECTION PROCESS

Tool→	DPD		DPF	
System→	JhotDraw7	Avro1.6	JhotDraw7	Avro1.6
Step↓	Times (s) ↓			
Parsing&AST extraction	-	-	69.97	255.23
Metamodel generation	-	-	428.15	1567.78
Pattern repository detection	-	-	627.09	2296.27
Total Time	1281.50	6320.81	1125.21	4120.29
Average per pattern	142.39	702.31	125.02	457.81

D. Threats to Validity

This section discusses the main threats to the validity of our study. Construct validity threats concern the relationship between theory and observation. There could be imprecision/omissions in the measurements made in this paper for several reasons. One of the most important limitations regards the generation of behavioral properties in presence of late binding. In this case, as already stated, we have built a call graph using Rapid Type Analysis (RTA) to reduce the set of possible callers. However the call graph still contains a super-set of the actual calls. In the property extraction algorithm we decided to take into account the sets of all possible targets. In this way, we surely do not miss any possible binding but this exposes the algorithm to the presence of false positives (since the set of successful binding can be a super-set of the actual ones). Further experimentation (with more strict policies) should be performed in order to assess if the behavior of the algorithm improves with respect to this conservative choice. Conclusion validity concerns the relationship between the treatment and the outcome. As explained, we performed a comparison with six systems of an open benchmark assuming it to be correct and complete. Hence our results are still exposed to bias and human mistakes or subject to interpretation but, being the benchmark publicly available and evolved over several years, these effects should be limited.

For Apache Avro and JHotDraw7 systems, the gold standard was computed comparing results obtained using Design Pattern Finder and DPD tool. This means that, for JhotDraw 7 and Apache Avro 1.6 systems, we cannot exclude that the computation of recall is imprecise (it could be higher than the actual one since there could exist pattern instances missed by both tools used in the study to compute the gold standard). This, in future works, can be improved by performing a full analysis of the searched source code base (or using available benchmarks also for tool comparison). Threats to internal validity concern factors that can influence our observations. In this case the identification of pattern instances was based on the expert examination of internal/external documentation and source code and hence could pose a threat to internal validity affecting the number of false negatives. Threats to external validity concern the generalization of our findings. Of course replication on further projects to confirm or contradicts the obtained results is always desirable. Moreover we cannot claim that our approach produces the same results on different (and larger) systems. Rather, we provide quantitative information on the quality of the search for several real world systems and can affirm that precision and recall have remained consistent and independent with respect to the system size. On the performance side there is a high dependency of the overall detection performance on the quality of pattern specifications. When specifications are badly written (that means few and overlapping constraints) the performance of the algorithm degrades rapidly.

VI. CONCLUSIONS AND FUTURE WORK

An approach to detect design patterns in OO systems has been presented and discussed. The approach exploits a metamodel (and a derived DSL) defined to represent both the pattern and the system under study. The detection process is carried out matching each design pattern model with the system model in order to detect pattern instances. To improve both flexibility and quality of detection, the approach allows to specify a pattern variant by overriding an already defined pattern specification. For each detected pattern instance, the system source code elements participating in its implementation are identified and assigned to the role they plays in the pattern specification. The approach has been applied to six open-source OO systems from an open benchmark. For the proposed approach, the average values of precision and recall, evaluated using the benchmark as gold standard, are satisfactory (respectively 0.89 and 0.93), independently of system size. For two additional systems the results provided by DPF were compared with the ones obtained from a similar approach existing in literature (DPD tool proposed in [2]). On these systems, results show that DPF performs better and is more efficient than the DPD tool. As future work, more open source systems and tools will be considered. Future work also involves the improvement of the metamodel (and the related DSL) in order to consider a wider set of properties to model more complex design and architectural patterns.

REFERENCES

[1] M. P. L. Prechelt, B. Unger-Lamprecht and W. Tichy, "Two controlled experiments assessing the usefulness of design pattern documentation in program maintenance," *IEEE Trans. Softw. Eng.*, vol. 28, no. 6, pp. 595–606, 2002.

[2] N. Tsantalis, A. Chatzigeorgiou, G. Stephanides, and S. T. Halkidis, "Design pattern detection using similarity scoring," *IEEE Trans. Softw. Eng.*, vol. 32, no. 11, pp. 896–909, Nov. 2006.

[3] J. M. Smith and D. Stotts, "Spqr: flexible automated design pattern extraction from source code," in *Proc. 18th IEEE International Conference on Automated Software Engineering*, Oct. 6–10, 2003, pp. 215–224.

[4] M. L. Bernardi, M. Cimitile, and G. Di Lucca, "Model driven design pattern mining using graph-matching heuristic approach," in *Software Engineering Technical Report - RCOST, University of Sannio, Benevento, 2012*, http://www.rcost.unisannio.it/ingsoft/dpf.

[5] Y. G. Guéhéneuc, "P-mart: Pattern-like micro architecture repository," 2007.

[6] F. Arcelli Fontana, A. Caracciolo, and M. Zanoni, "Dpb: A benchmark for design pattern detection tools," in *Software Maintenance and Reengineering (CSMR), 2012 16th European Conference on*, 2012, pp. 235–244.

[7] M. L. Bernardi and G. A. Di Lucca, "Model-driven detection of design patterns," in *Proceedings of the 26th IEEE International Conference on Software Maintenance*, ser. ICSM '10, September 12-18, Timioara, Romania, 2010.

[8] J. Dong, Y. Zhao, and T. Peng, "Architecture and design pattern discovery techniques - a review," in *Software Engineering Research and Practice*, H. R. Arabnia and H. Reza, Eds. CSREA Press, 2007, pp. 621–627.

[9] G. Rasool and D. Streitfdert, "A survey on design pattern recovery techniques," *IJCSI International Journal of Computer Science Issues*, vol. 8, no. 2, pp. 251 – 260, 2011.

[10] G. Rasool, I. Philippow, and P. Mäder, "Design pattern recovery based on annotations," *Adv. Eng. Softw.*, vol. 41, no. 4, pp. 519–526, Apr. 2010.

[11] K. Stencel and P. Wegrzynowicz, "Detection of diverse design pattern variants," in *Proceedings of the 15th Asia-Pacific Software Engineering Conference*, ser. APSEC '08. Washington, DC, USA: IEEE Computer Society, 2008, pp. 25–32.

[12] H. Lee, H. Youn, and E. Lee, "Automatic detection of design pattern for reverse engineering," in *Proceedings of the 5th ACIS International Conference on Software Engineering Research, Management & Applications*, ser. SERA '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 577–583.

[13] M. von Detten and S. Becker, "Combining clustering and pattern detection for the reengineering of component-based software systems," in *Proceedings of the joint ACM SIGSOFT conference – QoSA and ACM SIGSOFT symposium – ISARCS on Quality of software architectures – QoSA and architecting critical systems – ISARCS*, ser. QoSA-ISARCS '11. New York, NY, USA: ACM, 2011, pp. 23–32.

[14] J. Paakki, A. Karhinen, J. Gustafsson, L. Nenonen, and A. I. Verkamo, "Software metrics by architectural pattern mining," in *in Proceedings of the International Conference on Software: Theory and Practice (16th IFIP World Computer Congress*, 2000, pp. 325–332.

[15] G. Antoniol, R. Fiutem, and L. Cristoforetti, "Design pattern recovery in object-oriented software," in *Proceedings of the 6th International Workshop on Program Comprehension*, ser. IWPC '98. Washington, DC, USA: IEEE Computer Society, 1998, pp. 153–.

[16] Y. G. Guéhéneuc, J. Y. Guyomarc'H, and H. Sahraoui, "Improving design-pattern identification: a new approach and an exploratory study," *Software Quality Control*, vol. 18, no. 1, pp. 145–174, Mar. 2010.

[17] A. De Lucia, V. Deufemia, C. Gravino, and M. Risi, "Design pattern recovery through visual language parsing and source code analysis," *Journal of Systems and Software*, vol. 82, no. 7, pp. 1177 – 1193, 2009.

[18] J. Dong, Y. Zhao, and Y. Sun, "A matrix-based approach to recovering design patterns," *Trans. Sys. Man Cyber. Part A*, vol. 39, no. 6, pp. 1271–1282, Nov. 2009.

[19] F. Arcelli and M. Zanoni, "A tool for design pattern detection and software architecture reconstruction," *Inf. Sci.*, vol. 181, no. 7, pp. 1306–1324, Apr. 2011. [Online]. Available: http://dx.doi.org/10.1016/j.ins.2010.12.002

[20] P. Tonella, M. Torchiano, B. Du Bois, and T. Systä, "Empirical studies in reverse engineering: state of the art and future trends," *Empirical Softw. Engg.*, vol. 12, no. 5, pp. 551–571, Oct. 2007.

[21] J. Heering, P. R. H. Hendriks, P. Klint, and J. Rekers, "The syntax definition formalism sdf reference manual," *SIGPLAN Not.*, vol. 24, no. 11, pp. 43–75, Nov. 1989.

[22] D. Beyer, "Relational programming with crocopat," in *Proceedings of the 28th international conference on Software engineering*, ser. ICSE '06. New York, NY, USA: ACM, 2006, pp. 807–810.

[23] M. Goldstein and D. Moshkovich, "System grokking: a novel approach for software understanding, validation, and evolution," in *Proceedings of the 7th international conference on Next generation information technologies and systems*, ser. NGITS'09. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 38–49.

[24] R. J. J. V. E. Gamma, R. Helm, *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, 1998.

[25] N. Pettersson, W. Lowe, and J. Nivre, "Evaluation of accuracy in design pattern occurrence detection," *IEEE Trans. Softw. Eng.*, vol. 36, no. 4, pp. 575–590, Jul. 2010.

[26] Y. G. Gueheneuc and G. Antoniol, "Demima: A multilayered approach for design pattern identification," *IEEE Transactions on Software Engineering*, vol. 34, no. 5, pp. 667–684, 2008.

[27] J. Niere, W. Schäfer, J. P. Wadsack, L. Wendehals, and J. Welsh, "Towards pattern-based design recovery," in *ICSE '02: Proceedings of the 24th International Conference on Software Engineering*. New York, NY, USA: ACM, 2002, pp. 338–348.