DOI: 10.1145/3176643

A

Detecting the Behavior of Design Patterns through Model Checking and Dynamic Analysis

ANDREA DE LUCIA, University of Salerno
VINCENZO DEUFEMIA, University of Salerno
CARMINE GRAVINO, University of Salerno
MICHELE RISI, University of Salerno

We present a method and tool (ePAD) for the detection of design pattern instances in source code. The approach combines static analysis, based on visual language parsing and model checking, and dynamic analysis, based on source code instrumentation. Visual language parsing and static source code analysis identify candidate instances satisfying the structural properties of design patterns. Successively, model checking statically verifies the behavioral aspects of the candidates recovered in the previous phase. We encode the sequence of messages characterizing the correct behaviour of a pattern as LTL (Linear Temporal Logic) formulae and the sequence diagram representing the possible interaction traces among the objects involved in the candidates as Promela specifications. The model checker SPIN verifies which candidates satisfy the LTL formulae. Dynamic analysis is then performed on the obtained candidates by instrumenting the source code and monitoring those instances at run-time through the execution of test cases automatically generated using a search-based approach. The effectiveness of ePAD has been evaluated by detecting instances of twelve creational and behavioral patterns from six publicly available systems. The results reveal that ePAD outperforms other approaches by recovering more actual instances. Furthermore, on average ePAD achieves better results in terms of correctness and completeness.

CCS Concepts: •**Social and professional topics** → **Software maintenance;** •**Software and its engineering** → **Design patterns; Software reverse engineering; Maintaining software;**

General Terms: Design, Algorithms, Performance

Additional Key Words and Phrases: Design pattern recovery, software maintenance, reverse engineering

ACM Reference Format:
Andrea De Lucia, Vincenzo Deufemia, Carmine Gravino and Michele Risi, 2017. Behavioral Design Pattern Detection based on Model Checking and Dynamic Analysis. *ACM Trans. Softw. Eng. Methodol.* V, N, Article A (January YYYY), 40 pages.
DOI: 0000001.0000001

1. INTRODUCTION

Program comprehension is a key and expensive activity during software maintenance, which is complicated by many factors such as missing, incomplete, or obsolete documentation [Pigoski 1996; Bennett and Rajlich 2000]. Reverse engineering [Chikofsky and James Cross 1990] and in particular architectural reconstruction techniques [Koschke 2008] can help during program comprehension and maintenance.

Some studies have highlighted that the availability of documentation about design pattern instances allows to better comprehend the source code aiming to perform modification tasks [Prechelt et al. 2002; Vokác et al. 2004; Jeanmart et al. 2009; Gravino et al. 2011; Krein et al. 2011; Gravino et al. 2012]. According to [Gamma et al. 1995], a design pattern can be seen as a set of classes, related through aggregations and delegations. It represents a partial solution to a common non-trivial design problem, e.g.,

Author's addresses: A. De Lucia, V. Deufemia, C. Gravino, and M. Risi, Department of Computer Science, University of Salerno, 84084 Fisciano(SA), Italy; corresponding author's email: gravino@unisa.it.

to encapsulate command requests, to separate an interface from the different possible implementations, to use different platforms, and to wrap legacy systems [Gamma et al. 1995].

Reverse engineering design pattern instances from source code can play a crucial role, when no software documentation is available, by providing software maintainers with considerable insight on the software structure and its internal characteristics. However, recovering of design pattern instances from source code is an extremely time consuming task if performed manually. As a consequence, in the last decades researchers have proposed several reverse engineering techniques to automate the identification of design pattern instances in software programs [Wendehals 2003; Kaczor et al. 2006; Moha and Guéhéneuc 2007].

Design patterns are classified as structural, which concentrate on object composition and their relations in the runtime object structures, creational, which address object instantiation issues, and behavioral, which focus on the internal dynamics and object interaction in the system. In particular, creational and behavioral design patterns are very difficult to identify automatically in source code since their definition includes structural information, which defines the role of the participants in the pattern and the messages each of them can receive, and behavioral information, which describes the interactions between the runtime entities by detailing the precedence and concurrence relationships among the exchanged messages. The existing techniques for automatically identifying these types of design patterns in source code searches for the implementation of structural and behavioral properties of design patterns using only static analysis (e.g., [De Lucia et al. 2009b; **?**; Zanoni et al. 2015; Yu et al. 2015; Bafandeh Mayvan and Rasoolzadegan 2017]), or by combining static and dynamic analyses (e.g., [De Lucia et al. 2009a; Wendehals and Orso 2006; Heuzeroth et al. 2003; Ng et al. 2010]). For design patterns that comprise significant behavioral aspects the application of static analysis alone leads to the recovery of many false positives, i.e., instances that satisfies the structural requirements of a pattern, while violating the behavioral ones. On the other hand, the verification of design pattern behavioral aspects requires the monitoring and analysis of a high number of object interactions during the execution of the software system.

In this paper we present an approach able to recover instances of structural, creational, and behavioral design patterns according to the definitions provided in [Gamma et al. 1995]. The approach analyzes and verifies the structure and the behavior of design patterns in source code by combining static analysis, model checking, and dynamic analysis in two phases. In the structural analysis phase, the two steps approach previously proposed in [De Lucia et al. 2009b] is used, which identifies candidate instances by analyzing the design structure of the software system using visual language parsing and static source code analysis. In the behavioral phase, model checking [De Lucia et al. 2010b] and dynamic analysis [De Lucia et al. 2009a] are used to verify the behavior of the candidate instances produced as output of the structural phase. Model checking allows to statically verify whether the interactions among objects of design pattern candidate instances satisfy the behavioral definition of that pattern [Bultan and Betin-Can 2008]. This allows to identify and eliminate false positives obtained as result of the structural analysis phase, thus avoiding to verify them at runtime through dynamic analysis [De Lucia et al. 2010b; Bernardi et al. 2015]. The dynamic analysis is performed through the automatic instrumentation of the method calls involved in the design pattern candidate instances produced as a result of the structural analysis and model checking steps. The dynamic information obtained from the execution of test data is matched against the definition of the pattern behavior expressed in terms of monitoring grammars. The latter are constructed from UML se-

quence diagrams and used to generate a parser able to recognize the valid sequence of messages exchanged among the objects participating in the pattern instances.

Since creating test data manually is tedious and expensive, especially when the static analysis phase detects a high number of instances including many false positives, we automatically generate test data exercising the design pattern candidate instances using a genetic algorithm [Goldberg 1989]. To the best of our knowledge, the approach proposed in this paper is the first automating the generation of test data for the recovery of design pattern instances.

Our proposal implements a pipeline, which starts by applying an imprecise but fast program analysis to identify pattern candidate instances, and then it performs a more precise but expensive analysis to filter them. In the context of design pattern recovery this kind of pipeline has been adopted by other approaches, e.g., [Wendehals and Orso 2006; Heuzeroth et al. 2003; Pettersson 2005; Guéhéneuc and Antoniol 2008; Huang et al. 2005; Wendehals 2003; De Lucia et al. 2009a]. This is due to the unsuitability of using a recovery process based on dynamic analysis only, and to the easiness of verifying the structural characteristics of design patterns through a static analysis [Wendehals and Orso 2006]. The proposed pipeline differs from the ones used by other design pattern recovery approaches in the use of a model checking step aiming to reduce the number of false positives to be verified during dynamic analysis. It is worth noting that model checking has also been proposed for design pattern recovery [Peng et al. 2008; Bernardi et al. 2015], but it has not been used in combination with dynamic analysis to more precisely identify the behavioral aspects of design patterns. With respect to previous approaches which use dynamic analysis for design pattern recovery, in this paper we (i) define an approach based on genetic algorithms to automatically generate test data to exercise the design pattern candidate instances identified statically, (ii) consider a wider range of design patterns, (iii) conduct a thorough evaluation on six open source projects, and (iv) provide an on-line experimental package with all the results. Obviously, the dynamic analysis requires that the analyzed system compiles without errors and is executable. When this is not the case, the proposed pipeline can stop after the model checking step. This does not affect the number of recovered true instances, but can increase the number of false positives.

The retrieval accuracy of the proposed approach has been assessed by implementing a prototype, named ePAD, that is able to recover design patterns as defined in [Gamma et al. 1995], and by applying it on six open source software systems (namely, JHotDraw 5.1 and 6.0, JUnit 3.7, JRefactory 2.6.24, QuickUML 2001, and MapperXML 1.9.7). In this paper we report and discuss only the results obtained for creational and behavioral design patterns since the results for structural design patterns can be found in [De Lucia et al. 2009b]. Furthermore, the achieved results have been compared with the results achieved by other approaches validated on the same software systems. We have also empirically validated to what extent model checking and dynamic analysis (i.e., the steps verifying the behavior of the candidate instances) allow us to significantly reduce the number of false positives obtained at the end of the structural analysis phase. Finally, we also provide data about time performances of the different phases of the design pattern recovery approach and discuss scalability issues.

This paper extends our previous work published in [De Lucia et al. 2009a; 2010b] by:

— detailing the description of the model checking and dynamic analysis phases of the proposed design pattern recovery technique;
— introducing a genetic algorithm for the automatic generation of test data to be used during the dynamic analysis;

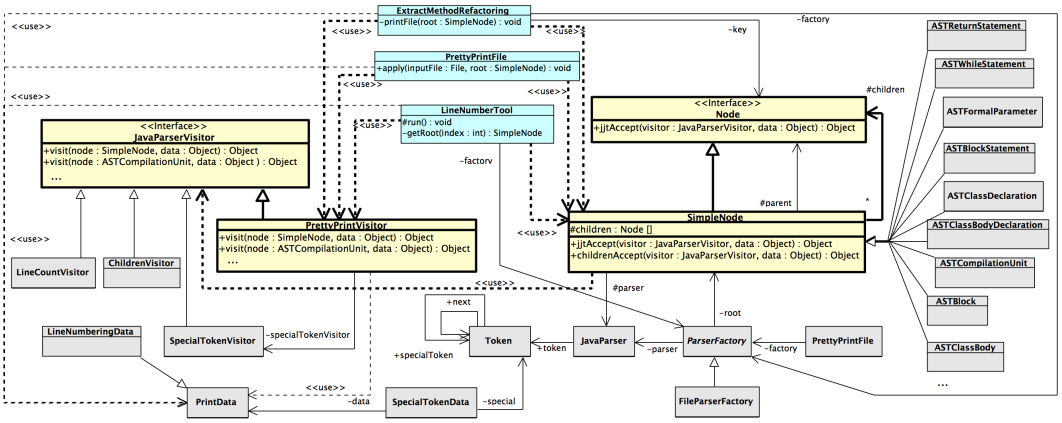

Fig. 1. An instance of Visitor pattern recovered from JRefactory 2.6.24 is highlighted by bold classes and relationships.

— evaluating the approach and tool on six open source software systems of different size and a higher number of design patterns;
— providing a comparison with related approaches that have been evaluated on the same software systems;
— enriching the related work discussion and comparison.

The paper is structured as follows. The recovery process is presented in Section 2, while Sections 3 and 4 provide details on model checking and dynamic analysis. The design and the results of the empirical study are described in Sections 5 and 6, respectively. Section 7 discusses related work, while conclusions and suggestions for future work conclude the paper.

2. THE DESIGN PATTERN RECOVERY APPROACH

In this section we present an overview of the proposed design pattern recovery approach. We start by presenting the Visitor behavioral design pattern, which is the running example used throughout the paper. The Visitor pattern represents a way to separate an algorithm from the structure of the object on which it operates and is typically used for traversing trees and similar data structures [Gamma et al. 1995]. The canonical definition of the Visitor pattern includes two main groups of classes: those related to the algorithms (their role name is *Visitor*) and those related to the data structure elements (their role name is *Element*). As an example, Figure 1 shows a portion of the JRefactory [Seguin 2002] class diagram highlighting the classes involved in a Visitor pattern instance. The behavior of this instance is to pretty print a Java source code by navigating its Abstract Syntax Tree (AST). This is accomplished by iteratively invoking the *visit* method of *PrettyPrintVisitor* class which defines the AST visit criteria. At each iteration, each visited *SimpleNode* declares through the *jjtAccept* method its identity to the *PrettyPrintVisitor* which in turn runs the appropriate *visit* method.

The structural and behavioral information of the Visitor design pattern can be defined in terms of UML class and sequence diagrams [Gamma et al. 1995] as shown in Figures 2 and 3. The class diagram defines a *Visitor* interface that declares a *visitConcreteElement* operation for each *ConcreteElement* class in the *ObjectStructure*. Each *ConcreteVisitor* class implements all the *Visitor* operations by invoking the algorithm defined for the corresponding object in the structure. *ObjectStructure* is a class containing all the objects that can be visited and offers a mechanism to iterate through

all these objects. All the *ConcreteElement* classes implement an *accept* operation that takes a visitor as an argument. The sequence diagram defines the behavior of the objects involved in the pattern. In particular, when the client creates a *ConcreteVisitor* object v then it goes through the object structure, visiting each element using v. When an element is visited, it calls the *visitConcreteElement(element)* operation. The element supplies itself as an argument to this operation to let the visitor access its state, if necessary [Gamma et al. 1995].

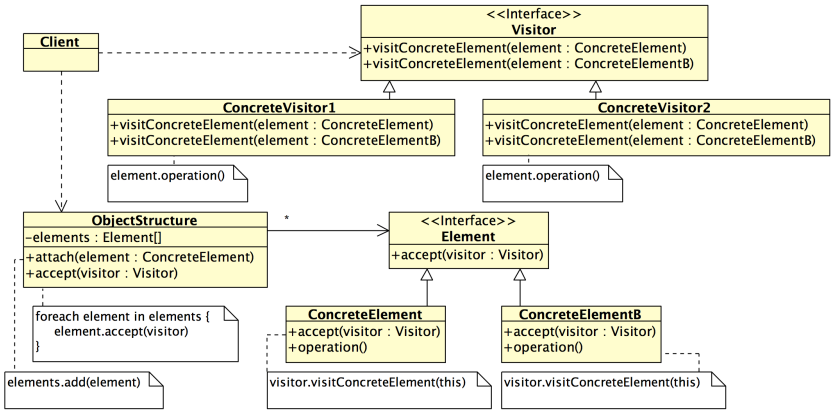

Fig. 2. Structural information for Visitor pattern.

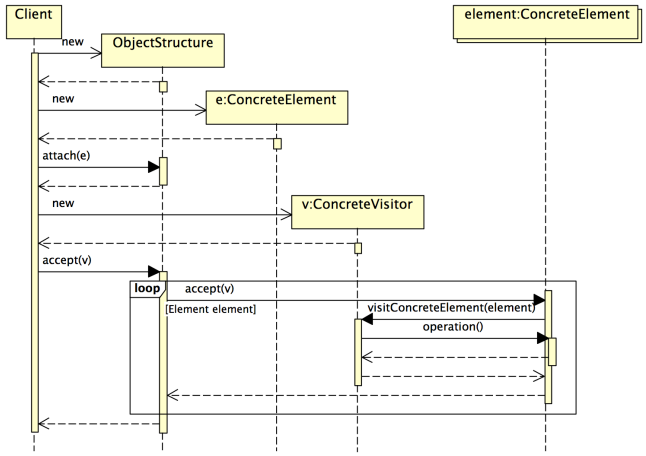

Fig. 3. Behavioral information for Visitor pattern.

According to the definition of design patterns [Gamma et al. 1995], the recovery technique proposed in this paper applies a combination of structural and behavioral analyses to identify instances of creational and behavioral design patterns from Java programs. Figure 4 shows the activity diagram describing the phases of the proposed recognition process, where rectangles represent data and rounded rectangles represent phases. The process is composed of two phases, namely Structural phase and

Behavioral phase. The first identifies an initial set of candidate instances by verifying structural properties of design patterns through static analysis. While this phase might be sufficient to accurately identify instances of structural design patterns [De Lucia et al. 2009b], an analysis of the behavior of the recovered candidate instances is required to prune false positives and more precisely identify instances of design patterns. This is the goal of the Behavioral phase, which is composed of a model checking and a dynamic analysis steps. Dynamic analysis is required because the behavior of pattern instances can only be accurately verified at runtime. However, dynamic analysis is rather expensive and challenging, as it requires an accurate selection of test data for each candidate instance. For this reason, we apply a Model Checking step to reduce the number of candidate instances and consequently the costs of the Dynamic analysis step. In particular, it allows to statically verify the behavior of the candidate instances produced by the Structural phase, thus eliminating many false positives. Furthermore, we employ a genetic algorithm to automatically generate test data exercising the candidate instances during Dynamic analysis. The design pattern recovery process has been implemented as an Eclipse plug-in, named ePAD [De Lucia et al. 2010a].

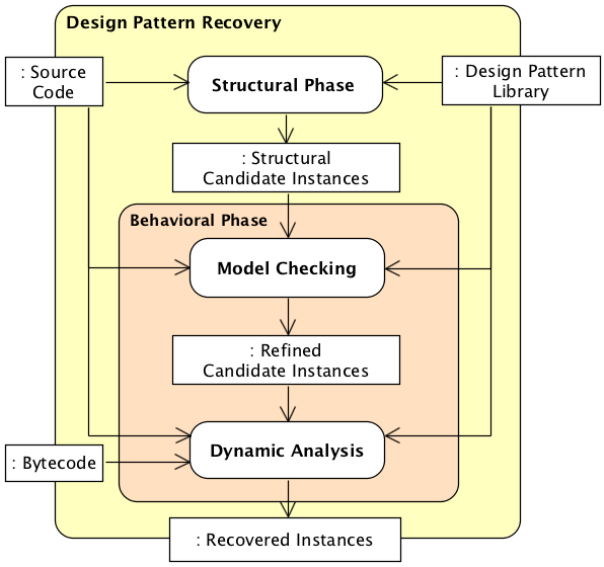

Fig. 4. The design pattern recovery process.

In the following we describe the phases of the process in Figure 4.

2.1. Structural Phase

This phase identifies design pattern candidate instances (Structural Candidate Instances in Figure 4) by exploiting structural information only. In order to detect such instances we employ the two-steps approach proposed in [De Lucia et al. 2009b]. It takes as input the *source code* of the software system to be analyzed as Java files and the *Design Pattern Library*. The latter contains the specification of both the structural and behavioral properties of design patterns, which are used during the different phases of the recovery process. In particular, the structural properties are specified by textual rules defining structural connections among the classes via call, delegation, or inheritance relationships [De Lucia et al. 2010a].

ePAD extracts structural information proper to recover design pattern instances, such as names and types of classes, class relationships, method declarations and method invocations, from the input source code and stores it in a repository. Design pattern instances are then identified by analyzing the class diagram structure in two steps:

(1) In the first step, the candidate pattern instances are identified at a coarse-grained level by analyzing through visual language parsing the class diagram constructed from the source code [Tonella and Potrich 2005]. The visual language parser analyzes the UML class diagram by searching for groups of classes and relations which satisfy the structural definitions of the design patterns stored in the *Design Pattern Library*. The structural design pattern definitions are encoded as eXtended Positional Grammars (XPG) which drive the search process of the visual language parser [Costagliola et al. 2004; Costagliola et al. 2006]. In this way, the problem of identifying design patterns is reduced to the problem of recognizing sub-diagrams in a class diagram, where each sub-diagram corresponds to an instance of a design pattern specified by a visual language grammar [Costagliola et al. 2007]. As an example, the visual language parser for the Visitor pattern identifies the instance in Figure 1 since the hierarchies *JavaParserVisitor* and *Node* match the structures of *Visitor* and *Element*, respectively (see Figure 2), and they are related through a dependency between *JavaParserVisitor* and *SimpleNode*. The latter matches the *ObjectStructure* class since it is related with *Node* through the association *children*. The clients of this pattern instance are *ExtractMethodRefactoring*, *PrettyPrintFile*, and *LineNumberTool* since they use both *PrettyPrintVisitor* and *SimpleNode*.
(2) In the second step, source code constraints characterizing the patterns are verified on the candidate instances. In particular, this is performed by exploiting a fine-grained source code analyzer that checks at source code level the role of classes participating in the pattern instances and the method declarations in the delegating classes [De Lucia et al. 2009b]. The checks are able to verify for the different types of design patterns (i) if there exists an inheritance relationship between two classes, and (ii) if there exists a method in a class that delegates a method of another class. As an example, the instance in Figure 1 also satisfies the source code level checks of the Visitor pattern since there exist the method *visit()* (declared in *JavaParserVisitor* and implemented in *PrettyPrintVisitor*) invoked by *jjtAccept()* (declared in *Node* and implemented in *SimpleNode*) and the method *jjAccept()* invoked by *childrenAccept()* (declared in *SimpleNode*).

Figure 5 shows the textual representation of the Structural Candidate Instance highlighted in Figure 1 and produced as output of the Structural phase. The textual description is composed of two sections. The first is a list of the classes involved in the design pattern instance, each associated with its playing role and the list of the invoked methods enclosed in the curly brackets. The second section provides the list of classes playing the client role with associated method, which invokes a method of the *objectStructure* class, i.e., it might trigger the execution of the pattern instance.

It is worth noting that the definition of a design pattern comes with a canonical form which includes the elements composing the pattern instances and their relationships [Gamma et al. 1995]. However, the implemented instances of design patterns can diverge from such canonical form for different reasons [Fowler 1999], such as additional design requirements (pattern variants). The ePAD tool implementing the design pattern recovery process includes an editor allowing to specify the structure of design patterns to be included in the Design Pattern Library [De Lucia et al. 2010a], in terms of coarse grained rules defining the relationships between classes or interfaces and fine grained rules verifying whether the interface extensions, class inheritances, and

```
PATTERN Visitor (
        objectStructure: SimpleNode { childrenAccept },
        element: Node { childrenAccept, jjtAccept },
        concreteElement: SimpleNode { childrenAccept, jjtAccept },
        visitor: JavaParserVisitor { visit },
        concreteVisitor: PrettyPrintVisitor { visit })
CALL_FROM_CLIENT (
        LineNumberTool.run,
        LineNumberTool.getRoot,
        ExtractMethodRefactoring.printFile,
        PrettyPrintFile.apply);
```

Fig. 5. Textual representation of the candidate Visitor pattern instance highlighted in Figure 1.

class delegations identified by the relationships in the class diagram, actually occur in the source code [De Lucia et al. 2009b]. The rules currently implemented in ePAD follow the canonical design pattern definition by [Gamma et al. 1995]. Different rules need to be defined to deal with pattern variants.

For sake of space limitations, in this paper we do not provide further details for the Structural phase. The interested reader can refer to our previous publication [De Lucia et al. 2009b] for both the details of the approach and the results of its application to the detection of structural design pattern instances. Furthermore, our on-line appendix [De Lucia et al. 2017] includes more details of the Structural phase and the Visitor pattern grammar.

2.2. Behavioral Phase

This phase consists of two steps: the behavior of a candidate instance is first analyzed statically through model checking, and then dynamically through program trace verification. The behavior of a design pattern is expressed in terms of the sequence of the method invocations defined by the sequence diagram. This kind of analysis allows to capture the dynamic information needed to unambiguously identify behavioral design pattern instances, which a static analysis is not able to precisely infer [Wendehals and Orso 2006]. As an example, the pattern instance shown in Figure 6 is identified as a Visitor candidate instance from the Structural phase but, in Section 3, we will show that it does not satisfy the behavioral properties characterizing the Visitor pattern.

The Model Checking step has the goal to statically analyze the behavior of the Structural Candidate Instances recovered by the Structural phase. This is performed by verifying for each instance whether there exists an instantiation of its classes whose interaction satisfies the pattern behavior specified in the Design Pattern Library. This verification is performed without executing the program. Indeed, by encoding the class instances and their interactions with the Promela formal language, the behavior of the obtained specification can be verified by using the SPIN model checker [Leue and Ladkin 1997]. The latter verifies, through the construction of an automaton, whether the objects involved in a candidate instance are able to exchange the sequence of messages specified in the sequence diagram defining the pattern behavior. It is worth noting that the verification process of the Model Checking step involves only the portions of the source code implementing the behavior of the candidate instances resulting from the Structural phase. The output of the Model Checking step is a set of Refined Candidate Instances.

The Dynamic Analysis step takes as input a JAR file containing the *bytecode* of the software system and the behavioral specifications of the design patterns from the *Design Pattern Library*. The ePAD tool provides an editor to support the specification of the behavior of a design pattern in terms of grammar rules describing the sequence

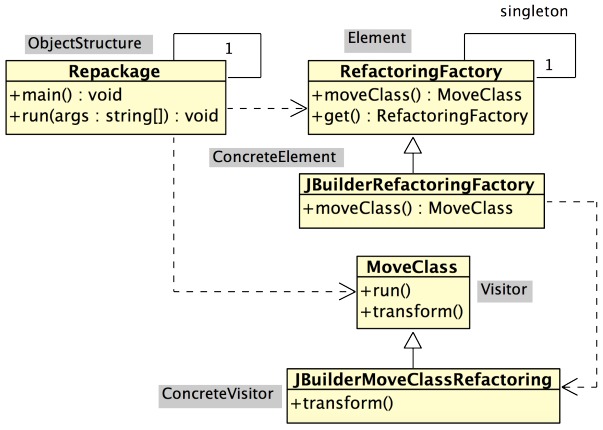

Fig. 6. A Visitor structural candidate instance.

of messages exchanged by objects [De Lucia et al. 2010a]. The behavioral analysis checks at runtime whether the actual behavior of the Refined Candidate Instances complies with the behavioral specifications in the Design Pattern Library. This is performed by monitoring the method invocations performed by the objects involved in the Refined Candidate Instances at runtime through the execution of a test suite automatically generated by using a search based approach. In particular, the bytecode of the classes involved in the Refined Candidate Instances is instrumented with Probekit tool [Eclipse Foundation 2008]. The execution of the instrumented bytecode on the input test cases produces the method trace of the classes. A parser that recognizes the sequence of method calls defining the design pattern behavior validates the obtained method traces. The result of the parsing process is the list of design pattern instances recovered by the approach (Recovered Instances in Figure 4).

In this paper, the structural and behavioral specifications of the design patterns included in the Design Pattern Library have been defined according to the design pattern definitions provided in [Gamma et al. 1995]. In Sections 3 and 4 we describe the two steps of the Behavioral phase in detail.

3. MODEL CHECKING

Figure 7 shows the phases of the Model Checking step of Figure 4. For each Structural Candidate Instance the approach generates the sequence diagram describing the interactions among the objects involved in the candidate instance. Such a diagram is then translated into a Promela formal specification that models an automaton whose states are the involved objects and the edges are the exchanged messages. In this way, we can infer the message flows and validate the behavior by using the SPIN tool. The latter operates as a simulator that follows possible execution paths through the system. These execution paths correspond to the behavior that should be satisfied by the Structural Candidate Instance and are expressed in terms of LTL formulae. In particular, a LTL formula defines the temporal message sequence that should be exchanged by the objects involved in a candidate instance. The LTL formulae are obtained by automatically parametrizing the LTL templates which encode the behavior of the design patterns and are stored in the Design Pattern Library (see Figure 7). The LTL templates are parametrized with the names of the methods invoked by the objects involved in the Structural Candidate Instance and identified through control flow analysis.

In the following we detail each step of the Model Checking process.

3.1. Constructing Sequence Diagrams Modeling Objects' Interactions Involved in Candidate Instances

The goal of this step is to extract from the source code all the information useful to model the object's interaction of the Structural Candidate Instances [Tonella and Potrich 2005]. This is performed through a static control flow analysis on the classes involved in a pattern instance. This analysis starts from the method invocations in the client classes, which might trigger the execution of the pattern. To this end, the Structural phase associates each candidate instance with such methods. The result of the control flow analysis is a sequence diagram modeling the interactions among the objects involved in the instance.

The control flow analysis is performed on the AST constructed from the source code by using the AST parser of the Java Development Tools (JDT) library [Eclipse Foundation 2009]. For each candidate instance, the control flow analysis visits the AST starting from a client method invocation $C_i.m_j()$ in order to construct a *method invocation tree*. The nodes of such a tree represent the classes encountered during the visit of the AST, while the edges represent the method invocations. To deal with infinite loop method invocations the tree is constructed by limiting its height to a fixed value. We analyzed a subset of instances of different types and verified that in general the height of the method invocation tree was never higher than 3. To be conservative and make sure that our approach worked also in other cases, in our empirical studies we set this parameter to 6. However, in the future we will perform further studies aiming to identify a strategy for selecting this parameter value.

Each node of the method invocation tree is annotated with a pair (id:$cname$), where id is the name of the variable on which a method is invoked and $cname$ is the class type of id, whereas each edge is annotated with a label c:m_1/m_2, where c is the value of the

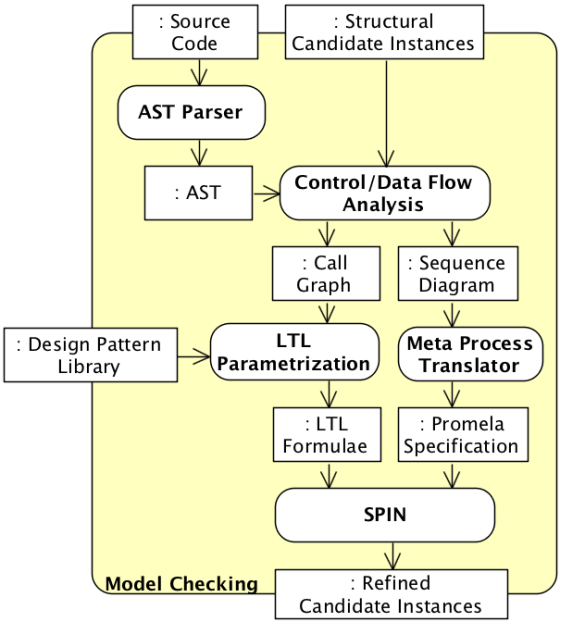

Fig. 7. Pattern behavior verification through model checking.

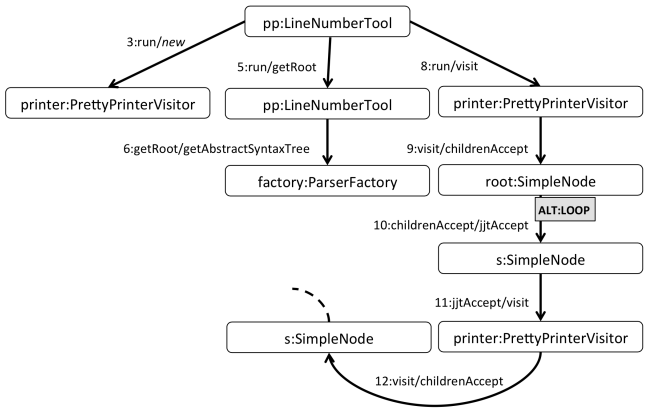

Fig. 8. The method invocation tree constructed from the Visitor candidate instance highlighted in Figure 1.

method invocation index[1], m_1 and m_2 are the caller and callee methods, respectively. As an example, the method invocation tree in Figure 8 is created for the instance in Figure 5 by starting the control flow analysis from method *run()* of class *LineNumberTool*. If the callee method m_2 is polymorphic, the control flow algorithm adds to the tree the names of all classes belonging to the pattern instance, which implement m_2. If a method contains generic parameters, the class type is automatically determined since the AST parser of JDT supports the type inference of generic types.

In order to determine all the possible object interactions we execute the following procedure that compacts the nodes referring to the same object identifier:

(1) discard the nodes representing the classes not involved in the Structural Candidate Instance. When a node is removed from the tree its father is linked to its children;
(2) join the nodes of the tree having the same class names and the same object identifiers. Notice that the method counters preserve the temporal order of the method invocations.

The data structure obtained from such operations is a *call graph*, which is represented as a sequence diagram according to the method invocation temporal sequence. In particular, the edges of the tree constructed during control flow analysis have associated contextual method information such as alternatives, loops, and object lifelines[2]. As an example, in Figure 8 the edge with method invocation index equals to 10 is annotated with a label indicating that *childrenAccept* invokes *jjAccept* in a loop contained in an alternative. The application of the compaction procedure on such tree generates the call graph shown in Figure 9, which is represented in Figure 10 as sequence diagram.

The algorithm for constructing the sequence diagrams has a linear complexity with respect to the number of nodes and edges contained in the call graph.

3.2. Generating PROMELA Specification from Sequence Diagrams

The SPIN model checker uses the Promela language for modeling the system under verification [Holzmann 2003]. Therefore the sequence diagrams obtained from the pro-

[1]The index has been implemented with a counter and it is used to track the order in which the methods are invoked.
[2]In case a method m of a class A contains branches, loops, or method invoked in any order a single sequence diagram is generated from $A.m()$.

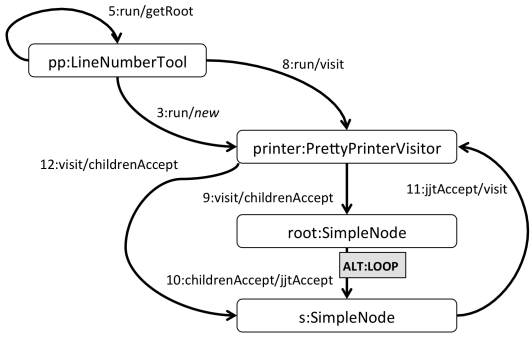

Fig. 9. The call graph constructed from the tree in Figure 8.

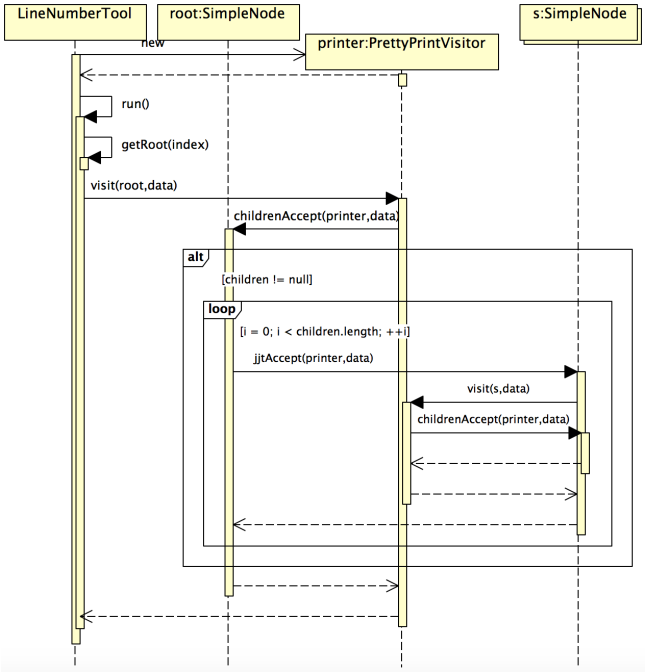

Fig. 10. Sequence diagram generated from the Visitor candidate instance in Figure 5.

cess described in the previous subsection are converted into an equivalent Promela specification by using the approach presented in [Leue and Ladkin 1997]. A Promela specification includes processes, variables, and message channels. Processes are global objects that represent entities of the system. Variables and message channels are declared within a process, with either a global scope or a local scope. The processes interact either by memory sharing, via global variables, or by message passing, via channels. The latter can either be synchronous (unbuffered) or asynchronous (buffered). Since our design pattern detection approach is based on the canonical definition of design patterns provided in [Gamma et al. 1995], the sequence diagram obtained from the code contains synchronous messages only, so we use an unbuffered channel for Promela message exchange.

Figure 11 shows the Promela specification corresponding to the sequence diagram of Figure 10. The specification consists of variable declarations and a process. The boolean variables keep track of the events occurring among the objects, whereas the message types *in* and *out* indicate that a method is invoked or is terminated, respectively. The process defines a sequence of states, each one representing an object of the sequence diagram. A set of decisional statements modeling the transitions among states is associated to each state. As an example, the fifth statement in state $s0$ models the method invocation *visit(root, data)* of *PrettyPrintVisitor* object from the *LineNumberTool* object. This statement has a pre-condition, that is the method *getRoot()* of the *LineNumberTool* object is terminated. The execution of such statement activates the state $s1$.

Note that in the Promela specification we encode the loops as alternatives because the number of executions of a loop body does not affect the identification of design pattern instances.

3.3. Generating LTL Formulae for the Pattern Candidate Instances

In order to verify whether the behavior of a candidate instance expressed as Promela code satisfies the behavioral design pattern definition, the latter has to be specified as LTL properties [Leue and Ladkin 1997]. The LTLs are mathematical annotated formulae to make statements on a linearly progressing time. The main LTL operators are "*next p*" (indicated as Xp), which means the property p is true at the next time, "*p until q*" (indicated as $p\ U\ q$), which means that the property p holds for some time after which the property q holds. The core syntax of an LTL formula φ is defined by the following grammar:

$$\varphi := !\varphi \mid \varphi \ \&\&\ \psi \mid a \mid X\varphi \mid \varphi\ U\ \psi \mid\ <>\ \varphi$$

where ! is the negation operator, $\&\&$ is the conjunction operator, a is a property, $<>$ is the operator meaning "eventually" in the future.

Such LTL formulae allow to specify properties that should (not) hold in a model of a system. For our purposes, we use LTL to express the order of send and receive events of messages that occur in a sequence diagram. In particular, for each design pattern we have defined a parametric LTL formula that is stored in the Design Pattern Library. For instance, the parametric LTL formula of the Visitor pattern has been created according to the sequence diagram in Figure 3 and is defined as:

$$\begin{aligned} tf \rightarrow <>(ts\ \&\&\ & \\ & (<>\ client)\ \mathrm{U} \\ & (accept\ \&\&\ !visitConcreteElement\ \&\&\ !operation)\ \mathrm{U} \\ & (visitConcreteElement\ \&\&\ !operation)\ \mathrm{U} \\ & operation) \end{aligned}$$

where tf is the defined property, ts is the starting condition for verifying the LTL formula, $client$ is a variable instantiated with the method invocations performed by the $Client$ before the $accept$ method is sent by the $ObjectStructure$ to the $ConcreteElement$. The remaining properties model the messages exchanged in the loop fragment of Figure 3.

The parametric LTL formulae are automatically instantiated by using the information on the candidate instances stored in the call graph. In particular, a set of atomic propositions are defined by considering all the method invocations in the call graph. Successively, the parametric variables in the LTL formula are instantiated with the atomic propositions based on the object's role specified in the candidate instance. As an example, to validate the behavior of the Visitor pattern instance shown in Figure 5, the previous parametric LTL formula is instantiated according to the call graph in

```
bool _run = false, newPrettyPrintVisitor = false, getRoot = false, visit = false,  childrenAccept = false, jjtAccept = false;
bool _end = false;

mtype = { in , out }
mtype = { c_run, c_newPrettyPrintVisitor, c_getRoot, c_visit, c_childrenAccept,  c_jjtAccept, c_end }
chan flow[10] = [1] of { mtype };

active proctype LineNumberTool_run()
{
 atomic{ printf("Start\n"); flow[c_run]!in; goto s0; }

 s0: /* LineNumberTool */
 if
  :: flow[c_run]?in →                        run = true; flow[c_newPrettyPrintVisitor]!in;                goto s1;
  :: flow[c_run]?out →                       _run = false; flow[c_end]!in;                                goto end;
  :: flow[c_newPrettyPrintVisitor]?out →     flow[c_getRoot]!in;                                          goto s0;
  :: flow[c_getRoot]?in →                    getRoot = true; flow[c_getRoot]!out;                         goto s0;
  :: flow[c_getRoot]?out →                   getRoot = false; flow[c_visit]!in;                           goto s1;
  :: flow[c_visit]?out →                     flow[c_run]!out;                                             goto s0;
 fi;

 s1: /* printer:PrettyPrintVisitor */
 if
  :: flow[c_newPrettyPrintVisitor]?in →      newPrettyPrintVisitor = true;  flow[c_newPrettyPrintVisitor]!out;   goto s1;
  :: flow[c_newPrettyPrintVisitor]?out →     newPrettyPrintVisitor = false; flow[c_newPrettyPrintVisitor]!out;   goto s0;
  :: flow[c_visit]?in →                      visit = true; flow[c_childrenAccept]!in;                     goto s2;
  :: flow[c_childrenAccept]?out →            visit = false; flow[c_visit]!out;                            goto s0;
  :: flow[c_visit]?in →                      visit = true; flow[c_childrenAccept]!in;                     goto s3;
  :: flow[c_childrenAccept]?out →            visit = false; flow[c_visit]!out;                            goto s3;
 fi;

 s2: /* root:SimpleNode */
 if
  :: flow[c_childrenAccept]?in →             atomic{ childrenAccept = true;
                                             if /* alt */
                                               :: flow[c_jjtAccept]!in;                                   goto s3;
                                               :: flow[c_jjtAccept]!out;                                  goto s2;
                                             fi; }
  :: flow[c_jjtAccept]?out →                 childrenAccept = false; flow[c_childrenAccept]!out;          goto s1;
 fi;

 s3: /* s:SimpleNode */
 if
  :: flow[c_ jjtAccept]?in →                 atomic { jjtAccept = true;
                                             if /* loop */
                                               :: flow[c_visit]!in;                                       goto s1;
                                               :: flow[c_visit]!out;                                      goto s3;
                                             fi; }
  :: flow[c_visit]?out →                     flow[c_jjtAccept]!out;                                       goto s2;
  :: flow[c_childrenAccept]?in →             childrenAccept = true; flow[c_childrenAccept]!out;           goto s3;
  :: flow[c_childrenAccept]?out →            childrenAccept = false; flow[c_childrenAccept]!out;          goto s1;
  :: flow[c_jjtAccept]?out ->                jjtAccept=false; flow[c_visit]!out;                          goto s1;
 fi;

 end:
 if
  :: flow[c_end]?_ →                         _end = true; printf("Accept\n");
 fi;
 }
```

Fig. 11. Promela specification of the sequence diagram in Figure 10.

Figure 9 and the roles associated to the classes in Figure 5. The instantiated LTL formula is depicted in Figure 12. Here, the atomic propositions ts, $t0$, ..., $t4$, and tf assert the state of each global boolean variable defined in the Promela specification and are used to check whether a method is active or not. In particular, the atomic propositions $t2$ and $t3$ are used to verify whether the methods *visit* and *childrenAccept* are active or not, while the clause $((t2$ **&&** $!t3)$ **U** $t3)$ defines the correct execution of these methods according to the sequence diagram in Figure 10 and it is true when $t3$ becomes active after $t2$. The Promela specification in Figure 11 satisfies the previous clause when state $s1$ is activated by the execution of the *visit* method (i.e., the variable *visit* is set

to true) and, successively, the activation of *childrenAccept* method triggers a transition to state $s3$ (i.e., the variable *childrenAccept* is set to true).

```
#define ts (_run == true)              /* ts is true during the verification */
#define t0 (newPrettyPrintVisitor == true)
#define t1 (getRoot == true)
#define t2 (visit == true)
#define t3 (childrenAccept == true)
#define t4 (jjtAccept == true)
#define tf (_end == true)              /* tf is true when a final state is reached*/

tf → <>        ( ts  &&
        ( <>t0 U                       /* eventually t0 happens */
          <>t1 U                       /* eventually t1 happens */
          <>t2 U                       /* eventually t2 happens */
          <>t3 U                       /* eventually t3 happens */
          ( t4 && !t2 && !t3 ) U       /* t4 happens but not t2 and t3 */
          ( t2 && !t3 ) U              /* t2 happens but not t3 */
          ( t3 ) )                     /* t3 happens */
      )
```

Fig. 12. Propositions and LTL formula for verifying the Visitor candidate instance in Figure 5.

3.4. Model Checking UML Sequence Diagrams

The Promela code describing the behavior of the pattern instance and the LTL formulae describing the behavior to be satisfied by the instance are given as input to SPIN, which creates a finite state model of the system to be checked. Thereafter, this state space is searched by an automated model checker to verify or falsify (by generating counterexamples) the LTL properties.

Since we want to verify whether there exists a sequence of messages in a sequence diagram that satisfies the pattern behavior, we have to specify the LTL properties introduced in the previous section in negative form. In particular, the LTL properties are specified by using the "never claim" construct, which is a process describing unwanted behaviors. In this way, the counterexample generated by SPIN corresponds to a correct behavior of the candidate instance. As an example, for the candidate instance in Figure 6 SPIN is not able to generate a counterexample since the messages among ConcreteVisitor (i.e., JBuilderMoveClassRefactoring) and ConcreteElement (i.e., JBuilderRefactoringFactory) are not exchanged in the order defined in Figure 3. In particular, the method *transform* in the ConcreteVisitor performs the refactoring operation chosen in the factory without interacting directly with it.

It is worth noting that the verification process is limited to the recovered candidate instances. Thus, the complexity of the generated models is linear with respect to the size of the call graph involving the classes participating to a candidate instance.

4. DYNAMIC ANALYSIS

In this section we detail the Dynamic Analysis step of Figure 4, which verifies whether the Refined Candidate Instances validated by the Model Checking step satisfy the behavioral requirements specified by the pattern definition. This task cannot be performed statically, but requires the execution of the program under analysis and, in particular, the analysis of the method invocations performed by the objects involved in the candidate instances at runtime.

The activity diagram in Figure 13 describes the steps of the dynamic analysis, which takes as input the source code of the software system to be analyzed, the set of candidate behavioral and creational design pattern instances resulting from the Model

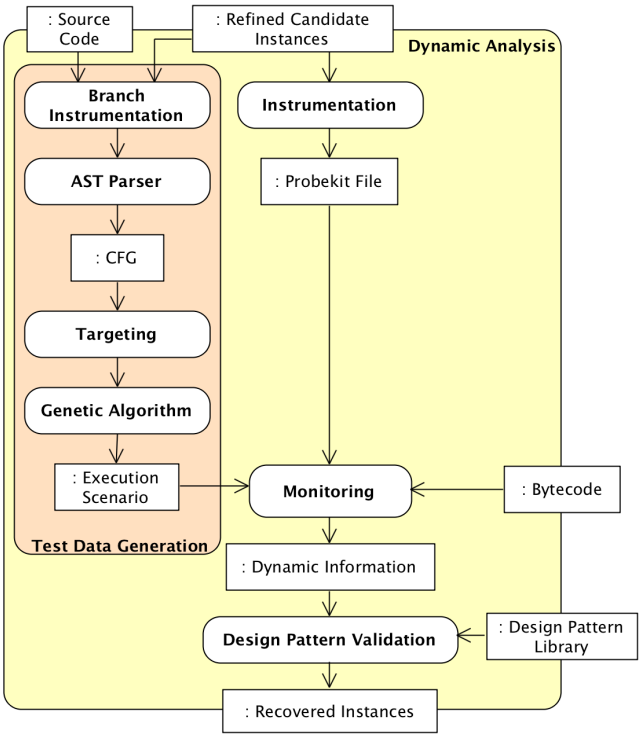

Fig. 13. The Dynamic Analysis step of the recovery process.

Checking step, the specification of the patterns' behaviors, and the executable program. The goal is to verify for each of the candidate instances whether its behavior complies with the corresponding pattern definition. This is performed for each candidate instance through the following steps. First, the Probekit tool [Eclipse Foundation 2008] is used to create the files for instrumenting the classes involved in the candidate instances (*Instrumentation*). These files are executed together with the program, i.e., the bytecode, to monitor the method calls of the candidate instances on a suitable set of test data (*Monitoring*). The latter is automatically obtained from the Refined Candidate Instances and source code by exploiting a genetic algorithm (*Test Data Generation*). Finally, the obtained method trace is validated against the specifications contained in the Design Pattern Library by a parser able to recognize the sequence of method calls defining the design pattern behavior (*Design Pattern Validation*). In the following we detail each of these steps.

4.1. Instrumentation

To instrument the bytecode of the classes involved in the candidate instances we use the Eclipse plug-in Probekit [Eclipse Foundation 2008], developed within the TPTP (Test and Performance Tools Platform Project) project [Eclipse Foundation 2008]. Probekit allows users to write Java code fragments, named *probes*, that can be injected at specified points in the classes, such as method entry, method exit, class loading, in order to collect runtime data about objects, instance variables, arguments, exceptions, and so on. One advantage of using Probekit is that a probe collects data about the por-

tion of code we are interested in. As a consequence, we can monitor only the Refined Candidate Instances by generating a set of probe files from them automatically.

4.2. Test data generation

The monitoring of the instrumented bytecode requires the definition of a set of test data. To reduce the effort of identifying suitable test data, we have defined an approach based on a genetic algorithm to generate data able to exercise the sequence of methods in the Refined Candidate Instances.

Although several search-based test generation tools have been developed in the literature, see for examples [Fraser and Arcuri 2011; Lakhotia et al. 2013; Pacheco and Ernst 2007], they are conceived to generate test data at unit level according to a coverage criterion and are not suitable to solve the specific problem of generating test data able to exercise the behavior of a design pattern instance. Indeed, in our previous work [De Lucia et al. 2015; De Lucia et al. 2015], we exploited EvoSuite, a search-based test generation tool that applies whole test suite generation in order to find test suites that achieve high code coverage on target units [Fraser and Arcuri 2011]. However, in several cases EvoSuite was not able to generate test data able to exercise candidate instances. Even if the generated test data is able to cover all the branches of each method we cannot be sure that it is able to exercise the sequence of messages characterizing the design pattern. To address this issue, we have implemented a module that automatically generates test data from the set of Refined Candidate Instances by exploiting a genetic algorithm.

Figure 13 shows the whole test data generation process consisting of four steps. The *Branch Instrumentation* instruments the branches of the Refined Candidate Instances with the global function BDAL (see details later in this Section), which determines the approach level and branch distance metrics [McMinn 2004]. These metrics are used in the fitness function of the genetic algorithm to compute the distance between the path executed by the input test data and the path exercising the sequence of messages of the pattern behavior (searched path). The *AST parser* creates a CFG for each method involved in a Refined Candidate Instance, while the *Targeting* phase automatically identifies the target nodes (statements) to be covered by the searched path. Indeed, forcing the genetic algorithm to identify an execution path covering the right sequence of method calls would have been too expensive, so we approximate this by ensuring that at least all the methods involved in these calls are executed. For this reason the *Targeting* phase selects as target nodes the statement following the call from the client class (this ensure that the first call is correctly performed) and the last statements (e.g., a return statement) of the methods involved in the behavior of the design pattern (this ensure that all involved methods are executed). The final phase is the execution of the genetic algorithms which search for test data exercising a path covering the target nodes. More details on the genetic algorithm are provided in the following.

The elementary process underlying a genetic algorithm can be summarized as follows [Goldberg 1989]. First, a random initial population is generated. New individuals (offspring) are created by applying genetic operators, and then a selection of the individuals is performed in order to establish who will survive among offspring and their parents, taking into account their closeness to the best solution. This is repeated until some stopping criteria hold. The individual providing the best solution for the problem under investigation is used. Thus, to accomplish the above process we need to: (i) encode the solution and set the initial population, (ii) define the combination of the genetic operators to explore the search space, (iii) define the fitness function to measure the goodness of a solution, (iv) define the stopping criteria. In the following we provide the details about the proposed genetic algorithm to automatically generate the set of test data.

Encoding the solution and setting the initial population. The proposed algorithm represents a solution to the problem as an array of objects, i.e., instances of Java classes belonging to the Refined Candidate Instances.
Genetic operators. New solutions are obtained by modifying the object's properties of the population with the operators tournament selection, crossover, and mutation [Koza 1992].The tournament selector determines the individuals that are included in the next generation (i.e., survivals) based on the solution quality. In particular, at each iteration, the tournament selector considers k individuals from the population (tournament) and assigns, to each of them, a probability of $p * (1 - p)^i$ to be selected, where i indicates the individual's position in the ranking of each individual tournament (determined according to the fitness function value) and p indicates the probability that the first individual (the best one) is chosen. Generally, a value less than 1 is assigned to p in order to reduce the risk of getting into a local optimum.

Crossover and mutation operators were defined to preserve well-formed object instances in all offsprings. To this end, we used a single point crossover which randomly selects the same point in each array of objects and swaps the subarrays corresponding to the selected element. Since the two subarrays are cut at the same point, the arrays resulting after the swapping have the same length as compared to those of original arrays. Concerning the mutation, we employed an operator that randomly selects a node of the array and changes a field of the corresponding object by applying the operator '+'. Crossover and mutation rate were fixed to 0.5 and 0.02, respectively, since for a population of 100 elements recommended crossover rate ranges from 0.45 to 0.95 and mutation rate ranged from 0.06 to 0.1 [Cobb and Grefenstette 1993].
Fitness function. Since the genetic algorithm aims to determine test data that allows to execute the methods of the Refined Candidate Instances, the fitness function measures the distance between the current execution (obtained with the current population) and the one that exercises the methods M_i of a Refined Candidate Instance C_i. Each execution is represented as a path S on the Control Flow Graph (CFG) of the software system, while the execution of C_i is represented as a path T containing the target nodes. The latter correspond to the statement following the invocation of m_1 from the client class and the last statements (e.g., return statements) of a method $m_j \in M_i$ with $j \neq 1$. As an example, Figure 14 shows a portion of the CFG for the instrumented JRefactory software system, which highlights the methods involved in the Visitor design pattern of Figure 1 and the target nodes with thick border lines. In particular, the method *run* of the client class contains the target node *data.flush()*, which is the statement following the pattern invocation performed by the *visit()* method.

The distance between the execution paths is computed by two metrics: approach level [McMinn 2004] and normalized branch distance [Arcuri 2010]. The first computes how many branch statements to get to the target node were not executed by a particular input (the fewer control dependent statements executed, the 'further away' the input is from executing the branch in control flow terms) [Lakhotia et al. 2013]. The latter computes the difference between evaluated and required values at the condition of the decision statement where execution diverts from the target branch [Patrick 2016]. The goal of the genetic algorithm is to minimize the following fitness function:

$$F(S,T) = \sum_{t_i} \left(AL(S,t_i) + \overline{BD}(S,t_i) \right)$$

where S is the set of statements of the CFG that are executed by current test data, $t_i \in T$ is a target node, AL and $\overline{BD}$ are the approach level and normalized branch distance metrics, respectively. In Figure 14, the function BDAL computes the approach level and branch distance metrics for the fitness function.

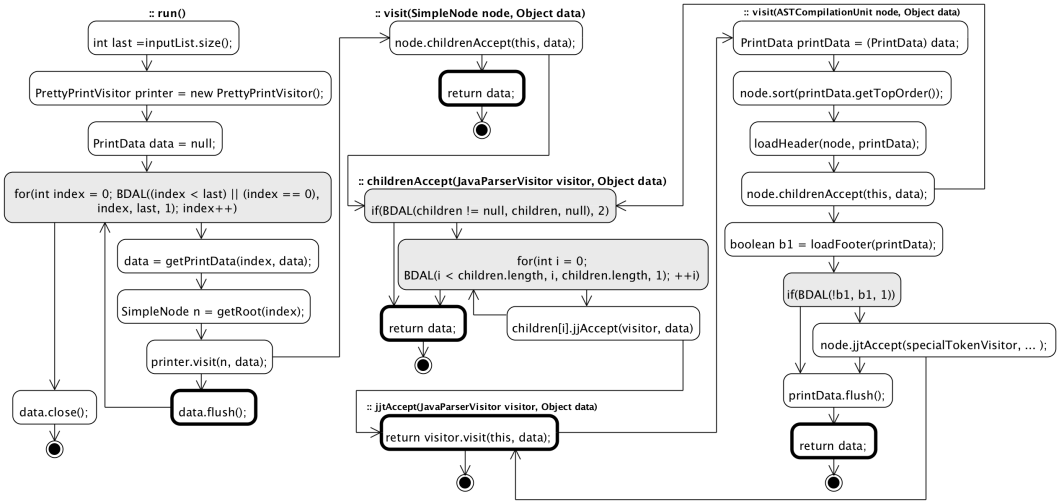

Fig. 14. A portion of CFG for JRefactory 2.6.24. The target nodes are denoted with thick border lines. The BDAL functions compute the metrics for the fitness function. The nodes corresponding to the first instruction executed within a method m are labeled with the signature of m.

It is worth noting that when the value of the fitness function $F(S,T)$ is zero then all target nodes in T are covered by S: this means that all the methods involved in the definition of a design pattern are exercised. This does not necessarily mean that the sequence of messages has been invoked in the right order, as we do not consider the sequence information as part of the fitness function, which would make the generation process computationally expensive. On the other hand, the candidate instances that correspond to false positives are detected during the Design Pattern Validation step detailed in the next Section.

Stopping Criteria. Genetic algorithms are used to solve optimization problems having huge search space, and it may happen that an algorithm does not converge towards a solution, e.g., due to loops [Harman and Jones 2001]. As a consequence, they are equipped with a stopping criteria that, for example, allow to stop the search after a certain number of generations or after some number of generations that do not provide an improvement in the fitness value. In our case, we have defined as stopping condition a maximum number of generations set to 10k.

4.3. Monitoring

The monitoring step consists of the execution of the instrumented Java program on a set of test data. The output of the monitoring process is the sequence of all monitored method calls. Table I shows the sequence of method events obtained during the monitoring of an execution of the instrumented bytecode of the Visitor pattern in Figure 1. The method invocations in the traces are described by using the notation "*obj1: A, obj2: B, obj1.M calls obj2.N*" where *obj1* and *obj2* are instances of classes A and B, respectively, and M and N are methods of A and B, respectively.

The sequence of methods obtained during the monitoring step is strongly dependent on the executed test data. As a consequence, the selection of non-representative set of test data would result in unreliable execution sequences, i.e., sequences that do not cover the fragments of the program's behavior including candidate instances. However, finding a representative set of executions is a problem for dynamic analysis in general

Table I. An excerpt of the execution trace for the Visitor candidate instance of Figure 1.

Number	ID Caller	ID Callee	Method Invocation
101	obj1: LNT	obj1: LNT	obj1.run() calls obj1.getRoot()
102	obj1: LNT	obj2: FPF	obj1.getRoot() calls obj2.getAbstractSyntaxTree()
		...	
110	obj1: LNT	obj3: PPV	obj1.run() calls obj3.visit()
111	obj3: PPV	obj4: SN	obj3.visit() calls obj4.childrenAccept()
112	obj4: SN	obj5: SN	obj4.childrenAccept() calls obj5.jjtAccept()
113	obj5: SN	obj3: PPV	obj5.jjtAccept() calls obj3.visit()
114	obj3: PPV	obj5: SN	obj3.visit(...) calls obj5.childrenAccept(...)
115	obj5: SN	obj6: SN	obj5.childrenAccept() calls obj6.jjtAccept()
116	obj6: SN	obj3: PPV	obj6.jjtAccept() calls obj3.visit()
		...	

LNT = LineNumberTool; FPF = FileParserFactory; PPV = PrettyPrintVisitor; SN = SimpleNode;

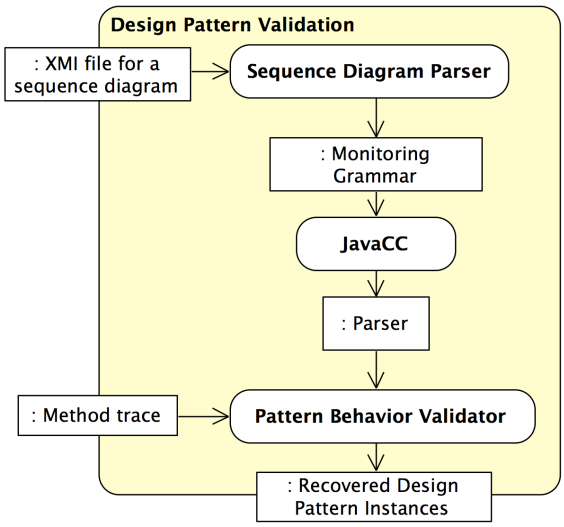

Fig. 15. The design pattern validation.

and, like any other dynamic analysis based technique, our technique is not immune from this problem.

4.4. Design Pattern Validation

The goal of this step is to validate the behavior of the monitored candidate instances by analyzing the sequence of method calls generated by the monitoring step. In particular, the sequence has to be matched against the pattern behavior described by the sequence diagram in the Design Pattern Library. To perform the match, we follow an approach similar to the one proposed in [Xu and Liang 2006] to compare the actual behaviors and expected behaviors of a software system. In particular, as shown in Figure 15, we use a compiler that takes as input the XML Metadata Interchange (XMI) representation of the UML sequence diagram describing the pattern behavior and constructs a monitoring grammar expressed in terms of the XPG formalism [Costagliola et al. 2004]. Such a grammar encodes the sequence of messages exchanged by the objects as specified in the sequence diagram. In this way, we can construct a *Pattern Behavior Validator* able to track the method calls that occur during the execution.

As an example, the grammar in Figure 16 describes the behavior of Visitor pattern as specified by the sequence diagram in Figure 3. *MethodInv* defined in production 5 is a nonterminal used in the monitoring grammars of all patterns since it encodes a method invocation entry in the trace, i.e., "*obj1.method1() calls obj2.method2()*". *MethodInv*

has associated four attributes: the callee and caller object identifiers, and the callee and caller method signatures. The Δ rule associated to the productions initializes the attribute values of nonterminals on the left-hand side using the attribute values of the symbols on the right-hand side of the production. In production 4 the nonterminal A is used to model the sequence of three method invocations occurring between the objects *ObjectStructure*, *ConcreteElement*, and *ConcreteVisitor* in the loop fragment of Figure 3. The production alternates nonterminal symbols with relations. The latter are highlighted in bold and specify constraints between the symbols' attributes. In particular, production 4 employs the following relations for defining the constraints on the method invocations:

(1) S **call** T holds if
  (a) *S.id_callee* = *T.id_caller*, and
  (b) *S.method_callee* = *T. method_caller*;
(2) S **same_caller** T holds if *S.id_caller* = *T.id_caller*
(3) S **same_callee** T holds if *S.id_callee* = *T.id_callee*

where S and T are two *MethodInv* nonterminals. As a consequence, production 4 specifies that: *MethodInv* calls *MethodInv'* and the caller of *MethodInv* is different from the caller of *MethodInv'* (**same_caller** relation is negated). These constraints are also specified between *MethodInv'* and *MethodInv"* together with the constraint that *MethodInv* and *MethodInv"* share the same callee. Notice that the superscript -1 associated to the relation **same_callee** indicates that it is defined between the symbol following **same_callee** (i.e., *MethodInv"*) and the first symbol preceding *MethodInv'* (i.e., *MethodInv*).

In production 1 the *VisitorPattern* nonterminal defines the behavior of the Visitor pattern as two *MethodInv* nonterminals having the same caller (they represent *attach* and *accept* method invocations of the Client object) and a *Loop* nonterminal. The latter describes the loop in Figure 3 by defining a recursion on A (see productions 2 and 3), which is triggered by the callee of *MethodInv'* in production 1. Relation call and Δ rule in production 3 guarantee that the sequence of methods in A is always triggered by the same caller.

The parser constructed from the monitoring grammar receives as input the monitored method events and tries to reduce the grammar productions. In case a method invocation does not match any production the parser ignores it, and if the event matches the first part of the starting event (*MethodInv* **same_caller** *MethodInv'* in the grammar for Visitor pattern) a new parser is launched. Thus, during the validation a set of parsers are active and the method invocations are given as input to all active parsers. If a parser rejects the input its state does not change. A pattern instance satisfies the

(1) VisitorPattern → *MethodInv* **same_caller** *MethodInv'* **call** *Loop*

(2) Loop → *A*
  Δ: { $Loop_{id\_caller} = A_{id\_caller}$ }

(3) Loop → *A* **call** *Loop'*
  Δ: { $Loop_{id\_caller} = A_{id\_caller}$ }

(4) A → *MethodInv* **<call, same_caller>** *MethodInv'* **<call, same_caller, same_callee^{-1}>** *MethodInv"*
  Δ: { $A_{id\_caller} = MethodInv_{id\_caller}$ }

(5) MethodInv → OBJ DOT METHOD_SIGNATURE CALLS OBJ' DOT METHOD_SIGNATURE'
  Δ: { $MethodInv_{id\_caller}$ = OBJ_{id}; $MethodInv_{method\_caller}$ = METHOD_SIGNATURE;
  $MethodInv_{id\_callee}$ = OBJ'_{id}; $MethodInv_{method\_callee}$ = METHOD_SIGNATURE'}

Fig. 16. Grammar productions describing the behavior of Visitor pattern.

Table II. Some statistics of the software systems considered in the case study.

Software	#LOC	#Files	#Classes	#Delegations & inheritances
JHD5.1	9013	144	155	5767
JUnit 3.7	3129	45	47	2635
JHD6.0	23296	268	300	12936
JRefactory 2.6.24	67680	573	572	18158
QuickUML 2001	12780	152	203	4745
MapperXML 1.9.7	22942	217	230	8057

pattern behavioral requirements when a parser reduces its starting production, i.e., the production with the starting symbol on the left-hand-side. Thus, at the end of the validation step we obtain all the instances with correct pattern behavior.

From the method events in Table I the parser recognizes a correct behavior of the pattern instance by reducing production 5 for each method invocation, productions 4 and 2 for method invocations 111, 112, and 113, productions 4, 3, and 1 for method invocations 114, 115, and 116.

5. EMPIRICAL STUDY DESIGN

In this section, we describe the design of the empirical study we performed to evaluate ePAD, i.e., the tool implementing the proposed approach. The study is presented following the Goal Question Metric guidelines [Basili et al. 1994].

5.1. Definition and Context

The *goal* of the empirical study was to assess ePAD as an approach to identify design pattern instances from source code, by also comparing ePAD with other recovery approaches. The *quality focus* was on obtaining better accuracy and the *perspective* was of a researcher, who intends to assess the improvements achieved by using the proposed design pattern recovery approach.

The context of our study is represented by six open source software systems. JHotDraw, ver. 5.1 and ver. 6.0 (JHD5.1 and JHD6.0 for short), a Java framework for drawing two-dimensional graphics in structured drawing editors [Gamma and Eggenschwiler 1998], which was originally developed to illustrate the good use of design patterns for designing and documenting systems [Johnson 1992]. Thus, it is an ideal candidate to verify the effectiveness of design pattern recovery tools. JUnit (ver. 3.7) is a simple framework to write repeatable tests [Beck et al. 2011]. QuickUML (2001) is a tool for designing software, which has been created to build and manipulate class diagrams based on a core set of the UML notation [Crahen et al. 2002]. JRefactory (ver. 2.6.24) is a system designed to refactor or restructure Java programs [Seguin 2002]. MapperXML (ver. 1.9.7) is a presentation framework for web applications based on the Model View Controller (MVC) pattern [Phelan 2000]. Table II contains some statistics on the software systems under analysis. We choose these systems since: (i) they have been used in several studies to assess the accuracy of other design pattern recovery approaches proposed in the literature (e.g., [Tsantalis et al. 2006b; Rasool and Mäder 2011; Bernardi et al. 2014]), (ii) they can be easily obtained from the web, (iii) statistics on design pattern instances recovered from these systems are available from the web (e.g., P-MARt repository [Guéhéneuc 2000] and home pages of researchers working on this topic [Tsantalis et al. 2006a; Rasool et al. 2011]), and (iv) they are developed in Java.

Here we report and discuss the results obtained for creational and behavioral design patterns, which represent the focus of the proposed recovery technique. Note that the current implementation of ePAD does not recover the Singleton design pattern since it is very simple causing implementation mistakes difficult to verify [Wojszczyk and

Khadzhynov 2015]. Results for structural patterns can be found in [De Lucia et al. 2009b].

5.2. Research Questions

The following research questions were defined to address our goal:

RQ1: What is the accuracy of ePAD in recovering design pattern instances?
RQ2: How does the accuracy of ePAD compare with the accuracy of other available design pattern recovery approaches?

RQ1 allows us to assess the accuracy of ePAD in recovering design pattern instances while RQ2 was defined to verify the accuracy of ePAD with respect to previous approaches proposed in the literature.

To respond to our research questions we recovered design pattern instances from the six software systems reported in Table II. In particular, to answer RQ2, we compared the results achieved by ePAD with those obtained by Design Pattern Detection (DPD) tool [Tsantalis et al. 2006b], the tool proposed in [Rasool and Mäder 2011] (named RM in the following), and Design Pattern Finder (DPF) [Bernardi et al. 2014], which are three approaches based on static analysis. DPD represents the software and the design patterns to be retrieved as graphs and matrices which are used to represent important aspects of their static structure [Tsantalis et al. 2006b]. Then, a graph similarity algorithm is employed to detect candidates of design pattern instances. The idea is to apply the similarity algorithm to class hierarchies restricting the analysis to smaller subsystems rather than to the whole system [Tsantalis et al. 2006b]. DPF represents both the software system and the design patterns to be recovered as graphs of source code elements with associated high-level properties, such as the static relationships among them and their behavior [Bernardi et al. 2014]. The recovery of design patterns is performed by matching each pattern graph with the overall system graph and by annotating the elements of the type hierarchy with information on the roles they play in the pattern. A domain specific language (DSL) is introduced to define the structure of both the software system and the DPs. RM integrates multiple search techniques to obtain more accurate results [Rasool and Mäder 2011]. The definition of a pattern is separated into recurring features describing the structural, relational, and behavioral parts of a pattern. These features are detected in the source code by using the search technique most fitting for its characteristics.

We chose these approaches since they made available the software and/or the list of identified pattern instances. In particular, the statistics for DPD have been obtained by running the tool on all the software systems considered in our study, while the statistics on the instances recovered by RM and DPF for the four software systems JHD5.1, JUnit, JRefactory, and QuickUML have been downloaded from the project web sites [Rasool et al. 2011; Bernardi et al. 2014]. With respect to the recovered design patterns, DPD, RM, and DPF consider all the GoF patterns.

We did not compare the design pattern instances recovered by ePAD with those achieved by design pattern recovery approaches based on dynamic analysis (see e.g., [Wendehals and Orso 2006; Heuzeroth et al. 2003; Guéhéneuc and Antoniol 2008; Ng et al. 2010; Wendehals 2003; Pettersson 2005; Huang et al. 2005]), because (i) the corresponding tools are not available for download and (ii) the authors only reported in their papers precision/recall values without providing an on-line experimental package including the list of recovered instances, or in some cases they provided results of a small case study conducted on a system different from those considered in our experimentation.

In Section 7 we provide a higher level comparison with the approaches combining static and dynamic analyses and whose recovery accuracy was assessed by performing empirical studies.

5.3. Metrics

Let TP_{t_i} be the set of true design pattern instances identified by t_i (*true positives*), FP_{t_i} be the set of pattern instances identified by the recovery technique t_i that are not true (*false positives*), and FN_{t_i} be the set of true design pattern instances not identified by t_i (*false negatives*). To evaluate the accuracy of ePAD and the baseline techniques DPD, RM, and DPF we have used the *precision* and *recall* measures as follows [Balanyi and Ferenc 2003]:

$$Precision_{t_i} = \frac{TP_{t_i}}{TP_{t_i} + FP_{t_i}}$$

$$Recall_{t_i} = \frac{TP_{t_i}}{TP_{t_i} + FN_{t_i}}$$

The precision denotes the fraction of recovered design pattern instances which are true. The recall denotes the fraction of true instances in the system which are recovered [Balanyi and Ferenc 2003]. Since the set of true design pattern instances included in the considered software systems, i.e., $TP_{t_i} + FN_{t_i}$, is not available and its manual construction is expensive and prone to subjectiveness, we approximate it with a set $True^*$ created using the following procedure:

(1) Set $True^*$ to the instances detected by analyzing the software documentation and source code comments;
(2) Add to $True^*$ the instances contained in the P-MARt repository [Guéhéneuc 2000], which includes the statistics on JHD5.1, JUnit, JRefactory, MapperXML, and QuickUML employed in our study;
(3) Add to $True^*$ the instances detected by DPD, RM, DPF, and ePAD approaches, and considered as true instances after a manual validation performed by three independent PhD students. Observe that when they did not agree on the classification, a meeting was performed to reach a general consensus. A design pattern instance was considered as true if at the end of the meeting the three students agreed.

As an example, the Observer pattern instance whose *ConnectionFigure* class plays the role of Observer has been discovered during the analysis of source code comments of JHD5.1 [Gamma and Eggenschwiler 1998], while the Factory Method instance having the *SelectionTool* class as Creator is contained in P-MARt [Guéhéneuc 2000]. The sets $True^*$ obtained by applying the previous procedure to the considered software systems are reported in the on-line appendix [De Lucia et al. 2017].

We have exploited precision and recall to assess the accuracy of ePAD and to verify whether ePAD achieved better results than the other considered design pattern recovery approaches, i.e., DPD, RM, and DPF. Furthermore, we have performed statistical significance tests to verify whether the accuracy of ePAD is significantly better than those of other approaches, taking into account the statistics achieved for each design pattern. Consequently, the number of observations corresponds to the number of considered design patterns, i.e., 12. To this end, for each comparison we considered the precision and recall values achieved by ePAD and the precision and recall values obtained by a technique T (where T = DPD, RM, or DPF) and performed a statistical test to reject the following null hypothesis:

H_{n_1} : *there is no statistically significant difference between the precision/recall values obtained by ePAD and* T.

We applied the Mann-Whitney test (which is the non-parametric version of the Wilcoxon rank-sum test) due to the sample size and (in some cases) the non-normality of the distributions [Conover 2006]. The results were intended as statistically significant at $\alpha = 0.05$. In order to provide information about the magnitude of the difference between two distributions, we also used the Cliffs d non-parametric effect size measure because it is suitable to compute the magnitude of the difference when a non parametric test is used [Field and Hole 2003] [Kampenes et al. 2007]. The magnitude of the effect size can be classified as: negligible ($d < 0.147$), small ($0.147 \leq d < 0.33$), medium ($0.33 \leq d \leq 0.474$), and large ($d > 0.474$).

Moreover, to strengthen the comparison between ePAD and the other three approaches (i.e., DPD, RM, and DPF), we have considered the true instances retrieved by the three approaches and applied the overlap metrics proposed in [Gethers et al. 2011]:

$$TP_{t_i \cap t_j} = \frac{\left|TP_{t_i} \cap TP_{t_j}\right|}{\left|TP_{t_i} \cup TP_{t_j}\right|}\%$$

$$TP_{t_i \setminus t_j} = \frac{\left|TP_{t_i} \setminus TP_{t_j}\right|}{\left|TP_{t_i} \cup TP_{t_j}\right|}\%$$

Thus, $TP_{t_i \cap t_j}$ measures the overlap between the set of actual pattern instances identified by t_i and t_j while $TP_{t_i \setminus t_j}$ measures the actual pattern instances recovered by t_i but missed by t_j. The first metric is able to give an indication of how t_i contributes to enrich the set of actual design pattern instances recovered by t_j. This information can be used to analyze the orthogonality of two methods [Gethers et al. 2011].

5.4. Threats to Validity

Some threats could affect the validity of the results of our empirical study and they should be taken into account in future investigations [Yin 1984; Kitchenham et al. 1995].

A threat that could affect the construct validity is related to the measurement and the choice of the evaluation criteria. As described when presenting the design of the empirical study, we tried to mitigate errors in the identification of true design pattern instances by performing manual validation and exploiting public repositories and results already assessed and made available by other researchers. Indeed, with benchmark publicly available and updated over several years, the effect of human mistakes or bad interpretations are reduced. As for the employed measures, in our analysis we exploited precision and recall metrics to evaluate and compare the design pattern recovery tools since they provide indication of correctness and completeness of the achieved recovery results. In the literature there are other criteria used to assess correctness and completeness. We decided to choose precision and recall since they have been widely employed in previous studies investigating the effectiveness of design pattern recovery tools and this allowed us to compare our results with those achieved in these studies. We complemented the use of precision and recall with overlap metrics proposed in [Conover 2006] that provide a good indication of the independence of the true design pattern instances identified by the tools analyzed in our study.

The choice of the software systems employed in our analysis to assess the recovery tools could affect the external validity. As mentioned in the empirical study design, from those available on the web we selected the systems developed in Java (since our

Table III. Results achieved by ePAD, DPD, RM, and DPF. A couple denotes the recovered instances and the true instances.

Software	Design Pattern	Abstract Factory	Builder	Factory Method	Prototype	Command	Iterator	Memento	Observer	State	Strategy	Template Method	Visitor
JHD5.1	True*	0	0	6	3	13	0	0	8	15	15	2	0
	ePAD	-	-	(5,5)	(3,3)	(15,13)	-	-	(9,7)	(34,15)	(45,14)	(1,1)	-
	DPD	-	(2,0)	(2,1)	(3,3)	(8,8)	-	-	(3,1)	(44,8)	(36,0)	(5,1)	-
	RM	-	-	(3,2)	(2,2)	(8,8)	-	-	(2,2)	(27,4)	(23,0)	(4,1)	-
	DPF	-	-	(6,1)	(2,2)	(21,6)	-	(1,0)	(3,1)	(32,6)	(27,1)	(10,2)	-
JUnit 3.7	True*	0	0	0	0	0	1	0	3	1	3	2	0
	ePAD	-	-	-	-	-	(2,1)	-	(3,1)	(8,1)	(11,3)	(1,1)	-
	DPD	-	-	-	-	-	-	-	(1,1)	(4,1)	(6,3)	(1,1)	-
	RM	-	-	-	-	-	-	-	(1,1)	(2,1)	(2,1)	(1,1)	-
	DPF	-	-	-	-	(5,0)	(1,0)	(1,0)	(1,1)	(4,1)	(7,3)	(9,2)	-
JHD6.0	True*	0	0	10	3	19	0	0	11	15	22	1	1
	ePAD	-	-	(8,8)	(3,3)	(21,19)	-	-	(12,9)	(30,14)	(44,20)	(1,1)	(1,1)
	DPD	-	-	(9,3)	(11,2)	(2,1)	-	-	(10,5)	(84,3)	(86,5)	(7,1)	(1,1)
JRefactory 2.6.24	True*	0	2	8	0	0	0	0	3	2	0	5	2
	ePAD	-	(2,2)	(4,3)	-	-	-	-	(3,3)	(1,1)	-	-	(2,2)
	DPD	-	-	(1,0)	-	-	-	-	(1,1)	(38,0)	(37,0)	(16,0)	(2,2)
	RM	-	(2,2)	(1,0)	-	-	-	-	-	(22,0)	(22,0)	(17,0)	(2,2)
	DPF	(3,0)	(2,2)	(28,7)	-	-	-	(6,0)	(1,1)	(37,2)	(35,0)	(29,5)	(2,2)
QuickUML 2001	True*	1	1	0	2	6	0	0	3	1	2	1	0
	ePAD	(1,1)	-	-	-	(7,6)	-	-	(3,3)	(3,1)	(1,1)	(1,1)	-
	DPD	-	-	-	(7,1)	-	-	-	-	(14,1)	(15,2)	(4,1)	-
	RM	-	(1,1)	-	-	-	-	-	(1,1)	(1,1)	(1,1)	(5,1)	-
	DPF	(1,1)	-	(4,0)	(6,1)	(5,2)	-	-	(1,1)	(14,1)	(15,2)	(5,1)	-
MapperXML 1.9.7	True*	1	0	1	0	0	0	0	11	0	1	4	0
	ePAD	(1,1)	-	(1,1)	-	-	-	-	(2,2)	(1,0)	(2,1)	(4,3)	-
	DPD	-	-	-	-	-	-	-	(11,11)	(23,0)	(24,1)	(4,1)	-

"-" means no instance recovered

ePAD implementation works for Java software systems) and employed in previous empirical studies to assess the effectiveness of other design pattern recovery tools. Indeed, unlike other approaches that work only on structures abstracting the source code (e.g., graphs, UML class diagrams), ePAD also perform source code analysis to check the candidate instances. Furthermore, we are aware that a detailed comparison with other recovery approaches performing dynamic analysis should be performed [Wendehals and Orso 2006; Heuzeroth et al. 2003; Guéhéneuc and Antoniol 2008; Ng et al. 2010; Wendehals 2003; Pettersson 2005; Huang et al. 2005]. However, these tools are not available for download and we were not able to achieve them. Indeed, the authors only reported in their papers precision/recall values without providing an on-line experimental package including the list of recovered instances, or in some cases they provided results of a small case study conducted on a system different from those considered in our experimentation.

As for the conclusion validity, we are aware that the number of observations when performing statistical tests is not high. However, we carefully applied the statistical tests performed, verifying all the required assumptions.

Concerning threats that can regard reliability validity, we have made available all the information to use our approach, all the recovered instances, and the comparison at instance level with the other approaches. The experimental package used in our study is available in the on-line appendix [De Lucia et al. 2017].

6. EMPIRICAL STUDY RESULTS

Table III reports the results achieved by the recovery tools ePAD, DPD, RM, and DPF. In particular, for each design pattern and each software system considered in our study the table contains the number of actual instances (i.e., $True^*$) and the pair (recovered instances, true instances) achieved by ePAD, DPD, RM, and DPF. Notice that the rows RM and DPF are not considered in the case of JHD6.0 and MapperXML since they

Table IV. Comparison of ePAD with DPD, RM, and DPF in terms of Precision and Recall.

Approach / Design Pattern	ePAD vs. DPD Precision/Recall		ePAD vs. RM vs. DPF Precision/Recall			Performing Better	
	ePAD	DPD	ePAD	RM	DPF	Precision	Recall
Abstract Factory	1/1	0/0	1/1	-/-	0.25/1	ePAD	=
Builder	1/0.67	0/0	1/0.67	1/1	1/0.67	=	RM
Factory Method	0.94/0.68	0.33/0.16	0.89/0.57	0.5/0.14	0.21/0.57	ePAD	ePAD
Prototype	1/0.75	0.29/0.75	1/0.6	1/0.4	0.38/0.6	=	=
Command	0.88/1	0.90/0.24	0.86/1	1/0.42	0.26/0.42	RM	ePAD
Iterator	0.5/1	0/0	0.5/1	-/0	0/0	ePAD	ePAD
Memento	-/-	-/-	-/-	0/-	-/-	-	-
Observer	0.78/0.64	0.73/0.49	0.78/0.82	1/0.24	0.67/0.24	RM	ePAD
State	0.42/0.94	0.06/0.38	0.39/0.95	0.12/0.32	0.11/0.53	ePAD	ePAD
Strategy	0.38/0.91	0.05/0.26	0.32/0.90	0.04/0.10	0.07/0.30	ePAD	ePAD
Template Method	0.88/0.47	0.14/0.33	1/0.3	0.11/0.3	0.19/1	ePAD	DPF
Visitor	1/1	1/1	1/1	1/1	1/1	=	=
Aggregated	0.58/0.82	0.13/0.33	0.54/0.80	0.21/0.29	0.17/0.49	ePAD	ePAD

"-" means that Precision or Recall cannot be calculated
"=" means that ePAD performs similarly to the best of the other approaches

were not applied on these systems [Rasool et al. 2011; Bernardi et al. 2014]. As a consequence, ePAD was compared with DPD on all the considered software systems while the comparison between ePAD and RM (and DPF) was performed only on JHD5.1, Junit, JRefactory, and QuickUML. Moreover, the table does not contain statistics about the design patterns Lazy Inizialization, Chain of Responsibility, Interpreter, and Mediator since $True^*$ does not contain instances of these design patterns.

We can observe from Table III that ePAD achieved a greater (or at least equal, in some cases) number of identified true instances than DPD, RM, and DPF for all the design patterns and software systems considered, except for the following cases: Strategy for QuickUML and Observer for MapperXML, where DPD worked better; Factory method, Template Method, and State for JRefactory, and Template Method for JHD5.1 and Junit, where DPF worked better. However, in the case of Strategy for QuickUML the difference is of just one instance and the precision achieved by DPD was very low (i.e., 0.13) due to the many false positives recovered (i.e., 13). Similarly, for the case where DPF recovered a greater number of instances with respect to ePAD, DPF achieved a worse precision.

Based on the results of Table III, we calculated precision and recall values and aggregated them by design pattern, that are reported in Table IV.

In the following we present the results achieved in our study for each of the considered research questions.

6.1. RQ1: *What is the accuracy of ePAD in recovering design pattern instances?*

The precision and recall values can be considered very interesting. Indeed, we can observe that the precision is lower than 0.50 only for State and Strategy and the recall is greater than 0.60 in all the cases (except for Template Method). The precision values achieved for State and Strategy depend on several reasons. First, these patterns have structural definitions that are quite similar. Second, the Model Checking step identified false positives but it was not able to distinguish between State and Strategy. Finally, since State and Strategy patterns have a simple behavior definition, the Dynamic Analysis step was not effective for them. In particular, most of the false positives recovered by ePAD are for State and Strategy, i.e., 109 out of 124, i.e., 88%.

It is important to note that for Factory Method, Command, and Observer, which are implemented in all the analyzed systems, we recovered more instances (see Table III) and obtained very high precision and recall values. This is due to the peculiarity features of their structure (e.g., the return message from *ConcreteObserver* to *ConcreteSubject* of the Observer pattern) and behavior (e.g., the invocation of the *update*

method on a collection of *Observer* objects) that make the recovery process more effective. Indeed, ePAD recovered only 1 false positive for Factory Method (JRefactory), 5 false positives for Command (2 from JHD5.1, 2 from JHD 6.0, and 1 from QuickUML), 7 false positives for Observer (2 from JHD5.1, 3 from JHD 6.0, and 2 from JUnit). The remaining false positives recovered by ePAD include 1 for Iterator (JUnit) and 1 for Template Method (MapperXML).

From the analysis of the results reported in Table V, we have also observed that most false positives (i.e., 109 out of 124) are obtained by ePAD for the patterns characterized by a simple behavior (i.e., State and Strategy) than those with a non-trivial behavior. Regarding the remaining 15 false positives, the model checking found (statically) a sequence of method invocations that corresponds to the behavior definition provided in the library. This was confirmed by the dynamic analysis that found test data that (dynamically) verify the requested pattern behavior. To alleviate this issue, in the future we could investigate the introduction of further constraints in the definition of the patterns' behavior. As said in Section 4, the effectiveness of the Dynamic Analysis step could depend on the test cases that verify the behavior of the candidate instances. Indeed, when the behavioral requirements of the candidate instances are not validated by test cases the Dynamic Analysis step could remove true positives. However, the experimental results showed that our strategy did not remove any true positives.

To better analyze the accuracy of ePAD we also verified the impact of model checking and dynamic analysis on the accuracy (false positive reduction) of ePAD. To this end, we have reported in Table V the candidates obtained at the end of the Structural phase (Sp), and the pattern instances obtained as result of applying the two steps of the Behavioral phase, i.e., Model Checking (MC) and Dynamic Analysis (DA). This allows to analyze the impact of the Behavioral phase on the accuracy achieved by ePAD.

We can observe that MC considerably reduces the number of candidates for the DA without eliminating true pattern instances. In particular, the model checker removed, on overage, 68% of false positives for all design patterns and software systems considered in our study. For design patterns Prototype and Template Method the percentage was very high, 85% and 91%, respectively. This is due to their simple structural definition, which allows Sp to identify many Structural Candidate Instances. This result demonstrates the usefulness of MC in the recovery process reducing the effort for the next step and allowing engineers to considerably save time for the validation of the candidate instances. As for the impact of DA, this step eliminated on average 34% of false positives obtained as output of the MC step. As in the case of MC, the highest percentage was obtained for the design patterns Prototype and Template Method, namely 78% and 56%, respectively. Thus, the behavioral phase has allowed us to eliminate many false positives obtained from Sp, which is based on a static analysis only.

We also verified whether there is significant difference between the number of instances achieved executing Sp and the number of instances obtained as results of MC and DA. To this aim, we formulated the following null hypothesis (that is, the condition that the study wants to reject):

H_{n_2} : *there is no statistically significant difference between the precision/recall values obtained by ePAD at the end of the Structural and Behavioral phases.*

To test H_{n_2}, we have performed Mann-Whitney test and the result (p-value = 0.037) revealed that MC and DA (i.e., the Behavioral phase) significantly reduced the number of false positives, with a large effect size (Cliff $d = 0.515$). Thus, we can conclude that the Model Checking and Dynamic Analysis steps impact on the accuracy of ePAD by significantly reducing the number of false positives.

Table V. Number of pattern instances recovered by ePAD during the phases of the recovery process.

Software / Design Pattern	JHD5.1	JUnit 3.7	JHD6.0	JRefactory 2.6.24	QuickUML 2001	MapperXML 1.9.7
	Sp/MC/DA	Sp/MC/DA	Sp/MC/DA	Sp/MC/DA	Sp/MC/DA	Sp/MC/DA
Abstract Factory	3/0/0	1/0/0	3/0/0	1/0/0	2/1/1	2/2/1
Builder	0/0/0	4/0/0	0/0/0	12/4/2	0/0/0	0/0/0
Factory Method	9/5/5	3/1/0	15/8/8	5/4/4	0/0/0	2/2/1
Prototype	26/13/3	104/0/0	30/14/3	12/0/0	12/0/0	0/0/0
Command	17/15/15	0/0/0	27/21/21	27/1/0	25/9/7	0/0/0
Iterator	0/0/0	8/3/2	0/0/0	0/0/0	0/0/0	0/0/0
Memento	0/0/0	0/0/0	0/0/0	0/0/0	0/0/0	0/0/0
Observer	20/12/9	15/3/3	13/13/12	7/5/3	6/3/3	5/3/2
State	54/45/34	104/23/8	61/47/30	29/2/1	13/6/3	7/2/1
Strategy	54/45/45	104/23/11	61/45/44	29/2/0	13/4/1	7/2/2
Template Method	12/4/1	15/2/1	15/5/1	132/1/0	19/2/1	5/4/4
Visitor	5/2/0	0/0/0	1/1/1	8/3/2	0/0/0	0/0/0

Table VI. Comparison between ePAD and DPD, RM, and DPF using Mann-Whitney test and effect size.

Criterion	Compared Approach	Significant Difference (p-value)	Effect Size (Cliff d)
Precision	ePAD vs. DPD	Yes (0.003)	Large (0.752)
	ePAD vs. RM	No (0.745)	Negligible (0.091)
	ePAD vs. DPF	Yes (0.005)	Large (0.682)
Recall	ePAD vs. DPD	Yes (0.002)	Large (0.760)
	ePAD vs. RM	Yes (0.011)	Large (0.645)
	ePAD vs. DPF	No (0.133)	Medium (0.380)

6.2. RQ2: *How does the accuracy of ePAD compare with the accuracy of other available design pattern recovery approaches?*

In order to answer RQ2, we have compared the accuracy of ePAD with the one achieved by DPD, RM, and DPF. From Table IV we can note that on average DPD (RM and DPF, respectively) achieved precision/recall values equal to 0.13/0.33 (0.21/0.29 and 0.17/0.49, respectively) that are less than those obtained by ePAD, i.e., 0.58/0.82 (0.54/0.80, respectively).

Focusing on the results in terms of precision, we can observe that ePAD outperforms DPD, RM, and DPF for all the design patterns, except for Observer (where RM is better) and Command (where both DPD and RM are better). ePAD and RM achieved the same results for the design patterns Builder, Prototype, and Visitor. ePAD and DPF achieved the same results for the design patterns Builder and Visitor. Notice that no instance of Memento was recovered by ePAD and DPD. Differently, RM and DPF recovered some instances of Memento but they were false positives.

Regarding the results in terms of recall, ePAD outperformed DPD and RM except for the design pattern Builder (where RM worked better). ePAD achieved the same results as DPD (RM, respectively) for Visitor and Prototype (for Visitor, respectively). ePAD performed worse than DPF only for Template method. The recall values of ePAD and DPF were equal in the cases of Abstract Factory, Builder, Factory Method, Prototype, and Visitor patterns. In the remaining cases, ePAD outperformed DPF. We want to highlight that the better Recall values achieved by ePAD w.r.t. DPD, RM, and DPF depend on the Structural phase that was able to identify more true instances. Indeed, the Behavioral phase did not affect the number of recovered true pattern instances. Moreover, we have applied two overlap metrics, i.e., $TP_{t_i \cap t_j}$ and $TP_{t_i \setminus t_j}$ [Conover 2006], which can help to understand the differences between ePAD and DPD, and between ePAD and RM, obtaining the results shown in Table VII. We can observe that the overlap between actual patterns recovered by ePAD and DPD (RM and DPF, respectively)

Table VII. Overlap metrics between ePAD and the compared approaches DPD, RM, and DPF.

Comparison / Design Pattern	ePAD vs. DPD			ePAD vs. RM			ePAD vs. DPF		
	ePAD ∩ DPD	ePAD \ DPD	DPD \ ePAD	ePAD ∩ RM	ePAD \ RM	RM \ ePAD	ePAD ∩ DPF	ePAD \ DPF	DPF \ ePAD
Abstract Factory	0%	100%	0%	0%	100%	0%	100%	0%	0%
Builder	0%	100%	0%	67%	0%	0%	67%	0%	33%
Factory Method	8%	60%	8%	14%	43%	0%	21%	36%	36%
Prototype	63%	13%	13%	40%	20%	20%	40%	20%	0%
Command	24%	76%	0%	42%	58%	0%	42%	58%	0%
Iterator	0%	100%	0%	0%	100%	0%	0%	100%	0%
Memento	-	-	-	-	-	-	-	-	-
Observer	23%	41%	26%	24%	59%	0%	24%	59%	0%
State	35%	59%	3%	32%	63%	0%	47%	47%	5%
Strategy	23%	67%	5%	10%	80%	0%	25%	65%	5%
Template Method	27%	20%	7%	30%	0%	0%	30%	0%	0%
Visitor	100%	0%	0%	100%	0%	0%	100%	0%	0%
Aggregated	26%	56%	8%	28%	52%	1%	35%	45%	13%

"-" means overlap metrics cannot be calculated

is not so high, i.e., on average 26% (28% and 35%, respectively). Moreover, the percentage of true instances recovered by DPD (RM and DPF, respectively) and not by ePAD is very low, i.e., on average 8% (1% and 13%, respectively). Furthermore, for some design patterns only ePAD is able to identify actual instances, e.g., Abstract Factory and Iterator.

In order to provide details about those design pattern instances recovered by DPD (i.e., 20 instances), DPF (19 instances) and RM (1 instance), and not by ePAD (i.e., false negatives), in Table VIII we have reported for each instance: i) its design pattern type; ii) the main classes characterizing (and used to retrieve) it; iii) an explanation of the reason why ePAD missed it; iv) the software system containing it; and v) the tool (DPD or RM) able to recover it. As an example, we did not identify 10 Observer instances since ePAD missed the data type in a Vector/ArrayList container (and the iteration is performed through the *next()* method). Notice that all the instances in Table VIII were missed by ePAD during the Structural phase. We can also observe that ePAD missed 9 out of the 11 Observer instances included in MapperXML, while for the remaining software systems ePAD recovered almost all the actual Observer instances.

From the analysis of the missed instances, although they satisfy the canonical GoF definition, we were not able to identify a common reason about why ePAD failed to recover them. We think that this issue deserves further investigations in the future. In particular, we can investigate the possibility of extending the structural analysis phase of ePAD with metrics based heuristics used in DPD. However, the analysis of the results in terms of overlap metrics has confirmed that ePAD was better than DPD, DPF, and RM in recovering pattern instances.

As designed in Section 5.3, we also analyzed whether there was statistically significant difference between the precision/recall achieved by ePAD and precision/recall obtained by DPD, RM, and DPF, by performing the Mann-Whitney test. The results reported in Table VI reveal that ePAD gained significantly better precision than DPD and DPF since Hn_1 can be rejected (i.e., *there is no statistically significant difference between the precision values obtained by ePAD and T, where T= DPD or DPF*), with a large effect size. On the other hand, we could not reject Hn_1 when analyzing the precision values achieved by ePAD and RM. Furthermore, Table VI suggests that the difference in terms of recall in favour of ePAD was significant (with a large effect size) with respect to RM and DPD. Differently, we could not reject Hn_1 when analyzing the difference between ePAD and DPF. Thus, the analysis and the results presented above

Table VIII. Design pattern instances missed by ePAD and recovered by DPD, RM or DPF.

Design Pattern	Missed instance (main classes)	Explanation	Software System	Tool
Builder	BuildAction, CodeBuilder	1) only one builder in the Director 2) it does not implement the interface Creator	QuickUML	RM
Prototype	Figure, ClipboardTool	The Object clone is not executed through the clone() method, but an instance of this object is passed as a parameter		DPD/DPF
Strategy	DiagramUI, FigureRenderer	The use of .class clause does not allow to detect the relationship between the Context and the Strategy classes		
	Figure, PeripheralLocator	No client/No Context	JHD5.1	DPD
Factory Method	DrawingView	The method in charge of creating an object is implemented in the ConcreteCreator but it is not defined in the interface Creator	JHD6.0	
	Handle			
	FileSettings		JRefactory	DPF
	LocalVariableSummary			
	TypeDeclSummary			
	LabelSizeComputation	No Creator		
	RefactoringFactory	No client/No Creator		
Template Method	LineQueue	No AbstractClass		
	PrintData			
	SegmentedLine			
	RefactoringFactory	No client		
	UMLLine	No ConcreteClass		
	TestSetup	No client/No ConcreteClass	JUnit	
	AttributeFigure	AttributeFigure extends AbstractFigure that is a Template Method	JHD5.1	
Observer	Dispatcher	The data type in a Vector/ArrayList container is missed (and the iteration is performed through an iterator method)	MapperXML	DPD
	OpenFormListener, Form			
	InputListener, Component			
	CloseFormListener, Form			
	InitContextListener, Form			
	InitFormListener, Form			
	ParameterListener, Component			DPD/DPF
	RecycleListener, Component			
	RenderListener, Component			
	TriggerListener, Component			
	ViewChangeListener, DrawApplication			
State	UndoableTool, Tool	A class having a role in the pattern instance is present in an external library	JHD6.0	DPD
	SummaryLoaderState	The class playing the role State is not extended	JRefactory	DPF

allowed us to highlight that *the accuracy of ePAD is in the majority of cases better than the accuracy of the other baselines.*

6.3. Time Performance

In this Section we discuss the time performances of the different phases of our design pattern detection approach and issues related to scalability. It is worth noting that only the Structural phase analyzes the whole system by reverse engineering the class diagram and applying visual language parsing techniques to identify a first set of candidate instances. Then, the second step of the structural phase only statically analyzes at a finer grained level and validates the candidate instances of the visual language parsing phase. Similarly, the two steps of the Behavorial phase only analyze and validate the different instances coming from the previous phase and discard false positives. Thus, the performances of the model checking and dynamic analysis depend on the size and other characteristics of the classes involved in the candidate pattern instances, besides the number of instances.

Table IX shows the execution times required by ePAD for each of the analyzed systems[3]. The table reports the total times, the times for accomplishing the structural phase together with the size of the input class diagram, the times for model checking and dynamic analysis together with the number of input candidate pattern instances. Note that for the dynamic analysis we separately report the times for test data generation and the times for instrumentation, monitoring, and validation. We can observe that in the worst cases ePAD needed 2.3 and 2.7 minutes for Sp (i.e.,

[3]ePAD was executed on an Intel i7 with dual-core 2.2GHz and 8 GB RAM running Ubuntu

JHD6.0 and JRefactory), while 1.1, 1.2, and 1.3 minutes for MD (i.e., JRefactory, JUnit, and JHD6.0). In the other cases both Sp and MC required less than 1 minute. As for DA, the Instrumentation, Monitoring, and Validation steps needed 3.3, 4.5, and 1.4 minutes for JHD5.1, JHD6.0, and JUnit, while for the remaining systems these steps needed less than 1 minute. As expected a greater amount of time is required to generate test data by applying the genetic algorithm.

As mentioned above, the time needed to accomplish the structural phase depends on the number of classes/relationships of the considered software systems. On the other hand, the time needed for the model checking depends on the number of pattern candidates resulting from the structural phase and on the number of method invocations from the client classes, which might trigger the execution of the pattern candidates. In our case studies, we have noted that the number of client method invocations for each candidate instance ranges between 1 and 5. So, the number of method client calls is not an issue for the scalability of the proposed approach. In addition, in case the Model Checking phase identifies a candidate pattern instance for one client call, it does not proceed further with the other client calls. Similarly, the time needed for the dynamic analysis depends on the number of pattern candidates resulting from the model checking. Thus, the results in Table IX show that ePAD exhibits good scalability in terms of execution times as the size of the analyzed software systems increases. Moreover, the execution times of the different phases of ePAD increase approximately linearly with respect to the number of candidate instances.

The test data generation is largely the most time consuming step of our approach. In particular, the time spent by ePAD to execute the genetic algorithm, for all the considered software systems, is about 19 hours (on average 2-3 minutes per candiate pattern instance) and the improvement in the precision is about 34%, i.e., 116 false positives have been removed. It is worth noting that while this automatic test data generation phase might seem expensive, the manual definition of test cases able to exercise the behavior of a candiate pattern instance or the manual validation of candidates to discard false positives might require much more than 2-3 minutes required on average by our approach to generate test data for a candidate instance.

As for the comparison with the other approaches in terms of execution time, no information can be obtained for RM and DPF for the software systems included in our analysis and the tools are not available as described in Section 5.1. Regarding the comparison with DPD, its execution times are slightly better than those required by structural phase and model checking of ePAD. However, the precision achieved by DPD (i.e., 0.13) is considerably worse than the one obtained by ePAD at the end of the model checking phase (i.e., 0.42).

Table IX. Execution times of the design pattern recovery process.

Software	Sp		MC		DA			Total
					Time			
	Time	#classes/ #delegations & inheritances	Time	#candidate instances from Sp	Test Data Generation	Instrumentation Monitoring Validation	#candidate instances from MC	
JHD5.1	0.6	155/5767	0.8	200	350.2	3.3	141	355.0
JUnit 3.7	0.3	47/2635	1.2	358	125.7	1.4	55	128.6
JHD6.0	2.3	300/12936	1.3	226	484.0	4.5	154	492.1
JRefactory 2.6.24	2.7	572/18158	1.1	262	73.2	0.6	22	77.6
QuickUML 2001	0.4	203/4745	0.4	90	70.4	0.7	25	71.9
MapperXML 1.9.7	0.7	230/8057	0.2	28	33.3	0.4	15	34.6

Times are expressed in minutes

7. RELATED WORK

The automatic recovery of design pattern instances in source code has been the subject of a considerable amount of previous work. Early work performed structural analysis

to find design patterns in source code [Kramer and Prechelt 1996; Keller et al. 1999; Tsantalis et al. 2006b; Philippow et al. 2005]. These approaches identify structural design pattern instances with a good accuracy, but have limitations in finding patterns with behavioral properties.

Other approaches verify the structural and behavioral properties of the patterns through a static analysis [Dong et al. 2007; Park et al. 2004; Shi and Olsson 2006; Peng et al. 2008; Dong and Zhao 2007; Rasool and Mäder 2011; Bernardi et al. 2014; Binun and Kniesel 2012; Alnusair et al. 2014; Di Martino and Esposito 2016; Bernardi et al. 2015; Zanoni et al. 2015; Yu et al. 2015; Bafandeh Mayvan and Rasoolzadegan 2017]. For example, the approach proposed by [Park et al. 2004] detects design patterns by implementing a static reference flow analyzer for Java that approximates the runtime behavior among pattern participants, while the approaches proposed by [Peng et al. 2008] and [Bernardi et al. 2015] formally verify the design pattern behavior by using model checking. Also these approaches obtained bad recovery results when used to identify behavioral and creational pattern instances, as shown in Section 6.2 for RM [Rasool and Mäder 2011] and DPF [Bernardi et al. 2014]. This is mainly due to their inability of determining the responsibilities and the collaborations among the objects involved in the patterns at runtime. To address this issue several approaches, including our approach, identify pattern instances through static and dynamic analyses [Wendehals and Orso 2006; Heuzeroth et al. 2003; Guéhéneuc and Antoniol 2008; Ng et al. 2010; Wendehals 2003; Pettersson 2005; Huang et al. 2005]. It is worth to mention that dynamic analysis is a time consuming task, mainly due to the production of the test cases needed to exercise the candidate instances, and our approach is the first one embedding a module for the automatic generation of test cases.

In the following we discuss the existing approaches that exploits static and dynamic analyses and compare them against our proposal. In particular, in Section 7.1 we provide a comparison with the approaches that also carried out empirical studies (i.e., [Guéhéneuc and Antoniol 2008; Ng et al. 2010; Huang et al. 2005]), while in Section 7.2 we discuss the remaining approaches.

7.1. Comparing ePAD with Empirically Evaluated Approaches

In this section we perform a comparison between ePAD and the approaches that combine static and dynamic analyses and whose recovery accuracy was assessed by performing empirical studies. In particular, we compare ePAD with DeMIMA [Guéhéneuc and Antoniol 2008], MoDeC [Ng et al. 2010], and PRAssistor [Huang et al. 2005]. As highlighted in Section 5.2 we could not perform a comparison at the instance level since the results of these approaches are not publicly available, except for MoDeC that provided the instances for JHD5.4b1. However, MoDeC was evaluated through a preliminary study employing only five scenarios to perform the dynamic analysis on JHD5.4b1, probably due to the cost of manually selecting test data. As a consequence, we did not compare recall values since the list of recovered instances were not reported in [Guéhéneuc and Antoniol 2008; Ng et al. 2010; Huang et al. 2005].

DeMIMA is a tool that constructs a model of the system by extracting programming idioms [Guéhéneuc and Antoniol 2008], which correspond to specific characteristics of classes or relationships between them. The relationships are derived by executing static and dynamic analyses, by exploiting information on the recovered UML class diagram, and by using the trace-analysis technique presented in [Guéhéneuc et al. 2002]. The design pattern instances are recovered from the constructed model by using explanation-based constraint programming and constraint relaxation. Thus, the design pattern recognition process is based on a static analysis limiting the dynamic analysis to disambiguate composition relationships from aggregations. Table X compares the number of recovered instances (R), the number of true instances (C), and the

Table X. Results achieved by ePAD and DeMIMA on shared patterns and systems.

Design Pattern	ePAD R/C/Prec	DeMIMA R/C/Prec
Abstract Factory	2/2/1	383/3/0.01
Factory Method	8/7/0.88	339/5/0.01
Prototype	3/3/1	4/2/0.5
Command	20/17/0.85	71/2/0.09
Observer	20/16/0.8	24/5/0.21
State/Strategy	104/34/0.33	81/9/0.11
Template Method	7/6/0.86	134/6/0.05
Visitor	2/2/1	6/2/0.33
Aggregated	166/87/0.52	1042/38/0.04

"average" calculated considering the instances obtained for all the patterns and systems (for DeMIMA those reported in [Guéhéneuc and Antoniol 2008])

precision (Prec) achieved by DeMIMA and ePAD on the design patterns and software systems (JHD5.1, QuickUML 2001, JRefactory 2.6.24, JUnit 3.7, and MapperXML 1.9.7) that the empirical study in [Guéhéneuc and Antoniol 2008] and our study share. We can observe that ePAD obtained better precision than DeMIMA for all the design patterns.

MoDeC was introduced to improve the low precision achieved by DeMIMA for behavioral and creational patterns [Ng et al. 2010] through the addition of a dynamic analysis. In particular, the dynamic analysis reduces the design pattern recovery process to a constraint satisfaction problem by verifying whether the set of constraints, derived from the scenario diagram of a behavioral or creational design pattern, is satisfied by the scenario diagram instantiated from the program execution trace. Approximate pattern instances are identified by relaxing or removing the constraints manually or automatically. However, this is an iterative process that produces a high number of instances, most of which are false positives. Conversely, our approach adopts a model checking technique that reduces the number of candidate instances before verifying their behavior. MoDeC [Ng et al. 2010] and ePAD were compared taking into account the results obtained for Command, Builder, and Visitor patterns on JRefactory and QuickUML 2001. Note that MoDeC was also ran on JHD5.4b1, a version different from the ones we employed, obtaining no instance for the Builder pattern. Thus, to perform a further comparison we ran ePAD also on JHD5.4b1 for retrieving Command and Visitor patterns. In Table XI we have reported the precision achieved by ePAD and MoDeC on these systems. The statistics on the recovered and true instances of MoDeC are not available in [Ng et al. 2010]. The analysis of the results reported in Table XI suggests that ePAD performed in general better than MoDeC in terms of precision.

PRAssistor is a tool that exploits predicate logic combined with Allen's interval-based temporal logic to specify both structural and behavioral properties of design patterns [Huang et al. 2005]. To recover pattern instances the formal specifications are

Table XI. Results achieved by ePAD and MoDeC.

Design Pattern	System	ePAD Precision	MoDeC Precision
Command	QuickUML	0.80	0.33
	JRefactory	-	-
	JHD5.4b1	0.90	0.33
Builder	QuickUML	-	0.50
	JRefactory	1	-
	JHD5.4b1	-	-
Visitor	QuickUML	-	1
	JRefactory	1	0.84
	JHD5.4b1	1	0.50

"-" means no instance obtained

converted into Prolog code and executed on the facts extracted statically, by the structure parser, and dynamically, by the behavior parser. The structure parser exploits the reflection mechanism of Java to get class structures and internal relationships among classes, while the behavior parser uses the JVMPI to capture the events issued by JVM at runtime. In order to improve the precision achieved by this logic-based approach the authors introduced naming conventions analysis. Conversely, our approach is independent from the names used by developers during coding. Another limitation of this approach is due to the fact that it is not able to access the instance information at runtime. As a consequence it cannot be used to recover instances of several design patterns, such as Command, State, and Memento. Conversely, our approach is able to identify all design patterns since it starts the analysis from source code. The empirical study reported in Section 6 does not share software systems with the study conducted in [Huang et al. 2005]. Indeed, PRAssistor has been evaluated on two systems: JUnit 3.8.1 and JHD5.2. On the first, it retrieved no instance of creational and behavioral patterns, whereas for JHD5.2 it recovered 1 instance of Factory Method, 2 instances of Observer, and 3 instances of Strategy, declaring an overall precision of 0.92 since 1 instance of Strategy is a false positive. In order to perform a higher level comparison between ePAD and PRAssistor in terms of precision we ran ePAD on JHD5.2 that were analyzed in [Huang et al. 2005]. For this version of JHD, ePAD recovered 5 instances of Factory Method (precision of 1), 9 instances of Observer (precision of 0.78), and 45 instances of Strategy (precision 0.29). We observed that PRAssistor recovered fewer instances for all patterns than ePAD. This is probably due to a strict definition of design patterns that also allowed to obtain good precision. On the other hand, this leads to worse recall results since we manually verified that more actual instances of those patterns are present in JHD5.2 (e.g., 13 actual instances of Strategy).

7.2. Other Related Work not Empirically Evaluated

The approach proposed in [Wendehals and Orso 2006] combines the static analysis used in [Wendehals 2003] with a dynamic analysis, which exploits automata to represent the pattern's behavior and validate the relevant method calls monitored at runtime. The automata are manually obtained from the behavioral requirements and present several limitations in the verification of runtime object properties. In particular, since the automata have no memory besides their states, they are not able to keep track of the relationships among object's messages exchanged at runtime. As an example, in the Observer pattern, the $getState$ message sent by an $Observer$ object to a $Subject$ object is related to an $update$ message previously sent by the $Subject$ to the $Observer$. In this case, an automaton is not able to verify that the same $Subject$ object is involved in these two messages. Conversely our approach exploits the parsers automatically generated from grammars to verify the pattern's behavior, and these parsers are able to keep track of objects' interactions.

The approach presented in [Heuzeroth et al. 2003] combines a static analysis based on predicate logic and a dynamic analysis based on test actions that verify whether the candidate instances monitored at runtime satisfy the behavioral rules of the design patterns. However, both steps of the approach suffer of drawbacks and limitations, which affect its applicability. In particular, the static analysis produces many false positives due to the conservative approximations of the objects possibly assigned to each reference variable during execution. This produces quite poor results in terms of precision and causes a higher complexity in dynamic analysis. Moreover, the information extracted during the monitoring phase does not allow test actions to identify pattern instances whose method invocations involve classes not playing a role in the pattern instances. Conversely, our approach embeds in the static analysis a model checking

step, which allows to remove many false positives, while the monitoring grammars are able to model method invocations involving intermediate objects.

The approach proposed in [Pettersson 2005] performs a static analysis by querying the information extracted from source code with the Crocopat tool [Beyer and Lewerentz 2003] and then a dynamic analysis that collects behavioral information by instrumenting the runtime environment and processes them with the aim of detecting false positives. In particular, for each candidate instance a dynamic analyzer object is created to verify violations to the dynamic protocol of the pattern. As a consequence, the approach does not exclude from the candidate set the tuples never covered during the dynamic analysis. Further limitations of this approach include the impreciseness of the static analysis constraints specified as logic formulas, the performance issues of the Crocopat queries and the runtime instrumentation, and the low precision results also due to the difficulty of verifying the large set of false positives obtained from static analysis. Our approach does not suffer these drawbacks since the static analysis allows to specify precise pattern definitions, e.g., including negative constraints, and the candidate instances are validated by model checking before performing dynamic analysis.

The recovery technique proposed in [Wendehals 2003] exploits graph-rewriting-rules for both static and dynamic analyses. In particular, a set of pattern candidate instances is retrieved by applying graph-rewriting-rules to the abstract syntax graph representing the input program. A debugger is used to collect dynamic information, which is represented as an attribute call graph. In order to verify the compliance with the pattern behavior, the graph-rewriting-rules built from the sequence diagram of the pattern to be recovered are applied to the call graph. Moreover, in order to identify implementation variants of patterns fuzzy values are associated to the pattern candidate instances and presented to the software engineer, who provides feedback on the obtained results. Unfortunately, the choice of good fuzzy beliefs to be assigned to rules represents an open challenge. Indeed, they should represent the ratio of the correct matches to all matches of the rule including false positive, which can only be determined at runtime, during the analysis of a system.

8. CONCLUSIONS AND FUTURE WORK

In this paper we have described a recovery approach and a tool, named ePAD, for behavioral design patterns. The approach combines static analysis, based on visual language parsing and model checking, and dynamic analysis, based on source code instrumentation and automatically generated test data. In particular, we have introduced a genetic algorithm to automatically generate test data to exercise the design pattern candidate instances identified statically. We have also performed an empirical study to assess the proposed recovery approach by applying ePAD on six software systems.

The results of the empirical study show that ePAD is able to achieve a good accuracy, generally better than other approaches. This is also due to the more sophisticated pipeline of ePAD with respect to approaches based on structural analysis only, like DPD, RM, and DPF. Indeed, the analysis has revealed that the Behavioral phase significantly reduces the number of false positives. Furthermore, the Structural phase of ePAD allowed to recover more actual design pattern instances than DPD, RM, and DPF thus obtaining a significantly better recall.

As future work we intend to assess the current implementation of ePAD by considering other software systems, also with the aim of further analyzing the reasons that did not allow ePAD to recover the instances reported in Table VIII. Moreover, we could improve the ePAD recovery process in different ways. As highlighted in Section 6, we could enhance the Structural phase by integrating heuristics based on structural

metrics and text mining. We could also extend it to deal with network-based systems, which interact with external applications, and GUI-based tools, which need interaction and are triggered by external events. Finally, we considered creational and behavioral patterns as defined in [Gamma et al. 1995]. Thus, we did not consider the case of parallel execution when specifying the possible behavior of a given design pattern. This can represent an aspect to investigate in the future, together with the possibility of dealing with variants of design patterns. Indeed, currently our approach would require that rules are defined (and anticipated) for each different variant we want to deal with. In the future, we aim at defining rules for a more approximate specification of design patterns, which would be able to identify (possibly unanticipated) variants of design patterns, without the need for defining specific rules for each different variant. Of course, future work will be also devoted to empirically analyzing the benefits of such an approach in terms of higher recall with the possibility of recovering more false positives, thus penalizing the precision.

REFERENCES

Awny Alnusair, Tian Zhao, and Gongjun Yan. 2014. Rule-based Detection of Design Patterns in Program Code. *Int. J. Softw. Tools Technol. Transf.* 16, 3 (2014), 315–334.

Andrea Arcuri. 2010. It Does Matter How You Normalise the Branch Distance in Search Based Software Testing. In *Procs. of Intl. Conf. on Software Testing, Verification and Validation*. IEEE Computer Society, 205–214.

Bahareh Bafandeh Mayvan and Abbas Rasoolzadegan. 2017. Design Pattern Detection Based on the Graph Theory. *Knowledge-Based Systems* 120 (2017), 211–225.

Zsolt Balanyi and Rudolf Ferenc. 2003. Mining design patterns from C++ source code. In *Procs. of Intl. Conf. on Softw. Maintenance (ICSM)*. IEEE Computer Society, 305–314.

Victor R. Basili, Gianluigi Caldiera, and H. Dieter Rombach. 1994. *Encyclopedia of Softw. Eng.* Vol. 1. John Wiley & Sons, Chapter Goal Question Metric Paradigm, 528–532.

Kent Beck, Erich Gamma, David Saff, and Mike Clark. 2011. JUnit Cookbook. http://junit.org/. (2011). [Online; 23-March-2017].

Keith H. Bennett and Václav T. Rajlich. 2000. Software Maintenance and Evolution: A Roadmap. In *Procs. of Conf. on The Future of Softw. Eng. (ICSE)*. ACM, New York, NY, USA, 73–87.

Mario Luca Bernardi and others. 2014. Design Pattern Finder home. https://github.com/UnisannioSoftEng/DPF/wiki/Design-Pattern-Finder-Home. (2014). [Online; accessed 23-March-2017].

Mario Luca Bernardi, Marta Cimitile, and Giuseppe A. Di Lucca. 2014. Design pattern detection using a DSL-driven graph matching approach. *J. of Softw.: Evolution and Process* 26, 12 (2014), 1233–1266.

M. L. Bernardi, M. Cimitile, G. De Ruvo, G. A. Di Lucca, and A. Santone. 2015. Model checking to improve precision of design pattern instances identification in OO systems. In *International Joint Conference on Software Technologies (ICSOFT)*. SciTePress, 1–11.

Dirk Beyer and Claus Lewerentz. 2003. CrocoPat: Efficient Pattern Analysis in Object-Oriented Programs. In *Procs. of Intl. Workshop on Program Comprehension (IWPC)*. IEEE Computer Society, 294–295.

Alexander Binun and Gunter Kniesel. 2012. DPJF - Design Pattern Detection with High Accuracy. In *Procs. of the 16th European Conf. on Softw. Maintenance and Reengineering (CSMR '12)*. IEEE Computer Society, 245–254.

Tevfik Bultan and Aysu Betin-Can. 2008. *Scalable Software Model Checking Using Design for Verification*. Springer Berlin Heidelberg, 337–346.

Elliot Chikofsky and II James Cross. 1990. Reverse engineering and design recovery: A taxonomy. *IEEE Software* 7, 1 (1990), 13–17.

Helen G. Cobb and John J. Grefenstette. 1993. Genetic Algorithms for Tracking Changing Environments. In *Proceedings of the 5th International Conference on Genetic Algorithms*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 523–530.

William Conover. 2006. *Practical Nonparametric Statistics* (3 ed.). Wiley India Pvt. Limited.

Gennaro Costagliola, Vincenzo Deufemia, Filomena Ferrucci, and Carmine Gravino. 2006. Constructing Meta-CASE Workbenches by Exploiting Visual Language Generators. *IEEE Trans. on Softw. Eng.* 32, 3 (2006), 156–175.

Gennaro Costagliola, Vincenzo Deufemia, and Giuseppe Polese. 2004. A Framework for Modeling and Implementing Visual Notations with Applications to Software Engineering. *ACM Trans. on Softw. Eng. Methodology* 13, 4 (2004), 431–487.

Gennaro Costagliola, Vincenzo Deufemia, and Giuseppe Polese. 2007. Visual language implementation through standard compiler-compiler techniques. *J. Vis. Lang. Comput.* 18, 2 (2007), 165–226.

Eric Crahen, Carl Alphonce, and Phil Ventura. 2002. QuickUML: A Beginner's UML Tool. In *Procs. of Conf. on Object-Oriented Programming, Systems Languages and Applications (OOPSLA)*. ACM Press, 62–63.

Andrea De Lucia, Vincenzo Deufemia, Carmine Gravino, and Michele Risi. 2009a. Behavioral Pattern Identification through Visual Language Parsing and Code Instrumentation. In *Procs. of European Conf. on Softw. Maintenance and Reengineering (CSMR)*. IEEE Computer Society, 99–108.

Andrea De Lucia, Vincenzo Deufemia, Carmine Gravino, and Michele Risi. 2009b. Design pattern recovery through visual language parsing and source code analysis. *J. of Systems and Softw.* 82, 7 (2009), 1177–1193.

Andrea De Lucia, Vincenzo Deufemia, Carmine Gravino, and Michele Risi. 2010a. An Eclipse plug-in for the detection of design pattern instances through static and dynamic analysis. In *Procs. of IEEE Intl. Conf. on Softw. Maintenance (ICSM)*. IEEE Computer Society, 1–6.

Andrea De Lucia, Vincenzo Deufemia, Carmine Gravino, and Michele Risi. 2010b. Improving Behavioral Design Pattern Detection through Model Checking. In *Procs. of European Conf. on Softw. Maintenance and Reengineering (CSMR)*. IEEE Computer Society, 176–185.

Andrea De Lucia, Vincenzo Deufemia, Carmine Gravino, and Michele Risi. 2015. Towards automating dynamic analysis for behavioral design pattern detection. In *Procs. of Intl. Conf. on Softw. Maintenance and Evolution (ICSME)*. IEEE Computer Society, 161–170.

Andrea De Lucia, Vincenzo Deufemia, Carmine Gravino, and Michele Risi. 2017. Web appendix. http://docenti.unisa.it/004724/risorse?categoria=335\&risorsa=1126. (2017). [Online; accessed 23-March-2017].

Andrea De Lucia, Vincenzo Deufemia, Carmine Gravino, Michele Risi, and Ciro Pirolli. 2015. ePadEvo: A tool for the detection of behavioral design patterns. In *Procs. of Intl. Conf. on Softw. Maintenance and Evolution (ICSME)*. IEEE Computer Society, 327–329.

Beniamino Di Martino and Antonio Esposito. 2016. A rule-based procedure for automatic recognition of design patterns in UML diagrams. *Software: Practice and Experience* 46, 7 (2016), 983–1007.

Jing Dong, Dushyant S. Lad, and Yajing Zhao. 2007. DP-Miner: Design Pattern Discovery Using Matrix. In *Procs. of Intl. Conf. and Workshops on the Eng. of Computer-Based Systems (ECBS)*. IEEE Computer Society, 371–380.

Jing Dong and Yajing Zhao. 2007. Experiments on Design Pattern Discovery. In *Procs. of Intl. Workshop on Predictor Models in Softw. Eng. (PROMISE)*. IEEE Computer Society.

Eclipse Foundation. 2008. TPTP (Test & Performance Tools Platform). https://projects.eclipse.org/projects/tptp.platform. (2008). [Online; accessed 23-March-2017].

Eclipse Foundation. 2009. Eclipse JDT. http://www.eclipse.org/jdt/. (2009). [Online; accessed 23-March-2017].

Andy Field and Graham J. Hole. 2003. *How to Design and Report Experiments* (1 ed.). Sage Publications Ltd.

Martin Fowler. 1999. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Longman, Boston, MA, USA.

Gordon Fraser and Andrea Arcuri. 2011. EvoSuite: Automatic Test Suite Generation for Object-oriented Software. In *Procs. of the ACM SIGSOFT Symp. and the European Conf. on Foundations of Softw. Eng. (ESEC/FSE)*. ACM Press, 416–419.

Erich Gamma and Thomas Eggenschwiler. 1998. JHotDraw. http://www.jhotdraw.org. (1998). [Online; accessed 23-March-2017].

Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. 1995. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Malcom Gethers, Rocco Oliveto, Denys Poshyvanyk, and Andrea De Lucia. 2011. On integrating orthogonal information retrieval methods to improve traceability recovery. In *Procs. of Intl. Conf. on Softw. Maintenance (ICSM)*. IEEE Computer Society, 133–142.

David E. Goldberg. 1989. *Genetic Algorithms in Search, Optimization and Machine Learning* (1st ed.). Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Carmine Gravino, Michele Risi, Giuseppe Scanniello, and Genoveffa Tortora. 2011. Does the Documentation of Design Pattern Instances Impact on Source Code Comprehension? Results from Two Controlled Experiments. In *Procs. of Working Conf. on Reverse Eng. (WCRE)*. IEEE Computer Society, 67–76.

Carmine Gravino, Michele Risi, Giuseppe Scanniello, and Genoveffa Tortora. 2012. Do Professional Developers Benefit from Design Pattern Documentation? A Replication in the Context of Source Code Comprehension. In *Model Driven Engineering Languages and Systems*, RobertB. France, Jrgen Kazmeier, Ruth Breu, and Colin Atkinson (Eds.). Lecture Notes in Computer Science, Vol. 7590. Springer Berlin Heidelberg, 185–201.

Yann-Gaël Guéhéneuc. 2000. P-MARt. http://www.iro.umontreal.ca/~labgelo/p-mart/index.php. (2000). [Online; accessed 23-March-2017].

Yann-Gaël Guéhéneuc and Giuliano Antoniol. 2008. DeMIMA: A Multilayered Approach for Design Pattern Identification. *IEEE Trans. on Softw. Eng.* 34, 5 (2008), 667–684.

Yann-Gael Guéhéneuc, Remi Douence, and Narendra Jussien. 2002. No Java without Caffeine: A Tool for Dynamic Analysis of Java Programs. In *Procs. of Intl. Conf. on Automated Softw. Engi. (ASE)*. IEEE Computer Society, 117–126.

Mark Harman and Bryan F. Jones. 2001. Search-based software engineering. *Information & Software Technology* 43, 14 (2001), 833–839.

Dirk Heuzeroth, Thomas Holl, Gustav Högström, and Welf Löwe. 2003. Automatic Design Pattern Detection. In *Procs. of Intl. Workshop on Program Compreh. (IWPC)*. IEEE Computer Society, 94–103.

Gerard Holzmann. 2003. *Spin Model Checker, the: Primer and Reference Manual* (first ed.). Addison-Wesley Professional.

Heyuan Huang, Shensheng Zhang, Jian Cao, and Yonghong Duan. 2005. A Practical Pattern Recovery Approach Based on Both Structural and Behavioral Analysis. *J. of Systems and Softw.* 75, 1-2 (2005), 69–87.

Sebastien Jeanmart, Yann-Gaël Guéhéneuc, Houari Sahraoui, and Naji Habra. 2009. Impact of the Visitor Pattern on Program Comprehension and Maintenance. In *Procs. of Intl. Symp. on Empirical Softw. Eng. and Measurement (ESEM)*. IEEE Computer Society, 69–78.

Ralph E. Johnson. 1992. Documenting Frameworks using Patterns. In *Procs. of Conf. on Object-Oriented Programming, Systems Languages and Applications (OOPSLA '92)*. ACM Press, 63–76.

Olivier Kaczor, Yann-Gael Guéhéneuc, and Sylvie Hamel. 2006. Efficient identification of design patterns with bit-vector algorithm. In *Procs. of European Conf. on Softw. Maintenance and Reengineering (CSMR)*. IEEE Computer Society, 173–182.

Vigdis Kampenes, Tore Dyba, Jo Hannay, and Dag Sjoberg. 2007. A Systematic Review of Effect Size in Software Engineering Experiments. *Information and Softw. Technology* 4, 11-12 (2007), 1073–1086.

Rudolf K. Keller, Reinhard Schauer, Sébastien Robitaille, and Patrick Pagé. 1999. Pattern-Based Reverse-Engineering of Design Components. In *Procs. of Intl. Conf. on Softw. Eng. (ICSE)*. ACM Press, 226–235.

Barbara Kitchenham, Lesley Pickard, and Shari Lawrence Pfleeger. 1995. Case Studies for Method and Tool Evaluation. *IEEE Softw.* 12, 4 (1995), 52–62.

Rainer Koschke. 2008. Architecture Reconstruction. In *International Summer Schools on Software Engineering, ISSSE 2006-2008, Salerno, Italy, Revised Tutorial Lectures (Lecture Notes in Computer Science)*, Andrea De Lucia and Filomena Ferrucci (Eds.), Vol. 5413. Springer, 140–173.

John R. Koza. 1992. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, Cambridge, MA, USA.

Christian Kramer and Lutz Prechelt. 1996. Design recovery by automated search for structural design patterns in object-oriented software. In *Procs. of Working Conf. on Reverse Eng.(WCRE)*. IEEE Computer Society, 208–215.

Jonathan L. Krein, Landon J. Pratt, Alan B. Swenson, Alexander C. MacLean, Charles D. Knutson, and Dennis L. Eggett. 2011. Design Patterns in Software Maintenance: An Experiment Replication at Brigham Young University. In *Procs. of Intl. Workshop on Replication in Empirical Softw. Eng. Research*. IEEE Computer Society, 25–34.

Kiran Lakhotia, Mark Harman, and Hamilton Gross. 2013. AUSTIN: An Open Source Tool for Search Based Software Testing of C Programs. *Inf. Softw. Technol.* 55, 1 (2013), 112–125.

Stefan Leue and Peter B. Ladkin. 1997. Implementing and Verifying MSC Specifications Using PROMELA/XSPIN. In *Procs. of 2nd Intl. Workshop on SPIN Verification System (SPIN) (Discrete Mathematics and Theoretical Computer Science)*, Vol. 32. American Mathematical Society, 65–89.

Phil McMinn. 2004. Search-based Software Test Data Generation: A Survey: Research Articles. *Softw. Test. Verif. Reliab.* 14, 2 (2004), 105–156.

Naouel Moha and Yann-Gaël Guéhéneuc. 2007. PTIDEJ and DECOR: Identification of Design Patterns and Design Defects. In *OOPSLA Companion*. ACM Press, 868–869.

Janice Ka-Yee Ng, Yann-Gaël Guéhéneuc, and Giuliano Antoniol. 2010. Identification of behavioural and creational design motifs through dynamic analysis. *Journal of Softw. Maintenance and Evolution: Research and Practice* 22, 8 (2010), 597–627.

Carlos Pacheco and Michael D. Ernst. 2007. Randoop: Feedback-directed Random Testing for Java. In *Companion to the 22nd ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications Companion (OOPSLA '07)*. ACM Press, 815–816.

Chanjin Park, Yoohoon Kang, Chisu Wu, and Kwangkeun Yi. 2004. A Static Reference Flow Analysis to Understand Design Pattern Behavior. In *Procs. of Working Conf. on Reverse Eng. (WCRE)*. IEEE Computer Society, 300–301.

Matthew Patrick. 2016. *Computational Intelligence and Quantitative Software Engineering*. Springer International Publishing, Chapter Metaheuristic Optimisation and Mutation-Driven Test Data Generation, 89–115.

Tu Peng, Jing Dong, and Yajing Zhao. 2008. Verifying Behavioral Correctness of Design Pattern Implementation.. In *Procs. of Intl. Conf. on Softw. Eng. & Knowledge Eng. (SEKE)*. Knowledge Systems Institute, 454–459.

Niklas Pettersson. 2005. Measuring Precision for Static and Dynamic Design Pattern Recognition As a Function of Coverage. *SIGSOFT Softw. Eng. Notes* 30, 4 (2005), 1–7.

Marty Phelan. 2000. MapperXML. http://mapper.sourceforge.net/mapperxml/. (2000). [Online; accessed 23-March-2017].

Ilka Philippow, Detlef Streitferdt, Matthias Riebisch, and Sebastian Naumann. 2005. An approach for reverse engineering of design patterns. *Softw. and System Modeling* 4, 1 (2005), 55–70.

Thomas M. Pigoski. 1996. *Practical Software Maintenance: Best Practices for Managing Your Software Investment*. John Wiley & Sons, Inc., New York, NY, USA.

Lutz Prechelt, Barbara Unger-Lamprecht, Michael Philippsen, and Walter Tichy. 2002. Two Controlled Experiments Assessing the Usefulness of Design Pattern Documentation in Program Maintenance. *IEEE Trans. Softw. Eng.* 28, 6 (2002), 595–606.

Ghulam Rasool and others. 2011. Software engineering research center. http://research.ciitlahore.edu.pk/Groups/SERC/DesignPatterns.aspx. (2011). [Online; accessed 23-March-2017].

Ghulam Rasool and Patrick Mäder. 2011. Flexible design pattern detection based on feature types.. In *Procs. of Intl. Conf. on Automated Softw. Eng. (ASE)*. IEEE Computer Society, 243–252.

Chris Seguin. 2002. JRefactory. http://jrefactory.sourceforge.net. (2002). [Online; accessed 23-March-2017].

Nija Shi and Ronald A. Olsson. 2006. Reverse Engineering of Design Patterns from Java Source Code. In *Procs. of Intl. Conf. on Automated Softw. Eng. (ASE)*. IEEE Computer Society, 123–134.

Paolo Tonella and Alessandra Potrich. 2005. *Reverse Engineering of Object Oriented Code*. Springer-Verlag, New York, USA.

Nikolaos Tsantalis, Alexander Chatzigeorgiou, George Stephanides, and Spyros T. Halkidis. 2006a. Design Pattern Detection Tool and Study. http://users.encs.concordia.ca/~nikolaos/pattern_detection.html. (2006). [Online; accessed 16-June-2016].

Nikolaos Tsantalis, Alexander Chatzigeorgiou, George Stephanides, and Spyros T. Halkidis. 2006b. Design Pattern Detection Using Similarity Scoring. *IEEE Trans. on Softw. Eng.* 32, 11 (2006), 896–909.

Marek Vokác, Walter F. Tichy, Dag I. K. Sjøberg, Erik Arisholm, and Magne Aldrin. 2004. A Controlled Experiment Comparing the Maintainability of Programs Designed with and without Design Patterns-A Replication in a Real Programming Environment. *Empirical Softw. Eng.* 9, 3 (2004), 149–195.

Lothar Wendehals. 2003. Improving Design Pattern Instance Recognition by Dynamic Analysis. In *Procs. of ICSE Workshop on Dynamic Analysis (WODA)*. 29–32.

Lothar Wendehals and Alessandro Orso. 2006. Recognizing Behavioral Patterns At Runtime Using Finite Automata. In *Procs. of Intl. Workshop on Dynamic Systems Analysis (WODA)*. ACM Press, 33–40.

R. Wojszczyk and W. Khadzhynov. 2015. Data models in the verification of the Singleton pattern. *Journal of Theoretical and Applied Computer Science* 9 (2015), 35–46. Issue 3.

Kai Xu and Donglin Liang. 2006. *A Monitoring Profile for UML Sequence Diagrams*. Technical Report 06-024. Department of Computer Science, Univ. of Minnesota. [Online; accessed 23-March-2017].

Robert K. Yin. 1984. *Case Study Research: Design and Methods*. Sage Publications.

Dongjin Yu, Yanyan Zhang, and Zhenli Chen. 2015. A Comprehensive Approach to the Recovery of Design Pattern Instances Based on Sub-patterns and Method Signatures. *Journal of Systems and Software* 103 (2015), 1–16.

Marco Zanoni, Francesca Arcelli Fontana, and Fabio Stella. 2015. On applying machine learning techniques for design pattern detection. *Journal of Systems and Software* 103 (2015), 102–117.

Received April 2017; revised September 2017; accepted December 2017