2-2020

Distinguishing similar design pattern instances through temporal behavior analysis

Renhao XIONG
Southeast University

David LO
Singapore Management University, davidlo@smu.edu.sg

Bixin LI
Southeast University

Distinguishing Similar Design Pattern Instances through Temporal Behavior Analysis

Renhao Xiong
School of Computer Science and Engineering
Southeast University
Nanjing, China
renhao.x@seu.edu.cn

David Lo
School of Information Systems
Singapore Management University
Singapore, Singapore
davidlo@smu.edu.sg

Bixin Li
School of Computer Science and Engineering
Southeast University
Nanjing, China
bx.li@seu.edu.cn

***Abstract*—Design patterns (DPs) encapsulate valuable design knowledge of object-oriented systems. Detecting DP instances helps to reveal the underlying rationale, thus facilitates the maintenance of legacy code. Resulting from the internal similarity of DPs, implementation variants, and missing roles, approaches based on static analysis are unable to well identify structurally similar instances. Existing approaches further employ dynamic techniques to test the runtime behaviors of candidate instances.**

Automatically verifying the runtime behaviors of DP instances is a challenging task in multiple aspects. This paper presents an approach to improve the verification process of existing approaches. To exercise the runtime behaviors of DP instances in cases that test cases of legacy systems are often unavailable, we propose a markup language, TSML (Test Script Markup Language), to direct the generation of test cases by putting a DP instance into use. The execution of test cases is monitored based on a trace method that enables us to specify runtime events of interest using regular expressions. To characterize runtime behaviors, we introduce a modeling and specification method employing Allen's interval-based temporal relations, which supports variant behaviors in a flexible way without hard-coded algorithms. A prototype tool has been implemented and evaluated on six open source systems to verify 466 instances reported by five existing approaches with respect to five DPs. The results show that the dynamic analysis increases the F_1-score by 53.6% in distinguishing similar DP instances.

***Index Terms*—Design Pattern Detection, Temporal Analysis, Reverse Engineering, Software Comprehension, Knowledge Representation**

I. Introduction

Program comprehension is a key and expensive activity during software maintenance [1] [2] [3]. Grappling the intents of designers requires rich experience and lots of manual efforts to review the code [4] [5] since the design documentation is often missing or ignored. In recent decades, DPs are widely accepted as a solution of a recurring problem in a context, thus provides a good means to facilitate the maintenance of legacy code [6] [7] [8]. DPs encapsulate valuable design knowledge that helps to construct well-structured and maintainable software systems [2], whereas recovering DPs assists maintainers in understanding legacy systems during their routine tasks [9].

In the well-known GoF (Gang of Four) catalog [8], DPs are assigned with different intents to address a particular design issue. Each DP specifies how the participating classes and objects collaborate. Based on a static analysis of source code, a detection approach [2] [10] [11] characterizes the structures of participants (e.g., inheritance and composition [8] [12]) and the static behaviors of code elements (e.g., method calls and object allocations extracted by parsing the source code [7] [13] [14]). A *candidate instance* is reported as a DP if the instance's participating roles (represented by classes) and operations (represented by methods) satisfy the constraints of the DP.

However, some DPs are difficult to distinguish due to their similarity. They lack unique features in the source code serving as clues to distinguish each other. The reasons for DPs' internal similarity reside in the common mechanisms that DPs are based on, e.g., composition and delegation. Although such features can be captured, they are helpless in uniquely distinguishing DPs based on the same mechanism. In addition to the internal similarity of DPs, the similarity of DP instances also results from implementation variants and missing roles. The implementation of a DP may not strictly comply with the textbook, causing a DP instance to be falsely identified or confused with another instance. Missing roles also weaken the characteristics of an instance. DP instances in APIs (Application Programming Interfaces) and frameworks are usually incomplete, leaving the client role of a DP unimplemented, since the role is the responsibility of the application based on the API or framework.

While static analysis techniques examine possible execution paths, the actual behaviors of objects cannot be decided until runtime due to the mechanisms such as dynamic binding [3] [8] [15]. As a result, the detection approaches based only on static analysis report many false instances on similar DPs. Especially, several pairs of patterns, e.g., Strategy / State, Adapter / Command, are often treated as the same pattern. Their instances are grouped together in the detection results [10] [16] [17] [18].

To verify candidate instances, existing approaches further analyze their runtime behaviors. Three of the core sequential processes towards the verification are: test case generation, execution monitoring, and behavior verification. The test cases aim at triggering the interactions of an instance's participants. Driven by test cases, the program under test is executed, during which runtime data (e.g., object allocation) are collected. Finally, obtained runtime data are verified by employing techniques such as model checking [3] [19] [20].

978-1-7281-5143-4/20/$31.00 © 2020 IEEE

The main limitations of existing approaches based on dynamic analysis reside in the generation of test cases and the monitoring of runtime events. In legacy systems, test cases are usually not available, or not complete so as to cover the instances under test. As manually creating test cases [15] [21] [22] is time-consuming, automated tools have been utilized to generate test cases [20] [23]. But general-purpose test tools are not dedicated to exercise the behavior of DPs. As noted in [3], test tools are not always able to generate useful test cases even after all the branches of each method are covered. Exclusively pursuing high code coverage of the whole search space is too expensive, while only limited execution paths are relevant to the instance to verify. To address the issue, algorithms are proposed to search execution paths more efficiently taking method calls involved in an instance as optimization objectives [3] [23].

For the purpose of covering the candidate instance to verify, it could be more straightforward from the viewpoint of how a DP intends to be used. As we detect DPs in source code as a way to understand the system, candidate instances are expected to follow the DP's intended usage scenario to solve the problem in a context. But existing approaches do not take full advantage of using the candidate instance itself, i.e., its participating roles and operations, to exercise its own behavior.

To collect data at runtime, the source code or bytecode is instrumented with a fragment of code before the program executes. The instrumentation is difficult to customize for specific analysis tasks. Injected code is tangled with the original system, which may interfere with the program's behavior [15] [24]. To avoid side-effects, the injection algorithm, usually hard-coded, needs to be carefully designed for the events selected and the data to be recorded. Besides, the events supported by existing approaches based on instrumentation are limited to object allocation and method invocation [3] [23] [25]. Other events are not well supported, e.g., field access and modification that help to identify the data exchange between objects [8] [26]. It limits the ability to analyze the behaviors of object interactions.

In this paper, we define "similar DP instances" as "the instances identified as the same DP", and refer to "distinguishing similar DP instances" as "classifying each instance to proper DP or avoiding the instances falsely identified as a DP". Aiming at reducing false instances resulting from their similarity, we present an approach to automatically verify the runtime behaviors of DP instances in Java systems. For automatic DP detection, a previous work [18] transforms software artifacts into a knowledge graph (KG) [27] by parsing AST (Abstract Syntax Tree), which represents the code artifacts for static analysis. We utilize the KG as the fundamental static facts for relation inference and code artifact search. By extending their approach with temporal analysis, the main contributions of this paper are:

- First, a markup language, TSML, is proposed to reproduce the usage scenario of a DP. Based on TSML, a test script puts a candidate instance into use by introducing marked-up fragments within a normal piece of test code. The marked-up fragments direct the generation of test cases by not only indicating the search boundary of code artifacts, but also enabling us to apply different search strategies.
- Second, we present a trace method to monitor runtime events without instrumentation. The underlying technique supports a wide range of runtime events and enables us to specify events of interest with regular expressions.
- Third, a modeling and specification method of runtime behaviors is introduced by employing interval-based temporal relations, based on which our understanding of participating roles' temporal behaviors at runtime can be expressed flexibly in behavior specifications.

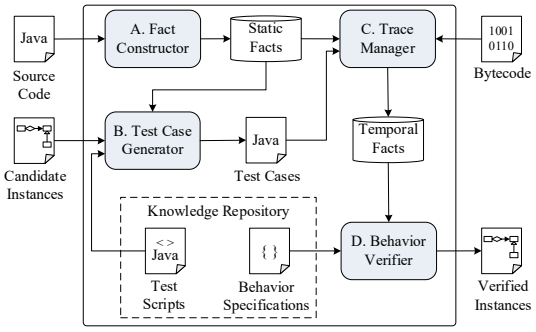

Fig. 1. Overview of the approach

The evaluation results of the prototype tool SparT-ETA (Software Architectural Pattern Recognition Tool Enhanced by Temporal Analysis) show that the temporal analysis improves the F_1-score from 24.2% to 77.8% in distinguishing similar instances. Compared with ePAD [3], another automatic approach based on dynamic analysis, (SparT-) ETA achieves better F_1-score (78.4% vs. 59.6%) on the systems and DPs considered by both approaches.

The paper is structured as follows. Section II presents the approach. After we set up the empirical evaluation in Section III, the evaluation results are analyzed in Section IV. Then we discuss the practical observations of SparT in Section V. Following the debate of threats to validity in Section VI, the discussion of related approaches in Section VII. Finally, the conclusions and future work are presented in Section VIII.

II. Approach

As presented in Fig. 1, the proposed approach contains four main modules. The Fact Constructor (Section II-A) transforms the source code into a KG that serves as the Static Facts of code artifacts. The Test Case Generator (Section II-B) parses a Test Script (marked up with TSML) according to a given DP instance and generates a set of Test Cases by searching relevant code artifacts and fulfilling marked-up fragments. A generated test case is a runnable piece of Java code aiming to put a DP instance into use. The execution of test cases is controlled by the Trace Manager (Section II-C) that is composed of a Trace Agent and a Log Parser. The former takes a regular expression as input to specify the events of interest, then outputs trace logs, while the latter parses log lines to create Temporal Facts. The temporal facts are integrated with static facts by linking retrieved events back to static facts. Finally, the Behavior Verifier matches the Temporal Facts with the Behavior Specifications to verify whether a DP instance acts as expected (Section II-D). The approach proposed in [18] formally defines DPs using description logic to constrain involved roles and operations. Further, DP detection is to satisfy the formal definition. To separate concerns, the KG is organized as a layered structure that

```
01 package example;
02
03 public class Door {
04   private DoorState state;
05
06   public Door() { setState(new ClosedState()); }
07   public Door(DoorState s) { setState(s); }
08
09   public void touch() { state.touch(this); }
10   protected void setState(DoorState s) { state = s; }
11
12   public static void main(String[] args) {
13     DoorState open = new OpenState();
14     Door door = new Door(open);
15     door.touch();
16   }
17 }
```

Fig. 2. A Door with a one-touch button that controls the Door's state: closed or open. The door can be constructed with a default (closed) state (line 06) or a given state (line 07)

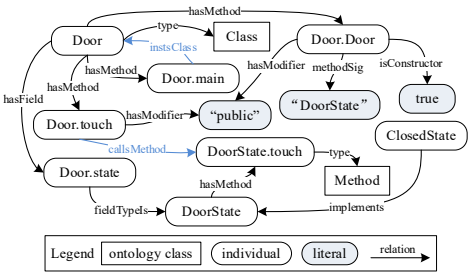

Fig. 3. The knowledge graph (KG) that involves Door and DoorState (The KG presented here is a part of the whole system under study. It is simplified for brevity, thus not all relevant individuals and relations are shown. The individual names are also in omitted form. In practice, we use the fully qualified names, e.g., "example.Door.touch" instead of "Door.touch".)

supports customizable inference rules and pattern templates. Based on the clarified roles and operations in the DP definition, the Test Scripts and Behavior Specifications focus on a DP instance's usage scenario and behavioral feature respectively. Together with the inference rules and pattern templates, the Test Scripts and Behavior Specifications exist as pluggable components of the Knowledge Repository (KR) that enables us to represent and share the experience of DP detection.

A. Knowledge Graph Construction and Inference

Fig. 1 shows an excerpt from a sample code [28] [29] of the State pattern in the textbooks [26] [30]. The class Door and the interface DoorState play the Context role and State role respectively. Corresponding to the sample code, generated facts are illustrated by the KG in Fig. 2, where code artifacts are represented as *language constructs* (e.g., class and method) and their *relations* (e.g., "extends" and "implements").

The constructs we extract include Class, Method, Package, Interface, Enumeration, Primitives (e.g., boolean), Parameter (method parameter), Field, and TypeParam (type parameter). The relations include *extends*, *implements*, *hasField* (field of a class), *hasParam* (parameter of a method), and *methodSig* (signature of a method), etc. Two main static behaviors that we consider are class instantiation (*instsClass*) and method call (*callsMethod*). Class instantiations can be extracted from the "new" operation ahead of a class (e.g., line 06, Fig. 1). Method calls can be extracted from statements, e.g., at line 09, Fig. 1 where Door.touch() calls DoorState.touch().

Based on these constructs and relations, two main steps to build a KG are: *extracting individuals* and *fulfilling relations*. An individual is one type of a language construct, e.g., Door (Fig. 1) is a type of Class. For automatic processing, the facts in the KG are manipulated in the form of triples:

(subject *property* object)

In a triple, a subject can be an individual; a property refers to a relation; and an object can be an individual or a literal. To extract individuals, each construct is enumerated to retrieve corresponding artifacts from source code. For example, the following facts are created for the construct Class: (Door *type* Class) and (ClosedState *type* Class), etc., where "Door" and "ClosedState" are created individuals. The relation *type* indicates that the individual is a type of Class. After all individuals of each construct are created, the relations are enumerated to associate these individuals. For example, to fulfill *hasMethod*, each individual of Class and Interface is connected to their containing methods. As a result, one of the generated facts is (Door *hasMethod* Door.touch). Finally, the KG is built, which contains the facts that represent the system under study.

Then an inference process is conducted to summarize indirect relations, which facilitate the search of code artifacts in subsequent steps. Examples of inferred relations are *inherits*, *realizes*, and *isA* that describe different relationships of classes and interfaces. While *inherits* and *realizes* describe multilayer inheritance, *isA* describes safe up-casting. Given x and y, each of which is a class or an interface, the relations are defined as:

(x *inherits* y), (x *isA* y)←(x *extends*+ y)
(x *realizes* y), (x *isA* y)←(x *implements* / *extends** y)
(x *isA* y)←(x *extends*+ / *implements* y)

The rules are based on SPARQL (Simple Protocol and RDF (Resource Description Framework) Query Language) grammar [31], where "/" connects two relations. For example, (x *implements* r) and (r *extends* y) can be expressed as (x *implements*/*extends* y) (an interface can extend another interface in Java). The "*" ("+") is equivalent to the connection of zero (one) or more relations ahead.

Inferred relations enable us to express complex structures in a concise way. For example, specifying (s *isA* DoorState) highly synthesizes possible variants of DoorState's descendant type "s". The inference technique also enables us to search relevant code artifacts according to the mechanisms of Java programming. For example, the static method call relation does not decide the real type of participants. Namely, if the field "Door.state" holds an object of DoorState's subclass (e.g., ClosedState) that overrides "DoorState.touch()", the actually called method will be "ClosedState.touch()". In spite of this, the actual methods possibly called at runtime can be located by searching overriding methods along inheritance relations. The override relation between two methods can be inferred from the methods' modifier, name, and signature. We employ the algorithms proposed in [18] for fact generation and relation inference. A description of all the constructs and relations we use can be found in online Appendix A [32].

B. Test Case Generation

A test script of the State pattern is presented in Fig. 3. The test script plays the client role and reproduces a usage idiom of the State pattern corresponding to Fig. 1 (line 13-15). A test script is a piece of normal code marked up with TSML blocks. Each type of TSML block is enclosed with a different pair of

```
01 package test.auto;
02
03 public class          (a)                    (b)          (c)
04   State_<name: {role: StaCtxRole}>_<name: {role: StaRole}>_<seq:>
05 {
06   public static void main(String[] args) {   (e)          (f)
07 (d) {role: StaRole} state = new [concrete: {role: StaRole}] (:);
08 (g) {role: StaCtxRole} context = new {role: StaCtxRole} (:state);
09     context.{oper: CtxReqOper} (:);          (h)          (i)
10   }              (j)       (k)
11 }
```

Fig. 4. A test script of the State pattern based on TSML (StaCtxRole: Context role of the State pattern, StaRole: State role, CtxReqOper: Request operation of the Context role)

brackets ({ }, [ ], ( ), or < >). The four types of blocks are pattern block (Fig. 3.d), search block (Fig. 3.e), function block (Fig. 3.a), and argument block (Fig. 3.f). Within each pair of brackets, a block is formatted as:

instruction: *parameter1*, *parameter2*, …

An instruction indicates a block's action taking the parameters that follow a colon (:) as inputs. Both instructions and parameters can be omitted. Nested blocks are allowed as an outer block's parameter (Fig. 3.a). Given a DP instance, the test cases are generated by enumerating the values of marked-up blocks.

The *pattern block* returns specified roles or operations according to its instruction, "role" or "oper". The instructions refer to the participating roles and operations of a DP instance. Assuming the given State instance is illustrated in Fig. 1, a value of the pattern block Fig. 3.d will be "example.DoorState". The generator enumerates the blocks' values in the order that they appear. During a walk-through of all blocks, their values are decided one after another. The "role" instruction uses the context information of decided blocks. For the first appearance of {role: StaRole} (Fig. 3.b), all possible values are prepared for subsequent enumeration, one of which is consumed for the current walk-through. When it comes to the same block in Fig. 3.d, the block will directly return the value previously decided. For the "oper" instruction (Fig. 3.j), the block value depends on not only its parameter, but also the type of the variable that invokes the operation.

The *search block* applies different strategies to search for code artifacts. An example is the "concrete" instruction in Fig. 3.e. It returns the classes that can be allocated by a "new" operation. Its parameter is the class or interface to which the returned classes can be safely up-casted. This instruction is executed by applying the *isA* relation. The *function block* modifies the value of a parameter block. The "name" instruction (Fig. 3.a) returns a local name of the input parameter. The "seq" instruction (Fig. 3.c) generates unique serial numbers during the output of test cases. The *argument block* represents the arguments passed to a method (or constructor). Given a method M, which is provided by the previous block, the argument block decides the values according to M's signature. The argument decision algorithm (detailed in Appendix B [32]) includes three steps: (i) searching the methods whose name equals to M and parameter types match the argument block's input parameters, (ii) deciding the positions of the argument block's parameters since their positions can be specified arbitrarily in the test script, and (iii) providing values to each parameter type of M and initializing allocated objects based on closure.

Algorithm 1 Test case generation algorithm

```
01  procedure generateTestCases(dpInstance,
      testScript, kg, outputPath)
02    context := new Context(dpInstance, kg)
03    blocks := testScript.parseBlocks()
04    stack := new Stack()
05
06    valIter := blocks[0].valueIterator(context)
07    stack.push(valIter)
08
09    blockValues := new BlockValue[blocks.size()]
10    while stack.size() > 0 then
11      last := stack.lastElement()
12      if last.hasNext() then
13        blockValues[stack.size()-1] := last.next()
14      else
15        stack.pop()
16        continue
17      end if
18      if stack.size() < blocks.size() then
19        if not hasSucceedingValues(context) then
20          continue
21        end if
22        nextBlock := blocks[stack.size()]
23        valIter := nextBlock.valueIterator(context)
24        stack.push(valIter)
25      else
26        testScript.generate(blockValues, outputPath)
27      end if
28    end while
29  end procedure
```

By abstracting the decision of block values as an iterator, the test case generation algorithm is presented in Algorithm 1. To record the context information during a walk-through of all blocks, a context object is constructed (line 02) taking the DP instance (dpInstance) and KG (kg) as inputs. Each block creates an iterator (line 06, 23) according to context information to prepare the values of the block. If nested blocks exist, the outer block clones the context object to nested blocks. The outer block recursively retrieves the values of nested blocks, so as to decide its own value. Before the walk-through, all outer blocks (shaded ones in Fig. 3) are indexed (line 03). A walk-through decides the values of all outer blocks in sequence and saves decided values to an array (line 09). During the parsing of blocks (line 03), a block is connected to relevant blocks to retrieve the decided value of that block, e.g., a variable's type. The walk-through (line 10-28) uses a stack (line 04) to traverse all possible combinations of outer blocks' values. Starting with the iterator of the first block (line 06-07), one value is decided at a time (line 13 and 22-24). If no value is available, the stack pops the iterator and switches to the previous iterator (line 15-16). Once all block values are decided, a test case is generated by replacing the blocks with decided values (line 26). Otherwise, no test case is generated since no combination of the values is possible.

The generation algorithm employs several strategies to prune unreachable and error branches. Before continuing to decide the next block, succeeding values are checked based on already decided values (line 19). If one of the blocks after the current block is not possible to return a value, the current value is abandoned and the walk-through turns to the next value of the current block (line 20). Another strategy is to pre-execute the generated test case and exploit the exception stack (e.g., null pointer exception) to suppress the values returned by relevant blocks during the subsequent generation.

To support variant usage scenarios as demonstrated in textbooks [8] [26] [30], multiple test scripts can be created independent of the generation algorithm. The instructions of

```
01 method_entry: Lexample/Door;main
02   method_entry: Lexample/OpenState;<init>, obj_tag: 1
03   method_exit: Lexample/OpenState;<init>
04   method_entry: Lexample/Door;<init>, obj_tag: 2, param_obj_tag: 1
05     method_entry: Lexample/Door;setState, obj_tag: 2,
   param_obj_tag: 1
06       field_modi: Lexample/DoorState;state, dec_cls:
   Lexample/Door;, in_method: Lexample/Door;setState, field_obj_tag:
   1, owner_obj_tag: 2
07     method_exit: Lexample/Door;setState
08   method_exit: Lexample/Door;<init>
 ~   ...
18 method_exit: Lexample/Door;main
```

Fig. 5. A fragment of the trace log during the execution of example.Door (The full log is available in [32]. The line numbers *do not* correspond to Fig. 1.)

blocks also can be easily extended since they are implemented outside the generation algorithm in a function-like way, which takes input parameters and returns values.

C. Runtime Trace and Behavior Modeling

Driven by test cases, the program is executed and traced. We implemented a trace agent to monitor runtime events. During the trace, the agent records key-value pairs in a log file in predefined format when notified of interested events. The agent supports a wide range of events, including method entry, method exit, field modification, and object allocation (a full list is available at [33]). Fig. 4 presents a fragment of the trace log during the execution of example.Door (Fig. 1) by specifying the command:

java -classpath {CLASSPATH} "-agentpath:{AGENTPATH}=method=^Lexample/#field=^Lexample/" example.Door

{CLASSPATH} represents the path of bytecode, while {AGENTPATH} represents the path of the agent. The string after {AGENTPATH} specifies the events to record using regular expressions. It means to capture events of methods and fields (separated by "#") whose JNI (Java Native Interface) type signature [34] starts with "Lexample/". The letter L is the JNI type signature of a fully qualified class. Thus, "Lexample/Door" (line 01, Fig. 4) refers to "example.Door". Each log line records multiple key-value pairs separated by ",". The first pair is an *event-value pair* and the rest pairs are *property-value pairs*. Method (field) names are recorded after a semicolon (;) in method (field) events. The method "<init>" is the initialization method of a class [35]. It is supplied by a Java compiler. We use this special method to identify object allocations.

To express runtime behaviors, we model the events based on Allen's interval-based temporal relations [36] [37]. The 13 binary relations between two intervals are: *before*, *after*, *meets*, *metBy*, *overlaps*, *overlappedBy*, *finishedBy*, *finishes*, *contains*, *during*, *startedBy*, *starts*, and *equals*. (They are displayed in Appendix C [32].) Exactly one of the relations holds, given a pair of excluding point-intervals. The interval of an event is determined by a pair of time instants, i.e., the timepoint it starts (*startsAt*) and ends (*endsAt*). Therefore, we can infer the temporal relation between two events given two pairs of time instants, e.g., (*t1*, *t4*) and (*t2*, *t3*), as presented in Fig. 5. Since the temporal facts are linked back to static facts, additional information is available, which provides traceable clues to characterize language constructs and relations.

To generate temporal facts, a trace log is transformed using a globally incremental timepoint for each log line. A method invocation event is created from a pair of method entry and

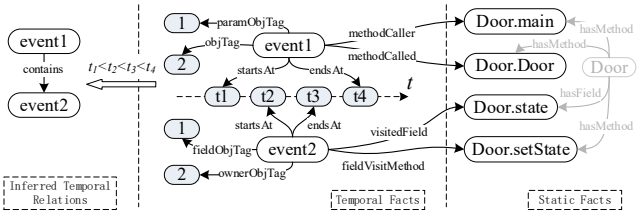

Fig. 6. Inferring the temporal relation between event1 (object allocation, line 04-08, Fig. 4) and event2 (field modification, line 06, Fig. 4) (The axis "t" indicates the flow of time. The static facts refer to Fig. 2.)

method exit events. The event *startsAt* the timepoint of a method entry (e.g., line 04, Fig. 4) and *endsAt* the timepoint of corresponding method exit (line 08). While some records, e.g., field modification (line 06), only occupy one log line, they consume two timepoints. For each log line, an individual is created from the event-value pair for the corresponding event. For line 04, e.g., the triple (event1 *methodCalled* Door.Door) is generated. "event1" is the individual created for the event. For each following property-value pair, a triple (event1 *property* value) is also generated, e.g., (event1 *objTag* 2). The property *objTag* (i.e., obj_tag) refers to the globally unique identifier of the object that owns the invoked method. The property "param_obj_tag" refers to the object passed to a method as a parameter. In a field modification event, the event individual is also linked to the method where the event happens (in_method, i.e., *fieldVisitMethod*), the field's declaring class (dec_cls), the owner object of the field (owner_obj_tag, i.e., *ownerObjTag*) and the modified value (field_obj_tag, i.e., *fieldObjTag*).

The dynamic analysis has the advantage to summarize implementation variants, regardless of, e.g., whether "Door.Door" (line 07, Fig. 1) is implemented as "setState(s);" or "state=s;". Both cases achieve the same purpose of initializing "Door.state" within "Door.Door".

D. Behavior Specification and Verification

A behavior specification describes expected behaviors of a DP. As we manipulate the KG with triples, the behavior specification is also in the form of triples. For example, the events "an object Q's method is invoked passing an object R as a parameter, during which a field of Q is set to hold R" can be specified as:

```
?e1 methodCalled ?method; objTag ?objQ; paramObjTag ?objR.
[] hasMethod ?method; hasField ?field.
?e1 contains ?e2. # or "?e2 during ?e1."
?e2 visitedField ?field; fieldObjTag ?objR; ownerObjTag ?objQ.
```

In a triple, the subjects or objects started with "?" are named variables; a "[ ]" refers to an unnamed variable; a ";" connects two triples with the same subject; a "." separates different triples; and a "#" starts a comment. By matching the specification with temporal facts, the variables will be bound to proper values if such events exist (e.g., ?e1 is bound to event1 in Fig. 5). Otherwise, the temporal facts do not comply with the specification if no such event is found.

A behavior specification aims to capture the key behavior features that distinguish different instances. In the example of Fig. 1, we expect to toggle the door's state by touching the control button. While we specify the behavior sequence "the control button is touched, then the door's state is toggled" as the

key behaviors, the triggering of key behaviors may depend on specific conditions. Touching the control button may not close an open door in the case that someone is walking through the door. In spite of this case, the door has the ability to exercise its full functions. The test case generation algorithm (Algorithm 1) enumerates possible combinations of block values, expecting a test case exists to trigger the key behaviors. Therefore, we define a verified instance as:

Definition 1. An instance of a DP is verified if there exists a test case whose runtime behaviors comply with the behavior specification of the DP.

As illustrated in the textbooks [8] [26] [30] and demonstration projects (TCPConnection [8], Door2 [28], and StatePatternEx [29]), the State pattern intends to allow an object to alter its behavior when its internal state changes. A Context usually encapsulates a State as its field to record its internal state. The request delivered to Context is delegated to State, thus the Context's behavior alters when the state changes. A Client usually does not directly operate State to perform the request but uses the interface provided by the Request operation of Context. Thus we specify "a request delivered to Context triggers the transition of state" as the key behaviors of the State pattern.

Initially, a Context instantiates a default State as presented at line 06, Fig. 1 [28]. A variant [29] is to provide a State object to Context as presented at line 07, Fig. 1. Who defines the state transitions is not forced in the State pattern as noted in the GoF textbook [8]. The textbook suggests the successor state can be specified either in Context (Appendix D.1) or State (Appendix D.2 [32]). By covering these variants, the behavior specification of State considers the following behavior features in addition to the delegation of request: (i) the temporal relations of instantiating Context and State, and (ii) the state of Context changes during the request delivered to Context. For (i), the State can be instantiated *before* or *during* the instantiation of Context. The State object also can be initiated in the field declaration of Context (Door2 [28]), e.g., initiating at line 04, Fig. 2 with the statement "state = new ClosedState();". Since field initiation is processed within the special method <init> at runtime, initiating the State object in field declaration acts the same as providing a default State object in the constructor of Context. For (ii), a state transition exists *during* the request, regardless of who defines the state transition.

For Strategy, which is almost structurally identical to State, the main differences with State are noted in the textbook [26]. First, "Strategy might allow a client to select or provide a strategy, an idea that rarely applies to State". To characterize this feature, in addition to providing the Strategy object through Context's constructor, the specification allows a setter operation of Context. A setter sets a strategy taking a Strategy object as input. As demonstrated in [29] (StrategyPatternEx), the setter is invoked *after* the instantiation of Context and State. The Context object is set to hold the State object *during* the invocation of the setter. Second, "state transitions are important when modeling State but strategy transitions are usually irrelevant when choosing Strategy". Based on this feature, in contrast to the State pattern, Strategy is specified as no strategy transition exists during the request, as presented in [8] (Composition), [28] (Customer2), and [29] (StrategyPatternEx). Based on these observations, behavior specifications allow us to encode our experience in identifying DP instances. The behavior specifications of Strategy and State, which support the mentioned variants, are detailed in Appendix E [32].

III. Empirical Evaluation Setup

In this section, we set up the evaluation of the prototype tool ETA. The empirical study follows the Goal Question Metric guidelines [38].

A. Context Selection

The *goal* of the evaluation was to assess the proposed approach for the *purpose* of improving existing approaches *with respect to* the accuracy of automatic DP detection from the *viewpoint* of system developers and maintainers who intend to employ the proposed approach by assessing its accuracy. The evaluation context consisted of six open source systems and five DP detection approaches with respect to five GoF DPs. The evaluation was performed on six systems (TABLE I), i.e., JHotDraw 5.1 (JHD), JUnit 3.7 (JUN), JRefactory 2.6.24 (JRF), QuickUML 2001 (QUM), PMD 1.8 (PMD), and MapperXML 1.9.7 (MPX). We selected these systems because (i) they are open source systems whose source code is publicly available, (ii) they have been studied by existing approaches in the literature [2] [3] [10] [11], and (iii) they are implemented in Java since our approach focuses on Java systems.

The five existing approaches involved in the evaluation are RaM [10], DPD (Design Pattern Detection) [2], DPF (Design Pattern Finder) [11], ePAD [3], and SparT (Software Architectural Pattern Recognition Tool) [18]. We chose these approaches since they report detailed detection results rather than only instance numbers. While RaM, DPD, and DPF focus on all the 23 GoF patterns, ePAD focuses on 12 creational and behavioral patterns, and SparT focuses on 22 GoF patterns except Façade. All the approaches report the instances of JHD, JUN, and JRF, while only DPD does not consider QUM; only DPF and SparT consider PMD; and only ePAD and SparT consider MPX. We also noticed other approaches, including the ones proposed in [15], [21], [39], [40], and [41]. We did not choose them since the systems used were unavailable [21] [39] [40] or only instance numbers were reported [15] [41].

The evaluation focused on five GoF patterns, i.e., Strategy, State, Bridge, Command, and Template Method. We chose them since they occupied the most proportion of false instances based on the investigation of the five approaches.

B. Research Questions and Metrics

To address our goal, the evaluation aimed to answer the following research questions (RQ):

RQ1. What is the accuracy of ETA in improving existing approaches?

RQ2. How is the accuracy of ETA compared with existing approaches based on dynamic analysis?

While RQ1 focused on the accuracy in verifying candidate instances to avoid false instances and keep true instances, RQ2 aimed to assess the accuracy compared with other dynamic approaches. As ETA aimed to improve the detection results by verifying existing instances, to answer RQ1, ETA took all the

TABLE I. SYSTEMS CONSIDERED IN THE EVALUATION

ID	System	#Files	#LoC [a]	#Classes	#Methods
1	JHotDraw 5.1 (JHD)	144	8,419	173	1,332
2	JUnit 3.7 (JUN)	78	4,886	157	714
3	JRefactory 2.6.24 (JRF)	569	55,871	575	4,865
4	QuickUML 2001 (QUM)	156	9,249	228	1,096
5	PMD 1.8 (PMD)	446	41,321	505	3,680
6	MapperXML 1.9.7 (MPX)	217	14,372	263	2,110

[a] LoC: Lines of Code (excluding comment lines and blank lines)

true and false instances reported by the five approaches as input candidate instances with respect to the five selected DPs.

While RaM, DPD, DPF, and SparT aim to detect DP instances in source code based on static analysis, ePAD employs dynamic analysis to further verify the instances reported by its static phase. Therefore, to address RQ2, we compared ETA with ePAD in accuracy. The comparison was carried out on four of the five selected DPs, i.e., Strategy, State, Command, and Template Method, since ePAD does not focus on Bridge. We also noticed other approaches based on dynamic analysis. ETA was not compared with them since they are evaluated on sample systems whose source code is not available [21] [22] [42], or only instance numbers are reported [1] [15] [25] [41].

To assess the accuracy, we employed *precision*, *recall*, *F_1-score* [43], and *MCC* (Matthews Correlation Coefficient) [44]. The *precision* measures the fraction of relevant instances among all detected instances, i.e., $|TP| / |TP \cup FP|$, where *TP* (True Positive) is the set of true instances and *FP* (False Positive) is the set of false instances. The *recall* measures the fraction of relevant ones in detected instances among all relevant instances, i.e., $|TP| / |TP \cup FN|$, in which *FN* (False Negative) is the set of missed instances. The harmonic average of *precision* and *recall* is measured by *F_1-score*, i.e., $2 \cdot precision \cdot recall / (precision + recall)$. To assess the verification of candidate instances as a binary classification problem, the *MCC* has the advantage over *F_1-score* to balance *TP*, *TN* (True Negative), *FP*, and *FN*. It can be calculated as $(|TP| \cdot |TN| - |FP| \cdot |FN|) / ((|TP| + |FP|) \cdot (|TP| + |FN|) \cdot (|TN| + |FP|) \cdot (|TN| + |FN|))^{1/2}$, where *TN* is the set of actual negative instances among unverified instances.

Over recent decades, several benchmarks have been built to evaluate DP detection results. We adopted the benchmarks published in [2], [3], [10], [11], and [18] to evaluate the accuracy. For the instances disagreed between these benchmarks, we followed the validation procedure proposed in [3] to decide each instance. First, all the instances disagreed between the benchmarks were collected. Second, three Ph.D. students independently analyzed the documentation, source code, and online resources to validate these instances. At last, for the instances that did not reach a consensus among the full group, a discussion was conducted. An instance is considered as true only if the full group agreed.

C. Experimental Setup

ETA was fully automated taking source code, bytecode, and candidate instances as inputs. ETA was implemented based on Jena [45], a Java framework for building linked data applications. It supports manipulating triples with SPARQL. The static facts were automatically generated using JDT (Java Development Tools) [46] through AST parsing. Then the trace agent employed JVM TI (Java Virtual Machine Tool Interface) [33] to monitor the execution of target systems. The test scripts and behavior specifications of the five DPs were built according to the textbooks [8] [26] [30] and demonstration projects [28] [29]. We also studied the source code of JHD since its documentation is available, which reduces the misunderstanding of the system. Finally, the compliance verification between behavior specifications and temporal facts was performed through triple matching. The evaluation was conducted on a desktop computer (Intel Core i7 4790@3.6GHz, 8GB RAM).

IV. ANALYSIS OF EVALUATION RESULTS

In this section, we analyze the evaluation results in response to the research questions.

A. RQ1. What Is the Accuracy of ETA in Improving Existing Approaches?

ETA verified 466 candidate instances reported by the five approaches. TABLE II aggregates the instances of each system. Candidate instances (C) include all the true and false instances, the true instances of which refer to the benchmark (BM). While verified instances (V) are the instances verified by ETA, the true positives (TP) refer to the true instances among verified instances. Since RaM, DPD, and SparT treat Strategy and State as the same pattern, in TABLE II, the instance numbers of the two patterns are counted respectively by regarding each of the two patterns as reporting the same instances.

Based on TABLE II, TABLE III compares the accuracy of candidate (Candidate) and verified instances (Verified) in terms of precision (P), recall (R), and F_1-score (FS) with respect to each pattern. (Detailed results are available online [32].) The recalls of candidate instances are always 100% since candidate instances include all true positives. There is no Bridge instance in the benchmark, thus the recall and F_1-score of Bridge are not available (/). Although the precisions of Bridge are zeros, the verification reduces 29 false candidate instances to one.

Among 312 Strategy and State instances, 280 ones are false instances; 99 ones are reported as both Strategy and State (i.e., overlapped); 32 ones are true instances. Through the dynamic analysis, 97% (271 of 280) false instances are avoided; 75% (24 of 32) instances are classified to proper DPs; finally, no overlapped instance exists. For Bridge, Command, and Template Method, 97% (118 of 122) false instances are avoided, four of which are actually Proxy or Adapter instances. In total, TABLE III shows that ETA achieves a precision of 79.0% and a recall of 76.7%. The overall F_1-score is 77.8%; the overall MCC is 0.74. From TABLE II, we can observe 13 false positives, i.e., the ones verified (V) but not among benchmark (BM); we can also observe 15 false negatives, i.e., the ones among benchmark but not verified. In the rest of this section, we analyze the evaluation results to identify why ETA failed at these instances. The main reasons (*RS*) are summarized by *RS1-4* below. While *RS1* and *RS2* explained the 13 false instances, *RS3* and *RS4* explained the 15 missed instances.

RS1. Relevant objects interact based on the same mechanisms as employed by a DP, but with different purposes.

The behaviors of 11 instances act similarly as a DP. For four Strategy instances (two of JHD and two of JRF), the interacting objects are composed by delegating the request from one object

TABLE II. CANDIDATE AND VERIFIED INSTANCE NUMBERS OF THE FIVE DESIGN PATTERNS IN THE SIX SYSTEMS

System			Str. [c]		Sta.		Bri.		Cmd.		TM.	
JHD [b]	***C***	***BM***	47	6	45	6	28	0	24	13	11	5
	V	***TP***	8	6	2	2	1	0	12	12	3	3
JUN	***C***	***BM***	18	2	10	1	0	0	0	0	6	1
	V	***TP***	1	1	1	1	0	0	0	0	1	1
JRF	***C***	***BM***	52	0	34	2	0	0	23	0	31	6
	V	***TP***	2	0	0	0	0	0	0	0	6	6
QUM	***C***	***BM***	18	0	10	1	0	0	8	0	9	4
	V	***TP***	0	0	0	0	0	0	0	0	2	2
PMD	***C***	***BM***	27	8	27	5	0	0	4	0	2	2
	V	***TP***	8	8	5	5	0	0	1	0	1	1
MPX	***C***	***BM***	12	1	12	0	1	0	0	0	7	1
	V	***TP***	6	1	0	0	0	0	0	0	2	0
(Total)	***C***	***BM***	174	17	138	15	29	0	59	13	66	19
	V	***TP***	25	16	8	8	1	0	13	12	15	13

[b.] C: Candidate, BM: Benchmark, V: Verified, TP: True Positives in Verified Instances

[c.] Str.: Strategy, Sta.: State, Bri.: Bridge, Cmd.: Command, TM.: Template Method

to the other, but not for encapsulating algorithms as in the Strategy pattern. Five Strategy instances of MPX are involved in the document (e.g. MDocument) that composites discrete models (e.g. TextModel) for the purpose of data manipulation. A model also composites a value holder (e.g. ListValueHolder) to save the model's data. Two Template Method instances of MPX employ abstract methods in the superclass to defer an operation to its subclass, but not to provide a primitive operation of an algorithm like the Template Method pattern.

RS2. The instances can be distinguished in structure, while ETA focuses on the interactions of runtime objects.

One Bridge instance of JHD (Abstraction / Implementor roles: DecoratorFigure / Figure) is actually a Decorator instance. Bridge and Decorator are partially similar in structure. Both the Abstraction role of Bridge and the Decorator role of Decorator delegate a request to the aggregated objects, except that Decorator aggregates its superclass. The aggregation of a superclass is a unique feature that can be utilized to distinguish Decorator from Bridge. ETA does not identify this difference since it focuses on runtime behaviors to verify candidate instances. For one Command instance of PMD (Command / ConcreteCommand / Receiver roles: ViewerModelListener / ASTPanel / ViewerModel), it is actually an Observer instance. The structure of the Command pattern's Command / ConcreteCommand / Receiver roles is similar to the structure of the Observer pattern's Observer / ConcreteObserver / ConcreteSubject roles. The difference is, while a Command instance does not force the existence of an Invoker role, the Subject role of Observer keeps the references to multiple observers and, meanwhile, exists as the superclass of the ConcreteSubject role. This structural difference can be used to distinguish Observer from Command.

RS3. The generated test cases do not meet the conditions to trigger expected behaviors.

Thirteen missed instances are involved in two situations: (i) expected behaviors rely on a series of operations, but the test cases do not successfully reproduce them, and (ii) no concrete class is available to allocate an object. Situation (i) explains 10 missed instances (one Strategy, five State, one Command, and three Template Method instances). An example is the instance of JHD, ConnectionTool / Figure (Context / State roles). The ConnectionTool is used by a drawing application to connect two figures with an arrowed line. It aggregates a Figure object to track the Figure under the mouse pointer during mouse action. To exercise the behaviors of ConnectionTool / Figure, the operations are to create two figures on the drawing palette, press the left mouse button on one of the figures, and drag the arrowed line to the other figure. The test case generator can hardly simulate the series of mouse events with proper event types at proper palette locations. Situation (ii) explains the other three Template Method instances, e.g., NodeIterator (AbstractClass role) in PMD. No class is available to allocate the object for an instance's role or an argument block's argument.

TABLE III. THE ACCURACY OF CANDIDATE AND VERIFIED RESULTS IN TERM OF PRECISION (P), RECALL (R), F_1-SCORE (FS) (IN PERCENTAGE), AND MATTHEWS CORRELATION COEFFICIENT (MCC) (IN DECIMAL)

Design Pattern	Candidate			Verified			
	P	***R***	***FS***	***P***	***R***	***FS***	***MCC***
Strategy	9.8	100.0	17.8	64.0	94.1	76.2	0.75
State	10.9	100.0	19.6	100.0	53.3	69.6	0.71
Bridge	0.0	/	/	0.0	/	/	/
Command	22.0	100.0	36.1	92.3	92.3	92.3	0.90
Template Method	28.8	100.0	44.7	86.7	68.4	76.5	0.69
(Total)	13.7	100.0	24.2	79.0	76.6	77.8	0.74

RS4. Non-GoF variants exist, which are not considered by ETA currently.

The other two missed State instances are XMLLinePrinter / State and FileSummary / SummaryLoaderState (Context / State roles) of JRF. Their State objects appear as local variables inside a class method, thus are not shared with other operations of the Context object. We consider this variant as a non-GoF pattern since the structure of the GoF State pattern is based on aggregation, which implies that "an aggregate object and its owner have identical lifetimes" as noted in the GoF textbook [8].

In summary, while 13 instances are falsely verified (mainly due to *RS1* and *RS2*) and 15 instances are falsely avoided (mainly due to *RS3* and *RS4*), ETA increases the overall F_1-score by 53.6% (from 24.2% to 77.8%) (TABLE III).

B. RQ2. How Is the Accuracy of ETA Compared with Existing Approaches Based On Dynamic Analysis?

ETA focuses on the verification of candidate instances. It can be applied to any DP detection approach that reports DP instances since it does not depend on intermediate processes of the approaches. While ePAD includes a dynamic phase to verify the instances reported by its static phase, ETA verifies all the true and false instances reported by the five existing approaches (including ePAD). Both ePAD and ETA consider JHD, JUN, JRF, QUM, and MPX with respect to Strategy, State, Command, and Template Method. To answer RQ2, we compared ETA with ePAD in accuracy on these systems and DPs.

For the Command pattern, the final results of ePAD achieve better average accuracy than ETA (*P* / *R* / *FS*: 86.4% / 100.0% / 92.7% vs. 92.3% / 92.3% / 92.3%). ETA falsely verifies one instance due to *RS2* and misses one instance due to *RS3*, while ePAD reports more false instances. For the State pattern, ETA outperforms ePAD in average accuracy (P / R / FS: 100.0% /

53.3% / 69.6% vs. 38.3% / 94.7% / 54.5%). ETA misses seven instances due to *RS3* and *RS4*; ePAD achieves higher recall but reports more false instances. For the other two patterns, ETA achieves better precision and recall. The total recall of ePAD (84.9%) is better than ETA (76.6%), while ETA achieves better precision (80.3%) than ePAD (45.9%). Overall, ETA achieves better F_1-score than ePAD (78.4% vs. 59.6%).

C. Time Performance

For each candidate instance, the Trace Manager (Fig. 1.C) parallelized the verification by manipulating multiple threads to consume the test cases generated for the instance. According to Definition 1, if an instance is verified by one of the threads, the Test Case Generator (Fig. 1.B) stops generating and turns to another instance. The thread number was experientially set to eight. We did not tune the time performance since the evaluation focused on accuracy. The initial results are presented in TABLE IV. For 466 candidate instances (#Candidate), 9,845 test cases (#Test Case) were generated. The total time consumed was 39.13 minutes, including the time to generate and compile test cases (Generation), and to monitor program execution and verify candidate instances (Verification). Compared with the dynamic analysis phase of ePAD, which took 659.2 minutes to verify 258 candidate instances from its previous static analysis phase, ETA achieved better average time performance to process each candidate instance (5.03s vs. 153.30s).

TABLE IV. THE NUMBER OF CANDIDATE INSTANCES (#CANDIDATE), GENERATED TEST CASES (#TEST CASE), AND THE TIME TO GENERATE TEST CASES (GENERATION) AND VERIFY CANDIDATE INSTANCES (VERIFICATION) (IN WALL CLOCK MINUTES)

	JHD	JUN	JRF	QUM	PMD	MPX	(Total)
#Candidate	155	34	140	45	60	32	466
#Test Case	4,868	17	621	787	2,451	1,101	9,845
Generation	9.00	0.01	2.85	0.80	0.69	0.75	14.10
Verification	14.71	0.07	1.84	1.46	5.28	1.67	25.03
Total Time	23.71	0.08	4.69	2.26	5.97	2.42	39.13

V. DISCUSSION

Towards incremental understanding of legacy systems. Manually reviewing the code to identify DP instances is a time-consuming task. To reduce manual efforts, we gain insights into the selection and use of tools from measurements, each of which has its typical application scenario as an evaluation criterion. Automated tools with higher recall cover more actual instances, but the results will be misleading if too many false instances are reported. In practice, we progressively comprehend the code, guided by the instances suggested by tools as a starting point. However, we observed that the experience we gain in a context, e.g., project-specific naming conventions and individual developers' coding preferences, is not well utilized by automated tools because the tools generally make few assumptions about how DPs are applied. Towards incremental understanding, customizable tools that integrate multiple stages or employ multiple strategies are more feasible in gradually improving the precision of coarse-grained results.

SparT in practice. In a previous work, SparT models and represents software artifacts employing a KG, which supports various static analysis tasks such as query and inference. The KG resides in a layered structure to be extensible, under the blueprint of which SparT-ETA extends the ontology models to support dynamic analysis. Since the test scripts and behavior specifications [32] serve as pluggable components of the KR, they can be shared in the form of text files (e.g., Fig. 3) among developers and maintainers who employ SparT. To customize or create test scripts, it does not require additional experience for developers who are familiar with DPs and Java. Based on SPARQL, a W3C (World Wide Web Consortium) recommendation, the behavior specifications in the form of triples are also easy to understand and create even for beginners. While establishing KR requires manual efforts, we believe that developers will harvest from shared knowledge. Feedback from developers should be studied in the future to assess to what extent the harvests are worth the efforts. The latest updates of SparT will be available at the demonstration project online [47].

VI. THREATS TO VALIDITY

A threat that could affect the internal validity arises during the adoption of benchmarks. As manually recovering all the DP instances of each software system requires much effort and may introduce subjectivity, we adopted the benchmarks proposed in [2], [3], [10], [11], and [18]. The benchmarks are publicly available and maintained by the researchers over the years. They are widely studied in the literature [7] [14] [17] [48], which reduces human mistakes. To eliminate disagreed instances between these benchmarks, three volunteers analyzed the instances separately following the validation procedure proposed in [3]. To mitigate human error and subjective bias, the final results were decided through a discussion of the whole group. Although the three volunteers have years of experience in developing industrial Java systems, they may misunderstand the system due to the lack of documentation. Intended to broaden the discussion out to more researchers, the reasons why the group supported or did not support each instance are reported in Appendix F [32], which refers to existing documentation and code comments of the systems.

The selection of the systems could pose a threat to external validity. The systems were chosen since they have been widely studied in existing approaches and published detailed results, which enables a thorough comparison. To mitigate this threat, the evaluation context included all the six systems evaluated by the five approaches, while the common systems evaluated are JHD, JUN, and JRF. We also noticed various other systems considered by existing approaches, e.g., Swing [41], Ant [49], AWT [50], and Log4J [51]. We did not choose them mainly because: (i) different versions are used, which hinders the comparison of different approaches, (ii) the evaluated version is no longer available for download, or (iii) only instance numbers are reported. While these systems were not selected in the evaluation context, we maintained a website [47] to update the evaluation results of them. In consideration of replicating the evaluation on other systems, all available resources of the evaluation are published online [32], including the static facts, generated test cases, trace agent (with a double-click-to-run example), trace logs, dynamic facts, and detailed results.

VII. RELATED WORK

Over recent decades, various approaches have been proposed for DP detection. Based on an investigation of the papers in the literature in addition to existing mapping studies [6] [52] and surveys [12] [51] [53], this section discusses related work

regarding the techniques employed to address the challenges in DP detection. (The investigation involved 112 papers, a comparison of which is detailed in Appendix G [32].)

Aimed at capturing the structures and static behaviors, numerous techniques are employed to detect DP instances in source code and UML (Unified Modeling Language) diagrams, including graph matching [2] [7] [11], visual language parsing [13], similar matrix [54] [55], and machine learning [14] [56] [57] [58]. Existing approaches based on static analysis are limited in supported DPs. While the approaches cover most types of GoF patterns, several pairs of patterns are considered as similar, including Object Adapter / Command [2], Adapter / Bridge [8], Composite / Decorator [8] [55], Composite / Observer [41], Decorator / Proxy [8], State / Strategy [2] [26], and Strategy / Bridge [59]. Different approaches vary in performance. According to the detection results published in [2], [3], [10], [11], and [18], the top-five DPs that report the most false instances are Strategy, State, Bridge, Command, and Template Method. Only a few approaches support Strategy and State, but treat the two patterns as the same one and group their instances together in the detection results [2] [17] [18]. While several approaches focus on all GoF patterns [60] [48] [61] or declare to distinguish Strategy and State by employing machine learning [62] [63] or linguistic aspects of source code [64], no data package is provided in order to replicate the evaluation of these approaches.

To reduce false positives, dynamic analysis techniques are utilized to verify the runtime behaviors of DP instances from UML sequence diagrams [60] [55] or bytecode. The proposed approach focuses on the latter, aiming at legacy systems whose design information is not available. To exercise runtime behaviors, existing approaches use several methods to generate test cases, i.e., creating manually [15] [21] [22], utilizing test tools [20] [23], or proposing dedicated algorithms [3], instead of assuming test cases are available [1] [22] [40] [41]. Manual ways require considerable effort to study the systems under test, thus are limited in practical use. Although test tools (e.g., EvoSuite [3]) are able to generate test cases automatically, they are not suitable to exercise the behavior of a DP instance since they work at unit level according to a coverage criterion [3].

To solve the problem, ePAD, the first approach that automatically generates test cases to recover DP instances, is proposed in [3] and [23] by employing a genetic algorithm. While forcing the genetic algorithm to cover the right sequence of method calls is still too expensive, their approach approximately ensures that at least all the methods involved in an instance are executed. To take full advantage of a DP instance to exercise its own behavior, this paper presents a new method to generate test cases from the viewpoint of a DP's usage idiom. The proposed test case generation algorithm is more efficient in covering relevant method calls since a test script indicates the invoking sequence of participating operations. A test script allows us to express the experience of applying a DP in the same way as we write a normal piece of test code. Test scripts also can be easily extended to support DP variants and emerging DPs.

Driven by test cases, target systems are executed and monitored to obtain runtime data. Some approaches employ instrumentation tools to inject the source code (e.g., Recoder [41]) or bytecode (e.g., Probekit [3] [20] [23] [25]) with a fragment of code that collects runtime data. The Probekit tool belongs to the Test and Performance Tools Platform (TPTP) [65], a project of Eclipse. To avoid the side-effects of instrumentation [15] [24], profiling techniques under the Java Platform Debugger Architecture (JPDA) [22] [40] [66], including JVMDI (JVM Debug Interface) [24], JDI (Java Debug Interface) [1] [21] [67], and JVMPI (JVM Profiling Interface) [15], are widely used to inspect the state of running applications. JVMDI and JDI work at different layers of JPDA, i.e., back-end and front-end respectively, where JVMDI is closer to the virtual machine, thus has better performance and is able to use low-level functionality. Since JDK (Java Development Kit) 5.0, JVMDI and the experimental JVMPI are removed [68] and replaced by a new interface JVM TI [33]. In this paper, the trace agent employs JVM TI whereas other supporting tools and platforms, i.e., Recoder and TPTP, stopped updating [69] or terminated [65].

Finally, candidate instances are verified by matching obtained runtime data with behavior specifications based on Prolog programming [1] [15] [21], model checking [3] [19] [20], database query [22] [24] [40], or dedicated algorithms [25] [41] [67]. The proposed approach employs Allen's interval-based temporal logic [36] to model runtime events. The behaviors of DPs are specified based on SPARQL grammar without hard-coded algorithms.

VIII. Conclusions and Future Work

An approach has been presented to automatically verify design pattern instances, based on which the prototype tool SparT-ETA has been evaluated and compared with another dynamic approach. The evaluation results support its accuracy and time performance in improving existing approaches. The analysis of evaluation results in Section IV-A indicates the future work mainly in three aspects. First, in cases that DP instances act similarly but with different intents (*RS1*), one improvement direction is to exploit semantic and linguistic aspects integrating multisource information from code comments, documentation, and online resources. Second, the test case generator can be enhanced based on the assessment of the feasibility to simulate user operations in typical application scenarios such as drawing (*RS3*). Third, to support emerging design patterns (*RS4*), the knowledge repository can be extended based on the investigation of how widely these patterns are approved by the development community.

Acknowledgement

We would like to thank the reviewers' insightful suggestions which helped to improve this paper. We especially thank the generous help of Prof. Nikolaos Tsantalis (nikolaos.tsantalis@concordia.ca), Prof. He Jiang (jianghe@dlut.edu.cn), Prof. Li Li (li.li@monash.edu), and an anonymous researcher during the research and writing. This work was supported partially by the National Key R&D Program of China under Grant No. 2018YFB1003902, the National Natural Science Foundation of China under Grant No. 61572126, No. 61872078 and No. 61402103.

REFERENCES

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

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

[3] C. G. Andrea De Lucia, Vincenzo Deufemia and M. Risi, "Detecting the behavior of design patterns through model checking and dynamic analysis," *ACM Transactions on Software Engineering and Methodology*, vol. 26, no. 4, pp. 1–41, 2018.

[4] D. Q. Hou, "Using structural constraints to specify and check design intent in source code," in *Proceedings - IEEE International Conference on Software Maintenance*. Los Alamitos: IEEE Computer Society, 2006, pp. 343–346.

[5] X. Liu, L. Huang, C. Li, and V. Ng, "Linking source code to untangled change intents," in *Proceedings-IEEE International Conference on Software Maintenance*, 2018, pp. 393–403.

[6] B. B. Mayvan, A. Rasoolzadegan, and Z. G. Yazdi, "The state of the art on design patterns: A systematic mapping of the literature," *Journal of Systems and Software*, vol. 125, pp. 93–118, 2017.

[7] B. B. Mayvan and A. Rasoolzadegan, "Design pattern detection based on the graph theory," *Knowledge-Based Systems*, vol. 120, pp. 211–225, 2017.

[8] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design patterns: elements of reusable object-oriented software*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995.

[9] V. E. Zafeiris, S. H. Poulias, N. A. Diamantidis, and E. A. Giakoumakis, "Automated refactoring of super-class method invocations to the Template Method design pattern," *Information and Software Technology*, vol. 82, pp. 19–35, 2017.

[10] G. Rasool and P. Mäder, "A customizable approach to design patterns recognition based on feature types," *Arabian Journal for Science and Engineering*, vol. 39, no. 12, pp. 8851–8873, 2014.

[11] M. L. Bernardi, M. Cimitile, and G. Di Lucca, "Design patterns detection using a DSL-driven graph matching approach," *Journal of Software-evolution and Process*, vol. 26, no. 12, pp. 1233–1266, 2014.

[12] J. Dong, Y. Zhao, and T. Peng, "A review of design pattern mining techniques," *International Journal of Software Engineering and Knowledge Engineering*, vol. 19, no. 6, pp. 823–855, 2009.

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

[14] M. Zanoni, F. Arcelli Fontana, and F. Stella, "On applying machine learning techniques for design pattern detection," *Journal of Systems and Software*, vol. 103, pp. 102–117, 2015.

[15] H. Y. Huang, S. S. Zhang, J. Cao, and Y. H. Duan, "A practical pattern recovery approach based on both structural and behavioral analysis," *Journal of Systems and Software*, vol. 75, no. 1-2, pp. 69–87, 2005.

[16] I. Issaoui, N. Bouassida, and H. Ben-Abdallah, "Using metric-based filtering to improve design pattern detection approaches," *Innovations in Systems and Software Engineering*, vol. 11, no. 1, pp. 39–53, 2015.

[17] A. Alnusair, T. Zhao, and G. Yan, "Rule-based detection of design patterns in program code," *International Journal on Software Tools for Technology Transfer*, vol. 16, no. 3, pp. 315–334, 2014.

[18] R. Xiong and B. Li, "Accurate design pattern detection based on idiomatic implementation matching in java language context," in *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2019, pp. 163–174.

[19] A. De Lucia, V. Deufemia, C. Gravino, and M. Risi, "An eclipse plug-in for the detection of design pattern instances through static and dynamic analysis," in *Proceedings - IEEE International Conference on Software Maintenance*. New York: IEEE, 2010.

[20] ——, "Improving behavioral design pattern detection through model checking," in *European Conference on Software Maintenance and Reengineering*. Los Alamitos: IEEE Computer Society, 2010, pp. 176–185.

[21] S. Hayashi, J. Katada, R. Sakamoto, T. Kobayashi, and M. Saeki, "Design pattern detection by using meta patterns," *IEICE Transactions on Information and Systems*, vol. E91D, no. 4, pp. 933–944, 2008.

[22] F. Arcelli, F. Perin, C. Raibulet, and S. Ravani, "Jadept: Dynamic analysis for behavioral design pattern detection." Setubal: Insticc-Inst Syst Technologies Information Control & Communication, 2009, pp. 95–106.

[23] A. De Lucia, V. Deufemia, C. Gravino, and M. Risi, "Towards automating dynamic analysis for behavioral design pattern detection," *2015 31ST International Conference on Software Maintenance and Evolution (ICSME) Proceedings*, pp. 161–170, 2015.

[24] N. Pettersson, "Measuring precision for static and dynamic design pattern recognition as a function of coverage." St. Louis, MO, United states: Association for Computing Machinery, Inc, 2005.

[25] H. Lei and K. Sartipi, "Dynamic analysis and design pattern detection in java programs," in *Twentieth International Conference on Software Engineering & Knowledge Engineering*, 2008.

[26] S. J. Metsker and W. C. Wake, *Design Patterns in Java*. Boston, MA, USA: Pearson Education, Inc., 2006.

[27] H. Li, S. Li, J. Sun, Z. Xing, X. Peng, M. Liu, and X. Zhao, "Improving api caveats accessibility by mining api caveats knowledge graph," in *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2018.

[28] S. J. Metsker and W. C. Wake, "Source Code of the Book "Design Patterns in Java"," https://xp123.com/oozinoz/designpatternsinjava.htm, accessed: 2019-02-10.

[29] V. Sarcar, "Source Code of the Book "Java Design Patterns: A Tour with 23 Gang of Four Design Patterns in Java"," https://github.com/apress/-java-design-patterns, accessed: 2019-02-10.

[30] ——, *Java Design Patterns: A Tour with 23 Gang of Four Design Patterns in Java*. New York, NY, USA: Apress Media, LLC, 2016.

[31] World Wide Web Consortium, "SPARQL 1.1 Update," https://-www.w3.org/TR/sparql11-update, 2013.

[32] Renhao Xiong, "Online Appendixes and Data Package for: Distinguishing Similar Design Patterns Instances through Temporal Behavior Analysis," https://github.com/Megre/Dataset4SparT-ETA, accessed: 2019-10-22.

[33] Oracle Co., Ltd., "JVM Tool Interface," https://docs.oracle.com/javase/-9/docs/specs/jvmti.html, accessed: 2019-02-12.

[34] ——, "JNI Types and Data Structures," https://docs.oracle.com/javase/8/-docs/technotes/guides/jni/spec/types.html, accessed: 2019-02-10.

[35] ——, "Java Virtual Machine Specification - Loading, Linking, and Initializing," https://docs.oracle.com/javase/specs/jvms/se8/html/jvms-5.html, accessed: 2019-02-07.

[36] J. F. Allen, "Maintaining knowledge about temporal intervals," *Readings in Qualitative Reasoning About Physical Systems*, vol. 26, no. 11, pp. 361–372, 1990.

[37] A. Alessandro and F. Enrico, "Chapter 12 – temporal description logics," *Foundations of Artificial Intelligence*, vol. 1, pp. 375–388, 2005.

[38] V. R. Basili, G. Caldiera, and H. D. Rombach, "Goal question metric paradigm," *Encyclopedia of Software Engineering*, 1994.

[39] H. Lee, H. Youn, and E. Lee, "Automatic detection of design pattern for reverse engineering." Los Alamitos: IEEE Computer Society, 2007, pp. 577–583.

[40] F. Arcelli, F. Perin, C. Raibulet, and S. Ravani, "Design pattern detection in java systems: A dynamic analysis based approach," in *Communications in Computer and Information Science*. Berlin: Springer-Verlag Berlin, 2010, vol. 69, pp. 163–179.

[41] D. Heuzeroth, T. Holl, G. Hogstrom, and W. Lowe, "Automatic design pattern detection," in *International Workshop on Program Comprehension*. Los Alamitos: IEEE Computer Society, 2003, pp. 94–103.

[42] D. Heuzeroth, S. Mandel, and W. Lowe, "Generating design pattern detectors from pattern specifications," pp. 245–248, 2003, 18th IEEE International Conference on Automated Software Engineering, Montreal, Canada, Oct 06-10, 2003.

[43] C. Sammut and G. I. Webb, *Encyclopedia of machine learning*. Springer Science & Business Media, 2011.

[44] D. Chicco, “Ten quick tips for machine learning in computational biology,” *Biodata Mining*, vol. 10, no. 1, p. 35, 2017.

[45] Apache Software Foundation, “Jena Ontology API,” http://-jena.apache.org/documentation/ontology, accessed: 2018-06-10.

[46] Oracle, “Eclipse Java development tools (JDT),” http://www.eclipse.org/-jdt, accessed: 2018-08-01.

[47] Renhao Xiong, “The Online Version of SparT,” http://www.spart.group, accessed: 2018-07-10.

[48] D. Yu, Y. Zhang, and Z. Chen, “A comprehensive approach to the recovery of design pattern instances based on sub-patterns and method signatures,” *Journal of Systems and Software*, vol. 103, pp. 1–16, 2015.

[49] M. Esmaeilpour, V. Naderifar, and Z. Shukur, “Design pattern mining using distributed learning automata and DNA sequence alignment,” *PLOS ONE*, vol. 9, no. e1063139, 2014.

[50] M. Oruc, F. Akal, and H. Sever, “Detecting design patterns in object-oriented design models by using a graph mining approach.” New York: IEEE, 2016, pp. 115–121.

[51] S. Z. Yang, A. Manzer, and V. Tzerpos, “Measuring the quality of design pattern detection results.” New York: IEEE, 2015, pp. 53–62.

[52] A. Ampatzoglou, S. Charalampidou, and I. Stamelos, “Research state of the art on gof design patterns: A mapping study,” *Journal of Systems and Software*, vol. 86, no. 7, pp. 1945–1964, 2013.

[53] G. Rasool, P. Maeder, and I. Philippow, “Evaluation of design pattern recovery tools,” in *Procedia Computer Science*. Amsterdam: Elsevier Science BV, 2011, vol. 3, pp. 813–819.

[54] J. Dong, Y. Zhao, and Y. Sun, “A Matrix-Based Approach to Recovering Design Patterns,” *IEEE Transactions on Systems Man and Cybernetics Part A-Systems and Humans*, vol. 39, no. 6, pp. 1271–1282, 2009.

[55] Y. Wang, H. Guo, H. Liu, and A. Abraham, “A fuzzy matching approach for design pattern mining,” *Journal of Intelligent & Fuzzy Systems*, vol. 23, no. 2-3, pp. 53–60, 2012.

[56] H. Thaller, L. Linsbauer, and A. Egyed, “Feature maps: A comprehensible software representation for design pattern detection,” 2019, pp. 207–217, 26th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER).

[57] S. Alhusain, S. Coupland, R. John, and M. Kavanagh, “Towards machine learning based design pattern recognition.” New York: IEEE, 2013, pp. 244–251.

[58] S. Hussain, J. Keung, A. A. Khan, A. Ahmad, S. Cuomo, F. Piccialli, G. Jeon, and A. Akhunzada, “Implications of deep learning for the automation of design patterns organization,” *Journal of Parallel and Distributed Computing*, vol. 117, pp. 256–266, 2018.

[59] J. Dong, D. S. Lad, and Y. Zhao, “Dp-miner: Design pattern discovery using matrix.” Los Alamitos: IEEE Computer Society, 2007, pp. 371–380.

[60] B. Di Martino and A. Esposito, “A rule-based procedure for automatic recognition of design patterns in UML diagrams,” *Software-Practice & Experience*, vol. 46, no. 7, pp. 983–1007, 2016.

[61] A. Pande, M. Gupta, and A. K. Tripathi, “A new approach for detecting design patterns by graph decomposition and graph isomorphism,” in *Communications in Computer and Information Science*. Berlin: Springer-Verlag Berlin, 2010, vol. 95, pp. 108–119.

[62] F. Arcelli and L. Cristina, “Enhancing software evolution through design pattern detection.” Los Alamitos: IEEE Computer Society, 2007, pp. 7–14, 3rd International IEEE Workshop on Software Evolvability.

[63] S. Uchiyama, H. Washizaki, Y. Fukazawa, and A. Kubo, “Design pattern detection using software metrics and machine learning,” vol. 708. Oldenburg, Germany: CEUR-WS, 2011, pp. 38–47.

[64] N. Bouassida and H. Ben-Abdallah, “A new approach for pattern problem detection,” in *Lecture Notes in Computer Science*. Berlin: Springer-Verlag Berlin, 2010, vol. 6051, pp. 150–164.

[65] Oracle Co., Ltd., “Eclipse Test and Performance Tools Platform,” https://-wiki.eclipse.org/TPTP, accessed: 2019-08-11.

[66] ——, “Java Platform Debugger Architecture,” https://docs.oracle.com/-javase/1.5.0/docs/guide/jpda/architecture.html, accessed: 2019-08-11.

[67] H. Lee, H. Youn, and E. Lee, “A design pattern detection technique that aids reverse engineering,” *International Journal of Security and its Applications*, vol. 2, no. 1, pp. 1–12, 2008.

[68] Oracle Co., Ltd., “The JVM Tool Interface (JVM TI): How VM Agents Work,” https://www.oracle.com/technetwork/articles/javase/index-140680.html, accessed: 2019-08-11.

[69] heuzeroth, mtrifu, and tgutzmann, “The Recoder Framework,” https://-sourceforge.net/projects/recoder/, accessed: 2019-08-11.