A Matrix-Based Approach to Recovering Design Patterns

Jing Dong, *Senior Member, IEEE*, Yajing Zhao, and Yongtao Sun

***Abstract*—Design patterns describe good solutions to common and recurring problems in software design. They have been widely applied in many software systems in industry. However, pattern-related information is typically not available in large system implementations. Recovering these design pattern instances in software systems can help not only to understand the original design decisions and tradeoffs but also to change the systems with quality assurance. This paper presents our approach on recovering design patterns based on matrices and weights. We formally specify our methods to encode both the systems and the design patterns into matrices and weight. Our formal specification rigorously defines the structural, behavioral, and semantic analyses of our approach. A set of experiments on real-world systems is also carried out to evaluate our approach with analysis results.**

***Index Terms*—Design pattern, Extensible Markup Language (XML), reverse engineering, Unified Modeling Language (UML).**

I. Introduction

DESIGN patterns [16] have emerged to be an important design guidance that provides good generic solutions to recurring problems. Each design pattern documents a guideline in the software development process and leaves room for application variations. Patterns were initially introduced by Alexander *et al.* in the context of architecture [1]. Inspired by the patterns of Alexander *et al.*, design patterns are suitable for reusing design experience at a high abstraction level. Software developers adopt patterns in their design to improve adaptability and extensibility. Patterns allow the designers to tackle the same problem by reusing expert solutions. They can also communicate with each other at a very high level of abstraction.

As a common design practice in forward engineering, design patterns have been widely applied in many software applications, such as knowledge-based systems [8], agent-based systems [4], and robotics [17]. When a design pattern is applied in a software system design, however, the names of its participants are normally changed to reflect application domain information. Therefore, the role that each participant plays is typically lost in the implementation. It is difficult to identify the patterns applied in a large system design. Consequently, the benefits of design patterns are compromised because the designers cannot communicate in terms of the design patterns they used. It is also hard to understand the systems since the original design decisions and tradeoffs embedded in the design patterns are not available. Due to lack of documentation, high-level architecture and design information is often missing in system implementations, particularly for legacy systems. Even if such documentation is available, the systems may be changed and migrated to satisfy new requirements or technologies. Nevertheless, the original architecture and design documents may not be changed accordingly such that they are not consistent with their implementations. Recovering the instances of design patterns applied in large systems can help human to understand the systems at the architecture and design level. It also assists in reengineering the systems with improved designs. The Unified Modeling Language (UML) has been used to model many different kinds of applications, such as agent-based systems [20], computer-integrated manufacturing development [25], and distributed virtual environments [27]. While most current software designs are modeled in UML, recovering design pattern instances from UML diagrams can also greatly help the understanding of the design and the communications among the designers.

Recent advances in reverse engineering have provided techniques and tools, such as IBM Rational Rose [48], which can recover the UML designs from source code in object-oriented programming languages. However, there is lack of work on recovering design pattern instances from these UML designs, which can help to trace back to the original design decisions and tradeoffs. While UML, the *de facto* standard for modeling software design, is widely applied in industry, recovering design pattern instances from UML diagram has immediate practical value in software and system engineering.

Several approaches [15] have been proposed to recover the design patterns applied in a software system, which may be displayed in UML design diagrams [9], [30], [31]. These approaches normally do not try to discover the design pattern instances from scratch. They typically use some existing reverse engineering tools to reach some intermediate representations of the source code, such as the design structures and the system behaviors in terms of workflows. Some existing approaches [2], [21], [28] use the Abstract Syntax Tree (AST) as the intermediate representation. AST is a tree representation of the abstract syntactic structure of source code. The tree nodes denote the language constructs in the source code. The syntax tree omits much details of the real syntax. For instance, an *IF condition THEN expression ELSE expression* syntactic construct may be denoted by a single node with two branches, without denoting

Manuscript received November 20, 2007; revised November 10, 2008. First published September 22, 2009; current version published October 16, 2009. This paper was recommended by Associate Editor H. Pham.

J. Dong and Y. Zhao are with the Department of Computer Science, The University of Texas at Dallas, Richardson, TX 75083 USA (e-mail: jdong@ieee.org; yxz045100@utdallas.edu).

Y. Sun is with the American Airlines, Fort Worth, TX 76155 USA, and also with the Department of Computer Science, The University of Texas at Dallas, Richardson, TX 75083 USA (e-mail: Yongtao.Sun@aa.com).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TSMCA.2009.2028012

the *IF*, *THEN*, and *ELSE* keywords. The generation of AST from source code can be assisted by existing tools. Different pattern recovery approaches use different methods to search the AST and match the patterns. However, the main problem with these AST-based approaches is that the recovery results are typically presented in an *ad hoc* manner such that they are difficult to understand. Significant efforts have to be made to map the results to the UML design diagrams which are normally used for visualization. Furthermore, some of the approaches only provide the number of pattern instances recovered without information about their particular locations in the design. It is even harder to map the results to the UML diagrams in this case. An intermediate representation based on some standardized languages, such as the Extensible Markup Language (XML), may greatly help interchanges of information. The XML has been applied to model corporate memory [23] and develop online recruitment services [26], intelligent shop floor [34], and Web business intelligence applications [18].

In this paper, we present a matrix-based approach to recovering design pattern instances from system designs. We define the structure, behavior, and semantics of design patterns in the software systems. Our approach uses the prime numbers to encode both system and pattern characteristics into matrices and weights. We present a formal specification to rigorously describe our approach which uses the XML Metadata Interchange (XMI) [50] as intermediate representation language. To evaluate our approach, we also conduct a set of experiments on real-world software systems, including the Java Abstract Window Toolkit (AWT) [42], JUnit [45], JEdit [43], JHotDraw [44], Chemistry Development Kit (CDK) [39], and Personal Finance Manager (GFP) [47].

The rest of this paper is organized as follows. We present an overview of our approach in the next section. Section III formally defines our design pattern recovery approach. Section IV describes our experiments on some large open-source systems. The last two sections are related work and conclusions.

II. Approach Overview

With the increasing complexity and size of the software systems, understanding and changing of these systems become difficult tasks, particularly when the architecture and design documentations are incomplete, missing over time, and inconsistent with the source code. Recovering the original design decisions and tradeoffs may help developers to understand large systems and make change more easily. Design patterns generally document the design decisions and tradeoffs as well as possible ways for future evolutions. Thus, recovering the design patterns applied in a software system can assist to cope with the complexity of large systems. Such recovery processes are typically not done from scratch but take advantages of some existing reverse engineering tools to extract the important information from source code into some intermediate representations, such as UML diagrams. When a design pattern is applied in a design, on the other hand, the role information about its participants is generally lost. Recovering such information from the UML diagrams can help the designers to understand the design and communicate with other designers.

While the UML diagrams are normally stored in some proprietary format, it is hard to directly manipulate them. To solve this problem, we use the XMI standard [50] to serialize UML into XML file and use it as the intermediate representation. XMI is an XML-based standard proposed by the Object Management Group that maps UML to XML. We use XMI as the intermediate representation based on the following reasons. First, XMI is an interchange format for metadata in terms of the Meta Object Facility [46]. While UML models are generally persisted in some proprietary format of certain tool platforms, XMI specifies how UML models [6] are mapped into a platform-independent XML file. By representing a UML model in XML, the UML model can be searched for patterns. Second, several tools are currently available to recover the UML models of a software system from its source code. There exist plug-ins for these tools that can help to obtain the XMI representations from the UML models. Hence, the software systems and design patterns modeled in UML diagrams can be automatically transformed into XML file in the XMI format by these plug-ins. Third, following the XMI standard allows our pattern detection techniques to be naturally integrated with other techniques and tools following the XMI standard.

Fig. 1 shows an overall architecture of our approach. Our design pattern recovery processes include structural, behavioral, and semantic analyses. The structural analysis parses the system design models in XMI to build a square matrix whose number of rows and number of columns both equal to the number of classes in the source system. Each row represents a class, so does each column. Each cell of the matrix encodes the relationships between the class on the row and the class on the column. In particular, we use prime numbers to encode the relationships. Each class-to-class relationship is assigned a unique prime number. The value of each cell of the matrix is the product of the prime numbers that represent the corresponding relationships between the class on the row and the class on the column. Because the product of prime numbers represents a unique combination of the prime numbers, all relationships between two classes can be easily decoded from their corresponding matrix cell value. Aside from encoding the class-to-class relationships into the matrix, our approach encodes the information of each individual class into class weight in a similar way. The method or attribute of a class is assigned a unique prime number. The weight of each class is encoded as the product of the prime numbers to the power of the numbers of the methods, attributes, and relationships the class has.

Our approach defines the design patterns using an XML file, which include their structural, behavioral, and semantic characteristics. These pattern characteristics are used in different phases. During structural analysis phase, our tool extracts the structural information of the pattern and encodes it into a matrix and weights in a similar way as we encode the system. Thus, the structural analysis can be reduced to the matching of the design pattern matrix with the system matrix as well as the weights of the design pattern classes with the weights of the system classes. We call it a match as long as there exists a submatrix of the system matrix such that all cells of the submatrix are the integral multiples of the corresponding cells in the design pattern matrix and that the weights of the classes

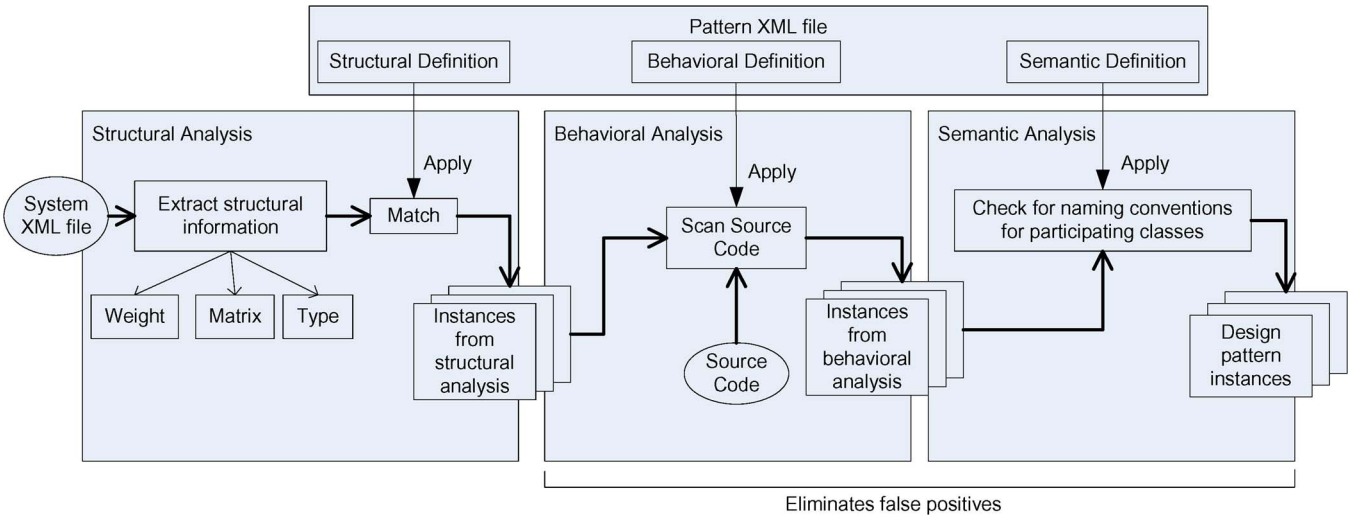


Fig. 1. Overall architecture of the approach.

in the submatrix are the integral multiples of the weights of the corresponding design pattern classes.

We relax the criteria of structural analysis to reduce the false negative cases. As a consequence, however, the number of false positive cases may increase. The false positive cases in the structural analysis results can be eliminated in the later analysis processes, i.e., the behavior and semantic analyses.

Our behavioral analysis checks whether a desired method invocation exists in a class with the right signatures and polymorphic definitions. Different design patterns may require different behavioral analyses which can be determined by the pattern behavioral characteristics described in the XML file of pattern definition.

Some design patterns, such as Bridge and Strategy, are similar in their structures and behaviors. They may only differ from their intents and motivations. Although source code generally retains no such semantic information from system design, the naming convention of classes may actually provide some trace of the original design intents and motivations. For example, several class names in the Java.awt contain "Strategy," which is a good indication of the original intents. Our semantic analysis checks the naming conventions when the distinctions are needed. Although naming conventions are not always observed by developers, they actually help to distinguish patterns in many cases as shown in our experiments. The semantic characteristics that need to check for each design pattern are also provided by the pattern definition file in XML.

The behavioral and semantic analyses may require checking the source code directly, in addition to the intermediate representations. However, such checks are based on the results from structural analysis so that only particular classes and methods, instead of the entire source code, are checked.

III. Formal Specifications of Our Approach

In the previous section, we introduce the main ideas of our approach. To be more precise, clear, and unambiguous, we formally specify our pattern recovery approach in this section. We use four patterns, namely, the Adapter, Bridge, Strategy, and Composite patterns, as examples to illustrate our approach.

A. Structural Analysis

As discussed in the previous section, the structural analysis concentrates on the classes in a software system and their attributes, operations, and relationships with other classes. More specifically, we define set ELM for attributes and operations as well as set REL for relations, such as association, generalization, dependence, aggregation, and realization as follows:

$$ELM = \{attr, oper\}$$

$$REL = \{assoc, gener, depd, aggr, realz\}.$$

Class represents a set of classes. PN is a set of prime numbers. For example

$$PN = \{2, 3, 5, 7, 11, 13\}.$$

Definition 3.1 (Class-to-Class Relation): The relation between two classes is a function

$$r : Class \times Class \rightarrow 2^{REL}.$$

Example 3.1 (Class-to-Class Relation):

$$r(A, B) = \{assoc, gener, depd\}$$

$$r(A, B) = \{aggr\}$$

$$r(A, B) = \{realz\}$$

represent that classes A and B have the association, generalization, and dependence relationships, aggregation relationship, or realization relationship, respectively.

Definition 3.2 (Encoding Function): The elements of sets ELM and REL are assigned with a unique prime number by a function

$$\rho : ELM \cup REL \rightarrow PN.$$

Example 3.2 (Encoding Function): We use the following encodings in this paper:

$$\begin{aligned}\rho(attr) &= 2\\ \rho(oper) &= 3\\ \rho(assoc) &= 5\\ \rho(gener) &= 7\\ \rho(depd) &= 11\\ \rho(aggr) &= 13.\end{aligned}$$

Definition 3.3 (Cell Value Function): All relationships between two classes are mapped into an integer by a function

$$\gamma : Class \times Class \rightarrow N$$

such that $\forall A, B \in Class$

$$\gamma(A,B) = \prod_i \rho(R_i) \qquad \forall R_i \in r(A,B) \text{ if } r(A,B) \neq \emptyset$$

$$\gamma(A,B) = 1, \qquad \text{if } r(A,B) = \emptyset.$$

Definition 3.4 (System Matrix): The relationships between the classes of a system are defined as a square matrix

$$A_m = (a_{ij})_m$$

where $a_{ij} = \gamma(C_i, C_j)$, C_i, $C_j \in Class$, $1 \leq i, j \leq m$, and $|Class| = m$.

Definition 3.5 (Weight): The weight of each class is defined as a function

$$\omega : Class \rightarrow N$$

such that $\forall A \in Class$

$$\omega(A) = \rho^m(attr) \times \rho^n(oper) \times \prod_i \gamma(A, C_i)$$

where $1 \leq i \leq l, C_i \in Class$, m is the number of attributes that class A contains, n is the number of operations that class A contains, and $l = |Class|$.

Definition 3.6 (System Weight Vector): The weights of all classes in a system are defined by the following vector:

$$\text{Vector } B_m = (b_i)_m$$

where $b_i = \omega(C_i), C_i \in Class, 1 \leq i \leq m$, and $|Class| = m$.

Consider a set of design patterns that need to be discovered from a software system

$$PATTERN = \{adapter, bridge, strategy, composite, \ldots\}.$$

Definition 3.7 (Pattern Class): All classes that participate in a design pattern are defined as its pattern classes, and

$$\forall p \in PATTERN, Class(p)$$

represents the set of classes participating pattern p.

Definition 3.8 (Pattern Matrix): The relationships between the classes of a design pattern are defined as a square matrix

$$D_M(p) = A_m = (a_{ij})_m, \qquad p \in PATTERN$$

where $a_{ij} = \gamma(C_i, C_j), C_i, C_j \in Class(p), 1 \leq i, j \leq m$, and $|Class(p)| = m$.

Definition 3.9 (Pattern Weight Vector): The weights of all classes in a design pattern are defined by the following vector:

$$\text{Vector } D_W(p) = B_m = (b_i)_m, \qquad p \in PATTERN$$

where $b_i = \omega(C_i)$, $C_i \in Class(p)$, $1 \leq i \leq m$, and $|Class(p)| = m$.

Definition 3.10 (Matrix Match): Consider a system matrix $A_m = (a_{ij})_m$, $|Class| = m$, and a design pattern $p \in PATTERN$, $|Class(p)| = n$ with its matrix $D_M(p) = (d_{ij})_n$. If there exists a submatrix of A_m

$$sub\ A_n = (s_{ij})_n = A[k_1, k_2, \ldots, k_n; k_1, k_2, \ldots, k_n], \qquad (1 \leq k_1, k_2, \ldots, k_n \leq m)$$

such that

$$s_{ij} \bmod d_{ij} = 0, \qquad 1 \leq i,\ j \leq n$$

then the pattern matrix matches the system matrix.

Definition 3.11 (Weight Match): Consider a system weight vector $B_m = (b_i)_m$, $|Class| = m$, and a design pattern $p \in PATTERN, |Class(p)| = n$ with its weight vector $D_W(p) = (d_i)_n$. If there exists a subvector of B_m

$$sub\ B_n = (s_i)_n = B[k_1, k_2, \ldots, k_n], \qquad (1 \leq k_1, k_2, \ldots, k_n \leq m)$$

such that

$$s_i \bmod d_i = 0, \qquad 1 \leq i \leq n$$

then the pattern weight vector matches the system weight vector.

Definition 3.12 (Pattern Structure Match): When both the matrix and weight of a design pattern match those of a system, it is defined as structure match. This definition can be derived directly from the previous two definitions.

Informally speaking, the previous definitions use matrices and weights to represent the structural information of systems and patterns and define the matching of a pattern structure with a system structure. More specifically, the system matrix (Definition 3.4) or pattern matrix (Definition 3.8) describes the relationships, such as generalization and association, between the classes in a system or a pattern, respectively. Pattern matrix and weights serve as the criteria of structural analysis. When there is a matrix match (Definition 3.10) between a system matrix and a pattern matrix, it shows that the system includes some classes having the same relationships as those in the pattern. If the weights of these classes in the system also match those of the classes in the pattern, which is called a weight match (Definition 3.11), it shows that these classes have the required numbers of attributes and operations by the corresponding pattern. Therefore, these classes, whose matrix and weights match those of the pattern, can be considered as a structure match with the pattern (Definition 3.12) and, thus, a candidate instance of the pattern.

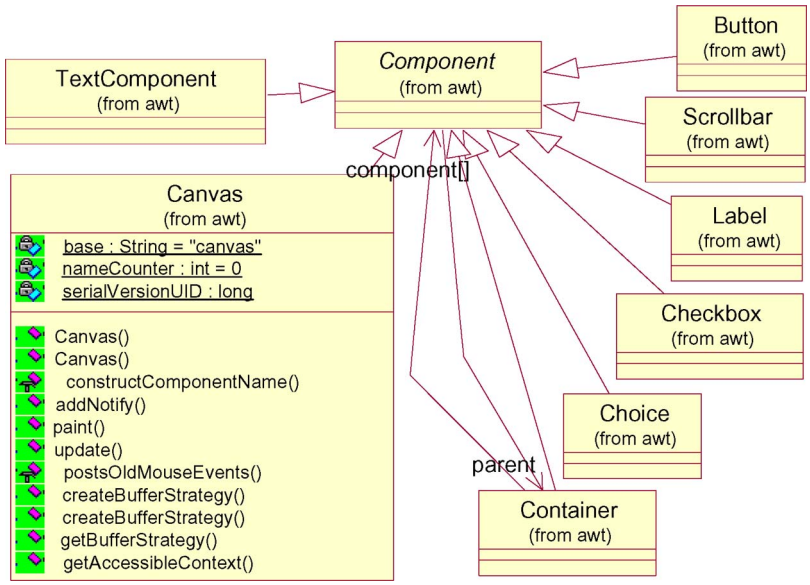


Fig. 2. Class diagram—from Java AWT.

TABLE I
MATRIX FOR PARTIAL DESIGN OF JAVA.AWT

	Button	Canvas	Checkbox	Choice	Component	Container	Label	Scrollbar	TextComponent
Button	1	1	1	1	7	1	1	1	1
Canvas	1	1	1	1	7	1	1	1	1
Checkbox	1	1	1	1	7	1	1	1	1
Choice	1	1	1	1	7	1	1	1	1
Component	1	1	1	1	1	5	1	1	1
Container	1	1	1	1	35	1	1	1	1
Label	1	1	1	1	7	1	1	1	1
Scrollbar	1	1	1	1	7	1	1	1	1
TextComponent	1	1	1	1	7	1	1	1	1

Consider an example of a partial design from the Java.awt package about the graphic user interface design as shown in Fig. 2. This class diagram of the partial design can be reverse engineered from the source code of Java.awt by existing reverse engineering tools such as IBM Rational Rose. Since most of these tools persist design diagrams in some proprietary format, we use existing plug-ins of the tools, such as Unisys, to generate the XMI representations of the design diagrams. In this way, we can parse the XMI files of the design diagrams and generate the matrix and weights of the system design based on our approach formally defined previously.

The set *Class* includes all classes of the design shown in Fig. 2. Table I shows the matrix of the design generated based on Definition 3.4. This matrix encodes the relationships between each pair of classes by a number. The object-oriented relationships between classes are encoded by a prime number based on Definition 3.2 and Example 3.2. When two classes have multiple relationships, the value of the corresponding cell of the matrix will be the product of all the corresponding prime numbers encoding the respective relationships. Because every cell value is an integral multiple of prime numbers which represents a unique combination of prime numbers, the relationships between each pair of the classes can be easily recovered from their corresponding cell value of the matrix. The value of each cell in the system matrix can be used to decode the relationships between the corresponding pair of classes. For example, there is a generalization relationship from the Button class to the Component class. Thus, the value of the cell on the first row and fifth column of the matrix is seven, as shown in Table I. For asymmetric relationships, we treat incoming relationships and outgoing relationships differently in that the two ends of the relationships are different. Similarly, there are the generalization and association relationships from the Container class to the Component class, which cause the value of the corresponding cell to be $35 = 5 \times 7$. There is an association relationship from the Component class to the Container class, making the corresponding cell value to five.

With such an encoding method, it is quite easy to decode the relationships between any pair of classes in the matrix. If the value of a cell is 35, which is the product of 5 and 7, for example, then it is easy to decode that the corresponding pair of classes has both the association and generalization relationships.

As defined in Definition 3.5, the weight of a class represents the structural characteristics of the class, including the numbers of its attributes, methods, and outgoing relationships which are encoded based on Definition 3.2 and Example 3.2. For example, the weight of the Canvas class in Fig. 2 is $2^3 \times 3^5 \times 7 = 13\,608$. Note that an optimization of our approach is to only consider a maximum of five for the number of attributes,

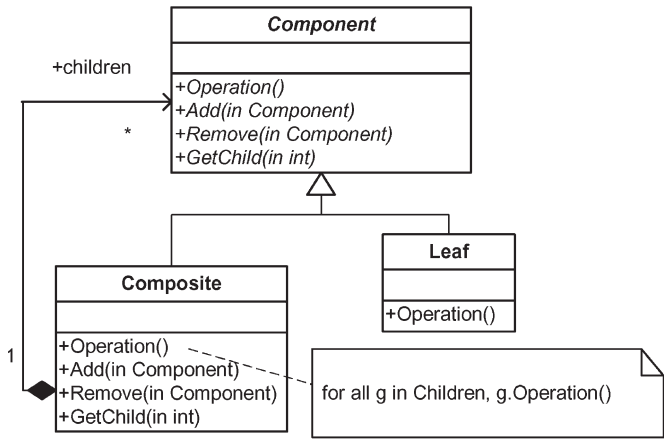


Fig. 3. Composite pattern.

TABLE II
MATRIX FOR THE COMPOSITE PATTERN

	Component	Composite	Leaf
Component	1	1	1
Composite	7	1	1
Leaf	35	1	1

methods, or relationships of a class since the classes of each design pattern generally only consist of a small number of attributes, methods, and relationships. It is not quite efficient to count the numbers of all attributes, methods, and relationships in a class. Although the Canvas class includes 11 methods, our approach only counts 5 for its weight. The weights of other classes in Fig. 2 can be computed similarly.

Let us consider the Composite pattern whose class diagram is shown in Fig. 3. It contains three main role classes, namely, Component, Composite, and Leaf, where the Leaf and Composite classes inherit from the Component class. The matrix of the Composite pattern, as shown in Table II, can be calculated based on Definition 3.8. The value of each cell encodes the relationships between the corresponding pair of classes. Similarly, the weights of the Component, Composite, and Leaf classes can be computed based on Definition 3.9, which are 81, 2835, and 21, respectively.

According to Definition 3.10 and Definition 3.11, our structural analysis (Definition 3.12) includes the matching from the matrix and weights of the pattern, e.g., the Composite pattern, to the matrix and weights of the system, e.g., the partial design of Java.awt shown in Fig. 2. For example, the submatrix, including the Component, Container, and Button classes, can be a match of the matrix of the Composite pattern according to Definition 3.10. Their corresponding weights also match those of the Composite pattern. Thus, these three classes form a candidate instance of the Composite pattern according to Definition 3.12.

B. Behavioral Analysis

The result of our structural analysis may contain false positive cases because it concentrates on the structural aspect of design patterns. Most design patterns have behavioral characteristics in addition to the structural ones. These behavioral characteristics may include control flows, e.g., some methods need to invoke some other methods. Our behavioral analysis looks for particular method invocations from the system source code. As introduced in Section II, patterns are defined in an XML file which will be consulted in each analysis phase. The behavioral analysis is conducted after the structural analysis by matching the behavior characteristics defined in the pattern definition XML file. Instead of searching the entire source code, the behavioral analysis considers only the classes in the candidate pattern instances obtained from the structural analysis. Thus, the search space can be reduced. In the rest of this section, we define the behavioral characteristics formally by using predicates. We first introduce the syntax and meaning of the predicates used in our specifications. We then formally present the behavioral characteristics of the Adapter, Bridge, Strategy, and Composite patterns, where the role names are based on the study in [16]. The behavioral properties of other patterns can be represented similarly as discussed in [14]. In addition to the formal specification, we use the Adapter pattern as an example to illustrate how the behavior analysis, which is an implementation corresponding to the formal specifications of behavioral characteristics, is performed. For simplicity, we omit the implementation details of other patterns.

Definition 3.13 (Syntax and Meaning of Predicates):

1) **methodList**$(\boldsymbol{A})$ where $A \in Class$: the set of methods in Class A.
2) **name**$(\boldsymbol{method_i})$: the name of method $method_i$.
3) **parameter**$(\boldsymbol{method_i})$: the list of parameter types of method $method_i$.
4) **returnType**$(\boldsymbol{method_i})$: the return type of method $method_i$.
5) **call**$(\boldsymbol{method_i}, \boldsymbol{method_j})$: $method_i$ invokes $method_j$.
6) **containSubstring** $(\boldsymbol{string_i}, \boldsymbol{string_j})$: $string_i$ contains $string_j$ as its substring.
7) **caseIgnore**$(\boldsymbol{string_i})$: Change all the characters in $string_i$ into lower case.

1) Adapter Pattern: In the Adapter pattern, there shall be a common method, called Request, defined in the Target and Adapter classes. The Request method in the Adapter class shall call a method, called SpecificRequest, defined in the Adaptee class. These behavioral characteristics of the Adapter pattern are formally defined as follows.

Definition 3.14 (Adapter Behavior):

$$\exists\, method_i\, \exists\, method_j\, \exists\, method_k$$
$$(method_i \in \text{methodList}(Target) \wedge \quad (1)$$
$$method_j \in \text{methodList}(Adapter) \wedge \quad (2)$$
$$method_k \in \text{metholdList}(Adaptee) \wedge \quad (3)$$
$$\text{name}(method_i) = \text{name}(method_j) \wedge \quad (4)$$
$$\text{parameter}(method_i) = \text{parameter}(method_j) \wedge \quad (5)$$
$$\text{returnType}(method_i) = \text{returnType}(method_j) \wedge \quad (6)$$
$$\text{call}(method_j, method_k)). \quad (7)$$

Our behavioral analysis is performed according to the formal specification of pattern behavioral characteristics. According to predicates (1), (2), and (3), it gets a list of methods from the class which plays the role of Target, a list of methods from the

class which plays the role of Adapter, and a list of methods from the class which plays the role of Adaptee. It then filters the first two lists and retains the methods which appear in both lists. This can be determined by checking the name [predicate (4)], the parameter [predicate (5)], and the return type [predicate (6)] of the methods. Finally, according to predicate (7), our behavioral analysis checks whether there is a method in the retained list that calls a method in the third list. If there is such a method, this pattern candidate passes the behavioral analysis. Otherwise, it is a false positive instance of the Adapter pattern.

2) Bridge Pattern: There shall be at least one common method, called OperationImp, between the Implementor and ConcreteImplementor in the Bridge pattern. Absence of such method indicates that there is no variation in the implementation. This does not conform to the definition of the Bridge pattern. The behavior of the pattern also requires that the Abstraction class shall invoke at least one of the OperationImp methods.

Definition 3.15 (Bridge Behavior):

$\exists\, method_i\, \exists\, method_j\, \exists\, method_k$
$(method_i \in \text{methodList}(Abstraction) \wedge$
$method_j \in \text{methodList}(Implementor) \wedge$
$method_k \in \text{metholdList}(ConcreteImplementor) \wedge$
$\text{name}(method_j) = \text{name}(method_k) \wedge$
$\text{parameter}(method_j) = \text{parameter}(method_k) \wedge$
$\text{returnType}(method_j) = \text{returnType}(method_k) \wedge$
$\text{call}(method_i, method_j))$.

3) Strategy Pattern: There shall be at least one common method, called AlgorithmInterface, which is defined in the Strategy and ConcreteStrategy classes. The behavior of Strategy also requires that the Context class shall invoke at least one of the AlgorithmInterface methods.

Definition 3.16 (Strategy Behavior):

$\exists\, method_i\, \exists\, method_j\, \exists\, method_k$
$(method_i \in \text{methodList}(Context) \wedge$
$method_j \in \text{methodList}(Strategy) \wedge$
$method_k \in \text{metholdList}(ConcreteStrategy) \wedge$
$\text{name}(method_j) = \text{name}(method_k) \wedge$
$\text{parameter}(method_j) = \text{parameter}(method_k) \wedge$
$\text{returnType}(method_j) = \text{returnType}(method_k) \wedge$
$\text{call}(method_i, method_j))$.

4) Composite Pattern: The Composite pattern requires that at least one common method, called Operation, shall exist among Component, Composite, and Leaf classes. The Operation method in the Composite and Leaf classes shall override that of their parent. The Operation method in the Composite class shall invoke the Operation method in the Component class. In the Composite class, there shall be a method playing the role of the Add or Delete method that has a parameter with the Component as its type.

Definition 3.17 (Composite Behavior):

$\exists\, method_i\, \exists\, method_j\, \exists\, method_k$
$(method_i \in \text{methodList}(Component) \wedge$
$method_j \in \text{methodList}(Composite) \wedge$
$method_k \in \text{metholdList}(Leaf) \wedge$
$\text{name}(method_i) = \text{name}(method_j) \wedge$
$\text{parameter}(method_i) = \text{parameter}(method_j) \wedge$
$\text{returnType}(method_i) = \text{returnType}(method_j) \wedge$
$\text{name}(method_j) = \text{name}(method_k) \wedge$
$\text{parameter}(method_j) = \text{parameter}(method_k) \wedge$
$\text{returnType}(method_j) = \text{returnType}(method_k) \wedge$
$\text{call}(method_j, method_i)) \wedge$
$\exists method_l (method_l \in \text{methodList}(Composite) \wedge$
$\text{parameter}(method_l) = Component)$.

C. Semantic Analysis

Some design patterns are similar to each other in their structural and behavioral aspects, for example, the Bridge and Strategy patterns. Structural and behavioral analyses are not enough to distinguish these kinds of patterns since they differ only in their intents and motivations. Such semantic difference is generally hard to check. Nevertheless, we found that most developers follow some naming conventions when they give the names to classes, operations, and attributes. Such naming conventions sometimes may leave some trace of their original design intents. For example, our structural and behavioral analyses of the Java.awt package return the same candidate instance set for the Bridge and Strategy patterns. Among them, some classes are named BufferStrategy, FlipBufferStrategy, SimpleBufferStrategy, and BltBufferStrategy. This can be a good indication that these are actually candidates of the Strategy, rather than Bridge pattern. As introduced in Section II, we specify the pattern characteristics in an XML file, which include the semantic characteristics of a pattern. We formally define such semantic analysis of the naming convention for the Strategy pattern as follows. The Bridge semantics can be defined similarly.

Definition 3.18 (Strategy Semantics):

$\text{containSubstring}(\text{caseIgnore}(\text{name}(Context)),$
$\text{caseIgnore}(\text{“strategy"})) \wedge$
$\text{containSubstring}(\text{caseIgnore}(\text{name}(Strategy)),$
$\text{caseIgnore}(\text{“strategy"})) \wedge$
$\text{containSubstring}(\text{caseIgnore}(\text{name}(ConcreteStrategy)),$
$\text{caseIgnore}(\text{“strategy"})) \wedge$
$(\exists\, method_i$
$(method_i \in (\text{metholdList}(Context) \cup$
$\text{methodList}(Strategy) \cup$
$\text{methodList}(ConcreteStrategy)) \wedge$
$\text{containSubstring}(\text{caseIgnore}(\text{name}(method_i)),$
$\text{caseIgnore}(\text{“strategy"}))))$.

IV. Experiments

To evaluate our approach, we developed a tool, called DP-Miner [11], which implements the recovery processes as defined formally in the previous sections. Our tool takes, as input, two XML files and a directory path. One XML file includes the system information, and the other includes the pattern definitions. The directory path is the locator of the system

TABLE III
PATTERN RECOVERY EXPERIMENT RESULTS

System	Version	Class#	File#	Pattern	Analysis		
					Structure	Behavior	Semantic
JavaAWT	JDK1.4.2	570	345	Adapter	57	21	21
				Bridge	100	76	65
				Strategy	100	76	76
				Composite	92	3	3
JUnit	3.8.2	126	93	Adapter	15	3	3
				Bridge	6	6	6
				Strategy	6	6	6
				Composite	9	3	3
JEdit	4.2	1001	394	Adapter	80	17	17
				Bridge	33	24	24
				Strategy	33	24	24
				Composite	0	0	0
JHotDraw	6.0 beta 1	530	484	Adapter	27	4	4
				Bridge	74	64	58
				Strategy	74	64	64
				Composite	0	0	0
CDK	1.0.1	1173	1031	Adapter	32	32	32
				Bridge	101	101	101
				Strategy	101	101	101
				Composite	9	9	9
GFP	0.1.0	360	189	Adapter	0	0	0
				Bridge	3	3	3
				Strategy	3	3	3
				Composite	0	0	0

source code. Our tool generates the matrices and weight vectors of the system as well as the design patterns to be discovered and performs the structural matching. It then performs the behavioral and semantic checking, when not only the two XML file are consulted again but also the source code locator is used so as to perform the search over the source code. Finally, it presents the result of the discovery and shows the lists of instances for each design pattern that have been recovered.

In this section, we present our experiments on six large open-source systems, namely, the Java AWT [42], JUnit [45], JEdit [43], JHotDraw [44], CDK [39], and GFP [47]. Java.AWT is a library for developing graphical user interfaces for Java programs. JUnit is a regression test framework that helps developers to implement unit tests in Java. JEdit is a mature text editor for programmers, which provides many features for ease of use. JHotDraw is a 2-D graphics framework for technical and structured drawing editors written in Java. The CDK is a Java library for structural chemo- and bioinformatics and computational chemistry. It provides methods for common tasks in molecular informatics. The GFP is designed for people with little financial knowledge, which can help in managing their finances. It offers a variety of reports and charts, which visualize the financial data. The left four columns of Table III show the system information in our experiments, including the versions of the systems and the number of classes and files of each system. We select these open-source software systems as the subjects of our experiments because the idea of design patterns was already mature and widely applied in the software industry at the time they were developed. Thus, these systems contain design patterns in their software design, and their usages of patterns follow the pattern principles introduced in [16].

We conduct the experiments to recover the instances of four design patterns, namely, Adapter, Bridge, Strategy, and Composite, from all six systems. Although we concentrate on the analysis of a small number of design patterns, our recovery tool is applicable for other design patterns. The structural, behavioral, and semantic characteristics of other patterns can be described in the XML file as an input of our tool because we separate our pattern-matching algorithm from pattern characteristics definition. Focusing on a small number of patterns allows us to study deeper in the pattern discovery problem.

The results of our recovery are shown in Table III which includes the structural, behavioral, and semantic analysis results. Each analysis phase may eliminate some false positives from previous phases. The results of the semantic analysis column in Table III show the final numbers of design pattern instances found in these six systems.

Our results show that the Composite pattern is the least used pattern, whereas the Bridge and Strategy patterns are the most used patterns among these four patterns. Our tool does not find any instance of the Composite pattern from the JEdit, JHotDraw, and GFP systems.

Table III shows that the behavioral analysis of the Adapter pattern has reduced the most percentages of false positives from the structural analysis. For example, the structural analysis of JHotDraw recovers 27 candidate instances of the Adapter pattern. The number of candidate instances is cut down to four after the behavioral analysis, which is about 85% reduction. The main reason is that the Adapter pattern includes more behavioral characteristics than other patterns.

We relax the criteria for structural analysis in order to reduce the number of false negative cases. For example, there should be an association relationship between the Abstraction and

Implementor classes in the Bridge pattern. In object-oriented programming languages, the association relationships can be implemented in many different ways, some of which may be hard to recognize by typical reverse engineering tools, such as IBM Rational Rose. As a consequence, some potential instances of the Bridge pattern may be filtered out at the structural analysis stage. In fact, the main purpose of such association relationship is to maintain a reference through which the Abstraction can delegate to the OperationImp in the Implementor class hierarchy. Therefore, the existence of such delegation, an invocation from the Operation method in the Abstraction class to the OperationImp method in the Implementor class hierarchy, actually implies the existence of such association relationship. Because our behavioral analysis checks this delegation, the criteria for the structural analysis are relaxed by not requiring the association relationships between the Abstraction and Implementor classes. Consequently, the structural analysis may result in a higher number of candidates. Our approach relies on the behavioral and semantic analyses to eliminate the false positives.

Our behavioral analysis is based on the results from the structural analysis. Thus, it deals with a smaller number of classes from the original systems. For example, the JHotDraw package contains a total of 530 classes. After structural analysis, only 27 candidate instances of the Adapter patterns are discovered. Thus, the search space of the behavioral analysis is reduced to the classes of these 27 pattern instances. After behavioral analysis, four candidate instances are left. This further limits the search space for the subsequent semantic analysis.

Among the six systems, Java.awt includes the majority of the instances of these four patterns even though it is not the largest system among them. In addition, the numbers of the instances of the Bridge and Strategy patterns are very similar. The main difference between them is their design intents, whether to define a family of algorithms or to decouple the abstraction from the implementation. The candidate sets obtained after the structural and behavioral analyses are the same for both patterns. The semantic analysis checks the naming conventions for clues. For example, one of the candidate instances of the potential Bridge or Strategy pattern contains a class called BufferStrategy in Java.awt. This is a good indication that it is a Strategy pattern instance.

For the CDK and GFP systems, our experiments show the same number of instances in each analysis phase. This indicates that, in these two systems, all design pattern candidates detected by the structural analysis carry the desired behavioral characteristic. That is the reason why the numbers of pattern instances remain the same after the behavioral analysis. It also indicates that the patterns in these two systems do not contain any semantic information. Therefore, our DP-Miner did not filter anything out.

Due to the lack of design documents of these open-source software systems, it is generally hard to validate the experimental results. Even if design documents are available, the system implementations may sometimes deviates from its original designs such that the design and implementation are inconsistent. In order to determine the precision of our approach, therefore, we manually checked the results generated by our tool and see whether they are real pattern instances. This manual checking was carried out with the results of JHotDraw as shown in Table IV, where TP and FP stand for true positive and false positive, respectively. It shows that there are five and six false positives in our result for the Bridge and Strategy patterns, respectively. These false positives are not eliminated during behavioral analysis. The main reason is that behavioral analysis sometimes requires checking the existence of some method invocation of an object; however, a method invocation may take many different forms in object-oriented programming languages. For example, a foo() method of an object can be invoked directly by object.foo(), where object can be a direct instance of the desired class, a copy of some other instance of the desired class, return value of a method with the desired class as return type, a cast of an object of the superclass of the desired class, etc. To deal with such complexity, DP-Miner checks only the existence of a method invocation to foo() without considering to which object it belongs. By including such checking in our behavioral analysis, it allows our tool to include more true positives. On the other hand, it may also introduce some false positives caused by the invocations of the foo() methods that belong to objects of undesired classes.

TABLE IV
RECOVERY PRECISION FOR JHOTDRAW

JHotDraw	TP	FP	Precision
Adapter	4	0	100%
Bridge	53	5	91.38%
Strategy	58	6	90.63%
Composite	0	0	100%
Total	115	11	91.27%

Table V shows the performance of our tool to recover the Adapter, Bridge, Strategy, and Composite patterns from six systems using a PC with 3.4-GHz Pentium processor, 1 GB of RAM, and Windows XP operating system. We separate the time to generate the system matrix and weights from the time to discover each design pattern because our tool only needs to generate the matrix and weights once for each given system. The discovery of different design patterns can reuse the matrix and weight of the system. Even though the total time for matrix and weight generation is relatively long, it may be amortized into a much shorter time for each pattern discovery. When many different patterns are required to be discovered from a single large system, our tool becomes more efficient. We also distinguish the display time from the generation and discovery time since the discovery results can be displayed in different ways. The display time may vary for different visualization methods, which is not a good indication of the core pattern discovery performance.

The time cost of pattern recovery increases with the size of the system because it takes time for our DP-Miner to read and process system information. Detecting pattern instances from CDK requires the longest time because this system contains the largest number of classes and the largest number of files.

To compare with other approaches, we list the performances of other approaches in Table VI. Niere *et al.* [28] had their analysis tool run up to 1307 s for the Java.AWT package. Dietrich and Elgar [10] mentioned their performance of

TABLE V
PERFORMANCE (IN MILLISECONDS)

System	Phases	Actions				
		Generate Matrix and Weights	Recover Adapter Instances	Recover Bridge Instances	Recover Strategy Instances	Recover Composite Instances
JavaAWT	Total Time	42423	8453	3985	4476	3751
	Generate/Recover Time	29891(70.5%)	8438(99.8%)	3954(99.2%)	4703(98.7%)	3735(99.6%)
	Display Time	12532(29.5%)	15(0.2%)	31(0.8%)	63(1.3%)	16(0.4%)
JUnit	Total Time	2532	172	281	218	500
	Generate/Recover Time	1954(77.2%)	297(86.3%)	266(94.7%)	203(93.1%)	485(97%)
	Display Time	578(22.8%)	31(9%)	15(5.3%)	15(6.9%)	15(3%)
JEdit	Total Time	97460	31815	5547	5173	17017
	Generate/Recover Time	50816(52.1%)	31783(99.9%)	5532(99.7%)	5157(99.7%)	16985(99.8%)
	Display Time	46644(47.9%)	32(0.1%)	15(0.3%)	16(0.3%)	16(0.1%)
JHotDraw	Total Time	24000	1562	2000	1891	2844
	Generate/Recover Time	17297(82.1%)	1515(97%)	1593(97.6%)	1875(99.2%)	2828(99.4%)
	Display Time	6703(27.9%)	31(2%)	32(1.6%)	16(0.8%)	16(0.6%)
CDK	Total Time	138863	8875	8969	9032	28251
	Generate/Recover Time	98424(70.9%)	8797(99.1%)	8875(99.0%)	8953(99.1%)	28204(99.8%)
	Display Time	40439(29.1%)	78(0.9%)	94(1%)	79(0.9%)	47(0.2%)
GFP	Total Time	11954	453	219	204	1125
	Generate/Recover Time	9813(82.1%)	391(86.3%)	188(85.8%)	188(92.2%)	1109(98.6%)
	Display Time	2141(17.9%)	62(13.7%)	31(14.2%)	16(0.8%)	16(2.4%)

TABLE VI
PERFORMANCE OF OTHER APPROACHES

System in [Reference]	#Class	LOC	Time (ms)
Java AWT in [28]		1,144,000	1,307,000
Java AWT and SWING in [10]	610		22,703
Galib++ in [7]	55	20,507	25,264
Libg++ in [7]	144	44,106	109,080
JHotDraw 5.1 in [24]	261		98,000
JHotDraw 5.1 in [9]	155	8,300	32,190
JHotDraw 6.0beta1 in [9]	544	24,222	263,190

searching for the Abstract Factory pattern instances in Java AWT and Swing, the Standard Java GUI Libraries, which took 22 703 ms. Costagliola *et al.* [7] conducted an experiment on Galib++ and Libg++. The performance turned out to be 25 264 ms and 109 080 ms, respectively. Kaczor *et al.* [24] mentioned the performance for JHotDraw 5.1 which requires 71 s for computation of string representations and 27 s for the identification of the Abstract Factory patterns. The two-phase approach [9] took 32 s to recover patterns from JHotDraw 5.1 and 263 s to recover patterns from JHotDraw 6.0beta1.

V. RELATED WORK

Antoniol *et al.* [2] propose to examine structural metrics and method delegation of design patterns in two phases. Structural metrics include the number of attributes, operations, and relations between classes. The intermediate representation in their approach is the Abstract Object Language AST. They conduct experiments with several public-domain software and industrial software. The results are presented in [1]. Different from their approach, we have three phases, namely, the structural, behavioral, and semantic analyses. To evaluate class structural information, we introduce weight and matrix calculated from structural metrics, which reduces the recovery process into arithmetic computation. We use XMI as the intermediate representation for system design and pattern.

Heuzeroth *et al.* use Prolog predicates to define the structural information of design patterns [21] and Prolog procedure based on the temporal logic of actions to define the behavioral aspect [22]. They use AST as the intermediate representation of the source code. Moreover, they develop a toolkit and conduct an experiment on the java source code of their toolkit. In contrast, our approach considers not only the structural and behavioral aspects of the design patterns but also the semantic meaning of source code. We use XMI, a standard format, to represent the source code under study.

A two-phase approach [9] is proposed to recover design patterns from system source code based on their previous work [7] using a visual language parsing technique. Design patterns are first expressed in terms of visual grammars, which can be parsed to generate the pattern candidates. In their second phase, the source code is examined to see whether the pattern candidates are positive instances. The properties checked in the second phase include both structural properties, such as inheritance relationship, and behavioral properties, such as method delegations. No semantic analysis was preformed. The structural analysis of our approach is based on the XMI format of the UML diagram, instead of visual grammar. We separate the structural, behavioral, and semantic analyses into three different phases. Their experiments show high precision as ours.

A minimal key element structure [32], [36] is defined for each design pattern by the required, forbidden, and don't-care elements. Both the positive and negative search criteria are defined to reduce false positives. The negative search criteria aim at refining the previous search results and are a mixture of both

structural and behavioral properties of patterns. Our approach reduces false positive, however, by using behavioral analysis, which focuses on checking method delegations in candidate instances. The structural analysis is usually completed before the second phase. We also have semantic analysis.

Guéhéneuc *et al.* [19] present a two-step pattern discovery approach, where the first step identifies candidate classes for key roles in design motifs by eliminating classes that do not match expected fingerprints, and the second step identifies candidate classes for the remaining roles starting from key-role candidates and using structural matching. Machine learning algorithms have been applied to classify the potential pattern candidates. In contrast, our multiphase approach identifies the classes for all roles in design patterns. We also conduct behavioral analysis in the second phase, instead of applying machine learning algorithms. In addition, they do not conduct semantic analysis.

The pattern recovery tool based on the FUJABA framework [41] also analyzes both structural [28], [29] and behavioral [38] aspects of design patterns. Structural analysis is performed on the AST of source code in both top–down and bottom–up manners, which increases detection precision and speeds up searching speed. Behavioral analysis is performed using runtime data acquired by manually setting breakpoints and getting the call stack with a debugging tool. Different from their manual data collection, our tool performs the behavioral characteristic collection automatically. Aside from the emphasis on the behavioral aspect, we noticed the usefulness of semantic analysis of source code, which helps to further eliminate false positive results.

Similarity scoring between graph vertices has been applied in the design pattern recovery process in [37]. The structural information of systems under study and design patterns is represented by graphs and matrices. Similarity scores between design pattern matrices and subsystem matrices are calculated using an inexact graph-matching algorithm and compared to a threshold. Our approach also uses a matrix as an intermediate representation. Instead of using multiple matrices, however, our approach uses one matrix to encode all kinds of structural information through prime numbers. This simplifies the pattern-matching process into arithmetic computations, instead of complex matrix calculations. Furthermore, our approach examines behavioral and semantic information to further eliminate false positives from the candidates acquired in the structural analysis.

PINOT [35] uses dataflow diagram analysis to examine the structural information of design patterns and control flow analysis to examine the behavioral information. Control flow analysis particularly works on the Singleton and Flyweight patterns which require control of behavior flow. PINOT has been tested on JDK, Java.awt, and JHotDraw.

The semantic information of source code is taken into account in [33], since naming conventions and programming guidelines leads to names of the classes containing particular keyword. Hedgehog [5] also explores the semantic meaning of the relationships between classes and method operations. Unlike the previous two studies, the approach presented in the paper emphasizes the pattern-related semantic meaning of source code.

Design Pattern Markup Language (DPML), a language extended from XML, is used to define design patterns in [3]. Source code, on the other hand, is analyzed and built into an Abstract Syntax Graph (ASG). Finding a design pattern in their approach relies on matching ASG substructures with the DPML description of the pattern. To avoid the complexity of matching between two specifications, we use XMI format to represent both source code and patterns. For class metrics and interclass relations particularly, we use weight and matrix to represent both source code and patterns. Therefore, finding matches between source code and patterns in the structural analysis phase becomes a simple computational problem.

VI. Conclusion and Future Work

In this paper, we have presented the formal specification of our design pattern recovery approach and experimental results on six open-source systems. Our approach uses XMI as the intermediate representation format to represent the UML diagram information extracted from source code. XMI is a standard for metadata exchange, which allows our approach to work with existing software design and development tools, such as IBM Rational Rose [48] and Eclipse [40]. Our approach includes structural, behavioral, and semantic analyses, each of which refines the results from the previous phases. Our approach also uses matrices and weights encoded by prime numbers to represent object-oriented design information, which facilitates the pattern-matching processes. Based on our approach, we have developed a tool, called DP-Miner [11], to recover design patterns. We also investigated the precision and recall of our approach in [13]. The recovery results can be naturally integrated with our pattern visualization tool [12] to dynamically display the pattern instances in a UML diagram because both of our tools are XMI-based.

We focused on recovering four design patterns in the experiments presented in this paper. There are certainly many more patterns, some of which are general and others are specific to some domains. We plan to continue our experiments on various patterns, including the new patterns presented for knowledge-based systems [8], agent-based systems [4], and robotics [17]. Our plan also includes experimenting more diverse systems in different domains. Our techniques and tools may help on the discovery of new patterns that recur in many systems but have not been documented in the literature.

References

[1] C. Alexander, S. Ishikawa, M. Silverstein, M. Jacobson, I. Fiksdahl-King, and S. Angel, *A Pattern Language*. London, U.K.: Oxford Univ. Press, 1977.

[2] G. Antoniol, R. Fiutem, and L. Cristoforetti, "Design pattern recovery in object-oriented software," in *Proc. 6th IEEE IWPC*, 1998, pp. 153–160.

[3] Z. Balanyi and R. Ferenc, "Mining design patterns from C++ source code," in *Proc. 19th IEEE ICSM*, Sep. 2003, pp. 305–314.

[4] E. A. Billard, "Patterns of agent interaction scenarios as use case maps," *IEEE Trans. Syst., Man, Cybern. B, Cybern.*, vol. 34, no. 4, pp. 1933–1939, Aug. 2004.

[5] A. Blewitt and A. Bundy, "Automatic verification of Java design patterns," in *Proc. 16th Annu. Int. Conf. ASE*, 2001, pp. 324–327.

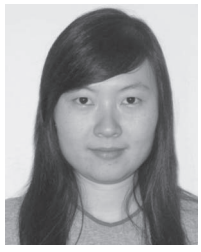
[6] G. Booch, J. Rumbaugh, and I. Jacobson, *The Unified Modeling Language User Guide*. Reading, MA: Addison-Wesley, 1999.

[7] G. Costagliola, A. De Lucia, V. Deufemia, C. Gravino, and M. Risi, "Design pattern recovery by visual language parsing," in *Proc. 9th Eur. CSMR*, 2005, pp. 102–111.
[8] L. Davis, R. F. Gamble, and S. Kimsen, "A patterned approach for linking knowledge-based systems to external resources," *IEEE Trans. Syst., Man, Cybern. B, Cybern.*, vol. 34, no. 1, pp. 222–233, Feb. 2004.
[9] A. De Lucia, V. Deufemia, C. Gravino, and M. Risi, "A two phase approach to design pattern recovery," in *Proc. 11th Eur. CSMR*, Mar. 2007, pp. 297–306.
[10] J. Dietrich and C. Elgar, "A formal description of design patterns using OWL," in *Proc. ASWEC*, 2005, pp. 243–250.
[11] J. Dong, D. S. Lad, and Y. Zhao, "DP-Miner: Design pattern discovery using matrix," in *Proc. 14th Annu. IEEE Int. Conf. ECBS*, Mar. 2007, pp. 371–380.
[12] J. Dong, S. Yang, and K. Zhang, "Visualizing design patterns in their applications and compositions," *IEEE Trans. Softw. Eng.*, vol. 33, no. 7, pp. 433–453, Jul. 2007.
[13] J. Dong and Y. Zhao, "Experiments on design pattern discovery," in *Proc. 3rd Int. Workshop PROMISE, in Conjunction With ICSE*, Minneapolis, MN, May 2007.
[14] J. Dong and Y. Zhao, "Classification of design pattern traits," in *Proc. 19th Int. Conf. SEKE*, Jul. 2007, pp. 473–476.
[15] J. Dong, Y. Zhao, and T. Peng, "A review of design pattern mining techniques," *Int. J. Softw. Eng. Knowl. Eng. (IJSEKE)*, 2009, to be published.
[16] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA: Addison-Wesley, 1995.
[17] A. R. Graves and C. Czarnecki, "Design patterns for behavior-based robotics," *IEEE Trans. Syst., Man, Cybern. A, Syst., Humans*, vol. 30, no. 1, pp. 36–41, Jan. 2000.
[18] D. G. Gregg and S. Walczak, "Exploiting the information web," *IEEE Trans. Syst., Man, Cybern. C, Appl. Rev.*, vol. 37, no. 1, pp. 109–125, Jan. 2007.
[19] Y. Guéhéneuc, H. Sahraoui, and F. Zaidi, "Fingerprinting design patterns," in *Proc. 11th WCRE*, 2004, pp. 172–181.
[20] W. Han and M. A. Jafari, "Component and agent-based FMS modeling and controller synthesis," *IEEE Trans. Syst., Man, Cybern. C, Appl. Rev.*, vol. 33, no. 2, pp. 193–206, May 2003.
[21] D. Heuzeroth, T. Holl, G. Hogstrom, and W. Lowe, "Automatic design pattern detection," in *Proc. 11th IWPC*, 2003, pp. 94–103.
[22] D. Heuzeroth, S. Mandel, and W. Lowe, "Generating design pattern detectors from pattern specifications," in *Proc. 18th IEEE Int. Conf. Autom. Softw. Eng.*, 2003, pp. 245–248.
[23] C.-C. Huang, T.-L. Tseng, and A. Kusiak, "XML-based modeling of corporate memory," *IEEE Trans. Syst., Man, Cybern. A, Syst., Humans*, vol. 35, no. 5, pp. 629–640, Sep. 2005.
[24] O. Kaczor, Y. Guéhéneuc, and S. Hamel, "Efficient identification of design patterns with bit-vector algorithm," in *Proc. CSMR*, 2006, pp. 175–184.
[25] C. Lin and M. Jeng, "An expanded SEMATECH CIM framework for heterogeneous applications integration," *IEEE Trans. Syst., Man, Cybern. A, Syst., Humans*, vol. 36, no. 1, pp. 76–90, Jan. 2006.
[26] P. De Meo, G. Quattrone, G. Terracina, and D. Ursino, "An XML-based multiagent system for supporting online recruitment services," *IEEE Trans. Syst., Man, Cybern. A, Syst., Humans*, vol. 37, no. 4, pp. 464–480, Jul. 2007.
[27] M. Matijasevic, D. Gracanin, K. P. Valavanis, and I. Lovrek, "A framework for multiuser distributed virtual environments," *IEEE Trans. Syst., Man, Cybern. B, Cybern.*, vol. 32, no. 4, pp. 416–429, Aug. 2002.
[28] J. Niere, W. Schafer, J. P. Wadsack, L. Wendehals, and J. Welsh, "Towards pattern-based design recovery," in *Proc. 24th Int. Conf. Softw. Eng.*, 2002, pp. 338–348.
[29] J. Niere, J. P. Wadsack, and L. Wendehals, "Handling large search space in pattern-based reverse engineering," in *Proc. 11th IEEE Int. Workshop Program Comprehension*, 2003, pp. 274–279.
[30] J. Paakki, A. Karhinen, J. Gustafsson, L. Nenonen, and A. Inkeri Verkamo, "Software metrics by architectural pattern mining," in *Proc. Int. Conf. Softw.: Theory Practice (16th IFIP World Comput. Congr.)*, Beijing, China, 2000, pp. 325–332.
[31] G. Pappalardo and E. Tramontana, "Automatically discovering design patterns and assessing concern separations for applications," in *Proc. ACM Symp. Appl. Comput.*, 2006, pp. 1591–1596.
[32] I. Philippow, D. Streitferdt, M. Riebisch, and S. Naumann, "An approach for reverse engineering of design patterns," *Softw. Syst. Modeling*, vol. 4, no. 1, pp. 55–70, Feb. 2005.
[33] J. Seemann and J. W. von Gudenberg, "Pattern-based design recovery of Java software," in *Proc. 6th ACM SIGSOFT Int. Symp. Found. Softw. Eng.*, 1998, pp. 10–16.
[34] W. Shen, S. Y. T. Lang, and L. Wang, "iShopFlor: An Internet-enabled agent-based intelligent shop floor," *IEEE Trans. Syst., Man, Cybern. C, Appl. Rev.*, vol. 35, no. 3, pp. 371–381, Aug. 2005.
[35] N. Shi and R. A. Olsson, "Reverse engineering of design patterns from Java source code," in *Proc. 21st IEEE/ACM Int. Conf. Autom. Softw. Eng.*, 2006, pp. 123–134.
[36] D. Streitferdt, C. Heller, and I. Philippow, "Searching design patterns in source code," in *Proc. 29th Annu. Int. COMPSAC*, 2005, pp. 33–34.
[37] N. Tsantalis, A. Chatzigeorgiou, G. Stephanides, and S. Halkidis, "Design pattern detection using similarity scoring," *IEEE Trans. Softw. Eng.*, vol. 32, no. 11, pp. 896–909, Nov. 2006.
[38] L. Wendehals, "Improving design pattern instance recognition by dynamic analysis," in *Proc. ICSE WODA*, May 2003, pp. 29–32.
[39] *Chemistry Development Kit*. [Online]. Available: http://cdk.sourceforge.net
[40] *Eclipse Website*. [Online]. Available: http://www.eclipse.org/
[41] *Fujaba User Documentation*. [Online]. Available: http://wwwcs.uni-paderborn.de/cs/fujaba/documents/user/manuals/FujabaDoc.pdf
[42] *Java.awt Resource Information*. Sep. 2006. [Online]. Available: http://java.sun.com/j2se/1.5.0/docs/guide/awt/index.html
[43] *JEdit—Programmer's Text Editor*. [Online]. Available: http://www.jedit.org/
[44] *JHotDraw Start Page*. [Online]. Available: http://www.jhotdraw.org/
[45] JUnit, *Testing Resources for Extreme Programming*. [Online]. Available: http://www.junit.org/
[46] *Model Driven Architecture*. [Online]. Available: http://www.omg.org/mda/
[47] *GFP—Personal Finance Manager*. [Online]. Available: http://gfd.sourceforge.net/
[48] *IBM Rational Rose Website*. [Online]. Available: http://www.ibm.com/software/rational/
[49] W3C, *Extensible Markup Language (XML)*. [Online]. Available: http://www.w3.org/
[50] *XML Metadata Interchange (XMI)*. [Online]. Available: http://www.omg.org/technology/documents/formal/xmi.htm

Jing Dong (S'98–A'02–M'03–SM'09) received the B.S. degree in computer science from Peking University, Beijing, China, in 1992 and the MMath and Ph.D. degrees in computer science from the University of Waterloo, Waterloo, ON, Canada, in 1997 and 2002, respectively.

He is currently an Assistant Professor with the Department of Computer Science, The University of Texas at Dallas, Richardson. His research and teaching interests include formal and automated methods for software engineering, software modeling and design, services computing, and visualization. He has had more than 100 publications in these areas.

Dr. Dong is a Senior Member of the ACM.



Yajing Zhao received the B.S. degree in computer science from Nankai University, Tianjin, China, in 2005 and the M.S. degree in software engineering from The University of Texas at Dallas, Richardson, in 2007, where she is currently working toward the Ph.D. degree in software engineering.

Her research interests include software architecture and design, service-oriented architecture, semantic web services, system modeling, and model transformation.

Yongtao Sun received the B.S. degree in mechanical engineering from the Beijing University of Aeronautics and Astronautics, Beijing, China, in 1996 and the M.S. degree in computer science from The University of Texas at Dallas, Richardson, in 2001, where he is currently working toward the Ph.D. degree in software engineering.

He is currently with the American Airlines, Fort Worth, TX. His research interests include software architecture and design, data mining, pattern recognition, and machine learning.