Efficient Identification of Design Patterns with Bit-vector Algorithm

Olivier Kaczor[1], Yann-Gaël Guéhéneuc[1], and Sylvie Hamel[2]
[1]GEODES – [2]LBIT
Department of Informatics and Operations Research
University of Montreal, Quebec, Canada
{kaczorol,guehene,hamelsyl}@iro.umontreal.ca

Abstract

Design patterns are important in software maintenance because they help in designing, in understanding, and in re-engineering programs. The identification of occurrences of a design pattern consists in identifying, in a program, classes which structure and organisation match—strictly or approximately—the structure and organisation of classes as suggested by the design pattern. We express the problem of design pattern identification with operations on finite sets of bit-vectors. We use the inherent parallelism of bit-wise operations to derive an efficient bit-vector algorithm that finds exact and approximate occurrences of design patterns in a program. We apply our algorithm on three small-to-medium size programs, JHotDraw, Juzzle, *and* QuickUML, *with the* Abstract Factory *and* Composite *design patterns and compare its performance and results with two existing constraint-based approaches.*

1 Introduction

Maintenance of object-oriented programs is difficult. It is a time- and resource-consuming activity that amounts to more than 50% of the total cost of a program [19, 21]. It is particularly difficult because documentation is often obsolete—if existing at all, and design information and choices are often lost. Therefore, a major task of maintainers during maintenance is design recovery. Design recovery consists in building higher-level abstractions from source code [5], the only source of information about a program always up-to-date, to understand the design and architecture of a program and to identify where to perform maintenance activities.

Design recovery benefits from the knowledge of used design patterns [10]. Indeed, design patterns provide "good" solutions to recurring design problems. Solutions of design patterns are design motifs, which are implemented in programs as micro-architectures: sets of entities (classes and interfaces) and of elements (methods, fields, binary class relationships). However, micro-architectures implementing design motifs are mingled in a program architecture and, therefore, important design decision are lost (design problems and the implemented solutions). The identification of micro-architectures similar to design motifs would help the design recovery task by highlighting potential uses of design patterns and, by extension, design problems and design choices made in a program architecture.

We address two aspects of design pattern identification: quality of the micro-architectures and quality of the identification process. Quality of the micro-architectures includes identifying complete and approximate occurrences of design motifs and the precision and recall of the identified occurrences. Quality of the identification process includes time- and resource-efficiency (cost of the process in processing time and memory), automation (automated versus manual process), and interaction (capability of following the maintainers' guidance).

Most previous approaches of design pattern identification are limited because of their performance. Some approaches use Prolog-like unification mechanism [26] or constraint programming [23], which have poor performance because of the combinatorial explosion of possible occurrences, *i.e.*, the possible combinations of entities in a program that form micro-architectures similar to a design motif. Other approaches based on metrics [1, 14] show promising increase in performance but are still too slow to be included in maintainers' day-to-day design recovery tasks.

We propose an efficient approach of design pattern identification using a high-performance bit-vector algorithm. Bit-vector algorithms are widely used in pattern matching [2, 15] and, more recently, in bio-informatics [3, 22]. We illustrate, with an example of a simple program and of the Composite design motif, the expression of the design pattern identification problem with bit-vectors operations and we detail our algorithm. Then, we describe a complete case study of our algorithm with JHotDraw [9], Juzzle, and QuickUML, and the Composite and Abstract Factory design motifs and we compare our algorithm with two existing approaches: explanation-based constraint programming [13] and metric-enhanced constraint programming [14].

Section 2 summarises related work and highlights their drawback; Section 3 describes our approach of design pattern identification using a dedicated bit-vector algorithm; Section 4 details our implementation; Section 5 presents a case study and a comparison; Section 6 discusses our algorithm and our approach; Finally, Section 7 concludes and introduces future work.

2 Related Work

Several works introduce approaches of design pattern identification. Most of the approaches use structural matching between micro-architectures and design motifs. Different structural matching techniques are used: rule inference [20, 26], queries [6, 18], fuzzy reasoning nets [16], constraint programming [13, 23]. For example, in his precursor work, Wuyts [26] introduces the SOUL environment. He describes design motifs as Prolog predicates and programs entities as facts. He applies a Prolog inference algorithm to unify predicates and facts and, thus, to identify entities playing roles in design motifs. The main problem of such a structural approach is the inherent combinatorial complexity of identifying subsets of entities matching design motifs, which corresponds to a problem of subgraph isomorphism [8]. Approaches bases on constraint programming [23] also face a combinatorial complexity, although explanations [17] help in reducing this complexity through user-interactions [13].

Antoniol *et al.* introduce an alternative approach, in which they reduce the search space using metrics [1]. They design a multi-stage filtering process to identify micro-architectures identical to design motifs using metrics. For each entity of a program, they compute some metrics (for example, numbers of inheritance, of association, and of aggregation relationships) and they compare the metric values with expected values for the corresponding role in a design motif, before applying a constraint-based structural matching. They infer expected metric values from design pattern descriptions manually. The main limitation of their work is the assumption that the micro-architectures accurately reflects the design motifs, which is rare. Moreover, the theoretical quantification of roles, when possible, does not reduce the search space significantly.

Recently, the second author and other collaborators attempted to improve on the two kinds of approaches by combining empirical metric values and explanation-based constraint programming [14]. Roles in design motifs are quantified empirically using P-MARt [14], a database of manually-identified micro-architectures similar to design motifs in several programs. This quantification is used to remove from the search space entities which *obviously* (from the empirical data) do not participate in a design motif. Explanation-based constraint programming is applied on the remaining entities to identify micro-architectures similar to design motifs. This approach shows promising results but suffers from a lack of data on manually-identified micro-architectures and from the performance of explanation-based constraint programming.

In this paper, we use a bit-vector algorithm to perform the structural matching between micro-architectures and design motifs. This matching is similar to sequence comparisons in bio-informatics. For example, duplication with modification is an essential process in proteins evolution. Genes mutations are also frequent in biology. Localising mutated genes in a long anonymous DNA sequence or modified proteins in a long amino-acid sequence are important problems in bio-informatics, which are similar to the identification of occurrences of design motifs in large programs. Authors tackle these problems in bio-informatics with approximate string matching and bit-vectors algorithms [3, 22]. Even if we cannot use their approaches for design pattern identification directly—a design motif being more a regular expression than a word—bit-vector algorithms are promising and have not been used but to represent binary decision diagrams [4].

3 Our approach

3.1 In a Nutshell

We summarise our approach of design pattern identification intuitively as follows: The identification of a design pattern consists in traversing in parallel a program and a design motif, entity from entity through elements, and in recording the entities in the program that match entities in the design motif, in structure and in organisation. Thus, design patterns identification is a combinatorial problem inherently, requiring all possible combinations of entities (through their elements)

to be compared against a design motif.

The use of a bit-vector algorithm for design pattern identification is interesting because such an algorithm could find the solution to the problem in a bounded number of vector operations, which is independent of the length of the program. Allowable operations in bit-vector algorithms are restricted to inherently-parallel bit-wise operations available in processors (including shifts), which implies that a bit-vector algorithm can be implemented efficiently.

In the next subsections, we show the application of bit-vector algorithms for design pattern identification. The first step is to convert a program and a design motif into strings, because bit-vector algorithms are designed for strings. First, we convert models of the design motif and of the program in digraphs (Subsection 3.2). Then, we convert these digraphs into Eulerian digraphs (Subsection 3.3) to generate unique string representations of the design motif and of the program (Subsection 3.4). Finally, we apply a new bit-vector algorithm on the string representations to identify exact and approximate occurrences of the design motif in the program efficiently (Subsections 3.5 and 3.6).

3.2 Design Motif and Program Models

Figure 1(a) shows the design motif of the Composite design pattern [10, page 163] with a UML-like graphic representation. A typical object-oriented program is represented statically by its source code describing the entities and elements interacting to perform some functionalities. Figure 2(a) shows the model of a simple example program with the same UML-like representation. Both representations are very similar and we can model design motifs and programs using a single formalism. We use a meta-model to describe entities and elements either forming a design motif or a program. A meta-model defines constituents which instances are entities and elements combined together to describe models of design motifs and of programs. A model of a design motif or of a program is actually a graph which vertices are entities and which edges are elements connecting entities. For the sake of simplicity, we only consider binary class relationships as elements: creation, specialisation, implementation, use, association, aggregation, and composition relationships. Thus, edges are directed because binary class relationships are directed. If more than one identical relationship (*e.g.*, two associations) exists between the two same entities, we only keep one relationship because the others do not provide extra information (see also Subsection 6.1).

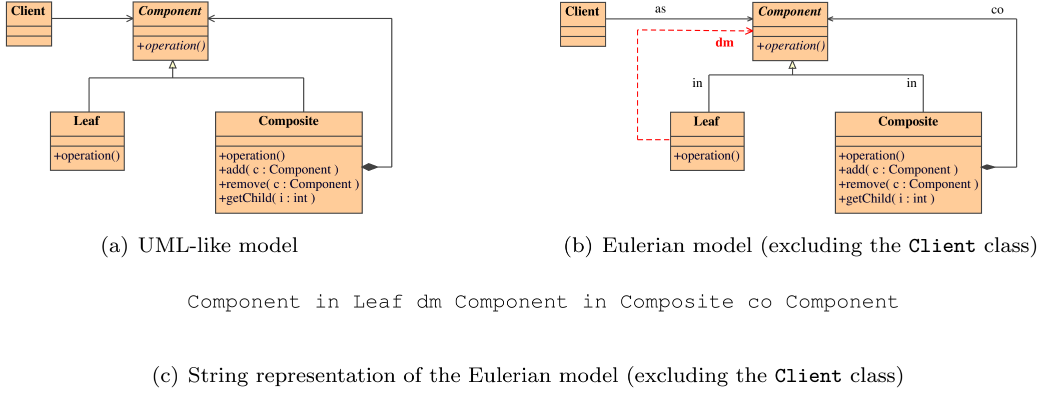
3.3 Eulerian Digraphs of Design Pattern and Program Models

From the previous representation, a model of a design motif or of a program is a digraph. A digraph is typically not Eulerian, *i.e.*, it does not contain a Eulerian circuit, a cycle which uses each edge exactly once. We transform a digraph of a design motif or of a program in a Eulerian graph automatically and consistently. A directed graph is Eulerian if and only if every vertex has equal in-degree and out-degree (respectively the numbers of incoming and outgoing edges for a vertex). The transformation consists in adding dummy edges between vertices with unequal in-degree and out-degree. We use the transportation simplex to obtain the number of dummy edges to be added among vertices. We consider the vertices with greater in-degree as suppliers and the vertices with greater out-degree as demanders. We assume uniform unitary shipping costs between suppliers and demanders. The transportation simplex computes the optimal solution (minimum cost), a list of flows among suppliers and demanders. In our case, a flow represents a dummy edge between vertices. If the flow is greater than one, then as many dummy edges must be added between the vertices. Figures 1(b) and 2(b) show the Eulerian models of the Composite design motif and of the simple example program.

3.4 String Representations of Design Pattern and Program Models

A Eulerian digraph contains a Eulerian circuit, which is a cycle traversing each edge exactly once. We compute the minimum Eulerian circuit using a dedicated algorithm to obtain unique string representations of a design motif and of a program models. The algorithm solves the directed Chinese Postman problem: the shortest tour of a graph which visits each edge at least once (see for example [7]). For a Eulerian graph, a Eulerian circuit is the optimal solution to the Chinese Postman problem.

Given a starting vertex v_s, the solution of the Chinese Postman problem is a unique list of edges starting and ending with v_s and containing all edges once. We iterate over the list of edges to build a unique (*wrt.* the root vertex) string representation of design motif and program models. Figures 1(c) and 2(c) show the string representations of the Composite design motif and of the example program. We discuss the uniqueness of a string representation in Subsection 6.1.



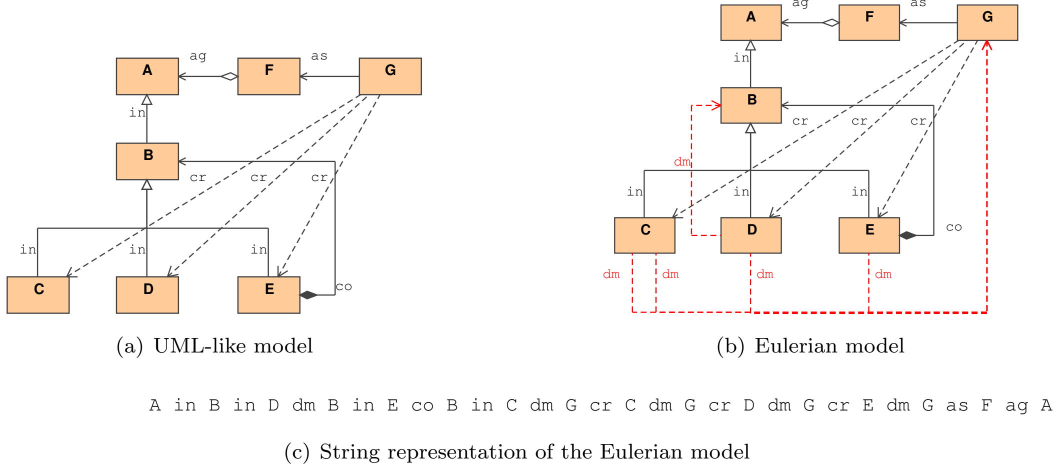
(a) UML-like model

(b) Eulerian model (excluding the `Client` class)

`Component in Leaf dm Component in Composite co Component`

(c) String representation of the Eulerian model (excluding the `Client` class)

Figure 1. Representations of the Composite design motif[1]



(a) UML-like model

(b) Eulerian model

`A in B in D dm B in E co B in C dm G cr C dm G cr D dm G cr E dm G as F ag A`

(c) String representation of the Eulerian model

Figure 2. Representations of a simple example program[1]

3.5 Iterative Bit-vector Algorithm

We match against each other the string representations of the design motif and of the program to identify, in the program, micro-architectures similar to the design motif. The length of the string representing a design motif is typically short, less than 20 tokens, while the length of a string representing a program might be arbitrarily long, depending on the size of the program to analyse, typically thousands of tokens.

We need an efficient mean to compute exact and approximate matches between the design motif and program string representations. We attempted to use an approximate string matching algorithm, developed in [3], but a design motif is more a regular expression than a word. Indeed, in Figure 1, the token `Component` in the design motif matches different classes (*i.e.*, `A`, `B`, `C`...) of the program, in Figure 2. Thus, we develop a dedicated iterative bit-vector algorithm to find exact and approximate occurrences of a design motif in a program. Let a token be any symbol appearing in a string `x` representing a program model, the characteristic vector of a token l associated to the string $\mathtt{x} = x_1 ... x_m$, denoted by the bold token $\boldsymbol{l}$, is

$$\boldsymbol{l}_i = \begin{cases} 1 & \text{if } x_i = l \\ 0 & \text{otherwise.} \end{cases}$$

For example, in the program in Figure 2, the characteristic vector of class `G` is

$$\boldsymbol{G} = \underbrace{00000000000000}_{14}10001000100010000$$

[1] In the models, `as`, `ag`, `co`, `cr`, `in`, and `dm` are association, aggregation, composition, creation (instantiation), inheritance, and dummy relationships.

while the vector for the inheritance relationship `in` is

$$\boldsymbol{in} = 010100010001\underbrace{0000000000000000000}_{19}.$$

Characteristic vectors are sequences of bits on which we operate with standard bit operations: bit-wise logical and, or operators, left and right shifts... Due to our construction of the string representations, tokens composing a design motif always appear in the same order modulo a shift, so we consider bit-vector as being *circular*. We define the *right* shift of a characteristic vector $\boldsymbol{x} = x_1 \dots x_m$ as $\rightarrow \boldsymbol{x} = x_m x_1 \dots x_{m-1}$ (all the elements have been shifted to the right by one position, circularly). Similarly, we defined the *left* shift of x as $\leftarrow \boldsymbol{x} = x_2 \dots x_m x_1$.

We use characteristic vectors to find the entities playing a role in a design motif. Our algorithm iteratively reads triplets of tokens (roles) in the design motif string representation and associates program entities to the roles by resolving a unification-like problem using the characteristic vectors. For example, if we want to identify the exact occurrences of the Composite design motif, in Figure 1, in the simple example program, in Figure 2, the algorithm reads the first triplet `Component in Leaf` and finds potential entities for the Component and Leaf roles in the string representation of the simple example program. It retrieves entities before and after the `in` token in the program string representation by applying bit-wise operations on its characteristic vectors. The following pseudo-code shows the retrieval of the entities before and after a specific token.

```
before := {}
after := {}
→token
FOR EACH ENTITY X IN THE STRING
    conjunctionX := X ∧ token
    IF conjunctionX IS NOT NULL
        ADD X IN after
        ←←conjunctionX
        FOR EACH ENTITY Y IN THE STRING
            conjunctionY := Y ∧ conjunctionX
            IF conjunctionY IS NOT NULL
                ADD Y IN before
            ENDIF
        ENDFOR
    ENDIF
ENDFOR
```

The algorithm now has initial sets of entities for the two roles. The next triplet represents a dummy relationship added by the transportation simplex and is therefore ignored. The algorithm performs the previous operations on the next triplet `Component in Composite`. However, it does not take potential Component entities in the set of all entities but in the Component role set, because it has already read the `Component` token. Thus, sets of entities represent potential occurrences and the algorithm tests each occurrence repeatedly after each triplet. In the case of the triplet `Component in Composite`, the algorithm searches for all possible entities for the Composite role for each occurrence by verifying if the conjunction between every entities characteristic vectors and the conjunction of the $\rightarrow \boldsymbol{in}$ characteristic vector and the $\rightarrow\rightarrow \boldsymbol{Component}$ characteristic vector is not null. For example, with the current occurrence {`Component = B, Leaf = C`}, we compute the following operations on the characteristic vectors

$$\begin{aligned}
\rightarrow\rightarrow \boldsymbol{B} &= 0000100010001\underbrace{0\dots0}_{18}\\
\rightarrow \boldsymbol{in} &= 0010100010001\underbrace{0\dots0}_{18}\\
(\rightarrow\rightarrow \boldsymbol{B}) \wedge (\rightarrow \boldsymbol{in}) &= 0000100010001\underbrace{0\dots0}_{18}\\
\boldsymbol{E} &= \underbrace{00000000}_{8}1\underbrace{0\dots0}_{15}1\underbrace{000000}_{6}\\
(\rightarrow\rightarrow \boldsymbol{B}) \wedge (\rightarrow \boldsymbol{in}) \wedge \boldsymbol{E} &= \underbrace{00000000}_{8}1\underbrace{0\dots0}_{22}.
\end{aligned}$$

Occurrence {`Component = B, Leaf = C, Composite = E`} is added to the list of occurences because $(\rightarrow\rightarrow \boldsymbol{B}) \wedge (\rightarrow \boldsymbol{in}) \wedge \boldsymbol{E}$ is not null: The entity `E` is found after the tokens `B in` in the program string representation.

Table 1 shows the occurrences after, respectively, the first, the third and the fourth triplets have been processed. The second triplet is ignored since it represents a dummy relationship. The order in which the triplets are read influences the identification time. It is preferable to treat less frequent relationships first to reduce the number of potential occurrences early in the process. This can be done by giving different weights to edges when resolving the Chinese Postman problem or by doing a post-treatment on a design motif string representation. For example, the Composite string representation could be read circularly beginning with the `Composite` token, so that the `co` relation would be treated first. Table 2 shows that it decreases the number of potential occurrences *wrt.* Table 1.

3.6 Approximate Iterative Bit-vector Algorithm

Design pattern identification can be considered as a unification problem (see the large body of work us-

Triplets									
First (in)		Third (in)			Fourth (co)				
Component	Leaf	Component	Leaf	Composite	Component	Leaf	Composite		
A	B	A	B	B	B	C	E		
B	C	B	C	C	B	D	E		
B	D	B	C	D	B	E	E		
B	E	B	C	E					
		B	D	C					
		B	D	D					
		B	D	E					
		B	E	C					
		B	E	D					
		B	E	E					

Table 1. Occurrences after processing the first, third, and fourth triplets.

Triplets							
First (co)		Second (in)			Fourth (in)		
Composite	Component	Composite	Component	Leaf	Composite	Component	Leaf
E	B	E	B	C	E	B	C
		E	B	D	E	B	D
		E	B	E	E	B	E

Table 2. Occurrences after processing the first, second, and fourth triplets.

ing Prolog-unification mechanism or constraint programming). However, it is an *approximate* unification problem, because a program model rarely reflects a design motif completely: Often, entities and elements are added or removed to integrate the design motif—solution of a *general* design problem—within the program architecture—where the design problem occurs *specifically*. Thus, we are not interested only in strict matches between models of design motif and of programs. For example, there are some micro-architectures similar to the Composite design motif where the composition relationship is replaced by an aggregation relationship and where a class is inserted in the Component hierarchy, see Figure 3.

We include automatic and manual approximation mechanisms in our iterative bit-vector algorithm. Maintainers can perform approximations manually by specifying which relationships should be *relaxed* but de-

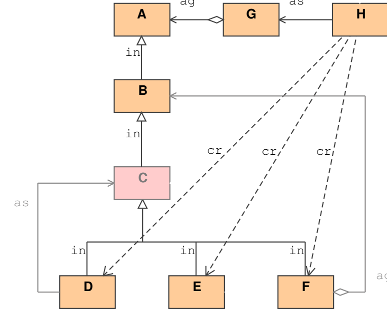


Figure 3. Example program with approximate occurrences of the Composite design motif[1]

scribing all possible approximations is not reliable because one possible approximation could be overlooked and because describing all possible approximations is time-consuming, tedious, and difficult to maintain. An automatic mechanism of approximations can be used to compute and to explain identified micro-architectures to maintainers by stating explicitly what parts of a design motif are not strictly implemented. The approximately identified micro-architectures, although they must be manually inspected by the maintainers, help in improving the code by applying corrections based on the design motifs.

Association Relationships. Approximate occurrences of design motifs with respect to association, aggregation, and composition relationships are easily obtained with bit-vectors. We use a conjunction of the characteristic vectors of the relaxed relationships in addition to characteristic vector of the expected relationship, following the order: composition, aggregation, association, and finally use relationships [12].

Entity Counts. All roles in a design motif must not necessarily be played by an entity in a program. For example, it is possible to find micro-architectures implementing the Composite design motif but without a Leaf. To identify those occurrences, all relationships may be relaxed or removed completely. Thus, a role can be overlooked if all the relationships related to that role are removed.

Inheritance Relationship. The design motif hierarchy can also be approximate. Indeed, some entities could be inserted or removed from a hierarchy. A documented occurrence of the Composite design motif in

JHotDraw v5.1 has a class inserted between the ones playing the roles of Component and Composite. Those approximate occurrences can be identified by adding occurrences with the parents and children of a class playing a role in a design motif as possible entities playing that role in the list of potential occurrences. We currently access the program model to retrieve the inheritance tree because it is directly available (see also Subsection 6.4).

4 Tools

We integrate freely available tools loosely to implement our approach to design pattern identification using our bit-vector algorithm. For performance and convenience reason, we decompose the iterative bit-vector algorithms in two parts: (1) Construction of the characteristic vectors; (2) Design pattern identification through unification, based on characteristic vectors.

Design Motif and Program Models. We use the PADL meta-model [11] to describe design motifs and programs. The PADL meta-model defines all the constituents required to describe the static structure of design motifs and of programs and part of their behaviour, including binary class relationships and message sends. The PADL meta-model is associated with several parsers to build models of programs from AOL, C++, Java. It also includes a design motif repository containing several well-known design patterns, such as Abstract Factory, Composite, Facade.

Eulerian Graphs of Design Pattern and Program Models. We iterate through the PADL models of a design motif and of a program to identify the entities with unequal in-degree and out-degree using adjacency matrices. Then, we use Park's implementation of the transportation simplex[2] to build flows among entities with unequal degrees. The obtained flows are added as dummy relationships in the design motif and program models, which thus become Eulerian digraphs.

String Representations of Design Pattern and Program Models. After transforming design motif and program models in Eulerian digraphs, we build string representations using Thimbleby's efficient implementation of an algorithm to solve the Chinese Postman problem [25]. This implementation uses several well-known algorithms for efficiency, such as Floyd-Warshall's for shortest path, cycle cancelling.

[2]See www.orlab.org.

Iterative bit-vector algorithm. We develop an iterative bit-vector algorithm in Java using the Eclipse platform. We use a sparse vector representation because our characteristic vectors can be long and because the majority of the bits in the vectors are 0-valued. This representation is backed by a hash map and only the 1-valued bits are stored in the map. This ensures space-efficiency. This is quite important because the number and the lengths of characteristic vectors can be large.

5 Case Study

In previous work, constraint programming showed promising results with respect to the quality of the identified micro-architectures, while metrics decrease identification time significantly. We compare our bit-vector algorithm to Ptidej, a framework dedicated to the analysis and maintenance of object-oriented architectures implementing metric-enhanced explanation-based constraint programming.

We apply Ptidej and our algorithm on three public domain software, JHotDraw v5.1, Juzzle v0.5, and QuickUML 2001. JHotDraw is a framework for technical and structured graphics, Juzzle is a puzzle game, and QuickUML is a UML class-diagram graphic editor. These programs are written in Java and are composed of, respectively, 261, 99, and 373 entities, which represent small-to-medium size programs. We compare Ptidej and our algorithm using the Abstract Factory and Composite design patterns, because these are well-known design patterns with different intents and solutions. We compare the three approaches with the Abstract Factory and Composite design motifs but all three could identify other design patterns.

The first part of our algorithm consists in building a string representation of the programs to compute the related characteristic vectors, using the transportation simplex and solving the Chinese postman problem. This computation is only performed once. Table 3 presents the average computation time on our test machine, an AMD Athlon 64bits at 2GHz. We retrieve identification times using the Eclipse-based profiling tool, Eclipse Profiler [24]. We perform all computations three times and report median times.

Programs	Sizes	Computation times
JHotDraw v5.1	261	71
Juzzle v0.5	99	5
QuickUML 2001	373	149

Table 3. Computation times (in seconds) for building the string representations

	CP	CP+M	BV	BV+O
	Abstract Factory			
JHOTDRAW v5.1	1202	275	218	27
JUZZLE v0.5	1	2	0.9	0.3
QUICKUML 2001	785	153	97	29
	Composite			
JHOTDRAW v5.1	$+\infty$	17362	129	25
JUZZLE v0.5	45	3	0.5	0.3
QUICKUML 2001	$+\infty$	26514	185	27

Table 4. Identification times (in seconds) of the design motifs

Table 4 presents the identification times in seconds when using explanation-based constraint programming (CP), metric-enhanced explanation-based constraint programming (CP+M), our bit-vector algorithm (BV), and an optimised version of our bit-vector algorithm (BV+O). The optimisations implemented in our algorithm (BV+O) to reduce identification times are, for example, computing the conjunctions of the characteristic bit-vectors for every entities with all relationships up-front and only once for every relationship in a design motif string representation.

Although several works exist on design pattern identification, none clearly states what it considers as *one* occurrence of a design motif. For example, in the case of the Composite design motif and of the simple example program in Figures 1 and 2, there could be one or two occurrences of the design motif

$\{\mathsf{Component} = \mathtt{B}, \mathsf{Composite} = \mathtt{F}, \mathsf{Leaf} = \{\mathtt{D}, \mathtt{E}\}\}$

or

$\{\mathsf{Component} = \mathtt{B}, \mathsf{Composite} = \mathtt{F}, \mathsf{Leaf} = \mathtt{D}\}$
$\wedge \{\mathsf{Component} = \mathtt{B}, \mathsf{Composite} = \mathtt{F}, \mathsf{Leaf} = \mathtt{E}\}$.

Whether an occurrence of the Composite design motif includes several leaves potentially or only one leaf is important because it varies between tools. In our case, we consider that micro-architectures similar to the Composite design motif with three leaves counts as three occurrences. We use this definition of occurrence to compare our results with those of PTIDEJ. Table 5 presents the number of occurrences of the Abstract Factory and Composite design motifs identified by the three approaches, constraint-programming (CP), constraint-programming with metrics (CP+M) and bit-vector (BV) on the three programs. It also presents the numbers of occurrences without *ghost* entities (entities known only through references).

We analyse the programs manually to identify micro-architectures similar to design motifs to validate the results. We do not claim that all micro-architectures similar to design motifs in a given program have been identified manually but the existing occurrences counted in Table 5 should be identified by the three systems. The numbers of approximate occurrences identified by the constraint-based approaches are very high because these approaches perform more automatic approximations. Indeed, when no occurrence is found, a constraint is replaced or removed until there is no more constraint. However, most of these occurrences are unrelated to the design motif intent. The number of retrieved occurrences by the bit vector approach is also high because of the approximations performed but they can be sorted by their distance to an exact occurrence. Our definition of an occurrence also contributes to the high numbers. Indeed, the 22 existing occurrences of the Composite design motif in QUICKUML represent only 2 micro-architectures with several leaves.

These results show the reliability of the three approaches because all existing occurrences of the Abstract Factory and Composite design motifs have been identified as exact or approximate occurrences. They also show the efficiency of the bit-vector approach compared with the other approaches. Indeed, several orders of magnitude separate the performance of our algorithm with constraint-based approaches. Nevertheless, the use of metrics is interesting because it reduces the noise and could be combined with our bit-vector algorithm to increase precision and performance even more.

6 Discussions

Our algorithm is interesting because of its efficiency in comparison to previous approaches. However, a number of concerns arose during its development and its study.

6.1 Impact of the Root Vertex on the String Representations

The generation of the string representations is modelled as a Chinese Postman problem. The resolution of the Chinese Postman problem requires choosing a root vertex for the traversal of the digraph. The choice of the root vertex impacts the string representations: The string representation of the Composite design motif from the `Component` class is: `Component in Leaf dm Component in Composite co Component`, while from the `Composite` class, it is: `Composite co Component in Leaf dm Component in Composite`.

Although different, these string representations are identical when considered as circular sequences, *i.e.*, the last token in a string *is* the first token in the string. Indeed, when first entering a vertex, our implementation of the Chinese Postman algorithm always chooses

	Existing Occurrences	Exact			Approximate		
		CP	CP+M	BV/BV+O	CP	CP+M	BV/BV+O
		Abstract Factory					
JHotDraw v5.1	0	216/221[3]	104/69	216/221	5245/2994	1444/849	408/194
Juzzle v0.5	0	19/0	0/0	19/0	179/9	6/0	53/13
QuickUML 2001	13	164/57	46/23	164/57	2002/593	356/124	273/118
		Composite					
JHotDraw v5.1	70	N/A	0/0	0/0	N/A	31709/16983	1083/609
Juzzle v0.5	0	0/0	0/0	0/0	1726/20	0/0	72/0
QuickUML 2001	22	N/A	0/0	0/0	N/A	14920/4743	5536/513

Table 5. Number of identified occurrences of the design motifs with and without ghost entities

the same edge to leave a vertex, using the edge with the smallest weight and a lexicographic order among equally-weighted edges.

6.2 Impact of the Direction of the Generalisation Relationships

We chose to describe specialisation (respectively implementation) relationships as directed edges from the specialised (implemented) entity to the specialising (implementing) entity. This choice impacts the solution of the transportation simplex used to build Eulerian digraphs and, thus, the string representations. However, the transformation of the digraphs in Eulerian digraphs absorbs this choice by adding dummy edges symmetrically when appropriate.

6.3 Approximate String Matching vs. Regular Bit-vectors Comparisons

Our first idea to identify all the approximate occurrences of a design motif in a program was to use an approximate string matching bit-vector algorithm, developed by Bergeron–Hamel in [3]. However, a design motif is more like a regular expression than a word. To the best of our knowledge, no bit-vector algorithms exist to find occurrences (or approximate occurrences) of a regular expression in a text. We decided to build an iterative bit-vector algorithm to identify all approximate occurrences of a design motif, because our main goal is to devise a really efficient algorithm in terms of time-complexity. We are now studying the possibility of generalising the approach described in [3] for a more direct approach to design pattern identification.

6.4 Performance of Characteristic Vectors

We obtain satisfactory results using characteristic vectors to find sets of entities before and after a relationship. However, interrogating the PADL model to find parents or children of a specific entity is faster than manipulating the characteristic vectors. Indeed, this data is available in the PADL models directly.

[3]Number of occurrences with/without ghost entities

7 Conclusion

We presented an adaptation of bio-informatics bit-vector algorithms to the software maintenance problem of design pattern identification. We detailed the conversion of the problem of identification of micro-architectures similar to design motifs in a problem of approximate string matching using bit-vectors. We implemented our approach and showed its efficiency on three small-to-medium size programs and the quality of its results, including its approximation capabilities. Thus, we addressed the two aspects of design pattern identification: Quality of the occurrences and quality of the identification process in time, resource, automation. Interactions with maintainers are possible when computing approximate occurrences.

Our algorithm can identify any design pattern found in the PADL design pattern repository by generating its string representation. However, the precision of the identification can vary depending on the design pattern. Indeed, our string representations contain what must be found but some design patterns also specify what must not be found. For example, the Adaptee role in the Adapter design motif must not know the Adapter role. The number of spurious occurrences could be reduced if our string representations include those *ignorance* relationships. An ignorance relationship can be considered as a set of negative relationships. Negative relationships are easy to manipulate with bit-vectors using negation operations.

As future work, we want to explore other ways to use our string representations of the design motifs and programs. One idea is to view the string representation of a design motif as a regular expression and, then, find a way to build automatically an automaton recognising the expression exactly or approximately. One passage of the string representation of a program through the automaton would provide the occurrences that we are looking for and their emplacements in the program.

Acknowledgement

The authors thank Jean-Yves Potvin for his invaluable help and the many fruitful discussions. This work has been partly funded by NSERC.

References

[1] G. Antoniol, R. Fiutem, and L. Cristoforetti. Design pattern recovery in object-oriented software. In *proceedings of the* 6th *International Workshop on Program Comprehension*, pages 153–160. IEEE Computer Society Press, June 1998.

[2] R. A. Baeza-Yates and G. H. Gonnet. A new approach to text searching. In *Communications of the ACM*, 35(10):74–82, October 1992.

[3] A. Bergeron and S. Hamel. Vector algorithms for approximate string matching. In *International Journal of Foundation of Computer Science*, 13(1):53–66, February 2002.

[4] D. Beyer, A. Noack, and C. Lewerentz. Simple and efficient relational querying of software structures. In *proceedings of the* 10th *Working Conference on Reverse Engineering*, pages 216–225. IEEE Computer Society Press, November 2003.

[5] T. J. Biggerstaff, B. G. Mitbander, and D. E. Webster. The concept assignment problem in program understanding. In *proceedings of the* 15th *International Conference on Software Engineering*, pages 482–498. IEEE Computer Society Press / ACM Press, May 1993.

[6] O. Ciupke. Automatic detection of design problems in object-oriented reengineering. In *proceeding of* 30th *conference on Technology of Object-Oriented Languages and Systems*, pages 18–32. IEEE Computer Society Press, August 1999.

[7] H. A. Eiselt, M. Gendreau, and G. Laporte. Arc routing problems. part I: The Chinese Postman problem. Technical Report CRT-960, Centre de Recherche sur les Transports, March 1994.

[8] D. Eppstein. Subgraph isomorphism in planar graphs and related problems. In *proceedings of the* 6th *annual Symposium On Discrete Algorithms*, pages 632–640. ACM Press, January 1995.

[9] E. Gamma and T. Eggenschwiler. JHotDraw. Web site, 1998. members.pingnet.ch/gamma/JHD-5.1.zip.

[10] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns – Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1st edition, 1994.

[11] Y.-G. Guéhéneuc. Ptidej: Promoting patterns with patterns. In *proceedings of the* 1st *ECOOP workshop on Building a System using Patterns*. Springer-Verlag, July 2005.

[12] Y.-G. Guéhéneuc and H. Albin-Amiot. Recovering binary class relationships: Putting icing on the UML cake. In *proceedings of the* 19th *conference on Object-Oriented Programming, Systems, Languages, and Applications*. ACM Press, October 2004.

[13] Y.-G. Guéhéneuc and N. Jussien. Using explanations for design-patterns identification. In *proceedings of the* 1st *IJCAI workshop on Modeling and Solving Problems with Constraints*, pages 57–64. AAAI Press, August 2001.

[14] Y.-G. Guéhéneuc, H. Sahraoui, and F. Zaidi. Fingerprinting design patterns. In *proceedings of the* 11th *Working Conference on Reverse Engineering*, pages 172–181. IEEE Computer Society Press, November 2004.

[15] J. Holub and B. Melichar. Implementation of nondeterministic finite automata for approximate pattern matching. In *proceedings of the* 3rd *international Workshop on Implementing Automata*, pages 92–99. Springer-Verlag, September 1998.

[16] J. H. Jahnke, W. Schäfer, and A. Zündorf. Generic fuzzy reasoning nets as a basis for reverse engineering relational database applications. In *proceedings of the* 6th *European Software Engineering Conference*, pages 193–210. ACM Press, September 1997.

[17] N. Jussien. e-Constraints: Explanation-based constraint programming. In 1st *CP workshop on User-Interaction in Constraint Satisfaction*, December 2001.

[18] R. K. Keller, R. Schauer, S. Robitaille, and P. Pagé. Pattern-based reverse-engineering of design components. In *proceedings of the* 21st *International Conference on Software Engineering*, pages 226–235. ACM Press, May 1999.

[19] J. Koskinen. Software maintenance costs, September 2004. Web site.

[20] C. Krämer and L. Prechelt. Design recovery by automated search for structural design patterns in object-oriented software. In *proceedings of the* 3rd *Working Conference on Reverse Engineering*, pages 208–215. IEEE Computer Society Press, November 1996.

[21] B. P. Lientz and E. B. Swanson. Problems in application software maintenance. In *Communications of the ACM*, 24(11):763–769, November 1981.

[22] G. Myers. A fast bit-vector algorithm for approximate string matching based on dynamic programming. In *journal of the ACM*, 46(3):395–415, May 1999.

[23] A. Quilici, Q. Yang, and S. Woods. Applying plan recognition algorithms to program understanding. In *journal of Automated Software Engineering*, 5(3):347–372, July 1997.

[24] K. Scheglov and J.-M. P. Shackelford. Eclipse profiler, September 2004.

[25] H. W. Thimbleby. The directed chinese postman problem. In *journal of Software – Practice and Experience*, 33(11):1081–1096, September 2003.

[26] R. Wuyts. Declarative reasoning about the structure of object-oriented systems. In *proceedings of the* 26th *conference on the Technology of Object-Oriented Languages and Systems*, pages 112–124. IEEE Computer Society Press, August 1998.