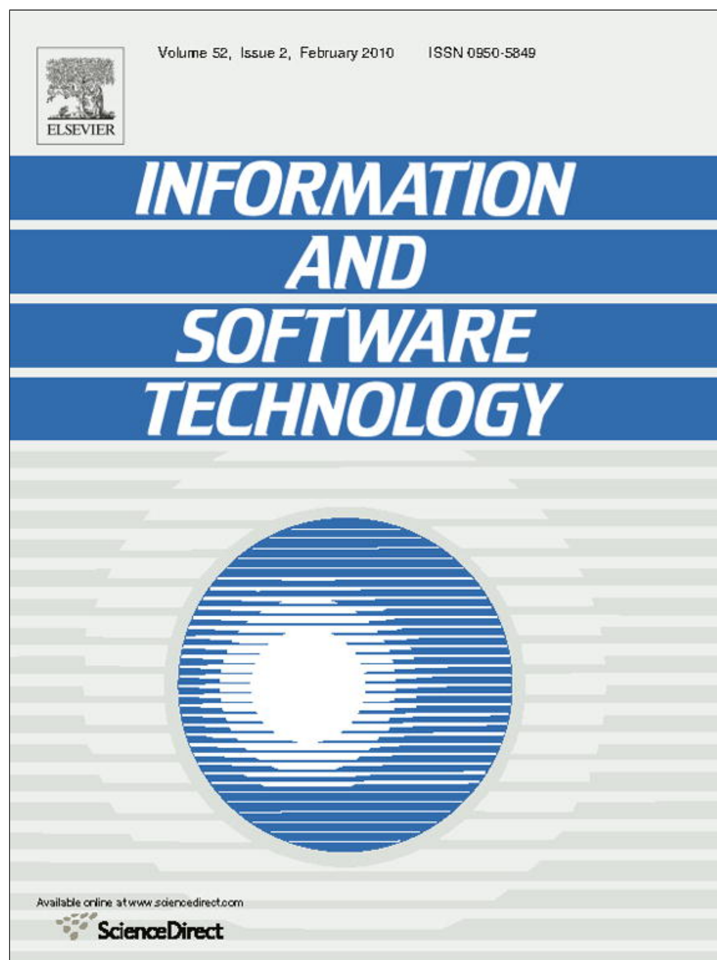
Provided for non-commercial research and education use.
Not for reproduction, distribution or commercial use.




This article appeared in a journal published by Elsevier. The attached copy is furnished to the author for internal non-commercial research and education use, including for instruction at the authors institution and sharing with colleagues.

Other uses, including reproduction and distribution, or selling or licensing copies, or posting to personal, institutional or third party websites are prohibited.

In most cases authors are permitted to post their version of the article (e.g. in Word or Tex form) to their personal website or institutional repository. Authors requiring further information regarding Elsevier's archiving and manuscript policies are encouraged to visit:

http://www.elsevier.com/copyright

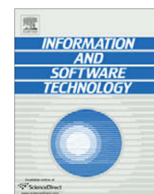
Author's personal copy

Information and Software Technology 52 (2010) 152–168

Contents lists available at ScienceDirect

Information and Software Technology

journal homepage: www.elsevier.com/locate/infsof



# Identification of design motifs with pattern matching algorithms

Olivier Kaczor, Yann-Gaël Guéhéneuc *, Sylvie Hamel

*DIRO, Université de Montréal, C.P. 6128 succursale Centre Ville, Montréal, Québec, Canada H3C 3J7*

ARTICLE INFO

*Article history:*
Received 21 January 2009
Received in revised form 22 August 2009
Accepted 24 August 2009
Available online 31 August 2009

*Keywords:*
Design patterns
Design motifs
Identification of occurrences
Bit-vector
Automata simulation
Experimental validation

ABSTRACT

Design patterns are important in software maintenance because they help in understanding and re-engineering systems. They propose design motifs, solutions to recurring design problems. The identification of occurrences of design motifs in large systems consists of identifying classes whose structure and organization match exactly or approximately the structure and organization of classes as suggested by the motif. We adapt two classical approximate string matching algorithms based on automata simulation and bit-vector processing to efficiently identify exact and approximate occurrences of motifs. We then carry out two case studies to show the performance, precision, and recall of our algorithms. In the first case study, we assess the performance of our algorithms on seven medium-to-large systems. In the second case study, we compare our approach with three existing approaches (an explanation-based constraint approach, a metric-enhanced explanation-based constraint approach, and a similarity scoring approach) by applying the algorithms on three small-to-medium size systems, JHotDraw, Juzzle, and QuickUML. Our studies show that approximate string matching based on bit-vector processing provides efficient algorithms to identify design motifs.

© 2009 Elsevier B.V. All rights reserved.

## 1. Introduction

Maintenance of object-oriented systems is a time- and resource-consuming activity that amounts to more than 50% of the total cost of the systems [1,2]. Documentation is often obsolete, if existing, and design information and decisions are often lost. A major task of maintainers is design recovery, which consists of building higher-level abstractions from source code [3], the only up-to-date source of information about a system, to understand the design and identify where to perform maintenance activities.

Design recovery benefits from the knowledge of design patterns [4] used by developers during design and implementation. Indeed, design patterns provide *design motifs* that are "good" solutions to recurring design problems. Design motifs are the solutions advocated by the design patterns, which are implemented in systems as *micro-architectures* where different *actual* entities (classes and interfaces) and elements (methods, fields, binary class relationships) play the roles defined in the motifs, for example a class `AttributeFigure` may play the role of `Leaf` defined in the `Composite` design motif [4, p. 163].

The identification of micro-architectures similar to design motifs helps the design recovery task by highlighting potential uses of design motifs and, by extension, design problems and design decisions made in a system architecture. The intent of the developers having implemented the micro-architectures may not have been to solve the related design problems; yet, the knowledge that these micro-architectures are similar to some motifs may help other developers to grasp the complexity and the relationships among the classes in the micro-architectures and the rest of the system. However, micro-architectures implementing design motifs are spread in a system architecture and, therefore, are difficult for maintainers to identify manually through code inspection. Important design decisions are lost.

Most previous approaches of design motif identification are often limited because of their time performance. For example, some approaches used a Prolog-like unification mechanism [5] or constraint programming [6], which are slow because of the combinatorial explosion of possible occurrences, *i.e.* the possible combinations of entities in a system that form micro-architectures similar to a design motif. Other approaches based on metrics [7,8] showed a promising increase in performance but are still too slow to be included in the maintainers' day-to-day design recovery tasks. Tsantalis et al. [9] recently proposed an approach based on similarity scoring with landmarking performance, as discussed in Section 2.

The structural matching between micro-architectures and design motifs is similar to sequences comparisons in bioinformatics. Indeed, duplication with modification is an essential process in protein evolution, and gene mutations are also frequent in biology [10]. Localizing mutated genes in a long anonymous DNA sequence or modified proteins in a long amino-acid sequence are important

* Corresponding author. Present address: Department of Computer Engineering and Software Engineering at École Polytechnique de Montréal. Tel.: +1 514 343 6782; fax: +1 514 343 5834.

*E-mail addresses:* kaczorol@iro.umontreal.ca (O. Kaczor), guehene@iro.umontreal.ca (Y.-G. Guéhéneuc), hamelsyl@iro.umontreal.ca (S. Hamel).

0950-5849/$ - see front matter © 2009 Elsevier B.V. All rights reserved.
doi:10.1016/j.infsof.2009.08.006

Author's personal copy



problems in bioinformatics, which are similar to the identification of occurrences of design motifs in large systems. Authors tackle these problems in bioinformatics with approximate string matching algorithms. Their algorithms are efficient because the length of the DNA or amino-acid sequences can reach billions of characters. Dynamic programming [11–13], automata simulation [14,15], and bit-vector processing [16–19] are three interesting pattern matching approaches. Yet, they cannot be used directly for design motif identification because they are designed for strings, and a design motif is more like a regular expression than a word.

While regular expression matching cannot be resolved easily and efficiently with a dynamic programming approach, it can be done with bit-vector processing and automata simulation algorithms. Bit-vector processing algorithms are particularly promising, for example they have been used to represent binary decision diagrams [20].

In our previous work [21], we introduced a process to convert object-oriented programs into strings and to analyze these strings to identify occurrences of design motifs. This previous work addressed two aspects of design motif 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 identification. Quality of the identification process includes performance (cost of the identification in processing time) and automation (automated versus manual process).

We build on our previous work to present and compare two approaches for design motif identification using automata simulation and bit-vector processing as well as to perform a comparison with previous approaches. Thus, we contribute a complete survey of the adaptation of the pattern matching algorithms used in bioinformatics to design motif identification, in which Section 2 summarizes related studies and highlights their drawbacks; Section 3 describes the pre-processing of our algorithms; Section 4 details our two algorithms; Section 5 presents the approximations that must be handled when searching for approximate occurrences; Section 6 details our implementation; Section 7 presents two case studies to compare the results with previous approaches in terms of performance, precision, and recall; and Section 8 concludes and introduces future work. The results of this work could be used as a basis for future work applying bioinformatics algorithms to motif identification.

## 2. Related work

Several approaches of design motif identification have been introduced in the literature. Most of these approaches use structural matching between micro-architectures and design motifs, with different algorithms being used: rule inference [5,22], queries [23,24], fuzzy reasoning nets [25], and constraint programming [6,26].

For example, Wuyts [5] described design motifs as Prolog predicates and system entities as facts. He applied a Prolog inference algorithm to unify predicates and facts and thus 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 [27]. Approaches based on constraint programming [6] also face a combinatorial complexity, although explanations [28] reduce this complexity through user-interactions [26].

Antoniol et al. introduced an approach to reduce the search space using metrics [7]. They designed a multi-stage filtering process to identify micro-architectures identical to design motifs. For each entity of a system, they computed some metrics, for example, the numbers of inheritance, association, and aggregation relationships, and they compared these values with expected values for the corresponding role in a design motif before applying constraint-based structural matching. They inferred expected metric values from design motif descriptions manually. The main limitation of their work was the assumption that the micro-architectures accurately reflect the design motifs, which is rare. Moreover, the theoretical quantification of roles, when possible, did not reduce the search space significantly.

Following [7], the second author and other collaborators improved on the two previous approaches by combining metric thresholds obtained using machine learning algorithms and explanation-based constraint programming [8]. Roles in design motifs were quantified empirically using P-MARt, a database of micro-architectures similar to design motifs manually identified in several systems. This quantification was used to remove from the search space entities which cannot participate in a design motif according to the empirical data. Explanation-based constraint programming was applied on the remaining entities to identify micro-architectures similar to design motifs. This approach showed promising results but suffers from a lack of data on manually-identified micro-architectures and, again, from the performance of explanation-based constraint programming.

Tsantalis et al. [9] recently introduced a novel approach to identify design motifs in object-oriented systems using similarity scoring. The authors converted the graphs representing (parts of) the structure of a motif and of a system into a set of matrices where rows and columns represent roles (or entities) and a value of 1 indicates a particular relation between roles and–or entities. They defined at least six different matrices to describe, for example, the association graph, the generalization graph, abstract classes, and the graph of similar method invocations. They introduced a similarity scoring between matrices to compute the similarity between a motif and a set of entities while considering the number of matrices in which roles are involved to normalize the score. This approach is interesting because it allows for fast computations. It compares to our approaches in which we convert the graph of a motifs into an automata (automata simulation) or into a string of tokens (bit-vector processing algorithm). However, in our approaches, we attempt to ease the description of approximations, while Tsantalis et al.'s constraint the possible approximations to few cases.

Moreover, we discovered some errors in the reported results.[1] For example in JHotDraw v5.1, the approach did not identify the `Observer` design motif where classes `Figure` and `FigureChangeListener` in package `CH.ifa.draw.framework` play the roles of `Subject` and `Observer` respectively. Still in JHotDraw v5.1, the approach reports `CommandButton` and `Command` in package `CH.ifa.draw.util` as `Context` and `State/Strategy`, respectively, in a `State` or `Strategy` design motif, while the `CommandButton` class in fact implements a `Command`-enabled button, encapsulating a given and unique `Command`, as confirmed by the system documentation.

It is important to identify similar but not identical occurrences to handle cases where a design motif has been adapted to better fit its context of use or the developers' needs. For example in Fig. 1, an excerpt from JHotDraw v5.1, the `CompositeFigure` class, which plays the role of `Composite`, is not in a composition relationship with class `Figure`, which plays the role of `Component`, and the class `AbstractFigure` has been inserted between these two classes. Most previous approaches, except Tsantalis et al.'s [9] and one by the second author and collaborators [29], could not identify

[1] Tsantalis et al. [9] kindly detail their results at http://java.uom.gr/~nikos/pattern-detection.html, last accessed on the 20/01/2009.

Author's personal copy



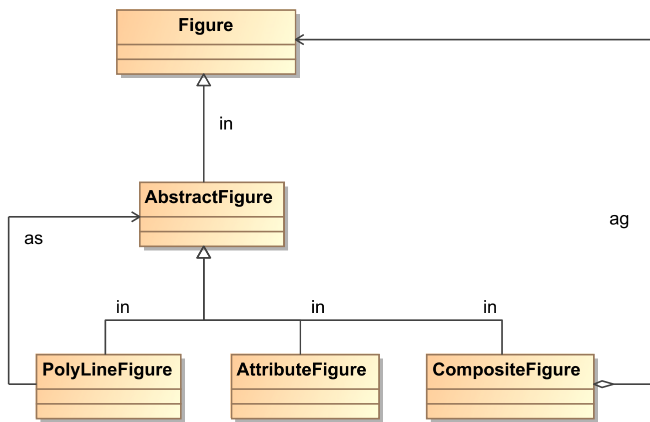



**Fig. 1.** Example of approximate implementation of the `Composite` in JHotDraw where `as`, `ag`, and `in` are association, aggregation, and inheritance relationships.

automatically this micro-architecture similar to the `Composite` design motif, shown in Fig. 2, although this information would be useful to maintainers to understand the design decisions behind the structure.

Therefore, the main limitations of previous work are threefold. Some previous approaches are slow [5,26] and thus cannot be used in industrial contexts. Other approaches only identify occurrences that match exactly their representations of the design motifs [5,7,9,23,24]. New descriptions can be added but a more interesting approach would be to start from the representation given in [4] and allow automated approximation to relieve the maintainers from describing all possible variants of a design motif. Tsantalis et al.'s approach is very promising but we follow a different research avenue by studying the use of two algorithms from bioinformatics that satisfy our needs for approximations and performance: automata simulation and bit-vector processing.

In addition to the previous limitations, structural approaches are inherently limited because they cannot take into account or assess the *intent* of the identified occurrences. Indeed, even if a set of classes are structurally similar to and have the same relationships than the classes describing a motif; this similarity could be accidental, *i.e.* does not necessarily reflect the developers' intent. The difference between structure and intent is the main reason for distinguishing between design patterns and design motifs [29]. To the best of our knowledge, only Kampffmeyer's work tried to address the intent of design patterns [30].

## 3. Pre-processing

Classical pattern matching algorithms are designed for strings and are interesting in bioinformatics, where sequences of tokens (nucleotides, amino-acids, and so on) are the main subject of study. Thus, before applying algorithms from bioinformatics to the identification of design motifs, we must convert systems and design motifs into strings to use classical approaches for our purpose. We must create the smallest possible strings because the complexity of the algorithms depends on their lengths.

This pre-processing of design motifs and systems into strings involves three steps. We first convert models of design motifs and systems into digraphs (Section 3.1). Then, we transform these digraphs into Eulerian digraphs (Section 3.2) to finally generate unique string representations (Section 3.3).

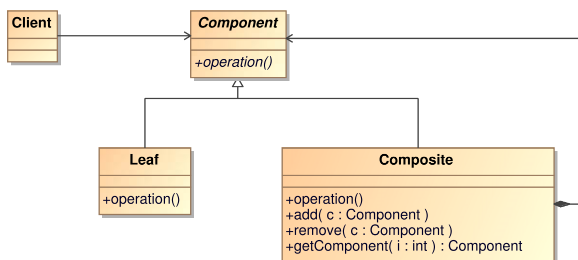
### 3.1. Design motif and system models

Fig. 2a shows the design motif of the `Composite` design pattern [4, p. 163] with a UML-like graphic representation. A typical object-oriented system is represented statically by its source code describing the entities and elements interacting to provide its functionality. Fig. 3a shows the model of a simple example system with the same UML-like representation. Both representations are very similar and we can model them using a single formalism.

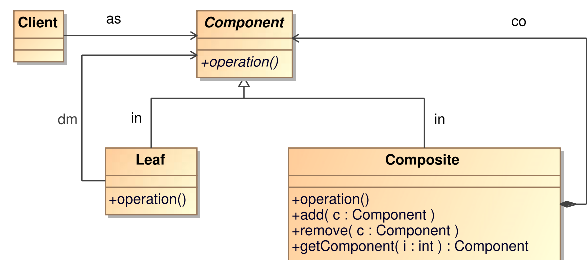
We use a same meta-model to describe entities and elements either forming a design motif or a system. The meta-model defines constituents whose instances are entities and elements combined together to describe models of design motifs and systems. We consider binary class relationships as elements: creation, specialization, implementation, use, association, aggregation, and composition relationships. We distinguish the association, aggregation, and composition relationships using rules on their implementations and part of their behaviour at runtime, as far as it can be inferred statically, as described in a previous work [31].

We also use ignorance relationships [32]. Indeed, the strings obtained from the design motifs specify what must be found in a system, while some motifs also specify what should not be found. For example, the entity playing the role of `Adaptee` in the `Adapter` design motif must ignore the entity `Adapter`, *i.e.* must not have any relationship with it, as shown in Fig. 4. The number of false positives is reduced by adding ignorance relationships in our design motifs strings.

A model of a design motif or a system is actually a graph whose vertices are entities and whose edges are elements connecting




(a) UML-like model.




(b) Eulerian model.

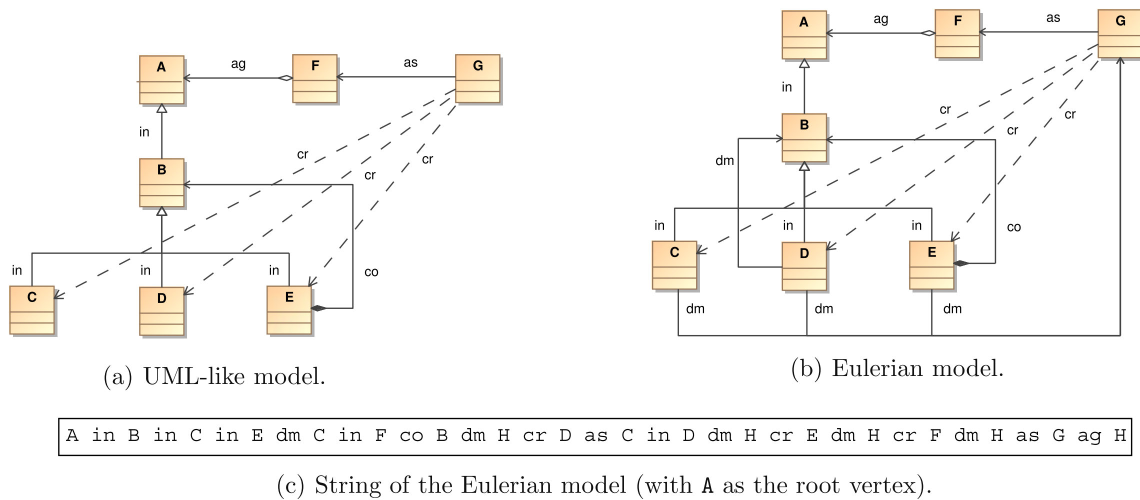
```
Component in Leaf dm Component in Composite co Leaf
```

(c) String of the Eulerian model (excluding the `Client` class which is not important to identify, with `Component` as the root vertex).

**Fig. 2.** Representations of the `Composite` design motif where `as`, `ag`, and `in` are as defined in Fig. 1 and `co` and `dm` are composition and dummy relationships.

Author's personal copy






(a) UML-like model.

(b) Eulerian model.

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

(c) String of the Eulerian model (with A as the root vertex).

**Fig. 3.** Representations of a simple example system where the `Composite` design motif is implemented by class `B` as component, class `E` as composite, and classes `C`, `D`, and `E` as leaves, and where the new relationship `cr` corresponds to creation (instantiation) relationships.

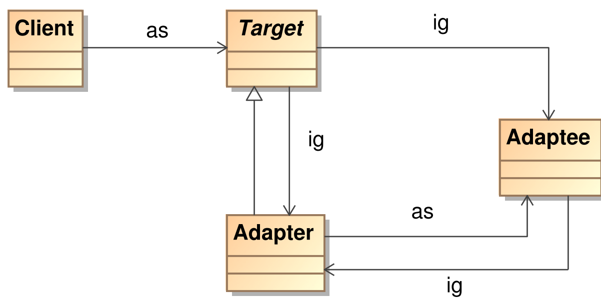



**Fig. 4.** `Adapter` design motif with ignorance relationships (`ig`).

entities. Thus, edges are directed because binary class relationships are directed. If more than one identical relationship (*e.g.* two associations) exists between the same two entities, we keep only one relationship of each type. We choose to keep one relationship because (1) a same relationship can be matched against any number of relationships in the design motif and (2) no design motif includes a class with two relationships of the same type pointing toward a same class. This is a simplifying assumption that leads certain motifs being not distinguishable. Yet, we never encountered such motifs so far.

We only consider entities and binary class relationships when defining our models of design motifs and obtaining the corresponding strings. This choice is motivated by the simplicity of converting our models into strings using the two steps described in the following subsections. Yet, it limits the type and amount of data available to perform the identification and, thus, may lead to false positive occurrences being identified. In particular, we do not use polymorphism and the presence of particular method invocations (such as the dual dispatch in the `Visitor` design pattern). However, our choice still allows us to identify meaningful occurrences, as shown in Section 7 and the use of other types of data, such as method invocations, is left to future work.

### 3.2. Eulerian digraphs of design motif and system models

With the previous meta-model, a model of a design motif or a system 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 a system into a Eulerian graph automatically and consistently by adding dummy edges between vertices with unequal in- and out-degrees. We use the transportation simplex [33] to obtain the number of dummy edges to be added among vertices and we consider those with greater in-degree as suppliers and those 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) and 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. Figs. 2b and 3b show respectively the Eulerian models of the `Composite` design motif and the simple example system.

### 3.3. Design motif and system strings

We compute the minimum Eulerian circuit using a dedicated algorithm to obtain unique strings of a design motif and a system model. The algorithm solves the directed Chinese Postman problem: the shortest tour of a graph which visits each edge at least once (see for example [34]). For a Eulerian graph, a Eulerian circuit is the optimal solution to the Chinese Postman problem.

Given a starting root vertex $v_{root}$, the solution of the Chinese Postman problem is a unique list of edges starting and ending with $v_{root}$ and containing all edges once. We iterate over the list of edges to build a unique string of design motif and system models with respect to the root vertices. Figs. 2c and 3c show strings of the `Composite` design motif and the example system with `Component` and `A` as root vertices respectively.

The resolution of the Chinese Postman problem requires choosing a root vertex for the traversal of the digraph. For example, the string 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 strings are identical when considered as circular sequences, *i.e.* the last token in the string is also its first token. Indeed, when first entering a vertex, our implementation of the Chinese Postman algorithm always chooses the same edge to leave a vertex, using the edge with the smallest weight and a lexicographic order among equally-weighted edges.

Author's personal copy



## 4. Identification

We summarise our approach of design motif identification intuitively as follows: we identify a micro-architecture similar to a design motif by simultaneously reading a system and a design motif and then recording the entities in the system that match those in the design motif in terms of structure and organization. Thus, design motif identification is an inherently combinatorial problem, requiring all possible combinations of entities through their elements to be compared against a motif.

We introduce two algorithms to retrieve occurrences of design motifs from the strings. The first identification algorithm uses automata simulation. The second one is an iterative bit-vector processing algorithm somewhat similar to bioinformatics approximate string matching algorithms.

### *4.1. Automata simulation*

Automata simulation is often used to search for occurrences of a regular expression in a text. The construction of a finite automaton from a regular expression can be automated [35,36]. There are two kinds of finite automata, nondeterministic finite automata (NFA) and deterministic finite automata (DFA), both of which can accept the same languages. It is always possible to construct a NFA from a DFA and vice versa [37]. Fig. 5 represents the NFA to identify occurrences of the Composite design motifs. A design motif identification NFA has $\lfloor m/2 \rfloor + 1$ states where $m$ represents the number of triplets in the design motif string. A triplet is composed of two classes connected by a relationship. A conditional transition is added for each triplet of the design motif string. The conditions are used to ensure that each occurrence of a role is replaced by the same system entity. Loop transitions are also added to every state except the last one to allow "holes" between triplets in the system string. This is similar to pushdown automata that can use and manipulate a stack containing data to determine the next transition.

When analyzing the simple example system represented in Fig. 3 to identify micro-architectures similar to the `Composite` design motif, the NFA reads the first triplet `A in B` and the states 0 and 1 activate. When state 1 activates, a part of a possible occurrence is created with `A` as `Component` and `B` as `Leaf`. State 0 is kept activated by the $\Sigma$ transition to allow other occurrences with different entities for the `Component` and `Leaf` roles. For example, a part of a new occurrence $\{\texttt{Component} = \texttt{B}, \texttt{Leaf} = \texttt{D}\}$ is created when reading the next triplet. The automaton finds the occurrence $\{\texttt{Component} = \texttt{B}, \texttt{Leaf} = \texttt{D}, \texttt{Composite} = \texttt{E}\}$ after passing through the states 0, 0, 1, 2, 3, and 4. If a transition contains a role already processed, it can only be taken if the symbol read for that role corresponds to the same previous entity. For example, the transition `Component in Composite` was followed when reading the triplet `B in E`.

In contrast to acceptors automata that test whether an input is accepted or not, the path taken during the simulation of the NFA is important. Each path to the final state is an occurrence of the motif. Fig. 6 presents the pseudo-code to identify a design motif using the NFA. The execution of the simulation with backtracking is done with the function **match** with the system string, the initial state, and an empty occurrence as parameters.

Because there can be a very large number of paths to manage, automata simulation is not really efficient. The use of a DFA can help because only one path can find every occurrence. For that rea-

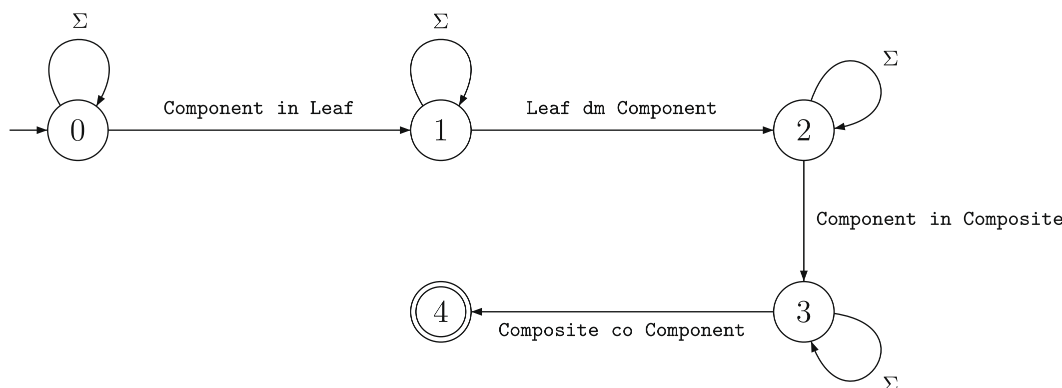



**Fig. 5.** NFA for the identification of occurrences of the `Composite` design motif. The $\Sigma$ symbol represents transitions that can be followed with any triplets of the input.

```
match(input, currentState, currentOccurrence)
occurrences := {}
IF currentState IS finalState
   ADD currentOccurrence IN occurrences
   RETURN occurrences
ENDIF
IF (NOT END OF input)
   FOR ALL transitions x IN currentState.transitions
      IF x.conditionRespected(input.getTriplet, currentOccurrence)
         UPDATE currentOccurrence
         ADD  match(input.removeTriplet,  x.destinationState,  currentOccurrence)
              IN occurrences
      ENDIF
   ENDFOR
ENDIF
RETURN occurrences
```

**Fig. 6.** Simplified pseudo-code to identify a design motif with a NFA.

Author's personal copy



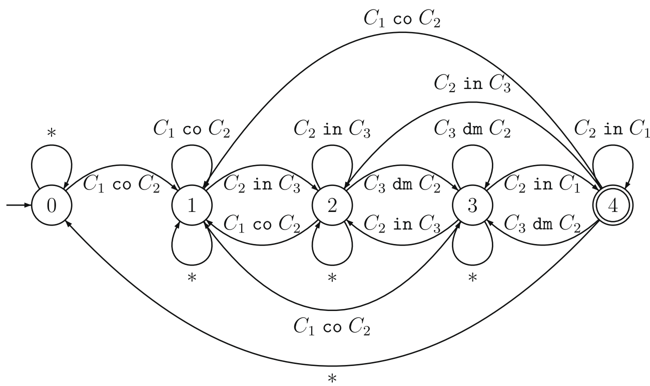



**Fig. 7.** DFA for the identification of occurrences of the `Composite` design motif. For a given state, the $*$ symbol represents the complementary of all output transitions for this state.

son, we convert our NFA into a DFA. Fig. 7 represents the DFA to identify occurrences of the `Composite` design motif. Each transition is also associated with a condition. For example, transition $C_2$ `in` $C_3$ between states 1 and 2 can only be taken if $C_2$ represents the entity already associated with the role `Component` (*i.e.* $C_2$ of the transition $C_1$ `co` $C_2$ between states 0 and 1).

Transitions labelled $*$ are only taken if no other transition can be followed. This corresponds to every other triplet not already specified by transitions. Transitions with a `dm` can be considered as $\epsilon$-transitions in NFA and as transitions without condition in DFA, *i.e.* they can be followed without moving forward in the input.

The automaton-based approach assumes that triplets always appear in the same order in the motif and system strings. The algorithm used to build the strings does indeed increase the likelihood that triplets appear in the same order, but in the cases where they do not, an automaton is not able to identify the motif and thus misses occurrences. This limitation can be solved by adding (1) an edge between certain pairs of states in the automaton and (2) the condition that an occurrence exists if and only if all states of the automaton have been reached.

Adding edges allows the automaton to identify occurrences in which the triplets are not in the expected order. Fig. 8 shows the NFA in Fig. 5 modified to handle unexpected orders in the triplets: the end nodes of any edges is now reachable from any state using the transition that was present in the original automaton. Thus,

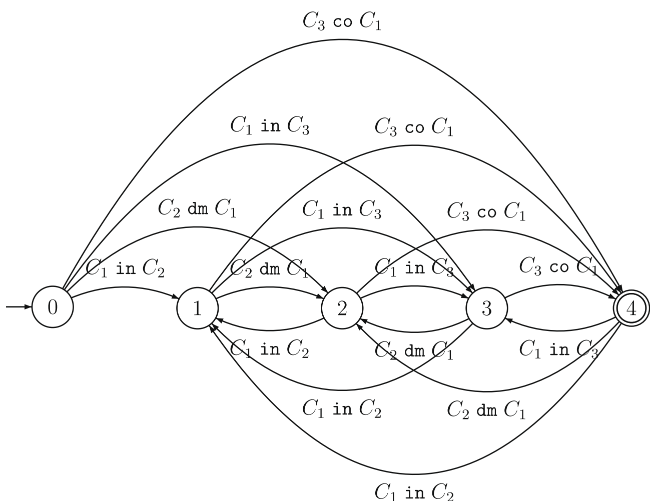



**Fig. 8.** NFA for the identification of occurrences of the `Composite` design motif in Fig. 5 modified to handle unexpected order.

from any node, it is possible to reach any other node to handle triplets in any order.

Adding the condition that all states must have been reached forces the automaton to cover all edges and, therefore, prevents the automaton to miss occurrences because of the triplets being in different order than expected.

However, these extra edges and condition further slow down the performance of the automata simulation. Moreover, automaton simulation does not handle elegantly ignorance relationships, because they cannot be expressed as edges among states. Ignorance relationships must be treated globally, either as conditions over the transitions or filters over the identified occurrences.

Therefore, we conclude that it is possible to apply pattern matching algorithms based on automata to the identification of design motifs. However, automata simulation has drawbacks (performance, ignorance) and, because of these drawbacks, we explore the possibilities of bit-vector processing algorithms.

### 4.2. Iterative bit-vector processing algorithm

The use of a bit-vector algorithm for design motif identification is interesting because such an algorithm can find a solution to a problem in a bounded number of vector operations, which is independent of the length of the system string. 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.

We developed a dedicated iterative bit-vector processing algorithm to find exact and approximate occurrences of a design motif in a system. Let a token be any symbol appearing in a string $x$ representing a system model. The characteristic vector of a token l associated with the string $x = x_1 \cdots x_m$, denoted by $\boldsymbol{l}$, is

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

For example, in the string of the system in Fig. 3c, the characteristic vector of class `G` is

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

while the vector for the inheritance relationship `in` is

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

Characteristic vectors are sequences of bits on which to apply standard bit operations: bit-wise logical "and", "or" operators, left and right shifts. Due to our construction of the strings, tokens composing a design motif always appear in the same order modulo a shift, so we consider our characteristic vectors as being circular. We define the right shift of a characteristic vector $\boldsymbol{l} = l_1 \cdots l_{m-1} l_m$ as $\rightarrow \boldsymbol{l} = l_m l_1 \cdots l_{m-1}$, the elements have been shifted to the right by one position, circularly. We similarly define the left shift of $l$ as $\leftarrow \boldsymbol{l} = l_2 \cdots l_m l_1$.

We use characteristic vectors to find the entities playing a role in a design motif. Our algorithm iteratively reads triplets of tokens in the design motif string, *e.g.* $t_{motif} = \{Role1, Relationship, Role2\}$, and identifies in the system string—using disjunctions and shifts—all possible triplets that matches $t_{motif}$, *i.e.* all possible pairs of entities between which the *Relationship* exists. It stores the entities identified as playing potentially *Role*1 and *Role*2 and then reads on the next triplet. While reading the next triplet and identifying the entities potentially playing each new role, the algorithm uses the sets of already identified entities to reduce the number of candidate entities, similarly to a unification algorithm. When the algorithm has read the last triplet, a set is associated with each role. If any one of these sets is empty, no occurrence of the motif has been

Author's personal copy



```
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 ∧ disjunctionX
         IF conjunctionY IS NOT NULL
            ADD Y IN before
         ENDIF
      ENDFOR
   ENDIF
ENDFOR
```

**Fig. 9.** Pseudo-code for the retrieval of the entities before and after a specific token.

identified, else as many occurrences as the size of the smallest set have been identified.

For example, to identify exact occurrences of the `Composite` design motif in Fig. 2 in the simple example system in Fig. 3, the algorithm reads the first triplet `Component in Leaf` and finds potential entities for the `Component` and `Leaf` roles in the string of the simple example system. It retrieves entities before and after the `in` token in the system string by applying bit-wise operations on its characteristic vectors. Fig. 9 shows the pseudo-code for the retrieval of the entities before and after a specific token.

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 then processes 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 disjunction between every entity characteristic vector and the disjunction between 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

$$\rightarrow\rightarrow \boldsymbol{B} = 0000100010001\underbrace{0\cdots0}_{18}$$

$$\rightarrow \boldsymbol{in} = 0010100010001\underbrace{0\cdots0}_{18}$$

$$(\rightarrow\rightarrow \boldsymbol{B}) \wedge (\rightarrow \boldsymbol{in}) = 0000100010001\underbrace{0\cdots0}_{18}$$

$$\boldsymbol{E} = \underbrace{00000000}_{8}1\underbrace{0\cdots0}_{15}1\underbrace{000000}_{6}$$

$$(\rightarrow\rightarrow \boldsymbol{B}) \wedge (\rightarrow \boldsymbol{in}) \wedge \boldsymbol{E} = \underbrace{00000000}_{8}1\underbrace{0\cdots0}_{22}.$$

and occurrence {`Component = B`, `Leaf = C`, `Composite = E`} is added to the list of occurrences 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 system string.

Table 1 shows the occurrences after the first, third, and fourth triplets have been processed sequentially. The second triplet is ignored because it corresponds to 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. For example, the `Composite` string could be read circularly beginning with the `Composite` token, so that the composition relation `co` would be treated first. Table 2 shows that this new ordering decreases the number of potential occurrences to 3 with respect to the original number of 10 in Table 1.

**Table 1**
Occurrences when beginning with the `in` relationship between `Component` and `Leaf`.

| Triplets | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| First (`in`) | | Third (`in`) | | | Fourth (`co`) | | | |
| Component | Leaf | Component | Leaf | Composite | Component | Leaf | Composite | |
| A | B | A | B | B | B | C | E | 10 |
| 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 | | | | |

Negative relationships are easily handled with the negation operator. For example, an ignorance relationship between the two classes `C` and `E` exists if:

$$((\rightarrow\rightarrow \boldsymbol{C}) \wedge \neg(\rightarrow\ \boldsymbol{in} \vee \rightarrow\ \boldsymbol{as} \vee \rightarrow\ \boldsymbol{ag} \vee \rightarrow\ \boldsymbol{co} \vee \cdots)) \wedge \boldsymbol{E} = \underbrace{0\cdots0}_{31}$$

## 5. Approximations

Design motif identification is an approximate unification problem. Approximation is necessary because, for example, a documented occurrence of the `Composite` design motif in JHotDraw v5.1 has a class inserted between those playing the roles of `Component` and `Composite`, as shown in Fig. 1 in Section 2. Thus, we include automatic and manual approximation mechanisms in our identification algorithms.

Maintainers can perform approximations manually by specifying which relationships should be relaxed. However, describing all possible approximations is not reliable because one approximation could be overlooked and because all these approximations are tedious and difficult to maintain. An automatic mechanism of

Author's personal copy



**Table 2**
Occurrences when beginning with the `co` relationship between `Composite` and `Component`.

| Triplets | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| First (co) | | Second (in) | | | Fourth (in) | | | |
| Composite | Component | Composite | Component | Leaf | Composite | Component | Leaf | |
| E | B | E | B | C | E | B | C | } 3 |
| | | E | B | D | E | B | D | |
| | | E | B | E | E | B | E | |

approximations must therefore be used to compute and explain identified micro-architectures to maintainers by stating explicitly what parts of a design motif are not exactly implemented. The approximately identified micro-architectures, once manually inspected by the maintainers, help in improving the code by applying corrections based on the design motifs.

In our model, where binary class relationships and classes are the main constituents, four types of approximations are possible:

- **Type 1 approximation** allows a relation between two entities to be different from expected. For example, an aggregation relationship could be replaced by a stronger relationship (composition) or a weaker one (association, use).
- **Type 2 approximation** allows the hierarchy of roles to be approximated: some entities could be inserted in or removed from the inheritance tree.
- **Type 3 approximation** handles cases where the kind of entities is not respected. Abstract entities could be played by concrete entities or vice-versa.
- **Type 4 approximation** manages cases where all roles in a design motif are not played by an entity in a system. For example, it is possible to find micro-architectures implementing the `Composite` design motif without a `Leaf`.

We limit possible approximations to the four types presented above. Therefore, for example, we do not take into account the case where a class serves as an intermediary in the association relationship between two classes. We disregard such an approximation because we believe that it would lead to too many false positives.

Other types of approximations are possible. For example, a client class could call a concrete method instead of an abstract method or references on objects could be obtained through a database instead of using a binary class relationship. However, approximations based on method types or field access are not possible in our algorithms because the system and motif models do not include such data and we leave for future work this extension of our model.

Fig. 1 includes four approximate occurrences of the `Composite` design motif. The occurrences

{`Component` = `Figure`,
`Composite` = `CompositeFigure`,
`Leaf` = `PolyLineFigure`}

and

{`Component` = `Figure`,
`Composite` = `CompositeFigure`,
`Leaf` = `AttributeFigure`}

are approximate occurrences of types 1 and 3. The composition relationship is replaced by an aggregation and the class `AbstractFigure` is inserted between the entities playing the roles of `Component` and `Composite`. Two other occurrences exist:

{`Component` = `AbstractFigure`,
`Composite` = `PolyLineFigure`,
`Leaf` = `AttributeFigure`}

and

{`Component` = `AbstractFigure`,
`Composite` = `PolyLineFigure`,
`Leaf` = `CompositeFigure`}

with type 1 approximation where the composition relationship is replaced by an association. An association from class `PolyLineFigure` to class `AbstractFigure` exists through the implicit call from the constructor of `PolyLineFigure` to the constructor of `AbstractFigure` [31]. The corresponding occurrences are therefore far approximations because `PolyLineFigure` does not *explicitly* declares a one-to-many composition with `AbstractFigure` and could be ignored by maintainers or automatically discarded by disallowing the approximation of compositions by associations.

Approximate occurrences are easily obtained with both automata simulation and bit-vector processing. With automata, type 1 and 2 approximations are obtained by adding transitions. For example, adding a transition $C_x$ `ag` $C_y$ where a transition $C_x$ `co` $C_y$ already exists allows the composition relationship to be replaced by an aggregation relationship. Adding transitions without condition allows type 2 approximations. Type 3 and 4 approximations are made possible by modifying the transition conditions, for example, by allowing the children or parents of an entity to play its role.

With bit-vector processing, type 1 approximation are obtained by using a disjunction between the characteristic vectors of the relaxed relationships and those of the expected relationships. For example, a disjunction between the characteristic vectors of the composition and aggregation relationships allows occurrences with either a composition or an aggregation relationship between two entities. All relationships may be relaxed or removed completely.

Type 2 approximate occurrences can be identified by adding the parents and children of an entity playing a role in a design motif as possible entities for that role. The children of an entity ***Parent*** are obtained by applying the equation $(\rightarrow\rightarrow \textbf{\textit{Parent}}) \wedge (\rightarrow \textbf{\textit{in}}) \wedge \textbf{\textit{Child}}$ (illustrated in Section 4) recursively on every ***Parent***. Similarly, the parents of an entity `Child` are obtained using the equation $(\leftarrow\leftarrow \textbf{\textit{Child}}) \wedge (\leftarrow \textbf{\textit{in}}) \wedge \textbf{\textit{Parent}}$.

Type 3 approximate occurrences can be identified by filtering entities for a specific role. For example, concrete classes could be removed if a specific role must be played by an abstract class. We need to add extra information to the system and motif strings describing whether an entity is abstract or not and whether a role must be played by an abstract or concrete entity. We can enforce that an abstract role is played by an abstract entity using a disjunction between the characteristic vectors representing all the abstract entities in a system and the characteristic vector of the entity that must be abstract.

To identify type 4 approximate occurrences, a role can be overlooked by removing the related class from the design motif model

Author's personal copy



and generating the string corresponding to this model. Type 4 approximations are necessary manual because only a maintainers may decide which role is essential or not for an occurrence of the motif to be of interest, in her context.

## 6. Tools

We integrated freely available tools to implement our algorithms to design motif identification. For performance and convenience, we break down our algorithms into two parts: preprocessing and identification through automata simulation or bit-vector processing.

### 6.1. Design motif and system models

We use the PADL meta-model [38] to describe design motifs and systems. The PADL meta-model defines all the constituents required to describe the static structure of design motifs and systems and part of their behaviour, including binary class relationships [31]. It is associated with several parsers to build models of systems from AOL, C++, and Java. It also includes a design motif repository containing several well-known design motifs such as `Abstract Factory`, `Composite`, `Facade`.

### 6.2. Eulerian graphs of design motif and system models

We iterate through the PADL models of a design motif and a system to identify the entities with unequal in-degree and out-degree using adjacency matrices. We then use a Java 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 system models, which thus become Eulerian digraphs.

### 6.3. Design motif and system strings

After transforming design motif and system models into Eulerian digraphs, we build strings using Thimbleby's implementation of an algorithm to solve the Chinese Postman problem [39]. This implementation uses several well-known algorithms for efficiency, such as Floyd–Warshall's shortest path and cycle cancelling.

### 6.4. Identification algorithms

We implemented the automata simulation algorithm and the iterative bit-vector processing algorithms in Java. We use a sparse vector representation for the bit-vector processing algorithm because our characteristic vectors can be long with most bits being 0-valued. This representation is backed up by a hash map and only the 1-valued bits are stored in the map to ensure space-efficiency. Our implementations are available upon request.

## 7. Case studies

In previous work, constraint programming showed promising results with respect to the quality of the identified micro-architectures, while metrics decreased identification time significantly. Similarity scoring showed the best performance. We present two case studies that (1) compare the results and times of our bit-vector processing algorithm with previous work and (2) that study its precision and recall.

[2] This implementation has been developed by Sung Ki-seok, So Young-seob, Choi Jin-min, Ju Hien-taek, Eom Soong-eun, and Lee Kyu-sung under the direction of Park Soon-dal.

### 7.1. General setup

#### 7.1.1. Precision and recall

Precision and recall are measures defined in information retrieval and are computed with the following modified formulas:

$$\text{precision} = \begin{cases} 100.00\% \text{ if} \\ \quad |\{\text{true occurrences}\}| \\ \qquad = |\{\text{identified occurrences}\}| = 0 \\ \text{else} \\ \quad \frac{|\{\text{true occurrences}\} \cap \{\text{identified occurrences}\}|}{|\{\text{identified occurrences}\}|} \end{cases}$$

$$\text{recall} = \begin{cases} 100.00\% \text{ if} \\ \quad |\{\text{true occurrences}\}| = |\{\text{identified occurrences}\}| = 0 \\ \text{else} \\ 0\% \text{ if} |\{\text{true occurrences}\}| = 0, |\{\text{identified occurrences}\}| > 0 \\ \text{else} \\ \quad \frac{|\{\text{true occurrences}\} \cap \{\text{identified occurrences}\}|}{|\{\text{true occurrences}\}|} \end{cases}$$

The formulas are slightly modified as in previous work [29] for cases where no true occurrence exists and our algorithm reports no occurrence because it would not be possible to compute precision and recall (division by zero). Yet, our algorithm does well not to produce false positives and therefore its precision and recall are 100.00%. If there is no true occurrence and our algorithm identifies false positives, then both its precision and recall are 0%.

#### 7.1.2. Time, memory, and automata simulation

All computations were performed on an AMD Athlon 64 bits at 2 GHz. We retrieved identification times using the Eclipse-based profiling tool, Eclipse Profiler [40]. We performed all computations three times and report averages. We do not provide measures of the memory performance because, in addition to the strings, we also maintain in memory different models of the design motifs before constructing the strings and did not want to impede the time performances by forcing the garbage collector of the Java virtual machine to run. However, all computations have been performed using a maximum heap size of 1024 M.

The first part of our algorithm consists in building strings of the systems to obtain their characteristic vectors, using the transportation simplex and solving the Chinese postman problem. This computation is only performed once.

We do not include the automata simulation algorithm in the following case studies because the quality of the identified micro-architectures and the time required to find them could not compare favorably with other approaches. Table 3 shows that the computation times for the identification of the `Composite` design motif in two small systems, Juzzle and JHotDraw, are long and, thus, that automata simulation cannot compare favorably with any other approaches, in particular Tsantalis et al.'s approach, which runs under 1 s.

#### 7.1.3. Definition of an occurrence

The definition of *one* occurrence of a design motif is important because it dramatically changes the numbers of identified occurrences and the computed precision and recall. For example, in the case of the `Composite` design motif and the simple example

**Table 3**
Computation times for the identification of the `Composite` design motif in two small systems, in seconds, by our automata simulation algorithm.

| Automata Simulation | Composite | |
|---|---|---|
| | Juzzle | JHotDraw |
| Time | 287 s | 2055 s |

Author's personal copy



system in Figs. 2 and 3, there could be one or three occurrences depending on the chosen definition:

$$\{\texttt{Component} = \texttt{B}, \texttt{Composite} = \texttt{E}, \texttt{Leaf} = \{\texttt{C}, \texttt{D}, \texttt{E}\}\}$$

or

$$\{\texttt{Component} = \texttt{B}, \texttt{Composite} = \texttt{E}, \texttt{Leaf} = \texttt{C}\} \wedge \{\texttt{Component} = \texttt{B}, \texttt{Composite} = \texttt{E}, \texttt{Leaf} = \texttt{D}\} \wedge \{\texttt{Component} = \texttt{B}, \texttt{Composite} = \texttt{E}, \texttt{Leaf} = \texttt{E}\}.$$

The design motifs originally introduced in [4] describe abstractions; most roles could be fulfilled by more than one class in a system. However, whether an occurrence of the `Composite` design motif includes several leaves or only one leaf is an important question because its answer varies between approaches and impacts precision and recall. For example, Tsantalis et al. report occurrences of the `Composite` design motif without the `Leaf` role:

$$\{\texttt{Component} = \texttt{B}, \texttt{Composite} = \texttt{E}\}.$$

The choice of this omitted role is sound but omitting roles cannot be performed for all motifs systematically. Consequently, we consider that micro-architectures similar to the `Composite` design motif with three leaves counts as three occurrences, as in other previous work. We use this definition of an occurrence to be able to compare different approaches because we can adapt our algorithms to count as previous work but cannot easily adapt previous work to count as our own.

#### 7.1.4. Choice of the design motifs

We report the number of occurrences and computation times for five different design motifs on seven systems. The design motifs are `Abstract Factory`, `Composite`, `Decorator`, `Observer`, `State/Strategy`. We have applied our algorithms on `Adapter`, `Bridge`, `Command` but cannot report their results in this article for the sake of clarity, we refer the interested reader to the companion Web site[3] that presents all the results in one place. We distinguish the motifs of the `Composite` and `Decorator` by enforcing the presence of a least one subclass of the class playing the role of `Decorator`, *i.e.* at least one concrete decorator. We cannot distinguish between the `State` and `Strategy` design motifs because they are identical [41,42] and, thus, only a maintainer can decide if an occurrence of one of these motifs actually implements either of the corresponding pattern, based on external clues, such as names, runtime behaviour, context.

We also compare different approaches using the `Abstract Factory` and `Composite` design patterns, because these are well-known design patterns with different intents and motifs, shown in Figs. 2 and 10a. The strings used for the identification are generated directly from the original class diagrams presented in [4] and modelled with PADL. No changes were performed to improve precision or recall.

[3] The companion Web site of this article is available at http://www.ptidej.net/research/ptidej/epi/.

We choose these two motifs to compare approaches because they highlight different properties of existing algorithms and because previous work also used these motifs. The `Composite` design motif includes a composition relationship and two inheritance relationships while the `Abstract Factory` design motif uses association, creation, and inheritance relationships. Since in a given system more association relationships exist than composition relationship and the `Abstract Factory` has a less constraining structure with weaker relationships than the `Composite`, we expect to identify more occurrences for the `Abstract Factory` than for the `Composite` design motif.

#### 7.1.5. Choice of the approximations

Theoretically, any combination of any number of approximations could lead to the identification of true occurrences of some design motif. Therefore, all possible combinations of the four approximations should be considered in our studies. However, in our previous work, *e.g.* [21,26], and in the following studies, we find that two types of approximations are enough to identify all the occurrences of a design motif. This finding is confirmed by the study of the approaches and results of other authors, in particular Tsantalis et al.'s. Moreover, with design motifs where relationships are important, such as `Composite` and `Abstract Factory`, approximations of types 1 and 2 are sufficient. We will only use the approximation of type 4 when comparing our algorithm with similarity scoring because it does not report all roles in its results.

For type 1 approximations, we chose the aggregation relationship as the weakest relationship, *i.e.* we did not replace aggregation relationships with weaker ones (association, use). This choice is sensible because replacing composition relationships by association or use relationships would return occurrences that are semantically too distant from the original motif. This choice results in the first two lines of the `Abstract Factory` Tables 5 and 10 being identical because this motif only includes an association relationship which is therefore never approximated.

Type 3 approximations are not considered because they would not provide more valid occurrences of the design motifs but artificially increase the numbers of false positive occurrences.

We do not report in the following approximation of type 4 (deletion) because this approximation provide occurrences that do not conform to our definition of an occurrence. Indeed, with the type 4 approximation, our approach returns occurrences where, for example in the `Composite`, no class plays the role of `Leaf`. Occurrences obtained with this approximation are discussed later when comparing our approach with Tsantalis et al.'s scoring algorithm.

#### 7.1.6. Choice of the systems

Table 4 shows the systems used in the following case studies. These systems range in size from 57 classes to 742 classes, which represent small to medium systems. They cover a wide range of domains, from unit testing to games to allow generalization. They are all open-source and easily accessible for further comparisons. Table 4 also presents the average computation times of the strings, performed once per system, on our test computer. The string of Gantt Project is longer than the string of Azureus, even though Azureus has a larger number of entities, because more relationships exist in Gantt Project and more dummy relationships are added by the transportation simplex. (The data on some of these systems has been updated in comparison to our previous work [21,29] based on recent improvements and studies by other authors [9], including numbers of occurrences and performances.)

Author's personal copy



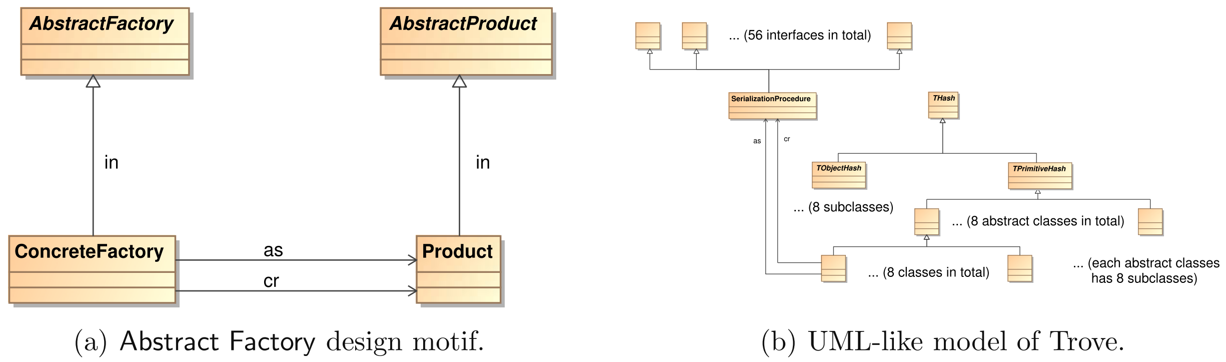



**Fig. 10.** `Abstract Factory` design motif and subset of Trove's design.

**Table 4**
Names, sizes, descriptions of the analyzed systems in the second case study. The last column shows the computation times in seconds for building the models.

| Systems | Sizes | Descriptions | Strings | |
|---|---|---|---|---|
| | | | Times (s) | Lengths (tokens) |
| Juzzle v0.5 | 57 | A puzzle game | 1 | 4817 |
| JUnit v3.7 | 209 | Unit testing framework | 59 | 19,467 |
| JHotDraw v5.1 | 261 | Framework for technical and structured graphics | 69 | 10,161 |
| Trove v1.1b5 | 296 | High performance collections for Java | 25 | 20,404 |
| QuickUML 2001 | 338 | A UML class-diagram graphic editor | 213 | 68,671 |
| Gantt Project v1.10.2 | 503 | Tasks management software | 752 | 111,942 |
| Azureus v2.3.0.6 | 742 | Peer-to-peer bit-torrent client | 1995 | 90,847 |

**Table 5**
Numbers of identified occurrences of the design motifs by our bit-vector processing algorithm (BV).

| BV | Juzzle | JUnit | JHotDraw | Trove | QuickUML | Gantt project | Azureus |
|---|---|---|---|---|---|---|---|
| | `Abstract Factory` | | | | | | |
| No approximation | 0 | 26 | 81 | 3192 | 6 | 63 | 136 |
| Relationships (1) | 0 | 26 | 81 | 3192 | 6 | 63 | 136 |
| Insertion (2) | 0 | 55 | 250 | 12,507 | 11 | 71 | 249 |
| 1 and 2 | 0 | 55 | 250 | 12,507 | 11 | 71 | 249 |
| | `Composite` | | | | | | |
| No approximation | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| Relationships (1) | 0 | 6 | 12 | 8 | 2 | 2 | 11 |
| Insertion (2) | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 and 2 | 0 | 39 | 103 | 64 | 16 | 2 | 41 |
| | `Decorator` | | | | | | |
| No approximation | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| Relationships (1) | 0 | 0 | 12 | 64 | 0 | 0 | 0 |
| Insertion (2) | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 and 2 | 0 | 0 | 114 | 512 | 0 | 0 | 0 |
| | `Observer` | | | | | | |
| No approximation | 0 | 1 | 17 | 0 | 10 | 15 | 11 |
| Relationships (1) | 0 | 1 | 17 | 0 | 10 | 15 | 11 |
| Insertion (2) | 0 | 4 | 37 | 13 | 10 | 15 | 11 |
| 1 and 2 | 0 | 4 | 37 | 13 | 10 | 15 | 11 |
| | `State/Strategy` | | | | | | |
| No approximation | 2 | 20 | 107 | 29 | 40 | 88 | 121 |
| Relationships (1) | 2 | 20 | 107 | 29 | 40 | 88 | 121 |
| Insertion (2) | 2 | 120 | 440 | 505 | 62 | 94 | 166 |
| 1 and 2 | 2 | 120 | 440 | 505 | 62 | 94 | 166 |

### *7.2. Study of the bit-vector processing algorithm*

#### *7.2.1. Objectives and design*

Most previous approaches give good results on small and medium size systems but are not usable on large systems, for example [5] applied his approach on 170 subclasses of class `VisualPart` in Smalltalk and the `Composite` design motif but reported a slow execution time, performance decreasing with the complexity of the queries. The objective of this first case study is to show that our bit-vector processing algorithm is scalable while keeping the numbers of occurrences tractable, thus showing the quality of the identification process.

#### *7.2.2. Identified occurrences*

Table 5 reports the numbers of occurrences found in each system for each motifs depending on the approximations made. As ex-

Author's personal copy



pected, when more types of approximations are allowed, more occurrences are identified by our algorithm. However, the numbers of occurrences remain tractable and, depending on the context and on her knowledge of a system, a maintainer could only focus on the types of approximations that are more likely to provide interesting occurrences. For example, for QuickUML 2001 and the `Composite` design motif, the occurrences identified with type 1 and 2 approximations represent the 16 true micro-architectures existing in the system, as further explained in Section 7.3.

Table 5 shows that design motifs are not implemented "by the book" but adapted to the particular context in which they are used, as highlighted by the first line of the table, where no exact occurrences are found for any of the studied systems.

Trove is interesting because of its particular design. This system is designed with two separate hierarchy trees. The first hierarchy tree includes 56 interfaces that are implemented by the single interface `SerializationProcedure` while the second one has for root the `THash` interface with 58 subclasses that reference `SerializationProcedure`, as shown in Fig. 10b. For the `Abstract Factory` design motif, this design leads to a large number of occurrences because interface `SerializationProcedure` plays the role of `Product` while any of its 56 super-interfaces can play the role of `Abstract Product`. Interface `THash` or any of its direct children plays the role of `Abstract Factory` while any of its descendant can play the role of `Concrete Factory`. Thus, leading to an large number of occurrences.

The number of occurrences for the `Abstract Factory` design motifs show that some approximations do not necessarily provide more occurrences: the two first lines of the first table in Table 5 are identical because the motif does not include the use of composition relationships that could be replaced by aggregation relationships. The two last lines of the same table are identical for the same reason. Only the type 2 approximation allows identifying additional occurrences, which means that some implementation of the motif include intermediate entities in their hierarchies.

#### 7.2.3. Precision and recall

Table 6 shows the number of true occurrences (TO), the number of identified occurrences by the bit-vector processing algorithm and the associated precision and recall. The true occurrences used to compute precision and recall were obtained by analyzing the systems manually to identify micro-architectures similar to the design motifs. Thus, true occurrences include exact and approximate occurrences of the motifs. This manual analysis was performed by two teams belonging to two different universities in the USA and Italy [43] and the results were put together in an XML database [8].

Our algorithm favors recall over precision, which is consistent with our objective of identifying all micro-architectures similar to some design motifs to recover lost design decisions. The somewhat low precision can be explained because of the number of performed approximations. We compare precision and recall for some of the systems with previous approaches in the next case study.

**Table 6**
Precision and recall of approximate identified occurrences of the two design motifs. TO means true occurrences (exact and approximate), BV stands for bit-vector processing algorithm.

| Juzzle | | JHotDraw | | QuickUML | |
|---|---|---|---|---|---|
| TO | BV | TO | BV | TO | BV |
| `Abstract Factory` | | | | | |
| 0 | 0 | 0 | 250 | 8 | 11 |
| Precision | 100.00% | Precision | 0.00% | Precision | 72.73% |
| Recall | 100.00% | Recall | 0.00% | Recall | 100.00% |
| `Composite` | | | | | |
| 0 | 0 | 52 | 103 | 16 | 16 |
| Precision | 100.00% | Precision | 50.48% | Precision | 100.00% |
| Recall | 100.00% | Recall | 100.00% | Recall | 100.00% |

#### 7.2.4. Performance

Table 7 presents the identification times in seconds of the algorithm. We consider the worst-case scenario for our algorithm where the user is interested in the occurrences identified with approximations of types 1 and 2, which include exact as well as approximate occurrences. In general, the performance of the bit-vector processing algorithm are reasonable, in particular for uses in industrial contexts.

As expected, the number of allowed approximations impacts performance. Certain types of approximation lead to more computation. For the `Composite` design motif, the type 1 approximation decreases performances, as seen in rows labeled "Relationships (1)" and "1 and 2". The reason for this decrease is the composition relationship between roles `Composite` and `Component` and the large number of aggregation relationships by which it can be replaced.

As other example, for the `Abstract Factory` design motif, the type 2 approximation decreases dramatically the performances for JHotDraw and Trove because these two systems have particular designs, as explained previously for Trove, leading to a large number of classes playing the roles of `Abstract Factory` and `Concrete Product`.

The dashed lines in Figs. 12 and 11 show that there is a general trend for computation times to depend on the string lengths. This dependence was expected since the bit-vector processing algorithms iterates over the strings to process triplets. For the `Composite` design motif, computation times tend to decrease with the numbers of occurrences while for `Abstract Factory` they do increase as expected. This trend is due to the composition relationship in `Composite`, which reduces the numbers of possible occurrences, as explained in Section 4.2 by Tables 1 and 2.

### 7.3. Comparison with previous approaches

#### 7.3.1. Objectives and design

This second case study has for objective to compare our approach with previous work to show the quality of the identified micro-architectures. We compare our bit-vector algorithm to DeMIMA [29], which uses explanation-based constraint programming, another approach using metric-enhanced explanation-based constraint programming [8], and Design Pattern Detection Tool, a tool implementing a similarity scoring algorithm [9].

We present results in terms of numbers of occurrences and computations times, and precision and recall. Results are presented in two steps, first we compare our approach with approaches that uses our definition of an occurrence, second with Tsantalis et al. approach, which uses a different definition. In this case study, we apply the algorithms on Juzzle v0.5, JHotDraw v5.1, and QuickUML 2001, presented in Table 4, because all approaches were run on these systems.

#### 7.3.2. Comparison with constraint-based approaches

Table 8 presents the numbers of occurrences of the `Abstract Factory` and `Composite` design motifs identified by three approaches and their precision and recall: constraint-programming (CP), constraint-programming with metrics (CP + M), and our bit-vector processing algorithm (BV) on the three systems.

The numbers[4] of approximate occurrences identified by the constraint-based approaches are high because these approaches

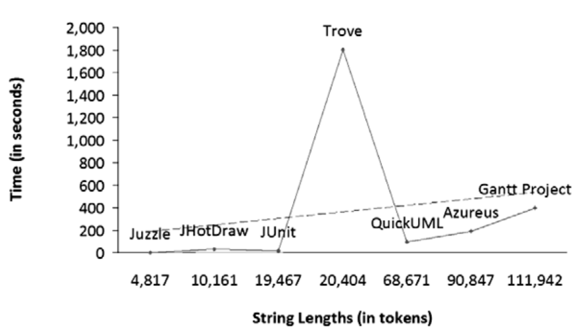
[4] The numbers of occurrences are given using the definition presented in Section 7.1.3, which differs from the definition used in our previous work [29].

Author's personal copy

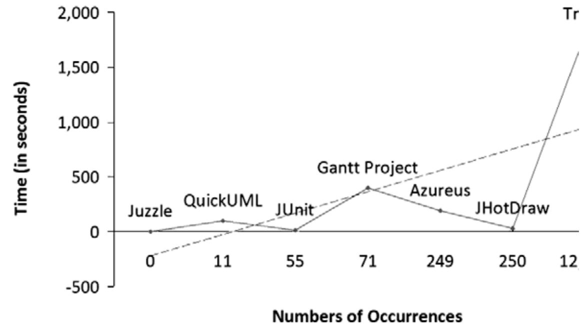


**Table 7**
Identification times of the design motifs, in seconds, by our bit-vector processing algorithm (BV).

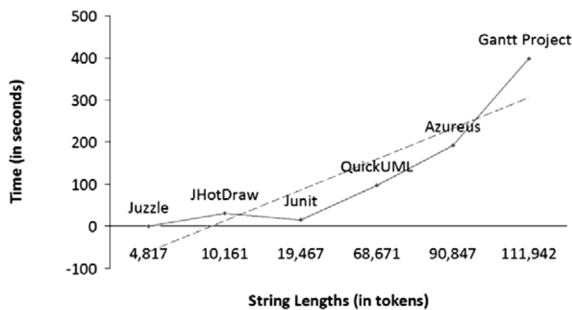
| BV | Juzzle (s) | JUnit (s) | JHotDraw (s) | Trove (s) | QuickUML (s) | Gantt project (s) | Azureus (s) |
|---|---|---|---|---|---|---|---|
| | `Abstract Factory` | | | | | | |
| No approximation | 0 | 15 | 6 | 15 | 95 | 277 | 145 |
| Relationships (1) | 0 | 15 | 6 | 16 | 95 | 331 | 147 |
| Insertion (2) | 0 | 15 | 31 | 1784 | 95 | 316 | 183 |
| 1 and 2 | 0 | 15 | 32 | 1802 | 97 | 398 | 192 |
| | `Composite` | | | | | | |
| No approximation | 0 | 3 | 1 | 5 | 22 | 76 | 45 |
| Relationships (1) | 0 | 4 | 1 | 5 | 28 | 105 | 53 |
| Insertion (2) | 0 | 3 | 1 | 5 | 23 | 82 | 46 |
| 1 and 2 | 0 | 4 | 3 | 20 | 28 | 157 | 53 |
| | `Decorator` | | | | | | |
| No approximation | 0 | 1 | 0 | 2 | 22 | 77 | 17 |
| Relationships (1) | 1 | 2 | 0 | 2 | 31 | 103 | 21 |
| Insertion (2) | 0 | 1 | 0 | 2 | 22 | 77 | 17 |
| 1 and 2 | 1 | 2 | 3 | 90 | 31 | 103 | 21 |
| | `Observer` | | | | | | |
| No approximation | 6 | 68 | 19 | 89 | 429 | 681 | 418 |
| Relationships (1) | 6 | 68 | 19 | 90 | 430 | 684 | 422 |
| Insertion (2) | 5 | 78 | 148 | 991 | 460 | 723 | 614 |
| 1 and 2 | 6 | 78 | 147 | 992 | 440 | 715 | 512 |
| | `State/Strategy` | | | | | | |
| No approximation | 4 | 13 | 12 | 19 | 91 | 277 | 114 |
| Relationships (1) | 3 | 13 | 12 | 19 | 91 | 276 | 114 |
| Insertion (2) | 3 | 46 | 59 | 177 | 126 | 311 | 145 |
| 1 and 2 | 3 | 46 | 59 | 184 | 126 | 313 | 145 |



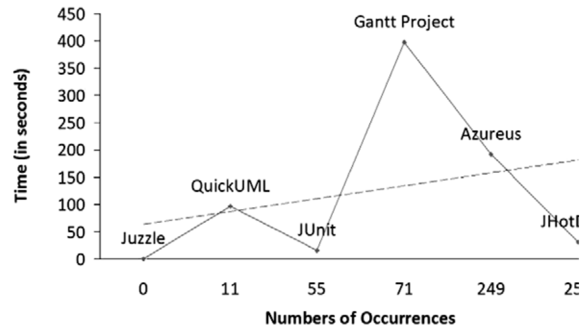

(a) String lengths vs. Time.




(b) Number of occurrences vs. Time.




(c) String lengths vs. Time without outlier TROVE.




(d) Numbers of occurrences vs. Time without outlier TROVE.

**Fig. 11.** Study of `Abstract Factory` computation times.

perform more automatic approximations than the bit-vector processing algorithms. Indeed, when no occurrence is found, constraints are relaxed/removed until there is no more allowed constraint to relax (or to remove). In the current implementation,

Author's personal copy



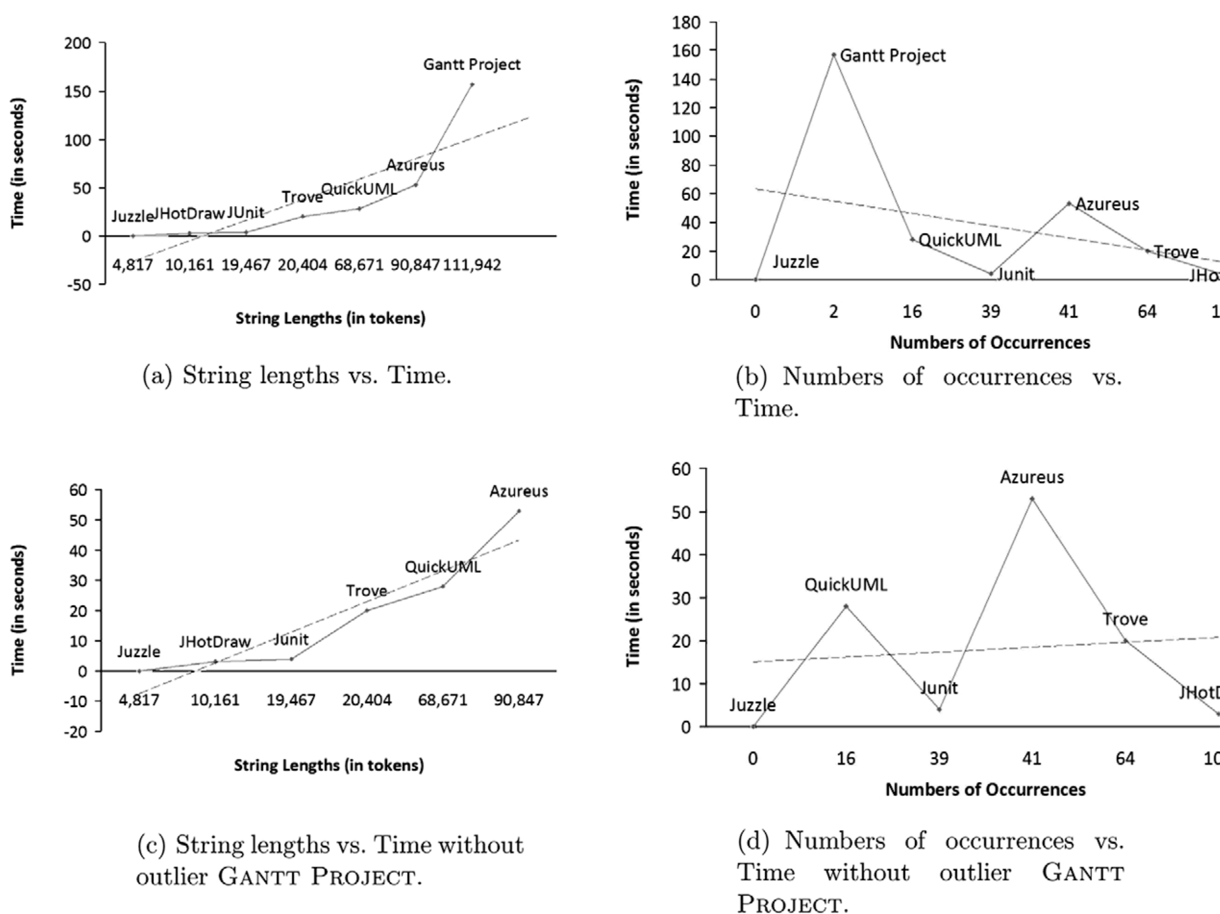



**Fig. 12.** Study of `Composite` computation times.

**Table 8**
Numbers of exact and approximate identified occurrences of the two design motifs as well as precision and recall, a "–" symbol means that we were not able to obtain the results. TO means true occurrences (exact and approximate), CP constraint programming, CP + M constraint programming with metrics, and BV bit-vector processing algorithm.

| Juzzle | | | | JHotDraw | | | | QuickUML | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| TO | CP | CP + M | BV | TO | CP | CP + M | BV | TO | CP | CP + M | BV |
| `Abstract Factory` | | | | | | | | | | | |
| 0 | 375 | 0 | 0 | 0 | 18,461 | 5235 | 250 | 8 | 5546 | 1159 | 11 |
| Precision | 0.00% | 100.00% | 100.00% | Precision | 0.00% | 0.00% | 0.00% | Precision | 0.14% | 0.69% | 72.73% |
| Recall | 0.00% | 100.00% | 100.00% | Recall | 0.00% | 0.00% | 0.00% | Recall | 100.00% | 100.00% | 100.00% |
| `Composite` | | | | | | | | | | | |
| 0 | 0 | 0 | 0 | 52 | 196 | 115 | 103 | 16 | 122 | 84 | 16 |
| Precision | 100.00% | 100.00% | 100.00% | Precision | 26.53% | 45.21% | 50.48% | Precision | 13.11% | 19.05% | 100.00% |
| Recall | 100.00% | 0.00% | 100.00% | Recall | 100.00% | 100.00% | 100.00% | Recall | 100.00% | 100.00% | 100.00% |

for example, binary class relationships are relaxed from composition to aggregation to association. Use relationships (or the lack of relationships) are not allowed.

The number of occurrences identified by the bit-vector processing algorithm is tractable as explained in the previous section. Moreover, the occurrences can be sorted by their distance from the design motif.

The CP and CP + M approaches have 100% recall by construction [8,29] thanks to their extensive approximations. Due to the greater numbers of occurrences identified by the CP and CP + M approaches, their precision is lower than that of BV.

An example of a "true" occurrence of the `Composite` design motif identified in JHotDraw is given below. This occurrence has been identified using type 1 and 2 approximation. It is a true occurrence as confirmed by the documentation of the system.

$$\{\texttt{Component} = \texttt{Figure},\ \texttt{Composite} = \texttt{CompositeFigure},\ \texttt{Leaf} = \texttt{ConnectionFigure}\}$$

An example of a "false" occurrence of the same motif in the same system is given below. This occurrence has been identified using a type 1 approximation. It exists because the class `DecoratorFigure` aggregates instances of `FigureChangeListener` (through its superclass `AbstractFigure` and methods such as `public void figureInvalidated(FigureChangeEvent)`) as part of its role of

Author's personal copy



**Table 9**
Identification times of the design motifs, in seconds. CP stands for constraint programming, CP + M for constraint programming with metrics, and BV for bit-vector processing algorithm.

| Juzzle | | | JHotDraw | | | QuickUML | | |
|---|---|---|---|---|---|---|---|---|
| CP | CP + M | BV | CP | CP + M | BV | CP | CP + M | BV |
| `Abstract Factory` | | | | | | | | |
| 813 s | 2 s | 0 s | 165,754 s | 37,922 s | 32 s | 104,258 s | 20,320 s | 97 s |
| `Composite` | | | | | | | | |
| 47 s | 3 s | 0 s | 907 s | 625 s | 3 s | 1688 s | 1164 s | 28 s |

`Subject` in the `Observer` motif and because both classes `ConnectionFigure` and `DecoratorFigure` implements the `FigureChangeListener` interface.

$$\{\texttt{Component} = \texttt{FigureChangeListener}, \texttt{Composite} = \texttt{DecoratorFigure}, \texttt{Leaf} = \texttt{ConnectionFigure}\}$$

Table 9 presents the identification times in seconds. The constraint programming approaches are slower. We explain these results by the extent of the performed approximations and the use of a generic solving algorithm. The performance of the bit-vector processing algorithm is adequate for daily use.

#### 7.3.3. Comparison with scoring algorithm

Tables 10 and 11 report the numbers of occurrences identified by our algorithm following the definition of an occurrence of Tsantalis et al. and the compared precision and recall.

The number of occurrences identified by the scoring algorithm for `Composite` is smaller than that of all other approaches because the scoring algorithm does not report the `Leaf` role and does not allow as many approximations as our algorithm. For `Abstract Factory`, both definitions of an occurrence return the same results because there is no reason to omit a particular role in this motif.

The similarity scoring algorithm, unlike our algorithm, cannot identify *all and every* motifs. For example, it cannot identify the `Abstract Factory` design motif as indicated by the "–" in the table. Similarly, our algorithm would have difficulty identifying the `Singleton` motif, because its structure involves only one class and no relationship.

The similarity scoring algorithm is faster than other approaches with times under 1 s for all systems. The bit-vector approach compares favorably with the scoring algorithm even though it is slightly slower but it has generally a better precision and recall. Comparison with other approaches is out of the scope of this article and left as future work.

## 8. Conclusion, discussion, and future work

We presented an adaptation of two classical pattern matching algorithms to the software maintenance problem of design motif identification: automata simulation and bit-vector processing. We detailed the conversion of the problem of design motif identification in a problem of approximate string matching.

We implemented automata simulation and bit-vector processing algorithms and studied their performance. We showed that automata simulation cannot compare favorably with previous approaches in term of performance. We also detailed the approximations required to solve the problem adequately and their introduction in automata simulation and bit-vector processing algorithms. We then studied the performance, precision, and recall of bit-vector processing using several small-to-medium systems and different type of approximations on two well-known yet different design motifs, `Composite` and `Abstract Factory`. We showed that bit-vector processing has interesting performance while providing tractable numbers of occurrences.

**Table 10**
Numbers of identified occurrences of the two design motifs when considering the same definition of an occurrence as Tsantalis et al. BV stands for bit-vector processing algorithm.

| BV | Juzzle | JUnit | JHotDraw | Trove | QuickUML | Gantt project | Azureus |
|---|---|---|---|---|---|---|---|
| | `Abstract Factory` | | | | | | |
| No approximation | 0 | 26 | 81 | 3192 | 6 | 63 | 136 |
| Deletion (4) | 0 | 26 | 81 | 3192 | 6 | 63 | 136 |
| Relationships (1) and 4 | 0 | 26 | 81 | 3192 | 6 | 63 | 136 |
| Insertion (2) and 4 | 0 | 55 | 250 | 12,507 | 11 | 71 | 194 |
| 1 and 2 and 4 | 0 | 55 | 250 | 12,507 | 11 | 71 | 194 |
| | `Composite` | | | | | | |
| No approximation | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| Deletion (4) | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| Relationships (1) and 4 | 0 | 2 | 2 | 7 | 1 | 1 | 2 |
| Insertion (2) and 4 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1, 2, and 4 | 0 | 3 | 8 | 7 | 2 | 1 | 2 |

**Table 11**
Precision and recall of approximate identified occurrences of the two design motifs when considering the same definition of an occurrence as Tsantalis et al., a "–" symbol means that we were not able to obtain the results. TO means true occurrences (exact and approximate), SC scoring algorithm, and BV bit-vector processing algorithm.

| Juzzle | | | JHotDraw | | | QuickUML | | |
|---|---|---|---|---|---|---|---|---|
| TO | SC | BV | TO | SC | BV | TO | SC | BV |
| `Abstract Factory` | | | | | | | | |
| 0 | – | 0 | 0 | – | 250 | 8 | – | 11 |
| Precision | – | 100.00% | Precision | – | 0.00% | Precision | – | 72.73% |
| Recall | – | 100.00% | Recall | – | 100.00% | Recall | – | 100.00% |
| `Composite` | | | | | | | | |
| 0 | 0 | 0 | 2 | 1 | 8 | 2 | 1 | 2 |
| Precision | 100.00% | 100.00% | Precision | 100.00% | 25.00% | Precision | 100.00% | 100.00% |
| Recall | 100.00% | 100.00% | Recall | 50.00% | 100.00% | Recall | 50.00% | 100.00% |

Author's personal copy



We also compared our bit-vector processing algorithm with previous approaches based on explanation-based constraint programming, metrics, and a scoring algorithm, and showed that it compares favorably in terms of precision and recall to all existing approaches and, thus, addresses two important aspects of design motif identification: quality of the occurrences and quality of the identification process.

Therefore, we can conclude that the two popular pattern matching approaches in bioinformatics, automata simulation and bit-vector processing, can be used to identify design motifs. Bit-vector processing algorithms provide interesting performance, precision, and recall, with respect to previous approaches. Automata simulation seems too slow to be useful in industrial contexts.

However, as with any other structural approaches, automata simulation and bit-vector processing algorithms suffer from a limited precision when considering the developers' *intent*. Indeed, even if a micro-architecture is identified as being similar to a motif, this similarity does not imply that the developers' introduced it to solve the related design problems for two reasons: first, it can have been implemented without the knowledge of the design pattern—a form a "rediscovery"; or, second, it could be similar to the motif accidentally, *i.e.* it structure is similar but its intent is different. A typical example of the latter reason includes the `State` and `Strategy` design motifs, which are identical and thus lead to similar micro-architectures, even though their intents are clearly different. Consequently, despite their advantages (performance, approximations, description), our approaches are limited by the kinds of data that they use during the identification process. Enriching the models with new kinds of data would lead to an increase in precision and, possibly, a greater correlation between the identifier micro-architecture, their related motifs, and the developers' intent. Yet, it is unclear what kinds of data could be used, which can also be automatically extracted from source code. Only few work address this problem, Kampffmeyer's work tried to address the intent of design patterns [30].

Previous limitations, along with reduced precision and long computation times, explain that the industry is not yet using design motif identification approaches on a daily basis. Moreover, no study has shown so far concretely and extensively a positive impact on the developers' development and maintenance efforts of these approaches. Our work, as some other previous work, is a step towards providing the industry with precise and fast identification approaches. Yet, some more work is necessary to provide *convenient* approaches, *i.e.* unintrusive and integrated to the developers' environment, as well as evidence of a positive impact.

In future work, we will study the use of metric-based analyses to reduce the search space to further improve performance, precision, and recall. Also, we will integrate data related to method and field declarations in the string matching process to have finer-grain definitions of the design motifs to improve precision and recall. We will also compare our approach with other approaches in details, from the choices of the design motif specifications to their performance, precision, and recall. Finally, we plan to investigate the impact on developers' development and maintenance efforts of our approach and others.

We are currently conducting more experiments on even larger systems and are assessing the generalisation of our bit-vector processing algorithm to other types of design motifs, in particular antipatterns [44].

## Acknowledgements

The authors are grateful to Jean-Yves Potvin for all the fruitful discussions. This work has been partially funded by FQRNT and NSERC.

## References

[1] J. Koskinen, Software maintenance costs, web site (September 2004). <http://www.cs.jyu.fi/~koskinen/smcosts.htm>.

[2] B.P. Lientz, E.B. Swanson, Problems in application software maintenance, Communications of the ACM 24 (11) (1981) 763–769. <http://portal.acm.org/citation.cfm?id=358790.358796>.

[3] T.J. Biggerstaff, B.G. Mitbander, D.E. Webster, The concept assignment problem in program understanding, in: V.R. Basili, R.A. DeMillo, T. Katayama (Eds.), Proceedings of the 15th International Conference on Software Engineering, IEEE Computer Society Press/ACM Press, 1993, pp. 482–498. <http://portal.acm.org/citation.cfm?id=257679>.

[4] E. Gamma, R. Helm, R. Johnson, J. Vlissides, Design Patterns – Elements of Reusable Object-Oriented Software, first ed., Addison-Wesley, 1994.

[5] R. Wuyts, Declarative reasoning about the structure of object-oriented systems, in: J. Gil (Ed.), Proceedings of the 26th Conference on the Technology of Object-Oriented Languages and Systems, IEEE Computer Society Press, 1998, pp. 112–124. <http://www.iam.unibe.ch/~wuyts/publications.html>.

[6] A. Quilici, Q. Yang, S. Woods, Applying plan recognition algorithms to program understanding, Journal of Automated Software Engineering 5 (3) (1997) 347–372. <http://www-ee.eng.hawaii.edu/~alex/Research/Abstracts/ause98.html>.

[7] G. Antoniol, R. Fiutem, L. Cristoforetti, Design pattern recovery in object-oriented software, in: S. Tilley, G. Visaggio (Eds.), Proceedings of the Sixth International Workshop on Program Comprehension, IEEE Computer Society Press, 1998, pp. 153–160. <http://citeseer.nj.nec.com/antoniol98design. html>.

[8] Y.-G. Guéhéneuc, H. Sahraoui, Farouk Zaidi, Fingerprinting design patterns, in: E. Stroulia, A. de Lucia (Eds.), Proceedings of the 11th Working Conference on Reverse Engineering, IEEE Computer Society Press, 2004, pp. 172–181. <http://www-etud.iro.umontreal.ca/ptidej/Publications/Documents/WCRE04.doc.pdf>.

[9] G.S. Nikolaos Tsantalis, Alexander Chatzigeorgiou, S.T. Halkidis, Design pattern detection using similarity scoring, Transactions on Software Engineering 32 (2006) 896–909.

[10] D. Gusfield, Algorithms on Strings, Trees and Sequences, Cambridge University Press, Cambridge, UK, 1997.

[11] V.I. Levenshtein, Binary codes capable of correcting deletions, insertions, and reversals, Soviet Physics – Doklady 10 (8) (1966) 707–710.

[12] T. Smith, M. Waterman, Identification of common molecular subsequences, Journal of Molecular Biology 147 (1981) 195–197.

[13] S. Needlman, C. Wunsch, A general method applicable to the search for similarities in the amino acid sequence of two proteins, Journal of Molecular Biology 48 (1970) 443–453.

[14] E. Ukkonen, Finding approximate patterns in strings, Journal of Algorithms 6 (1985) 132–137.

[15] J. Holub, Simulation of NFA in approximate string and sequence matching, in: Proceedings of the Prague Stringology Club Workshop '97, 1997, pp. 39–46.

[16] A. Bergeron, S. Hamel, Vector algorithms for approximate string matching, International Journal of Foundation of Computer Science 13 (1) (2002) 53–66. <www.informatik.uni-trier.de/~ley/db/journals/ijfcs/ijfcs13.html>.

[17] G. Myers, A fast bit-vector algorithm for approximate string matching based on dynamic programming, Journal of the ACM 46 (3) (1999) 395–415. <www.portal.acm.org/citation.cfm?id=316550>.

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

[19] J. Holub, B. Melichar, Implementation of nondeterministic finite automata for approximate pattern matching, in: J.-M. Champarnaud, D. Maurel, D. Ziadi (Eds.), Proceedings of the Third International Workshop on Implementing Automata, Springer-Verlag, 1998, pp. 92–99.

[20] D. Beyer, A. Noack, C. Lewerentz, Simple and efficient relational querying of software structures, in: E. Stroulia, A. van Deursen (Eds.), Proceedings of the 10th Working Conference on Reverse Engineering, IEEE Computer Society Press, 2003, pp. 216–225. <http://citeseer.ist.psu.edu/beyer03simple.html>.

[21] Olivier Kaczor, Y.-G. Guéhéneuc, S. Hamel, Efficient identification of design patterns with bit-vector algorithm, in: G.A. di Lucca, N. Gold (Eds.), Proceedings of the 10th Conference on Software Maintenance and Reengineering IEEE Computer Society Press, 2006, pp. 173–182. <http://www-etud.iro.umontreal.ca/~ptidej/Publications/Documents/ CSMR06a.doc. pdf>.

[22] C. Krämer, L. Prechelt, Design recovery by automated search for structural design patterns in object-oriented software, in: L.M. Wills, I. Baxter (Eds.), Proceedings of the Third Working Conference on Reverse Engineering, IEEE Computer Society Press, 1996, pp. 208–215. <http://www.computer.org/proceedings/wcre/7674/76740208abs.htm>.

[23] O. Ciupke, Automatic detection of design problems in object-oriented reengineering, in: D. Firesmith (Ed.), Proceeding of 30th Conference on Technology of Object-Oriented Languages and Systems, IEEE Computer Society Press, 1999, pp. 18–32. <http://www.computer.org/proceedings/tools/0278/02780018abs.htm>.

[24] R.K. Keller, R. Schauer, S. Robitaille, P. Pagé, Pattern-based reverse-engineering of design components, in: D. Garlan, J. Kramer (Eds.), Proceedings of the 21st International Conference on Software Engineering, ACM Press, 1999, pp. 226–235. <http://www.iro.umontreal.ca/~schauer/Private/Publications/icse1999/icse1999.html>.

[25] J.H. Jahnke, W. Schäfer, A. Zündorf, Generic fuzzy reasoning nets as a basis for reverse engineering relational database applications, in: M. Jazayeri (Ed.),

Author's personal copy



Proceedings of the Sixth European Software Engineering Conference, ACM Press, pp. 193–210. <http://www.uni-paderborn.de/cs/varlet/docs. html>.

[26] Y.-G. Guéhéneuc, N. Jussien, Using explanations for design-patterns identification, in: C. Bessière (Ed.), Proceedings of the 1st IJCAI Workshop on Modeling and Solving Problems with Constraints, AAAI Press, 2001, pp. 57–64. <http://www-etud.iro.umontreal.ca/~ptidej/Publications/Documents/IJCAI01MSPC.doc.pdf>.

[27] D. Eppstein, Subgraph isomorphism in planar graphs and related problems, in: K. Clarkson (Ed.), Proceedings of the Sixth Annual Symposium on Discrete Algorithms, ACM Press, 1995, pp. 632–640. <www.ics.uci.edu/$~$eppstein/pubs/Epp-TR-94-25.pdf>.

[28] N. Jussien, e-Constraints: Explanation-based constraint programming, in: B. O'Sullivan, E. Freuder (Eds.), First CP workshop on User-Interaction in Constraint Satisfaction, 2001. <http://www.emn.fr/jussien/publications/jussien-WCP01.pdf>.

[29] Y.-G. Guéhéneuc, G. Antoniol, DeMIMA: A multi-layered framework for design pattern identification, Transactions on Software Engineering 34 (5) (2008) 667–684. <http://www-etud.iro.umontreal.ca/~ptidej/Publications/Documents/TSE08.doc. pdf>.

[30] H. Kampffmeyer, S. Zschaler, Finding the pattern you need: The design pattern intent ontology, in: G. Engels, B. Opdyke, D.C. Schmidt, F. Weil (Eds.), Proceedings of the 10th International Conference on Model Driven Engineering Languages and Systems, Springer, 2007, pp. 211–225. <http://www.springerlink.com/content/303028v312560v26/>.

[31] Y.-G. Guéhéneuc, H. Albin-Amiot, Recovering binary class relationships: putting icing on the UML cake, in: D.C. Schmidt (Ed.), Proceedings of the 19th Conference on Object-Oriented Programming, Systems, Languages, and Applications, ACM Press, 2004, pp. 301–314. http://www-etud.iro.umontreal.ca/ptidej/Publications/Documents/OOPSLA04.doc. pdf.

[32] K. Mens, R. Wuyts, T. D'Hondt, Declaratively codifying software architectures using virtualsoftware classifications, in: D. Firesmith (Ed.), Proceedings of the 30th international conference on Technology of Object-Oriented Languages and Systems, IEEE Computer Society Press, 1999, pp. 33–45. <http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=778997>.

[33] G.B. Dantzig, Linear Programming and Extensions, Princeton University Press, 1963.

[34] H.A. Eiselt, M. Gendreau, G. Laporte, Arc routing problems. Part I: The Chinese Postman problem, Tech. Rep. CRT-960, Centre de Recherche sur les Transports (March 1994).

[35] V.M.G. Gluskov, The abstract theory of automata, Russian Mathematical Surveys 16 (1961) 1–53.

[36] K. Thompson, Regular expression search algorithm, Communications of the ACM 11 (1968) 419–422.

[37] J.E. Hopcroft, J.D. Ullman, Introduction to Automata Theory, Languages, and Computation, Addison-Wesley, 1979.

[38] Y.-G. Guéhéneuc, Ptidej: Promoting patterns with patterns, in: M.E. Fayad (Ed.), Proceedings of the First ECOOP workshop on Building a System using Patterns, Springer-Verlag, 2005. <http://www-etud.iro.umontreal.ca/~ptidej/Publications/ Documents/ECOOP05BSUP.doc.pdf>.

[39] H.W. Thimbleby, The directed chinese postman problem, Journal of Software – Practice and Experience 33 (11) (2003) 1081–1096. <http://www.uclic.ucl.ac.uk/harold/cpp/>.

[40] K. Scheglov, J.-M.P. Shackelford, Eclipse profiler (September 2004). <www.eclipsecolorer.sourceforge.net/index_profiler.html>.

[41] E. Agerbo, A. Cornils, How to preserve the benefits of design patterns, in: C. Chambers (Ed.), Proceedings of the 13th Conference on Object-Oriented Programming, Systems, Languages, and Applications, ACM Press, 1998, pp. 134–143. <http://citeseer.nj.nec.com/31381.html>.

[42] L. Wendehals, Improving design pattern instance recognition by dynamic analysis, in: J.E. Cook, M.D. Ernst (Eds.), Proceedings of the First ICSE Workshop on Dynamic Analysis, IEEE Computer Society Press, 2003. <http://www.cs.nmsu.edu/jcook/woda2003/papers/Wendehals.pdf>.

[43] J. Bieman, G. Straw, H. Wang, P.W. Munger, R.T. Alexander, Design patterns and change proneness: an examination of five evolving systems, in: M. Berry, W. Harrison (Eds.), Proceedings of the Ninth International Software Metrics Symposium, IEEE Computer Society Press, 2003, pp. 40–49. <http://csdl.computer.org/comp/proceedings/metrics/2003/1987/00/19870040abs. htm>.

[44] W.J. Brown, R.C. Malveau, W.H. Brown, H.W. McCormick, W. Hays III, T.J. Mowbray, Anti Patterns: Refactoring Software, Architectures, and Projects in Crisis, first ed., John Wiley and Sons, 1998. <http://www.amazon.com/exec/obidos/tg/detail/-/0471197130/ref=ase_theantipatterngr/103-4749445-6141457>.