DeMIMA: A Multilayered Approach for Design Pattern Identification

Yann-Gaël Guéhéneuc, *Member*, *IEEE*, and Giuliano Antoniol, *Member*, *IEEE*

Abstract—Design patterns are important in object-oriented programming because they offer design motifs, elegant solutions to recurrent design problems, which improve the quality of software systems. Design motifs facilitate system maintenance by helping maintainers to understand design and implementation. However, after implementation, design motifs are spread throughout the source code and are thus not directly available to maintainers. We present DeMIMA, an approach to semiautomatically identify microarchitectures that are similar to design motifs in source code and to ensure the traceability of these microarchitectures between implementation and design. DeMIMA consists of three layers: two layers to recover an abstract model of the source code, including binary class relationships, and a third layer to identify design patterns in the abstract model. We apply DeMIMA to five open-source systems and, on average, we observe 34 percent precision for the 12 design motifs considered. Through the use of explanation-based constraint programming, DeMIMA ensures 100 percent recall on the five systems. We also apply DeMIMA on 33 industrial components.

Index Terms—Maintenance traceability, design patterns, interclass relationships.

1 Introduction

Maintainers must be aware of design choices in order to modify an object-oriented software system appropriately. Design choices include all decisions made by developers when designing and implementing the system: the structures of classes and the relationships among them. However, design choices are often scattered in the source code of systems after implementation because, with available object-oriented programming languages, they do not transcribe directly into source code; developers must write several lines of code using constructs of the languages to implement their choices. Moreover, documentation is often obsolete, if it even exists, and these choices are thus lost.

However, design choices are often implemented with recurring patterns, "a form or model proposed for imitation" [1], to facilitate writing and understanding the source code. Idioms and design patterns are two types of patterns; architectural patterns and micropatterns are others. Idioms are low-level patterns specific to some programming languages and to the implementation of particular characteristics of classes or their relationships. They are intraclass patterns describing typical implementation of, for example, relationships, object containment, and collection traversal. Design patterns [2] are recurring interclass patterns that define solutions to common design problems in the organization of classes. They are "tactics" that generate the structure and behavior of classes and their relationships [3]. They influence the design of modules and classes but not the overall architecture. They are defined in terms of classes and relationships; thus their implementation uses idioms.

We use the term motif to express the solution of a pattern as "a reliable sample of traits, acts, tendencies, or other observable characteristics" [1]. We distinguish between patterns and motifs because patterns often encompass information that is not readily available for their identification. For example, the Composite design pattern [2, p.163] also includes information about its intent, motivation, applicability, and consequences, which are not observable characteristics. Only its structure, its participants, and their collaborations are observable in the source code. Thus, strictly speaking, we cannot use the terms design pattern "identification," "detection," or "instantiation" but rather the instantiation and identification of microarchitectures similar to some motifs; thus, we use the term "design motif identification" for the process traditionally called design pattern identification.

We define the term microarchitectures as concrete manifestations of some motifs in the implementation of a system. A microarchitecture is composed of classes, methods, fields, and relationships having structure and organization similar to one or more motifs. A microarchitecture can be similar to more than one motif because only developers may decide intent, motivation, and consequences.

Developers usually search for some kinds of patterns in order to understand a system [4]; by recognizing concrete manifestations of these patterns, they deduce, from their experience, the design choices underlying the presence of motifs in the source code. During maintenance and evolution, maintainers would greatly benefit from knowing the design choices made during implementation, see, for example, [5].

To support design pattern identification and program comprehension, we combine and extend our previous work [6], [7], [8] in a new multilayered approach named the Design Motif Identification Multilayered Approach (DeMIMA). DeMIMA makes it possible to recover two kinds of design

- *Y.G. Guéhéneuc is with the Département d'Informatique et Recherche Opérationnelle, Université de Montréal, C.P. 6128, succ. Centre Ville, Montréal, Québec, H3C 3J7, Canada. E-mail: guehene@iro.umontreal.ca.*
- *G. Antoniol is with the Département d'Informatique, École Polytechnique de Montréal, C.P. 6079, succ. Centre Ville, Montréal, Québec, H3C 3A7, Canada. E-mail: antoniol@ieee.org.*

Manuscript received 18 Apr. 2007; revised 1 Apr. 2008; accepted 29 May 2008; published online 27 June 2008.
Recommended for acceptance by R. Taylor.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number TSE-2007-04-0133.
Digital Object Identifier no. 10.1109/TSE.2008.48.

choices from source code: idioms pertaining to the relationships among classes and design motifs characterizing the organization of the classes. DeMIMA is extensible and scalable; it ensures traceability between motifs and source code by first identifying idioms related to binary class relationships to obtain an idiomatic model of the source code and then using this model to identify design motifs and generate a design model of the system. On average, we observe 34 percent precision for the 12 design motifs considered and the five open-source systems on which we apply our approach. DeMIMA ensures 100 percent recall on the five systems. We also apply DeMIMA on industrial system source code and designs.

The remainder of the paper is organized as follows: In Section 2, we give an overview of the approach and justify its rationale. In Section 3, we summarize related work and present essential characteristics of the identification steps. In Sections 4.2 and 4.3, we describe our approach and discuss its characteristics. In Section 6, we apply the approach on a testbed of open source and industrial systems. In Section 7, we summarize our work and discuss future challenges.

2 Design Motif Identification

2.1 Context

We have broken down the comprehension process that maintainers use to identify recurring motifs in the source code into three tasks.

1. Identifying a microarchitecture μA similar to some motifs from a set of known patterns $\mathcal{S}_{DP}$. Maintainers analyze a system source code S, either manually or using tools, and identify subsets of the source code that are similar to known motifs.
2. Contextualizing μA to keep a unique motif from $\mathcal{S}_{DP}$ using semantic data extrinsic to S. Maintainers choose in $\mathcal{S}_{DP}$ the pattern DP whose corresponding motif DM is embodied by μA. Contextualization depends on the system domain and on the maintainers' experience and understanding of the system.
3. Comprehending S. Maintainers deduce from DP, whose motif DM was manifested by μA during the implementation of S, the design choice behind μA, including the intent and motivation of the developers and the consequences on the overall system design.

Because subtasks 2 and 3 depend on the maintainers' experience and the system domain, they are difficult to automate. In contrast, task 1, which is tedious and error prone [9], [10] is a good candidate for automation.

2.2 Problem

Design motifs are described with UML-like class and sequence diagrams,[1] which represent different aspects of software systems [11]. Class diagrams are global models of systems, representing their entities and the relationships among entities, while sequence diagrams specify local interactions in entities and sequences of method calls among entities.

In the rest of this paper, we only consider class diagrams because they are most frequently used to describe design motifs [12]. Also, class diagrams are often produced early in the development cycle and are the sole reliable documentation because they can be reverse engineered with reasonable accuracy. We will use other information in future work.

DeMIMA assists maintainers in task 1 by providing a three-step identification process of a design motif DM in the source code S of a system based on UML-like class diagram models:

1. Model the source code S as a model $\mathcal{M}_S$ using a subset of the language used to describe models of motifs and including all of the constituents corresponding to constructs of S, as explained in Section 4.1.
2. Enrich model $\mathcal{M}_S$ with idioms that reveal binary class relationships to obtain a model $\mathcal{M}_I$, which uses the same language used to describe models of motifs, as detailed in Section 4.2.
3. Enrich the model $\mathcal{M}_I$ through the following three substeps, as shown in Section 4.3:
   a. Build a model $\mathcal{M}_{DM}$ of a motif DM as a class diagram with the formalism used to describe $\mathcal{M}_I$.
   b. Identify microarchitectures similar to $\mathcal{M}_{DM}$ in $\mathcal{M}_I$. A microarchitecture μA might be either a complete form if its entities and their relationships match one to one the entities and relationships in $\mathcal{M}_{DM}$ or an approximate form if they do not, e.g., if a suggested relationship between two entities does not exist.
   c. Instantiate a model $\mathcal{M}_D$ based on $\mathcal{M}_I$ and enriched with models $\mathcal{M}_{\mu A}$ of the identified microarchitectures.

Any approach to design motif identification should maintain a traceability link between the different layers from source code up to the identified microarchitectures:

$$S \overset{1}{\rightleftharpoons} \mathcal{M}_S \overset{2}{\rightleftharpoons} \mathcal{M}_I \overset{3}{\rightleftharpoons} \mathcal{M}_D \left(\supset \{\mathcal{M}_{\mu A}\}\right), \tag{1}$$

where $\overset{x}{\rightleftharpoons}$ describes the xth layer to produce the next model.

Example. In the rest of this paper, we use the simple example taken from [6] and shown in Fig. 1 to illustrate the different steps performed by DeMIMA. The example uses two classes, `C1` and `C2`, linked by an aggregation relationship. The aggregation relationship exists through the field `C2 c2` and the `void operation1()` method body.

We want to identify in this source code any microarchitecture similar to the design motif represented by the UML-like class diagram in Fig. 2a. Thus, we need to first recover a model $\mathcal{M}_S$ of the system, then refine this model into $\mathcal{M}_I$, which includes the aggregation relationship, and, finally, model and match the motif $\mathcal{M}_{DM}$ against $\mathcal{M}_I$ to create a model $\mathcal{M}_D$, which includes the result of the matching, $\mathcal{M}_{\mu A}$, as shown in Fig. 2.

2.3 Our Solution

In DeMIMA, we characterize the constituents of class diagrams and propose algorithms to identify these constituents in source code. Basically, class diagrams consist of classes, fields, methods, interfaces, inheritance, and implementation relationships. We concur with Dave Thomas that "Every model needs a metamodel" [13]. Thus, we define a metamodel, *Pattern and Abstract-level Description*

1. Design motifs notation borrows from OMT class diagrams, OBJECTORY interaction diagrams, and the BOOCH method [2].

```
1  public class Example {
2      public static void main(String[] args) {
3          C2 c2 = new C2();
4          C1 c1 = new C1(c2);
5          c1.operation1();
6          c2.operation2();
7          ...
8      }
9  }
10 public class C1 {
11     private C2 c2;
12     public C1(C2 c2) {
13         this.c2 = c2;
14     }
15     public void operation1() {
16         this.c2.operation2();
17     }
18 }
19 public class C2 {
20     public void operation2() {
21     }
22 }
```

Fig. 1. Source code of the running example.

Language (PADL), to express these constituents. New constituents may be added to PADL using inheritance to enrich the descriptions of systems. The methods of the constituents define the semantics of models obtained from the metamodel. Our objective in defining PADL is to have a simple and extensible language to describe and reason about abstractions pertinent to our problem, namely, $\mathcal{M}_S$, $\mathcal{M}_I$, $\mathcal{M}_D$, $\mathcal{M}_{DM}$, and $\mathcal{M}_{\mu A}$.

DeMIMA reuses the definitions in [6] of the use, creation, association, aggregation, and composition relationships to formalize these relationships with four language-independent properties: exclusivity, message receiver type, lifetime, and multiplicity. DeMIMA distinguishes use, association, aggregation, and composition relationships because such relationships exist in most notations to model systems, for example, in UML, and because design motifs are defined using these relationships. Thus, it is able to identify these relationships in $\mathcal{M}_S$ and represent these in $\mathcal{M}_I$.

The language to describe design motifs is the same as that used to describe models of systems. DeMIMA uses explanation-based constraint programming and constraint relaxation to identify microarchitectures, complete or approximate, similar to the modeled design motifs, without explicitly enumerating all possible variants to produce a model $\mathcal{M}_D$.

DeMIMA ensures the traceability of design motifs between implementation and design because it uses the same language to describe $\mathcal{M}_S$, $\mathcal{M}_I$, and $\mathcal{M}_D$ (by construction, each layer is a refinement of the previous layer) and because it explicitly records, in the more abstract constituents, the set of lower-level constituents that led to their existence. Thus, a constituent of $\mathcal{M}_S$ (respectively, of $\mathcal{M}_I$

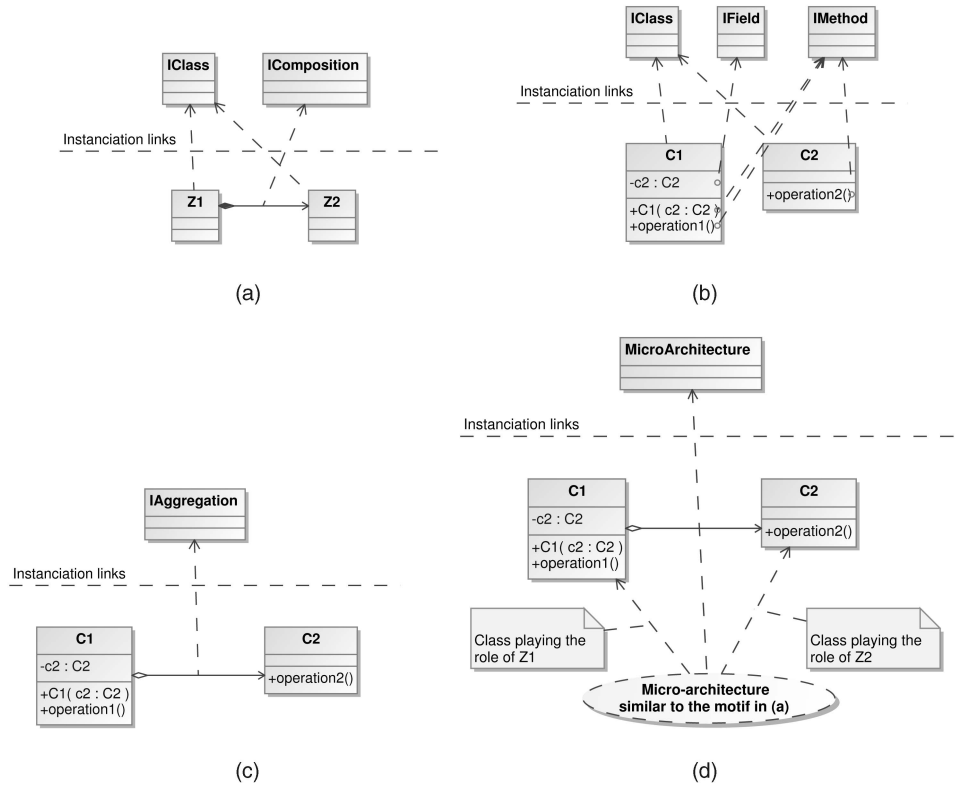

Fig. 2. Models of the motif and of the source code for the running example. (a) The bottom part shows the UML-like diagram of a simple motif; this part, together with the upper part, represents $\mathcal{M}_{DM}$. (b) UML-like diagram of $\mathcal{M}_S$. (c) UML-like diagram of $\mathcal{M}_I$ (some instantiation links are omitted). (d) UML-like diagram of $\mathcal{M}_D$ (some instantiation links are omitted).

and $\mathcal{M}_{\mathcal{D}}$ ($\supset \mathcal{M}_{\mu A}$)) can be traced back to the source code constructs in S (respectively, to the constituents in $\mathcal{M}_S$ and $\mathcal{M}_I$) from which it originates.

3 Related Work

We classify the related work according to the recovered models because obtaining and abstracting the data needed to identify design motifs is problematic. We conclude with a summary of essential characteristics of any identification approach for design motifs.

3.1 Related Work on $\mathcal{M}_S$

Building a model of the source code is the first step of any static analysis. The objective of this step is to obtain a model of the source code that can be manipulated programmatically. This step can be performed using a readily available parser technology such as JAVACC or COLUMBUS [14].

3.2 Related Work on $\mathcal{M}_I$

Several authors proposed approaches to extract binary class relationships, which is an important concern when building models of source code. Indeed, these relationships are not explicit constructs of mainstream object-oriented programming languages, such as C++, Java, or Smalltalk, and they lack precise definitions.

Jahnke et al. [15] and Niere et al. [16] introduced generic fuzzy reasoning nets (GFRN) to recover association relationships among entities in the context of the Fujaba project. They proposed a set of clichés from source code. Source code clichés used together with GFRN allow identifying associations relationships while managing variations of implementation. Although their work is promising, the use of GFRN is complex and they consider association relationships only, not aggregation and composition relationships. More recently, Niere et al. [17] introduced an approach based on fuzzy beliefs able to recover association and aggregation relationships in large software systems while handling impreciseness.

Jackson and Waingold [18] developed WOMBLE, a tool for the lightweight extraction of object models from Java bytecodes. They described an object model as a graph wherein nodes are entities and links are binary class relationships. Relationships considered in WOMBLE are inheritance, association, and aggregation. WOMBLE includes heuristics to infer the target entities of association and aggregation relationships. This work is a source of inspiration even though it did not consider composition relationships.

In general, previous work was limited by the lack of commonly agreed upon definitions for binary class relationships. Moreover, to the best of our knowledge, no definitions of the association, aggregation, and composition relationships existed, describing how these relationships must be implemented in source code. For example, [19], [20], [21], [22] proposed definitions of these relationships, but there were no hints on their concrete implementation. Thus, the first step toward design motif identification is to define the association, aggregation, and composition relationships and to obtain models of systems that integrate these relationships. A complete survey of the subject is available in [6].

3.3 Related Work on $\mathcal{M}_D$ (Including $\mathcal{M}_{DM}$ and $\mathcal{M}_{\mu A}$)

Several authors proposed approaches to identify microarchitectures similar to design motifs. In general, these approaches rely on a design motif library; thus they are similar to the program understanding and architectural recovery approaches based on *clichés* matching and plan recognition. The main problems of these approaches as identified by Wills' precursor work [23] and put forward recently by Niere et al. [24] is that a design motif may appear in several different forms due to variants. Wills classifies the main sources of variants as syntactic variation, implementation variation, delocalization, organization variation, redundancy, unrecognizable code, and function sharing. Syntactic variation is mostly with regard to the syntactic level clichés. Cliché recognizers traditionally embody the knowledge of all of the different forms that a certain cliché can assume. This is not the case in our approach, where the use of explanation-based constraint programming accounts for syntactic variants. Implementation variation is related to the fact that a given concept may be implemented in different ways: An aggregation may be implemented with a list or a set or any other user-defined type. We define such relationships using language-independent properties to avoid this problem. Another example concerns the depth of the inheritance tree between a superclass and a derived class participating in a motif (see, for example, the Composite design motif). Again, the use of explanation-based constraint programming deals with such variants. The other problems highlighted by Wills—delocalization, redundancy, unrecognizable code, and function sharing—do not concern our approach.

Rich and Waters [4] proposed the use of constraint programming to recognize plans in Cobol source code. Cobol systems are modeled by their abstract syntax trees. A plan is modeled as nodes of the abstract syntax tree and constraints among nodes (control and data-flow, function calls...). The identification of a plan in source code is converted to a constraint satisfaction problem in which nodes of the plan are variables, constraints among nodes are constraints among variables, and the source code abstract syntax tree is the domain of the variables. This work is the first account of the use of constraint programming for plan identification. However, it does not apply to design motif identification because plans are low level and it does not identify approximate forms of the plans. Nevertheless, we draw from this work two important characteristics of design motif identification: the need for explanations and for approximations [4, pp. 83 and 181].

Other approaches to design motif identification used clichés recognition algorithms such as unification, see the precursor work by Krämer and Prechelt [25]. An example is the SOUL environment [5], a logic programming environment based on Smalltalk that directly manipulates Smalltalk constructs through predicates. The SOUL environment allows direct representation of the abstract syntax tree of the Smalltalk source code managed by the underlying environment as logic facts. Using these facts, it is possible to build a library of predicates and to identify entities whose structures and organizations correspond to design motifs. However, the use of logic programming requires the definitions of predicates for all possible variants, i.e., all expected variations of implementation. The definition of all variants of implementation is cumbersome. Also, the use of logic programming does not explain the presence or absence of microarchitectures similar to design motifs.

Other authors introduced the use of queries to identify entities whose structure and organization are similar to design motifs [26], [27]. In particular, Keller et al. [27] introduced the SPOOL environment for reverse engineering, which allows manual, semiautomated, or automated identification of abstract design components using queries on source code models. A query is manually associated with an abstract design component and applied to a source code model. The main limitation of this work is the need to develop and associate queries with abstract design components manually and with each possible variant of their implementation.

Generic fuzzy reasoning nets have also been applied to the identification of design motifs [24], [28]. A design motif is described as a generic fuzzy reasoning net representing rules to identify microarchitectures similar to its implantation in source code. However, this approach has not been pursued or implemented despite its promises. Moreover, it is difficult to express design motifs as generic fuzzy reasoning nets and to modify them.

Graphs and graph-transformation techniques also have been used to describe and identify design motifs in system source code [29], [30]. A design motif is described as a graph whose nodes represent entities and whose edges represent relationships among entities. The identification of microarchitectures corresponds to a graph isomorphism: the identification of a subgraph similar to a given graph in a graph, which is a difficult problem [31]. Pettersson and Löwe [32] proposed transforming graphs of systems into planar graphs to improve performance with interesting results. An approach based on similarity scoring has also been proposed [33] which provides an efficient means to compute the similarity between the graph of a design motif and the graph of a system to identify classes potentially playing a role in the design motif. Although efficient, these approaches are not interactive, do not explain their results, and only allow a limited set of approximations.

Finally, several authors proposed dedicated syntactic analyses to identify design motifs in source code, for example, [34], [35], [36], [37]. These analyses are efficient in time, recall, and precision but are specialized to particular design motifs. We propose a more general solution that uses standard algorithms, as offered by constraint programming. Some authors, such as Heuzeroth et al. [38], combined static and dynamic analyses to improve the precision of the identification but faced the problem of the choice of the methods to instrument and of the scenarios to execute.

3.4 Summary of the Characteristics of DeMIMA

From our study of the related work, DeMIMA must possess the following characteristics:

- Models of source code must differentiate among use, association, aggregation, and composition relationships so that design motif models are as close as possible to their usual descriptions in [2].
- A given model of a design motif must serve to identify both complete and approximate forms of microarchitectures similar to the design motif without explicitly enumerating all variants.
- The algorithms must be semiautomatic or automatic and must explain the identified microarchitectures so that maintainers can direct their search to easily distinguish possible false positives.

Contributions of DeMIMA are the following: For the first time, as suggested in previous work, an approach brings a solution to the identification of microarchitectures similar to design motifs using commonly agreed-upon definitions of the unidirectional binary class relationships, unique representations of design motifs, and semiautomated and/or automated algorithms explaining identified microarchitectures. Thus, it complies with the characteristics of the identification of microarchitectures similar to design motifs. In particular, explanation-based constraint programming explains identified microarchitectures for maintainers to direct their search and discriminate among possible false positives easily. Explanation and constraint relaxation lead to interactive or automatic algorithms while naturally tackling the problem of variants identified by Wills [23].

4 Multilayered Approach

DeMIMA relies on a multilayered approach, detailed in the following sections.

4.1 First Layer: Source Code Model $\mathcal{M}_S$

The first layer consists of an infrastructure, e.g., parsers, to obtain models $\mathcal{M}_S$ of the source code of systems. $\mathcal{M}_S$ is expressed using the language defined by the metamodel shown in Fig. 3 (Part 1 exclusively) and inspired by UML. It includes all of the constituents found directly in any Java object-oriented system: class, interface, member class and interface, method, field, inheritance and implementation relationships, and rules controlling their interactions. The constituents describe the structure of systems and a subset of their behavior. The main constituents in the metamodel and their relationships are the following:

- *Class* `Entity` *to describe entities of a system*. An entity might be a `Class` or an `Interface`.
- *Class* `Element`*, to describe elements of entities*. An element might be a `Method` or a `Field`.

A model of a system is an instance of class `Program-Model`. It contains a set of entities, each of which contains a set of elements.

We have implemented the first layer to cope with any number of parsers for various programming languages (e.g., C++ and Java) and produce an instance of `Program-Model` representative of the parsed source code:

$$S \overset{1}{\rightleftharpoons} \mathcal{M}_S. \tag{2}$$

Example. Fig. 2b shows a UML-like diagram of the model $\mathcal{M}_S$ of the source code illustrated in Fig. 1, as well as the instantiation links between the objects in $\mathcal{M}_S$ and their classes reported in Part 1 of Fig. 3.

4.2 Second Layer: Idiom-Level Model $\mathcal{M}_I$

The second layer describes systems at a higher level of abstraction than their source code by making explicit certain programming idioms. Idioms reveal particular characteristics of classes or their relationships. For example, a class could be stereotyped as a UML Data Type according to certain idioms used in its implementation [39]. Thus, in general, idioms can implement other characteristics of classes than binary class relationships. Nevertheless, in the rest of this paper, we only study binary class relationships as they are relevant to design motif identification; the

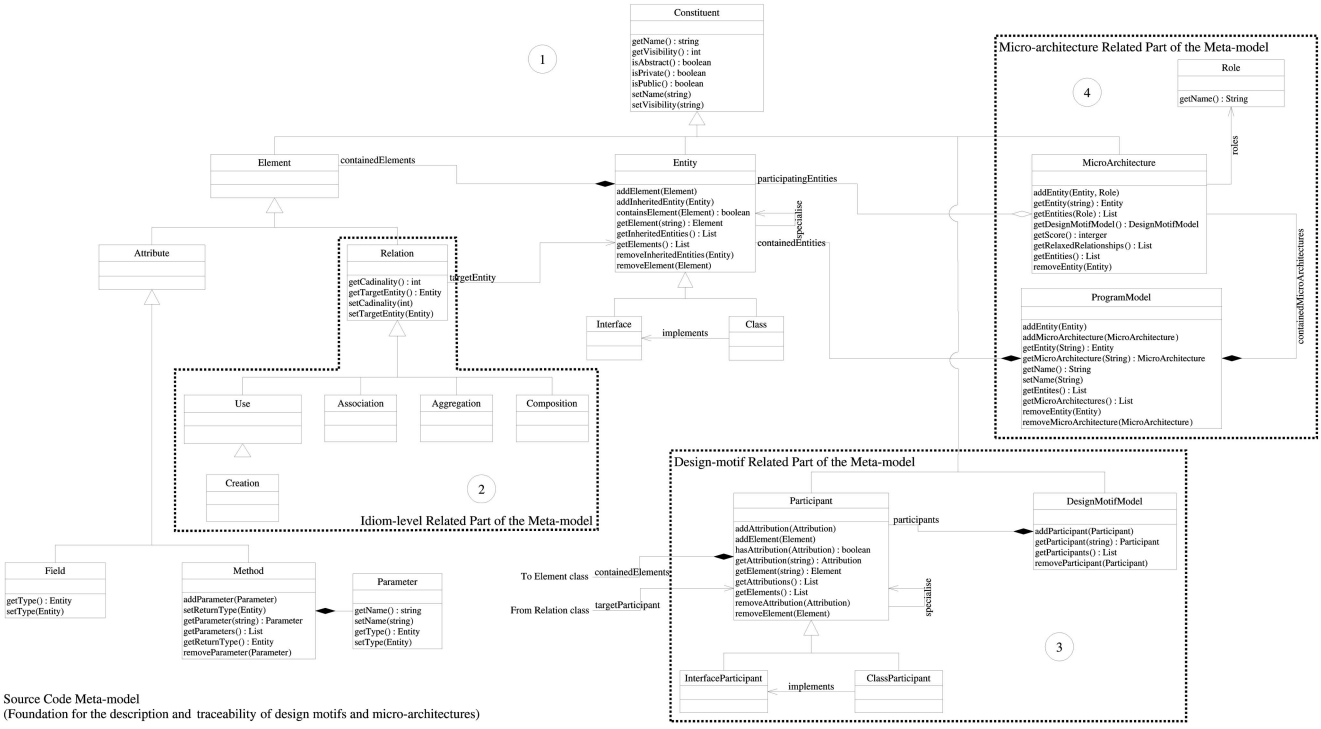

Fig. 3. Metamodel to describe the source code of systems.

terms idioms and binary class relationships are therefore interchangeable.

This layer provides models $\mathcal{M}_I$ of systems in which binary class relationships are reified as first-class entities. We focus on the use, association, aggregation, and composition unidirectional binary class relationships as commonly advocated in UML-like notations because these relationships are used to describe design patterns [2]. Parts 1 and 2 (exclusively) in Fig. 3 present the language to describe idiom-level models.

4.2.1 Informal Definitions

An extensive survey of the literature related to the relationships in different domains such as database, software engineering, or reverse engineering can be found in [6]. Table 1 summarizes the definitions of the relationships used in DeMIMA from the existing links among instances. Association, aggregation, and composition are relationships among instances of classes. Relationships involving classes (not instances) are modeled as use relationships.

Let `A` and `B` be two classes. Association and aggregation relationships allow multiple instances of `A` and `B` to take part in the relationship. The composition relationship allows multiple instances of `B` to be in a relationship with one instance of `A` at a time. In an aggregation relationship, instances of `A` access instances of `B` through a field as a particular type of message receiver. In a composition relationship, instances of `B` are exclusive to their corresponding instances of `A` and instances of `A` and `B` have related lifetimes.

4.2.2 Definitions of the Properties

The definitions of the binary class relationships use four language-independent properties. We present here only information needed to explain the subsequent formal definitions; more details and examples of each property are available elsewhere [6].

An instance of class `B` involved at a given time in a relationship with an instance of class `A` may also participate in another relationship at the same time. We name $\mathbb{B}$ the set $\{true, false\}$. We define the *exclusivity* property EX as

$$EX : Class \times Class \rightarrow \mathbb{B}.$$

Instances of class `A` involved in a relationship send messages to instances of class `B`. We name any the set of all possible message receivers:

$$any = \{\texttt{field},\ \texttt{array field},\ \texttt{collection field},\ \texttt{parameter},\ \texttt{array parameter},\ \texttt{collection parameter},\ \texttt{local variable},\ \texttt{local array},\ \texttt{local collection}\}.$$

We distinguish three types of message receivers: fields, parameters, and local variables. Also, we distinguish "simple" message receivers from arrays and collections because they imply different sets of programming idioms for their declarations and uses and thus different identification strategies. The set any of receivers is language independent and its elements correspond to concepts available in object-oriented programming languages, such as C++, Java, and Smalltalk. We define the receiver type property RT [2] as

$$RT : Class \times Class \rightarrow any.$$

The lifetime property LT constrains the lifetime of all instances of class `B` with respect to the lifetime of all instances of class `A`. It relates to the difference between the

2. The RT property was formerly named "invocation site" IS in [6] but is renamed to avoid confusion with the location of a method invocation.

TABLE 1
Definitions and Applicability of the Unidirectional Relationships in Our Model

Link			Is described by	Relationship		
Origin	*Means*	*Target*		*Origin*	*Name*	*Target*
Class	Any	Class		Class	Use	Class
Instance	Direct	Instance		Class	Association	Class
Instance	Field	Instance		Class	Aggregation	Class
Instance	Field + Lifetime property	Instance		Class	Composition	Class

times of destruction LT_d of two instances of classes A and B [21]. The time is in any convenient unit such as seconds or CPU ticks:[3]

$$LT_d : Instance \rightarrow \mathbb{N}.$$

In programming languages with garbage collection, LT_d matches the moment where an instance is ready to be collected for garbage. We infer from LT_d a relation between the lifetimes of all instances of two classes A and B. We name $\|$ the set $\{-, +\}$:

$$LT : Class \times Class \rightarrow \ \|.$$

The multiplicity property MU specifies the number of instances of class B allowed in a relationship. We express this property as[4]

$$MU : Class \times Class \rightarrow \mathbb{N} \cup \{+\infty\}.$$

The four properties are orthogonal, but the exclusivity and multiplicity properties are closely related. For example, in the Country-Language relationship, we have the following:

- The multiplicity property states the number of instances of class Language that each instance of class Country possesses:

  $$MU(\texttt{Country}, \texttt{Language}) = [1, +\infty].$$

  (For example, Canada possesses two official languages, English and French, and several spoken languages, Inuktitut, Punjabi, Portuguese, and so on.)
- The exclusivity property states that an instance of class Language is shared among instances of class Country and of other classes:

  $$EX(\texttt{Country}, \texttt{Language}) = false.$$

  (French is spoken in Canada, in France, ...)

Example. The values of the four properties are reported and commented on in Table 2 for the source code of the running example in Fig. 1.

3. $\mathbb{N}$ represents the set of all natural numbers.
4. We need $+\infty$ to denote multiplicities with no limit in the numbers of instances in the relationships.

4.2.3 Formalizations of the Relationships

Using EX, LT, MU, and RT, formalizations of the relationships are expressed as three conjunctions, respectively, AS, association, AG, aggregation, and CO, composition. The formalizations of the relationships are important because they are the basis of the identification algorithms needed to abstract $\mathcal{M}_S$ into $\mathcal{M}_I$.

An association between classes A and B characterizes the ability of an instance of A to send a message to an instance of B. Nothing prevents other relationships from linking classes A and B. We define $AS(\texttt{A}, \texttt{B})$ as

$$\begin{aligned} &AS(\texttt{A}, \texttt{B}) = \\ &(RT(\texttt{A}, \texttt{B}) = any) \quad \wedge \quad (RT(\texttt{B}, \texttt{A}) = \emptyset). \end{aligned}$$

An aggregation exists between classes A and B when the definition of A, the whole, contains instances of B, its part. The whole must define a field ("simple," array, or collection) of the type of its part. Instances of the whole send messages to instances of its part. We formalize $AG(\texttt{A}, \texttt{B})$ as

$$\begin{aligned} AG(\texttt{A}, \texttt{B}) = \\ \big(RT(\texttt{A}, \texttt{B}) \quad &\subseteq \quad \{\texttt{field}, \texttt{array field}, \\ &\qquad \texttt{collection field}\}\big) \quad \wedge \\ (RT(\texttt{B}, \texttt{A}) \quad &= \quad \emptyset) \qquad\qquad\qquad\qquad \wedge \\ (MU(\texttt{A}, \texttt{B}) = [1, +\infty]) \quad &\wedge \quad (MU(\texttt{B}, \texttt{A}) = [0, +\infty]). \end{aligned}$$

A composition is an aggregation with a constraint between the lifetimes of the whole and its part and a constraint on the ownership of the part by the whole. Instances of the whole own the instances of its part. Instances of the part might be instantiated before the whole is instantiated, but they must not belong to any other whole. They are exclusive to the instance of the whole. The definition of the composition relationship allows only an association between part and whole to ensure the lifetime and ownership properties between whole and part. We define $CO(\texttt{A}, \texttt{B})$ as

TABLE 2
Values of the Four Properties Instantiated for the Running Example

$EX(\texttt{C1},\texttt{C2}) = false$	On line 3 and subsequent lines, the instance of `C2` could be passed on to other instances of `C1`
$EX(\texttt{C2},\texttt{C1}) = true$	On line 4, the instance of `C1` is built with one particular instance of `C2`
$RT(\texttt{C1},\texttt{C2}) = \{\texttt{field}\}$	On line 11 by construction of the example
$RT(\texttt{C2},\texttt{C1}) = \emptyset$	There is no receiver type by construction of the example
$LT(\texttt{C1},\texttt{C2}) = \parallel$	By construction of the main method, LT can either be $+$ or $-$
$LT(\texttt{C2},\texttt{C1}) = \parallel$	
$MU(\texttt{C1},\texttt{C2}) = 1$	On line 11 by construction of the example
$MU(\texttt{C2},\texttt{C1}) = 0$	There is no receiver type by construction of the example

$$
\begin{aligned}
CO(\texttt{A},\texttt{B}) = \\
(EX(\texttt{A},\texttt{B}) = true) \;\wedge\; (EX(\texttt{B},\texttt{A}) = false) \;\wedge\; \\
\big(RT(\texttt{A},\texttt{B}) \;\subseteq\; \{\texttt{field},\ \texttt{array field},\ \\
\texttt{collection field}\}\big) \;\wedge\; \\
(RT(\texttt{B},\texttt{A}) \;=\; \emptyset) \;\wedge\; \\
(LT(\texttt{A},\texttt{B}) = +) \;\wedge\; (LT(\texttt{B},\texttt{A}) = -) \;\wedge\; \\
(MU(\texttt{A},\texttt{B}) = [1,+\infty]) \;\wedge\; (MU(\texttt{B},\texttt{A}) = [1,1]).
\end{aligned}
$$

Example. According to the values of the properties detailed in Table 2 for the running example and to the formalizations of the relationships $AS(\texttt{C1},\texttt{C2}) = false$, $AG(\texttt{C1},\texttt{C2}) = true$, $CO(\texttt{C1},\texttt{C2}) = false$. No relationships are identified between `C2` and `C1`.

4.2.4 Discussions

The formalizations of the relationships consist of two fundamental parts: a static part corresponding to the MU and RT properties and a dynamic part corresponding to the EX and LT properties. Association and aggregation are inherently static, so their static parts are important for their detection. A composition is an aggregation with additional constraints on the behavior of composed instances; thus, its dynamic parts are important for its distinction from an aggregation and its detection.

Minimality of the properties and common usage of the relationships supported by our formalizations are explained in [6].

4.2.5 Creation of the Model

With the formalizations of the relationships, we define algorithms to identify in models $\mathcal{M}_S$ association, aggregation, and composition relationships to produce models $\mathcal{M}_I$. These algorithms depend only on the properties which isolate the formalizations of the relationships from any coding conventions, similar to the concept of subpatterns in [24].

Identification of association relationships requires collecting the value of the RT property. Identification of aggregation relationships requires inferring the values of the RT and MU properties. Identification of composition relationships requires collecting the value of the RT and MU properties and of the EX and LT properties. DeMIMA computes the RT and MU properties using static analyses and can infer values of the EX and LT properties using dynamic analyses.

Any algorithm recovering aggregation relationships needs to deal with the difficulty that arises when message receivers are untyped collections [18], `collection field`, `collection parameter`, `local collection`, because they are typed with the class hierarchy root `Object`. Algorithms have been proposed to deal with this difficulty, for example, [18]. Drawing inspiration from these algorithms, DeMIMA implements the detection of aggregation relationships with static analyses and heuristics expressing common programming idioms, i.e., a collection is generally accessed through specific accessors to infer the type of stored instances. It assumes that these kinds of collections are homogeneous, i.e., containing instances with a common superclass different from `Object`. It is possible to determine their types by using well-known Java programming idioms such as pairs of `add()-remove()` accessors. DeMIMA also recognizes user-defined collections in addition to collections from the standard Java class libraries such as `Map`, `List`, and `Set` and their implementations. Recently, systems to convert programs to use generics have been proposed that could potentially solve the difficulty with untyped collections [40].

Detection of the values of the MU property also uses message receivers. For example, we assign value $[0, 1]$ to the MU property if the message receiver is `field`, `parameter`, or `local variable` and value $[0, +\infty]$ if the receiver is `array field`, `array parameter`, `local array`, `collection field`, `collection parameter`, or `local collection`.

The dynamic part—the EX and LT properties—of the composition relationship is difficult to detect due to the well-known limitations of dynamic analyses. We use a trace-analysis technique presented in [7] to compute, for each aggregation relationship, values of the exclusivity and lifetime properties and, if the values match, to convert it into a composition relationship. The results depend on the scenario executed; we assume the existence of unit tests and execute all available tests to infer values for the EX and LT properties. A low coverage by the unit tests would lead to a number of false negatives, i.e., candidate composition relationships missed by our algorithms. Missed compositions relationships impact DeMIMA by decreasing the number of *complete* occurrences of any design motifs including such relationships in their representations. However, thanks to the use of explanation-based constraint programming, DeMIMA would identify and report *approximate* occurrences corresponding to these motifs in which

composition relationships would be replaced by aggregation relationships and, thus, its recall would not be impacted.

4.2.6 $\mathcal{M}_I$ Construction in Summary

We formalized the definitions of the use, association, aggregation, and composition relationships and developed algorithms based on dynamic and static analyses to build a model $\mathcal{M}_I$ of a system from its source code model $\mathcal{M}_S$, thus creating the traceability link:

$$\mathcal{M}_S \overset{2}{\rightleftharpoons} \mathcal{M}_I.$$

Example. Fig. 2c shows how the UML-like model $\mathcal{M}_S$ is enriched into a model $\mathcal{M}_I$ by adding an aggregation relationship between `C1` and `C2`, instance of the `Aggregation` relationship class.

4.3 Third Layer: Design-Level Model, $\mathcal{M}_D$ ($\supset \mathcal{M}_{\mu A}$)

In the third layer, we first describe a model $\mathcal{M}_{DM}$ of a design motif with the same language used for $\mathcal{M}_I$. Then, DeMIMA looks for microarchitectures $\mathcal{M}_{\mu A}$ similar to the design motif DM in a model $\mathcal{M}_I$ of a system. To identify microarchitectures similar to $\mathcal{M}_{DM}$, it transforms $\mathcal{M}_{DM}$ into a constraints system. It then solves the constraint satisfaction problem using explanation-based constraint programming [8]. The solutions of the constraint satisfaction problem represent microarchitectures similar to $\mathcal{M}_{DM}$ in $\mathcal{M}_I$.

4.3.1 Modeling of Design Motifs

Parts 1, 2, and 3 in Fig. 3 show the language used to describe design motifs as first-class entities that can be manipulated programmatically. A design motif is represented by an instance of the class `DesignMotifModel` and is composed of `Participants`, each having different `Elements`.

Example. Fig. 2a shows the UML-like diagram of the model $\mathcal{M}_{DM}$ of the motif that we want to identify as well as instantiation links with some of the classes in Fig. 3.

4.3.2 Transformation of Design Motifs

With DeMIMA, the identification of microarchitectures similar to a design motif translates into a constraint satisfaction problem, which we list as follows:

- Variables correspond to the participants of the design motif model, $\mathcal{M}_{DM}$.
- Domains of the variables correspond to the entities of $\mathcal{M}_I$ in which to identify microarchitectures.
- Constraints among variables correspond to the relationships among the participants of $\mathcal{M}_{DM}$.

The transformation of a design motif into a constraint system requires dedicated constraints that represent relationships among participants. For example, constraint Strict Inheritance, in the case of Java-like single inheritance, creates a partial order on the set of entities and is satisfied for any couple (v_1, v_2) if the domain D_1 of v_1 represents a set of entities inheriting from the entities in the domain D_2 of v_2.

We proceed in a similar fashion for all relationships and define the following constraints:

- *Inheritance constraint*. The domains of two variables may contain the same entities, in contrast to strict inheritance.
- *Strict transitive inheritance constraint*. The domains of two variables contain entities that belong to the same branch of the inheritance tree.
- *Transitive inheritance constraint*. The domains of two variables contain entities that belong to the same branch of the inheritance tree or that are identical.
- *Use constraint*. The entities in the domain of variable v_1 use the entities in the domain of variable v_2.
- *Ignorance constraint*. This constraint explicitly states that two entities must not have any relationship.
- *Association constraint*. Association relationships link the entities in the domain of v_1 with the entities in the domain of v_2.
- *Aggregation constraint*. Aggregation relationships link the entities in the domain of v_1 with the entities in the domain of v_2.
- *Composition constraint*. Composition relationships link the entities in the domain of v_1 with the entities in the domain of v_2.
- *Creation constraint*. Entities in the domain of v_1 instantiate (at least once) entities in the domain of v_2.

We add standard (in)equality constraints to these constraints which ensure that different entities play different roles. We associate a weight with each constraint, an integer value $p \in \{1, 2, 3 \ldots, 100\}$, which indicates the relative importance of the constraints with one another or an order among constraints.

Example. The model $\mathcal{M}_{DM}$ of the motif of the running example transforms into a constraint system with two variables $v_{\mathtt{Z1}}$ and $v_{\mathtt{Z2}}$ corresponding to the classes `Z1` and `Z2` and the composition constraint $\mathtt{composition}(v_{\mathtt{Z1}}, v_{\mathtt{Z2}}, 100)$.

4.3.3 Resolution of the Constraint System

DeMIMA uses explanation-based constraint programming [8], [41] as a technique to solve constraint satisfaction problems translated from the identification of microarchitectures similar to design motifs. Explanation-based constraint programming justifies solutions, and lack thereof, of a constraint satisfaction problem by remembering constraints that can or cannot be satisfied. Explanation-based constraint programming is an extension of constraint programming in which the solver justifies its behavior at each step of the resolution process.

We implemented an explanation-based constraint resolution system dedicated to design motif identification reusing the JPALM [42] explanation-based constraint library. This extension includes a generic algorithm for the resolution of constraint satisfaction problems with explanations and a backtrack algorithm to manage contradiction.

Example. In the running example, no solution of the constraint system is found and, thus, no microarchitecture is identified and reported.

4.3.4 Relaxation of the Constraint System

Constraint relaxation consists of replacing the constraints that led to a contradiction with semantically weaker constraints.

As shown in Table 1 and from the formalizations of the binary class relationships, an order exists among the use, association, aggregation, and composition relationships. The properties of the use relationship are less constraining than those of the association relationship, which in turn are

less constraining than those of the aggregation relationship. Finally, the properties of the aggregation relationship are less constraining than those of the composition relationship. Inheritance-related constraints are also ordered from the most constraining to the least: strict inheritance, inheritance, strict transitive inheritance, and transitive inheritance.

We take advantage of these orders; for example, if a composition relationship between two entities prevents microarchitectures from being found, then this constraint can be replaced by an aggregation relationship between the same two entities. The microarchitectures found are semantically similar to the design motif model to the extent of the semantic similarity between the relationships. Problem relaxation is a special case of constraint relaxation in which no semantically weaker constraint is added to the constraint system.

DeMIMA enables experts to relax constraints and problems interactively as a guide in the identification of microarchitectures similar to a design motif. Relaxation is important because entities or relationships among entities in a model may differ from the expected entities and their relationships as defined in a design motif model. First, the solver searches for microarchitectures identical to a design motif model and provides maintainers with explanations of contradiction. A maintainer chooses one or more constraints which she believes are not essential to the design motif model and removes them from the constraint system dynamically, replacing them with semantically weaker constraints; the solver then searches for approximate microarchitectures. This process goes on until the maintainer decides that too many constraints have been relaxed and the microarchitectures are becoming too distant from the design motif model. Weights associated with each constraint are used to score a microarchitecture to help maintainers in choosing which constraints to relax. The score of a microarchitecture is

$$score = \left(\sum_{p\in\{p_1,\ldots,p_n\}} p\right) \times \left(\sum_{p\in\{p_j,\ldots,p_k\}} p/100\right),$$

where $\{p_1, \ldots, p_n\}$ is the set of weights of all constraints and $\{p_i, \ldots, p_j\}$ is the set of weights of the relaxed constraints. If all constraints from the design motif model are satisfied, then $score = 100$ else $score < 100$.

The solver may be automated to compute all combinations of constraint relaxations. The set of all possible microarchitectures (complete and approximate) is identical manually or automatically. This set only depends on the design motif and system models. The difference between automated and manual constraint relaxation is that maintainers may choose to relax constraints in a different order than that suggested by the design motif model and thus may direct the search more quickly toward useful microarchitectures.

Example. The composition constraint would be relaxed into an aggregation constraint $\texttt{aggregation}(v_{\texttt{Z1}}, v_{\texttt{Z2}}, \texttt{100})$ according to Table 1. A solution to this constraint system exists with $v_{\texttt{Z1}} = \texttt{C1}$ and $v_{\texttt{Z1}} = \texttt{C2}$.

4.3.5 $\mathcal{M}_D$ Construction in Summary

DeMIMA solves constraint satisfaction problems representing the identification of microarchitectures similar to $\mathcal{M}_{DM}$ in $\mathcal{M}_I$. Parts 1, 2, and 4 in Fig. 3 show the language used to describe models $\mathcal{M}_D$ and the set of models $\{\mathcal{M}_{\mu A}\}$ of microarchitectures similar to design motifs:

- The `MicroArchitecture` class describes microarchitectures similar to design motifs models. A microarchitecture model aggregates a set of entities which play a role in the microarchitecture. It also records the score of the solution and the set of relaxed constraints.
- An instance of class `ProgramModel` may contain instances of class `MicroArchitecture`.

Thus, DeMIMA can build models $\mathcal{M}_{\mu A}$ of microarchitectures identified as similar to $\mathcal{M}_{DM}$ models in $\mathcal{M}_I$ and ensure the traceability between their constituents:

$$\mathcal{M}_I \overset{3}{\rightleftharpoons} \mathcal{M}_D(\supset \mathcal{M}_{\mu A}).$$

Example. The model $\mathcal{M}_I$ is enriched by the microarchitecture corresponding to the found approximate solution into a model $\mathcal{M}_D$ shown in Fig. 2d.

5 Tooling

We implement DeMIMA on top of the PTIDEJ framework. The main programming language for the tools is Java. We use Prolog for the computation of the EX and LT properties and JPALM to implement the constraint solver to benefit from existing libraries. We present here only the components of the PTIDEJ framework relevant to DeMIMA:

1. PADL provides the language needed to describes models $\mathcal{M}_S$, $\mathcal{M}_I$, and $\mathcal{M}_D$ of systems. Its implementation is general enough to cope with different programming languages, such as C++ and Java.
2. The PADL CLASSFILE CREATOR parser analyzes the Java class files associated with a system to produce a model $\mathcal{M}_S$ of the system.
3. RELATIONSHIP STATIC ANALYSER computes values of the RT and MU properties and infers use, association, and aggregation relationships among entities of $\mathcal{M}_S$ to refine $\mathcal{M}_S$ into $\mathcal{M}_I$.
4. CAFFEINE performs dynamic analyses of a system to compute values for the EX and LT properties. Results are integrated within $\mathcal{M}_I$ to refine aggregation relationships into composition relationships if required.
5. PTIDEJ UI allows the visualization and refinement of $\mathcal{M}_S$, $\mathcal{M}_I$, and $\mathcal{M}_D$. It displays the models as UML-like class diagrams with a Sugiyama-based layout algorithm. It is also responsible to convert a chosen design motif $\mathcal{M}_{DM}$ into a constraint system and $\mathcal{M}_I$ into a domain for its variables.
6. Finally, the constraint solver PTIDEJ SOLVER is applied on the generated constraint satisfaction problem to solve the problem either interactively or automatically. The constraint solver produces microarchitectures $\mathcal{M}_{\mu A}$ similar to the design motif to create $\mathcal{M}_D$.

6 Experimentation

We apply DeMIMA to identify microarchitectures similar to several design motifs in both public domain and industrial systems. We analyze public domain systems because their

TABLE 3
Public Domain Systems Features

Systems	LOC	Design Characteristics					
		Classes	Methods	Uses	Associations	Aggregations	Inheritances
JHotDraw v5.1	4,587	191	2,824	1,006	1,454	292	234
JRefactory v2.6.34	36,358	512	8,804	3,736	5,004	765	570
JUnit v3.7	2,893	236	1,995	941	1,266	248	202
MapperXML v1.9.7	8,221	309	4,741	1,469	2,503	831	322
QuickUML 2001	5,117	293	3,187	1,312	2,380	583	277

From left to right: the name of the system, the total LOC, numbers of classes, methods, uses, associations, aggregations, and inheritances.

source code can be easily obtained and results can be compared with those of other researchers. We also analyze industrial systems for which we have both code and design. We used industrial systems because, to the best of our knowledge, no public domain system has both code and design available.

6.1 Subjects of the Experiment

We choose five well-known open source systems for our experiments, summarized in Table 3. JHOTDRAW [43] is a two-dimensional graphics framework for structured drawing editors. It includes several examples of editors, in particular a simple one to draw and color rectangles, circles, and texts. JREFACTORY is a tool that can perform several different refactorings on Java source files. It has been integrated in various IDE, including Sun's NetBeans. JUNIT is a unit-test framework developed to ease the implementation and running of unit tests for Java systems. MAPPERXML, a presentation framework for Web applications, is based on the Model-View Controller architectural pattern. Finally, QUICKUML is an object-oriented design tool that supports the design of a core set of UML models. The sizes of the open source systems range from about 2, 000 lines of code (LOC[5]) to about 36,000 LOC with a number of classes from about 200 to 500.

We also analyze 33 components from four complete industrial systems in the area of telecommunications developed by Sodalia, a medium-size company (about 250 programmers) Trento, Italy. Initially, it was a joint venture between Bell Atlantic and Telecom Italia; nowadays, Sodalia is an IT Telecom company belonging to Telecom Italia Group, the sector leader in Italy. Although the software engineering environment, tools, middleware, and general corporate culture can be considered uniform across projects, it is difficult to control all factors—especially the human factors. The 33 components were thus selected as representatives of the corporate system domain, the corporate skill, and the teams. Our analyses are performed at the component level, rather than the system level, because that is the level at which the design is generally documented and developers work. All components were documented with OMT class diagrams and developed by teams using C++ and the CORBA platform for distributed computing.

5. LOC is measured as the number of nonblank, noncomment lines including preprocessor directives.

The design class diagram and the final code were available for each component. The class diagrams were produced during the stage of detailed design. Classes almost always have a constructor (with void argument) and a destructor. Nested inner classes were not represented. Many, but not all, methods have their parameters specified in full detail. A few classes are completely unspecified (no attributes or methods). Thus, design class diagrams represent a mixture of high level and detailed design, perhaps closer to high-level design.

Component sizes range from a few hundred to about 50,000 lines of code, for a total of about 350 KLOC. The mean system size measured in LOC is 9,983 (standard deviation 11,578 LOC); design documents contain a fairly spread number of classes with a maximum of 113 and a minimum of 1 (mean value 17, standard deviation 20). Design documents have quite different levels of details; for example, the mean number of specified methods is 98; however, the method standard deviation is 95, i.e., there are designs specifying no methods. Table 4 presents detailed data on the 33 components.

6.2 Objects of the Experiments

In the following experiments, we use a set of well-known design patterns [2], identical to those used by Tsantalis et al., which includes Adapter, Command, Composite, Decorator, Factory Method, Observer, Prototype, Singleton, State, Strategy, Template Method, and Visitor. Contrary to Tsantalis et al., we distinguish between Composite and Decorator; however, we merge State and Strategy because their structures are identical. This choice is consistent with what previous authors did in the absence of semantic or behavioral data.

We only use the "canonical" representations of the design motifs because DeMIMA takes care of relaxations to find similar microarchitectures. In an interactive environment, a maintainer would direct the search by choosing the order of the relaxations. In the following experiments, we mimic the decision of the maintainer by limiting the number of relaxations to one per type of constraint (binary class relationships or inheritance relationships) and by imposing the next constraint depending on the design motif. For example, we permit an aggregation to be relaxed into an association but not into a use relationship. In the case of inheritance relationships, there are two possibilities, relax a strict inheritance constraint into an inheritance constraint or into a strict transitive inheritance constraint. Out of the 13 design motifs studied, it is our opinion that relaxing strict inheritance into inheritance makes sense only for the Abstract Factory and Observer motifs, while relaxing into a strict transitive inheritance is more suitable for the others. In the following, we report the number of microarchitectures identified as similar to each design motif according to the relaxations mentioned above. Microarchitectures are validated manually using the approach described in Section 6.3.

6.3 Performance, Accuracy, and Threat to Validity

For the purpose of design motif identification, with the present level of efficiency of Java environments (exploiting JIT compile technology), DeMIMA time execution and memory requirements are not an issue and, even for the largest systems in our subjects, the identification process requires resources that are compatible with the program comprehension process introduced in Section 2.

TABLE 4
Industrial Component Features

Components	LOC	Design Characteristics				
		Classes	Methods	Associations	Aggregations	Inheritances
C1	23,874	52	1,251	513	57	22
C2	24,016	52	1,255	503	57	22
C3	1,888	29	233	79	9	9
C4	10,966	16	621	68	10	9
C5	4,134	15	312	115	3	11
C6	33,320	106	1,750	905	36	83
C7	33,637	106	1,759	920	36	83
C8	32,025	102	1,686	903	28	79
C9	1,493	10	105	46	15	1
C10	11,026	30	337	190	2	31
C11	2,034	7	187	58	3	4
C12	19,301	28	696	335	35	7
C13	19,358	52	1,161	478	32	23
C14	17,760	87	1,406	564	30	54
C15	5,332	32	326	170	1	16
C16	7,719	18	331	182	8	12
C17	2,826	11	250	61	1	7
C18	5,240	4	106	39	3	0
C19	1,260	2	11	2	0	0
C20	22,280	12	138	6	0	3
C21	9,377	7	121	38	1	3
C22	330	7	72	22	3	3
C23	10,413	17	257	79	2	9
C24	7,605	12	232	68	0	5
C25	9,131	9	141	31	0	6
C26	3,152	4	91	15	1	1
C27	3,392	3	43	18	0	1
C28	2,220	6	76	44	0	1
C29	11,513	16	274	158	3	4
C30	1,421	3	76	13	0	1
C31	3,980	5	59	19	0	2
C32	4,285	12	163	117	5	9
C33	3,554	5	59	19	1	2

From left to right: the NAME of the component (component identifier), the total number of LOC, numbers of classes, methods, associations, aggregations, and inheritances.

All computations are performed on a AMD Athlon 64-bit processor running Microsoft Windows XP. We allocate a maximum of 800 megabytes of memory to the Java virtual machine. Computations take an average of 50 minutes to identify all of the microarchitectures similar to one given design motif in one given system.

We assess DeMIMA as an information retrieval system for which the most commonly used measures of accuracy are recall and precision [44]. Recall is the ratio of relevant documents retrieved for a given query over the number of relevant documents for that query in all of the given documents. Precision is the ratio of the number of relevant documents retrieved over the number of retrieved documents.

Although the number of relevant documents in all of the given documents (i.e., design motifs present in a given system) is not known a priori, we only need to assess the number of relevant documents retrieved for a given query (i.e., number of identified microarchitecture really implementing a given design motif) because we identify both complete and approximate microarchitectures. However, precision and recall depend on the accuracy of the static and dynamic analyses producing $\mathcal{M}_S$ and $\mathcal{M}_I$. Through relaxations, we ensure that we do not miss design motifs due to misclassifications of binary class relationships. However, more microarchitectures are identified: The approach ensures 100 percent recall for the five systems at the cost of a lower precision. The desired trade-off between precision and recall mostly depends on the maintainers' objectives: For the program comprehension task, we believe that perfect recall might be preferable because the maintainers do not want to miss any actual microarchitectures.

Three programs in our subjects were also studied in previous work [33]. Although, in theory, given a common problem and a benchmark, comparison should be feasible, it was not possible with this previous work due to the imprecision of the published data. To our surprise, we discovered errors in the results.[6] For example, in JHOTDRAW v5.1, the authors 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, they report `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 merely implements a `Command`-enabled button, encapsulating a given and unique `Command`, as confirmed by its documentation. Thus, we rely on other available results manually validated by independent software engineers of two different teams at two different research institutions [45], [46] and assembled for convenience in the publicly available P-MART database [47].

For the industrial components, the validation of the microarchitectures identified in the source code was performed manually, starting from the results of the identification process and assessing which of the microarchitectures implemented a design motif. A team of five independent software engineers performed the validation. Each time a doubt on a microarchitecture arose, they considered the design-pattern book [2] as the reference in deciding by consensus whether or not that microarchitecture implemented the design motif.

The average precisions by system vary mostly because of the design choices made in each system. JHOTDRAW is famous for the use of design patterns made by its authors. Thus, it is not surprising to have a higher precision than for the other systems because its design contains either microarchitectures that are very similar to design motifs or microarchitectures that are very dissimilar to design motifs. In the other systems, authors may have accidentally implemented some design motifs, thus producing designs

6. Tsantalis et al. [33] kindly provide their results at java.uom.gr/ nikos/ pattern-detection.html, last accessed on the 21 Feb. 2007.

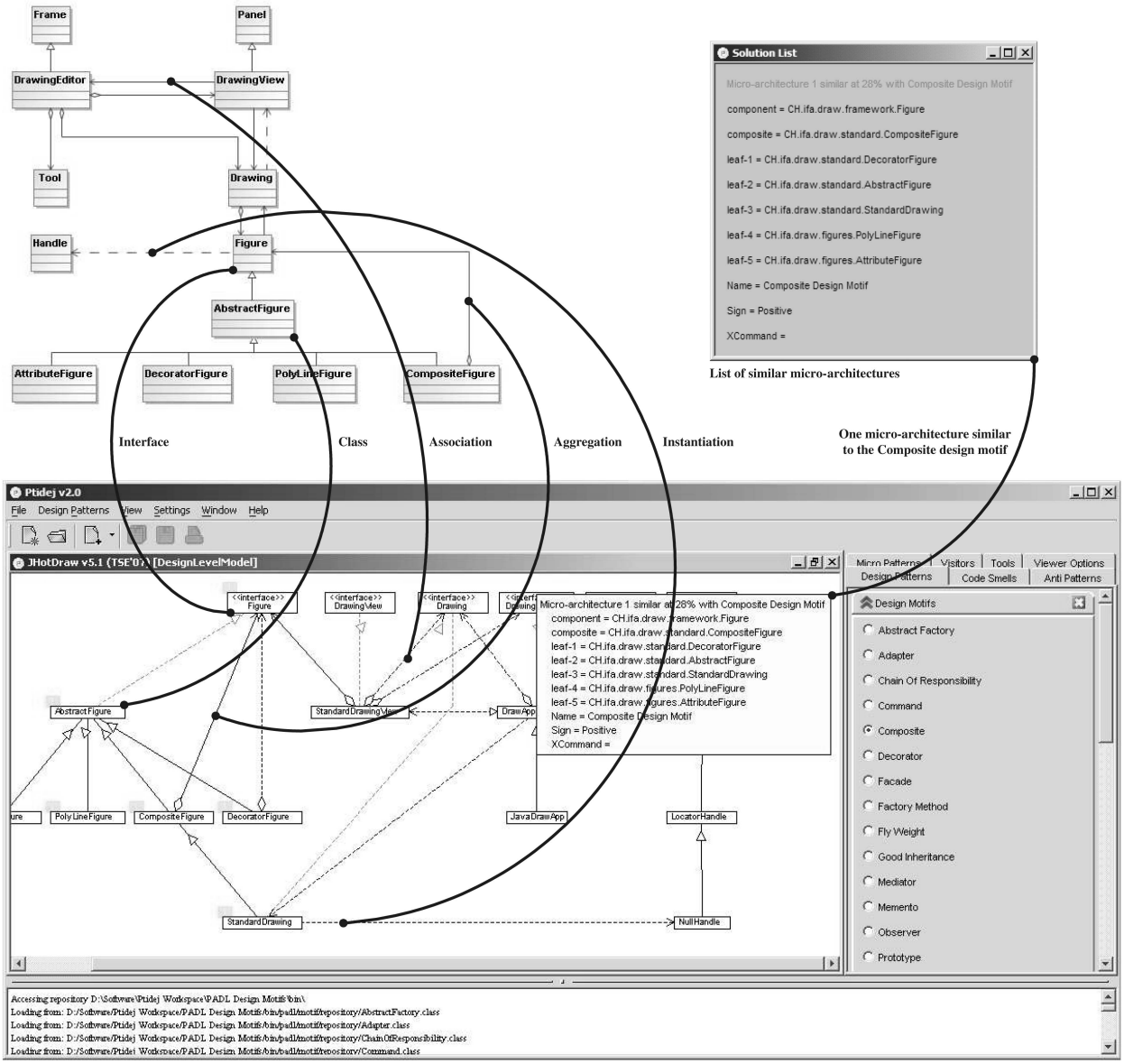

Fig. 4. Comparison of JHOTDRAW documented and recovered $\mathcal{M}_D$ model. The list and box show one selected $\mathcal{M}_{\mu A}$ similar to Composite.

where several microarchitectures are similar to design motifs yet do not implement their intents and motivations.

As described above and shown in Tables 3 and 4, our subjects are comprised of systems from different domains, complexities, and sizes. Thus, results reported below support the feasibility of DeMIMA and its ability to identify design motifs based on structural properties captured by AS, AG, and CO relationships and the set of defined constraints. Results are encouraging; future work will include studying generalization to other object-oriented programming languages, domains, and design motifs.

Internal validity is defined as the ability to detect a cause-effect relationship between independent and dependent variables. DeMIMA obviously detects design motifs and thus highlights microarchitectures to help program comprehension and documentation of reverse-engineered design choices; however, the extent to which these microarchitectures correspond to the intention or motivation of the developers has not been assessed and will be studied in future work.

6.4 A Step-by-Step Identification of Composite in JHotDraw

We perform a step-by-step identification of the Composite design motif in JHOTDRAW to illustrate the use of DeMIMA.

The top-left part of Fig. 4 shows a subset of the system design as presented in its documentation. We apply DeMIMA to build a model $\mathcal{M}_I$ of JHOTDRAW from its source code. Fig. 4 compares the recovered design-level model of the system and its documented design. The recovered model presents essentially the same data as the documented architecture. Some relationships among classes and interfaces differ because the authors of the documentation summarized the main classes and interfaces of the framework and reported against these entities some relationships existing only among their subclasses. For example, the instantiation relationship between interfaces `Figure` and `Handle` only exists between class `StandardDrawingView` (which implements `Figure`) and class `NullHandle` (which implements `Handle`). Thus, with

TABLE 5
Results of Design Motif Identification in Public-Domain Systems

	JHotDraw v5.1			JRefactory v2.6.34			JUnit v3.7			MapperXML v1.9.7			QuickUML 2001			Average Precision
Design Motifs	I	T	P	I	T	P	I	T	P	I	T	P	I	T	P	
Abstract Factory	0	0	100.0%	167	0	0.0%	27	0	0.0%	152	1	0.7%	37	2	5.4%	21.2%
Adapter	28	1	3.6%	47	17	36.2%	9	0	0.0%	18	1	5.6%	20	0	0.0%	9.1%
Command	11	1	9.1%	25	0	0.0%	10	0	0.0%	21	0	0.0%	4	1	25.0%	6.8%
Composite	3	1	33.3%	0	0	100.0%	1	1	100.0%	4	1	25.0%	8	3	37.5%	59.2%
Decorator	13	1	7.7%	1	0	0.0%	1	1	100.0%	1	0	0.0%	0	0	100.0%	41.5%
Factory Method	189	3	1.6%	71	1	1.4%	23	0	0.0%	46	1	2.2%	10	0	0.0%	1.0%
Observer	7	2	25.0%	1	0	0.0%	4	1	25.0%	9	1	11.1%	2	1	50.0%	22.9%
Prototype	2	2	100.0%	0	0	100.0%	0	0	100.0%	0	0	100.0%	2	0	0.0%	80.0%
Singleton	2	2	100.0%	14	2	14.3%	0	0	100.0%	3	3	100.0%	1	1	100.0%	82.9%
State/Strategy	21	6	28.6%	22	2	9.1%	8	0	0.0%	21	1	4.8%	9	0	0.0%	8.5%
Template Method	31	2	6.5%	49	0	0.0%	11	0	0.0%	21	4	19.0%	22	0	0.0%	5.1%
Visitor	0	0	100.0%	4	2	50.0%	0	0	100.0%	2	0	0.0%	0	0	100.0%	70%
Average Precision			43.2%			25.9%			43.8%			22.4%			34.8%	

DeMIMA, we obtain a model $\mathcal{M}_I$ of a system source code S and ensure the traceability between $\mathcal{M}_I$ and S:

$$S \rightleftharpoons \mathcal{M}_S \rightleftharpoons \mathcal{M}_I.$$

The Composite design motif [2, p. 163] defines three participants, Component, Composite, and Leaf, and three relationships among them, an inheritance between Component and Composite and between Component and Leaf and a composition between Composite and Component. It translates into the following constraint system: three variables, `component`, `composite`, and `leaf`, and three constraints:

- Two inheritance constraints between variables `leaf` and `component`, `composite` and `component`: `inheritance(component, composite, 100)` and `inheritance(component, leaf, 100)`.
- A composition constraint between variables `composite` and `component`: `composition(composite, component, 100)`.

DeMIMA solves the constraint satisfaction problem defined by the constraint system from the Composite design motif using as domain the JHOTDRAW idiom-level model. During the process, the composition constraint is relaxed because only aggregation relationships are present in the model $\mathcal{M}_I$ of JHOTDRAW; the inheritance constraints are also relaxed because an intermediate class, `AbstractFigure`, exists in the framework. Then, the identified microarchitectures are integrated in a design-level model. The bottom part of Fig. 4 shows the design-level model $\mathcal{M}_D$ of JHOTDRAW and, together with the top-right list, highlights a microarchitecture similar to the Composite design motif.

Thus, with DeMIMA, we obtain models $\mathcal{M}_{\mu A}$ similar to a design motif model $\mathcal{M}_{DM}$ in a model $\mathcal{M}_D$. DeMIMA also ensures the traceability between $\mathcal{M}_{\mu A}$, $\mathcal{M}_D$, and S:

$$S \rightleftharpoons \mathcal{M}_S \rightleftharpoons \mathcal{M}_I \rightleftharpoons \mathcal{M}_D \left(\supset \{\mathcal{M}_{\mu A}\}\right).$$

Models $\mathcal{M}_{\mu A}$ of microarchitectures similar to the Composite design motif help maintainers in understanding the design of the JHOTDRAW system by explaining the roles of the highlighted classes, which solve the problem of composing "objects into tree structures to represent part-whole hierarchies" and "let clients treat individual objects and compositions of objects uniformly," as defined by the Composite design pattern. Maintainers are guided by the identification in their comprehension of the system. Thus, DeMIMA may ease Task 3 of comprehending the system, as presented in Section 2.

6.5 Open Source Systems Case Studies

Table 5 gives the number of microarchitectures identified for each system in the public domain for each design motif. Columns labeled with I report detected motifs, with T microarchitectures manually classified as true motifs and with P the corresponding precision. It can be observed that the most frequently found design motifs are the Abstract Factory and Factory Method because they use characteristics at the core of object-oriented programming. The last row of Table 5 gives the precision of the design motif identification. Precision is computed over all motifs by summing the numbers of each column and then computing T/I, assuming a precision of 100 percent when $I = T = 0$.

In some cases, DeMIMA does not identify any microarchitecture similar to some design motifs. The reason is twofold: First, we only allow one approximation for each type of relationship; thus, it is possible that we do not

TABLE 6
Results of Design Motif Identification
in the Source Code of Industrial Components

	Abstract Factory	Adapter	Command	Composite	Decorator	Factory Method	Observer	Prototype	Singleton	State/Strategy	Template Method	Visitor
C1	5	4	0	0	0	0	2	0	–	7	7	0
C2	5	4	0	0	0	0	2	0	–	7	7	0
C3	2	1	0	0	0	0	0	0	–	1	1	0
C4	1	1	0	0	0	0	0	0	–	1	1	0
C5	7	1	0	0	0	0	0	0	–	1	1	0
C6	74	7	0	0	0	0	4	0	–	9	18	1
C7	72	7	0	0	0	0	4	0	–	9	18	1
C8	72	7	0	0	0	0	3	0	–	7	16	1
C9	2	0	0	0	0	0	0	0	–	1	1	0
C10	0	1	0	0	0	0	0	0	–	1	1	0
C11	0	0	0	0	0	0	0	0	–	0	1	0
C13	6	8	0	0	0	0	6	0	–	8	13	0
C14	180	16	0	2	0	0	3	0	–	18	19	0
C15	11	3	0	0	0	0	0	0	–	3	3	0
C16	60	6	0	3	0	0	3	0	–	7	7	1
C17	9	1	0	0	0	0	0	0	–	2	2	0
C22	1	1	0	0	0	0	0	0	–	2	2	0
C23	1	1	0	0	0	0	0	0	–	1	1	0
C24	1	1	0	0	0	0	0	0	–	1	1	0
C25	0	1	0	0	0	0	0	0	–	1	1	0
C29	2	1	0	0	0	0	0	0	–	1	2	0
C30	0	1	0	0	0	0	0	0	–	1	1	0
C31	1	0	0	0	0	0	0	0	–	1	1	0
C32	14	5	0	1	0	0	2	0	–	7	7	1
C33	1	0	0	0	0	0	0	0	–	1	1	0

identify a highly approximated form of the motif. However, this approximated form would very unlikely implement the motif and, thus, we do not affect the recall. Second, the systems are known for not containing such a design motif. For example, there is no Visitor in JHOTDRAW v5.1; two were implemented in later versions.

The average precisions by design motif vary because of the number of constraints and approximations. The more constraints and the fewer allowed relaxations, the higher the precision because DeMIMA does not report microarchitecture too far from the design motif of interest. For example, the Proxy design motif requires a method to return an instance of the declaring class, which is a strong constraint with respect to the five systems under study. In contrast, the Factory Method design pattern has low precision because many microarchitectures have structure similar to the structure of this design motif. The identification algorithms would require dynamic and semantic data to automatically distinguish true from false positives.

6.6 Industrial Systems Case Studies

With respect to the industrial systems, both design and code are analyzed. Design information has been recovered from the corporate database using the CASE2AOL TRANSLATOR

TABLE 7
Results of Design Motif Identification
in the Design of Industrial Components

	Abstract Factory	Adapter	Command	Composite	Decorator	Factory Method	Observer	Prototype	Singleton	State/Strategy	Template Method	Visitor
C1	3	2	0	0	0	0	0	0	–	2	2	0
C2	3	2	0	0	0	0	0	0	–	2	2	0
C5	1	0	0	0	0	0	0	0	–	1	1	0
C6	19	3	0	3	0	0	0	0	–	6	6	0
C7	19	3	0	3	0	0	0	0	–	6	6	0
C8	19	3	0	3	0	0	0	0	–	6	6	0
C11	2	0	0	0	0	0	0	0	–	1	1	0
C13	1	1	0	0	0	0	0	0	–	1	1	0
C14	5	1	0	2	0	0	0	0	–	3	3	0
C22	2	0	0	0	0	0	0	0	–	1	1	0
C31	2	0	0	0	0	0	0	0	–	1	1	0
C32	1	1	0	1	0	0	0	0	–	1	1	0
C33	2	0	0	0	0	0	0	0	–	1	1	0

[46] that we developed for StP/OMT. Tables 6 and 7 report the identified motifs in the source code and in the design, respectively. The tables do not include components where no microarchitecture was identified. A zero value means that no microarchitecture was identified for the corresponding design motif, while "-" means that the motif could not be searched for lack of available data.

Several observations can be made based on Tables 6 and 7. First, design motifs were not retrieved in several components, either in the design or in code: Design patterns seem seldom used.

Second, obtained results confirm results reported in previous work [46]. Due to a company takeover, lack of detailed documentation, and programmer turnover, only design patterns verified in previous work are verified, with a resulting precision of 100 percent. A full evaluation pertaining to the entire set of identified microarchitectures is unfortunately not feasible for the above reasons. Furthermore, for confidentiality reasons, we cannot distribute design or code and thus cannot report true positives and precisions computed by independent experts. Therefore, we only report the number of identified microarchitectures, not the precision and recall.

Third, a comparison of the microarchitectures identified in the design and those identified in source code shows that there is no intersection between these two sets: It would seem that different design motifs have been used in the design and in the implementation of the components. This fact can be partially explained by three reasons: First, when working with design, we do not have dynamic data so we cannot find composition relationships. Second, source code often includes a collection of classes reused from libraries or COTS that are not modeled in the design. Finally, our design documents are inconsistent with the source code: After code modifications, they were not properly updated to reflect the changes; hence the gap between design and code is relevant.

6.7 Discussion

Explanation-based constraint programming can assign entities to all the roles in a design motif, including, for example, the Client role or the Leaf role. Also, the identified microarchitectures contain more information than previous approaches such as [33] and [46]. In contrast with previous work, DeMIMA distinguishes microarchitectures similar to the Adapter and the Command design motifs because the constraint system locates entities playing the roles of Client and Invoker, which differentiate the two structural motifs.

The Singleton design motif must hold a single piece of information, its own unique instance. Nevertheless, in JREFACTORY, DeMIMA identified a microarchitecture similar to the Singleton but mapping unique instances of the Singleton with given objects. This variant of the Singleton is akin to the Identity Map described by Fowler in [48]. This accounts for the difference in the reported numbers between our work and the work by Tsantalis et al.

7 Conclusions

Microarchitectures similar to design motifs may help maintainers understand systems and ease their tasks. We introduced DeMIMA, a multilayered approach for design motif identification that defines

- simple class diagram constituents to build a model $\mathcal{M}_S$ of a system source code S,
- idiom-level constituents, in particular use, association, aggregation, and composition relationships to build a model $\mathcal{M}_I$ from $\mathcal{M}_S$, and
- microarchitectures similar to design motifs to enhance $\mathcal{M}_S$ into $\mathcal{M}_D$ with models $\mathcal{M}_{\mu A}$ of the microarchitectures.

In the second layer, DeMIMA depends on a set of definitions for unidirectional binary class relationships that we proposed and formalized. The formalizations define the relationships in terms of four language-independent properties that are derivable from static and dynamic analyses of systems: exclusivity, type of message receiver, lifetime, and multiplicity. DeMIMA keeps track of data and links to identify and ensure the traceability of these relationships.

In the third layer, DeMIMA uses explanation-based constraint programming to identify microarchitectures similar to design motifs. This technique makes it possible to identify microarchitectures similar to a model of a design motif without having to describe all possible variants explicitly.

We illustrated DeMIMA with the identification of microarchitectures similar to the Composite design pattern in the JHOTDRAW framework. We showed that the identified microarchitectures indeed highlight entities implementing the motif as documented by the authors of the system. We also applied DeMIMA on both open source and industrial systems and discussed its precision and recall.

In future work, we plan to improve our analyses of source code and integrate other sources of data such as sequence diagrams to enhance precision and identify behavioral and creational design motifs. We will also study object lifetime dependencies. We also plan to study the relation between identified microarchitectures and the concrete intent and motivation of software engineers. Finally, we would like to further assess the use of approximations in an automatic environment.

Acknowledgments

The authors thank Hervé Albin-Amiot for his work on the precursor of the PADL metamodel and Narendra Jussien for his help with explanation-based constraint programming. Giuliano Antoniol was partially supported by NSERC, Canada, research chair in software change and evolution. Yann-Gaël Guéhéneuc was partially supported by Object Technology International, Inc., an IBM Eclipse Fellowship Grant, and an NSERC Discovery Grant.

References

[1] Merriam-Webster, *Merriam-Webster Online Dictionary*, www.merriam-webster.com/, Mar. 2003.

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

[3] K. Beck and R.E. Johnson, "Patterns Generate Architectures," *Proc. Eighth European Conf. for Object-Oriented Programming*, M. Tokoro and R. Pareschi, eds., pp. 139-149, http://citeseer.nj.nec.com/27318.html, July 1994.

[4] C. Rich and R.C. Waters, *The Programmer's Apprentice*, first ed. ACM Press Frontier Series and Addison-Wesley, Jan. 1990.

[5] R. Wuyts, "Declarative Reasoning About the Structure of Object-Oriented Systems," *Proc. 26th Conf. Technology of Object-Oriented Languages and Systems*, J. Gil, ed., pp. 112-124, http://www.iam.unibe.ch/~wuyts/publications.html, Aug. 1998.

[6] Y.-G. Guéhéneuc and H. Albin-Amiot, "Recovering Binary Class Relationships: Putting Icing on the UML Cake," *Proc. 19th Conf. Object-Oriented Programming, Systems, Languages, and Applications*, D.C. Schmidt, ed., pp. 301-314, http://www.iro.umontreal.ca/ptidej/Publications/Documents/OOPSLA04.doc.pdf, Oct. 2004.

[7] Y.-G. Guéhéneuc, R. Douence, and N. Jussien, "No Java without Caffeine—A Tool for Dynamic Analysis of Java Programs," *Proc. 17th Conf. Automated Software Eng.*, W. Emmerich and D. Wile, eds., pp. 117-126, http://www.iro.umontreal.ca/~ptidej/Publications/Documents/ASE02.doc.pdf, Sept. 2002.

[8] Y.-G. Guéhéneuc and N. Jussien, "Using Explanations for Design-Patterns Identification," *Proc. First IJCAI Workshop Modeling and Solving Problems with Constraints*, C. Bessière, ed., pp. 57-64, http://www.iro.umontreal.ca/ptidej/Publications/Documents/IJCAI01MSPC.doc.pdf, Aug. 2001.

[9] J. Bansiya, "Automating Design-Pattern Identification," *Dr. Dobb's J.*, http://www.ddj.com/articles/1998/9806/9806a/9806a.htm?topic=patterns, June 1998.

[10] T. Richner and S. Ducasse, "Recovering High-Level Views of Object-Oriented Applications from Static and Dynamic Information," *Proc. Seventh Int'l Conf. Software Maintenance*, H. Yang and L. White, eds., pp. 13-22, http://www.computer.org/proceedings/icsm/0016/00160013abs.htm, Aug. 1999.

[11] D. Jackson and M.C. Rinard, "Software Analysis: A Roadmap," *Proc. 22nd Int'l Conf. Software Eng., Future of Software Eng. Track*, M. Jazayeri and A. Wolf, eds., pp. 133-145, http://sdg.lcs.mit.edu/%20dnj/talks/roadmap/, June 2000.

[12] P. Tonella and A. Potrich, "Reverse Engineering of the UML Class Diagram from C++ Code in Presence of Weakly Typed Containers," *Proc. Int'l Conf. Software Maintenance*, G. Canfora and A.A.A.-V. Maryhauser, eds., pp. 376-385, http://www.computer.org/proceedings/icsm/1189/11890376abs.htm, Nov. 2001.

[13] D. Thomas, "Reflective Software Engineering—From MOPS to AOSD," *J. Object Technology*, vol. 1, no. 4, pp. 17-26, http://www.jot.fm/jot/issues/issue_2002_09/column1/index.html, Sept. 2002.

[14] T. Gyimóthy, R. Ferenc, and I. Siket, "Empirical Validation of Object Oriented Metrics on Open Source Software for Fault Prediction," *IEEE Trans. Software Eng.*, vol. 31, no. 10, pp. 897-910, http://csdl2.computer.org/dl/trans/ts/2005/10/e0897.pdf, Oct. 2005.

[15] J.H. Jahnke, W. Schäfer, and A. Zündorf, "Generic Fuzzy Reasoning Nets as a Basis for Reverse Engineering Relational Database Applications," *Proc. Sixth European Software Eng. Conf.*, M. Jazayeri, ed., pp. 193-210, http://www.uni-paderborn.de/cs/varlet/docs.html, Sept. 1997.

[16] J. Niere, J.P. Wadsack, and A. Zündorf, "Recovering UML Diagrams from Java Code Using Patterns," *Proc. Second Workshop Soft Computing Applied to Software Eng.,* J.H. Jahnke and C. Ryan, eds., pp. 89-97, http://trese.cs.utwente.nl/scase/scase-2/Proceedings.pdf, Feb. 2001.

[17] J. Niere, J.P. Wadsack, and L. Wendehals, "Handling Large Search Space in Pattern-Based Reverse Engineering," *Proc. 11th Int'l Workshop Program Comprehension,* K. Wong and R. Koschke, eds., pp. 274-280, http://portal.acm.org/citation.cfm?id=857020, May 2003.

[18] D. Jackson and A. Waingold, "Lightweight Extraction of Object Models from Bytecode," *Proc. 21st Int'l Conf. Software Eng.,* D. Garlan and J. Kramer, eds., pp. 194-202, http://sdg.lcs.mit.edu/ dnj/, May 1999.

[19] Object Management Group, *UML v1.5 Specification,* http://www.omg.org/cgi-bin/doc?formal/03-03-01, Mar. 2003.

[20] J. Noble and J. Grundy, "Explicit Relationships in Object-Oriented Development," *Proc. 18th Conf. Technology of Object-Oriented Languages and Systems,* B. Meyer, ed., pp. 211-226, http://citeseer.nj.nec.com/noble95explicit.html, Nov. 1995.

[21] F. Civello, "Roles for Composite Objects in Object-Oriented Analysis and Design," *Proc. Eighth Conf. Object-Oriented Programming, Systems, Languages, and Applications,* A. Paepcke, ed., pp. 376-393, http://www.it.bton.ac.uk/staff/frc/papers/aboops93.html, Sept. 1993.

[22] S. Ducasse, M. Blay-Fornarino, and A.-M. Pinna-Dery, "A Reflective Model for First Class Dependencies," *Proc. 10th Conf. Object-Oriented Programming, Systems, Languages, and Applications,* F. Manola, ed., pp. 265-280, http://www.iam.unibe.ch/~ducasse/WebPages/Publications.html, Oct. 1995.

[23] L. Wills, "Automated Program Recognition by Graph Parsing," PhD dissertation, Massachusetts Inst. of Technology, 1992.

[24] J. Niere, W. Schäfer, J.P. Wadsack, L. Wendehals, and J. Welsh, "Towards Pattern-Based Design Recovery," *Proc. 24th Int'l Conf. Software Eng.,* M. Young and J. Magee, eds., pp. 338-348, http://portal.acm.org/citation.cfm?id=581382, May 2002.

[25] C. Krämer and L. Prechelt, "Design Recovery by Automated Search for Structural Design Patterns in Object-Oriented Software," *Proc. Third Working Conf. Reverse Eng.,* L.M. Wills and I. Baxter, eds., pp. 208-215, http://www.computer.org/proceedings/wcre/7674/76740208abs.htm, Nov. 1996.

[26] B. Kullbach and A. Winter, "Querying as an Enabling Technology in Software Reengineering," *Proc. Third Conf. Software Maintenance and Reengineering,* P. Nesi and C. Verhoef, eds., pp. 42-50, http://www.computer.org/proceedings/csmr/0090/00900042abs.htm, Mar. 1999.

[27] R.K. Keller, R. Schauer, S. Robitaille, and P. Pagé"Pattern-Based Reverse-Engineering of Design Components," *Proc. 21st Int'l Conf. Software Eng.,* D. Garlan and J. Kramer, eds., pp. 226-235, http://www.iro.umontreal.ca/~schauer/Private/Publications/icse1999/icse1999.html, May 1999.

[28] J.H. Jahnke and A. Zündorf, "Rewriting Poor Design Patterns by Good Design Patterns," *Proc. First ESEC/FSE Workshop Object-Oriented Reengineering,* S. Demeyer and H.C. Gall, eds., http://www.iam.unibe.ch/~famoos/ESEC97/, Distributed Systems Group, Technical Univ. of Vienna, UV-1841-97-10, Sept. 1997.

[29] G. Antoniol, R. Fiutem, and L. Cristoforetti, "Design Pattern Recovery in Object-Oriented Software," *Proc. Sixth Int'l Workshop Program Comprehension,* S. Tilley and G. Visaggio, eds., pp. 153-160, http://citeseer.nj.nec.com/antoniol98design.html, June 1998.

[30] J. Seemann and J.W. von Gudenberg, "Pattern-Based Design Recovery of Java Software," *Proc. Fifth Int'l Symp. Foundations of Software Eng.,* B. Scherlis, ed., pp. 10-16, http://www.informatik.uni-trier.de/~ley/db/indices/a-tree/s/Seemann:Jochen.html, Nov. 1998.

[31] D. Eppstein, "Subgraph Isomorphism in Planar Graphs and Related Problems," *Proc. Sixth Ann. Symp. Discrete Algorithms,* K. Clarkson, ed., pp. 632-640, www.ics.uci.edu/~eppstein/pubs/Epp-TR-94-25.pdf, Jan. 1995.

[32] N. Pettersson and W. Löwe, "Efficient and Accurate Software Pattern Detection," *Proc. 13th Asia Pacific Software Eng. Conf.,* P. Jalote, ed., pp. 317-326, http://ieeexplore.ieee.org/xpls/abs_all.jsp?isnumber=4137387&arnumber=4137433&count=65&index=43, Dec. 2006.

[33] N. Tsantalis, A. Chatzigeorgiou, G. Stephanides, and S. Halkidis, "Design Pattern Detection Using Similarity Scoring," *IEEE Trans. Software Eng.,* vol. 32, no. 11, Nov. 2006.

[34] K. Brown, "Design Reverse-Engineering and Automated Design Pattern Detection in Smalltalk," Technical Report TR-96-07, Dept. of Computer Science, Univ. of Illinois at Urbana-Champaign, http://citeseer.nj.nec.com/context/734211/0, July 1996.

[35] G. Hedin, "Language Support for Design Patterns Using Attribute Extension," *Proc. First ECOOP Workshop Language Support for Design Patterns and Frameworks),* J. Bosch and S. Mitchell, eds., Springer, pp. 137-140, http://www.cs.lth.se/Research/ProgEnv/LSDF.html, June 1997.

[36] H. Albin-Amiot and Y.-G. Guéhéneuc, "Meta-Modeling Design Patterns: Application to Pattern Detection and Code Synthesis," *Proc. First ECOOP Workshop Automating Object-Oriented Software Development Methods,* P. van den Broek, P. Hruby, M. Saeki, G. Sunyé, and B. Tekinerdogan, eds., http://www.iro.umontreal.ca/~ptidej/Publications/Documents/ECOOP01AOOSDM.doc.pdf, Centre for Telematics and Information Technology, Univ. of Twente, tR-CTIT-01-35, Oct. 2001.

[37] I. Philippow, D. Streitferdt, M. Riebisch, and S. Naumann, "An Approach for Reverse Engineering of Design Patterns," *Software and System Modeling,* vol. 4, no. 1, pp. 55-70, http://www.springerlink.com/content/0dn4pmqh5uhnbk69/, Feb. 2005.

[38] D. Heuzeroth, T. Holl, and W. Löwe, "Combining Static and Dynamic Analyses to Detect Interaction Patterns," *Proc. Sixth World Conf. Integrated Design and Process Technology,* H. Ehrig, B.J. Krämer, and A. Ertas, eds., http://www.info.uni-karlsruhe.de/publications.php/bib=281, June 2002.

[39] Y.-G. Guéhéneuc"A Systematic Study of UML Class Diagram Constituents for Their Abstract and Precise Recovery," *Proc. 11th Asia-Pacific Software Eng. Conf.,* D.-H. Bae and W.C. Chu, eds., pp. 265-274, http://www.iro.umontreal.ca/~ptidej/Publications/Documents/APSEC04.doc.pdf, Nov.-Dec. 2004.

[40] A. Donovan, A. Kiezun, M.S. Tschantz, and M.D. Ernst, "Converting Java Programs to Use Generic Libraries," *Proc. 19th Conf. Object-Oriented Programming Systems, Languages, and Applications,* D. Schmidt, ed., pp. 15-34, http://portal.acm.org/citation.cfm?id=1035292.1028979, Oct. 2004.

[41] N. Jussien and V. Barichard, "The PaLM System: Explanation-Based Constraint Programming," *Proc. Techniques for Implementing Constraint Programming Systems),* N. Beldiceanu, W. Harvey, M. Henz, F. Laburthe, E. Monfroy, T. Müller, L. Perron, and C. Schulte, eds., pp. 118-133, Sept. 2000, School of Computing, Nat'l Univ. of Singapore, tRA9/00.

[42] N. Jussien, "e-Constraints: Explanation-Based Constraint Programming," *Proc. First CP Workshop User-Interaction in Constraint Satisfaction,* B. O'Sullivan and E. Freuder, eds., http://www.emn.fr/jussien/publications/jussien-WCP01.pdf, Dec. 2001.

[43] E. Gamma and T. Eggenschwiler, "JHotDraw," http://members.pingnet.ch/gamma/JHD-5.1.zip, 1998.

[44] W.B. Frakes and R. Baeza-Yates, *Information Retrieval: Data Structures and Algorithms.* Prentice Hall, 1992.

[45] J. Bieman, G. Straw, H. Wang, P.W. Munger, and R.T. Alexander"Design Patterns and Change Proneness: An Examination of Five Evolving Systems," *Proc. Ninth Int'l Software Metrics Symp.,* M. Berry and W. Harrison, eds., pp. 40-49, http://csdl.computer.org/comp/proceedings/metrics/2003/1987/00/19870040abs.htm, Sept. 2003.

[46] G. Antoniol, G. Casazza, M. di Penta, and R. Fiutem, "Object-Oriented Design Patterns Recovery," *J. Systems and Software,* vol. 59, pp. 181-196, http://web.soccerlab.polymtl.ca/~antoniol/publications/index.html, Nov. 2001.

[47] Y.-G. Guéhéneuc, H. Sahraoui, and F. Zaidi, "Fingerprinting Design Patterns," *Proc. 11th Working Conf. Reverse Eng.,* E. Stroulia and A. de Lucia, eds., pp. 172-181, http://www.iro.umontreal.ca/~ptidej/Publications/Documents/WCRE04.doc.pdf, Nov. 2004.

[48] M. Fowler, *Patterns of Enterprise Application Architecture,* first ed. Addison-Wesley Professional, http://www.amazon.com/Patterns-Enterprise-Application-Architecture-Martin/dp/0321127420, Nov. 2002.

Yann-Gaël Guéhéneuc received the engineering diploma from the École des Mines of Nantes, France, in 1998 and the PhD degree in software engineering from the University of Nantes, France (under Professor Pierre Cointe's supervision) in 2003. His PhD thesis was funded by Object Technology International, Inc. (now IBM OTI Labs.) in 1999 and 2000. He is an assistant professor in the Department of Computing Science and Operations Research at the University of Montreal, where he leads the Ptidej team on evaluating and enhancing the quality of object-oriented programs by promoting the use of patterns at the language, design, or architectural levels. His research interests are program understanding and program quality during development and maintenance, in particular through the use and the identification of recurring patterns. He is also interested in empirical software engineering; he uses eye trackers to understand and to develop theories about program comprehension. He has published many papers in international conference proceedings and journals. He is a member of the IEEE.

Giuliano Antoniol received the degree in electronic engineering from the Università di Padova in 1982 and the PhD degree in electrical engineering from the École Polytechnique de Montréal, Canada, in 2004. He has worked in companies, research institutions, and universities. He is currently an associate professor at the the École Polytechnique de Montréal, where he works on software evolution, software traceability, software quality, and maintenance. He has published more than 100 papers in journals and international conference proceedings. He has served as a member of the program committees of international conferences and workshops such as the International Conference on Software Maintenance, the International Conference on Program Comprehension, and the International Symposium on Software Metrics. He is currently a member of the editorial board of the *Journal Software Testing Verification and Reliability*, the *Journal Information and Software Technology*, the *Journal of Empirical Software Engineering,* and the *Journal of Software Quality*. In 2005, he was awarded the Canada Research Chair Tier I in software change and evolution. He is a member of the IEEE.