Software Qual J
DOI 10.1007/s11219-009-9082-y

Improving design-pattern identification: a new approach and an exploratory study

Yann-Gaël Guéhéneuc · Jean-Yves Guyomarc'h · Houari Sahraoui

Abstract The identification of occurrences of design patterns in programs can help maintainers to understand the program design and implementation. It can also help them to make informed changes. Current identification approaches are limited to complete occurrences, are time- and resource-consuming, and lead to many false positives. We propose to combine a structural and a numerical approach to improve the identification of complete and incomplete occurrences of design patterns. We develop a structural approach using explanation-based constraint programming and we enhance this approach using experimentally built numerical signatures. We show that the use of numerical signatures improves the identification of complete and incomplete occurrences in terms of performance and precision.

Keywords Program understanding · Design patterns · Explanation-based constraint programming · Metrics · Exploratory study

1 Introduction

Design-pattern identification is the process by which occurrences of design motifs (Guéhéneuc and Antoniol 2008), the solutions of design patterns (Gamma et al. 1994), are identified in (the model of) a program. We present an exploratory study of an approach for the identification of complete and incomplete occurrences of design motifs in object-oriented source code.

Maintenance activities account for at least 50% of the total cost of programs (Boehm 1976; Koskinen 2004). Among these activities, maintainers spend more than half their time in the

Y.-G. Guéhéneuc (✉)
Ptidej Team, École Polytechnique de Montréal, Montréal, QC, Canada
e-mail: yann-gael.gueheneuc@polymtl.ca

J.-Y. Guyomarc'h · H. Sahraoui
GEODES, DIRO, Université de Montréal, Montréal, QC, Canada
e-mail: jy.guyomarch@gmail.com

H. Sahraoui
e-mail: sahraouh@iro.umontreal.ca

Published online: 14 August 2009

program-comprehension activity, trying to understand the source code of programs and, for object-oriented programs especially, their designs (Corbi 1989; Spinellis 2003; Wilde 1994).

Several authors suggested that the identification of *complete* occurrences of design motifs eases program comprehension, as shown in Sect. 2. A complete occurrence is a set of classes having structures and organisation identical to a design motif. Complete occurrences are very interesting during program comprehension, because they highlight classes that conform entirely to a design motif. However, they seldom exist in programs: developers rarely follow design motifs strictly because (1) they might use the motifs without knowledge of the actual patterns, (2) they might misinterpret the patterns and misuse their motifs, (3) they must adapt the structures and organisation of their classes to many other constraints.

Thus, *incomplete* occurrences of design motifs are also interesting during program comprehension. Incomplete occurrences of design motifs are sets of classes that participate in the solution of a design pattern while not strictly following the structures and organisation suggested by the motif. For example, in JHotDraw, the classes `Figure`, `CompositeFigure`, and `PolyLineFigure` form a micro-architecture similar to the composite design motif. This micro-architecture is an incomplete occurrence because its classes do not conform strictly to the motif: there is no direct inheritance relation between `Figure`, on the one hand, `CompositeFigure` and `PolyLineFigure`, on the other hand. An extra class, `Abstract Figure` has been inserted in the hierarchy.

The identification of incomplete occurrences is (1) costly in time, because of the large search space that includes all possible combinations of classes, which prohibits on-the-fly identification and (2) returns many false positives, impeding program comprehension and cluttering maintainers' cognitive capabilities. We propose the combination of two design-motif identification approaches—constraint satisfaction and comparison of metrics—and devise an exploratory study to assess the efficiency of our approach.

This study joins together our previous study on design-motif identification using explanation-based constraint programming, also known as DeMIMA (Guéhéneuc and Antoniol 2008), and on numerical signatures of design motifs (Guéhéneuc et al. 2004). Its contribution is to put together these two previous studies and to provide a validation of the benefit of combining the two studies.

1.1 Definition of the problem

A design pattern is a semi-formal description of a recurring design problem and of its solution. Design patterns described by Gamma et al. (1994) use textual notations and OMT-like class,[1] object, and interaction diagrams to describe solutions to design problems. The solutions described in design patterns are design motifs, prototypical micro-architectures from which developers draw inspiration to design, and to implement their programs.

Figure 1 shows a meta-model of design motifs. A design motif declares a set of roles, which will be played by classes in a program. These roles have one or more relations with other roles, such as inheritance or composition.

A micro-architecture is a set of classes, which have structures and organisation identical or similar to the roles defined in a design motif. Figure 2 sketches a meta-model to describe micro-architectures: a micro-architecture contains one or more roles (the same roles as the ones in the corresponding design motif) that are played by one or more classes.

When developers use design patterns to solve design problems, their resulting program designs potentially contain micro-architectures that are similar to some design motifs, i.e.

[1] We consider the concept of class extensively, including abstract classes, and interfaces.

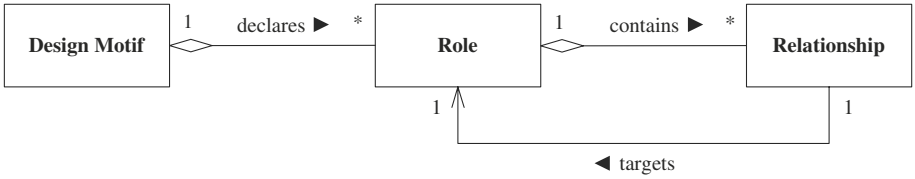

Fig. 1 Design motif meta-model

Fig. 2 Micro-architecture meta-model

contain *both* complete and incomplete occurrences of the motifs. We developed DeMIMA (Albin-Amiot et al. 2001; Guéhéneuc and Antoniol 2008; Guéhéneuc and Jussien 2001), an explanation-based constraint-programming approach to identify design motifs. This structural approach goes beyond existing approaches by identifying complete *and* incomplete occurrences.

The problem of identifying design motifs with any structural approach, however, is analogous to the problem of identifying similar sub-graphs in a graph, which is complex (see for example Antoniol et al. (1998) and Eppstein (1995)), resource consuming, and yields imprecise and incomplete results: DeMIMA has an average precision of 34% for a recall of 100%. Its precision varies greatly with the design motif, depending on the possibility to describe it precisely, for example Command (6.8%), Composite (59.2%), Decorator (41.5%), Observer (22.9%), Singleton (100%), and State (8.5%). One possible way to circumvent the limitations of structural identification is to apply heuristics that reduce the search space by eliminating a significant number of candidate classes.

We build heuristics in the form of numerical signatures common to classes playing roles in design motifs (Guéhéneuc et al. 2004). Numerical signatures for design motifs are analogous to biometric data for individuals: they allow efficient and automated elimination of false positives. The process of building numerical signatures consists of two activities. First, independent software engineers perform a manual identification of classes playing roles in design motifs in several object-oriented programs, confirming the developers' *intent* through a thorough study of the source code and documentation of the programs. Second, we infer numerical signatures linking classes playing confirmed roles in design motifs with the internal attributes of these classes using a machine-learning algorithm.

By their very construction, numerical signatures add semantics to the design-motif identification process because they are derived from a manual identification of micro-architectures similar to design motifs. They improve structural identification by eliminating classes that *obviously*, given the manual identification, cannot play a role in a given design motif. The identification of occurrences of design patterns is no longer limited by the large number of false positives, nor by the complexity of the structural identification.

1.2 Scope of the study

In the following, the term *occurrence* relates to either a *complete* or an *incomplete* occurrence. We present an exploratory study of the identification of complete and incomplete

occurrences of design motifs. The study compares our constraint-based approach with and without the use of numerical signatures. We show that the use of numerical signatures reduces the search space of the structural identification efficiently and significantly and, thus, improves the identification by preventing the identification of many false positives.

Section 2 presents previous work on design-motif identification and summarises the properties that an approach to design-pattern identification must exhibit. Section 3 introduces our original approach to the identification of design motifs using explanation-based constraint programming. Section 4 presents our experimental study of the construction of numerical signatures for roles in design motifs. Section 5 describes and discusses our exploratory study of the use of numerical signatures to reduce the search space and to increase the efficiency of design-motif identification with constraint programming. Section 6 concludes the presentation of our approach to design-motif identification and introduces a future study.

2 Previous study and desirable characteristics

There is a large body of study that proposes approaches to identify design motifs in source code (semi-)automatically. We summarise the main approaches, describe the properties that an approach to design-motif identification should exhibit, and show that none of the existing approaches possesses all these properties. Finally, we sketch our approach.

2.1 Previous study

Most previous approaches use a structural matching between a design motif and candidate micro-architectures. Different techniques have been used to perform the structural matching.

2.1.1 Unification

In his precursor study, Brown (1996) implemented algorithms to identify some design motifs in Smalltalk code using Smalltalk reflective capabilities. Wuyts (1998) developed the `SOUL` environment in which design motifs are described as Prolog predicates and program constituents (classes, methods, fields…) as facts. A Prolog inference algorithm unifies predicates and facts to identify classes playing roles in design motifs. Recently, Fabry and Mens (2004) used the LiCoR library, build on top of SOUL, to specify and identify design motifs. Other similar studies include Kramer's (1996). The main limitation of these approaches is the inherent combinatorial complexity of identifying subsets of all the classes matching design motifs (Eppstein 1995).

2.1.2 Queries

Bansiya (1998) introduced the DP++ tool to identify occurrences of some design motifs in C++ source code. The source code is compiled using Microsoft Visual Studio and ad hoc algorithms are implemented as queries over the intermediate code generated during the compilation. Other query-based approaches include (Ciupke 1999; Keller et al. 1999). Queries have the potential to be extremely fast (Beyer et al. 2005) but so far have been used only to specify motifs in a non-systematic way.

2.1.3 Constraint resolution

Quilici et al. (1997) used constraint programming to identify design motifs. Their approach consists of translating the problem of design-motif identification into a problem of constraint satisfaction. Design motifs are described as constraint systems for which the classes of a program form the domains of the variables. The resolution of the constraint systems provides micro-architectures consisting of classes representing the constraints among the roles of a design motif. As with the unification approach, the combinatorial complexity of the resolution proves to be prohibitive. Other such work exists (Guéhéneuc and Jussien 2001; Straw 2004).

2.1.4 Quantitative evaluation

Antoniol et al. (1998) used constraint programming extended with metrics to reduce the search space before design-motif identification. They designed a multi-stage filtering process to identify micro-architectures *identical* to design motifs. For each class of a program, they computed some metrics (for example, numbers of relations of inheritance, of association, and of aggregation) and they compared the metric values with expected values for a design motif to potentially exclude the class from the identification process and, thus, to reduce the search space. Then, they applied a constraint-based approach to identify micro-architectures. The expected values of the metrics are derived from the theoretical descriptions of design motifs. The main limitation of their study lies in the assumption that implementation (micro-architectures) accurately reflects theory (design motifs), which is often not the case. Guéhéneuc et al. (2004) recently built on Antoniol's approach.

2.1.5 Fuzzy reasoning

Jahnke amd Zündorf (1997) introduced fuzzy-reasoning nets to identify design motifs. Design motifs are described as fuzzy-reasoning nets, expressing rules of identification of micro-architecture similar but not identical to design motifs. They illustrated their approach with the identification of poor implementations of the `Singleton` design motif in legacy C++ code. They expressed identification rules with the formalism of fuzzy-reasoning nets and then computed the certainty of a class being a `Singleton` starting from a user's assumption. The main advantage of their approach is that fuzzy-reasoning nets deal with inconsistent and incomplete knowledge. However, their approach requires the description of all possible approximations of a design motif and users' assumptions.

2.1.6 Similarity scoring

Tsantalis et al. (2006) proposed an approach based on similarity scoring, which provides an efficient means to compute the similarity between the graph of a design motif and the graph of a program to identify classes potentially playing a role in the design motif. This approach is fast and has reasonable precision and recall. They illustrated their approach on three programs and 10 design motifs. Yet, although efficient in time, this approach is not interactive, does not explain its results, and only includes a limited set of approximations. Also, later work showed that its precision and recall are actually lower than presented (Guéhéneuc and Antoniol 2008).

2.1.7 Structural and dynamic analyses

Some authors, such as Heuzeroth et al. (2002), 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. Recently, Ng and Guéhéneuc (2007) introduced a trace analysis technique to identify occurrences of creational and behavioural design motifs but this approach can only work as a complement to a structural analysis.

2.2 Desirable characteristics

An approach to design-motif identification should combine the best of the approaches summarised in Table 1. Thus, it must:

1. *Identify incomplete* occurrences of design motifs in addition to complete occurrences, according to the maintainers' choices and context.
2. *Explain* why a micro-architecture is similar to a design motif, whether the micro-architecture is a complete or an incomplete occurrence.
3. *Interact* (if required and desired) with maintainers by allowing them to guide the identification of micro-architectures interactively.
4. *Be computationally efficient* to allow fast and on-line design-motif identification to help maintainers in their daily tasks, and not to slow down their work.
5. *Have good precision* to limit the number of spurious micro-architectures, which would require too much attention from maintainers.
6. *Have good recall* to limit the number of micro-architectures missed during the identification process, which would prevent maintainers from performing their tasks accurately.

Table 1 summarises the qualitative properties of existing approaches. None of these approaches to the identification of design motifs possesses all these properties

- Unification provides a convenient means to model both the design motif and the program in which to identify the motif. However, unification does not provide a convenient means to describe and to identify incomplete occurrences: it requires up-front the description of all possible incomplete occurrences. Moreover, unification identifies a large number of false positive micro-architectures and does not provide an easy way for maintainers to guide the search, and therefore has poor performance.
- Constraint programming uses a single model—a constraint system—to identify both complete and incomplete occurrences, using explanation-based constraint programming.

Table 1 Summary of the properties of current approaches to design-motif identification

	Unification	Constraints resolution	Fuzzy reasoning	Quantitative evaluation	Similarity scoring
Incomplete occurrences	No	No	Yes	Yes	Yes
Explanations	No	No	No	No	No
Interactions	No	No	No	No	No
Performance	Limited	Limited	Limited	High	High
Precision	Limited	Low	Low	Low	High
Recall	Limited	High	Low	Low	High

Maintainers can guide the identification. However, constraint programming does not solve the problems of performance, precision, and recall: identification takes many hours and provides false positive micro-architectures or misses false negative micro-architectures.
- Metric-based approaches offer an alternative to previous structural approaches. They are computationally efficient. They have, however, low precision and recall, because the metric values of classes are compared against theoretical metric values for roles rather than against experimentally validated values. Also, metric-based approaches do not allow maintainers to guide the search.
- Fuzzy logic uses the power of neural networks to describe design motifs and to perform design-motif identification. However, due to the very nature of neural networks, it is difficult to provide explanations for the identified micro-architectures and to allow interactions with maintainers during the search. Moreover, performance, precision, and recall are currently low and should be confirmed in future study.
- Dynamic analyses have the potential to improve the precision and recall of structural approaches but have not yet been studied in depth. Future study includes exploring the use of such dynamic analyses and empirically validating their benefits and disadvantages.

We propose an original combination of constraint programming and metric-based approaches for design-motif identification that possessess all required properties: identification of both complete and incomplete occurrences, explanations and interactions, satisfactory performance, and good precision and recall.

We use explanation-based constraint programming to identify, in programs, micro-architectures similar to design motifs, while providing explanations and interactions. We characterise the roles in design motifs experimentally with numerical signatures. Finally, we use the numerical signatures to reduce the size of the search space and to limit the number of false positives.

Our approach circumvents the problems of constraint programming by allowing incomplete occurrences, explanations, and interactions (detailed in Sect. 3). Also, it reduces the size of the search space by using numerical signatures characterised experimentally (detailed in Sect. 4 and evaluated in an exploratory study, in Sect. 5).

3 Design-motif identification process

We use explanation-based constraint programming to identify both complete and incomplete occurrences of design motifs while providing explanations and allowing interactions. This section provides all the details necessary to understand our approach and, thus, extends our previous presentation of DeMIMA in (Guéhéneuc and Antoniol 2008).

3.1 Explanation-based constraint programming

Explanation-based constraint programming has already proved its worth in many applications (Jussien and Barichard 2000). We recall the fundamentals of explanation-based constraint programming.

3.1.1 Contradiction explanations

We consider a constraint satisfaction problem (CSP) (V, D, C) where V is the set of variables, D is the set of domains for the variables, and C is the set of constraints

(constraint system) among variables. Decisions made during enumeration—variable assignments—are represented by unary constraints added to or removed from the current constraint system. These unary constraints are called *decision constraints* because they are not defined in the original constraint system but are generated by the solver to represent decisions taken during the resolution.

A *contradiction explanation* (also know as *nogood* (Schiex and Verfaillie 1994)) is a subset of the current constraint system that leads to a contradiction—no solution. A contradiction explanation divides in two parts: a subset of the original set of constraints ($C' \subset C$ in Eq. 1) and a subset of the decision constraints introduced during the search.

$$C \vdash \neg\, C' \wedge v_1 = a_1 \wedge \cdots \wedge v_k = a_k. \tag{1}$$

A contradiction explanation without a decision constraint denotes an over-constrained problem. In a contradiction explanation containing at least one decision constraint, we choose a variable v_j and rewrite Eq. 1 as Eq. 2.

$$C \vdash C' \wedge \bigwedge_{i \in [1\ldots k] \setminus j} (v_i = a_i) \rightarrow v_j \neq a_j. \tag{2}$$

The left-hand side of the implication is an *eliminating explanation* for the removal of value a_j from the domain of variable v_j. The eliminating explanation is denoted:

$$\texttt{expl}(v_j \neq a_j).$$

Classical solvers use domain-reduction techniques to solve constraint-satisfaction problems by removing values from the domains of variables. Thus, recording eliminating explanations is sufficient to compute contradiction explanations. Indeed, a contradiction is identified when the solver empties the domain of a variable v_j. A contradiction explanation can be computed with the eliminating explanations associated with each removed value, as shown in Eq. 3.

$$C \vdash \neg \left(\bigwedge_{a \in d(v_j)} \texttt{expl}(v_j \neq a) \right). \tag{3}$$

Several eliminating explanations generally exist for the removal of a given value. Recording all eliminating explanations would lead to an exponential space complexity. Thus, we must *forget* (erase) eliminating explanations that are no longer relevant to the current variable assignment. An eliminating explanation is said to be relevant if all its decision constraints are valid in the current search state (Bayardo Jr. and Miranker 1996). We keep only *one* explanation at a time for any value removal and the space complexity consequently remains polynomial.

In the context of design-motif identification, contradiction explanations include the constraints that could not be satisfied on the classes forming the domains of the variables. More details and examples are available in our previous study (Guéhéneuc and Jussien 2001).

3.1.2 Computing contradiction explanations

Minimal contradiction explanations (with respect to inclusion) are the most interesting. They provide data on dependencies among variables and constraints identified during the search. Unfortunately, computing such explanations is time-consuming (Junker 2001). A

compromise between size and computability consists of using the knowledge *inside* the solver. Indeed, solvers always know why they remove values from the domains of variables, although often not explicitly. They can compute minimal contradiction explanations with this knowledge.

3.2 Contradiction explanations and design-motif identification

The process of design-motif identification using explanation-based constraint programming can be divided in the following steps:

1. *Modelling a set of design motifs as CSP* A variable is associated with each class defined by a design motif. The variables of our model are integer-valued. The domain of a variable is a set of integers identifying existing classes in the source code uniquely. Relationships among classes (inheritance, association…) are represented by constraints among variables.
2. Modelling the maintainers' source code to keep only the data needed to apply the constraints: class names—forming the domains of the variables—and the relations among classes—verifying the constraints or not.
3. Resolving the CSP to identify both incomplete and complete micro-architectures: when all solutions to the CSP are found, i.e. when all micro-architectures identical to a design motif are identified, the resolution is guided by the maintainers to identify incomplete micro-architectures interactively. Contradiction explanations provided by the constraint solver can help the maintainers to guide the identification.

We build a library of specialised constraints from the relations among classes used to describe design motifs (Gamma et al. 1994). Specialised constraints express the relations of inheritance, creation, association, and so on among classes. Our library offers constraints covering a broad range of design motifs. However, some design motifs are difficult to express as CSP and require additional relations or the decomposition of existing relations into sub-relations. For example, some structural motifs include an important behavioural aspect, such as the Observer motif: in addition to the typical classes forming the motif and their inheritance and association relations, an important aspect of the motif is described in the behaviours of the `notify()` and `udpate()` methods. This difficulty is common to all structural and metric-based approaches.

For example, we provide the following constraints:

- `strictInheritance` and `inheritance` establish inheritance relations between two classes. A strict inheritance relation links two classes playing, respectively, the role of superclass and subclass. When considering single inheritance, the strict inheritance relation is a partial order, denoted $<$, on the set of classes V. For any pair of distinct classes `A` and `B` in V, if `B` inherits from `A` then: `A` $<$ `B`. The constraint associated with the strict inheritance relation is a binary constraint between variables `A` and `B`. From this definition of strict inheritance, we derive an inheritance relation, and its associated constraint, such that the variables may have for values the same class: `A` $<$ `B` or `A` = `B`.
- `association, aggregation`, and `composition` enforce the requirement that two classes are associated, aggregated, or composed with one another (Guéhéneuc and Albin-Amiot 2004), respectively. For example, a class `A` is composed with instances of a class `B` if the `A` class defines one or more fields of type `B`. Given three classes `A, B`, and `C`, this relation is:

- binary, such a relation links two and only two classes, i.e. `association(A,B)`;
- oriented, a relation from `A` to `B` does not imply that `B` is related to `A`, i.e. `association(A,B)`$\not\Rightarrow$`association(B,A)`;
- intransitive, a relation from `A` to `B` and from `B` to `C` does not satisfy the related constraint, i.e. `association(A,C)`.

3.3 Example of a CSP and its application

We describe a design motif as a CSP: each role is represented as a variable and relations among roles are represented as constraints among the variables. Additional variables and constraints may be added to improve the precision and recall of the identification process. Variables have identical domains: all the classes in the program in which to identify the design motif.

For example, the identification of micro-architectures similar to the `Composite` design motif, shown in Fig. 3, translates into the constraint system:

```
Variables:

  client
  component
  composite
  leaf

Constraints:

  association(client, component)
  inheritance(component, composite)
  inheritance(component, leaf)
  composition(composite, component)
```

where the four constraints represent the association, inheritance, and composition relations suggested by the `Composite` design motif. When applying this CSP to identify occurrences of `Composite` in JHotDraw (Gamma and Eggenschwiler 1998), the four variables `client`, `component`, `composite`, and `leaf` have identical domains.

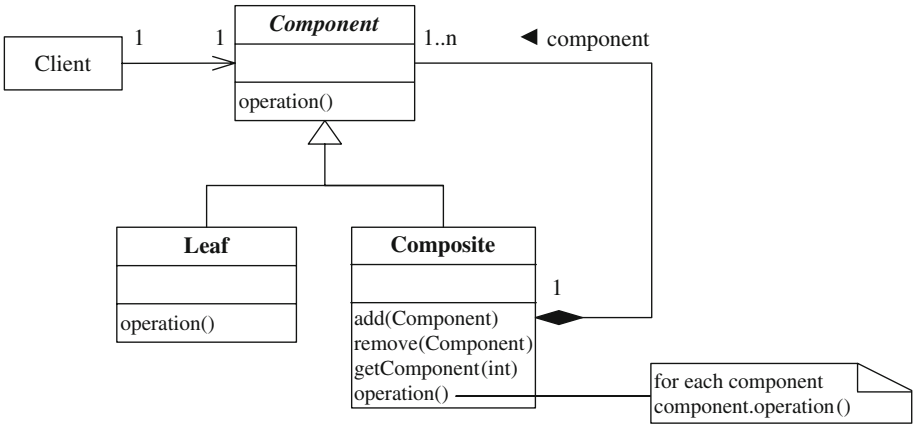

Fig. 3 `Composite` design motif

These domains contain all the 266 classes composing JHotDraw (Gamma and Eggenschwiler 1998):

$$\texttt{Domains: d(client)} = \texttt{d(component)} = \texttt{d(composite)} = \texttt{d(leaf)} = \{\ldots, \texttt{CH.ifa.draw.figures.AttributeFigure}, \ldots\}$$

3.4 Behaviour of the CSP solver

The library of specialised constraints cannot alone identify complete and incomplete occurrences of design motifs. We need a specialised explanation-based CSP solver to identify micro-architectures similar to design motifs.

Given a design motif expressed as a CSP, the specialised CSP solver computes complete occurrences first. The resolution ends by a contradiction, i.e. there are no more micro-architectures. Explanation-based constraint programming provides a contradiction explanation for this contradiction: the set of constraints justifying that other combinations of classes do not verify the constraints describing the design motif.

We do not need to relax constraints other than those provided by the contradiction explanation: we would find no additional micro-architectures. The explanation contradiction provides knowledge about available incomplete occurrences. This knowledge allows maintainers to lead the identification process towards interesting incomplete occurrences, from their viewpoint, by letting constraints relax. Removing a constraint suggested by a contradiction explanation does not necessarily lead to new micro-architectures, so the removal is applied iteratively.

A typical user session with our explanation-based constraint solver, JPtidejSolver, when looking for the `Composite` design motif in JHotDraw, is illustrated in Fig. 4. First, the problem and the domain of the variables are loaded and the result file is cleared. Second, the solver attempts to find complete occurrences. In this particular case, it cannot find any complete occurrence. Then, it provides the user with the set of constraints preventing complete occurrences to be found and requests the user to choose which constraint(s) to relax. In this case, first one constraint is shown, then four constraints. Relaxing the first constraint leads to finding one incomplete occurrence, where the composition relation between the roles `Composite` and `Component` (coded as `[#]`$\rightarrow$) is replaced by an aggregation relation (coded as `[]`$\rightarrow$). The interaction continues until no more incomplete occurrences can be found or the user chooses to relax no more constraints. Figure 4 shows both interactions and explanations.

3.5 Discussion of the identification process

The use of explanation-based constraint programming to identify micro-architectures similar to design motifs provides three interesting properties:

- Identification of both complete and incomplete occurrences of micro-architectures.
- Explanations about the identified micro-architectures.
- Interactions with the maintainers.

However, as with other structural approaches, our approach has limited performance and precision. Indeed, it is not possible to know, in general, if removing a constraint will lead to false positive occurrences. Only a maintainer may decide a priori that removing a constraint would lead to false positives or a posteriori by examining the obtained occurrences. It is possible that removing a constraint leads to both true and false occurrences.

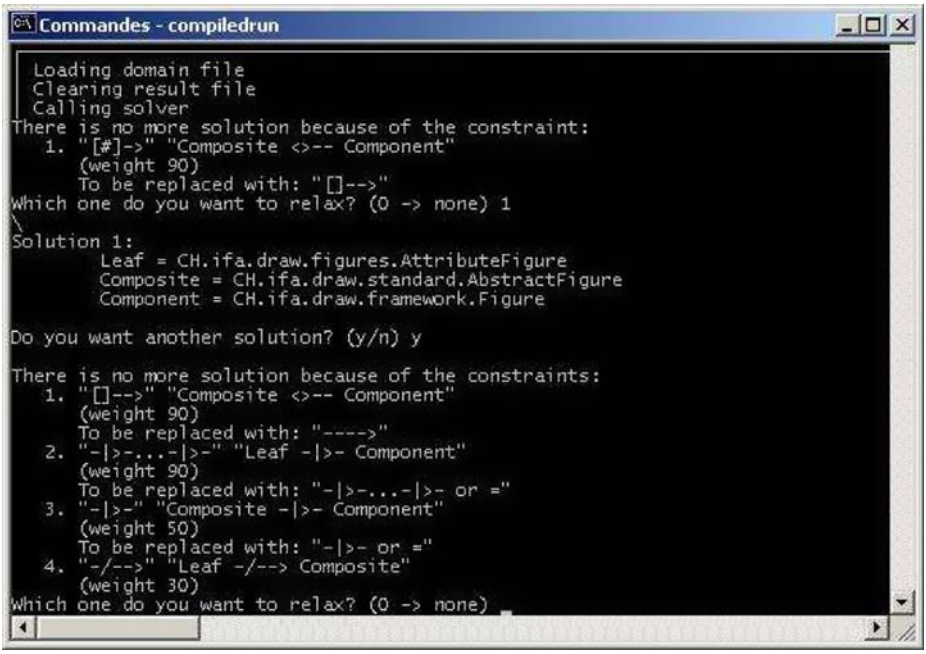

Fig. 4 User session with the design-motif identification process based on explanation-based constraint programming

We showed in our previous study (Guéhéneuc and Antoniol 2008) that, when favoring a 100% recall, our approach has an average precision of 34%. The reason of this limited precision is the potential number of micro-architectures: for example, the `Composite` design motif describes four roles, which are expressed as four variables. The identification of micro-architectures similar to the `Composite` design motif in the JHotDraw framework, which contains 266 classes, yields potentially $266^4 = 5{,}006{,}411{,}536$ micro-architectures. To reduce the search space and improve both performance and recall, we introduce numerical signatures associated with roles in design motifs.

4 Improving the identification process with numerical signatures

We seek to improve the performance and the precision of the structural identification process using quantitative values by associating numerical signatures with roles in design motifs. With numerical signatures, we can reduce the search space in two ways:

- We can assign to each variable a domain containing only those classes for which the numerical signatures match the expected numerical signatures for the role.
- We can add unary constraints to each variable to match the numerical signatures of the classes in its domain with the numerical signature of the corresponding role.

These two ways achieve the same result: they remove classes for which the numerical signatures do not match the expected numerical signature from the domain of a variable, reducing the search space by reducing the domains of the variables.

4.1 Numerical signatures

Numerical signatures characterise classes that play roles in design motifs. We identify classes playing roles in motifs using their internal attributes. We measure these internal attributes using the following families of metrics:

- Size/complexity, i.e. number of methods, of fields.
- Filiation, i.e. number of parents, number of children, and depth of the inheritance tree.
- Cohesion, i.e. degree to which the methods and attributes of a class belong together.
- Coupling, i.e. strength of the link between classes (through either use, association, aggregation, or composition relations).

We study the use of internal attributes of classes to quantify design-motif roles: we devise *numerical signatures* for design-motif roles using internal attributes of classes. We group these numerical signatures in rules to identify classes playing a given role. For example, a rule for the role of `Singleton` in the `Singleton` design motif could be:

```
Rule for ''Singleton'' role:
  Filiation: Number of parents low
             Number of children low
```

because a class playing the role of `Singleton` is normally high in the inheritance tree and has usually no (or only a few) subclasses. A rule for the role of `Observer` in the `Observer` design motif could be:

```
Rule for ''Observer'' role:
  Coupling: Coupling with other classes low
```

because the purpose of the `Observer` design motif is to reduce the coupling between the classes playing the roles of `Observer` and the rest of the program.

4.2 Building the numerical signatures

Figure 5 shows the process of assigning numerical signatures to design-motif roles. First, we build a *repository* of classes forming micro-architectures similar to design motifs in different programs. Roles played by classes in design motifs are identified manually. Then, we *extract metrics* from the programs in which we found micro-architectures to associate a set of values for the internal attributes to each class in the repository. Then, we feed a propositional *rule-learner* algorithm with the sets of metric values. The rule learner returns a set of rules characterising roles with the metric values of the classes playing these roles. We *cross-validate* the rules using the leave-one-out method, a *k*-fold cross validation where *k* equals the number of classes (Kohavi 1995; Stone 1974). Finally, we *interpret* the rules obtained (or lack thereof) for roles in design motifs. The following paragraphs detail each step of the process.

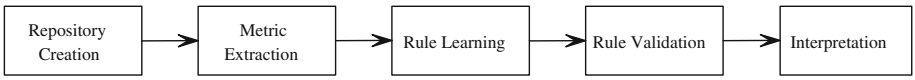

Fig. 5 Process of assigning numerical signatures to design-motifs roles

4.2.1 Repository creation

We need a repository of classes forming micro-architectures similar to design motifs to analyse these classes quantitatively. We built such a repository, PATTERN-LIKE MICRO-ARCHITECTURE REPOSITORY (P-MART[2]) over the course of the past years. We created this repository using different sources:

- Studies in the literature, such as the original study from Bieman et al. (2003), which recorded classes playing roles in design motifs from several different C++, Java, and Smalltalk programs.
- Our tool suite for the identification of design motifs, *Pattern Trace Identification, Detection, and Enhancement in Java* (PTIDEJ) (Albin-Amiot et al. 2001; Guéhéneuc and Albin-Amiot 2001), which implements JPTIDEJSOLVER, our explanation-based constraint solver to identify design motifs as described in Sect. 3.
- Assignments in undergraduate and graduate courses, during which students performed analyses of Java programs.

The repository of micro-architectures similar to design motifs contains, as explained in our previous study (Guéhéneuc et al. 2004) (which includes a technical description of the file format):

- For each program, design motifs for which we found similar micro-architectures.
- For each design motif, similar micro-architectures that we found in the program.
- For each micro-architecture, roles played by their classes in the corresponding design motif.

We validated all the micro-architectures manually before their inclusion in the repository. We do not claim that we have identified *all* micro-architectures similar to design motifs in a given program, but are confident that we found *most* of them through repetitive manual analyses of the same programs by different teams of undergraduate and graduate students. Each team was assigned the task of identifying occurrences of the 23 motifs in Gamma et al. (1994). Each identified occurrence was evaluated by at least one other team. If both the teams agreed (possibly after discussions), the occurrence was kept, else the occurrence was discarded.

As of October 2008, P-MART contains data from nine programs, for a total of 4,376 classes[3] and 138 micro-architectures representing 19 different design motifs. We record this data in an XML tree, which allows us to traverse the data to compute metrics and various statistics automatically.

Table 2 summarises the data in P-MART. The two first rows give the names and number of classes of the surveyed programs. The following rows indicate, for a given design pattern (per row), the number of micro-architectures found similar to its design motif in each program (per column). The table also summarises the number of roles defined by a design motif and the number of classes playing a role in a design motif for all the programs (two last columns).

4.2.2 Metric extraction

We parse the programs surveyed in P-MART and calculate metrics on their classes automatically. Parsing and calculation are performed in a three-step process: first, we build a

[2] See www.ptidej.net/downloads/pmart/.

[3] We excluded local classes, i.e. classes defined in methods, because of their rarity.

Table 2 Overview of the data set: programs, design motifs, micro-architectures, and roles

Numbers of classes: / Design motifs	JHotDraw v5.1 155 Numbers of micro-architectures	JRefactory v2.6.24 569	JUnit v3.7 79	Lexiv 0.0.1α 217	MapperXML v1.9 172	Nutch v0.4 2,558	PMD v1.8 447	NetBeans v1.0.x 24	QuickUML 2001 155	Total 4,376	Numbers of roles (Gamma et al. 1994)	Numbers of classes playing a role
Abstract Factory					1	12			1	14	5	242
Adapter	1	17			2	8	2	1		31	4	252
Bridge							2			2	4	25
Builder		2		1				2	1	6	4	44
Command	1					1	2		1	5	5	85
Composite	1		1		1			2	2	7	4	147
Decorator	1		1							2	4	64
Facade					1					1	2	11
Factory Method	3	1			1			3		8	4	111
Iterator			1			5	1	1		8	5	41
Memento								2		2	3	15
Observer	2		3	2	1			2	1	11	4	135
Prototype	2									2	3	32
Proxy								1		1	3	3
Singleton	2	2	2	2	3		1		1	13	1	13
State	2	2								4	3	32
Strategy	4				1		2			7	3	47
Template Method	2				4		3	1		10	2	102
Visitor		2						1		3	5	143
									Total	138	70	1552

Table 3 External attributes for classes and corresponding metrics

	Acronyms	Descriptions and references
Size/complexity	NM	Number of methods (Lorenz and Kidd 1994)
	NMA	Number of new methods (Lorenz and Kidd 1994)
	NMI	Number of inherited methods (Lorenz and Kidd 1994)
	NMO	Number of overridden methods (Lorenz and Kidd 1994)
	WMC	Weighted methods count (Chidamber and Kemerer 1993)
Filiation	CLD	Class-to-leaf depth (Tegarden et al. 1995)
	DIT	Depth in inheritance tree (Chidamber and Kemerer 1993)
	NOC	Number of children (Chidamber and Kemerer 1993)
Cohesion	C	Connectivity 'C' (Hitz and Montazeri 1995)
	LCOM5	Lack of cohesion in methods 5 (Briand et al. 1997b)
Coupling	ACMIC	Ancestors class-method import (Briand et al. 1997a)
	CBO	Coupling between object (Chidamber and Kemerer 1993)
	DCMEC	Descendants class-method export (Briand et al. 1997a)

model of a program using the *Pattern and Abstract- level Description Language* (PADL) meta-model and its parsers; second, we compute metrics using *Primitives, Operators, Metrics* (POM), an extensible framework for metric calculation based on PADL; third, we store the results of the metric calculation, names and values, in P-MART, by adding specific attributes and nodes to the XML tree representation.

We use metrics from the literature to associate values with internal attributes of classes playing a role in a design motif. Table 3 presents the metrics: for size/complexity, we use the metrics by Lorenz and Kidd (1994) on new, inherited, and overridden methods and on the total number of methods, and the count of methods weighted with the number of method invocations by Chidamber and Kemerer (1993). We do not use metrics related to fields because no design motif role is characterised by fields specifically: only the `Flyweight`, `Memento`, `Observer`, and `Singleton` design motifs (5 out of 23) expose the structures of some roles to exemplify typical implementation choices. Moreover, fields should always be private to their classes with respect to the principle of encapsulation. For filiation, we use the depth in the inheritance tree, the number of children (Chidamber and Kemerer 1993), and the number of hierarchical levels below a class, class-to-leaf depth (Tegarden et al. 1995). For cohesion, we use the 'C' metric measuring the connectivity of a class with the rest of a program (Hitz and Montazeri 1995) and the fifth metric on lack of cohesion in methods (Briand et al. 1997b). Finally, for coupling, we use two metrics on class-method import, export coupling (Briand et al. 1997a), and the metric on coupling between objects (Chidamber and Kemerer 1993).

4.2.3 Rule learning and validation

We use a machine-learning algorithm to find commonalities among classes playing the same role in a design motif in P-MART. We supply the data to a propositional rule-learner algorithm, JRIP, implemented in WEKA, an open-source program collecting machine-learning algorithms for data-mining tasks (Witten and Frank 1999).

We do not provide JRIP with all the data in P-MART because the disparities among roles, classes, and metric values would lead to uninteresting results. We provide JRIP with

subsets of the data related to each role. A subset σ of the data contains the metric values for the n classes playing a role (positive examples) in all the micro-architectures similar to a design motif. We add to this subset σ the metric values of $3 \times n$ classes *not* playing the role (negative examples), chosen randomly in the rest of the data. We make sure the classes chosen randomly have the expected structure for the role to increase their likeness with the classes playing the role. The rule learner infers rules related to each role from the subsets σ. We validate the rules using the leave-one-out method with each set of metric values in the subsets σ (Kohavi 1995; Stone 1974).

4.2.4 Rule interpretation

The rule learner infers rules that express the experimental relations among metric values, on the one hand, and roles in design motifs, on the other. Typically, a rule inferred by the rule learner for a role `ROLE` has the form:

`Rule for ''ROLE'' role:`

- `Numerical signature 1, confidence 1,`
- `Numerical signature 2, confidence 2,`
- `...`
- `Numerical signature`N`, confidence`N.

where

$$\texttt{Numerical signature 1} = \{metric_1 \in V_{11}, \ldots, metric_m \in V_{m1}\}$$
$$\ldots$$
$$\texttt{Numerical signature } N = \{metric_1 \in V_{1n}, \ldots, metric_m \in V_{mn}\}$$

and the values of a metric $metric_i$ computed on classes playing the role `ROLE` belong to a set $V_{ij} \subset \mathbb{N}$. The degree of `confidence`$K$ is the number of classes concerned by a numerical signature in a subset σ, which we use to compute error and recall ratios.

We collect all rules inferred from the rule learner and process the rules with the following criteria to remove uncharacteristic rules:

- We remove rules with a recall ratio less than 75% to ensure that the rule characterises classes much better than random chance.
- We remove rules inferred from small subsets σ, i.e. those for which not enough classes play a given role.

4.3 Discussion of the numerical signatures

We decompose the data in P-MARt into 56 subsets σ and infer as many rules with the rule learner, which decompose in 78 numerical signatures. The two first steps in the analysis process are quantitative and aim at eliminating roles that do not have a sufficient number of examples for mining numerical signatures and that do not have a high enough recall ratio. In the first step, we remove 20 of the 56 rules from the rules inferred by the rule learner. The removed rules correspond to:

- Design motif roles with few corresponding micro-architectures and with a unique (or only a few) classes in the micro-architectures. Some examples are the roles of `Decorator` in the `Decorator` design motif and of `Prototype` in `Prototype`.

Table 4 Roles with inferred rules with recall ratio greater than 75%

Design motifs	Roles	False positives[a] (%)	Errors[b] (%)	Recalls (%)
Iterator	Client	0.00	0.00	100.00
Observer	Subject	0.00	0.00	100.00
Observer	Observer	2.38	6.67	100.00
Template Method	Concrete Class	0.00	0.00	97.06
Prototype	Concrete Prototype	0.00	0.00	96.30
Decorator	Concrete Component	4.17	12.24	89.58
Visitor	Concrete Visitor	0.00	0.00	88.89
Strategy	Context	3.70	11.11	88.89
Visitor	Concrete Element	2.04	6.45	88.78
Singleton	Singleton	8.33	22.22	87.50
Factory Method	Concrete Creator	4.30	12.90	87.10
Factory Method	Concrete Product	3.45	10.71	86.21
Adapter	Target	4.00	12.50	84.00
Composite	Leaf	6.47	19.12	82.09
Decorator	Concrete Decorator	0.00	0.00	80.00
Iterator	Iterator	0.00	0.00	80.00
Command	Receiver	6.67	20.00	80.00
State	Concrete State	6.67	20.00	80.00
Strategy	Concrete Strategy	2.38	8.33	78.57
Command	ConcreteCommand	3.23	11.11	77.42

[a] Refer footnote 5

[b] Refer footnote 6

– Design motifs' roles played by "ghost" classes, i.e., classes known only from import references, such as classes in external libraries. Some examples are the classes playing the roles of `Command` in the `Command` design motif and of `Builder` in `Builder`.

In the second step, we select the 20 rules with a recall ratio greater than 75%, shown in Table 4, from the 36 remaining rules. All these rules exhibit false positives, with a ratio[4] less than 10% (less than 5% for 16 of them). These rules have errors[5] less than 23% (less than 10% for half of them). The differences between the ratios of misclassified false positives and errors illustrate that, disregarding the size of σ, the rules do not overly misclassify counter-examples.

Most of the rules removed because of their low recall ratios concern non-key roles in design motifs, i.e. that theoretically do not have a particular numerical signature. For example, any class may play the role of `Client` in the `Composite` design motif. Similarly, any class may play the role of `Invoker` in the `Command` design motif. (For some researchers, `Client`, `Invoker`... are not *real* roles and are not be taken into account in the design motifs, see for example (Tsantalis et al. 2006).)

[4] The ratio of misclassified false positives is (*number of counter-examples in σ classified as playing a role*)/(*number of counter-examples in σ*).

[5] The error is computed as (*number of examples in σ classified as playing a role*)/(*size of σ*).

Table 5 Rules inferred for the role of `Target` in the `Adapter` design motif

`Rule ''Target''` $-WMC \Leftarrow 2, 24/25.$

In many cases, we obtained a unique numerical signature for a given role in a design motif. Classes playing the same role have similar structures and organisations generally. For example, all the classes playing the role of `Target` in the `Adapter` design motif have a low complexity, represented by low values of WMC, as shown in Table 5 (the degree of confidence is <1 because this numerical signature misclassifies one class; its error rate is 4%, as shown in Table 4). Such a low complexity is expected because of the behaviour suggested by the `Adapter` design motif. Likewise, many other numerical signatures confirm claims and beliefs about design motifs. For example, classes playing the role of `Observer` in the `Observer` design motif have low coupling, i.e. a low CBO. Classes playing the roles of `Singleton` in the `Singleton` design motif have low coupling and generally belong to the upper part of the inheritance tree.

In a few cases, we obtain more than one numerical signature for a role. An example is the role of `ConcreteVisitor` in the `Visitor` design motif. The most frequent numerical signatures characterise classes with low coupling (low CBO) and a large number of methods (high NM), as expected from the problem dealt with by the `Visitor` design motif. The second numerical signature states that the number of inherited methods is low (low NMI) for some classes playing the role of `ConcreteVisitor`. When exploring the micro-architectures similar to the `Visitor` design motif in our repository, we notice that, in JRefactory, some classes play the roles of both `ConcreteVisitor` and `Visitor`, which limits the number of inherited methods. This second numerical signature is particular to one program, thus unveiling design choices specific to the program or to a coding style.

Numerical signatures cannot be used to identify design motifs alone. Indeed, two or more classes may play an identical role in different uses of a design motif and the same class may play two or more roles in one or more design motifs. They define necessary but not sufficient conditions for classes. For example, a potential `Target` in the `Adapter` design motif must not be complex according to the rule in Table 5. However, any non-complex class is not a `Target`.

5 Exploratory study

The numerical signatures can help to improve the identification process of both complete and incomplete occurrences in terms of performance and precision, by removing from the domains of the variables the classes for which metric values do not match the expected numerical signatures, i.e. by removing the classes that do not obviously play a role in the design motif. Thus, the identification process can:

- Be computationally efficient, with a reduced search space.
- Have good precision through the removal of classes that do not play a role in the design motif.
- Keep the perfect recall favoured in our approach, so that maintainers can focus on a small number of interesting micro-architectures.

We now validate the new identification process combining both explanation-based constraint programming and numerical signatures. The objective of this exploratory study

is to assess whether or not search-space reduction using numerical signatures improves the performance and the precision of the constraint-based identification. We do not evaluate the constraint-based identification per se because we do not explicitly assess the validity of the occurrences identified by either the original or the enhanced version of the constraint-based identification process.

5.1 Assumptions

This exploratory study is based on two main assumptions. The first assumption for the applicability of our approach is the availability of enough known micro-architectures from which to derive the numerical signatures. The second assumption is that the design motif roles have distinguishing structures and organisations. Consequently, the explorational setting is chosen in the context of these assumptions. Also, the results will be further discussed in Sect. 5.6 bearing these assumptions in mind.

5.2 Research questions

To assess whether our approach indeed improves the identification process, we need to answer the following two research questions:

RQ1: *Does the use of numerical signatures reduce significantly the time of identifying occurrences of design motifs?* The tasks of computing metrics for each class, and verifying if the obtained values match numerical signatures, take time. This time must be significantly lower than the time we gain by reducing the search space.

RQ2: *Does the use of numerical signatures reduce the number of false positives while preserving all occurrences?*We must ensure that numerical signatures exclude from the search space only classes *not* playing a role in a design motif, thus improving the precision while maintaining the perfect recall.

We answer these questions through an explorational protocol that we instantiate for nine programs and three design motifs.

5.3 Protocol

Let t_{cp} and t_{cp+ns} be the identification times using, respectively, explanation-based constraint programming (cp) and explanation-based constraint programming enhanced with numerical signatures (cp + ns). The time t_{cp+ns} includes the time required to reduce the size of the domains of the variables by applying the numerical signatures. Each identification process produces a set of candidate occurrences, o_{cp} and o_{cp+ns}, respectively. We propose the following protocol to answer the two previous research questions:

1. First, we identify all the occurrences of a design motif in a program P, using the constraint-based approach (with the set of all classes as domains for the variables). As results, we obtain t_{cp} and o_{cp}.
2. Second, we identify the occurrences of the design motif using the enhanced approach (i.e. reducing the domain of each variable using numerical signatures associated with the role). We thus obtain t_{cp+ns} and o_{cp+ns}. The search-space reduction does not yield more occurrences than those identified by the original algorithm.

3. Third for RQ1, we compute the difference between the identification times $t_d = t_{cp+ns} - t_{cp}$ to assess whether the difference is significant.
4. Fourth for RQ2, we compute the set difference $o_e = o_{cp} - o_{cp+ns}$ containing all the occurrences excluded from the identification process thanks to the numerical signatures. We check manually if each micro-architecture in o_e is a false positive.

5.4 Setting

We apply the protocol to nine programs to identify three design motifs.

5.4.1 Studied design motifs

There exists a large number of design patterns in the literature in addition to those proposed originally (Gamma et al. 1994). We validate our approach on the three design motifs that have the largest numbers of classes playing roles in P-MART: `Abstract Factory`, `Adapter`, and `Composite` (see column "Numbers of classes playing a role" in Table 2). We choose the `Composite` design motif also because of its popularity: `Composite` is often used as a typical example of structural design pattern in the literature. We choose the `Abstract Factory` design motif also because other approaches do not identify it although it is commonly used in many known programs as shown in P-MART.

5.4.2 Used programs

The data we used was extracted from nine open-source and freely available programs: GANTT PROJECT v1.10.2, HOLUBSQL v1.0, JHOTDRAW v5.1, JSETTLERS v1.0.5, JTANS v1.0, JUZZLE v0.5, LEXI v0.0.1α, RISK v1.0.7.5. Table 6 presents these programs and related data. The sizes of the programs vary from 99 to 616 classes which are characteristics of small-to-medium scale programs.

Although three of these programs have been used to build numerical signatures (JHOTDRAW v5.1, JUNIT v3.7, and LEXI v0.0.1α, as seen in Table 2), we avoid a bias by excluding these three programs from any experiment involving numerical signatures. For example, JHOTDRAW, from which we derive signatures for the `Adapter` and `Composite` design motifs, is used only to evaluate the identification of `Abstract Factory`.

Table 6 Information on the programs used for the experiments

Programs	Sizes	Information
GANTT PROJECT v1.10.2	616	Tasks management software ganttproject.sourceforge.net
HOLUBSQL v1.0	151	Embedded SQL interpreter www.holub.com/software/holubSQL/
JHOTDRAW v5.1	266	Graph drawing framezork www.jhotdraw.org
JSETTLERS v1.0.5	255	Settlers of Catan board game jsettlers.sourceforge.net
JTANS v1.0	206	Tangram puzzle game jtans.sourceforge.net
JUNIT v3.7	289	Unit testing framework www.junit.org
JUZZLE v0.5	99	Simple puzzle game juzzle.sourceforge.net
LEXI v0.0.1α	216	Text editor lexi.sourceforge.net
RISK v1.0.7.5	256	Strategy game javarisk.sourceforge.net

5.4.3 Tools

The experiment is supported by the PTIDEJ tool suite for metrics computation and micro-architectures identification. The constraint-based identification was implemented by JPTIDEJSOLVER, a dedicated constraint solver that uses PALM, the Java reference implementation of explanation-based constraint programming, built with the JCHOCO library (Labuthe 2000). We integrate numerical signatures as pre-constraints in a variant of JPTIDEJSOLVER.

5.4.4 Time computation

We need to retrieve the computation time required to identify design motifs in a program with and without numerical signatures to verify our first question. We retrieve identification times using a profiler, ECLIPSE PROFILER (Scheglov and Shackelford 2004). We perform all computations seven times on an AMD ATHLON 64 bits processor at 2 GHz. Computations take an average of 50 min to identify all the micro-architectures similar to one given design motif in one given system. We consider in this experiment that a computation time greater than 1 hour is unrealistic in an industrial and/or interactive setting: computation is aborted and computation time is said to be infinite.

5.4.5 Manipulation of the micro-architectures

Our dedicated constraint solver, JPTIDEJSOLVER, and its variant with numerical signatures, generates several files containing textual representations of the identified micro-architectures and various statistics. We store and manually analyse the micro-architectures found with and without numerical signatures to check our second question. We analyse the textual representations by hand and using our tool, PTIDEJ, which loads the textual representations and displays the micro-architectures on top of the UML-like representations of the programs.

5.5 Results

5.5.1 RQ1

Using the raw data in Table 7, Table 8 presents the percentages of time differences between the identification with and without numerical signatures; N/A indicates that both complete and incomplete occurrences of the motif considered were used in the signature inference process of the searched motif. $100 - \varepsilon$ indicates that the identification process without numerical signatures was aborted because of computation time (>1 h). In such cases, the time reduction between the completed part of the process and the identification process with numerical signatures is close to 100%. This was the case for the identification of the `Adapter` and `Composite` design motifs in GANTT PROJECT (616 classes).

Although the computation of the numerical signatures introduces overhead, the overall identification times significantly decrease for almost all design motif–program combinations. For the `Composite` design motif, the identification time is reduced by more that 90% (more than 99% for most of the programs). For the `Abstract Factory` and `Adapter` design motif, we find only one case (JUZZLE) where no improvement occurred. We explain this lack of improvement by the fact that this program is small and contains

Table 7 Times for identification of micro-architectures similar to design motifs, with and without numerical signatures

Programs	Times for identification (in seconds)		
	Without numerical signatures	With numerical signatures	Differences
`Adapter`			
GANTT PROJECT v1.10.2	N/A	157127	N/A
HOLUBSQL v1.0	1106	3	1103
JHOTDRAW v5.1	Excluded because used to compute numerical signatures		
JSETTLERS v1.0.5	417139	799	416340
JTANS v1.0	1445	53	1392
JUNIT v3.7	68172	6372	61800
JUZZLE v0.5	21	21	0
LEXI v0.0.1α	4936	156	4780
RISK v1.0.7.5	2435	134	2301
`Abstract Factory`			
GANTT PROJECT v1.10.2	6480	575	5905
HOLUBSQL v1.0	13	10	3
JHOTDRAW v5.1	1202	275	927
JSETTLERS v1.0.5	2449	278	2171
JTANS v1.0	25	5	20
JUNIT v3.7	1301	512	789
JUZZLE v0.5	1	1	0
LEXI v0.0.1α	21	7	14
RISK v1.0.7.5	83	29	54
`Composite`			
GANTT PROJECT v1.10.2	N/A	266	N/A
HOLUBSQL v1.0	21073	12	21061
JHOTDRAW v5.1	Excluded because used to compute numerical signatures		
JSETTLERS v1.0.5	N/A	1615	N/A
JTANS v1.0	2824	19	2805
JUNIT v3.7	Excluded because used to compute numerical signatures		
JUZZLE v0.5	44	3	41
LEXI v0.0.1α	34926	21	34905
RISK v1.0.7.5	3373	23	3350

only a few occurrences of design motifs. In general, the gain in time is more than 90% for the `Adapter` design motif and between 23 and 90% for the `Abstract Factory` design motif. These results answer RQ1: *the use of numerical signatures reduces significantly the time of identifying occurrences of design motifs*.

An interesting finding is that, without search-space reduction, identification time is correlated positively with the number of elements in a program (classes and relations). For two of the three design motifs, identification time follows an exponential function. With the numerical signature, identification time is independent of the size of a program.

Table 8 Time reductions for each program and pattern

Programs	Time Reductions in %		
	`Adapter`	`Abstract Factory`	`Composite`
GANTT PROJECT v1.10.2	$100 - \varepsilon$	91.13	$100 - \varepsilon$
HOLUBSQL v1.0	99.73	23.08	99.94
JHOTDRAW v5.1	N/A	77.12	N/A
JSETTLERS v1.0.5	99.81	88.65	$100 - \varepsilon$
JTANS v1.0	96.33	80.00	99.33
JUNIT v3.7	90.65	60.65	N/A
JUZZLE v0.5	0.00	0.00	93.18
LEXI v0.0.1α	96.84	66.67	99.94
RISK v1.0.7.5	94.50	65.06	99.32

Table 9 Summary of the results of design-motif occurrences identification

Programs	`Adapter`			
	NIO	NIOS	NFP	NTP
GANTTPROJECT v1.10.2	N/A	588	N/A	N/A
HOLUBSQL v1.0	70	17	53	0
JHOTDRAW v5.1	N/A	N/A	N/A	N/A
JSETTLERS v1.0.5	11,479	0	11,479	0
JTANS v1.0	128	2	126	0
JUNIT v3.7	2,490	234	2,256	0
JUZZLE v0.5	3	3	0	0
LEXI v0.0.1α	57	3	54	0
RISK v1.0.7.5	11	11	0	0

NIO, NIOS, NFP, and NTP are the numbers of identified occurrences without and with numerical signatures, of false positives, and of true positives, respectively

5.5.2 RQ2

Tables 9, 10, and 11 show, for each design motif and for each program, the numbers of identified occurrences (both complete and incomplete), without (NIO) and with (NIOS) numerical signatures, respectively. Occurrences eliminated by the use of numerical signatures can be either false positives (NFP) or true positives (NTP). As summarised in the tables, all the eliminated occurrences are false positives: no true positive occurrence is missed as a result of the use of numerical signatures. Therefore, these results answer RQ2: *the use of numerical signatures reduces the number of false positives while preserving all valid occurrences, it improves the precision while keeping the perfect recall.*

We explain this finding by comparing existing structural approaches and our numerical signature-based approach in terms of syntax and semantics. Numerical signatures describe by their very construction both the syntax and the semantics of design motifs. Indeed, numerical signatures, beyond their computation on structural elements of programs

Table 10 Summary of the results of design-motif occurrences identification

Programs	`Abstract Factory`			
	NIO	NIOS	NFP	NTP
Gantt Project v1.10.2	706	143	563	0
HolubSQL v1.0	50	10	40	0
JHotDraw v5.1	2,934	817	2,117	0
JSettlers v1.0.5	631	20	611	0
JTans v1.0	51	0	51	0
JUnit v3.7	327	72	255	0
Juzzle v0.5	9	0	9	0
Lexi v0.0.1α	18	1	17	0
Risk v1.0.7.5	43	0	43	0

NIO, NIOS, NFP, and NTP are the numbers of identified occurrences without and with numerical signatures, of false positives, and of true positives, respectively

Table 11 Summary of the results of design-motif occurrences identification

Programs	`Composite`			
	NIO	NIOS	NFP	NTP
Gantt Project v1.10.2	N/A	0	N/A	N/A
HolubSQL v1.0	455	0	455	0
JHotDraw v5.1	N/A	N/A	N/A	N/A
JSettlers v1.0.5	N/A	584	N/A	N/A
JTans v1.0	525	0	525	0
JUnit v3.7	N/A	N/A	N/A	N/A
Juzzle v0.5	20	0	20	0
Lexi v0.0.1α	88	0	88	0
Risk v1.0.7.5	72	0	72	0

NIO, NIOS, NFP, and NTP are the numbers of identified occurrences without and with numerical signatures, of false positives, and of true positives, respectively

(classes and relations), integrate semantics through the manual analyses performed on the data used to built the signatures. Thus, in comparison to purely structural approaches, which only consider the syntax (structure) of programs, our approach uses semantics by considering the semantics (or lack thereof) of a class playing a role in a design motif.

5.6 Threats to the validity

Our experiment is subject to several threats to its validity.

Construct validity threats concern the relation between theory and observation. It is possible that the numerical signatures do not describe the role that a class plays in a design motif but some other concerns of the class. Following our first and second assumptions, we limited this threat by eliminating from our study roles that do not have a sufficient number of examples and any numerical signatures with a low recall ratio.

Table 12 Comparisons of the properties of current approaches to design-motif identification with our approach

	Unification	Constraints resolution	Fuzzy reasoning	Quantitative evaluation	Our approach
Incomplete forms	No	No	Yes	Yes	Yes
Explanations	No	No	No	No	Yes
Interactions	No	No	No	No	Yes
Performance	Limited	Limited	Limited	High	High
Precision	High	High	Low	Low	High
Recall	Low	Low	Low	Low	High

Internal validity and *Conclusion validity* threats concern the relation between the observed improvements and the use of numerical signatures. We believe that the main threat concern the experimenter bias. Although the differences between the constraint-based approach and the enhanced constraint-based approach is most likely due to the use of the numerical signatures, we are both the developers and the experimenters of the approaches and, therefore, we could have biased inadvertently the results. All data and programs are available on-line[6] for other researchers to confirm our results.

External validity threats concern the generalisation of the results. Although we considered nine diverse small-to-medium size Java programs and three design motifs with very different designs, we cannot generalise the improvements in performance and precision to *all* programs and motifs. Future study includes studying more design motifs to assess the generalisability of the improvements. We could also apply our approach to other identification approaches, although previous study such as Tsantalis' (2006) have equivalent or lower precisions (Guéhéneuc and Antoniol 2008).

Reliability validity threats concern the replicability of our study. Our implementations and data are available on-line (refer footnote 6). The data in P-MART is freely available as are the programs with which the experiment was performed.[7]

6 Conclusion and future study

We presented an exploratory study of an approach to identify complete and incomplete occurrences of design motifs in object-oriented programs. We developed an explanation-based approach enhanced through the use of experimentally built numerical signatures, which characterise the classes playing roles in design motifs. We showed that our approach has better performance and precision than a purely structural approach, while preserving its favoured perfect recall. Indeed, in comparison with previous approaches, our approach possesses the following properties, as summarised in Table 12:

- Identification of complete and incomplete occurrences of design motifs.
- Explanations of the identified micro-architectures and interactions with maintainers.
- High performance and precision; perfect recall.

[6] http://www.ptidej.org/downloads/experiments/SQJ09/.

[7] http://www.ptidej.net/downloads/pmart/.

The exploratory study of our approach, involving nine programs with three common design motifs, allowed us to conclude that numerical signatures reduce identification time significantly as well as reduce the number of false positive occurrences.

Future study includes improving the representation of design motifs to include non-structural design motifs by using dynamic data in addition to structural data. It also includes the use of incomplete occurrences to identify refactoring opportunities. We also plan to generate and study the improvement in performance and precision for other design motifs to perform a complete empirical study of the improvements. In general, replication of this study with other programs and motifs is required to confirm the results discussed here.

Acknowledgements We thank James Bieman, Greg Straw, Huxia Wang, P. Willard, and Roger T. Alexander (2003) for kindly sharing their data. We are grateful to our students, Saliha Bouden, Janice Ka-Yee Ng, Nawfal Chraibi, Duc-Loc Huynh, and Taleb Ikbal, who helped in the creation of the repository. We are indebt with Neil Stewart for his kind helpful comments.

References

Albin-Amiot, H., Cointe, P., Guéhéneuc, Y.-G., & Jussien, N. (2001). Instantiating and detecting design patterns: Putting bits and pieces together. In D. Richardson, M. Feather, & M. Goedicke (Eds.), *Proceedings of the 16th conference on automated software engineering (ASE)*, November 2001 (pp. 166–173). IEEE Computer Society Press.

Antoniol, G., Fiutem, R., & Cristoforetti, L. (1998). Design pattern recovery in object-oriented software. In S. Tilley & G. Visaggio (Eds.), *Proceedings of the 6th international workshop on program comprehension* (pp. 153–160). IEEE Computer Society Press.

Bansiya, J. (1998). Automating design-pattern identification. *Dr. Dobb's Journal*. http://www.ddj.com/184410578.

Bayardo, R. J. Jr., & Miranker, D. P. (1996). A complexity analysis of space-bounded learning algorithms for the constraint satisfaction problem. In D. Weld & B. Clancey (Eds.), *Proceedings of the 13th national conference on artificial intelligence* (pp. 298–304). AAAI Press, The MIT Press.

Beyer, D., Noack, A., & Lewerentz, C. (2005). Efficient relational calculation for software analysis. *Transactions on Software Engineering, 31*(2), 137–149.

Bieman, J., Straw, G., Wang, H., Munger, P. W., & Alexander, R. T. (2003). Design patterns and change proneness: An examination of five evolving systems. In M. Berry & W. Harrison (Eds.), *Proceedings of the 9th international software metrics symposium* (pp. 40–49). IEEE Computer Society Press.

Boehm, B. (1976). Software engineering. *IEEE Transactions on Computers, 25*(12), 1226–1241.

Briand, L., Devanbu, P., & Melo, W. (1997a). An investigation into coupling measures for C++. In Adrion W. R. (Ed.), *Proceedings of the 19th international conference on software engineering* (pp. 412–421). ACM Press.

Briand, L. C., Daly, J. W., & Wüst, J. K. (1997b). A unified framework for cohesion measurement. In S. L. Pfleeger & L. Ott (Eds.), *Proceedings of the 4th international software metrics symposium* (pp. 43–53). IEEE Computer Society Press.

Brown, K. (1996). Design reverse-engineering and automated design pattern detection in Smalltalk. Technical Report TR-96-07, Department of Computer Science, University of Illinois at Urbana-Champaign.

Chidamber, S. R., & Kemerer, C. F. (1993). A metrics suite for object-oriented design. Technical Report E53-315, MIT Sloan School of Management.

Ciupke, O. (1999). Automatic detection of design problems in object-oriented reengineering. In D. Firesmith (Ed.), *Proceeding of 30th conference on technology of object-oriented languages and systems* (pp.18–32). IEEE Computer Society Press.

Corbi, T. A. (1989). Program understanding: Challenge for the 1990's. *IBM Systems Journal, 28*(2), 294–306.

Eppstein, D. (1995). Subgraph isomorphism in planar graphs and related problems. In K. Clarkson (Ed.), *Proceedings of the 6th annual symposium on discrete algorithms* (pp. 632–640). ACM Press.

Fabry, J., & Mens, T. (2004). Language-independent detection of object-oriented design patterns. *Computer Languages, Systems, and Structures, 30*(1–2), 21–33.

Gamma, E., & Eggenschwiler, T. (1998). JHotDraw. members.pingnet.ch/gamma/JHD-5.1.zip.

Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1994). *Design patterns—elements of reusable object-oriented software* (1st ed.). Addison-Wesley.

Guéhéneuc, Y.-G., & Albin-Amiot, H. (2001). Using design patterns and constraints to automate the detection and correction of inter-class design defects. In Q. Li, R. Riehle, G. Pour, & B. Meyer (Eds.), *Proceedings of the 39th conference on the technology of object-oriented languages and systems (TOOLS USA)* (pp. 296–305). IEEE Computer Society Press.

Guéhéneuc, Y.-G., & Albin-Amiot, H. (2004). Recovering binary class relationships: Putting icing on the UML cake. In D. C. Schmidt (Ed.), *Proceedings of the 19th conference on object-oriented programming, systems, languages, and applications (OOPSLA)* (pp. 301–314). ACM Press.

Guéhéneuc, Y.-G., & Antoniol, G. (2008). DeMIMA: A multi-layered framework for design pattern identification. *Transactions on Software Engineering (TSE), 34*(5), 667–684.

Guéhéneuc, Y.-G., & Jussien, N. (2001). Using explanations for design-patterns identification. In C. Bessière (Ed.), *Proceedings of the 1st IJCAI workshop on modeling and solving problems with constraints* (pp 57–64). AAAI Press.

Guéhéneuc, Y.-G., Sahraoui, H., & Zaidi, F. (2004). Fingerprinting design patterns. In E. Stroulia & A. de Lucia, (Eds.), *Proceedings of the 11th working conference on reverse engineering (WCRE)* (pp.172–181). IEEE Computer Society Press.

Heuzeroth, D., Holl, T., & Löwe, W. (2002). Combining static and dynamic analyses to detect interaction patterns. In H. Ehrig, B. J. Krämer, & A. Ertas (Eds.), *Proceedings the 6th world conference on integrated design and process technology*. Society for Design and Process Science.

Hitz, M., & Montazeri, B. (1995). Measuring coupling and cohesion in object-oriented systems. In *Proceedings of the 3rd international symposium on applied corporate computing* (pp. 25–27). Texas A&M University.

Jahnke, J. H., & Zündorf, A. (1997). Rewriting poor design patterns by good design patterns. In S. Demeyer & H. C. Gall (Eds.), *Proceedings the 1st ESEC/FSE workshop on object-oriented reengineering*. Distributed Systems Group, Technical University of Vienna. TUV-1841-97-10.

Junker, U. (2001). QUICKXPLAIN: Conflict detection for arbitrary constraint propagation algorithms. Technical report, Ilog SA.

Jussien, N., & Barichard, V. (2000). The PaLM system: Explanation-based constraint programming. In N. Beldiceanu, W. Harvey, M. Henz, F. Laburthe, E. Monfroy, T. Müller, L. Perron, & C. Schulte (Eds.), *Proceedings of TRICS: Techniques for implementing constraint programming systems* (pp. 118–133). School of Computing, National University of Singapore, Singapore. TRA9/00.

Keller, R. K., Schauer, R., Robitaille, S., & Pagé, P. (1999). Pattern-based reverse-engineering of design components. In D. Garlan & J. Kramer (Eds.), *Proceedings of the 21st international conference on software engineering* (pp. 226–235). ACM Press.

Kohavi, R. (1995). A study of cross-validation and bootstrap for accuracy estimation and model selection. In *Proceedings of the 14th international joint conference on artificial intelligence* (pp. 1137–1145). Morgan Kaufmann.

Koskinen, J. (2004). Software maintenance costs. http://users.jyu.fi/~koskinen/smcosts.htm.

Krämer, C., & Prechelt, L. (1996). Design recovery by automated search for structural design patterns in object-oriented software. In L. M. Wills & I. Baxter (Eds.), *Proceedings of the 3rd working conference on reverse engineering* (pp. 208–215). IEEE Computer Society Press.

Labuthe, F. (2000). Choco: Implementing a CP kernel. In N. Beldiceanu (Ed.), *Proceedings of the 1st workshop on techniques for implementing constraint programming systems*, CP'00 at Singapore.

Lorenz, M., & Kidd, J. (1994). *Object-oriented software metrics: A practical approach* (1st ed.). Prentice-Hall.

Ng, K.-Y., & Guéhéneuc, Y.-G. (2007). Identification of behavioral and creational design patterns through dynamic analysis. In A. Zaidman, A. Hamou-Lhadj, & O. Greevy (Eds.), *Proceedings of the 3rd international workshop on program comprehension through dynamic analysis (PCODA)* (pp. 34–42). Delft University of Technology. TUD-SERG-2007-022.

Quilici, A., Yang, Q., & Woods, S. (1997). Applying plan recognition algorithms to program understanding. *Journal of Automated Software Engineering, 5*(3), 347–372.

Scheglov, K., & Shackelford, J.-M. P. (2004). Eclipse profiler. http://eclipsecolorer.sourceforge.net/index_profiler.html.

Schiex, T., & Verfaillie, G. (1994). Nogood recording for static and dynamic constraint satisfaction problems. *International Journal of Artificial Intelligence Tools, 3*(2), 187–207.

Spinellis, D. (2003). *Code reading: The open source perspective* (1st ed.). Addison Wesley.

Stone, M. (1974). Cross-validatory choice and assessment of statistical predictions. *Journal of the Royal Statistical Society Series B: Statistical Methodology, 36*, 111–147.

Straw, G. B. (2004). Detecting design pattern use in software designs. Master's thesis, Colorado State University.
Tegarden, D. P., Sheetz, S. D., & Monarchi, D. E. (1995). A software complexity model of object-oriented systems. *Decision Support Systems, 13*(3–4), 241–262.
Tsantalis, N., Chatzigeorgiou, A., Stephanides, G., & Halkidis, S. (2006). Design pattern detection using similarity scoring. *Transactions on Software Engineering, 32*(11), 896–909.
Wilde, N. (1994). Faster reuse and maintenance using software reconnaissance. Technical Report CSE-301, Software Engineering Research Center.
Witten, I. H., & Frank, E. (1999). *Data mining: Practical machine learning tools and techniques with Java implementations* (1st ed.). Morgan Kaufmann.
Wuyts, R. (1998). Declarative reasoning about the structure of object-oriented systems. In J. Gil (Ed.), *Proceedings of the 26th conference on the technology of object-oriented languages and systems* (pp. 112–124). IEEE Computer Society Press.

Author Biographies

Yann-Gaël Guéhéneuc is an associate professor at the Department of computing and software engineering, Ecole Polytechnique 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. In 2009, he was awarded the NSERC Research Chair Tier II on Software Patterns and Patterns of Software. He holds a Ph.D. in software engineering from University of Nantes, France (under Professor Pierre Cointe's supervision) since 2003 and an Engineering Diploma from École des Mines of Nantes since 1998. His Ph.D. thesis was funded by Object Technology International, Inc. (now IBM OTI Labs.), where he worked in 1999 and 2000. 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 was the first to use explanation-based constraint programming in the context of software engineering to identify occurrences of 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 conferences and journals.

Jean-Yves Guyomarc'h is a software developer for the SAP Business Suite UI Framework department at SAP Labs, Canada. He received a master's degree in software engineering from the Department of Informatics and Operations Research, University of Montreal, Canada (2006). He also holds a computer science engineer degree from l'École polytechnique universitaire de Nice Sophia-Antipolis (EPUNSA), France (2004). His interests are focused on software quality and metrics applied to object- and aspect-oriented conception.

Houari Sahraoui is a professor at the Department of Computer Science and Operations Research (GEODES, software engineering group), University of Montreal. Before joining the university, he held the position of lead researcher of the software engineering group at CRIM (Research center on computer science, Montreal). He holds an Engineering Diploma from the National Institute of computer science (1990), Algiers, and a Ph.D. in Computer Science, Pierre & Marie Curie University LIP6, Paris, 1995. His research interests include the application of artificial intelligence techniques to software engineering, object-oriented metrics and quality, software visualization, and re-engineering. He has published around 100 papers in conferences, workshops, books, and journals, edited three books, and has given regularly invited talks. He has served as program committee member in several major conferences (IEEE ASE, ECOOP, METRICS, etc.), as member of the editorial boards of two journals, and as organization member of many conferences and workshops (ICSM, ASE, QAOOSE, etc). He was the general chair of IEEE Automated Software Engineering Conference in 2003.